The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, 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 distributed file systems, our design has been driven by observations of our application workloads and technological environment, 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 radically 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 service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds 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.

Categories and Subject Descriptors

D [**4**]: 3—*Distributed file systems*

General Terms

Design, reliability, performance, measurement

Keywords

Fault tolerance, scalability, data storage, clustered storage

*The authors can be reached at the following addresses: {*sanjay,hgobioff,shuntak*}*@google.com*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.
Copyright 2003 ACM 1-58113-757-5/03/0010 ...$5.00.

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, files are huge by traditional standards. Multi-GB files are common. Each file typically contains many application objects such as web documents. When we are regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of approximately KB-sized files even when the file system could support it. As a result, design assumptions and parameters such as I/O operation and block sizes have to be revisited.

Third, most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once written, the files are only read, and often only sequentially. A variety of data share these characteristics. Some may constitute large repositories that data analysis programs scan through. Some may be data streams continuously generated by running applications. Some may be archival data. Some may be intermediate results produced on one machine and processed on another, whether simultaneously or later in time. Given this access pattern on huge files, appending becomes the focus of performance optimization and atomicity guarantees, while caching data blocks in the client loses its appeal.

Fourth, co-designing the applications and the file system API benefits the overall system by increasing our flexibility.

For example, we have relaxed GFS's consistency model to vastly simplify the file system without imposing an onerous burden on the applications. We have also introduced an atomic append operation so that multiple clients can append concurrently to a file without extra synchronization between them. These will be discussed in more details later in the paper.

Multiple GFS clusters are currently deployed for different purposes. The largest ones have over 1000 storage nodes, over 300 TB of disk storage, and are heavily accessed by hundreds of clients on distinct machines on a continuous basis.

2. DESIGN OVERVIEW

2.1 Assumptions

In designing a file system for our needs, we have been guided by assumptions that offer both challenges and opportunities. We alluded to some key observations earlier and now lay out our assumptions in more details.

- The system is built from many inexpensive commodity components that often fail. It must constantly monitor itself and detect, tolerate, and recover promptly from component failures on a routine basis.
- The system stores a modest number of large files. We expect a few million files, each typically 100 MB or larger in size. Multi-GB files are the common case and should be managed efficiently. Small files must be supported, but we need not optimize for them.
- The workloads primarily consist of two kinds of reads: large streaming reads and small random reads. In large streaming reads, individual operations typically read hundreds of KBs, more commonly 1 MB or more. Successive operations from the same client often read through a contiguous region of a file. A small random read typically reads a few KBs at some arbitrary offset. Performance-conscious applications often batch and sort their small reads to advance steadily through the file rather than go back and forth.
- The workloads also have many large, sequential writes that append data to files. Typical operation sizes are similar to those for reads. Once written, files are seldom modified again. Small writes at arbitrary positions in a file are supported but do not have to be efficient.
- The system must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file. Our files are often used as producer-consumer queues or for many-way merging. Hundreds of producers, running one per machine, will concurrently append to a file. Atomicity with minimal synchronization overhead is essential. The file may be read later, or a consumer may be reading through the file simultaneously.
- High sustained bandwidth is more important than low latency. Most of our target applications place a premium on processing data in bulk at a high rate, while few have stringent response time requirements for an individual read or write.

2.2 Interface

GFS provides a familiar file system interface, though it does not implement a standard API such as POSIX. Files are organized hierarchically in directories and identified by pathnames. We support the usual operations to *create*, *delete*, *open*, *close*, *read*, and *write* files.

Moreover, GFS has *snapshot* and *record append* operations. Snapshot creates a copy of a file or a directory tree at low cost. Record append allows multiple clients to append data to the same file concurrently while guaranteeing the atomicity of each individual client's append. It is useful for implementing multi-way merge results and producer-consumer queues that many clients can simultaneously append to without additional locking. We have found these types of files to be invaluable in building large distributed applications. Snapshot and record append are discussed further in Sections 3.4 and 3.3 respectively.

2.3 Architecture

A GFS cluster consists of a single *master* and multiple *chunkservers* and is accessed by multiple *clients*, as shown in Figure 1. Each of these is typically a commodity Linux machine running a user-level server process. It is easy to run both a chunkserver and a client on the same machine, as long as machine resources permit and the lower reliability caused by running possibly flaky application code is acceptable.

Files are divided into fixed-size *chunks*. Each chunk is identified by an immutable and globally unique 64 bit *chunk handle* assigned by the master at the time of chunk creation. Chunkservers store chunks on local disks as Linux files and read or write chunk data specified by a chunk handle and byte range. For reliability, each chunk is replicated on multiple chunkservers. By default, we store three replicas, though users can designate different replication levels for different regions of the file namespace.

The master maintains all file system metadata. This includes the namespace, access control information, the mapping from files to chunks, and the current locations of chunks. It also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunkservers. The master periodically communicates with each chunkserver in *HeartBeat* messages to give it instructions and collect its state.

GFS client code linked into each application implements the file system API and communicates with the master and chunkservers to read or write data on behalf of the application. Clients interact with the master for metadata operations, but all data-bearing communication goes directly to the chunkservers. We do not provide the POSIX API and therefore need not hook into the Linux vnode layer.

Neither the client nor the chunkserver caches file data. Client caches offer little benefit because most applications stream through huge files or have working sets too large to be cached. Not having them simplifies the client and the overall system by eliminating cache coherence issues. (Clients do cache metadata, however.) Chunkservers need not cache file data because chunks are stored as local files and so Linux's buffer cache already keeps frequently accessed data in memory.

2.4 Single Master

Having a single master vastly simplifies our design and enables the master to make sophisticated chunk placement

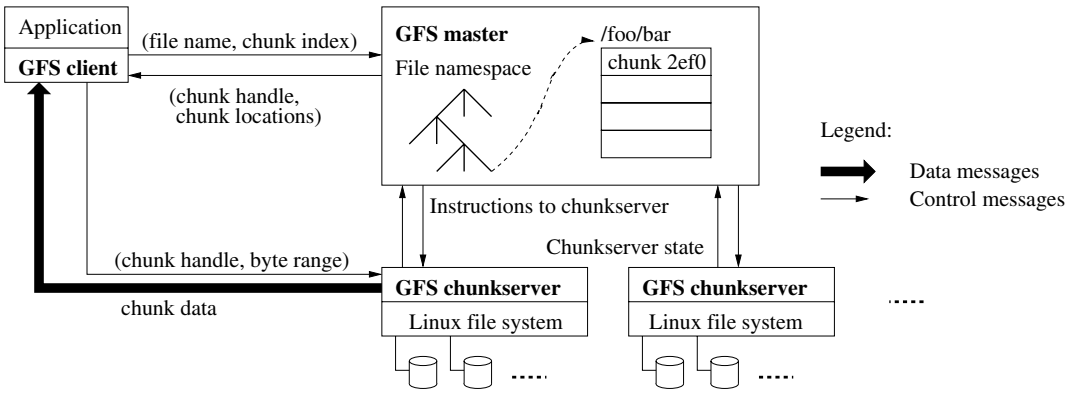


Figure 1: GFS Architecture

and replication decisions using global knowledge. However, we must minimize its involvement in reads and writes so that it does not become a bottleneck. Clients never read and write file data through the master. Instead, a client asks the master which chunkservers it should contact. It caches this information for a limited time and interacts with the chunkservers directly for many subsequent operations.

Let us explain the interactions for a simple read with reference to Figure 1. First, using the fixed chunk size, the client translates the file name and byte offset specified by the application into a chunk index within the file. Then, it sends the master a request containing the file name and chunk index. The master replies with the corresponding chunk handle and locations of the replicas. The client caches this information using the file name and chunk index as the key.

The client then sends a request to one of the replicas, most likely the closest one. The request specifies the chunk handle and a byte range within that chunk. Further reads of the same chunk require no more client-master interaction until the cached information expires or the file is reopened. In fact, the client typically asks for multiple chunks in the same request and the master can also include the information for chunks immediately following those requested. This extra information sidesteps several future client-master interactions at practically no extra cost.

2.5 Chunk Size

Chunk size is one of the key design parameters. We have chosen 64 MB, which is much larger than typical file system block sizes. Each chunk replica is stored as a plain Linux file on a chunkserver and is extended only as needed. Lazy space allocation avoids wasting space due to internal fragmentation, perhaps the greatest objection against such a large chunk size.

A large chunk size offers several important advantages. First, it reduces clients' need to interact with the master because reads and writes on the same chunk require only one initial request to the master for chunk location information. The reduction is especially significant for our workloads because applications mostly read and write large files sequentially. Even for small random reads, the client can comfortably cache all the chunk location information for a multi-TB working set. Second, since on a large chunk, a client is more likely to perform many operations on a given chunk, it can reduce network overhead by keeping a persistent TCP connection to the chunkserver over an extended period of time. Third, it reduces the size of the metadata stored on the master. This allows us to keep the metadata in memory, which in turn brings other advantages that we will discuss in Section 2.6.1.

On the other hand, a large chunk size, even with lazy space allocation, has its disadvantages. A small file consists of a small number of chunks, perhaps just one. The chunkservers storing those chunks may become hot spots if many clients are accessing the same file. In practice, hot spots have not been a major issue because our applications mostly read large multi-chunk files sequentially.

However, hot spots did develop when GFS was first used by a batch-queue system: an executable was written to GFS as a single-chunk file and then started on hundreds of machines at the same time. The few chunkservers storing this executable were overloaded by hundreds of simultaneous requests. We fixed this problem by storing such executables with a higher replication factor and by making the batch-queue system stagger application start times. A potential long-term solution is to allow clients to read data from other clients in such situations.

2.6 Metadata

The master stores three major types of metadata: the file and chunk namespaces, the mapping from files to chunks, and the locations of each chunk's replicas. All metadata is kept in the master's memory. The first two types (namespaces and file-to-chunk mapping) are also kept persistent by logging mutations to an *operation log* stored on the master's local disk and replicated on remote machines. Using a log allows us to update the master state simply, reliably, and without risking inconsistencies in the event of a master crash. The master does not store chunk location information persistently. Instead, it asks each chunkserver about its chunks at master startup and whenever a chunkserver joins the cluster.

2.6.1 In-Memory Data Structures

Since metadata is stored in memory, master operations are fast. Furthermore, it is easy and efficient for the master to periodically scan through its entire state in the background. This periodic scanning is used to implement chunk garbage collection, re-replication in the presence of chunkserver failures, and chunk migration to balance load and disk space

usage across chunkservers. Sections 4.3 and 4.4 will discuss these activities further.

One potential concern for this memory-only approach is that the number of chunks and hence the capacity of the whole system is limited by how much memory the master has. This is not a serious limitation in practice. The master maintains less than 64 bytes of metadata for each 64 MB chunk. Most chunks are full because most files contain many chunks, only the last of which may be partially filled. Similarly, the file namespace data typically requires less then 64 bytes per file because it stores file names compactly using prefix compression.

If necessary to support even larger file systems, the cost of adding extra memory to the master is a small price to pay for the simplicity, reliability, performance, and flexibility we gain by storing the metadata in memory.

2.6.2 Chunk Locations

The master does not keep a persistent record of which chunkservers have a replica of a given chunk. It simply polls chunkservers for that information at startup. The master can keep itself up-to-date thereafter because it controls all chunk placement and monitors chunkserver status with regular *HeartBeat* messages.

We initially attempted to keep chunk location information persistently at the master, but we decided that it was much simpler to request the data from chunkservers at startup, and periodically thereafter. This eliminated the problem of keeping the master and chunkservers in sync as chunkservers join and leave the cluster, change names, fail, restart, and so on. In a cluster with hundreds of servers, these events happen all too often.

Another way to understand this design decision is to realize that a chunkserver has the final word over what chunks it does or does not have on its own disks. There is no point in trying to maintain a consistent view of this information on the master because errors on a chunkserver may cause chunks to vanish spontaneously (e.g., a disk may go bad and be disabled) or an operator may rename a chunkserver.

2.6.3 Operation Log

The operation log contains a historical record of critical metadata changes. It is central to GFS. Not only is it the only persistent record of metadata, but it also serves as a logical time line that defines the order of concurrent operations. Files and chunks, as well as their versions (see Section 4.5), are all uniquely and eternally identified by the logical times at which they were created.

Since the operation log is critical, we must store it reliably and not make changes visible to clients until metadata changes are made persistent. Otherwise, we effectively lose the whole file system or recent client operations even if the chunks themselves survive. Therefore, we replicate it on multiple remote machines and respond to a client operation only after flushing the corresponding log record to disk both locally and remotely. The master batches several log records together before flushing thereby reducing the impact of flushing and replication on overall system throughput.

The master recovers its file system state by replaying the operation log. To minimize startup time, we must keep the log small. The master checkpoints its state whenever the log grows beyond a certain size so that it can recover by loading the latest checkpoint from local disk and replaying only the limited number of log records after that. The checkpoint is in a compact B-tree like form that can be directly mapped into memory and used for namespace lookup without extra parsing. This further speeds up recovery and improves availability.

Because building a checkpoint can take a while, the master's internal state is structured in such a way that a new checkpoint can be created without delaying incoming mutations. The master switches to a new log file and creates the new checkpoint in a separate thread. The new checkpoint includes all mutations before the switch. It can be created in a minute or so for a cluster with a few million files. When completed, it is written to disk both locally and remotely.

Recovery needs only the latest complete checkpoint and subsequent log files. Older checkpoints and log files can be freely deleted, though we keep a few around to guard against catastrophes. A failure during checkpointing does not affect correctness because the recovery code detects and skips incomplete checkpoints.

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Table 1: File Region State After Mutation

2.7 Consistency Model

GFS has a relaxed consistency model that supports our highly distributed applications well but remains relatively simple and efficient to implement. We now discuss GFS's guarantees and what they mean to applications. We also highlight how GFS maintains these guarantees but leave the details to other parts of the paper.

2.7.1 Guarantees by GFS

File namespace mutations (e.g., file creation) are atomic. They are handled exclusively by the master: namespace locking guarantees atomicity and correctness (Section 4.1); the master's operation log defines a global total order of these operations (Section 2.6.3).

The state of a file region after a data mutation depends on the type of mutation, whether it succeeds or fails, and whether there are concurrent mutations. Table 1 summarizes the result. A file region is *consistent* if all clients will always see the same data, regardless of which replicas they read from. A region is *defined* after a file data mutation if it is consistent and clients will see what the mutation writes in its entirety. When a mutation succeeds without interference from concurrent writers, the affected region is defined (and by implication consistent): all clients will always see what the mutation has written. Concurrent successful mutations leave the region undefined but consistent: all clients see the same data, but it may not reflect what any one mutation has written. Typically, it consists of mingled fragments from multiple mutations. A failed mutation makes the region inconsistent (hence also undefined): different clients may see different data at different times. We describe below how our applications can distinguish defined regions from undefined

regions. The applications do not need to further distinguish between different kinds of undefined regions.

Data mutations may be *writes* or *record appends*. A write causes data to be written at an application-specified file offset. A record append causes data (the "record") to be appended *atomically at least once* even in the presence of concurrent mutations, but at an offset of GFS's choosing (Section 3.3). (In contrast, a "regular" append is merely a write at an offset that the client believes to be the current end of file.) The offset is returned to the client and marks the beginning of a defined region that contains the record. In addition, GFS may insert padding or record duplicates in between. They occupy regions considered to be inconsistent and are typically dwarfed by the amount of user data.

After a sequence of successful mutations, the mutated file region is guaranteed to be defined and contain the data written by the last mutation. GFS achieves this by (a) applying mutations to a chunk in the same order on all its replicas (Section 3.1), and (b) using chunk version numbers to detect any replica that has become stale because it has missed mutations while its chunkserver was down (Section 4.5). Stale replicas will never be involved in a mutation or given to clients asking the master for chunk locations. They are garbage collected at the earliest opportunity.

Since clients cache chunk locations, they may read from a stale replica before that information is refreshed. This window is limited by the cache entry's timeout and the next open of the file, which purges from the cache all chunk information for that file. Moreover, as most of our files are append-only, a stale replica usually returns a premature end of chunk rather than outdated data. When a reader retries and contacts the master, it will immediately get current chunk locations.

Long after a successful mutation, component failures can of course still corrupt or destroy data. GFS identifies failed chunkservers by regular handshakes between master and all chunkservers and detects data corruption by checksumming (Section 5.2). Once a problem surfaces, the data is restored from valid replicas as soon as possible (Section 4.3). A chunk is lost irreversibly only if all its replicas are lost before GFS can react, typically within minutes. Even in this case, it becomes unavailable, not corrupted: applications receive clear errors rather than corrupt data.

2.7.2 Implications for Applications

GFS applications can accommodate the relaxed consistency model with a few simple techniques already needed for other purposes: relying on appends rather than overwrites, checkpointing, and writing self-validating, self-identifying records.

Practically all our applications mutate files by appending rather than overwriting. In one typical use, a writer generates a file from beginning to end. It atomically renames the file to a permanent name after writing all the data, or periodically checkpoints how much has been successfully written. Checkpoints may also include application-level checksums. Readers verify and process only the file region up to the last checkpoint, which is known to be in the defined state. Regardless of consistency and concurrency issues, this approach has served us well. Appending is far more efficient and more resilient to application failures than random writes. Checkpointing allows writers to restart incrementally and keeps readers from processing successfully written file data that is still incomplete from the application's perspective.

In the other typical use, many writers concurrently append to a file for merged results or as a producer-consumer queue. Record append's append-at-least-once semantics preserves each writer's output. Readers deal with the occasional padding and duplicates as follows. Each record prepared by the writer contains extra information like checksums so that its validity can be verified. A reader can identify and discard extra padding and record fragments using the checksums. If it cannot tolerate the occasional duplicates (e.g., if they would trigger non-idempotent operations), it can filter them out using unique identifiers in the records, which are often needed anyway to name corresponding application entities such as web documents. These functionalities for record I/O (except duplicate removal) are in library code shared by our applications and applicable to other file interface implementations at Google. With that, the same sequence of records, plus rare duplicates, is always delivered to the record reader.

3. SYSTEM INTERACTIONS

We designed the system to minimize the master's involvement in all operations. With that background, we now describe how the client, master, and chunkservers interact to implement data mutations, atomic record append, and snapshot.

3.1 Leases and Mutation Order

A mutation is an operation that changes the contents or metadata of a chunk such as a write or an append operation. Each mutation is performed at all the chunk's replicas. We use leases to maintain a consistent mutation order across replicas. The master grants a chunk lease to one of the replicas, which we call the *primary*. The primary picks a serial order for all mutations to the chunk. All replicas follow this order when applying mutations. Thus, the global mutation order is defined first by the lease grant order chosen by the master, and within a lease by the serial numbers assigned by the primary.

The lease mechanism is designed to minimize management overhead at the master. A lease has an initial timeout of 60 seconds. However, as long as the chunk is being mutated, the primary can request and typically receive extensions from the master indefinitely. These extension requests and grants are piggybacked on the *HeartBeat* messages regularly exchanged between the master and all chunkservers. The master may sometimes try to revoke a lease before it expires (e.g., when the master wants to disable mutations on a file that is being renamed). Even if the master loses communication with a primary, it can safely grant a new lease to another replica after the old lease expires.

In Figure 2, we illustrate this process by following the control flow of a write through these numbered steps.

1. The client asks the master which chunkserver holds the current lease for the chunk and the locations of the other replicas. If no one has a lease, the master grants one to a replica it chooses (not shown).
2. The master replies with the identity of the primary and the locations of the other (*secondary*) replicas. The client caches this data for future mutations. It needs to contact the master again only when the primary

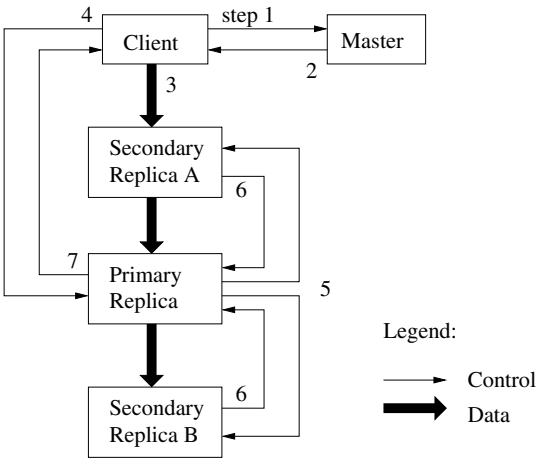


Figure 2: Write Control and Data Flow

becomes unreachable or replies that it no longer holds a lease.

3. The client pushes the data to all the replicas. A client can do so in any order. Each chunkserver will store the data in an internal LRU buffer cache until the data is used or aged out. By decoupling the data flow from the control flow, we can improve performance by scheduling the expensive data flow based on the network topology regardless of which chunkserver is the primary. Section 3.2 discusses this further.
4. Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary. The request identifies the data pushed earlier to all of the replicas. The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients, which provides the necessary serialization. It applies the mutation to its own local state in serial number order.
5. The primary forwards the write request to all secondary replicas. Each secondary replica applies mutations in the same serial number order assigned by the primary.
6. The secondaries all reply to the primary indicating that they have completed the operation.
7. The primary replies to the client. Any errors encountered at any of the replicas are reported to the client. In case of errors, the write may have succeeded at the primary and an arbitrary subset of the secondary replicas. (If it had failed at the primary, it would not have been assigned a serial number and forwarded.) The client request is considered to have failed, and the modified region is left in an inconsistent state. Our client code handles such errors by retrying the failed mutation. It will make a few attempts at steps (3) through (7) before falling back to a retry from the beginning of the write.

If a write by the application is large or straddles a chunk boundary, GFS client code breaks it down into multiple write operations. They all follow the control flow described above but may be interleaved with and overwritten by concurrent operations from other clients. Therefore, the shared file region may end up containing fragments from different clients, although the replicas will be identical because the individual operations are completed successfully in the same order on all replicas. This leaves the file region in consistent but undefined state as noted in Section 2.7.

3.2 Data Flow

We decouple the flow of data from the flow of control to use the network efficiently. While control flows from the client to the primary and then to all secondaries, data is pushed linearly along a carefully picked chain of chunkservers in a pipelined fashion. Our goals are to fully utilize each machine's network bandwidth, avoid network bottlenecks and high-latency links, and minimize the latency to push through all the data.

To fully utilize each machine's network bandwidth, the data is pushed linearly along a chain of chunkservers rather than distributed in some other topology (e.g., tree). Thus, each machine's full outbound bandwidth is used to transfer the data as fast as possible rather than divided among multiple recipients.

To avoid network bottlenecks and high-latency links (e.g., inter-switch links are often both) as much as possible, each machine forwards the data to the "closest" machine in the network topology that has not received it. Suppose the client is pushing data to chunkservers S1 through S4. It sends the data to the closest chunkserver, say S1. S1 forwards it to the closest chunkserver S2 through S4 closest to S1, say S2. Similarly, S2 forwards it to S3 or S4, whichever is closer to S2, and so on. Our network topology is simple enough that "distances" can be accurately estimated from IP addresses.

Finally, we minimize latency by pipelining the data transfer over TCP connections. Once a chunkserver receives some data, it starts forwarding immediately. Pipelining is especially helpful to us because we use a switched network with full-duplex links. Sending the data immediately does not reduce the receive rate. Without network congestion, the ideal elapsed time for transferring B bytes to R replicas is $B/T + RL$ where T is the network throughput and L is latency to transfer bytes between two machines. Our network links are typically 100 Mbps (T), and L is far below 1 ms. Therefore, 1 MB can ideally be distributed in about 80 ms.

3.3 Atomic Record Appends

GFS provides an atomic append operation called *record append*. In a traditional write, the client specifies the offset at which data is to be written. Concurrent writes to the same region are not serializable: the region may end up containing data fragments from multiple clients. In a record append, however, the client specifies only the data. GFS appends it to the file at least once atomically (i.e., as one continuous sequence of bytes) at an offset of GFS's choosing and returns that offset to the client. This is similar to writing to a file opened in `O_APPEND` mode in Unix without the race conditions when multiple writers do so concurrently.

Record append is heavily used by our distributed applications in which many clients on different machines append to the same file concurrently. Clients would need additional complicated and expensive synchronization, for example through a distributed lock manager, if they do so with traditional writes. In our workloads, such files often

serve as multiple-producer/single-consumer queues or contain merged results from many different clients.

Record append is a kind of mutation and follows the control flow in Section 3.1 with only a little extra logic at the primary. The client pushes the data to all replicas of the last chunk of the file Then, it sends its request to the primary. The primary checks to see if appending the record to the current chunk would cause the chunk to exceed the maximum size (64 MB). If so, it pads the chunk to the maximum size, tells secondaries to do the same, and replies to the client indicating that the operation should be retried on the next chunk. (Record append is restricted to be at most one-fourth of the maximum chunk size to keep worst-case fragmentation at an acceptable level.) If the record fits within the maximum size, which is the common case, the primary appends the data to its replica, tells the secondaries to write the data at the exact offset where it has, and finally replies success to the client.

If a record append fails at any replica, the client retries the operation. As a result, replicas of the same chunk may contain different data possibly including duplicates of the same record in whole or in part. GFS does not guarantee that all replicas are bytewise identical. It only guarantees that the data is written at least once as an atomic unit. This property follows readily from the simple observation that for the operation to report success, the data must have been written at the same offset on all replicas of some chunk. Furthermore, after this, all replicas are at least as long as the end of record and therefore any future record will be assigned a higher offset or a different chunk even if a different replica later becomes the primary. In terms of our consistency guarantees, the regions in which successful record append operations have written their data are defined (hence consistent), whereas intervening regions are inconsistent (hence undefined). Our applications can deal with inconsistent regions as we discussed in Section 2.7.2.

3.4 Snapshot

The snapshot operation makes a copy of a file or a directory tree (the "source") almost instantaneously, while minimizing any interruptions of ongoing mutations. Our users use it to quickly create branch copies of huge data sets (and often copies of those copies, recursively), or to checkpoint the current state before experimenting with changes that can later be committed or rolled back easily.

Like AFS [5], we use standard copy-on-write techniques to implement snapshots. When the master receives a snapshot request, it first revokes any outstanding leases on the chunks in the files it is about to snapshot. This ensures that any subsequent writes to these chunks will require an interaction with the master to find the lease holder. This will give the master an opportunity to create a new copy of the chunk first.

After the leases have been revoked or have expired, the master logs the operation to disk. It then applies this log record to its in-memory state by duplicating the metadata for the source file or directory tree. The newly created snapshot files point to the same chunks as the source files.

The first time a client wants to write to a chunk C after the snapshot operation, it sends a request to the master to find the current lease holder. The master notices that the reference count for chunk C is greater than one. It defers replying to the client request and instead picks a new chunk handle C'. It then asks each chunkserver that has a current replica of C to create a new chunk called C'. By creating the new chunk on the same chunkservers as the original, we ensure that the data can be copied locally, not over the network (our disks are about three times as fast as our 100 Mb Ethernet links). From this point, request handling is no different from that for any chunk: the master grants one of the replicas a lease on the new chunk C' and replies to the client, which can write the chunk normally, not knowing that it has just been created from an existing chunk.

4. MASTER OPERATION

The master executes all namespace operations. In addition, it manages chunk replicas throughout the system: it makes placement decisions, creates new chunks and hence replicas, and coordinates various system-wide activities to keep chunks fully replicated, to balance load across all the chunkservers, and to reclaim unused storage. We now discuss each of these topics.

4.1 Namespace Management and Locking

Many master operations can take a long time: for example, a snapshot operation has to revoke chunkserver leases on all chunks covered by the snapshot. We do not want to delay other master operations while they are running. Therefore, we allow multiple operations to be active and use locks over regions of the namespace to ensure proper serialization.

Unlike many traditional file systems, GFS does not have a per-directory data structure that lists all the files in that directory. Nor does it support aliases for the same file or directory (i.e, hard or symbolic links in Unix terms). GFS logically represents its namespace as a lookup table mapping full pathnames to metadata. With prefix compression, this table can be efficiently represented in memory. Each node in the namespace tree (either an absolute file name or an absolute directory name) has an associated read-write lock.

Each master operation acquires a set of locks before it runs. Typically, if it involves `/d1/d2/.../dn/leaf`, it will acquire read-locks on the directory names `/d1`, `/d1/d2`, ..., `/d1/d2/.../dn`, and either a read lock or a write lock on the full pathname `/d1/d2/.../dn/leaf`. Note that `leaf` may be a file or directory depending on the operation.

We now illustrate how this locking mechanism can prevent a file `/home/user/foo` from being created while `/home/user` is being snapshotted to `/save/user`. The snapshot operation acquires read locks on `/home` and `/save`, and write locks on `/home/user` and `/save/user`. The file creation acquires read locks on `/home` and `/home/user`, and a write lock on `/home/user/foo`. The two operations will be serialized properly because they try to obtain conflicting locks on `/home/user`. File creation does not require a write lock on the parent directory because there is no "directory", or *inode*-like, data structure to be protected from modification. The read lock on the name is sufficient to protect the parent directory from deletion.

One nice property of this locking scheme is that it allows concurrent mutations in the same directory. For example, multiple file creations can be executed concurrently in the same directory: each acquires a read lock on the directory name and a write lock on the file name. The read lock on the directory name suffices to prevent the directory from being deleted, renamed, or snapshotted. The write locks on

file names serialize attempts to create a file with the same name twice.

Since the namespace can have many nodes, read-write lock objects are allocated lazily and deleted once they are not in use. Also, locks are acquired in a consistent total order to prevent deadlock: they are first ordered by level in the namespace tree and lexicographically within the same level.

4.2 Replica Placement

A GFS cluster is highly distributed at more levels than one. It typically has hundreds of chunkservers spread across many machine racks. These chunkservers in turn may be accessed from hundreds of clients from the same or different racks. Communication between two machines on different racks may cross one or more network switches. Additionally, bandwidth into or out of a rack may be less than the aggregate bandwidth of all the machines within the rack. Multi-level distribution presents a unique challenge to distribute data for scalability, reliability, and availability.

The chunk replica placement policy serves two purposes: maximize data reliability and availability, and maximize network bandwidth utilization. For both, it is not enough to spread replicas across machines, which only guards against disk or machine failures and fully utilizes each machine's network bandwidth. We must also spread chunk replicas across racks. This ensures that some replicas of a chunk will survive and remain available even if an entire rack is damaged or offline (for example, due to failure of a shared resource like a network switch or power circuit). It also means that traffic, especially reads, for a chunk can exploit the aggregate bandwidth of multiple racks. On the other hand, write traffic has to flow through multiple racks, a tradeoff we make willingly.

4.3 Creation, Re-replication, Rebalancing

Chunk replicas are created for three reasons: chunk creation, re-replication, and rebalancing.

When the master *creates* a chunk, it chooses where to place the initially empty replicas. It considers several factors. (1) We want to place new replicas on chunkservers with below-average disk space utilization. Over time this will equalize disk utilization across chunkservers. (2) We want to limit the number of "recent" creations on each chunkserver. Although creation itself is cheap, it reliably predicts imminent heavy write traffic because chunks are created when demanded by writes, and in our append-once-read-many workload they typically become practically read-only once they have been completely written. (3) As discussed above, we want to spread replicas of a chunk across racks.

The master *re-replicates* a chunk as soon as the number of available replicas falls below a user-specified goal. This could happen for various reasons: a chunkserver becomes unavailable, it reports that its replica may be corrupted, one of its disks is disabled because of errors, or the replication goal is increased. Each chunk that needs to be re-replicated is prioritized based on several factors. One is how far it is from its replication goal. For example, we give higher priority to a chunk that has lost two replicas than to a chunk that has lost only one. In addition, we prefer to first re-replicate chunks for live files as opposed to chunks that belong to recently deleted files (see Section 4.4). Finally, to minimize the impact of failures on running applications, we boost the priority of any chunk that is blocking client progress.

The master picks the highest priority chunk and "clones" it by instructing some chunkserver to copy the chunk data directly from an existing valid replica. The new replica is placed with goals similar to those for creation: equalizing disk space utilization, limiting active clone operations on any single chunkserver, and spreading replicas across racks. To keep cloning traffic from overwhelming client traffic, the master limits the numbers of active clone operations both for the cluster and for each chunkserver. Additionally, each chunkserver limits the amount of bandwidth it spends on each clone operation by throttling its read requests to the source chunkserver.

Finally, the master *rebalances* replicas periodically: it examines the current replica distribution and moves replicas for better disk space and load balancing. Also through this process, the master gradually fills up a new chunkserver rather than instantly swamps it with new chunks and the heavy write traffic that comes with them. The placement criteria for the new replica are similar to those discussed above. In addition, the master must also choose which existing replica to remove. In general, it prefers to remove those on chunkservers with below-average free space so as to equalize disk space usage.

4.4 Garbage Collection

After a file is deleted, GFS does not immediately reclaim the available physical storage. It does so only lazily during regular garbage collection at both the file and chunk levels. We find that this approach makes the system much simpler and more reliable.

4.4.1 Mechanism

When a file is deleted by the application, the master logs the deletion immediately just like other changes. However instead of reclaiming resources immediately, the file is just renamed to a hidden name that includes the deletion timestamp. During the master's regular scan of the file system namespace, it removes any such hidden files if they have existed for more than three days (the interval is configurable). Until then, the file can still be read under the new, special name and can be undeleted by renaming it back to normal. When the hidden file is removed from the namespace, its in-memory metadata is erased. This effectively severs its links to all its chunks.

In a similar regular scan of the chunk namespace, the master identifies orphaned chunks (i.e., those not reachable from any file) and erases the metadata for those chunks. In a *HeartBeat* message regularly exchanged with the master, each chunkserver reports a subset of the chunks it has, and the master replies with the identity of all chunks that are no longer present in the master's metadata. The chunkserver is free to delete its replicas of such chunks.

4.4.2 Discussion

Although distributed garbage collection is a hard problem that demands complicated solutions in the context of programming languages, it is quite simple in our case. We can easily identify all references to chunks: they are in the file-to-chunk mappings maintained exclusively by the master. We can also easily identify all the chunk replicas: they are Linux files under designated directories on each chunkserver. Any such replica not known to the master is "garbage."

The garbage collection approach to storage reclamation offers several advantages over eager deletion. First, it is simple and reliable in a large-scale distributed system where component failures are common. Chunk creation may succeed on some chunkservers but not others, leaving replicas that the master does not know exist. Replica deletion messages may be lost, and the master has to remember to resend them across failures, both its own and the chunkserver's. Garbage collection provides a uniform and dependable way to clean up any replicas not known to be useful. Second, it merges storage reclamation into the regular background activities of the master, such as the regular scans of namespaces and handshakes with chunkservers. Thus, it is done in batches and the cost is amortized. Moreover, it is done only when the master is relatively free. The master can respond more promptly to client requests that demand timely attention. Third, the delay in reclaiming storage provides a safety net against accidental, irreversible deletion.

In our experience, the main disadvantage is that the delay sometimes hinders user effort to fine tune usage when storage is tight. Applications that repeatedly create and delete temporary files may not be able to reuse the storage right away. We address these issues by expediting storage reclamation if a deleted file is explicitly deleted again. We also allow users to apply different replication and reclamation policies to different parts of the namespace. For example, users can specify that all the chunks in the files within some directory tree are to be stored without replication, and any deleted files are immediately and irrevocably removed from the file system state.

4.5 Stale Replica Detection

Chunk replicas may become stale if a chunkserver fails and misses mutations to the chunk while it is down. For each chunk, the master maintains a *chunk version number* to distinguish between up-to-date and stale replicas.

Whenever the master grants a new lease on a chunk, it increases the chunk version number and informs the up-to-date replicas. The master and these replicas all record the new version number in their persistent state. This occurs before any client is notified and therefore before it can start writing to the chunk. If another replica is currently unavailable, its chunk version number will not be advanced. The master will detect that this chunkserver has a stale replica when the chunkserver restarts and reports its set of chunks and their associated version numbers. If the master sees a version number greater than the one in its records, the master assumes that it failed when granting the lease and so takes the higher version to be up-to-date.

The master removes stale replicas in its regular garbage collection. Before that, it effectively considers a stale replica not to exist at all when it replies to client requests for chunk information. As another safeguard, the master includes the chunk version number when it informs clients which chunkserver holds a lease on a chunk or when it instructs a chunkserver to read the chunk from another chunkserver in a cloning operation. The client or the chunkserver verifies the version number when it performs the operation so that it is always accessing up-to-date data.

5. FAULT TOLERANCE AND DIAGNOSIS

One of our greatest challenges in designing the system is dealing with frequent component failures. The quality and quantity of components together make these problems more the norm than the exception: we cannot completely trust the machines, nor can we completely trust the disks. Component failures can result in an unavailable system or, worse, corrupted data. We discuss how we meet these challenges and the tools we have built into the system to diagnose problems when they inevitably occur.

5.1 High Availability

Among hundreds of servers in a GFS cluster, some are bound to be unavailable at any given time. We keep the overall system highly available with two simple yet effective strategies: fast recovery and replication.

5.1.1 Fast Recovery

Both the master and the chunkserver are designed to restore their state and start in seconds no matter how they terminated. In fact, we do not distinguish between normal and abnormal termination; servers are routinely shut down just by killing the process. Clients and other servers experience a minor hiccup as they time out on their outstanding requests, reconnect to the restarted server, and retry. Section 6.2.2 reports observed startup times.

5.1.2 Chunk Replication

As discussed earlier, each chunk is replicated on multiple chunkservers on different racks. Users can specify different replication levels for different parts of the file namespace. The default is three. The master clones existing replicas as needed to keep each chunk fully replicated as chunkservers go offline or detect corrupted replicas through checksum verification (see Section 5.2). Although replication has served us well, we are exploring other forms of cross-server redundancy such as parity or erasure codes for our increasing read-only storage requirements. We expect that it is challenging but manageable to implement these more complicated redundancy schemes in our very loosely coupled system because our traffic is dominated by appends and reads rather than small random writes.

5.1.3 Master Replication

The master state is replicated for reliability. Its operation log and checkpoints are replicated on multiple machines. A mutation to the state is considered committed only after its log record has been flushed to disk locally and on all master replicas. For simplicity, one master process remains in charge of all mutations as well as background activities such as garbage collection that change the system internally. When it fails, it can restart almost instantly. If its machine or disk fails, monitoring infrastructure outside GFS starts a new master process elsewhere with the replicated operation log. Clients use only the canonical name of the master (e.g. gfs-test), which is a DNS alias that can be changed if the master is relocated to another machine.

Moreover, "shadow" masters provide read-only access to the file system even when the primary master is down. They are shadows, not mirrors, in that they may lag the primary slightly, typically fractions of a second. They enhance read availability for files that are not being actively mutated or applications that do not mind getting slightly stale results. In fact, since file content is read from chunkservers, applications do not observe stale file content. What could be

stale within short windows is file metadata, like directory contents or access control information.

To keep itself informed, a shadow master reads a replica of the growing operation log and applies the same sequence of changes to its data structures exactly as the primary does. Like the primary, it polls chunkservers at startup (and infrequently thereafter) to locate chunk replicas and exchanges frequent handshake messages with them to monitor their status. It depends on the primary master only for replica location updates resulting from the primary's decisions to create and delete replicas.

5.2 Data Integrity

Each chunkserver uses checksumming to detect corruption of stored data. Given that a GFS cluster often has thousands of disks on hundreds of machines, it regularly experiences disk failures that cause data corruption or loss on both the read and write paths. (See Section 7 for one cause.) We can recover from corruption using other chunk replicas, but it would be impractical to detect corruption by comparing replicas across chunkservers. Moreover, divergent replicas may be legal: the semantics of GFS mutations, in particular atomic record append as discussed earlier, does not guarantee identical replicas. Therefore, each chunkserver must independently verify the integrity of its own copy by maintaining checksums.

A chunk is broken up into 64 KB blocks. Each has a corresponding 32 bit checksum. Like other metadata, checksums are kept in memory and stored persistently with logging, separate from user data.

For reads, the chunkserver verifies the checksum of data blocks that overlap the read range before returning any data to the requester, whether a client or another chunkserver. Therefore chunkservers will not propagate corruptions to other machines. If a block does not match the recorded checksum, the chunkserver returns an error to the requestor and reports the mismatch to the master. In response, the requestor will read from other replicas, while the master will clone the chunk from another replica. After a valid new replica is in place, the master instructs the chunkserver that reported the mismatch to delete its replica.

Checksumming has little effect on read performance for several reasons. Since most of our reads span at least a few blocks, we need to read and checksum only a relatively small amount of extra data for verification. GFS client code further reduces this overhead by trying to align reads at checksum block boundaries. Moreover, checksum lookups and comparison on the chunkserver are done without any I/O, and checksum calculation can often be overlapped with I/Os.

Checksum computation is heavily optimized for writes that append to the end of a chunk (as opposed to writes that overwrite existing data) because they are dominant in our workloads. We just incrementally update the checksum for the last partial checksum block, and compute new checksums for any brand new checksum blocks filled by the append. Even if the last partial checksum block is already corrupted and we fail to detect it now, the new checksum value will not match the stored data, and the corruption will be detected as usual when the block is next read.

In contrast, if a write overwrites an existing range of the chunk, we must read and verify the first and last blocks of the range being overwritten, then perform the write, and finally compute and record the new checksums. If we do not verify the first and last blocks before overwriting them partially, the new checksums may hide corruption that exists in the regions not being overwritten.

During idle periods, chunkservers can scan and verify the contents of inactive chunks. This allows us to detect corruption in chunks that are rarely read. Once the corruption is detected, the master can create a new uncorrupted replica and delete the corrupted replica. This prevents an inactive but corrupted chunk replica from fooling the master into thinking that it has enough valid replicas of a chunk.

5.3 Diagnostic Tools

Extensive and detailed diagnostic logging has helped immeasurably in problem isolation, debugging, and performance analysis, while incurring only a minimal cost. Without logs, it is hard to understand transient, non-repeatable interactions between machines. GFS servers generate diagnostic logs that record many significant events (such as chunkservers going up and down) and all RPC requests and replies. These diagnostic logs can be freely deleted without affecting the correctness of the system. However, we try to keep these logs around as far as space permits.

The RPC logs include the exact requests and responses sent on the wire, except for the file data being read or written. By matching requests with replies and collating RPC records on different machines, we can reconstruct the entire interaction history to diagnose a problem. The logs also serve as traces for load testing and performance analysis.

The performance impact of logging is minimal (and far outweighed by the benefits) because these logs are written sequentially and asynchronously. The most recent events are also kept in memory and available for continuous online monitoring.

6. MEASUREMENTS

In this section we present a few micro-benchmarks to illustrate the bottlenecks inherent in the GFS architecture and implementation, and also some numbers from real clusters in use at Google.

6.1 Micro-benchmarks

We measured performance on a GFS cluster consisting of one master, two master replicas, 16 chunkservers, and 16 clients. Note that this configuration was set up for ease of testing. Typical clusters have hundreds of chunkservers and hundreds of clients.

All the machines are configured with dual 1.4 GHz PIII processors, 2 GB of memory, two 80 GB 5400 rpm disks, and a 100 Mbps full-duplex Ethernet connection to an HP 2524 switch. All 19 GFS server machines are connected to one switch, and all 16 client machines to the other. The two switches are connected with a 1 Gbps link.

6.1.1 Reads

N clients read simultaneously from the file system. Each client reads a randomly selected 4 MB region from a 320 GB file set. This is repeated 256 times so that each client ends up reading 1 GB of data. The chunkservers taken together have only 32 GB of memory, so we expect at most a 10% hit rate in the Linux buffer cache. Our results should be close to cold cache results.

Figure 3(a) shows the aggregate read rate for N clients and its theoretical limit. The limit peaks at an aggregate of 125 MB/s when the 1 Gbps link between the two switches is saturated, or 12.5 MB/s per client when its 100 Mbps network interface gets saturated, whichever applies. The observed read rate is 10 MB/s, or 80% of the per-client limit, when just one client is reading. The aggregate read rate reaches 94 MB/s, about 75% of the 125 MB/s link limit, for 16 readers, or 6 MB/s per client. The efficiency drops from 80% to 75% because as the number of readers increases, so does the probability that multiple readers simultaneously read from the same chunkserver.

6.1.2 Writes

N clients write simultaneously to N distinct files. Each client writes 1 GB of data to a new file in a series of 1 MB writes. The aggregate write rate and its theoretical limit are shown in Figure 3(b). The limit plateaus at 67 MB/s because we need to write each byte to 3 of the 16 chunkservers, each with a 12.5 MB/s input connection.

The write rate for one client is 6.3 MB/s, about half of the limit. The main culprit for this is our network stack. It does not interact very well with the pipelining scheme we use for pushing data to chunk replicas. Delays in propagating data from one replica to another reduce the overall write rate.

Aggregate write rate reaches 35 MB/s for 16 clients (or 2.2 MB/s per client), about half the theoretical limit. As in the case of reads, it becomes more likely that multiple clients write concurrently to the same chunkserver as the number of clients increases. Moreover, collision is more likely for 16 writers than for 16 readers because each write involves three different replicas.

Writes are slower than we would like. In practice this has not been a major problem because even though it increases the latencies as seen by individual clients, it does not significantly affect the aggregate write bandwidth delivered by the system to a large number of clients.

6.1.3 Record Appends

Figure 3(c) shows record append performance. N clients append simultaneously to a single file. Performance is limited by the network bandwidth of the chunkservers that store the last chunk of the file, independent of the number of clients. It starts at 6.0 MB/s for one client and drops to 4.8 MB/s for 16 clients, mostly due to congestion and variances in network transfer rates seen by different clients.

Our applications tend to produce multiple such files concurrently. In other words, N clients append to M shared files simultaneously where both N and M are in the dozens or hundreds. Therefore, the chunkserver network congestion in our experiment is not a significant issue in practice because a client can make progress on writing one file while the chunkservers for another file are busy.

6.2 Real World Clusters

We now examine two clusters in use within Google that are representative of several others like them. Cluster A is used regularly for research and development by over a hundred engineers. A typical task is initiated by a human user and runs up to several hours. It reads through a few MBs to a few TBs of data, transforms or analyzes the data, and writes the results back to the cluster. Cluster B is primarily used for production data processing. The tasks last much longer and continuously generate and process multi-TB data sets with only occasional human intervention. In both cases, a single "task" consists of many processes on many machines reading and writing many files simultaneously.

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Table 2: Characteristics of two GFS clusters

6.2.1 Storage

As shown by the first five entries in the table, both clusters have hundreds of chunkservers, support many TBs of disk space, and are fairly but not completely full. "Used space" includes all chunk replicas. Virtually all files are replicated three times. Therefore, the clusters store 18 TB and 52 TB of file data respectively.

The two clusters have similar numbers of files, though B has a larger proportion of dead files, namely files which were deleted or replaced by a new version but whose storage have not yet been reclaimed. It also has more chunks because its files tend to be larger.

6.2.2 Metadata

The chunkservers in aggregate store tens of GBs of metadata, mostly the checksums for 64 KB blocks of user data. The only other metadata kept at the chunkservers is the chunk version number discussed in Section 4.5.

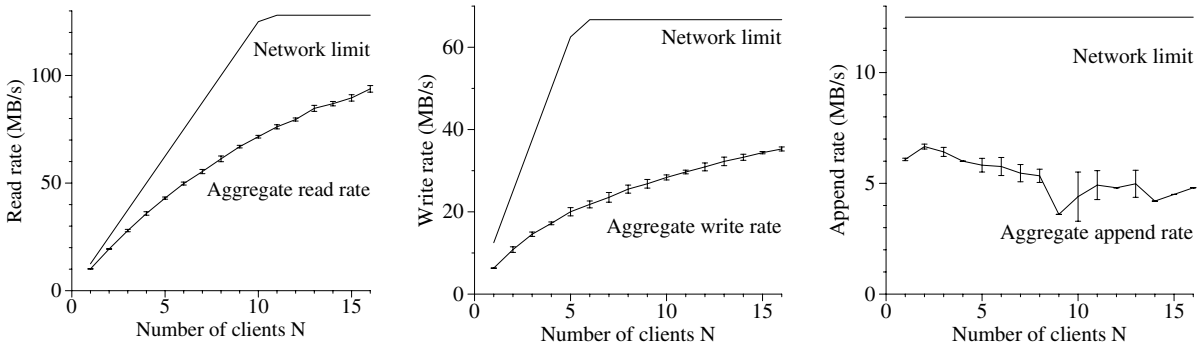
The metadata kept at the master is much smaller, only tens of MBs, or about 100 bytes per file on average. This agrees with our assumption that the size of the master's memory does not limit the system's capacity in practice. Most of the per-file metadata is the file names stored in a prefix-compressed form. Other metadata includes file ownership and permissions, mapping from files to chunks, and each chunk's current version. In addition, for each chunk we store the current replica locations and a reference count for implementing copy-on-write.

Each individual server, both chunkservers and the master, has only 50 to 100 MB of metadata. Therefore recovery is fast: it takes only a few seconds to read this metadata from disk before the server is able to answer queries. However, the master is somewhat hobbled for a period – typically 30 to 60 seconds – until it has fetched chunk location information from all chunkservers.

6.2.3 Read and Write Rates

Table 3 shows read and write rates for various time periods. Both clusters had been up for about one week when these measurements were taken. (The clusters had been restarted recently to upgrade to a new version of GFS.)

The average write rate was less than 30 MB/s since the restart. When we took these measurements, B was in the middle of a burst of write activity generating about 100 MB/s of data, which produced a 300 MB/s network load because writes are propagated to three replicas.



(a) Reads (b) Writes (c) Record appends

Figure 3: Aggregate Throughputs. Top curves show theoretical limits imposed by our network topology. Bottom curves show measured throughputs. They have error bars that show 95% confidence intervals, which are illegible in some cases because of low variance in measurements.

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

Table 3: Performance Metrics for Two GFS Clusters

The read rates were much higher than the write rates. The total workload consists of more reads than writes as we have assumed. Both clusters were in the middle of heavy read activity. In particular, A had been sustaining a read rate of 580 MB/s for the preceding week. Its network configuration can support 750 MB/s, so it was using its resources efficiently. Cluster B can support peak read rates of 1300 MB/s, but its applications were using just 380 MB/s.

6.2.4 Master Load

Table 3 also shows that the rate of operations sent to the master was around 200 to 500 operations per second. The master can easily keep up with this rate, and therefore is not a bottleneck for these workloads.

In an earlier version of GFS, the master was occasionally a bottleneck for some workloads. It spent most of its time sequentially scanning through large directories (which contained hundreds of thousands of files) looking for particular files. We have since changed the master data structures to allow efficient binary searches through the namespace. It can now easily support many thousands of file accesses per second. If necessary, we could speed it up further by placing name lookup caches in front of the namespace data structures.

6.2.5 Recovery Time

After a chunkserver fails, some chunks will become underreplicated and must be cloned to restore their replication levels. The time it takes to restore all such chunks depends on the amount of resources. In one experiment, we killed a single chunkserver in cluster B. The chunkserver had about 15,000 chunks containing 600 GB of data. To limit the impact on running applications and provide leeway for scheduling decisions, our default parameters limit this cluster to 91 concurrent clonings (40% of the number of chunkservers) where each clone operation is allowed to consume at most 6.25 MB/s (50 Mbps). All chunks were restored in 23.2 minutes, at an effective replication rate of 440 MB/s.

In another experiment, we killed two chunkservers each with roughly 16,000 chunks and 660 GB of data. This double failure reduced 266 chunks to having a single replica. These 266 chunks were cloned at a higher priority, and were all restored to at least 2x replication within 2 minutes, thus putting the cluster in a state where it could tolerate another chunkserver failure without data loss.

6.3 Workload Breakdown

In this section, we present a detailed breakdown of the workloads on two GFS clusters comparable but not identical to those in Section 6.2. Cluster X is for research and development while cluster Y is for production data processing.

6.3.1 Methodology and Caveats

These results include only client originated requests so that they reflect the workload generated by our applications for the file system as a whole. They do not include interserver requests to carry out client requests or internal background activities, such as forwarded writes or rebalancing.

Statistics on I/O operations are based on information heuristically reconstructed from actual RPC requests logged by GFS servers. For example, GFS client code may break a read into multiple RPCs to increase parallelism, from which we infer the original read. Since our access patterns are highly stylized, we expect any error to be in the noise. Explicit logging by applications might have provided slightly more accurate data, but it is logistically impossible to recompile and restart thousands of running clients to do so and cumbersome to collect the results from as many machines.

One should be careful not to overly generalize from our workload. Since Google completely controls both GFS and its applications, the applications tend to be tuned for GFS, and conversely GFS is designed for these applications. Such mutual influence may also exist between general applications

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
0K	0.4	2.6	0	0	0	0
1B..1K	0.1	4.1	6.6	4.9	0.2	9.2
1K..8K	65.2	38.5	0.4	1.0	18.9	15.2
8K..64K	29.9	45.1	17.8	43.0	78.0	2.8
64K..128K	0.1	0.7	2.3	1.9	< .1	4.3
128K..256K	0.2	0.3	31.6	0.4	< .1	10.6
256K..512K	0.1	0.1	4.2	7.7	< .1	31.2
512K..1M	3.9	6.9	35.5	28.7	2.2	25.5
1M..inf	0.1	1.8	1.5	12.3	0.7	2.2

Table 4: Operations Breakdown by Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested.

and file systems, but the effect is likely more pronounced in our case.

6.3.2 Chunkserver Workload

Table 4 shows the distribution of operations by size. Read sizes exhibit a bimodal distribution. The small reads (under 64 KB) come from seek-intensive clients that look up small pieces of data within huge files. The large reads (over 512 KB) come from long sequential reads through entire files.

A significant number of reads return no data at all in cluster Y. Our applications, especially those in the production systems, often use files as producer-consumer queues. Producers append concurrently to a file while a consumer reads the end of file. Occasionally, no data is returned when the consumer outpaces the producers. Cluster X shows this less often because it is usually used for short-lived data analysis tasks rather than long-lived distributed applications.

Write sizes also exhibit a bimodal distribution. The large writes (over 256 KB) typically result from significant buffering within the writers. Writers that buffer less data, checkpoint or synchronize more often, or simply generate less data account for the smaller writes (under 64 KB).

As for record appends, cluster Y sees a much higher percentage of large record appends than cluster X does because our production systems, which use cluster Y, are more aggressively tuned for GFS.

Table 5 shows the total amount of data transferred in operations of various sizes. For all kinds of operations, the larger operations (over 256 KB) generally account for most of the bytes transferred. Small reads (under 64 KB) do transfer a small but significant portion of the read data because of the random seek workload.

6.3.3 Appends versus Writes

Record appends are heavily used especially in our production systems. For cluster X, the ratio of writes to record appends is 108:1 by bytes transferred and 8:1 by operation counts. For cluster Y, used by the production systems, the ratios are 3.7:1 and 2.5:1 respectively. Moreover, these ratios suggest that for both clusters record appends tend to be larger than writes. For cluster X, however, the overall usage of record append during the measured period is fairly low and so the results are likely skewed by one or two applications with particular buffer size choices.

As expected, our data mutation workload is dominated by appending rather than overwriting. We measured the amount of data overwritten on primary replicas. This approximates the case where a client deliberately overwrites previous written data rather than appends new data. For cluster X, overwriting accounts for under 0.0001% of bytes mutated and under 0.0003% of mutation operations. For cluster Y, the ratios are both 0.05%. Although this is minute, it is still higher than we expected. It turns out that most of these overwrites came from client retries due to errors or timeouts. They are not part of the workload *per se* but a consequence of the retry mechanism.

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
1B..1K	< .1	< .1	< .1	< .1	< .1	< .1
1K..8K	13.8	3.9	< .1	< .1	< .1	0.1
8K..64K	11.4	9.3	2.4	5.9	2.3	0.3
64K..128K	0.3	0.7	0.3	0.3	22.7	1.2
128K..256K	0.8	0.6	16.5	0.2	< .1	5.8
256K..512K	1.4	0.3	3.4	7.7	< .1	38.4
512K..1M	65.9	55.1	74.1	58.0	.1	46.8
1M..inf	6.4	30.1	3.3	28.0	53.9	7.4

Table 5: Bytes Transferred Breakdown by Operation Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested. The two may differ if the read attempts to read beyond end of file, which by design is not uncommon in our workloads.

Cluster	X	Y
Open	26.1	16.3
Delete	0.7	1.5
FindLocation	64.3	65.8
FindLeaseHolder	7.8	13.4
FindMatchingFiles	0.6	2.2
All other combined	0.5	0.8

Table 6: Master Requests Breakdown by Type (%)

6.3.4 Master Workload

Table 6 shows the breakdown by type of requests to the master. Most requests ask for chunk locations (*FindLocation*) for reads and lease holder information (*FindLeaseLocker*) for data mutations.

Clusters X and Y see significantly different numbers of *Delete* requests because cluster Y stores production data sets that are regularly regenerated and replaced with newer versions. Some of this difference is further hidden in the difference in *Open* requests because an old version of a file may be implicitly deleted by being opened for write from scratch (mode "w" in Unix open terminology).

FindMatchingFiles is a pattern matching request that supports "ls" and similar file system operations. Unlike other requests for the master, it may process a large part of the namespace and so may be expensive. Cluster Y sees it much more often because automated data processing tasks tend to examine parts of the file system to understand global application state. In contrast, cluster X's applications are under more explicit user control and usually know the names of all needed files in advance.

7. EXPERIENCES

In the process of building and deploying GFS, we have experienced a variety of issues, some operational and some technical.

Initially, GFS was conceived as the backend file system for our production systems. Over time, the usage evolved to include research and development tasks. It started with little support for things like permissions and quotas but now includes rudimentary forms of these. While production systems are well disciplined and controlled, users sometimes are not. More infrastructure is required to keep users from interfering with one another.

Some of our biggest problems were disk and Linux related. Many of our disks claimed to the Linux driver that they supported a range of IDE protocol versions but in fact responded reliably only to the more recent ones. Since the protocol versions are very similar, these drives mostly worked, but occasionally the mismatches would cause the drive and the kernel to disagree about the drive's state. This would corrupt data silently due to problems in the kernel. This problem motivated our use of checksums to detect data corruption, while concurrently we modified the kernel to handle these protocol mismatches.

Earlier we had some problems with Linux 2.2 kernels due to the cost of `fsync()`. Its cost is proportional to the size of the file rather than the size of the modified portion. This was a problem for our large operation logs especially before we implemented checkpointing. We worked around this for a time by using synchronous writes and eventually migrated to Linux 2.4.

Another Linux problem was a single reader-writer lock which any thread in an address space must hold when it pages in from disk (reader lock) or modifies the address space in an `mmap()` call (writer lock). We saw transient timeouts in our system under light load and looked hard for resource bottlenecks or sporadic hardware failures. Eventually, we found that this single lock blocked the primary network thread from mapping new data into memory while the disk threads were paging in previously mapped data. Since we are mainly limited by the network interface rather than by memory copy bandwidth, we worked around this by replacing `mmap()` with `pread()` at the cost of an extra copy.

Despite occasional problems, the availability of Linux code has helped us time and again to explore and understand system behavior. When appropriate, we improve the kernel and share the changes with the open source community.

8. RELATED WORK

Like other large distributed file systems such as AFS [5], GFS provides a location independent namespace which enables data to be moved transparently for load balance or fault tolerance. Unlike AFS, GFS spreads a file's data across storage servers in a way more akin to xFS [1] and Swift [3] in order to deliver aggregate performance and increased fault tolerance.

As disks are relatively cheap and replication is simpler than more sophisticated RAID [9] approaches, GFS currently uses only replication for redundancy and so consumes more raw storage than xFS or Swift.

In contrast to systems like AFS, xFS, Frangipani [12], and Intermezzo [6], GFS does not provide any caching below the file system interface. Our target workloads have little reuse within a single application run because they either stream through a large data set or randomly seek within it and read small amounts of data each time.

Some distributed file systems like Frangipani, xFS, Minnesota's GFS[11] and GPFS [10] remove the centralized server and rely on distributed algorithms for consistency and management. We opt for the centralized approach in order to simplify the design, increase its reliability, and gain flexibility. In particular, a centralized master makes it much easier to implement sophisticated chunk placement and replication policies since the master already has most of the relevant information and controls how it changes. We address fault tolerance by keeping the master state small and fully replicated on other machines. Scalability and high availability (for reads) are currently provided by our shadow master mechanism. Updates to the master state are made persistent by appending to a write-ahead log. Therefore we could adapt a primary-copy scheme like the one in Harp [7] to provide high availability with stronger consistency guarantees than our current scheme.

We are addressing a problem similar to Lustre [8] in terms of delivering aggregate performance to a large number of clients. However, we have simplified the problem significantly by focusing on the needs of our applications rather than building a POSIX-compliant file system. Additionally, GFS assumes large number of unreliable components and so fault tolerance is central to our design.

GFS most closely resembles the NASD architecture [4]. While the NASD architecture is based on network-attached disk drives, GFS uses commodity machines as chunkservers, as done in the NASD prototype. Unlike the NASD work, our chunkservers use lazily allocated fixed-size chunks rather than variable-length objects. Additionally, GFS implements features such as rebalancing, replication, and recovery that are required in a production environment.

Unlike Minnesota's GFS and NASD, we do not seek to alter the model of the storage device. We focus on addressing day-to-day data processing needs for complicated distributed systems with existing commodity components.

The producer-consumer queues enabled by atomic record appends address a similar problem as the distributed queues in River [2]. While River uses memory-based queues distributed across machines and careful data flow control, GFS uses a persistent file that can be appended to concurrently by many producers. The River model supports m-to-n distributed queues but lacks the fault tolerance that comes with persistent storage, while GFS only supports m-to-1 queues efficiently. Multiple consumers can read the same file, but they must coordinate to partition the incoming load.

9. CONCLUSIONS

The Google File System demonstrates the qualities essential 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 processing tasks of a similar magnitude and cost consciousness.

We started by reexamining traditional file system assumptions 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 sequentially), and both extend and relax the standard file system interface to improve the overall system.

Our system provides fault tolerance by constant monitoring, replicating crucial data, and fast and automatic recovery. Chunk replication allows us to tolerate chunkserver

failures. The frequency of these failures motivated a novel online repair mechanism that regularly and transparently repairs 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 involvement in common operations is minimized by a large chunk size and by chunk leases, which delegates authority to primary replicas in data mutations. This makes possible a simple, 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.

ACKNOWLEDGMENTS

We wish to thank the following people for their contributions to the system or the paper. Brain Bershad (our shepherd) and the anonymous reviewers gave us valuable comments and suggestions. Anurag Acharya, Jeff Dean, and David desJardins contributed to the early design. Fay Chang worked on comparison of replicas across chunkservers. Guy Edjlali worked on storage quota. Markus Gutschke worked on a testing framework and security enhancements. David Kramer worked on performance enhancements. Fay Chang, Urs Hoelzle, Max Ibel, Sharon Perl, Rob Pike, and Debby Wallach commented on earlier drafts of the paper. Many of our colleagues at Google bravely trusted their data to a new file system and gave us useful feedback. Yoshka helped with early testing.

REFERENCES

[1] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang. Serverless network file systems. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, Georgia, May 1999.

[3] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computer Systems*, 4(4):405–436, 1991.

[4] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th Architectural Support for Programming Languages and Operating Systems*, pages 92–103, San Jose, California, October 1998.

[5] John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[6] InterMezzo. http://www.inter-mezzo.org, 2003.

[7] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *13th Symposium on Operating System Principles*, pages 226–238, Pacific Grove, CA, October 1991.

[8] Lustre. http://www.lustreorg, 2003.

[9] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, Illinois, September 1988.

[10] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, pages 231–244, Monterey, California, January 2002.

[11] Steven R. Soltis, Thomas M. Ruwart, and Matthew T. O'Keefe. The Gobal File System. In *Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, College Park, Maryland, September 1996.

[12] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 224–237, Saint-Malo, France, October 1997.

yore

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated 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 automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily 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 terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

for a rewrite of our production indexing system. Section 7 discusses related and future work.

2 Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.

Map, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the *Reduce* function.

The *Reduce* function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

2.1 Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudo-code:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

The `map` function emits each word plus an associated count of occurrences (just '1' in this simple example). The `reduce` function sums together all counts emitted for a particular word.

In addition, the user writes code to fill in a *mapreduce specification* object with the names of the input and output files, and optional tuning parameters. The user then invokes the *MapReduce* function, passing it the specification object. The user's code is linked together with the MapReduce library (implemented in C++). Appendix A contains the full program text for this example.

2.2 Types

Even though the previous pseudo-code is written in terms of string inputs and outputs, conceptually the map and reduce functions supplied by the user have associated types:

map	`(k1,v1)`	$\rightarrow$ `list(k2,v2)`
reduce	`(k2,list(v2))`	$\rightarrow$ `list(v2)`

I.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

Our C++ implementation passes strings to and from the user-defined functions and leaves it to the user code to convert between strings and appropriate types.

2.3 More Examples

Here are a few simple examples of interesting programs that can be easily expressed as MapReduce computations.

Distributed Grep: The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

Count of URL Access Frequency: The map function processes logs of web page requests and outputs $\langle$`URL`, `1`$\rangle$. The reduce function adds together all values for the same URL and emits a $\langle$`URL`, `total count`$\rangle$ pair.

Reverse Web-Link Graph: The map function outputs $\langle$`target`, `source`$\rangle$ pairs for each link to a `target` URL found in a page named `source`. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: $\langle$`target`, $list$(`source`)$\rangle$

Term-Vector per Host: A term vector summarizes the most important words that occur in a document or a set of documents as a list of $\langle word, frequency \rangle$ pairs. The map function emits a $\langle$`hostname`, `term vector`$\rangle$ pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final $\langle$`hostname`, `term vector`$\rangle$ pair.

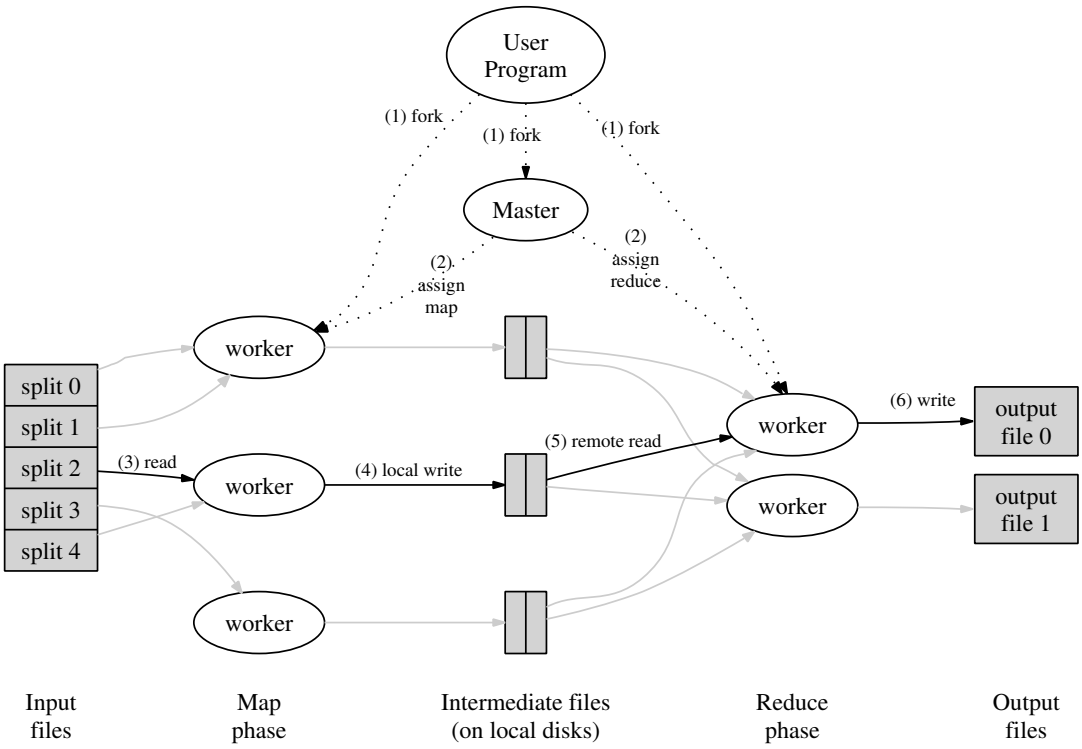


Figure 1: Execution overview

Inverted Index: The map function parses each document, and emits a sequence of $\langle$`word, document ID`$\rangle$ pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a $\langle$`word`, $list$(`document ID`)$\rangle$ pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

Distributed Sort: The map function extracts the key from each record, and emits a $\langle$`key, record`$\rangle$ pair. The reduce function emits all pairs unchanged. This computation depends on the partitioning facilities described in Section 4.1 and the ordering properties described in Section 4.2.

3 Implementation

Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines.

This section describes an implementation targeted to the computing environment in wide use at Google: large clusters of commodity PCs connected together with switched Ethernet [4]. In our environment:

(1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.

(2) Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.

(3) A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.

(4) Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system [8] developed in-house is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.

(5) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

3.1 Execution Overview

The *Map* invocations are distributed across multiple machines by automatically partitioning the input data

into a set of M *splits*. The input splits can be processed in parallel by different machines. *Reduce* invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., $hash(key) \bmod R$). The number of partitions (R) and the partitioning function are specified by the user.

Figure 1 shows the overall flow of a MapReduce operation in our implementation. When the user program calls the `MapReduce` function, the following sequence of actions occurs (the numbered labels in Figure 1 correspond to the numbers in the list below):

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.

2. One of the copies of the program is special – the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.

3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined *Map* function. The intermediate key/value pairs produced by the *Map* function are buffered in memory.

4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's *Reduce* function. The output of the *Reduce* function is appended to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the `MapReduce` call in the user program returns back to the user code.

After successful completion, the output of the mapreduce execution is available in the R output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these R output files into one file – they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

3.2 Master Data Structures

The master keeps several data structures. For each map task and reduce task, it stores the state (*idle*, *in-progress*, or *completed*), and the identity of the worker machine (for non-idle tasks).

The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have *in-progress* reduce tasks.

3.3 Fault Tolerance

Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.

Worker Failure

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial *idle* state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to *idle* and becomes eligible for rescheduling.

Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

When a map task is executed first by worker A and then later executed by worker B (because A failed), all

workers executing reduce tasks are notified of the re-execution. Any reduce task that has not already read the data from worker A will read the data from worker B.

MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce master simply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the MapReduce operation.

Master Failure

It is easy to make the master write periodic checkpoints of the master data structures described above. If the master task dies, a new copy can be started from the last checkpointed state. However, given that there is only a single master, its failure is unlikely; therefore our current implementation aborts the MapReduce computation if the master fails. Clients can check for this condition and retry the MapReduce operation if they desire.

Semantics in the Presence of Failures

When the user-supplied *map* and *reduce* operators are deterministic functions of their input values, our distributed implementation produces the same output as would have been produced by a non-faulting sequential execution of the entire program.

We rely on atomic commits of map and reduce task outputs to achieve this property. Each in-progress task writes its output to private temporary files. A reduce task produces one such file, and a map task produces R such files (one per reduce task). When a map task completes, the worker sends a message to the master and includes the names of the R temporary files in the message. If the master receives a completion message for an already completed map task, it ignores the message. Otherwise, it records the names of R files in a master data structure.

When a reduce task completes, the reduce worker atomically renames its temporary output file to the final output file. If the same reduce task is executed on multiple machines, multiple rename calls will be executed for the same final output file. We rely on the atomic rename operation provided by the underlying file system to guarantee that the final file system state contains just the data produced by one execution of the reduce task.

The vast majority of our *map* and *reduce* operators are deterministic, and the fact that our semantics are equivalent to a sequential execution in this case makes it very easy for programmers to reason about their program's behavior. When the *map* and/or *reduce* operators are non-deterministic, we provide weaker but still reasonable semantics. In the presence of non-deterministic operators, the output of a particular reduce task R_1 is equivalent to the output for R_1 produced by a sequential execution of the non-deterministic program. However, the output for a different reduce task R_2 may correspond to the output for R_2 produced by a different sequential execution of the non-deterministic program.

Consider map task M and reduce tasks R_1 and R_2. Let $e(R_i)$ be the execution of R_i that committed (there is exactly one such execution). The weaker semantics arise because $e(R_1)$ may have read the output produced by one execution of M and $e(R_2)$ may have read the output produced by a different execution of M.

3.4 Locality

Network bandwidth is a relatively scarce resource in our computing environment. We conserve network bandwidth by taking advantage of the fact that the input data (managed by GFS [8]) is stored on the local disks of the machines that make up our cluster. GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines. The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data (e.g., on a worker machine that is on the same network switch as the machine containing the data). When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.

3.5 Task Granularity

We subdivide the map phase into M pieces and the reduce phase into R pieces, as described above. Ideally, M and R should be much larger than the number of worker machines. Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails: the many map tasks it has completed can be spread out across all the other worker machines.

There are practical bounds on how large M and R can be in our implementation, since the master must make $O(M + R)$ scheduling decisions and keeps $O(M * R)$ state in memory as described above. (The constant factors for memory usage are small however: the $O(M * R)$ piece of the state consists of approximately one byte of data per map task/reduce task pair.)

Furthermore, R is often constrained by users because the output of each reduce task ends up in a separate output file. In practice, we tend to choose M so that each individual task is roughly 16 MB to 64 MB of input data (so that the locality optimization described above is most effective), and we make R a small multiple of the number of worker machines we expect to use. We often perform MapReduce computations with $M = 200,000$ and $R = 5,000$, using 2,000 worker machines.

3.6 Backup Tasks

One of the common causes that lengthens the total time taken for a MapReduce operation is a "straggler": a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation. Stragglers can arise for a whole host of reasons. For example, a machine with a bad disk may experience frequent correctable errors that slow its read performance from 30 MB/s to 1 MB/s. The cluster scheduling system may have scheduled other tasks on the machine, causing it to execute the MapReduce code more slowly due to competition for CPU, memory, local disk, or network bandwidth. A recent problem we experienced was a bug in machine initialization code that caused processor caches to be disabled: computations on affected machines slowed down by over a factor of one hundred.

We have a general mechanism to alleviate the problem of stragglers. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining *in-progress* tasks. The task is marked as completed whenever either the primary or the backup execution completes. We have tuned this mechanism so that it typically increases the computational resources used by the operation by no more than a few percent. We have found that this significantly reduces the time to complete large MapReduce operations. As an example, the sort program described in Section 5.3 takes 44% longer to complete when the backup task mechanism is disabled.

4 Refinements

Although the basic functionality provided by simply writing *Map* and *Reduce* functions is sufficient for most needs, we have found a few extensions useful. These are described in this section.

4.1 Partitioning Function

The users of MapReduce specify the number of reduce tasks/output files that they desire (R). Data gets partitioned across these tasks using a partitioning function on the intermediate key. A default partitioning function is provided that uses hashing (e.g. "$hash(key) \bmod R$"). This tends to result in fairly well-balanced partitions. In some cases, however, it is useful to partition data by some other function of the key. For example, sometimes the output keys are URLs, and we want all entries for a single host to end up in the same output file. To support situations like this, the user of the MapReduce library can provide a special partitioning function. For example, using "$hash(Hostname(urlkey)) \bmod R$" as the partitioning function causes all URLs from the same host to end up in the same output file.

4.2 Ordering Guarantees

We guarantee that within a given partition, the intermediate key/value pairs are processed in increasing key order. This ordering guarantee makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output find it convenient to have the data sorted.

4.3 Combiner Function

In some cases, there is significant repetition in the intermediate keys produced by each map task, and the user-specified *Reduce* function is commutative and associative. A good example of this is the word counting example in Section 2.1. Since word frequencies tend to follow a Zipf distribution, each map task will produce hundreds or thousands of records of the form `<the, 1>`. All of these counts will be sent over the network to a single reduce task and then added together by the *Reduce* function to produce one number. We allow the user to specify an optional *Combiner* function that does partial merging of this data before it is sent over the network.

The *Combiner* function is executed on each machine that performs a map task. Typically the same code is used to implement both the combiner and the reduce functions. The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function. The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate file that will be sent to a reduce task.

Partial combining significantly speeds up certain classes of MapReduce operations. Appendix A contains an example that uses a combiner.

4.4 Input and Output Types

The MapReduce library provides support for reading input data in several different formats. For example, "text"

mode input treats each line as a key/value pair: the key is the offset in the file and the value is the contents of the line. Another common supported format stores a sequence of key/value pairs sorted by key. Each input type implementation knows how to split itself into meaningful ranges for processing as separate map tasks (e.g. text mode's range splitting ensures that range splits occur only at line boundaries). Users can add support for a new input type by providing an implementation of a simple *reader* interface, though most users just use one of a small number of predefined input types.

A *reader* does not necessarily need to provide data read from a file. For example, it is easy to define a *reader* that reads records from a database, or from data structures mapped in memory.

In a similar fashion, we support a set of output types for producing data in different formats and it is easy for user code to add support for new output types.

4.5 Side-effects

In some cases, users of MapReduce have found it convenient to produce auxiliary files as additional outputs from their map and/or reduce operators. We rely on the application writer to make such side-effects atomic and idempotent. Typically the application writes to a temporary file and atomically renames this file once it has been fully generated.

We do not provide support for atomic two-phase commits of multiple output files produced by a single task. Therefore, tasks that produce multiple output files with cross-file consistency requirements should be deterministic. This restriction has never been an issue in practice.

4.6 Skipping Bad Records

Sometimes there are bugs in user code that cause the *Map* or *Reduce* functions to crash deterministically on certain records. Such bugs prevent a MapReduce operation from completing. The usual course of action is to fix the bug, but sometimes this is not feasible; perhaps the bug is in a third-party library for which source code is unavailable. Also, sometimes it is acceptable to ignore a few records, for example when doing statistical analysis on a large data set. We provide an optional mode of execution where the MapReduce library detects which records cause deterministic crashes and skips these records in order to make forward progress.

Each worker process installs a signal handler that catches segmentation violations and bus errors. Before invoking a user *Map* or *Reduce* operation, the MapReduce library stores the sequence number of the argument in a global variable. If the user code generates a signal, the signal handler sends a "last gasp" UDP packet that contains the sequence number to the MapReduce master. When the master has seen more than one failure on a particular record, it indicates that the record should be skipped when it issues the next re-execution of the corresponding Map or Reduce task.

4.7 Local Execution

Debugging problems in *Map* or *Reduce* functions can be tricky, since the actual computation happens in a distributed system, often on several thousand machines, with work assignment decisions made dynamically by the master. To help facilitate debugging, profiling, and small-scale testing, we have developed an alternative implementation of the MapReduce library that sequentially executes all of the work for a MapReduce operation on the local machine. Controls are provided to the user so that the computation can be limited to particular map tasks. Users invoke their program with a special flag and can then easily use any debugging or testing tools they find useful (e.g. `gdb`).

4.8 Status Information

The master runs an internal HTTP server and exports a set of status pages for human consumption. The status pages show the progress of the computation, such as how many tasks have been completed, how many are in progress, bytes of input, bytes of intermediate data, bytes of output, processing rates, etc. The pages also contain links to the standard error and standard output files generated by each task. The user can use this data to predict how long the computation will take, and whether or not more resources should be added to the computation. These pages can also be used to figure out when the computation is much slower than expected.

In addition, the top-level status page shows which workers have failed, and which map and reduce tasks they were processing when they failed. This information is useful when attempting to diagnose bugs in the user code.

4.9 Counters

The MapReduce library provides a counter facility to count occurrences of various events. For example, user code may want to count total number of words processed or the number of German documents indexed, etc.

To use this facility, user code creates a named counter object and then increments the counter appropriately in the *Map* and/or *Reduce* function. For example:

```
Counter* uppercase;
uppercase = GetCounter("uppercase");

map(String name, String contents):
  for each word w in contents:
    if (IsCapitalized(w)):
      uppercase->Increment();
    EmitIntermediate(w, "1");
```

The counter values from individual worker machines are periodically propagated to the master (piggybacked on the ping response). The master aggregates the counter values from successful map and reduce tasks and returns them to the user code when the MapReduce operation is completed. The current counter values are also displayed on the master status page so that a human can watch the progress of the live computation. When aggregating counter values, the master eliminates the effects of duplicate executions of the same map or reduce task to avoid double counting. (Duplicate executions can arise from our use of backup tasks and from re-execution of tasks due to failures.)

Some counter values are automatically maintained by the MapReduce library, such as the number of input key/value pairs processed and the number of output key/value pairs produced.

Users have found the counter facility useful for sanity checking the behavior of MapReduce operations. For example, in some MapReduce operations, the user code may want to ensure that the number of output pairs produced exactly equals the number of input pairs processed, or that the fraction of German documents processed is within some tolerable fraction of the total number of documents processed.

5 Performance

In this section we measure the performance of MapReduce on two computations running on a large cluster of machines. One computation searches through approximately one terabyte of data looking for a particular pattern. The other computation sorts approximately one terabyte of data.

These two programs are representative of a large subset of the real programs written by users of MapReduce – one class of programs shuffles data from one representation to another, and another class extracts a small amount of interesting data from a large data set.

5.1 Cluster Configuration

All of the programs were executed on a cluster that consisted of approximately 1800 machines. Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE disks, and a gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

Out of the 4GB of memory, approximately 1-1.5GB was reserved by other tasks running on the cluster. The programs were executed on a weekend afternoon, when the CPUs, disks, and network were mostly idle.

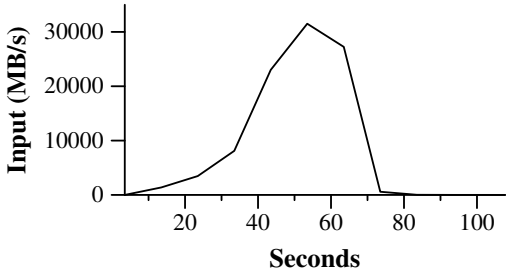


Figure 2: Data transfer rate over time

5.2 Grep

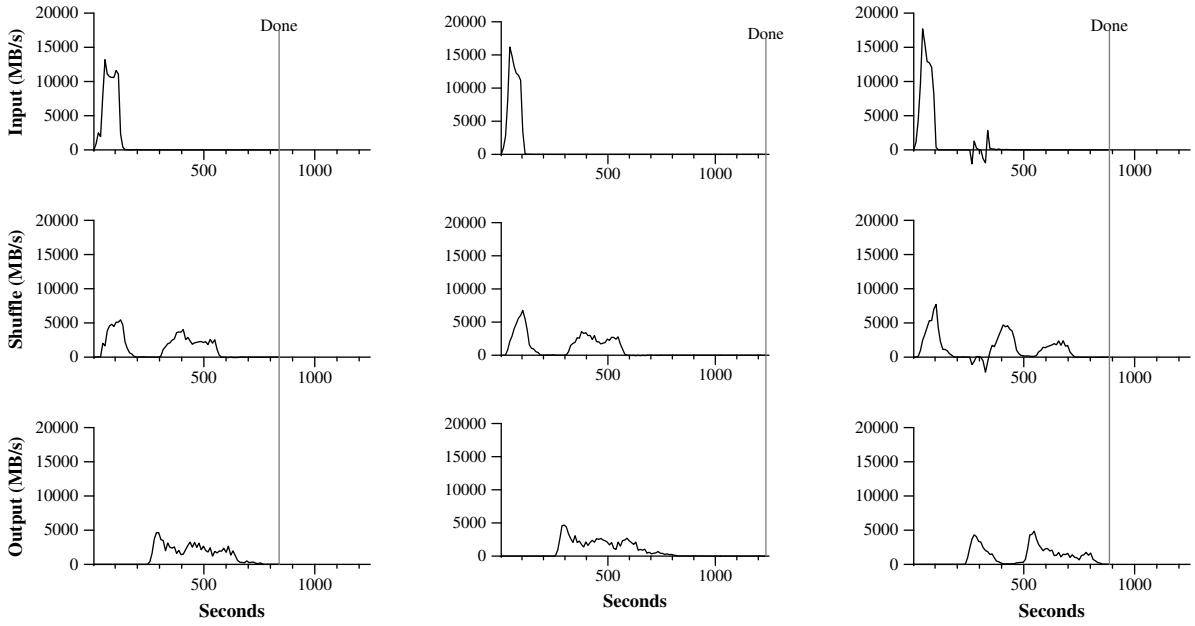
The *grep* program scans through 10^{10} 100-byte records, searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces ($M = 15000$), and the entire output is placed in one file ($R = 1$).

Figure 2 shows the progress of the computation over time. The Y-axis shows the rate at which the input data is scanned. The rate gradually picks up as more machines are assigned to this MapReduce computation, and peaks at over 30 GB/s when 1764 workers have been assigned. As the map tasks finish, the rate starts dropping and hits zero about 80 seconds into the computation. The entire computation takes approximately 150 seconds from start to finish. This includes about a minute of startup overhead. The overhead is due to the propagation of the program to all worker machines, and delays interacting with GFS to open the set of 1000 input files and to get the information needed for the locality optimization.

5.3 Sort

The *sort* program sorts 10^{10} 100-byte records (approximately 1 terabyte of data). This program is modeled after the TeraSort benchmark [10].

The sorting program consists of less than 50 lines of user code. A three-line *Map* function extracts a 10-byte sorting key from a text line and emits the key and the



(a) Normal execution (b) No backup tasks (c) 200 tasks killed

Figure 3: Data transfer rates over time for different executions of the sort program

original text line as the intermediate key/value pair. We used a built-in *Identity* function as the *Reduce* operator. This functions passes the intermediate key/value pair unchanged as the output key/value pair. The final sorted output is written to a set of 2-way replicated GFS files (i.e., 2 terabytes are written as the output of the program).

As before, the input data is split into 64MB pieces ($M = 15000$). We partition the sorted output into 4000 files ($R = 4000$). The partitioning function uses the initial bytes of the key to segregate it into one of R pieces.

Our partitioning function for this benchmark has built-in knowledge of the distribution of keys. In a general sorting program, we would add a pre-pass MapReduce operation that would collect a sample of the keys and use the distribution of the sampled keys to compute split-points for the final sorting pass.

Figure 3 (a) shows the progress of a normal execution of the sort program. The top-left graph shows the rate at which input is read. The rate peaks at about 13 GB/s and dies off fairly quickly since all map tasks finish before 200 seconds have elapsed. Note that the input rate is less than for *grep*. This is because the sort map tasks spend about half their time and I/O bandwidth writing intermediate output to their local disks. The corresponding intermediate output for grep had negligible size.

The middle-left graph shows the rate at which data is sent over the network from the map tasks to the reduce tasks. This shuffling starts as soon as the first map task completes. The first hump in the graph is for the first batch of approximately 1700 reduce tasks (the entire MapReduce was assigned about 1700 machines, and each machine executes at most one reduce task at a time). Roughly 300 seconds into the computation, some of these first batch of reduce tasks finish and we start shuffling data for the remaining reduce tasks. All of the shuffling is done about 600 seconds into the computation.

The bottom-left graph shows the rate at which sorted data is written to the final output files by the reduce tasks. There is a delay between the end of the first shuffling period and the start of the writing period because the machines are busy sorting the intermediate data. The writes continue at a rate of about 2-4 GB/s for a while. All of the writes finish about 850 seconds into the computation. Including startup overhead, the entire computation takes 891 seconds. This is similar to the current best reported result of 1057 seconds for the TeraSort benchmark [18].

A few things to note: the input rate is higher than the shuffle rate and the output rate because of our locality optimization – most data is read from a local disk and bypasses our relatively bandwidth constrained network. The shuffle rate is higher than the output rate because the output phase writes two copies of the sorted data (we make two replicas of the output for reliability and availability reasons). We write two replicas because that is the mechanism for reliability and availability provided by our underlying file system. Network bandwidth requirements for writing data would be reduced if the underlying file system used erasure coding [14] rather than replication.

5.4 Effect of Backup Tasks

In Figure 3 (b), we show an execution of the sort program with backup tasks disabled. The execution flow is similar to that shown in Figure 3 (a), except that there is a very long tail where hardly any write activity occurs. After 960 seconds, all except 5 of the reduce tasks are completed. However these last few stragglers don't finish until 300 seconds later. The entire computation takes 1283 seconds, an increase of 44% in elapsed time.

5.5 Machine Failures

In Figure 3 (c), we show an execution of the sort program where we intentionally killed 200 out of 1746 worker processes several minutes into the computation. The underlying cluster scheduler immediately restarted new worker processes on these machines (since only the processes were killed, the machines were still functioning properly).

The worker deaths show up as a negative input rate since some previously completed map work disappears (since the corresponding map workers were killed) and needs to be redone. The re-execution of this map work happens relatively quickly. The entire computation finishes in 933 seconds including startup overhead (just an increase of 5% over the normal execution time).

6 Experience

We wrote the first version of the MapReduce library in February of 2003, and made significant enhancements to it in August of 2003, including the locality optimization, dynamic load balancing of task execution across worker machines, etc. Since that time, we have been pleasantly surprised at how broadly applicable the MapReduce library has been for the kinds of problems we work on. It has been used across a wide range of domains within Google, including:

- large-scale machine learning problems,
- clustering problems for the Google News and Froogle products,
- extraction of data used to produce reports of popular queries (e.g. Google Zeitgeist),
- extraction of properties of web pages for new experiments and products (e.g. extraction of geographical locations from a large corpus of web pages for localized search), and
- large-scale graph computations.

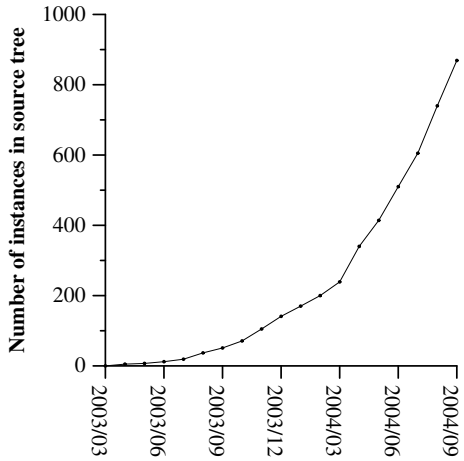


Figure 4: MapReduce instances over time

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426

Table 1: MapReduce jobs run in August 2004

Figure 4 shows the significant growth in the number of separate MapReduce programs checked into our primary source code management system over time, from 0 in early 2003 to almost 900 separate instances as of late September 2004. MapReduce has been so successful because it makes it possible to write a simple program and run it efficiently on a thousand machines in the course of half an hour, greatly speeding up the development and prototyping cycle. Furthermore, it allows programmers who have no experience with distributed and/or parallel systems to exploit large amounts of resources easily.

At the end of each job, the MapReduce library logs statistics about the computational resources used by the job. In Table 1, we show some statistics for a subset of MapReduce jobs run at Google in August 2004.

6.1 Large-Scale Indexing

One of our most significant uses of MapReduce to date has been a complete rewrite of the production index-

ing system that produces the data structures used for the Google web search service. The indexing system takes as input a large set of documents that have been retrieved by our crawling system, stored as a set of GFS files. The raw contents for these documents are more than 20 terabytes of data. The indexing process runs as a sequence of five to ten MapReduce operations. Using MapReduce (instead of the ad-hoc distributed passes in the prior version of the indexing system) has provided several benefits:

- The indexing code is simpler, smaller, and easier to understand, because the code that deals with fault tolerance, distribution and parallelization is hidden within the MapReduce library. For example, the size of one phase of the computation dropped from approximately 3800 lines of C++ code to approximately 700 lines when expressed using MapReduce.
- The performance of the MapReduce library is good enough that we can keep conceptually unrelated computations separate, instead of mixing them together to avoid extra passes over the data. This makes it easy to change the indexing process. For example, one change that took a few months to make in our old indexing system took only a few days to implement in the new system.
- The indexing process has become much easier to operate, because most of the problems caused by machine failures, slow machines, and networking hiccups are dealt with automatically by the MapReduce library without operator intervention. Furthermore, it is easy to improve the performance of the indexing process by adding new machines to the indexing cluster.

7 Related Work

Many systems have provided restricted programming models and used the restrictions to parallelize the computation automatically. For example, an associative function can be computed over all prefixes of an N element array in $\log N$ time on N processors using parallel prefix computations [6, 9, 13]. MapReduce can be considered a simplification and distillation of some of these models based on our experience with large real-world computations. More significantly, we provide a fault-tolerant implementation that scales to thousands of processors. In contrast, most of the parallel processing systems have only been implemented on smaller scales and leave the details of handling machine failures to the programmer.

Bulk Synchronous Programming [17] and some MPI primitives [11] provide higher-level abstractions that make it easier for programmers to write parallel programs. A key difference between these systems and MapReduce is that MapReduce exploits a restricted programming model to parallelize the user program automatically and to provide transparent fault-tolerance.

Our locality optimization draws its inspiration from techniques such as active disks [12, 15], where computation is pushed into processing elements that are close to local disks, to reduce the amount of data sent across I/O subsystems or the network. We run on commodity processors to which a small number of disks are directly connected instead of running directly on disk controller processors, but the general approach is similar.

Our backup task mechanism is similar to the eager scheduling mechanism employed in the Charlotte System [3]. One of the shortcomings of simple eager scheduling is that if a given task causes repeated failures, the entire computation fails to complete. We fix some instances of this problem with our mechanism for skipping bad records.

The MapReduce implementation relies on an in-house cluster management system that is responsible for distributing and running user tasks on a large collection of shared machines. Though not the focus of this paper, the cluster management system is similar in spirit to other systems such as Condor [16].

The sorting facility that is a part of the MapReduce library is similar in operation to NOW-Sort [1]. Source machines (map workers) partition the data to be sorted and send it to one of R reduce workers. Each reduce worker sorts its data locally (in memory if possible). Of course NOW-Sort does not have the user-definable Map and Reduce functions that make our library widely applicable.

River [2] provides a programming model where processes communicate with each other by sending data over distributed queues. Like MapReduce, the River system tries to provide good average case performance even in the presence of non-uniformities introduced by heterogeneous hardware or system perturbations. River achieves this by careful scheduling of disk and network transfers to achieve balanced completion times. MapReduce has a different approach. By restricting the programming model, the MapReduce framework is able to partition the problem into a large number of fine-grained tasks. These tasks are dynamically scheduled on available workers so that faster workers process more tasks. The restricted programming model also allows us to schedule redundant executions of tasks near the end of the job which greatly reduces completion time in the presence of non-uniformities (such as slow or stuck workers).

BAD-FS [5] has a very different programming model from MapReduce, and unlike MapReduce, is targeted to

the execution of jobs across a wide-area network. However, there are two fundamental similarities. (1) Both systems use redundant execution to recover from data loss caused by failures. (2) Both use locality-aware scheduling to reduce the amount of data sent across congested network links.

TACC [7] is a system designed to simplify construction of highly-available networked services. Like MapReduce, it relies on re-execution as a mechanism for implementing fault-tolerance.

8 Conclusions

The MapReduce programming model has been successfully 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 optimization, and load balancing. Second, a large variety of problems are easily expressible as MapReduce computations. For example, MapReduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning, and many other systems. Third, we have developed an implementation of MapReduce that scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources 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 parallelize 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 allows 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 machine failures and data loss.

Acknowledgements

Josh Levenberg has been instrumental in revising and extending the user-level MapReduce API with a number of new features based on his experience with using MapReduce and other people's suggestions for enhancements. MapReduce reads its input from and writes its output to the Google File System [8]. We would like to thank Mohit Aron, Howard Gobioff, Markus Gutschke, David Kramer, Shun-Tak Leung, and Josh Redstone for their work in developing GFS. We would also like to thank Percy Liang and Olcan Sercinoglu for their work in developing the cluster management system used by MapReduce. Mike Burrows, Wilson Hsieh, Josh Levenberg, Sharon Perl, Rob Pike, and Debby Wallach provided helpful comments on earlier drafts of this paper. The anonymous OSDI reviewers, and our shepherd, Eric Brewer, provided many useful suggestions of areas where the paper could be improved. Finally, we thank all the users of MapReduce within Google's engineering organization for providing helpful feedback, suggestions, and bug reports.

References

[1] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, Georgia, May 1999.

[3] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.

[4] Luiz A. Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, April 2003.

[5] John Bent, Douglas Thain, Andrea C.Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI*, March 2004.

[6] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11), November 1989.

[7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Saint-Malo, France, 1997.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *19th Symposium on Operating Systems Principles*, pages 29–43, Lake George, New York, 2003.

[9] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.

[10] Jim Gray. Sort benchmark home page. http://research.microsoft.com/barc/SortBenchmark/.

[11] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[12] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 2004 USENIX File and Storage Technologies FAST Conference*, April 2004.

[13] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[14] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[15] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *IEEE Computer*, pages 68–74, June 2001.

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[17] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1997.

[18] Jim Wyllie. Spsort: How to sort a terabyte quickly. http://alme1.almaden.ibm.com/cs/spsort.pdf.

A Word Frequency

This section contains a program that counts the number of occurrences of each unique word in a set of input files specified on the command line.

```
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
 public:
  virtual void Map(const MapInput& input) {
    const string& text = input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while ((i < n) && isspace(text[i]))
        i++;

      // Find word end
      int start = i;
      while ((i < n) && !isspace(text[i]))
        i++;

      if (start < i)
        Emit(text.substr(start,i-start),"1");
    }
  }
};
REGISTER_MAPPER(WordCounter);

// User's reduce function
class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }

    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};
REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);

  MapReduceSpecification spec;

  // Store list of input files into "spec"
  for (int i = 1; i < argc; i++) {
    MapReduceInput* input = spec.add_input();
    input->set_format("text");
    input->set_filepattern(argv[i]);
    input->set_mapper_class("WordCounter");
  }

  // Specify the output files:
  //    /gfs/test/freq-00000-of-00100
  //    /gfs/test/freq-00001-of-00100
  //    ...
  MapReduceOutput* out = spec.output();
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Adder");

  // Optional: do partial sums within map
  // tasks to save network bandwidth
  out->set_combiner_class("Adder");

  // Tuning parameters: use at most 2000
  // machines and 100 MB of memory per task
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);

  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();

  // Done: 'result' structure contains info
  // about counters, time taken, number of
  // machines used, etc.

  return 0;
}
```

Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

Abstract

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 Finance. 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 simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

1 Introduction

Over the last two and a half years we have designed, implemented, and deployed a distributed storage system for managing structured data at Google called Bigtable. Bigtable is designed to reliably scale to petabytes of data and thousands of machines. Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability. Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth. These products use Bigtable for a variety of demanding workloads, which range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

In many ways, Bigtable resembles a database: it shares many implementation strategies with databases. Parallel databases [14] and main-memory databases [13] have achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names that can be arbitrary strings. Bigtable also treats data as uninterpreted strings, although clients often serialize various forms of structured and semi-structured data into these strings. Clients can control the locality of their data through careful choices in their schemas. Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

Section 2 describes the data model in more detail, and Section 3 provides an overview of the client API. Section 4 briefly describes the underlying Google infrastructure on which Bigtable depends. Section 5 describes the fundamentals of the Bigtable implementation, and Section 6 describes some of the refinements that we made to improve Bigtable's performance. Section 7 provides measurements of Bigtable's performance. We describe several examples of how Bigtable is used at Google in Section 8, and discuss some lessons we learned in designing and supporting Bigtable in Section 9. Finally, Section 10 describes related work, and Section 11 presents our conclusions.

2 Data Model

A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

(row:string, column:string, time:int64) → string

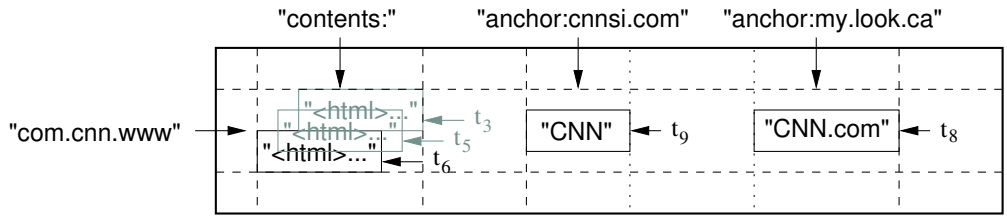


Figure 1: A slice of an example table that stores Web pages. The row name is a reversed URL. The contents column family contains the page contents, and the anchor column family contains the text of any anchors that reference the page. CNN's home page is referenced by both the Sports Illustrated and the MY-look home pages, so the row contains columns named anchor:cnnsi.com and anchor:my.look.ca. Each anchor cell has one version; the contents column has three versions, at timestamps t_3, t_5, and t_6.

We settled on this data model after examining a variety of potential uses of a Bigtable-like system. As one concrete example that drove some of our design decisions, suppose we want to keep a copy of a large collection of web pages and related information that could be used by many different projects; let us call this particular table the *Webtable*. In Webtable, we would use URLs as row keys, various aspects of web pages as column names, and store the contents of the web pages in the contents: column under the timestamps when they were fetched, as illustrated in Figure 1.

Rows

The row keys in a table are arbitrary strings (currently up to 64KB in size, although 10-100 bytes is a typical size for most of our users). Every read or write of data under a single row key is atomic (regardless of the number of different columns being read or written in the row), a design decision that makes it easier for clients to reason about the system's behavior in the presence of concurrent updates to the same row.

Bigtable maintains data in lexicographic order by row key. The row range for a table is dynamically partitioned. Each row range is called a *tablet*, which is the unit of distribution and load balancing. As a result, reads of short row ranges are efficient and typically require communication with only a small number of machines. Clients can exploit this property by selecting their row keys so that they get good locality for their data accesses. For example, in Webtable, pages in the same domain are grouped together into contiguous rows by reversing the hostname components of the URLs. For example, we store data for maps.google.com/index.html under the key com.google.maps/index.html. Storing pages from the same domain near each other makes some host and domain analyses more efficient.

Column Families

Column keys are grouped into sets called *column families*, which form the basic unit of access control. All data stored in a column family is usually of the same type (we compress data in the same column family together). A column family must be created before data can be stored under any column key in that family; after a family has been created, any column key within the family can be used. It is our intent that the number of distinct column families in a table be small (in the hundreds at most), and that families rarely change during operation. In contrast, a table may have an unbounded number of columns.

A column key is named using the following syntax: *family*:*qualifier*. Column family names must be printable, but qualifiers may be arbitrary strings. An example column family for the Webtable is language, which stores the language in which a web page was written. We use only one column key in the language family, and it stores each web page's language ID. Another useful column family for this table is anchor; each column key in this family represents a single anchor, as shown in Figure 1. The qualifier is the name of the referring site; the cell contents is the link text.

Access control and both disk and memory accounting are performed at the column-family level. In our Webtable example, these controls allow us to manage several different types of applications: some that add new base data, some that read the base data and create derived column families, and some that are only allowed to view existing data (and possibly not even to view all of the existing families for privacy reasons).

Timestamps

Each cell in a Bigtable can contain multiple versions of the same data; these versions are indexed by timestamp. Bigtable timestamps are 64-bit integers. They can be assigned by Bigtable, in which case they represent "real time" in microseconds, or be explicitly assigned by client

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

Figure 2: Writing to Bigtable.

```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
  printf("%s %s %lld %s\n",
         scanner.RowName(),
         stream->ColumnName(),
         stream->MicroTimestamp(),
         stream->Value());
}
```

Figure 3: Reading from Bigtable.

applications. Applications that need to avoid collisions must generate unique timestamps themselves. Different versions of a cell are stored in decreasing timestamp order, so that the most recent versions can be read first.

To make the management of versioned data less onerous, we support two per-column-family settings that tell Bigtable to garbage-collect cell versions automatically. The client can specify either that only the last n versions of a cell be kept, or that only new-enough versions be kept (e.g., only keep values that were written in the last seven days).

In our Webtable example, we set the timestamps of the crawled pages stored in the contents: column to the times at which these page versions were actually crawled. The garbage-collection mechanism described above lets us keep only the most recent three versions of every page.

3 API

The Bigtable API provides functions for creating and deleting tables and column families. It also provides functions for changing cluster, table, and column family metadata, such as access control rights.

Client applications can write or delete values in Bigtable, look up values from individual rows, or iterate over a subset of the data in a table. Figure 2 shows C++ code that uses a `RowMutation` abstraction to perform a series of updates. (Irrelevant details were elided to keep the example short.) The call to `Apply` performs an atomic mutation to the Webtable: it adds one anchor to www.cnn.com and deletes a different anchor.

Figure 3 shows C++ code that uses a `Scanner` abstraction to iterate over all anchors in a particular row. Clients can iterate over multiple column families, and there are several mechanisms for limiting the rows, columns, and timestamps produced by a scan. For example, we could restrict the scan above to only produce anchors whose columns match the regular expression anchor:*.cnn.com, or to only produce anchors whose timestamps fall within ten days of the current time.

Bigtable supports several other features that allow the user to manipulate data in more complex ways. First, Bigtable supports single-row transactions, which can be used to perform atomic read-modify-write sequences on data stored under a single row key. Bigtable does not currently support general transactions across row keys, although it provides an interface for batching writes across row keys at the clients. Second, Bigtable allows cells to be used as integer counters. Finally, Bigtable supports the execution of client-supplied scripts in the address spaces of the servers. The scripts are written in a language developed at Google for processing data called Sawzall [28]. At the moment, our Sawzall-based API does not allow client scripts to write back into Bigtable, but it does allow various forms of data transformation, filtering based on arbitrary expressions, and summarization via a variety of operators.

Bigtable can be used with MapReduce [12], a framework for running large-scale parallel computations developed at Google. We have written a set of wrappers that allow a Bigtable to be used both as an input source and as an output target for MapReduce jobs.

4 Building Blocks

Bigtable is built on several other pieces of Google infrastructure. Bigtable uses the distributed Google File System (GFS) [17] to store log and data files. A Bigtable cluster typically operates in a shared pool of machines that run a wide variety of other distributed applications, and Bigtable processes often share the same machines with processes from other applications. Bigtable depends on a cluster management system for scheduling jobs, managing resources on shared machines, dealing with machine failures, and monitoring machine status.

The Google *SSTable* file format is used internally to store Bigtable data. An SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings. Operations are provided to look up the value associated with a specified

key, and to iterate over all key/value pairs in a specified key range. Internally, each SSTable contains a sequence of blocks (typically each block is 64KB in size, but this is configurable). A block index (stored at the end of the SSTable) is used to locate blocks; the index is loaded into memory when the SSTable is opened. A lookup can be performed with a single disk seek: we first find the appropriate block by performing a binary search in the in-memory index, and then reading the appropriate block from disk. Optionally, an SSTable can be completely mapped into memory, which allows us to perform lookups and scans without touching disk.

Bigtable relies on a highly-available and persistent distributed lock service called Chubby [8]. A Chubby service consists of five active replicas, one of which is elected to be the master and actively serve requests. The service is live when a majority of the replicas are running and can communicate with each other. Chubby uses the Paxos algorithm [9, 23] to keep its replicas consistent in the face of failure. Chubby provides a namespace that consists of directories and small files. Each directory or file can be used as a lock, and reads and writes to a file are atomic. The Chubby client library provides consistent caching of Chubby files. Each Chubby client maintains a *session* with a Chubby service. A client's session expires if it is unable to renew its session lease within the lease expiration time. When a client's session expires, it loses any locks and open handles. Chubby clients can also register callbacks on Chubby files and directories for notification of changes or session expiration.

Bigtable uses Chubby for a variety of tasks: to ensure that there is at most one active master at any time; to store the bootstrap location of Bigtable data (see Section 5.1); to discover tablet servers and finalize tablet server deaths (see Section 5.2); to store Bigtable schema information (the column family information for each table); and to store access control lists. If Chubby becomes unavailable for an extended period of time, Bigtable becomes unavailable. We recently measured this effect in 14 Bigtable clusters spanning 11 Chubby instances. The average percentage of Bigtable server hours during which some data stored in Bigtable was not available due to Chubby unavailability (caused by either Chubby outages or network issues) was 0.0047%. The percentage for the single cluster that was most affected by Chubby unavailability was 0.0326%.

5 Implementation

The Bigtable implementation has three major components: a library that is linked into every client, one master server, and many tablet servers. Tablet servers can be dynamically added (or removed) from a cluster to accomodate changes in workloads.

The master is responsible for assigning tablets to tablet servers, detecting the addition and expiration of tablet servers, balancing tablet-server load, and garbage collection of files in GFS. In addition, it handles schema changes such as table and column family creations.

Each tablet server manages a set of tablets (typically we have somewhere between ten to a thousand tablets per tablet server). The tablet server handles read and write requests to the tablets that it has loaded, and also splits tablets that have grown too large.

As with many single-master distributed storage systems [17, 21], client data does not move through the master: clients communicate directly with tablet servers for reads and writes. Because Bigtable clients do not rely on the master for tablet location information, most clients never communicate with the master. As a result, the master is lightly loaded in practice.

A Bigtable cluster stores a number of tables. Each table consists of a set of tablets, and each tablet contains all data associated with a row range. Initially, each table consists of just one tablet. As a table grows, it is automatically split into multiple tablets, each approximately 100-200 MB in size by default.

5.1 Tablet Location

We use a three-level hierarchy analogous to that of a B^+-tree [10] to store tablet location information (Figure 4).

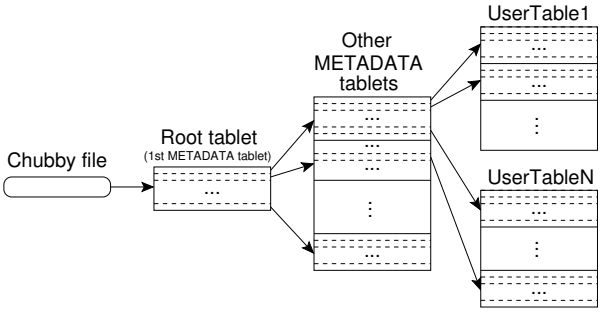


Figure 4: Tablet location hierarchy.

The first level is a file stored in Chubby that contains the location of the *root tablet*. The *root tablet* contains the location of all tablets in a special `METADATA` table. Each `METADATA` tablet contains the location of a set of user tablets. The *root tablet* is just the first tablet in the `METADATA` table, but is treated specially—it is never split—to ensure that the tablet location hierarchy has no more than three levels.

The `METADATA` table stores the location of a tablet under a row key that is an encoding of the tablet's table

identifier and its end row. Each METADATA row stores approximately 1KB of data in memory. With a modest limit of 128 MB METADATA tablets, our three-level location scheme is sufficient to address 2^{34} tablets (or 2^{61} bytes in 128 MB tablets).

The client library caches tablet locations. If the client does not know the location of a tablet, or if it discovers that cached location information is incorrect, then it recursively moves up the tablet location hierarchy. If the client's cache is empty, the location algorithm requires three network round-trips, including one read from Chubby. If the client's cache is stale, the location algorithm could take up to six round-trips, because stale cache entries are only discovered upon misses (assuming that METADATA tablets do not move very frequently). Although tablet locations are stored in memory, so no GFS accesses are required, we further reduce this cost in the common case by having the client library prefetch tablet locations: it reads the metadata for more than one tablet whenever it reads the METADATA table.

We also store secondary information in the METADATA table, including a log of all events pertaining to each tablet (such as when a server begins serving it). This information is helpful for debugging and performance analysis.

5.2 Tablet Assignment

Each tablet is assigned to one tablet server at a time. The master keeps track of the set of live tablet servers, and the current assignment of tablets to tablet servers, including which tablets are unassigned. When a tablet is unassigned, and a tablet server with sufficient room for the tablet is available, the master assigns the tablet by sending a tablet load request to the tablet server.

Bigtable uses Chubby to keep track of tablet servers. When a tablet server starts, it creates, and acquires an exclusive lock on, a uniquely-named file in a specific Chubby directory. The master monitors this directory (the *servers directory*) to discover tablet servers. A tablet server stops serving its tablets if it loses its exclusive lock: e.g., due to a network partition that caused the server to lose its Chubby session. (Chubby provides an efficient mechanism that allows a tablet server to check whether it still holds its lock without incurring network traffic.) A tablet server will attempt to reacquire an exclusive lock on its file as long as the file still exists. If the file no longer exists, then the tablet server will never be able to serve again, so it kills itself. Whenever a tablet server terminates (e.g., because the cluster management system is removing the tablet server's machine from the cluster), it attempts to release its lock so that the master will reassign its tablets more quickly.

The master is responsible for detecting when a tablet server is no longer serving its tablets, and for reassigning those tablets as soon as possible. To detect when a tablet server is no longer serving its tablets, the master periodically asks each tablet server for the status of its lock. If a tablet server reports that it has lost its lock, or if the master was unable to reach a server during its last several attempts, the master attempts to acquire an exclusive lock on the server's file. If the master is able to acquire the lock, then Chubby is live and the tablet server is either dead or having trouble reaching Chubby, so the master ensures that the tablet server can never serve again by deleting its server file. Once a server's file has been deleted, the master can move all the tablets that were previously assigned to that server into the set of unassigned tablets. To ensure that a Bigtable cluster is not vulnerable to networking issues between the master and Chubby, the master kills itself if its Chubby session expires. However, as described above, master failures do not change the assignment of tablets to tablet servers.

When a master is started by the cluster management system, it needs to discover the current tablet assignments before it can change them. The master executes the following steps at startup. (1) The master grabs a unique *master* lock in Chubby, which prevents concurrent master instantiations. (2) The master scans the servers directory in Chubby to find the live servers. (3) The master communicates with every live tablet server to discover what tablets are already assigned to each server. (4) The master scans the METADATA table to learn the set of tablets. Whenever this scan encounters a tablet that is not already assigned, the master adds the tablet to the set of unassigned tablets, which makes the tablet eligible for tablet assignment.

One complication is that the scan of the METADATA table cannot happen until the METADATA tablets have been assigned. Therefore, before starting this scan (step 4), the master adds the root tablet to the set of unassigned tablets if an assignment for the root tablet was not discovered during step 3. This addition ensures that the root tablet will be assigned. Because the root tablet contains the names of all METADATA tablets, the master knows about all of them after it has scanned the root tablet.

The set of existing tablets only changes when a table is created or deleted, two existing tablets are merged to form one larger tablet, or an existing tablet is split into two smaller tablets. The master is able to keep track of these changes because it initiates all but the last. Tablet splits are treated specially since they are initiated by a tablet server. The tablet server commits the split by recording information for the new tablet in the METADATA table. When the split has committed, it notifies the master. In case the split notification is lost (either

because the tablet server or the master died), the master detects the new tablet when it asks a tablet server to load the tablet that has now split. The tablet server will notify the master of the split, because the tablet entry it finds in the METADATA table will specify only a portion of the tablet that the master asked it to load.

5.3 Tablet Serving

The persistent state of a tablet is stored in GFS, as illustrated in Figure 5. Updates are committed to a commit log that stores redo records. Of these updates, the recently committed ones are stored in memory in a sorted buffer called a *memtable*; the older updates are stored in a sequence of SSTables. To recover a tablet, a tablet server

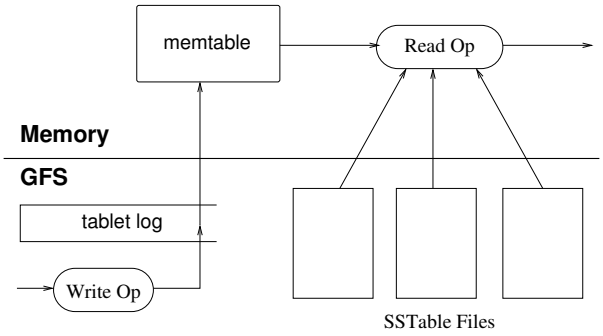


Figure 5: Tablet Representation

reads its metadata from the METADATA table. This metadata contains the list of SSTables that comprise a tablet and a set of a redo points, which are pointers into any commit logs that may contain data for the tablet. The server reads the indices of the SSTables into memory and reconstructs the memtable by applying all of the updates that have committed since the redo points.

When a write operation arrives at a tablet server, the server checks that it is well-formed, and that the sender is authorized to perform the mutation. Authorization is performed by reading the list of permitted writers from a Chubby file (which is almost always a hit in the Chubby client cache). A valid mutation is written to the commit log. Group commit is used to improve the throughput of lots of small mutations [13, 16]. After the write has been committed, its contents are inserted into the memtable.

When a read operation arrives at a tablet server, it is similarly checked for well-formedness and proper authorization. A valid read operation is executed on a merged view of the sequence of SSTables and the memtable. Since the SSTables and the memtable are lexicographically sorted data structures, the merged view can be formed efficiently.

Incoming read and write operations can continue while tablets are split and merged.

5.4 Compactions

As write operations execute, the size of the memtable increases. When the memtable size reaches a threshold, the memtable is frozen, a new memtable is created, and the frozen memtable is converted to an SSTable and written to GFS. This *minor compaction* process has two goals: it shrinks the memory usage of the tablet server, and it reduces the amount of data that has to be read from the commit log during recovery if this server dies. Incoming read and write operations can continue while compactions occur.

Every minor compaction creates a new SSTable. If this behavior continued unchecked, read operations might need to merge updates from an arbitrary number of SSTables. Instead, we bound the number of such files by periodically executing a *merging compaction* in the background. A merging compaction reads the contents of a few SSTables and the memtable, and writes out a new SSTable. The input SSTables and memtable can be discarded as soon as the compaction has finished.

A merging compaction that rewrites all SSTables into exactly one SSTable is called a *major compaction*. SSTables produced by non-major compactions can contain special deletion entries that suppress deleted data in older SSTables that are still live. A major compaction, on the other hand, produces an SSTable that contains no deletion information or deleted data. Bigtable cycles through all of its tablets and regularly applies major compactions to them. These major compactions allow Bigtable to reclaim resources used by deleted data, and also allow it to ensure that deleted data disappears from the system in a timely fashion, which is important for services that store sensitive data.

6 Refinements

The implementation described in the previous section required a number of refinements to achieve the high performance, availability, and reliability required by our users. This section describes portions of the implementation in more detail in order to highlight these refinements.

Locality groups

Clients can group multiple column families together into a *locality group*. A separate SSTable is generated for each locality group in each tablet. Segregating column families that are not typically accessed together into separate locality groups enables more efficient reads. For example, page metadata in Webtable (such as language and checksums) can be in one locality group, and the contents of the page can be in a different group: an ap-

plication that wants to read the metadata does not need to read through all of the page contents.

In addition, some useful tuning parameters can be specified on a per-locality group basis. For example, a locality group can be declared to be in-memory. SSTables for in-memory locality groups are loaded lazily into the memory of the tablet server. Once loaded, column families that belong to such locality groups can be read without accessing the disk. This feature is useful for small pieces of data that are accessed frequently: we use it internally for the location column family in the `METADATA` table.

Compression

Clients can control whether or not the SSTables for a locality group are compressed, and if so, which compression format is used. The user-specified compression format is applied to each SSTable block (whose size is controllable via a locality group specific tuning parameter). Although we lose some space by compressing each block separately, we benefit in that small portions of an SSTable can be read without decompressing the entire file. Many clients use a two-pass custom compression scheme. The first pass uses Bentley and McIlroy's scheme [6], which compresses long common strings across a large window. The second pass uses a fast compression algorithm that looks for repetitions in a small 16 KB window of the data. Both compression passes are very fast—they encode at 100–200 MB/s, and decode at 400–1000 MB/s on modern machines.

Even though we emphasized speed instead of space reduction when choosing our compression algorithms, this two-pass compression scheme does surprisingly well. For example, in Webtable, we use this compression scheme to store Web page contents. In one experiment, we stored a large number of documents in a compressed locality group. For the purposes of the experiment, we limited ourselves to one version of each document instead of storing all versions available to us. The scheme achieved a 10-to-1 reduction in space. This is much better than typical Gzip reductions of 3-to-1 or 4-to-1 on HTML pages because of the way Webtable rows are laid out: all pages from a single host are stored close to each other. This allows the Bentley-McIlroy algorithm to identify large amounts of shared boilerplate in pages from the same host. Many applications, not just Webtable, choose their row names so that similar data ends up clustered, and therefore achieve very good compression ratios. Compression ratios get even better when we store multiple versions of the same value in Bigtable.

Caching for read performance

To improve read performance, tablet servers use two levels of caching. The Scan Cache is a higher-level cache that caches the key-value pairs returned by the SSTable interface to the tablet server code. The Block Cache is a lower-level cache that caches SSTables blocks that were read from GFS. The Scan Cache is most useful for applications that tend to read the same data repeatedly. The Block Cache is useful for applications that tend to read data that is close to the data they recently read (e.g., sequential reads, or random reads of different columns in the same locality group within a hot row).

Bloom filters

As described in Section 5.3, a read operation has to read from all SSTables that make up the state of a tablet. If these SSTables are not in memory, we may end up doing many disk accesses. We reduce the number of accesses by allowing clients to specify that Bloom filters [7] should be created for SSTables in a particular locality group. A Bloom filter allows us to ask whether an SSTable might contain any data for a specified row/column pair. For certain applications, a small amount of tablet server memory used for storing Bloom filters drastically reduces the number of disk seeks required for read operations. Our use of Bloom filters also implies that most lookups for non-existent rows or columns do not need to touch disk.

Commit-log implementation

If we kept the commit log for each tablet in a separate log file, a very large number of files would be written concurrently in GFS. Depending on the underlying file system implementation on each GFS server, these writes could cause a large number of disk seeks to write to the different physical log files. In addition, having separate log files per tablet also reduces the effectiveness of the group commit optimization, since groups would tend to be smaller. To fix these issues, we append mutations to a single commit log per tablet server, co-mingling mutations for different tablets in the same physical log file [18, 20].

Using one log provides significant performance benefits during normal operation, but it complicates recovery. When a tablet server dies, the tablets that it served will be moved to a large number of other tablet servers: each server typically loads a small number of the original server's tablets. To recover the state for a tablet, the new tablet server needs to reapply the mutations for that tablet from the commit log written by the original tablet server. However, the mutations for these tablets

were co-mingled in the same physical log file. One approach would be for each new tablet server to read this full commit log file and apply just the entries needed for the tablets it needs to recover. However, under such a scheme, if 100 machines were each assigned a single tablet from a failed tablet server, then the log file would be read 100 times (once by each server).

We avoid duplicating log reads by first sorting the commit log entries in order of the keys ⟨table, row name, log sequence number⟩. In the sorted output, all mutations for a particular tablet are contiguous and can therefore be read efficiently with one disk seek followed by a sequential read. To parallelize the sorting, we partition the log file into 64 MB segments, and sort each segment in parallel on different tablet servers. This sorting process is coordinated by the master and is initiated when a tablet server indicates that it needs to recover mutations from some commit log file.

Writing commit logs to GFS sometimes causes performance hiccups for a variety of reasons (e.g., a GFS server machine involved in the write crashes, or the network paths traversed to reach the particular set of three GFS servers is suffering network congestion, or is heavily loaded). To protect mutations from GFS latency spikes, each tablet server actually has two log writing threads, each writing to its own log file; only one of these two threads is actively in use at a time. If writes to the active log file are performing poorly, the log file writing is switched to the other thread, and mutations that are in the commit log queue are written by the newly active log writing thread. Log entries contain sequence numbers to allow the recovery process to elide duplicated entries resulting from this log switching process.

Speeding up tablet recovery

If the master moves a tablet from one tablet server to another, the source tablet server first does a minor compaction on that tablet. This compaction reduces recovery time by reducing the amount of uncompacted state in the tablet server's commit log. After finishing this compaction, the tablet server stops serving the tablet. Before it actually unloads the tablet, the tablet server does another (usually very fast) minor compaction to eliminate any remaining uncompacted state in the tablet server's log that arrived while the first minor compaction was being performed. After this second minor compaction is complete, the tablet can be loaded on another tablet server without requiring any recovery of log entries.

Exploiting immutability

Besides the SSTable caches, various other parts of the Bigtable system have been simplified by the fact that all of the SSTables that we generate are immutable. For example, we do not need any synchronization of accesses to the file system when reading from SSTables. As a result, concurrency control over rows can be implemented very efficiently. The only mutable data structure that is accessed by both reads and writes is the memtable. To reduce contention during reads of the memtable, we make each memtable row copy-on-write and allow reads and writes to proceed in parallel.

Since SSTables are immutable, the problem of permanently removing deleted data is transformed to garbage collecting obsolete SSTables. Each tablet's SSTables are registered in the `METADATA` table. The master removes obsolete SSTables as a mark-and-sweep garbage collection [25] over the set of SSTables, where the `METADATA` table contains the set of roots.

Finally, the immutability of SSTables enables us to split tablets quickly. Instead of generating a new set of SSTables for each child tablet, we let the child tablets share the SSTables of the parent tablet.

7 Performance Evaluation

We set up a Bigtable cluster with N tablet servers to measure the performance and scalability of Bigtable as N is varied. The tablet servers were configured to use 1 GB of memory and to write to a GFS cell consisting of 1786 machines with two 400 GB IDE hard drives each. N client machines generated the Bigtable load used for these tests. (We used the same number of clients as tablet servers to ensure that clients were never a bottleneck.) Each machine had two dual-core Opteron 2 GHz chips, enough physical memory to hold the working set of all running processes, and a single gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

The tablet servers and master, test clients, and GFS servers all ran on the same set of machines. Every machine ran a GFS server. Some of the machines also ran either a tablet server, or a client process, or processes from other jobs that were using the pool at the same time as these experiments.

R is the distinct number of Bigtable row keys involved in the test. R was chosen so that each benchmark read or wrote approximately 1 GB of data per tablet server.

The *sequential write* benchmark used row keys with names 0 to $R - 1$. This space of row keys was partitioned into $10N$ equal-sized ranges. These ranges were assigned to the N clients by a central scheduler that as-

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

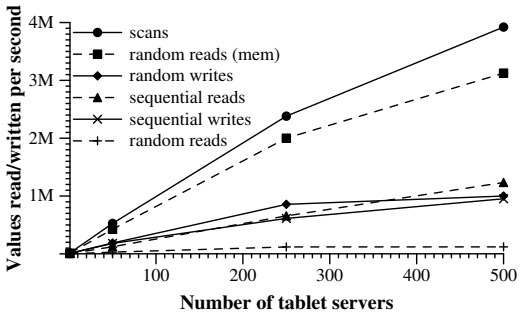


Figure 6: Number of 1000-byte values read/written per second. The table shows the rate per tablet server; the graph shows the aggregate rate.

signed the next available range to a client as soon as the client finished processing the previous range assigned to it. This dynamic assignment helped mitigate the effects of performance variations caused by other processes running on the client machines. We wrote a single string under each row key. Each string was generated randomly and was therefore uncompressible. In addition, strings under different row key were distinct, so no cross-row compression was possible. The *random write* benchmark was similar except that the row key was hashed modulo R immediately before writing so that the write load was spread roughly uniformly across the entire row space for the entire duration of the benchmark.

The *sequential read* benchmark generated row keys in exactly the same way as the sequential write benchmark, but instead of writing under the row key, it read the string stored under the row key (which was written by an earlier invocation of the sequential write benchmark). Similarly, the *random read* benchmark shadowed the operation of the random write benchmark.

The *scan* benchmark is similar to the sequential read benchmark, but uses support provided by the Bigtable API for scanning over all values in a row range. Using a scan reduces the number of RPCs executed by the benchmark since a single RPC fetches a large sequence of values from a tablet server.

The *random reads (mem)* benchmark is similar to the random read benchmark, but the locality group that contains the benchmark data is marked as *in-memory*, and therefore the reads are satisfied from the tablet server's memory instead of requiring a GFS read. For just this benchmark, we reduced the amount of data per tablet server from 1 GB to 100 MB so that it would fit comfortably in the memory available to the tablet server.

Figure 6 shows two views on the performance of our benchmarks when reading and writing 1000-byte values to Bigtable. The table shows the number of operations per second per tablet server; the graph shows the aggregate number of operations per second.

Single tablet-server performance

Let us first consider performance with just one tablet server. Random reads are slower than all other operations by an order of magnitude or more. Each random read involves the transfer of a 64 KB SSTable block over the network from GFS to a tablet server, out of which only a single 1000-byte value is used. The tablet server executes approximately 1200 reads per second, which translates into approximately 75 MB/s of data read from GFS. This bandwidth is enough to saturate the tablet server CPUs because of overheads in our networking stack, SSTable parsing, and Bigtable code, and is also almost enough to saturate the network links used in our system. Most Bigtable applications with this type of an access pattern reduce the block size to a smaller value, typically 8KB.

Random reads from memory are much faster since each 1000-byte read is satisfied from the tablet server's local memory without fetching a large 64 KB block from GFS.

Random and sequential writes perform better than random reads since each tablet server appends all incoming writes to a single commit log and uses group commit to stream these writes efficiently to GFS. There is no significant difference between the performance of random writes and sequential writes; in both cases, all writes to the tablet server are recorded in the same commit log.

Sequential reads perform better than random reads since every 64 KB SSTable block that is fetched from GFS is stored into our block cache, where it is used to serve the next 64 read requests.

Scans are even faster since the tablet server can return a large number of values in response to a single client RPC, and therefore RPC overhead is amortized over a large number of values.

Scaling

Aggregate throughput increases dramatically, by over a factor of a hundred, as we increase the number of tablet servers in the system from 1 to 500. For example, the

# of tablet servers	# of clusters
0 .. 19	259
20 .. 49	47
50 .. 99	20
100 .. 499	50
> 500	12

Table 1: Distribution of number of tablet servers in Bigtable clusters.

performance of random reads from memory increases by almost a factor of 300 as the number of tablet server increases by a factor of 500. This behavior occurs because the bottleneck on performance for this benchmark is the individual tablet server CPU.

However, performance does not increase linearly. For most benchmarks, there is a significant drop in per-server throughput when going from 1 to 50 tablet servers. This drop is caused by imbalance in load in multiple server configurations, often due to other processes contending for CPU and network. Our load balancing algorithm attempts to deal with this imbalance, but cannot do a perfect job for two main reasons: rebalancing is throttled to reduce the number of tablet movements (a tablet is unavailable for a short time, typically less than one second, when it is moved), and the load generated by our benchmarks shifts around as the benchmark progresses.

The random read benchmark shows the worst scaling (an increase in aggregate throughput by only a factor of 100 for a 500-fold increase in number of servers). This behavior occurs because (as explained above) we transfer one large 64KB block over the network for every 1000-byte read. This transfer saturates various shared 1 Gigabit links in our network and as a result, the per-server throughput drops significantly as we increase the number of machines.

8 Real Applications

As of August 2006, there are 388 non-test Bigtable clusters running in various Google machine clusters, with a combined total of about 24,500 tablet servers. Table 1 shows a rough distribution of tablet servers per cluster. Many of these clusters are used for development purposes and therefore are idle for significant periods. One group of 14 busy clusters with 8069 total tablet servers saw an aggregate volume of more than 1.2 million requests per second, with incoming RPC traffic of about 741 MB/s and outgoing RPC traffic of about 16 GB/s.

Table 2 provides some data about a few of the tables currently in use. Some tables store data that is served to users, whereas others store data for batch processing; the tables range widely in total size, average cell size, percentage of data served from memory, and complexity of the table schema. In the rest of this section, we briefly describe how three product teams use Bigtable.

8.1 Google Analytics

Google Analytics (analytics.google.com) is a service that helps webmasters analyze traffic patterns at their web sites. It provides aggregate statistics, such as the number of unique visitors per day and the page views per URL per day, as well as site-tracking reports, such as the percentage of users that made a purchase, given that they earlier viewed a specific page.

To enable the service, webmasters embed a small JavaScript program in their web pages. This program is invoked whenever a page is visited. It records various information about the request in Google Analytics, such as a user identifier and information about the page being fetched. Google Analytics summarizes this data and makes it available to webmasters.

We briefly describe two of the tables used by Google Analytics. The raw click table (˜200 TB) maintains a row for each end-user session. The row name is a tuple containing the website's name and the time at which the session was created. This schema ensures that sessions that visit the same web site are contiguous, and that they are sorted chronologically. This table compresses to 14% of its original size.

The summary table (˜20 TB) contains various predefined summaries for each website. This table is generated from the raw click table by periodically scheduled MapReduce jobs. Each MapReduce job extracts recent session data from the raw click table. The overall system's throughput is limited by the throughput of GFS. This table compresses to 29% of its original size.

8.2 Google Earth

Google operates a collection of services that provide users with access to high-resolution satellite imagery of the world's surface, both through the web-based Google Maps interface (maps.google.com) and through the Google Earth (earth.google.com) custom client software. These products allow users to navigate across the world's surface: they can pan, view, and annotate satellite imagery at many different levels of resolution. This system uses one table to preprocess data, and a different set of tables for serving client data.

The preprocessing pipeline uses one table to store raw imagery. During preprocessing, the imagery is cleaned and consolidated into final serving data. This table contains approximately 70 terabytes of data and therefore is served from disk. The images are efficiently compressed already, so Bigtable compression is disabled.

Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency-sensitive?
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
Google Earth	0.5	64%	8	7	2	33%	Yes
Google Earth	70	–	9	8	3	0%	No
Orkut	9	–	0.9	8	5	1%	Yes
Personalized Search	4	47%	6	93	11	5%	Yes

Table 2: Characteristics of a few tables in production use. *Table size* (measured before compression) and *# Cells* indicate approximate sizes. *Compression ratio* is not given for tables that have compression disabled.

Each row in the imagery table corresponds to a single geographic segment. Rows are named to ensure that adjacent geographic segments are stored near each other. The table contains a column family to keep track of the sources of data for each segment. This column family has a large number of columns: essentially one for each raw data image. Since each segment is only built from a few images, this column family is very sparse.

The preprocessing pipeline relies heavily on MapReduce over Bigtable to transform data. The overall system processes over 1 MB/sec of data per tablet server during some of these MapReduce jobs.

The serving system uses one table to index data stored in GFS. This table is relatively small (˜500 GB), but it must serve tens of thousands of queries per second per datacenter with low latency. As a result, this table is hosted across hundreds of tablet servers and contains in-memory column families.

8.3 Personalized Search

Personalized Search (www.google.com/psearch) is an opt-in service that records user queries and clicks across a variety of Google properties such as web search, images, and news. Users can browse their search histories to revisit their old queries and clicks, and they can ask for personalized search results based on their historical Google usage patterns.

Personalized Search stores each user's data in Bigtable. Each user has a unique userid and is assigned a row named by that userid. All user actions are stored in a table. A separate column family is reserved for each type of action (for example, there is a column family that stores all web queries). Each data element uses as its Bigtable timestamp the time at which the corresponding user action occurred. Personalized Search generates user profiles using a MapReduce over Bigtable. These user profiles are used to personalize live search results.

The Personalized Search data is replicated across several Bigtable clusters to increase availability and to reduce latency due to distance from clients. The Personalized Search team originally built a client-side replication mechanism on top of Bigtable that ensured eventual consistency of all replicas. The current system now uses a replication subsystem that is built into the servers.

The design of the Personalized Search storage system allows other groups to add new per-user information in their own columns, and the system is now used by many other Google properties that need to store per-user configuration options and settings. Sharing a table amongst many groups resulted in an unusually large number of column families. To help support sharing, we added a simple quota mechanism to Bigtable to limit the storage consumption by any particular client in shared tables; this mechanism provides some isolation between the various product groups using this system for per-user information storage.

9 Lessons

In the process of designing, implementing, maintaining, and supporting Bigtable, we gained useful experience and learned several interesting lessons.

One lesson we learned is that large distributed systems are vulnerable to many types of failures, not just the standard network partitions and fail-stop failures assumed in many distributed protocols. For example, we have seen problems due to all of the following causes: memory and network corruption, large clock skew, hung machines, extended and asymmetric network partitions, bugs in other systems that we are using (Chubby for example), overflow of GFS quotas, and planned and unplanned hardware maintenance. As we have gained more experience with these problems, we have addressed them by changing various protocols. For example, we added checksumming to our RPC mechanism. We also handled

some problems by removing assumptions made by one part of the system about another part. For example, we stopped assuming a given Chubby operation could return only one of a fixed set of errors.

Another lesson we learned is that it is important to delay adding new features until it is clear how the new features will be used. For example, we initially planned to support general-purpose transactions in our API. Because we did not have an immediate use for them, however, we did not implement them. Now that we have many real applications running on Bigtable, we have been able to examine their actual needs, and have discovered that most applications require only single-row transactions. Where people have requested distributed transactions, the most important use is for maintaining secondary indices, and we plan to add a specialized mechanism to satisfy this need. The new mechanism will be less general than distributed transactions, but will be more efficient (especially for updates that span hundreds of rows or more) and will also interact better with our scheme for optimistic cross-data-center replication.

A practical lesson that we learned from supporting Bigtable is the importance of proper system-level monitoring (i.e., monitoring both Bigtable itself, as well as the client processes using Bigtable). For example, we extended our RPC system so that for a sample of the RPCs, it keeps a detailed trace of the important actions done on behalf of that RPC. This feature has allowed us to detect and fix many problems such as lock contention on tablet data structures, slow writes to GFS while committing Bigtable mutations, and stuck accesses to the `METADATA` table when `METADATA` tablets are unavailable. Another example of useful monitoring is that every Bigtable cluster is registered in Chubby. This allows us to track down all clusters, discover how big they are, see which versions of our software they are running, how much traffic they are receiving, and whether or not there are any problems such as unexpectedly large latencies.

The most important lesson we learned is the value of simple designs. Given both the size of our system (about 100,000 lines of non-test code), as well as the fact that code evolves over time in unexpected ways, we have found that code and design clarity are of immense help in code maintenance and debugging. One example of this is our tablet-server membership protocol. Our first protocol was simple: the master periodically issued leases to tablet servers, and tablet servers killed themselves if their lease expired. Unfortunately, this protocol reduced availability significantly in the presence of network problems, and was also sensitive to master recovery time. We redesigned the protocol several times until we had a protocol that performed well. However, the resulting protocol was too complex and depended on the behavior of Chubby features that were seldom exercised by other applications. We discovered that we were spending an inordinate amount of time debugging obscure corner cases, not only in Bigtable code, but also in Chubby code. Eventually, we scrapped this protocol and moved to a newer simpler protocol that depends solely on widely-used Chubby features.

10 Related Work

The Boxwood project [24] has components that overlap in some ways with Chubby, GFS, and Bigtable, since it provides for distributed agreement, locking, distributed chunk storage, and distributed B-tree storage. In each case where there is overlap, it appears that the Boxwood's component is targeted at a somewhat lower level than the corresponding Google service. The Boxwood project's goal is to provide infrastructure for building higher-level services such as file systems or databases, while the goal of Bigtable is to directly support client applications that wish to store data.

Many recent projects have tackled the problem of providing distributed storage or higher-level services over wide area networks, often at "Internet scale." This includes work on distributed hash tables that began with projects such as CAN [29], Chord [32], Tapestry [37], and Pastry [30]. These systems address concerns that do not arise for Bigtable, such as highly variable bandwidth, untrusted participants, or frequent reconfiguration; decentralized control and Byzantine fault tolerance are not Bigtable goals.

In terms of the distributed data storage model that one might provide to application developers, we believe the key-value pair model provided by distributed B-trees or distributed hash tables is too limiting. Key-value pairs are a useful building block, but they should not be the only building block one provides to developers. The model we chose is richer than simple key-value pairs, and supports sparse semi-structured data. Nonetheless, it is still simple enough that it lends itself to a very efficient flat-file representation, and it is transparent enough (via locality groups) to allow our users to tune important behaviors of the system.

Several database vendors have developed parallel databases that can store large volumes of data. Oracle's Real Application Cluster database [27] uses shared disks to store data (Bigtable uses GFS) and a distributed lock manager (Bigtable uses Chubby). IBM's DB2 Parallel Edition [4] is based on a shared-nothing [33] architecture similar to Bigtable. Each DB2 server is responsible for a subset of the rows in a table which it stores in a local relational database. Both products provide a complete relational model with transactions.

Bigtable locality groups realize similar compression and disk read performance benefits observed for other systems that organize data on disk using column-based rather than row-based storage, including C-Store [1, 34] and commercial products such as Sybase IQ [15, 36], SenSage [31], KDB+ [22], and the ColumnBM storage layer in MonetDB/X100 [38]. Another system that does vertical and horizontal data partioning into flat files and achieves good data compression ratios is AT&T's Daytona database [19]. Locality groups do not support CPU-cache-level optimizations, such as those described by Ailamaki [2].

The manner in which Bigtable uses memtables and SSTables to store updates to tablets is analogous to the way that the Log-Structured Merge Tree [26] stores updates to index data. In both systems, sorted data is buffered in memory before being written to disk, and reads must merge data from memory and disk.

C-Store and Bigtable share many characteristics: both systems use a shared-nothing architecture and have two different data structures, one for recent writes, and one for storing long-lived data, with a mechanism for moving data from one form to the other. The systems differ significantly in their API: C-Store behaves like a relational database, whereas Bigtable provides a lower level read and write interface and is designed to support many thousands of such operations per second per server. C-Store is also a "read-optimized relational DBMS", whereas Bigtable provides good performance on both read-intensive and write-intensive applications.

Bigtable's load balancer has to solve some of the same kinds of load and memory balancing problems faced by shared-nothing databases (e.g., [11, 35]). Our problem is somewhat simpler: (1) we do not consider the possibility of multiple copies of the same data, possibly in alternate forms due to views or indices; (2) we let the user tell us what data belongs in memory and what data should stay on disk, rather than trying to determine this dynamically; (3) we have no complex queries to execute or optimize.

11 Conclusions

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 implementation before that date. As of August 2006, more than sixty projects are using Bigtable. Our users like the performance and high availability provided by the Bigtable implementation, 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 interesting 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 demonstrates that our design works well in practice.

We are in the process of implementing several additional 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 product groups, so that individual groups do not need to maintain 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 advantages 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 addition, our control over Bigtable's implementation, and the other Google infrastructure upon which Bigtable depends, means that we can remove bottlenecks and inefficiencies as they arise.

Acknowledgements

We thank the anonymous reviewers, David Nagle, and our shepherd Brad Calder, for their feedback on this paper. The Bigtable system has benefited greatly from the feedback of our many users within Google. In addition, we thank the following people for their contributions to Bigtable: Dan Aguayo, Sameer Ajmani, Zhifeng Chen, Bill Coughran, Mike Epstein, Healfdene Goguen, Robert Griesemer, Jeremy Hylton, Josh Hyman, Alex Khesin, Joanna Kulik, Alberto Lerner, Sherry Listgarten, Mike Maloney, Eduardo Pinheiro, Kathy Polizzi, Frank Yellin, and Arthur Zwiegincew.

References

[1] ABADI, D. J., MADDEN, S. R., AND FERREIRA, M. C. Integrating compression and execution in column-oriented database systems. *Proc. of SIGMOD* (2006).

[2] AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND SKOUNAKIS, M. Weaving relations for cache performance. In *The VLDB Journal* (2001), pp. 169–180.

[3] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: A new facility for resource management in server systems. In *Proc. of the 3rd OSDI* (Feb. 1999), pp. 45–58.

[4] BARU, C. K., FECTEAU, G., GOYAL, A., HSIAO, H., JHINGRAN, A., PADMANABHAN, S., COPELAND,

yore

G. P., AND WILSON, W. G. DB2 parallel edition. *IBM Systems Journal 34*, 2 (1995), 292–322.

[5] BAVIER, A., BOWMAN, M., CHUN, B., CULLER, D., KARLIN, S., PETERSON, L., ROSCOE, T., SPALINK, T., AND WAWRZONIAK, M. Operating system support for planetary-scale network services. In *Proc. of the 1st NSDI* (Mar. 2004), pp. 253–266.

[6] BENTLEY, J. L., AND MCILROY, M. D. Data compression using long common strings. In *Data Compression Conference* (1999), pp. 287–295.

[7] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *CACM 13*, 7 (1970), 422–426.

[8] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of the 7th OSDI* (Nov. 2006).

[9] CHANDRA, T., GRIESEMER, R., AND REDSTONE, J. Paxos made live — An engineering perspective. In *Proc. of PODC* (2007).

[10] COMER, D. Ubiquitous B-tree. *Computing Surveys 11*, 2 (June 1979), 121–137.

[11] COPELAND, G. P., ALEXANDER, W., BOUGHTER, E. E., AND KELLER, T. W. Data placement in Bubba. In *Proc. of SIGMOD* (1988), pp. 99–108.

[12] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *Proc. of the 6th OSDI* (Dec. 2004), pp. 137–150.

[13] DEWITT, D., KATZ, R., OLKEN, F., SHAPIRO, L., STONEBRAKER, M., AND WOOD, D. Implementation techniques for main memory database systems. In *Proc. of SIGMOD* (June 1984), pp. 1–8.

[14] DEWITT, D. J., AND GRAY, J. Parallel database systems: The future of high performance database systems. *CACM 35*, 6 (June 1992), 85–98.

[15] FRENCH, C. D. One size fits all database architectures do not work for DSS. In *Proc. of SIGMOD* (May 1995), pp. 449–450.

[16] GAWLICK, D., AND KINKADE, D. Varieties of concurrency control in IMS/VS fast path. *Database Engineering Bulletin 8*, 2 (1985), 3–10.

[17] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proc. of the 19th ACM SOSP* (Dec. 2003), pp. 29–43.

[18] GRAY, J. Notes on database operating systems. In *Operating Systems — An Advanced Course*, vol. 60 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[19] GREER, R. Daytona and the fourth-generation language Cymbal. In *Proc. of SIGMOD* (1999), pp. 525–526.

[20] HAGMANN, R. Reimplementing the Cedar file system using logging and group commit. In *Proc. of the 11th SOSP* (Dec. 1987), pp. 155–162.

[21] HARTMAN, J. H., AND OUSTERHOUT, J. K. The Zebra striped network file system. In *Proc. of the 14th SOSP* (Asheville, NC, 1993), pp. 29–43.

[22] KX.COM. kx.com/products/database.php. Product page.

[23] LAMPORT, L. The part-time parliament. *ACM TOCS 16*, 2 (1998), 133–169.

[24] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. of the 6th OSDI* (Dec. 2004), pp. 105–120.

[25] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine. *CACM 3*, 4 (Apr. 1960), 184–195.

[26] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). *Acta Inf. 33*, 4 (1996), 351–385.

[27] ORACLE.COM. www.oracle.com/technology/products/-database/clustering/index.html. Product page.

[28] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal 13*, 4 (2005), 227–298.

[29] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *Proc. of SIGCOMM* (Aug. 2001), pp. 161–172.

[30] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware 2001* (Nov. 2001), pp. 329–350.

[31] SENSAGE.COM. sensage.com/products-sensage.htm. Product page.

[32] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of SIGCOMM* (Aug. 2001), pp. 149–160.

[33] STONEBRAKER, M. The case for shared nothing. *Database Engineering Bulletin 9*, 1 (Mar. 1986), 4–9.

[34] STONEBRAKER, M., ABADI, D. J., BATKIN, A., CHEN, X., CHERNIACK, M., FERREIRA, M., LAU, E., LIN, A., MADDEN, S., O'NEIL, E., O'NEIL, P., RASIN, A., TRAN, N., AND ZDONIK, S. C-Store: A column-oriented DBMS. In *Proc. of VLDB* (Aug. 2005), pp. 553–564.

[35] STONEBRAKER, M., AOKI, P. M., DEVINE, R., LITWIN, W., AND OLSON, M. A. Mariposa: A new architecture for distributed data. In *Proc. of the Tenth ICDE* (1994), IEEE Computer Society, pp. 54–65.

[36] SYBASE.COM. www.sybase.com/products/database-servers/sybaseiq. Product page.

[37] ZHAO, B. Y., KUBIATOWICZ, J., AND JOSEPH, A. D. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, CS Division, UC Berkeley, Apr. 2001.

[38] ZUKOWSKI, M., BONCZ, P. A., NES, N., AND HEMAN, S. MonetDB/X100 — A DBMS in the CPU cache. *IEEE Data Eng. Bull. 28*, 2 (2005), 17–22.

Google File System 中文版[1]

摘要

我们设计并实现了 Google GFS 文件系统，一个面向大规模数据密集型应用的、可伸缩的分布式文件系统。GFS 虽然运行在廉价的普遍硬件设备上，但是它依然了提供灾难冗余的能力，为大量客户机提供了高性能的服务。

虽然 GFS 的设计目标与许多传统的分布式文件系统有很多相同之处，但是，我们的设计还是以我们对自己的应用的负载情况和技术环境的分析为基础的，不管现在还是将来，GFS 和早期的分布式文件系统的设想都有明显的不同。所以我们重新审视了传统文件系统在设计上的折衷选择，衍生出了完全不同的设计思路。

GFS 完全满足了我们对存储的需求。GFS 作为存储平台已经被广泛的部署在 Google 内部，存储我们的服务产生和处理的数据，同时还用于那些需要大规模数据集的研究和开发工作。目前为止，最大的一个集群利用数千台机器的数千个硬盘，提供了数百 TB 的存储空间，同时为数百个客户机服务。

在本论文中，我们展示了能够支持分布式应用的文件系统接口的扩展，讨论我们设计的许多方面，最后列出了小规模性能测试以及真实生产系统中性能相关数据。

1. 分类和主题描述

D [4]: 3—D 分布文件系统

2. 常用术语

设计，可靠性，性能，测量

3. 关键词

容错，可伸缩性，数据存储，集群存储

1 简介

为了满足 Google 迅速增长的数据处理需求，我们设计并实现了 Google 文件系统(Google File System – GFS)。GFS 与传统的分布式文件系统有着很多相同的设计目标，比如，性能、可伸缩性、可靠性以及可用性。但是，我们的设计还基于我们对我们自己的应用的负载情况和技术环境的观察的影响，不管现在还是将来，GFS 和早期文件系统的假设都有明显的不同。所以我们重新审视了传统文件系统在设计上的折衷选择，衍生出了完全不同的设计思路。

首先，组件失效被认为是常态事件，而不是意外事件。GFS 包括几百甚至几千台普通的廉价设备组装的

[1] 译者 alex，原文地址 http://blademaster.ixiezi.com/

存储机器，同时被相当数量的客户机访问。GFS 组件的数量和质量导致在事实上，任何给定时间内都有可能发生某些组件无法工作，某些组件无法从它们目前的失效状态中恢复。我们遇到过各种各样的问题，比如应用程序 bug、操作系统的 bug、人为失误，甚至还有硬盘、内存、连接器、网络以及电源失效等造成的问题。所以，持续的监控、错误侦测、灾难冗余以及自动恢复的机制必须集成在 GFS 中。

其次，以通常的标准衡量，我们的文件非常巨大。数 GB 的文件非常普遍。每个文件通常都包含许多应用程序对象，比如 web 文档。当我们经常需要处理快速增长的、并且由数亿个对象构成的、数以 TB 的数据集时，采用管理数亿个 KB 大小的小文件的方式是非常不明智的，尽管有些文件系统支持这样的管理方式。因此，设计的假设条件和参数，比如 I/O 操作和 Block 的尺寸都需要重新考虑。

第三，绝大部分文件的修改是采用在文件尾部追加数据，而不是覆盖原有数据的方式。对文件的随机写入操作在实际中几乎不存在。一旦写完之后，对文件的操作就只有读，而且通常是按顺序读。大量的数据符合这些特性，比如：数据分析程序扫描的超大的数据集；正在运行的应用程序生成的连续的数据流；存档的数据；由一台机器生成、另外一台机器处理的中间数据，这些中间数据的处理可能是同时进行的、也可能是后续才处理的。对于这种针对海量文件的访问模式，客户端对数据块缓存是没有意义的，数据的追加操作是性能优化和原子性保证的主要考量因素。

第四，应用程序和文件系统 API 的协同设计提高了整个系统的灵活性。比如，我们放松了对 GFS 一致性模型的要求，这样就减轻了文件系统对应用程序的苛刻要求，大大简化了 GFS 的设计。我们引入了原子性的记录追加操作，从而保证多个客户端能够同时进行追加操作，不需要额外的同步操作来保证数据的一致性。本文后面还有对这些问题的细节的详细讨论。

Google 已经针对不同的应用部署了多套 GFS 集群。最大的一个集群拥有超过 1000 个存储节点，超过 300TB 的硬盘空间，被不同机器上的数百个客户端连续不断的频繁访问。

2 设计概述

2.1 设计预期

在设计满足我们需求的文件系统时候，我们的设计目标既有机会、又有挑战。之前我们已经提到了一些需要关注的关键点，这里我们将设计的预期目标的细节展开讨论。

系统由许多廉价的普通组件组成，组件失效是一种常态。系统必须持续监控自身的状态，它必须将组件失效作为一种常态，能够迅速地侦测、冗余并恢复失效的组件。

系统存储一定数量的大文件。我们预期会有几百万文件，文件的大小通常在 100MB 或者以上。数个 GB 大小的文件也是普遍存在，并且要能够被有效的管理。系统也必须支持小文件，但是不需要针对小文件做专门的优化。

系统的工作负载主要由两种读操作组成：**大规模的流式读取和小规模的随机读取**。大规模的流式读取通常一次读取数百 KB 的数据，更常见的是一次读取 1MB 甚至更多的数据。来自同一个客户机的连续操作通常是读取同一个文件中连续的一个区域。小规模的随机读取通常是在文件某个随机的位置读取几个 KB 数据。如果应用程序对性能非常关注，通常的做法是把小规模的随机读取操作合并并排序，之后按顺序批量读取，这样就避免了在文件中前后来回的移动读取位置。

系统的工作负载还包括许多大规模的、顺序的、数据追加方式的写操作。一般情况下，每次写入的数据的大小和大规模读类似。数据一旦被写入后，文件就很少会被修改了。系统支持小规模的随机位置写入操作，但是可能效率不彰。

系统必须高效的、行为定义明确的[2]实现多客户端并行追加数据到同一个文件里的语意。我们的文件通常被用于“生产者-消费者”队列，或者其它多路文件合并操作。通常会有数百个生产者，每个生产者进程运行在一台机器上，同时对一个文件进行追加操作。使用最小的同步开销来实现的原子的多路追加数据操作是必不可少的。文件可以在稍后读取，或者是消费者在追加的操作的同时读取文件。

高性能的稳定网络带宽远比低延迟重要。我们的目标程序绝大部分要求能够高速率的、大批量的处理数据，极少有程序对单一的读写操作有严格的响应时间要求。

2.2 接口

GFS 提供了一套类似传统文件系统的 API 接口函数，虽然并不是严格按照 POSIX 等标准 API 的形式实现的。文件以分层目录的形式组织，用路径名来标识。我们支持常用的操作，如创建新文件、删除文件、打开文件、关闭文件、读和写文件。

另外，GFS 提供了快照和记录追加操作。快照以很低的成本创建一个文件或者目录树的拷贝。记录追加操作允许多个客户端同时对一个文件进行数据追加操作，同时保证每个客户端的追加操作都是原子性的。这对于实现多路结果合并，以及“生产者-消费者”队列非常有用，多个客户端可以在不需要额外的同步锁定的情况下，同时对一个文件追加数据。我们发现这些类型的文件对于构建大型分布应用是非常重要的。快照和记录追加操作将在 3.4 和 3.3 节分别讨论。

2.3 架构

一个 GFS 集群包含一个单独的 Master 节点[3]、多台 Chunk 服务器，并且同时被多个客户端访问，如图 1 所示。所有的这些机器通常都是普通的 Linux 机器，运行着用户级别(user-level)的服务进程。我们可以很容易

[2] 原文：well-defined

[3] 这里的一个单独的 Master 节点的含义是 GFS 系统中只存在一个逻辑上的 Master 组件。后面我们还会提到 Master 节点复制，因此，为了理解方便，我们把 Master 节点视为一个逻辑上的概念，一个逻辑的 Master 节点包括两台物理主机

的把 Chunk 服务器和客户端都放在同一台机器上，前提是机器资源允许，并且我们能够接受不可靠的应用程序代码带来的稳定性降低的风险。

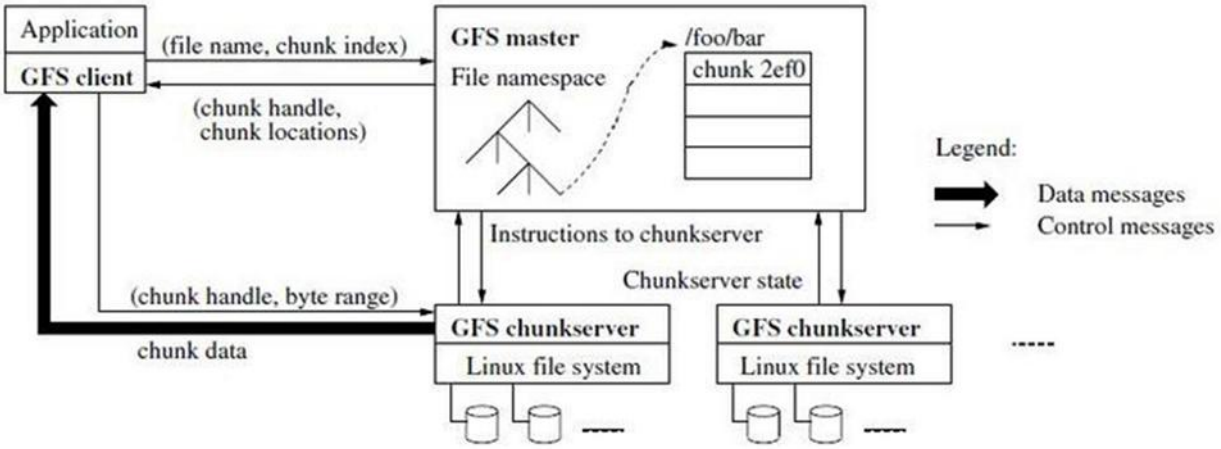


Figure 1: GFS Architecture

GFS 存储的文件都被分割成固定大小的 Chunk。在 Chunk 创建的时候，Master 服务器会给每个 Chunk 分配一个不变的、全球唯一的 64 位的 Chunk 标识。Chunk 服务器把 Chunk 以 Linux 文件的形式保存在本地硬盘上，并且根据指定的 Chunk 标识和字节范围来读写块数据。出于可靠性的考虑，每个块都会复制到多个块服务器上。缺省情况下，我们使用 3 个存储复制节点，不过用户可以为不同的文件命名空间设定不同的复制级别。

Master 节点管理所有的文件系统元数据。这些元数据包括名字空间、访问控制信息、文件和 Chunk 的映射信息、以及当前 Chunk 的位置信息。Master 节点还管理着系统范围内的活动，比如，Chunk 租用管理[4]、孤儿 Chunk[5]的回收、以及 Chunk 在 Chunk 服务器之间的迁移。Master 节点使用心跳信息周期地和每个 Chunk 服务器通讯，发送指令到各个 Chunk 服务器并接收 Chunk 服务器的状态信息。

GFS 客户端代码以库的形式被链接到客户程序里。客户端代码实现了 GFS 文件系统的 API 接口函数、应用程序与 Master 节点和 Chunk 服务器通讯、以及对数据进行读写操作。客户端和 Master 节点的通信只获取元数据，所有的数据操作都是由客户端直接和 Chunk 服务器进行交互的。我们不提供 POSIX 标准的 API 的功能，因此，GFS API 调用不需要深入到 Linux vnode 级别。

无论是客户端还是 Chunk 服务器都不需要缓存文件数据。客户端缓存数据几乎没有什么用处，因为大部分程序要么以流的方式读取一个巨大文件，要么工作集太大根本无法被缓存。无需考虑缓存相关的问题也简化了客户端和整个系统的设计和实现[6]。Chunk 服务器不需要缓存文件数据的原因是，Chunk 以本地文件的方

[4] BDB 也有关于 lease 的描述，不知道是否相同

[5] 原文：orphaned chunks

[6] 注：不过，客户端会缓存元数据。

式保存，Linux 操作系统的文件系统缓存会把经常访问的数据缓存在内存中。

2.4 单一 Master 节点

单一的 Master 节点的策略大大简化了我们的设计。单一的 Master 节点可以通过全局的信息精确定位 Chunk 的位置以及进行复制决策。另外，我们必须减少对 Master 节点的读写，避免 Master 节点成为系统的瓶颈。客户端并不通过 Master 节点读写文件数据。反之，客户端向 Master 节点询问它应该联系的 Chunk 服务器。客户端将这些元数据信息缓存一段时间，后续的操作将直接和 Chunk 服务器进行数据读写操作。

我们利用图 1 解释一下一次简单读取的流程。首先，客户端把文件名和程序指定的字节偏移，根据固定的 Chunk 大小，转换成文件的 Chunk 索引。然后，它把文件名和 Chunk 索引发送给 Master 节点。Master 节点将相应的 Chunk 标识和副本的位置信息发还给客户端。客户端用文件名和 Chunk 索引作为 key 缓存这些信息。

之后客户端发送请求到其中的一个副本处，一般会选择最近的。请求信息包含了 Chunk 的标识和字节范围。在对这个 Chunk 的后续读取操作中，客户端不必再和 Master 节点通讯了，除非缓存的元数据信息过期或者文件被重新打开。实际上，客户端通常会在一次请求中查询多个 Chunk 信息，Master 节点的回应也可能包含了紧跟着这些被请求的 Chunk 后面的 Chunk 的信息。在实际应用中，这些额外的信息在没有任何代价的情况下，避免了客户端和 Master 节点未来可能会发生的几次通讯。

2.5 Chunk 尺寸

Chunk 的大小是关键的设计参数之一。我们选择了 64MB，这个尺寸远远大于一般文件系统的 Block size。每个 Chunk 的副本都以普通 Linux 文件的形式保存在 Chunk 服务器上，只有在需要的时候才扩大。惰性空间分配策略避免了因内部碎片造成的空间浪费，内部碎片或许是对选择这么大的 Chunk 尺寸最具争议一点。

选择较大的 Chunk 尺寸有几个重要的优点。首先，它减少了客户端和 Master 节点通讯的需求，因为只需要一次和 Mater 节点的通信就可以获取 Chunk 的位置信息，之后就可以对同一个 Chunk 进行多次的读写操作。这种方式对降低我们的工作负载来说效果显著，因为我们的应用程序通常是连续读写大文件。即使是小规模的随机读取，采用较大的 Chunk 尺寸也带来明显的好处，客户端可以轻松的缓存一个数 TB 的工作数据集所有的 Chunk 位置信息。其次，采用较大的 Chunk 尺寸，客户端能够对一个块进行多次操作，这样就可以通过与 Chunk 服务器保持较长时间的 TCP 连接来减少网络负载。第三，选用较大的 Chunk 尺寸减少了 Master 节点需要保存的元数据的数量。这就允许我们把元数据全部放在内存中，在 2.6.1 节我们会讨论元数据全部放在内存中带来的额外的好处。

另一方面，即使配合惰性空间分配，采用较大的 Chunk 尺寸也有其缺陷。小文件包含较少的 Chunk，甚至只有一个 Chunk。当有许多的客户端对同一个小文件进行多次的访问时，存储这些 Chunk 的 Chunk 服务器

就会变成热点。在实际应用中，由于我们的程序通常是连续的读取包含多个 Chunk 的大文件，热点还不是主要的问题。

然而，当我们第一次把 GFS 用于批处理队列系统的时候，热点的问题还是产生了：一个可执行文件在 GFS 上保存为 single-chunk 文件，之后这个可执行文件在数百台机器上同时启动。存放这个可执行文件的几个 Chunk 服务器被数百个客户端的并发请求访问导致系统局部过载。我们通过使用更大的复制参数来保存可执行文件，以及错开批处理队列系统程序的启动时间的方法解决了这个问题。一个可能的长效解决方案是，在这种的情况下，允许客户端从其它客户端读取数据。

2.6 元数据

Master 服务器[7]存储 3 种主要类型的元数据，包括：文件和 Chunk 的命名空间、文件和 Chunk 的对应关系、每个 Chunk 副本的存放地点。所有的元数据都保存在 Master 服务器的内存中。前两种类型的元数据[8]同时也会以记录变更日志的方式记录在操作系统的系统日志文件中，日志文件存储在本地磁盘上，同时日志会被复制到其它的远程 Master 服务器上。采用保存变更日志的方式，我们能够简单可靠的更新 Master 服务器的状态，并且不用担心 Master 服务器崩溃导致数据不一致的风险。Master 服务器不会持久保存 Chunk 位置信息。Master 服务器在启动时，或者有新的 Chunk 服务器加入时，向各个 Chunk 服务器轮询它们所存储的 Chunk 的信息。

2.6.1 内存中的数据结构

因为元数据保存在内存中，所以 Master 服务器的操作速度非常快。并且，Master 服务器可以在后台简单而高效的周期性扫描自己保存的全部状态信息。这种周期性的状态扫描也用于实现 Chunk 垃圾收集、在 Chunk 服务器失效的时重新复制数据、通过 Chunk 的迁移实现跨 Chunk 服务器的负载均衡以及磁盘使用状况统计等功能。4.3 和 4.4 章节将深入讨论这些行为。

将元数据全部保存在内存中的方法有潜在问题：Chunk 的数量以及整个系统的承载能力都受限于 Master 服务器所拥有的内存大小。但是在实际应用中，这并不是一个严重的问题。Master 服务器只需要不到 64 个字节的元数据就能够管理一个 64MB 的 Chunk。由于大多数文件都包含多个 Chunk，因此绝大多数 Chunk 都是满的，除了文件的最后一个 Chunk 是部分填充的。同样的，每个文件的在命名空间中的数据大小通常在 64 字节以下，因为保存的文件名是用前缀压缩算法压缩过的。

即便是需要支持更大的文件系统，为 Master 服务器增加额外内存的费用是很少的，而通过增加有限的费用，我们就能够把元数据全部保存在内存里，增强了系统的简洁性、可靠性、高性能和灵活性。

---

[7] 注意逻辑的 Master 节点和物理的 Master 服务器的区别。后续我们谈的是每个 Master 服务器的行为，如存储、内存等等，因此我们将全部使用物理名称

[8] 元数据：命名空间、文件和 Chunk 的对应关系

2.6.2 Chunk 位置信息

Master 服务器并不保存持久化保存哪个 Chunk 服务器存有指定 Chunk 的副本的信息。Master 服务器只是在启动的时候轮询 Chunk 服务器以获取这些信息。Master 服务器能够保证它持有的信息始终是最新的，因为它控制了所有的 Chunk 位置的分配，而且通过周期性的心跳信息监控 Chunk 服务器的状态。

最初设计时，我们试图把 Chunk 的位置信息持久的保存在 Master 服务器上，但是后来我们发现在启动的时候轮询 Chunk 服务器，之后定期轮询更新的方式更简单。这种设计简化了在有 Chunk 服务器加入集群、离开集群、更名、失效、以及重启的时候，Master 服务器和 Chunk 服务器数据同步的问题。在一个拥有数百台服务器的集群中，这类事件会频繁的发生。

可以从另外一个角度去理解这个设计决策：只有 Chunk 服务器才能最终确定一个 Chunk 是否在它的硬盘上。我们从没有考虑过在 Master 服务器上维护一个这些信息的全局视图，因为 Chunk 服务器的错误可能会导致 Chunk 自动消失(比如，硬盘损坏了或者无法访问了)，亦或者操作人员可能会重命名一个 Chunk 服务器。

2.6.3 操作日志

操作日志包含了关键的元数据变更历史记录。这对 GFS 非常重要。这不仅仅是因为操作日志是元数据唯一的持久化存储记录，它也作为判断同步操作顺序的逻辑时间基线[9]。文件和 Chunk，连同它们的版本(参考 4.5 节)，都由它们创建的逻辑时间唯一的、永久的标识。

操作日志非常重要，我们必须确保日志文件的完整，确保只有在元数据的变化被持久化后，日志才对客户端是可见的。否则，即使 Chunk 本身没有出现任何问题，我们仍有可能丢失整个文件系统，或者丢失客户端最近的操作。所以，我们会把日志复制到多台远程机器，并且只有把相应的日志记录写入到本地以及远程机器的硬盘后，才会响应客户端的操作请求。Master 服务器会收集多个日志记录后批量处理，以减少写入磁盘和复制对系统整体性能的影响。

Master 服务器在灾难恢复时，通过重演操作日志把文件系统恢复到最近的状态。为了缩短 Master 启动的时间，我们必须使日志足够小[10]。Master 服务器在日志增长到一定量时对系统状态做一次 Checkpoint[11]，将所有的状态数据写入一个 Checkpoint 文件[12]。在灾难恢复的时候，Master 服务器就通过从磁盘上读取这个 Checkpoint 文件，以及重演 Checkpoint 之后的有限个日志文件就能够恢复系统。Checkpoint 文件以压缩 B-树形势的数据结构存储，可以直接映射到内存，在用于命名空间查询时无需额外的解析。这大大提高了恢复速

[9] 也就是通过逻辑日志的序号作为操作发生的逻辑时间，类似于事务系统中的 LSN

[10] 即重演系统操作的日志量尽量的少

[11] Checkpoint 是一种行为，一种对数据库状态作一次快照的行为

[12] 注：并删除之前的日志文件

度，增强了可用性。

由于创建一个 Checkpoint 文件需要一定的时间，所以 Master 服务器的内部状态被组织为一种格式，这种格式要确保在 Checkpoint 过程中不会阻塞正在进行的修改操作。Master 服务器使用独立的线程切换到新的日志文件和创建新的 Checkpoint 文件。新的 Checkpoint 文件包括切换前所有的修改。对于一个包含数百万个文件的集群，创建一个 Checkpoint 文件需要 1 分钟左右的时间。创建完成后，Checkpoint 文件会被写入在本地和远程的硬盘里。

Master 服务器恢复只需要最新的 Checkpoint 文件和后续的日志文件。旧的 Checkpoint 文件和日志文件可以被删除，但是为了应对灾难性的故障[13]，我们通常会多保存一些历史文件。Checkpoint 失败不会对正确性产生任何影响，因为恢复功能的代码可以检测并跳过没有完成的 Checkpoint 文件。

2.7 一致性模型

GFS 支持一个宽松的一致性模型，这个模型能够很好的支撑我们的高度分布的应用，同时还保持了相对简单且容易实现的优点。本节我们讨论 GFS 的一致性的保障机制，以及对应用程序的意义。我们也着重描述了 GFS 如何管理这些一致性保障机制，但是实现的细节将在本论文的其它部分讨论。

2.7.1 GFS 一致性保障机制

文件命名空间的修改（例如，文件创建）是原子性的。它们仅由 Master 节点的控制：命名空间锁提供了原子性和正确性（4.1 章）的保障；Master 节点的操作日志定义了这些操作在全局的顺序（2.6.3 章）。

	写	记录追加
串行成功	已定义	已定义
并行成功	一致但是未定义	部分不一致
失败	不一致	

表1 操作后的文件状态

数据修改后文件 region[14]的状态取决于操作的类型、成功与否、以及是否同步修改。表 1 总结了各种操作的结果。

如果所有客户端，无论从哪个副本读取，读到的数据都一样，那么我们认为文件 region 是“一致的”；

如果对文件的数据修改之后，region 是一致的，并且客户端能够看到写入操作全部的内容，那么这个 region 是“已定义的”。

当一个数据修改操作成功执行，并且没有受到同时执行的其它写入操作的干扰，那么影响的 region 就是已定义的（隐含了一致性）：所有的客户端都可以看到写入的内容。并行修改操作成功完成之后，region 处于

[13] catastrophes，数据备份相关文档中经常会遇到这个词，表示一种超出预期范围的灾难性事件

[14] region 这个词用中文非常难以表达，我认为应该是修改操作所涉及的文件中的某个范围

一致的、未定义的状态：所有的客户端看到同样的数据，但是无法读到任何一次写入操作写入的数据。通常情况下，文件 region 内包含了来自多个修改操作的、混杂的数据片段。失败的修改操作导致一个 region 处于不一致状态（同时也是未定义的）：不同的客户在不同的时间会看到不同的数据。后面我们将描述应用如何区分已定义和未定义的 region。应用程序没有必要再去细分未定义 region 的不同类型。

数据修改操作分为写入或者记录追加两种。写入操作把数据写在应用程序指定的文件偏移位置上。即使有多个修改操作并行执行时，记录追加操作至少可以把数据原子性的追加到文件中一次，但是偏移位置是由 GFS 选择的（3.3 章）[15]。GFS 返回给客户端一个偏移量，表示了包含了写入记录的、已定义的 region 的起点。另外，GFS 可能会在文件中间插入填充数据或者重复记录。这些数据占据的文件 region 被认定是不一致的，这些数据通常比用户数据小的多。

经过了一系列的成功的修改操作之后，GFS 确保被修改的文件 region 是已定义的，并且包含最后一次修改操作写入的数据。GFS 通过以下措施确保上述行为：（a） 对 Chunk 的所有副本的修改操作顺序一致（3.1 章），（b）使用 Chunk 的版本号来检测副本是否因为它所在的 Chunk 服务器宕机（4.5 章）而错过了修改操作而导致其失效。失效的副本不会再进行任何修改操作，Master 服务器也不再返回这个 Chunk 副本的位置信息给客户端。它们会被垃圾收集系统尽快回收。

由于 Chunk 位置信息会被客户端缓存，所以在信息刷新前，客户端有可能从一个失效的副本读取了数据。在缓存的超时时间和文件下一次被打开的时间之间存在一个时间窗，文件再次被打开后会清除缓存中与该文件有关的所有 Chunk 位置信息。而且，由于我们的文件大多数都是只进行追加操作的，所以，一个失效的副本通常返回一个提前结束的 Chunk 而不是过期的数据。当一个 Reader[16]重新尝试并联络 Master 服务器时，它就会立刻得到最新的 Chunk 位置信息。

即使在修改操作成功执行很长时间之后，组件的失效也可能损坏或者删除数据。GFS 通过 Master 服务器和所有 Chunk 服务器的定期“握手”来找到失效的 Chunk 服务器，并且使用 Checksum 来校验数据是否损坏（5.2 章）。一旦发现问题，数据要尽快利用有效的副本进行恢复（4.3 章）。只有当一个 Chunk 的所有副本在 GFS 检测到错误并采取应对措施之前全部丢失，这个 Chunk 才会不可逆转的丢失。在一般情况下 GFS 的反应时间[17]是几分钟。即使在这种情况下，Chunk 也只是不可用了，而不是损坏了：应用程序会收到明确的错误信息而不是损坏的数据。

---

[15] 这句话有点费解，其含义是所有的追加写入都会成功，但是有可能被执行了多次，而且每次追加的文件偏移量由 GFS 自己计算。相比而言，通常说的追加操作写的偏移位置是文件的尾部

[16] 本文中将用到两个专有名词，Reader 和 Writer，分别表示执行 GFS 读取和写入操作的程序

[17] 指 Master 节点检测到错误并采取应对措施

2.7.2 程序的实现

使用 GFS 的应用程序可以利用一些简单技术实现这个宽松的一致性模型，这些技术也用来实现一些其它的目标功能，包括：尽量采用追加写入而不是覆盖，Checkpoint，自验证的写入操作，自标识的记录。

在实际应用中，我们所有的应用程序对文件的写入操作都是尽量采用数据追加方式，而不是覆盖方式。一种典型的应用，应用程序从头到尾写入数据，生成了一个文件。写入所有数据之后，应用程序自动将文件改名为一个永久保存的文件名，或者周期性的作 Checkpoint，记录成功写入了多少数据。Checkpoint 文件可以包含程序级别的校验和。Readers 仅校验并处理上个 Checkpoint 之后产生的文件 region，这些文件 region 的状态一定是已定义的。这个方法满足了我们一致性和并发处理的要求。追加写入比随机位置写入更加有效率，对应用程序的失败处理更具有弹性。Checkpoint 可以让 Writer 以渐进的方式重新开始，并且可以防止 Reader 处理已经被成功写入，但是从应用程序的角度来看还并未完成的数据。

我们再来分析另一种典型的应用。许多应用程序并行的追加数据到同一个文件，比如进行结果的合并或者是一个生产者-消费者队列。记录追加方式的“至少一次追加”的特性保证了 Writer 的输出。Readers 使用下面的方法来处理偶然性的填充数据和重复内容。Writers 在每条写入的记录中都包含了额外的信息，例如 Checksum，用来验证它的有效性。Reader 可以利用 Checksum 识别和抛弃额外的填充数据和记录片段。如果应用不能容忍偶尔的重复内容(比如，如果这些重复数据触发了非幂等操作)，可以用记录的唯一标识符来过滤它们，这些唯一标识符通常用于命名程序中处理的实体对象，例如 web 文档。这些记录 I/O 功能[18]都包含在我们的程序共享的库中，并且适用于 Google 内部的其它的文件接口实现。所以，相同序列的记录，加上一些偶尔出现的重复数据，都被分发到 Reader 了。

3 系统交互

我们在设计这个系统时，一个重要的原则是最小化所有操作和 Master 节点的交互。

带着这样的设计理念，我们现在描述一下客户机、Master 服务器和 Chunk 服务器如何进行交互，以实现数据修改操作、原子的记录追加操作以及快照功能。

3.1 租约（lease）[19]和变更顺序

变更是一个会改变 Chunk 内容或者元数据的操作，比如写入操作或者记录追加操作。变更操作会在 Chunk 的所有副本上执行。我们使用租约（lease）机制来保持多个副本间变更顺序的一致性。Master 节点为 Chunk

[18] These functionalities for record I/O，除了剔除重复数据的功能

[19] lease 是数据库中的一个术语

的一个副本建立一个租约，我们把这个副本叫做主 Chunk。主 Chunk 对 Chunk 的所有更改操作进行序列化。所有的副本都遵从这个序列进行修改操作。因此，修改操作全局的顺序首先由 Master 节点选择的租约的顺序决定，然后由租约中主 Chunk 分配的序列号决定。

设计租约机制的目的是为了最小化 Master 节点的管理负担。租约的初始超时设置为 60 秒。不过，只要 Chunk 被修改了，主 Chunk 就可以申请更长的租期，通常会得到 Master 节点的确认并收到租约延长的时间。这些租约延长请求和批准的信息通常都是附加在 Master 节点和 Chunk 服务器之间的心跳消息中来传递。有时 Master 节点会试图提前取消租约（例如，Master 节点想取消在一个已经被改名的文件上的修改操作）。即使 Master 节点和主 Chunk 失去联系，它仍然可以安全地在旧的租约到期后和另外一个 Chunk 副本签订新的租约。

在图 2 中，我们依据步骤编号，展现写入操作的控制流程。

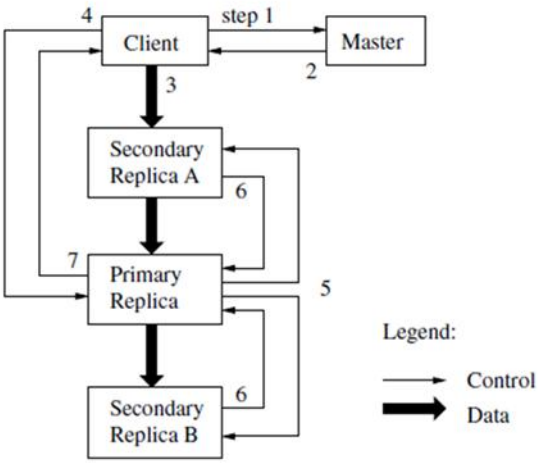


Figure 2: Write Control and Data Flow

客户机向 Master 节点询问哪一个 Chunk 服务器持有当前的租约，以及其它副本的位置。如果没有一个 Chunk 持有租约，Master 节点就选择其中一个副本建立一个租约（这个步骤在图上没有显示）。

Master 节点将主 Chunk 的标识符以及其它副本（又称为 secondary 副本、二级副本）的位置返回给客户机。客户机缓存这些数据以便后续的操作。只有在主 Chunk 不可用，或者主 Chunk 回复信息表明它已不再持有租约的时候，客户机才需要重新跟 Master 节点联系。

客户机把数据推送到所有的副本上。客户机可以以任意的顺序推送数据。Chunk 服务器接收到数据并保存在它的内部 LRU 缓存中，一直到数据被使用或者过期交换出去。由于数据流的网络传输负载非常高，通过分离数据流和控制流，我们可以基于网络拓扑情况对数据流进行规划，提高系统性能，而不用去理会哪个 Chunk 服务器保存了主 Chunk。3.2 章节会进一步讨论这点。

当所有的副本都确认接收到了数据，客户机发送写请求到主 Chunk 服务器。这个请求标识了早前推送到所有副本的数据。主 Chunk 为接收到的所有操作分配连续的序列号，这些操作可能来自不同的客户机，序列号保证了操作顺序执行。它以序列号的顺序把操作应用到它自己的本地状态中（alex 注：也就是在本地执行

这些操作，这句话按字面翻译有点费解，也许应该翻译为“它顺序执行这些操作，并更新自己的状态”)。

主 Chunk 把写请求传递到所有的二级副本。每个二级副本依照主 Chunk 分配的序列号以相同的顺序执行这些操作。

所有的二级副本回复主 Chunk，它们已经完成了操作。

主 Chunk 服务器[20]回复客户机。任何副本产生的任何错误都会返回给客户机。在出现错误的情况下，写入操作可能在主 Chunk 和一些二级副本执行成功。（如果操作在主 Chunk 上失败了，操作就不会被分配序列号，也不会被传递。）客户端的请求被确认为失败，被修改的 region 处于不一致的状态。我们的客户机代码通过重复执行失败的操作来处理这样的错误。在从头开始重复执行之前，客户机会先从步骤（3）到步骤（7）做几次尝试。

如果应用程序一次写入的数据量很大，或者数据跨越了多个 Chunk，GFS 客户机代码会把它们分成多个写操作。这些操作都遵循前面描述的控制流程，但是可能会被其它客户机上同时进行的操作打断或者覆盖。因此，共享的文件 region 的尾部可能包含来自不同客户机的数据片段，尽管如此，由于这些分解后的写入操作在所有的副本上都以相同的顺序执行完成，Chunk 的所有副本都是一致的。这使文件 region 处于 2.7 节描述的一致的、但是未定义的状态。

3.2 数据流

为了提高网络效率，我们采取了把数据流和控制流分开的措施。在控制流从客户机到主 Chunk、然后再到所有二级副本的同时，数据以管道的方式，顺序的沿着一个精心选择的 Chunk 服务器链推送。我们的目标是充分利用每台机器的带宽，避免网络瓶颈和高延时的连接，最小化推送所有数据的延时。

为了充分利用每台机器的带宽，数据沿着一个 Chunk 服务器链顺序的推送，而不是以其它拓扑形式分散推送（例如，树型拓扑结构）。线性推送模式下，每台机器所有的出口带宽都用于以最快的速度传输数据，而不是在多个接受者之间分配带宽。

为了尽可能的避免出现网络瓶颈和高延迟的链接（eg，inter-switch 最有可能出现类似问题），每台机器都尽量的在网络拓扑中选择一台还没有接收到数据的、离自己最近的机器作为目标推送数据。假设客户机把数据从 Chunk 服务器 S1 推送到 S4。它把数据推送到最近的 Chunk 服务器 S1。S1 把数据推送到 S2，因为 S2 和 S4 中最接近的机器是 S2。同样的，S2 把数据传递给 S3 和 S4 之间更近的机器，依次类推推送下去。我们的网络拓扑非常简单，通过 IP 地址就可以计算出节点的“距离”。

最后，我们利用基于 TCP 连接的、管道式数据推送方式来最小化延迟。Chunk 服务器接收到数据后，马上开始向前推送。管道方式的数据推送对我们帮助很大，因为我们采用全双工的交换网络。接收到数据后

[20] 即主 Chunk 所在的 Chunk 服务器

立刻向前推送不会降低接收的速度。在没有网络拥塞的情况下，传送 B 字节的数据到 R 个副本的理想时间是 B/T+RL ，T 是网络的吞吐量，L 是在两台机器数据传输的延迟。通常情况下，我们的网络连接速度是 100Mbps（T），L 将远小于 1ms。因此，1MB 的数据在理想情况下 80ms 左右就能分发出去。

3.3 原子的记录追加

GFS 提供了一种原子的数据追加操作－记录追加。传统方式的写入操作，客户程序会指定数据写入的偏移量。对同一个 region 的并行写入操作不是串行的：region 尾部可能会包含多个不同客户机写入的数据片段。使用记录追加，客户机只需要指定要写入的数据。GFS 保证至少有一次原子的写入操作成功执行（即写入一个顺序的 byte 流），写入的数据追加到 GFS 指定的偏移位置上，之后 GFS 返回这个偏移量给客户机。这类似于在 Unix 操作系统编程环境中，对以 O_APPEND 模式打开的文件，多个并发写操作在没有竞态条件时的行为。

记录追加在我们的分布应用中非常频繁的使用，在这些分布式应用中，通常有很多的客户机并行地对同一个文件追加写入数据。如果我们采用传统方式的文件写入操作，客户机需要额外的复杂、昂贵的同步机制，例如使用一个分布式的锁管理器。在我们的工作中，这样的文件通常用于多个生产者/单一消费者的队列系统，或者是合并了来自多个客户机的数据的结果文件。

记录追加是一种修改操作，它也遵循 3.1 节描述的控制流程，除了在主 Chunk 有些额外的控制逻辑。客户机把数据推送给文件最后一个 Chunk 的所有副本，之后发送请求给主 Chunk。主 Chunk 会检查这次记录追加操作是否会使 Chunk 超过最大尺寸（64MB）。如果超过了最大尺寸，主 Chunk 首先将当前 Chunk 填充到最大尺寸，之后通知所有二级副本做同样的操作，然后回复客户机要求其对下一个 Chunk 重新进行记录追加操作。（记录追加的数据大小严格控制在 Chunk 最大尺寸的 1/4，这样即使在最坏情况下，数据碎片的数量仍然在可控的范围。）通常情况下追加的记录不超过 Chunk 的最大尺寸，主 Chunk 把数据追加到自己的副本内，然后通知二级副本把数据写在跟主 Chunk 一样的位置上，最后回复客户机操作成功。

如果记录追加操作在任何一个副本上失败了，客户端就需要重新进行操作。重新进行记录追加的结果是，同一个 Chunk 的不同副本可能包含不同的数据－重复包含一个记录全部或者部分的数据。GFS 并不保证 Chunk 的所有副本在字节级别是完全一致的。它只保证数据作为一个整体原子的被至少写入一次。这个特性可以通过简单观察推导出来：如果操作成功执行，数据一定已经写入到 Chunk 的所有副本的相同偏移位置上。这之后，所有的副本至少都到了记录尾部的长度，任何后续的记录都会追加到更大的偏移地址，或者是不同的 Chunk 上，即使其它的 Chunk 副本被 Master 节点选为了主 Chunk。就我们的一致性保障模型而言，记录追加操作成功写入数据的 region 是已定义的（因此也是一致的），反之则是不一致的（因此也就是未定义的）。正

如我们在 2.7.2 节讨论的，我们的程序可以处理不一致的区域。

3.4 快照[21]

快照操作几乎可以瞬间完成对一个文件或者目录树（“源”）做一个拷贝，并且几乎不会对正在进行的其它操作造成任何干扰。我们的用户可以使用快照迅速的创建一个巨大的数据集的分支拷贝（而且经常是递归的拷贝拷贝），或者是在做实验性的数据操作之前，使用快照操作备份当前状态，这样之后就可以轻松的提交或者回滚到备份时的状态。

就像 AFS（alex 注：AFS，即 Andrew File System，一种分布式文件系统），我们用标准的 copy-on-write 技术实现快照。当 Master 节点收到一个快照请求，它首先取消作快照的文件的所有 Chunk 的租约。这个措施保证了后续对这些 Chunk 的写操作都必须与 Master 交互以找到租约持有者。这就给 Master 节点一个率先创建 Chunk 的新拷贝的机会。

租约取消或者过期之后，Master 节点把这个操作以日志的方式记录到硬盘上。然后，Master 节点通过复制源文件或者目录的元数据的方式，把这条日志记录的变化反映到保存在内存的状态中。新创建的快照文件和源文件指向完全相同的 Chunk 地址。

在快照操作之后，当客户机第一次想写入数据到 Chunk C，它首先会发送一个请求到 Master 节点查询当前的租约持有者。Master 节点注意到 Chunk C 的引用计数超过了 1[22]。Master 节点不会马上回复客户机的请求，而是选择一个新的 Chunk 句柄 C`。之后，Master 节点要求每个拥有 Chunk C 当前副本的 Chunk 服务器创建一个叫做 C`的新 Chunk。通过在源 Chunk 所在 Chunk 服务器上创建新的 Chunk，我们确保数据在本地而不是通过网络复制（我们的硬盘比我们的 100Mb 以太网大约快 3 倍）。从这点来讲，请求的处理方式和任何其它 Chunk 没什么不同：Master 节点确保新 Chunk C`的一个副本拥有租约，之后回复客户机，客户机得到回复后就可以正常的写这个 Chunk，而不必理会它是从一个已存在的 Chunk 克隆出来的。

4 Master 节点的操作

Master 节点执行所有的名称空间操作。此外，它还管理着整个系统里所有 Chunk 的副本：它决定 Chunk 的存储位置，创建新 Chunk 和它的副本，协调各种各样的系统活动以保证 Chunk 被完全复制，在所有的 Chunk 服务器之间的进行负载均衡，回收不再使用的存储空间。本节我们讨论上述的主题。

---

[21] 这一节非常难以理解，总的来说依次讲述了什么是快照、快照使用的 COW 技术、快照如何不干扰当前操作

[22] 不太明白为什么会大于 1.难道是 Snapshot 没有释放引用计数？

4.1 名称空间管理和锁

Master 节点的很多操作会花费很长的时间：比如，快照操作必须取消 Chunk 服务器上快照所涉及的所有的 Chunk 的租约。我们不希望在这些操作的运行时，延缓了其它的 Master 节点的操作。因此，我们允许多个操作同时进行，使用名称空间的 region 上的锁来保证执行的正确顺序。

不同于许多传统文件系统，GFS 没有针对每个目录实现能够列出目录下所有文件的数据结构。GFS 也不支持文件或者目录的链接（即 Unix 术语中的硬链接或者符号链接）。在逻辑上，GFS 的名称空间就是一个全路径和元数据映射关系的查找表。利用前缀压缩，这个表可以高效的存储在内存中。在存储名称空间的树型结构上，每个节点（绝对路径的文件名或绝对路径的目录名）都有一个关联的读写锁。

每个 Master 节点的操作在开始之前都要获得一系列的锁。通常情况下，如果一个操作涉及/d1/d2/…/dn/leaf，那么操作首先要获得目录/d1，/d1/d2，…，/d1/d2/…/dn 的读锁，以及/d1/d2/…/dn/leaf 的读写锁。注意，根据操作的不同，leaf 可以是一个文件，也可以是一个目录。

现在，我们演示一下在/home/user 被快照到/save/user 的时候，锁机制如何防止创建文件/home/user/foo。快照操作获取/home 和/save 的读取锁，以及/home/user 和/save/user 的写入锁。文件创建操作获得/home 和/home/user 的读取锁，以及/home/user/foo 的写入锁。这两个操作要顺序执行，因为它们试图获取的/home/user 的锁是相互冲突。文件创建操作不需要获取父目录的写入锁，因为这里没有“目录”，或者类似 inode 等用来禁止修改的数据结构。文件名的读取锁足以防止父目录被删除。

采用这种锁方案的优点是支持对同一目录的并行操作。比如，可以再同一个目录下同时创建多个文件：每一个操作都获取一个目录名的上的读取锁和文件名上的写入锁。目录名的读取锁足以的防止目录被删除、改名以及被快照。文件名的写入锁序列化文件创建操作，确保不会多次创建同名的文件。

因为名称空间可能有很多节点，读写锁采用惰性分配策略，在不再使用的时候立刻被删除。同样，锁的获取也要依据一个全局一致的顺序来避免死锁：首先按名称空间的层次排序，在同一个层次内按字典顺序排序。

4.2 副本的位置

GFS 集群是高度分布的多层布局结构，而不是平面结构。典型的拓扑结构是有数百个 Chunk 服务器安装在许多机架上。Chunk 服务器被来自同一或者不同机架上的数百个客户机轮流访问。不同机架上的两台机器间的通讯可能跨越一个或多个网络交换机。另外，机架的出入带宽可能比机架内所有机器加和在一起的带宽要小。多层分布架构对数据的灵活性、可靠性以及可用性方面提出特有的挑战。

Chunk 副本位置选择的策略服务两大目标：最大化数据可靠性和可用性，最大化网络带宽利用率。为了实现这两个目的，仅仅是在多台机器上分别存储这些副本是不够的，这只能预防硬盘损坏或者机器失效带来

的影响，以及最大化每台机器的网络带宽利用率。我们必须在多个机架间分布储存 Chunk 的副本。这保证 Chunk 的一些副本在整个机架被破坏或掉线（比如，共享资源，如电源或者网络交换机造成的问题）的情况下依然存在且保持可用状态。这还意味着在网络流量方面，尤其是针对 Chunk 的读操作，能够有效利用多个机架的整合带宽。另一方面，写操作必须和多个机架上的设备进行网络通信，但是这个代价是我们愿意付出的。

4.3 创建，重新复制，重新负载均衡

Chunk 的副本有三个用途：Chunk 创建，重新复制和重新负载均衡。

当 Master 节点创建一个 Chunk 时，它会选择在哪里放置初始的空的副本。Master 节点会考虑几个因素

（1）我们希望在低于平均硬盘使用率的 Chunk 服务器上存储新的副本。这样的做法最终能够平衡 Chunk 服务器之间的硬盘使用率。

（2）我们希望限制在每个 Chunk 服务器上“最近”的 Chunk 创建操作的次数。虽然创建操作本身是廉价的，但是创建操作也意味着随之会有大量的写入数据的操作，因为 Chunk 在 Writer 真正写入数据的时候才被创建，而在我们的“追加一次，读取多次”的工作模式下，Chunk 一旦写入成功之后就会变为只读的了。

（3）如上所述，我们希望把 Chunk 的副本分布在多个机架之间。

当 Chunk 的有效副本数量少于用户指定的复制因数的时候，Master 节点会重新复制它。这可能是由几个原因引起的：一个 Chunk 服务器不可用了，Chunk 服务器报告它所存储的一个副本损坏了，Chunk 服务器的一个磁盘因为错误不可用了，或者 Chunk 副本的复制因数提高了。每个需要被重新复制的 Chunk 都会根据几个因素进行排序。一个因素是 Chunk 现有副本数量和复制因数相差多少。例如，丢失两个副本的 Chunk 比丢失一个副本的 Chunk 有更高的优先级。另外，我们优先重新复制活跃（live）文件的 Chunk 而不是最近刚被删除的文件的 Chunk（查看 4.4 节）。最后，为了最小化失效的 Chunk 对正在运行的应用程序的影响，我们提高会阻塞客户机程序处理流程的 Chunk 的优先级。

Master 节点选择优先级最高的 Chunk，然后命令某个 Chunk 服务器直接从可用的副本”克隆”一个副本出来。选择新副本的位置的策略和创建时类似：平衡硬盘使用率、限制同一台 Chunk 服务器上的正在进行的克隆操作的数量、在机架间分布副本。为了防止克隆产生的网络流量大大超过客户机的流量，Master 节点对整个集群和每个 Chunk 服务器上的同时进行的克隆操作的数量都进行了限制。另外，Chunk 服务器通过调节它对源 Chunk 服务器读请求的频率来限制它用于克隆操作的带宽。

最后，Master 服务器周期性地对副本进行重新负载均衡：它检查当前的副本分布情况，然后移动副本以便更好的利用硬盘空间、更有效的进行负载均衡。而且在这个过程中，Master 服务器逐渐的填满一个新的 Chunk 服务器，而不是在短时间内用新的 Chunk 填满它，以至于过载。新副本的存储位置选择策略和上面讨论的相同。另外，Master 节点必须选择哪个副本要被移走。通常情况，Master 节点移走那些剩余空间低于平均值的

Chunk 服务器上的副本，从而平衡系统整体的硬盘使用率。

4.4 垃圾回收

GFS 在文件删除后不会立刻回收可用的物理空间。GFS 空间回收采用惰性的策略，只在文件和 Chunk 级的常规垃圾收集时进行。我们发现这个方法使系统更简单、更可靠。

4.4.1 机制

当一个文件被应用程序删除时，Master 节点象对待其它修改操作一样，立刻把删除操作以日志的方式记录下来。但是，Master 节点并不马上回收资源，而是把文件名改为一个包含删除时间戳的、隐藏的名字。当 Master 节点对文件系统命名空间做常规扫描的时候，它会删除所有三天前的隐藏文件（这个时间间隔是可以设置的）。直到文件被真正删除，它们仍旧可以用新的特殊的名字读取，也可以通过把隐藏文件改名为正常显示的文件名的方式“反删除”。当隐藏文件被从名称空间中删除，Master 服务器内存中保存的这个文件的相关元数据才会被删除。这也有效的切断了文件和它包含的所有 Chunk 的连接[23]。

在对 Chunk 名字空间做类似的常规扫描时，Master 节点找到孤儿 Chunk（不被任何文件包含的 Chunk）并删除它们的元数据。Chunk 服务器在和 Master 节点交互的心跳信息中，报告它拥有的 Chunk 子集的信息，Master 节点回复 Chunk 服务器哪些 Chunk 在 Master 节点保存的元数据中已经不存在了。Chunk 服务器可以任意删除这些 Chunk 的副本。

4.4.2 讨论

虽然分布式垃圾回收在编程语言领域是一个需要复杂的方案才能解决的难题，但是在 GFS 系统中是非常简单的。我们可以轻易的得到 Chunk 的所有引用：它们都只存储在 Master 服务器上的文件到块的映射表中。我们也可以很轻易的得到所有 Chunk 的副本：它们都以 Linux 文件的形式存储在 Chunk 服务器的指定目录下。所有 Master 节点不能识别的副本都是“垃圾”。

垃圾回收在空间回收方面相比直接删除有几个优势。首先，对于组件失效是常态的大规模分布式系统，垃圾回收方式简单可靠。Chunk 可能在某些 Chunk 服务器创建成功，某些 Chunk 服务器上创建失败，失败的副本处于无法被 Master 节点识别的状态。副本删除消息可能丢失，Master 节点必须重新发送失败的删除消息，包括自身的和 Chunk 服务器的[24]。垃圾回收提供了一致的、可靠的清除无用副本的方法。第二，垃圾回收把存储空间的回收操作合并到 Master 节点规律性的后台活动中，比如，例行扫描和与 Chunk 服务器握手等。因此，操作被批量的执行，开销会被分散。另外，垃圾回收在 Master 节点相对空闲的时候完成。这样 Master

[23] 原文：This effectively severs its links to all its chunks

[24] 自身的指删除 metadata 的消息

节点就可以给那些需要快速反应的客户机请求提供更快捷的响应。第三，延缓存储空间回收为意外的、不可逆转的删除操作提供了安全保障。

根据我们的使用经验，延迟回收空间的主要问题是，延迟回收会阻碍用户调优存储空间的使用，特别是当存储空间比较紧缺的时候。当应用程序重复创建和删除临时文件时，释放的存储空间不能马上重用。我们通过显式的再次删除一个已经被删除的文件的方式加速空间回收的速度。我们允许用户为命名空间的不同部分设定不同的复制和回收策略。例如，用户可以指定某些目录树下面的文件不做复制，删除的文件被即时的、不可恢复的从文件系统移除。

4.5 过期失效的副本检测

当 Chunk 服务器失效时，Chunk 的副本有可能因错失了一些修改操作而过期失效。Master 节点保存了每个 Chunk 的版本号，用来区分当前的副本和过期副本。

无论何时，只要 Master 节点和 Chunk 签订一个新的租约，它就增加 Chunk 的版本号，然后通知最新的副本。Master 节点和这些副本都把新的版本号记录在它们持久化存储的状态信息中。这个动作发生在任何客户机得到通知以前，因此也是对这个 Chunk 开始写之前。如果某个副本所在的 Chunk 服务器正好处于失效状态，那么副本的版本号就不会被增加。Master 节点在这个 Chunk 服务器重新启动，并且向 Master 节点报告它拥有的 Chunk 的集合以及相应的版本号的时候，就会检测出它包含过期的 Chunk。如果 Master 节点看到一个比它记录的版本号更高的版本号，Master 节点会认为它和 Chunk 服务器签订租约的操作失败了，因此会选择更高的版本号作为当前的版本号。

Master 节点在例行的垃圾回收过程中移除所有的过期失效副本。在此之前，Master 节点在回复客户机的 Chunk 信息请求的时候，简单的认为那些过期的块根本就不存在。另外一重保障措施是，Master 节点在通知客户机哪个 Chunk 服务器持有租约、或者指示 Chunk 服务器从哪个 Chunk 服务器进行克隆时，消息中都附带了 Chunk 的版本号。客户机或者 Chunk 服务器在执行操作时都会验证版本号以确保总是访问当前版本的数据。

5 容错和诊断

我们在设计 GFS 时遇到的最大挑战之一是如何处理频繁发生的组件失效。组件的数量和质量让这些问题出现的频率远远超过一般系统意外发生的频率：我们不能完全依赖机器的稳定性，也不能完全相信硬盘的可靠性。组件的失效可能造成系统不可用，更糟糕的是，还可能产生不完整的数据。我们讨论我们如何面对这些挑战，以及当组件失效不可避免的发生时，用 GFS 自带工具诊断系统故障。

5.1 高可用性

在 GFS 集群的数百个服务器之中，在任何给定的时间必定会有些服务器是不可用的。我们使用两条简单

但是有效的策略保证整个系统的高可用性：快速恢复和复制。

5.1.1 快速恢复

不管 Master 服务器和 Chunk 服务器是如何关闭的，它们都被设计为可以在数秒钟内恢复它们的状态并重新启动。事实上，我们并不区分正常关闭和异常关闭；通常，我们通过直接 kill 掉进程来关闭服务器。客户机和其它的服务器会感觉到系统有点颠簸[25]，正在发出的请求会超时，需要重新连接到重启后的服务器，然后重试这个请求。6.6.2 章节记录了实测的启动时间。

5.1.2 Chunk 复制

正如之前讨论的，每个 Chunk 都被复制到不同机架上的不同的 Chunk 服务器上。用户可以为文件命名空间的不同部分设定不同的复制级别。缺省是 3。当有 Chunk 服务器离线了，或者通过 Chksum 校验（参考 5.2 节）发现了已经损坏的数据，Master 节点通过克隆已有的副本保证每个 Chunk 都被完整复制[26]。虽然 Chunk 复制策略对我们非常有效，但是我们也在寻找其它形式的跨服务器的冗余解决方案，比如使用奇偶校验、或者 Erasure codes[27]来解决我们日益增长的只读存储需求。我们的系统主要的工作负载是追加方式的写入和读取操作，很少有随机的写入操作，因此，我们认为在我们这个高度解耦合的系统架构下实现这些复杂的冗余方案很有挑战性，但并非不可实现。

5.1.3 Master 服务器的复制

为了保证 Master 服务器的可靠性，Master 服务器的状态也要复制。Master 服务器所有的操作日志和 checkpoint 文件都被复制到多台机器上。对 Master 服务器状态的修改操作能够提交成功的前提是，操作日志写入到 Master 服务器的备节点和本机的磁盘。简单说来，一个 Master 服务进程负责所有的修改操作，包括后台的服务，比如垃圾回收等改变系统内部状态活动。当它失效的时，几乎可以立刻重新启动。如果 Master 进程所在的机器或者磁盘失效了，处于 GFS 系统外部的监控进程会在其它的存有完整操作日志的机器上启动一个新的 Master 进程。客户端使用规范的名字访问 Master（比如 gfs-test）节点，这个名字类似 DNS 别名，因此也就可以在 Master 进程转到别的机器上执行时，通过更改别名的实际指向访问新的 Master 节点。

此外，GFS 中还有些“影子”Master 服务器，这些“影子”服务器在“主”Master 服务器宕机的时候提供文件系统的只读访问。它们是影子，而不是镜像，所以它们的数据可能比“主”Master 服务器更新要慢，通常是不到 1 秒。对于那些不经常改变的文件、或者那些允许获取的数据有少量过期的应用程序，“影子”

---

[25] a minor hiccup

[26] 即每个 Chunk 都有复制因子制定的个数个副本，缺省是 3

[27] Erasure codes 用来解决链接层中不相关的错误，以及网络拥塞和 buffer 限制造成的丢包错误

Master 服务器能够提高读取的效率。事实上，因为文件内容是从 Chunk 服务器上读取的，因此，应用程序不会发现过期的文件内容。在这个短暂的时间窗内，过期的可能是文件的元数据，比如目录的内容或者访问控制信息。

“影子”Master 服务器为了保持自身状态是最新的，它会读取一份当前正在进行的操作的日志副本，并且依照和主 Master 服务器完全相同的顺序来更改内部的数据结构。和主 Master 服务器一样，“影子”Master 服务器在启动的时候也会从 Chunk 服务器轮询数据（之后定期拉数据），数据中包括了 Chunk 副本的位置信息；“影子”Master 服务器也会定期和 Chunk 服务器“握手”来确定它们的状态。在主 Master 服务器因创建和删除副本导致副本位置信息更新时，“影子”Master 服务器才和主 Master 服务器通信来更新自身状态。

5.2 数据完整性

每个 Chunk 服务器都使用 Checksum 来检查保存的数据是否损坏。考虑到一个 GFS 集群通常都有好几百台机器、几千块硬盘，磁盘损坏导致数据在读写过程中损坏或者丢失是非常常见的（第 7 节讲了一个原因）。我们可以通过别的 Chunk 副本来解决数据损坏问题，但是跨越 Chunk 服务器比较副本来检查数据是否损坏很不实际。另外，GFS 允许有歧义的副本存在：GFS 修改操作的语义，特别是早先讨论过的原子纪录追加的操作，并不保证副本完全相同(alex 注：副本不是 byte-wise 完全一致的)。因此，每个 Chunk 服务器必须独立维护 Checksum 来校验自己的副本的完整性。

我们把每个 Chunk 都分成 64KB 大小的块。每个块都对应一个 32 位的 Checksum。和其它元数据一样，Checksum 与其它的用户数据是分开的，并且保存在内存和硬盘上，同时也记录操作日志。

对于读操作来说，在把数据返回给客户端或者其它的 Chunk 服务器之前，Chunk 服务器会校验读取操作涉及的范围内的块的 Checksum。因此 Chunk 服务器不会把错误数据传递到其它的机器上。如果发生某个块的 Checksum 不正确，Chunk 服务器返回给请求者一个错误信息，并且通知 Master 服务器这个错误。作为回应，请求者应当从其它副本读取数据，Master 服务器也会从其它副本克隆数据进行恢复。当一个新的副本就绪后，Master 服务器通知副本错误的 Chunk 服务器删掉错误的副本。

Checksum 对读操作的性能影响很小，可以基于几个原因来分析一下。因为大部分的读操作都至少要读取几个块，而我们只需要读取一小部分额外的相关数据进行校验。GFS 客户端代码通过每次把读取操作都对齐在 Checksum block 的边界上，进一步减少了这些额外的读取操作的负面影响。另外，在 Chunk 服务器上，Checksum 的查找和比较不需要 I/O 操作，Checksum 的计算可以和 I/O 操作同时进行。

Checksum 的计算针对在 Chunk 尾部的追加写入操作做了高度优化（与之对应的是覆盖现有数据的写入操作），因为这类操作在我们的工作中占了很大比例。我们只增量更新最后一个不完整的块的 Checksum，并且用所有的追加来的新 Checksum 块来计算新的 Checksum。即使是最后一个不完整的 Checksum 块已经损坏了，

而且我们不能够马上检查出来，由于新的 Checksum 和已有数据不吻合，在下次对这个块进行读取操作的时候，会检查出数据已经损坏了。

相比之下，如果写操作覆盖已经存在的一个范围内的 Chunk，我们必须读取和校验被覆盖的第一个和最后一个块，然后再执行写操作；操作完成之后再重新计算和写入新的 Checksum。如果我们不校验第一个和最后一个被写的块，那么新的 Checksum 可能会隐藏没有被覆盖区域内的数据错误。

在 Chunk 服务器空闲的时候，它会扫描和校验每个不活动的 Chunk 的内容。这使得我们能够发现很少被读取的 Chunk 是否完整。一旦发现有 Chunk 的数据损坏，Master 可以创建一个新的、正确的副本，然后把损坏的副本删除掉。这个机制也避免了非活动的、已损坏的 Chunk 欺骗 Master 节点，使 Master 节点认为它们已经有了足够多的副本了。

5.3 诊断工具

详尽的、深入细节的诊断日志，在问题隔离、调试、以及性能分析等方面给我们带来无法估量的帮助，同时也只需要很小的开销。没有日志的帮助，我们很难理解短暂的、不重复的机器之间的消息交互。GFS 的服务器会产生大量的日志，记录了大量关键的事件（比如，Chunk 服务器启动和关闭）以及所有的 RPC 的请求和回复。这些诊断日志可以随意删除，对系统的正确运行不造成任何影响。然而，我们在存储空间允许的情况下会尽量的保存这些日志。

RPC 日志包含了网络上发生的所有请求和响应的详细记录，但是不包括读写的文件数据。通过匹配请求与回应，以及收集不同机器上的 RPC 日志记录，我们可以重演所有的消息交互来诊断问题。日志还用来跟踪负载测试和性能分析。

日志对性能的影响很小（远小于它带来的好处），因为这些日志的写入方式是顺序的、异步的。最近发生的事件日志保存在内存中，可用于持续不断的在线监控。

6 度量

本节中，我们将使用一些小规模基准测试来展现 GFS 系统架构和实现上的一些固有瓶颈，还有些来自 Google 内部使用的真实的 GFS 集群的基准数据。

6.1 小规模基准测试

我们在一个包含 1 台 Master 服务器，2 台 Master 服务器复制节点，16 台 Chunk 服务器和 16 个客户机组成的 GFS 集群上测量性能。注意，采用这样的集群配置方案只是为了易于测试。典型的 GFS 集群有数百个 Chunk 服务器和数百个客户机。

所有机器的配置都一样：两个 PIII 1.4GHz 处理器，2GB 内存，两个 80G/5400rpm 的硬盘，以及 100Mbps

全双工以太网连接到一个 HP2524 交换机。GFS 集群中所有的 19 台服务器都连接在一个交换机，所有 16 台客户机连接到另一个交换机上。两个交换机之间使用 1Gbps 的线路连接。

6.1.1 读取

N 个客户机从 GFS 文件系统同步读取数据。每个客户机从 320GB 的文件集合中随机读取 4MB region 的内容。读取操作重复执行 256 次，因此，每个客户机最终都读取 1GB 的数据。所有的 Chunk 服务器加起来总共只有 32GB 的内存，因此，我们预期只有最多 10%的读取请求命中 Linux 的文件系统缓冲。我们的测试结果应该和一个在没有文件系统缓存的情况下读取测试的结果接近。

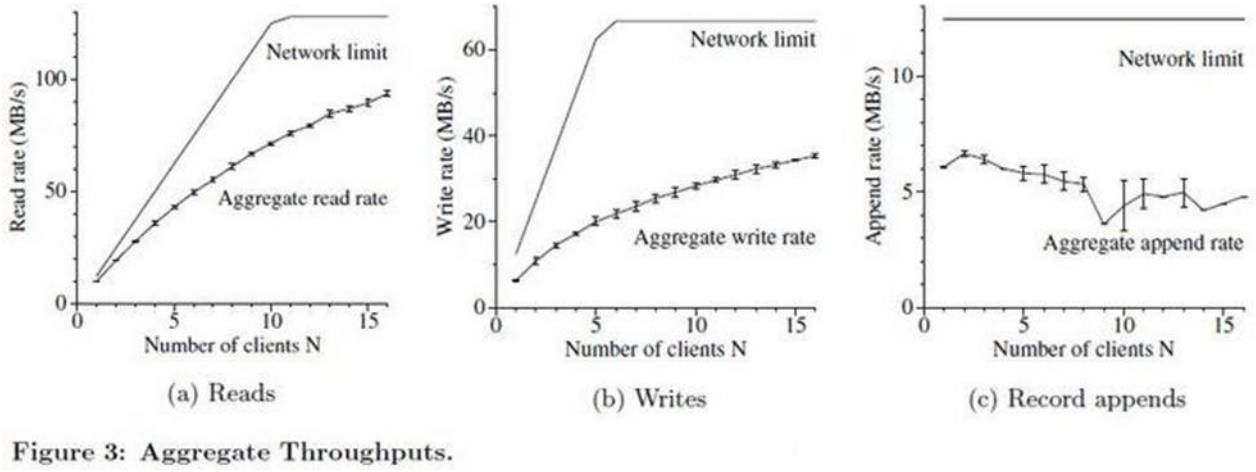


Figure 3: Aggregate Throughputs.

图三：合计吞吐量

上边的曲线显示了我们网络拓扑下的合计理论吞吐量上限。下边的曲线显示了观测到的吞吐量。这个曲线有着 95%的可靠性，因为有时候测量会不够精确。

图 3（a）显示了 N 个客户机整体的读取速度以及这个速度的理论极限。当连接两个交换机的 1Gbps 的链路饱和时，整体读取速度达到理论的极限值是 125MB/S，或者说每个客户机配置的 100Mbps 网卡达到饱和时，每个客户机读取速度的理论极限值是 12.5MB/s。实测结果是，当一个客户机读取的时候，读取的速度是 10MB/s，也就是说达到客户机理论读取速度极限值的 80%。对于 16 个客户机，整体的读取速度达到了 94MB/s，大约是理论整体读取速度极限值的 75%，也就是说每个客户机的读取速度是 6MB/s。读取效率从 80%降低到了 75%，主要的原因是当读取的客户机增加时，多个客户机同时读取一个 Chunk 服务器的几率也增加了，导致整体的读取效率下降。

6.1.2 写入

N 个客户机同时向 N 个不同的文件中写入数据。每个客户机以每次 1MB 的速度连续写入 1GB 的数据。图 3（b）显示了整体的写入速度和它们理论上的极限值。理论上的极限值是 67MB/s，因为我们需要把每一 byte 写入到 16 个 Chunk 服务器中的 3 个上，而每个 Chunk 服务器的输入连接速度是 12.5MB/s。

一个客户机的写入速度是 6.3MB，大概是理论极限值的一半。导致这个结果的主要原因是我们的网络协议栈。它与我们推送数据到 Chunk 服务器时采用的管道模式不相适应。从一个副本到另一个副本的数据传输延迟降低了整个的写入速度。

16 个客户机整体的写入速度达到了 35MB/s（即每个客户机 2.2MB/s），大约只是理论极限值的一半。和多个客户机读取的情形很类型，随着客户机数量的增加，多个客户机同时写入同一个 Chunk 服务器的几率也增加了。而且，16 个客户机并行写入可能引起的冲突比 16 个客户机并行读取要大得多，因为每个写入都会涉及三个不同的副本。

写入的速度比我们想象的要慢。在实际应用中，这没有成为我们的主要问题，因为即使在单个客户机上能够感受到延时，它也不会在有大量客户机的时候对整体的写入带宽造成显著的影响。

6.1.3 记录追加

图 3（c）显示了记录追加操作的性能。N 个客户机同时追加数据到一个文件。记录追加操作的性能受限于保存文件最后一个 Chunk 的 Chunk 服务器的带宽，而与客户机的数量无关。记录追加的速度由一个客户机的 6.0MB/s 开始，下降到 16 个客户机的 4.8MB/s 为止，速度的下降主要是由于不同客户端的网络拥塞以及网络传输速度的不同而导致的。

我们的程序倾向于同时处理多个这样的文件。换句话说，即 N 个客户机同时追加数据到 M 个共享文件中，这里 N 和 M 都是数十或者数百以上。所以，在我们的实际应用中，Chunk 服务器的网络拥塞并没有成为一个严重问题，如果 Chunk 服务器的某个文件正在写入，客户机会去写另外一个文件。

6.2 实际应用中的集群

我们现在来仔细评估一下 Google 内部正在使用的两个集群，它们具有一定的代表性。集群 A 通常被上百个工程师用于研究和开发。典型的任务是被人工初始化后连续运行数个小时。它通常读取数 MB 到数 TB 的数据，之后进行转化或者分析，最后把结果写回到集群中。集群 B 主要用于处理当前的生产数据。集群 B 的任务持续的时间更长，在很少人工干预的情况下，持续的生成和处理数 TB 的数据集。在这两个案例中，一个单独的“任务”都是指运行在多个机器上的多个进程，它们同时读取和写入多个文件。

Cluster	A	B
Chunkservers	342	227
可用硬盘空间	72TB	180TB
已用硬盘空间	55TB	155TB
文件数量	735k	737k
死文件数量	22k	232k
Chunk数量	992k	1550k
chunkserver元数据	13GB	21GB
master元数据	48MB	60MB

表2：两个GFS集群特性

6.2.1 存储

如上表前五行所描述的，两个集群都由上百台 Chunk 服务器组成，支持数 TB 的硬盘空间；两个集群虽然都存储了大量的数据，但是还有剩余的空间。“已用空间”包含了所有的 Chunk 副本。实际上所有的文件都复制了三份。因此，集群实际上各存储了 18TB 和 52TB 的文件数据。

两个集群存储的文件数量都差不多，但是集群 B 上有大量的死文件。所谓“死文件”是指文件被删除了或者是被新版本的文件替换了，但是存储空间还没有来得及被回收。由于集群 B 存储的文件较大，因此它的 Chunk 数量也比较多。

6.2.2 元数据

Chunk 服务器总共保存了十几 GB 的元数据，大多数是来自用户数据的、64KB 大小的块的 Checksum。保存在 Chunk 服务器上其它的元数据是 Chunk 的版本号信息，我们在 4.5 节描述过。

在 Master 服务器上保存的元数据就小的多了，大约只有数十 MB，或者说平均每个文件 100 字节的元数据。这和我们设想的是一样的，Master 服务器的内存大小在实际应用中并不会成为 GFS 系统容量的瓶颈。大多数文件的元数据都是以前缀压缩模式存放的文件名。Master 服务器上存放的其它元数据包括了文件的所有者和权限、文件到 Chunk 的映射关系，以及每一个 Chunk 的当前版本号。此外，针对每一个 Chunk，我们都保存了当前的副本位置以及对它的引用计数，这个引用计数用于实现写时拷贝（即 COW，copy-on-write）。

对于每一个单独的服务器，无论是 Chunk 服务器还是 Master 服务器，都只保存了 50MB 到 100MB 的元数据。因此，恢复服务器是非常快速的：在服务器响应客户请求之前，只需要花几秒钟时间从磁盘上读取这些数据就可以了。不过，Master 服务器会持续颠簸一段时间－通常是 30 到 60 秒－直到它完成轮询所有的 Chunk

服务器，并获取到所有 Chunk 的位置信息。

6.2.3 读写速率

集群	A	B
读速率（最后一分钟）	583MB/s	380MB/s
读取率（最后一小时）	562MB/s	384MB/s
读速率（自从重启动）	589MB/s	49MB/s
写速率（最后一分钟）	1MB/s	101MB/s
写取率（最后一小时）	1MB/s	117MB/s
写速率（自从重启动）	25MB/s	13MB/s
master操作（最后一分钟）	325Ops/s	533Ops/s
master操作（最后一小时）	381Ops/s	518Ops/s
master操作（自从重启动）	202Ops/s	347Ops/s

表三：两个GFS集群的性能表

表三显示了不同时段的读写速率。在测试的时候，这两个集群都运行了一周左右的时间。（这两个集群最近都因为升级新版本的 GFS 重新启动过了）。

集群重新启动后，平均写入速率小于 30MB/s。当我们提取性能数据的时候，集群 B 正进行大量的写入操作，写入速度达到了 100MB/s，并且因为每个 Chunk 都有三个副本的原因，网络负载达到了 300MB/s。

读取速率要比写入速率高的多。正如我们设想的那样，总的工作负载中，读取的比例远远高于写入的比例。两个集群都进行着繁重的读取操作。特别是，集群 A 在一周时间内都维持了 580MB/s 的读取速度。集群 A 的网络配置可以支持 750MB/s 的速度，显然，它有效的利用了资源。集群 B 支持的峰值读取速度是 1300MB/s，但是它的应用只用到了 380MB/s。

6.2.4 Master 服务器的负载

表 3 的数据显示了发送到 Master 服务器的操作请求大概是每秒钟 200 到 500 个。Master 服务器可以轻松的应付这个请求速度，所以 Master 服务器的处理能力不是系统的瓶颈。

在早期版本的 GFS 中，Master 服务器偶尔会成为瓶颈。它大多数时间里都在顺序扫描某个很大的目录（包含数万个文件）去查找某个特定的文件。因此我们修改了 Master 服务器的数据结构，通过对名字空间进行二分查找来提高效率。现在 Master 服务器可以轻松的每秒钟进行数千次文件访问。如果有需要的话，我们可以通过在名称空间数据结构之前设置名称查询缓冲的方式进一步提高速度。

6.2.5 恢复时间

当某个 Chunk 服务器失效了，一些 Chunk 副本的数量可能会低于复制因子指定的数量，我们必须通过克隆副本使 Chunk 副本数量达到复制因子指定的数量。恢复所有 Chunk 副本所花费的时间取决于资源的数量。在我们的试验中，我们把集群 B 上的一个 Chunk 服务器 Kill 掉。这个 Chunk 服务器上大约有 15000 个 Chunk，共计 600GB 的数据。为了减小克隆操作对正在运行的应用程序的影响，以及为 GFS 调度决策提供修正空间，我们缺省的把集群中并发克隆操作的数量设置为 91 个（Chunk 服务器的数量的 40%），每个克隆操作最多允许使用的带宽是 6.25MB/s（50mbps）。所有的 Chunk 在 23.2 分钟内恢复了，复制的速度高达 440MB/s。

在另外一个测试中，我们 Kill 掉了两个 Chunk 服务器，每个 Chunk 服务器大约有 16000 个 Chunk，共计 660GB 的数据。这两个故障导致了 266 个 Chunk 只有单个副本。这 266 个 Chunk 被 GFS 优先调度进行复制，在 2 分钟内恢复到至少有两个副本；现在集群被带入到另外一个状态，在这个状态下，系统可以容忍另外一个 Chunk 服务器失效而不丢失数据。

6.3 工作负荷分析(Workload Breakdown)

本节中，我们展示了对两个 GFS 集群工作负载情况的详细分析，这两个集群和 6.2 节中的类似，但是不完全相同。集群 X 用于研究和开发，集群 Y 用于生产数据处理。

6.3.1 方法论和注意事项

本章节列出的这些结果数据只包括客户机发起的原始请求，因此，这些结果能够反映我们的应用程序对 GFS 文件系统产生的全部工作负载。它们不包含那些为了实现客户端请求而在服务器间交互的请求，也不包含 GFS 内部的后台活动相关的请求，比如前向转发的写操作，或者重新负载均衡等操作。

我们从 GFS 服务器记录的真实的 RPC 请求日志中推导重建出关于 IO 操作的统计信息。例如，GFS 客户程序可能会把一个读操作分成几个 RPC 请求来提高并行度，我们可以通过这些 RPC 请求推导出原始的读操作。因为我们的访问模式是高度程式化，所以我们认为任何不符合的数据都是误差[28]。应用程序如果能够记录更详尽的日志，就有可能提供更准确的诊断数据；但是为了这个目的去重新编译和重新启动数千个正在运行的客户机是不现实的，而且从那么多客户机上收集结果也是个繁重的工作。

应该避免从我们的工作负荷数据中过度的归纳出普遍的结论[29]。因为 Google 完全控制着 GFS 和使用 GFS 的应用程序，所以，应用程序都针对 GFS 做了优化，同时，GFS 也是为了这些应用程序而设计的。这样的相互作用也可能存在于一般程序和文件系统中，但是在我们的案例中这样的作用影响可能更显著。

---

[28] 原文：Since our access patterns are highly stylized, we expect any error to be in the noise

[29] 即不要把本节的数据作为基础的指导性数据

6.3.2 Chunk 服务器工作负荷

操作	读		写		纪录增加	
集群	X	Y	X	Y	X	Y
0K	0.4	2.6	0	0	0	0
1B...1K	0.1	4.1	6.6	4.9	0.2	9.2
1K...8K	65.2	38.5	0.4	1.0	18.9	15.2
8K...64K	29.9	45.1	17.8	43.0	78.0	2.8
64K...128K	0.1	0.7	2.3	1.9	<.1	4.3
128K...256K	0.2	0.3	31.6	0.4	<.1	10.6
256K...512K	0.1	0.1	4.2	7.7	<.1	31.2
512K...1M	3.9	6.9	35.5	28.7	2.2	25.5
1M...inf	0.1	1.8	1.5	12.3	0.7	2.2

表4：操作大小百分比细目（%）。对于读操作，大小是实际读取的数据和传送的数据，而不是请求的数据量。

表 4 显示了操作按涉及的数据量大小的分布情况。读取操作按操作涉及的数据量大小呈现了双峰分布。小的读取操作（小于 64KB）一般是由查找操作的客户端发起的，目的在于从巨大的文件中查找小块的数据。大的读取操作（大于 512KB）一般是从头到尾顺序的读取整个文件。

在集群 Y 上，有相当数量的读操作没有返回任何的数据。在我们的应用中，尤其是在生产系统中，经常使用文件作为生产者-消费者队列。生产者并行的向文件中追加数据，同时，消费者从文件的尾部读取数据。某些情况下，消费者读取的速度超过了生产者写入的速度，这就会导致没有读到任何数据的情况。集群 X 通常用于短暂的数据分析任务，而不是长时间运行的分布式应用，因此，集群 X 很少出现这种情况。

写操作按数据量大小也同样呈现为双峰分布。大的写操作（超过 256KB）通常是由于 Writer 使用了缓存机制导致的。Writer 缓存较小的数据，通过频繁的 Checkpoint 或者同步操作，或者只是简单的统计小的写入（小于 64KB）的数据量(alex 注：即汇集多次小的写入操作，当数据量达到一个阈值，一次写入)，之后批量写入。

再来观察一下记录追加操作。我们可以看到集群 Y 中大的记录追加操作所占比例比集群 X 多的多，这是因为集群 Y 用于我们的生产系统，针对 GFS 做了更全面的调优。

操作	读		写		纪录增加	
集群	X	Y	X	Y	X	Y
1B..1K	<.1	<.1	<.1	<.1	<.1	<.1
1K...8K	13.8	3.9	<.1	<.1	<.1	0.1
8k...64k	11.4	9.3	2.4	5.9	2.3	0.3
64k...128k	0.3	0.7	0.3	0.3	22.7	1.2
128k...256k	0.8	0.6	16.5	0.2	<.1	5.8
256k...512k	1.4	0.3	3.4	7.7	<.1	38.4
512k...1M	65.9	55.1	74.1	58.0	0.1	46.8
1M...inf	6.4	30.1	3.3	28.0	53.9	7.4

表5：操作大小字节传输细目表（%）。对于读取来说，这个大小是实际读取的并且传输的，而不是请求读取的数量。两个不同点是如果读取尝试读超过文件结束的大小，那么实际读取大小就不是试图读取的大小，尝试读取超过文件结束的大小这样的操作在我们的负载中并不常见。

表 5 显示了按操作涉及的数据量的大小统计出来的总数据传输量。在所有的操作中，大的操作（超过 256KB）占据了主要的传输量。小的读取（小于 64KB）虽然传输的数据量比较少，但是在读取的数据量中仍占了相当的比例，这是因为在文件中随机 Seek 的工作负荷而导致的。

6.3.3 记录追加 vs. 写操作

记录追加操作在我们的生产系统中大量使用。对于集群 X，记录追加操作和普通写操作的比例按照字节比是 108:1，按照操作次数比是 8:1。对于作为我们的生产系统的集群 Y 来说，这两个比例分别是 3.7:1 和 2.5:1。更进一步，这一组数据说明在我们的两个集群上，记录追加操作所占比例都要比写操作要大。对于集群 X，在整个测量过程中，记录追加操作所占比率都比较低，因此结果会受到一两个使用某些特定大小的 buffer 的应用程序的影响。

如同我们所预期的，我们的数据修改操作主要是记录追加操作而不是覆盖方式的写操作。我们测量了第一个副本的数据覆盖写的情况。这近似于一个客户机故意覆盖刚刚写入的数据，而不是增加新的数据。对于集群 X，覆盖写操作在写操作所占据字节上的比例小于 0.0001%，在所占据操作数量上的比例小于 0.0003%。对于集群 Y，这两个比率都是 0.05%。虽然这只是某一片断的情况，但是仍然高于我们的预期。这是由于这些覆盖写的操作，大部分是由于客户端在发生错误或者超时以后重试的情况。这在本质上应该不算作工作负荷的一部分，而是重试机制产生的结果。

6.3.4 Master 的工作负荷

集群	X	Y
Open	26.1	16.3
Delete	0.7	1.5
FindLocation	64.3	65.8
FindLeaseHolder	7.8	13.4
FindMatchingFiles	0.6	2.2
其他	0.5	0.8

表6：master请求类型明细（%）

表 6 显示了 Master 服务器上的请求按类型区分的明细表。大部分的请求都是读取操作查询 Chunk 位置信息（FindLocation）、以及修改操作查询 lease 持有者的信息（FindLease-Locker）。

集群 X 和 Y 在删除请求的数量上有着明显的不同，因为集群 Y 存储了生产数据，一般会重新生成数据以及用新版本的数据替换旧有的数据。数量上的差异也被隐藏在了 Open 请求中，因为旧版本的文件可能在以重新写入的模式打开时，隐式的被删除了（类似 UNIX 的 open 函数中的“w”模式）。

FindMatchingFiles 是一个模式匹配请求，支持“ls”以及其它类似的文件系统操作。不同于 Master 服务器的其它请求，它可能会检索 namespace 的大部分内容，因此是非常昂贵的操作。集群 Y 的这类请求要多一些，因为自动化数据处理的任务进程需要检查文件系统的各个部分，以便从全局上了解应用程序的状态。与之不同的是，集群 X 的应用程序更加倾向于由单独的用户控制，通常预先知道自己所需要使用的全部文件的名称。

7 经验

在建造和部署 GFS 的过程中，我们经历了各种各样的问题，有些是操作上的，有些是技术上的。

起初，GFS 被设想为我们的生产系统的后端文件系统。随着时间推移，在 GFS 的使用中逐步的增加了对研究和开发任务的支持。我们开始增加一些小的功能，比如权限和配额，到了现在，GFS 已经初步支持了这些功能。虽然我们生产系统是严格受控的，但是用户层却不总是这样的。需要更多的基础架构来防止用户间的相互干扰。

我们最大的问题是磁盘以及和 Linux 相关的问题。很多磁盘都声称它们支持某个范围内的 Linux IDE 硬盘驱动程序，但是实际应用中反映出来的情况却不是这样，它们只支持最新的驱动。因为协议版本很接近，所以大部分磁盘都可以用，但是偶尔也会有由于协议不匹配，导致驱动和内核对于驱动器的状态判断失误。这

会导致数据因为内核中的问题意外的被破坏了。这个问题促使我们使用 Checksum 来校验数据，同时我们也修改内核来处理这些因为协议不匹配带来的问题。

较早的时候，我们在使用 Linux 2.2 内核时遇到了些问题，主要是 fsync()的效率问题。它的效率与文件的大小而不是文件修改部分的大小有关。这在我们的操作日志文件过大时给出了难题，尤其是在我们尚未实现 Checkpoint 的时候。我们费了很大的力气用同步写来解决这个问题，但是最后还是移植到了 Linux2.4 内核上。

另一个和 Linux 相关的问题是单个读写锁的问题，也就是说，在某一个地址空间的任意一个线程都必须在从磁盘 page in（读锁）的时候先 hold 住，或者在 mmap()调用（写锁）的时候改写地址空间。我们发现即使我们的系统负载很轻的情况下也会有偶尔的超时，我们花费了很多的精力去查找资源的瓶颈或者硬件的问题。最后我们终于发现这个单个锁在磁盘线程交换以前映射的数据到磁盘的时候，锁住了当前的网络线程，阻止它把新数据映射到内存。由于我们的性能主要受限于网络接口，而不是内存 copy 的带宽，因此，我们用 pread()替代 mmap()，用了一个额外的 copy 动作来解决这个问题。

尽管偶尔还是有其它的问题，Linux 的开放源代码还是使我们能够快速探究和理解系统的行为。在适当的时候，我们会改进内核并且和公开源码组织共享这些改动。

8 相关工作

和其它的大型分布式文件系统，比如 AFS[5]类似，GFS 提供了一个与位置无关的名字空间，这使得数据可以为了负载均衡或者灾难冗余等目的在不同位置透明的迁移。不同于 AFS 的是，GFS 把文件分布存储到不同的服务器上，这种方式更类似 Xfs[1]和 Swift[3]，这是为了提高整体性能以及灾难冗余的能力。

由于磁盘相对来说比较便宜，并且复制的方式比 RAID[9]方法简单的多，GFS 目前只使用复制的方式来进行冗余，因此要比 xFS 或者 Swift 占用更多的裸存储空间(alex 注：Raw storage，裸盘的空间)。

与 AFS、xFS、Frangipani[12]以及 Intermezzo[6]等文件系统不同的是，GFS 并没有在文件系统层面提供任何 Cache 机制。我们主要的工作在单个应用程序执行的时候几乎不会重复读取数据，因为它们的工作方式要么是流式的读取一个大型的数据集，要么是在大型的数据集中随机 Seek 到某个位置，之后每次读取少量的数据。

某些分布式文件系统，比如 Frangipani、xFS、Minnesota’s GFS[11]、GPFS[10]，去掉了中心服务器，只依赖于分布式算法来保证一致性和可管理性。我们选择了中心服务器的方法，目的是为了简化设计，增加可靠性，能够灵活扩展。特别值得一提的是，由于处于中心位置的 Master 服务器保存有几乎所有的 Chunk 相关信息，并且控制着 Chunk 的所有变更，因此，它极大地简化了原本非常复杂的 Chunk 分配和复制策略的实现方法。我们通过减少 Master 服务器保存的状态信息的数量，以及将 Master 服务器的状态复制到其它节点来保

证系统的灾难冗余能力。扩展能力和高可用性（对于读取）目前是通过我们的影子 Master 服务器机制来保证的。对 Master 服务器状态更改是通过预写日志的方式实现持久化。为此，我们可以调整为使用类似 Harp[7]中的 primary-copy 方案，从而提供比我们现在的方案更严格的一致性保证。

我们解决了一个难题，这个难题类似 Lustre[8]在如何在有大量客户端时保障系统整体性能遇到的问题。不过，我们通过只关注我们的应用程序的需求，而不是提供一个兼容 POSIX 的文件系统，从而达到了简化问题的目的。此外，GFS 设计预期是使用大量的不可靠节点组建集群，因此，灾难冗余方案是我们设计的核心。

GFS 很类似 NASD 架构[4]。NASD 架构是基于网络磁盘的，而 GFS 使用的是普通计算机作为 Chunk 服务器，就像 NASD 原形中方案一样。所不同的是，我们的 Chunk 服务器采用惰性分配固定大小的 Chunk 的方式，而不是分配变长的对象存储空间。此外，GFS 实现了诸如重新负载均衡、复制、恢复机制等等在生产环境中需要的特性。

不同于与 Minnesota’s GFS 和 NASD，我们并不改变存储设备的 Model[30]。我们只关注用普通的设备来解决非常复杂的分布式系统日常的数据处理。

我们通过原子的记录追加操作实现了生产者-消费者队列，这个问题类似 River[2]中的分布式队列。River 使用的是跨主机的、基于内存的分布式队列，为了实现这个队列，必须仔细控制数据流；而 GFS 采用可以被生产者并发追加记录的持久化的文件的方式实现。River 模式支持 m-到-n 的分布式队列，但是缺少由持久化存储提供的容错机制，GFS 只支持 m-到-1 的队列。多个消费者可以同时读取一个文件，但是它们输入流的区间必须是对齐的。

9 结束语

Google 文件系统展示了一个使用普通硬件支持大规模数据处理的系统的特质。虽然一些设计要点都是针对我们的特殊的需要定制的，但是还是有很多特性适用于类似规模的和成本的数据处理任务。

首先，我们根据我们当前的和可预期的将来的应用规模和技术环境来评估传统的文件系统的特性。我们的评估结果将我们引导到一个使用完全不同于传统的设计思路上。根据我们的设计思路，我们认为组件失效是常态而不是异常，针对采用追加方式（有可能是并发追加）写入、然后再读取（通常序列化读取）的大文件进行优化，以及扩展标准文件系统接口、放松接口限制来改进整个系统。

我们系统通过持续监控，复制关键数据，快速和自动恢复提供灾难冗余。Chunk 复制使得我们可以对 Chunk 服务器的失效进行容错。高频率的组件失效要求系统具备在线修复机制，能够周期性的、透明的修复损坏的数据，也能够第一时间重新建立丢失的副本。此外，我们使用 Checksum 在磁盘或者 IDE 子系统级别检测数据损坏，在这样磁盘数量惊人的大系统中，损坏率是相当高的。

[30] 对这两个文件系统不了解，因为不太明白改变存储设备的 Model 用来做什么，这不明白这个 model 是模型、还是型号

我们的设计保证了在有大量的并发读写操作时能够提供很高的合计吞吐量。我们通过分离控制流和数据流来实现这个目标，控制流在 Master 服务器处理，而数据流在 Chunk 服务器和客户端处理。当一般的操作涉及到 Master 服务器时，由于 GFS 选择的 Chunk 尺寸较大(alex 注：从而减小了元数据的大小)，以及通过 Chunk Lease 将控制权限移交给主副本，这些措施将 Master 服务器的负担降到最低。这使得一个简单、中心的 Master 不会成为成为瓶颈。我们相信我们对网络协议栈的优化可以提升当前对于每客户端的写入吞吐量限制。

GFS 成功的实现了我们对存储的需求，在 Google 内部，无论是作为研究和开发的存储平台，还是作为生产系统的数据处理平台，都得到了广泛的应用。它是我们持续创新和处理整个 WEB 范围内的难题的一个重要工具。

10 致谢

We wish to thankt he following people for their contributions to the system or the paper. Brain Bershad (our shepherd) and the anonymous reviewers gave us valuable comments and suggestions. Anurag Acharya, Jeff Dean, and David des-Jardins contributed to the early design. Fay Chang worked on comparison of replicas across chunkservers. Guy Edjlali worked on storage quota. Markus Gutschke worked on a testing frameworkan d security enhancements. David

Kramer worked on performance enhancements. Fay Chang, Urs Hoelzle, Max Ibel, Sharon Perl, Rob Pike, and Debby Wallach commented on earlier drafts of the paper. Many of our colleagues at Google bravely trusted their data to a new file system and gave us useful feedback. Yoshka helped with early testing.

11 参考

[1] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang. Serverless networkfil e systems. In Proceedings of the 15th ACM Symposium on Operating System Principles, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS ’99), pages 10–22, Atlanta, Georgia, May 1999.

[3] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disks triping to provide high I/O data rates. Computer Systems, 4(4):405–436, 1991.

[4] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin,

yore

ErikR iedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In Proceedings of the 8th Architectural Support for Programming Languages and Operating Systems, pages 92–103, San Jose, California, October 1998.

[5] John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in a distributed file system. ACM Transactions on Computer Systems, 6(1):51–81, February 1988.

[6] InterMezzo. http://www.inter-mezzo.org, 2003.

[7] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In 13th Symposium on Operating System Principles, pages 226–238, Pacific Grove, CA, October 1991.

[8] Lustre. http://www.lustreorg, 2003.

[9] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, pages 109–116, Chicago, Illinois, September 1988.

[10] FrankS chmuck and Roger Haskin. GPFS: A shared-diskfi le system for large computing clusters. In Proceedings of the First USENIX Conference on File and Storage Technologies, pages 231–244, Monterey, California, January 2002.

[11] Steven R. Soltis, Thomas M. Ruwart, and Matthew T.O'Keefe. The Gobal File System. In Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies, College Park, Maryland, September 1996.

[12] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In Proceedings of the 16th ACM Symposium on Operating System Principles, pages 224–237, Saint-Malo, France, October 1997

yore

Google MapReduce 中文版[1]

摘要

MapReduce 是一个编程模型，也是一个处理和生成超大数据集的算法模型的相关实现。用户首先创建一个 Map 函数处理一个基于 key/value pair 的数据集合，输出中间的基于 key/value pair 的数据集合；然后再创建一个 Reduce 函数用来合并所有的具有相同中间 key 值的中间 value 值。现实世界中有很多满足上述处理模型的例子，本论文将详细描述这个模型。

MapReduce 架构的程序能够在大量的普通配置的计算机上实现并行化处理。这个系统在运行时只关心：如何分割输入数据，在大量计算机组成的集群上的调度，集群中计算机的错误处理，管理集群中计算机之间必要的通信。采用 MapReduce 架构可以使那些没有并行计算和分布式处理系统开发经验的程序员有效利用分布式系统的丰富资源。

我们的 MapReduce 实现运行在规模可以灵活调整的由普通机器组成的集群上：一个典型的 MapReduce 计算往往由几千台机器组成、处理以 TB 计算的数据。程序员发现这个系统非常好用：已经实现了数以百计的 MapReduce 程序，在 Google 的集群上，每天都有 1000 多个 MapReduce 程序在执行。

1 介绍

在过去的 5 年里，包括本文作者在内的 Google 的很多程序员，为了处理海量的原始数据，已经实现了数以百计的、专用的计算方法。这些计算方法用来处理大量的原始数据，比如，文档抓取（类似网络爬虫的程序）、Web 请求日志等等；也为了计算处理各种类型的衍生数据，比如倒排索引、Web 文档的图结构的各种表示形势、每台主机上网络爬虫抓取的页面数量的汇总、每天被请求的最多的查询的集合等等。大多数这样的数据处理运算在概念上很容易理解。然而由于输入的数据量巨大，因此要想在可接受的时间内完成运算，只有将这些计算分布在成百上千的主机上。如何处理并行计算、如何分发数据、如何处理错误？所有这些问题综合在一起，需要大量的代码处理，因此也使得原本简单的运算变得难以处理。

为了解决上述复杂的问题，我们设计一个新的抽象模型，使用这个抽象模型，我们只要表述我们想要执行的简单运算即可，而不必关心并行计算、容错、数据分布、负载均衡等复杂的细节，这些问题都被封装在了一个库里面。设计这个抽象模型的灵感来自 Lisp 和许多其他函数式语言的 Map 和 Reduce 的原语。我们意识到我们大多数的运算都包含这样的操作：在输入数据的“逻辑”记录上应用 Map 操作得出一个中间 key/value pair 集合，然后在所有具有相同 key 值的 value 值上应用 Reduce 操作，从而达到合并中间的数据，得到一个

[1] 译者 alex，原文地址 http://blademaster.ixiezi.com/

想要的结果的目的。使用 MapReduce 模型，再结合用户实现的 Map 和 Reduce 函数，我们就可以非常容易的实现大规模并行化计算；通过 MapReduce 模型自带的“再次执行”（re-execution）功能，也提供了初级的容灾实现方案。

这个工作(实现一个 MapReduce 框架模型)的主要贡献是通过简单的接口来实现自动的并行化和大规模的分布式计算，通过使用 MapReduce 模型接口实现在大量普通的 PC 机上高性能计算。

第二部分描述基本的编程模型和一些使用案例。

第三部分描述了一个经过裁剪的、适合我们的基于集群的计算环境的 MapReduce 实现。

第四部分描述我们认为在 MapReduce 编程模型中一些实用的技巧。

第五部分对于各种不同的任务，测量我们 MapReduce 实现的性能。

第六部分揭示了在 Google 内部如何使用 MapReduce 作为基础重写我们的索引系统产品，包括其它一些使用 MapReduce 的经验。

第七部分讨论相关的和未来的工作。

2 编程模型

MapReduce 编程模型的原理是：利用一个输入 key/value pair 集合来产生一个输出的 key/value pair 集合。MapReduce 库的用户用两个函数表达这个计算：Map 和 Reduce。

用户自定义的 Map 函数接受一个输入的 key/value pair 值，然后产生一个中间 key/value pair 值的集合。MapReduce 库把所有具有相同中间 key 值 I 的中间 value 值集合在一起后传递给 reduce 函数。

用户自定义的 Reduce 函数接受一个中间 key 的值 I 和相关的一个 value 值的集合。Reduce 函数合并这些 value 值，形成一个较小的 value 值的集合。一般的，每次 Reduce 函数调用只产生 0 或 1 个输出 value 值。通常我们通过一个迭代器把中间 value 值提供给 Reduce 函数，这样我们就可以处理无法全部放入内存中的大量的 value 值的集合。

2.1 例子

例如，计算一个大的文档集合中每个单词出现的次数，下面是伪代码段：

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, “1″);
```

```
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

Map 函数输出文档中的每个词、以及这个词的出现次数(在这个简单的例子里就是 1)。Reduce 函数把 Map 函数产生的每一个特定的词的计数累加起来。

另外，用户编写代码，使用输入和输出文件的名字、可选的调节参数来完成一个符合 MapReduce 模型规范的对象，然后调用 MapReduce 函数，并把这个规范对象传递给它。用户的代码和 MapReduce 库链接在一起(用 C++实现)。附录 A 包含了这个实例的全部程序代码。

2.2 类型

尽管在前面例子的伪代码中使用了以字符串表示的输入输出值，但是在概念上，用户定义的 Map 和 Reduce 函数都有相关联的类型：

```
map(k1,v1) ->list(k2,v2)
reduce(k2,list(v2)) ->list(v2)
```

比如，输入的 key 和 value 值与输出的 key 和 value 值在类型上推导的域不同。此外，中间 key 和 value 值与输出 key 和 value 值在类型上推导的域相同。[2]

我们的 C++中使用字符串类型作为用户自定义函数的输入输出，用户在自己的代码中对字符串进行适当的类型转换。

2.3 更多的例子

这里还有一些有趣的简单例子，可以很容易的使用 MapReduce 模型来表示：

分布式的 Grep：Map 函数输出匹配某个模式的一行，Reduce 函数是一个恒等函数，即把中间数据复制到输出。

计算 URL 访问频率：Map 函数处理日志中 web 页面请求的记录，然后输出(URL,1)。Reduce 函数把相同

[2] 原文中这个 domain 的含义不是很清楚，我参考 Hadoop、KFS 等实现，map 和 reduce 都使用了泛型，因此，我把 domain 翻译成类型推导的域。

URL 的 value 值都累加起来，产生(URL,记录总数)结果。

倒转网络链接图：Map 函数在源页面（source）中搜索所有的链接目标（target）并输出为(target,source)。Reduce 函数把给定链接目标（target）的链接组合成一个列表，输出(target,list(source))。

每个主机的检索词向量：检索词向量用一个(词,频率)列表来概述出现在文档或文档集中的最重要的一些词。Map 函数为每一个输入文档输出(主机名,检索词向量)，其中主机名来自文档的 URL。Reduce 函数接收给定主机的所有文档的检索词向量，并把这些检索词向量加在一起，丢弃掉低频的检索词，输出一个最终的(主机名,检索词向量)。

倒排索引：Map 函数分析每个文档输出一个(词,文档号)的列表，Reduce 函数的输入是一个给定词的所有（词，文档号），排序所有的文档号，输出(词,list（文档号）)。所有的输出集合形成一个简单的倒排索引，它以一种简单的算法跟踪词在文档中的位置。

分布式排序：Map 函数从每个记录提取 key，输出(key,record)。Reduce 函数不改变任何的值。这个运算依赖分区机制(在 4.1 描述)和排序属性(在 4.2 描述)。

3 实现

MapReduce 模型可以有多种不同的实现方式。如何正确选择取决于具体的环境。例如，一种实现方式适用于小型的共享内存方式的机器，另外一种实现方式则适用于大型 NUMA 架构的多处理器的主机，而有的实现方式更适合大型的网络连接集群。

本章节描述一个适用于 Google 内部广泛使用的运算环境的实现：用以太网交换机连接、由普通 PC 机组成的大型集群。在我们的环境里包括：

1. x86 架构、运行 Linux 操作系统、双处理器、2-4GB 内存的机器。
2. 普通的网络硬件设备，每个机器的带宽为百兆或者千兆，但是远小于网络的平均带宽的一半。
3. 集群中包含成百上千的机器，因此，机器故障是常态。
4. 存储为廉价的内置 IDE 硬盘。一个内部分布式文件系统用来管理存储在这些磁盘上的数据。文件系统通过数据复制来在不可靠的硬件上保证数据的可靠性和有效性。
5. 用户提交工作（job）给调度系统。每个工作（job）都包含一系列的任务（task），调度系统将这些任务调度到集群中多台可用的机器上。

3.1 执行概括

通过将 Map 调用的输入数据自动分割为 M 个数据片段的集合，Map 调用被分布到多台机器上执行。输入的数据片段能够在不同的机器上并行处理。使用分区函数将 Map 调用产生的中间 key 值分成 R 个不同分区

（例如，hash(key) mod R），Reduce 调用也被分布到多台机器上执行。分区数量（R）和分区函数由用户来指定。

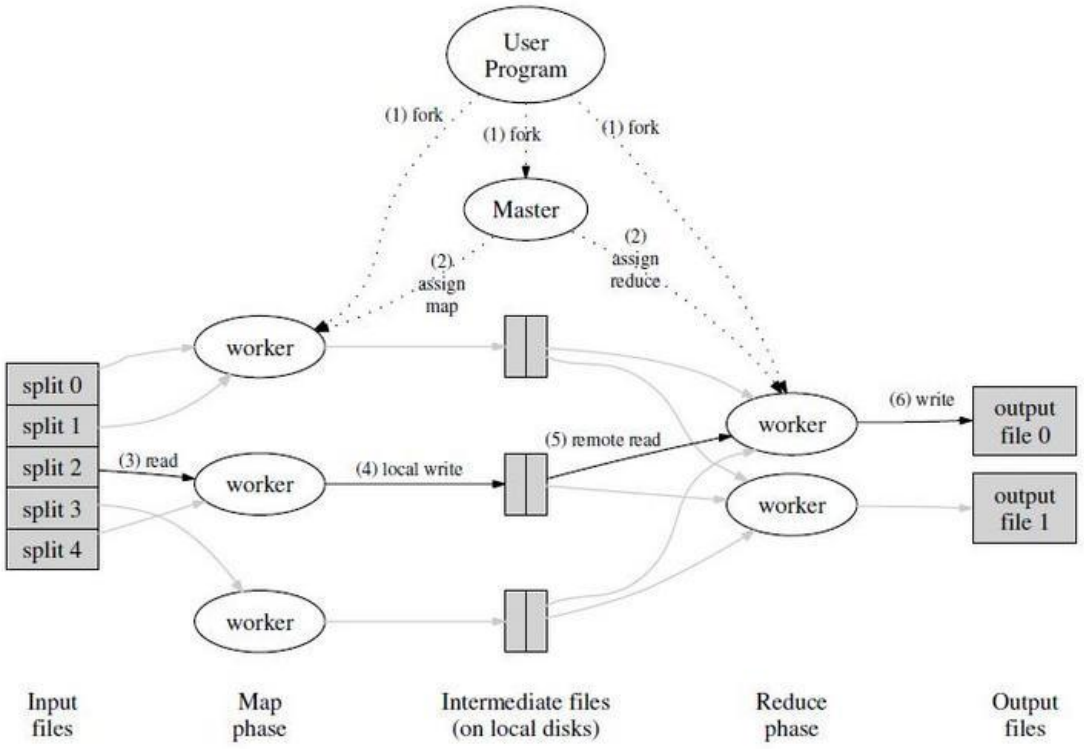


Figure 1: Execution overview

图 1 展示了我们的 MapReduce 实现中操作的全部流程。当用户调用 MapReduce 函数时，将发生下面的一系列动作（下面的序号和图 1 中的序号一一对应）：

1. 用户程序首先调用的 MapReduce 库将输入文件分成 M 个数据片度，每个数据片段的大小一般从 16MB 到 64MB(可以通过可选的参数来控制每个数据片段的大小)。然后用户程序在机群中创建大量的程序副本。
2. 这些程序副本中的有一个特殊的程序－master。副本中其它的程序都是 worker 程序，由 master 分配任务。有 M 个 Map 任务和 R 个 Reduce 任务将被分配，master 将一个 Map 任务或 Reduce 任务分配给一个空闲的 worker。
3. 被分配了 map 任务的 worker 程序读取相关的输入数据片段，从输入的数据片段中解析出 key/value pair，然后把 key/value pair 传递给用户自定义的 Map 函数，由 Map 函数生成并输出的中间 key/value pair，并缓存在内存中。
4. 缓存中的 key/value pair 通过分区函数分成 R 个区域，之后周期性的写入到本地磁盘上。缓存的 key/value pair 在本地磁盘上的存储位置将被回传给 master，由 master 负责把这些存储位置再传送给 Reduce worker。

5. 当 Reduce worker 程序接收到 master 程序发来的数据存储位置信息后，使用 RPC 从 Map worker 所在主机的磁盘上读取这些缓存数据。当 Reduce worker 读取了所有的中间数据后，通过对 key 进行排序后使得具有相同 key 值的数据聚合在一起。由于许多不同的 key 值会映射到相同的 Reduce 任务上，因此必须进行排序。如果中间数据太大无法在内存中完成排序，那么就要在外部进行排序。
6. Reduce worker 程序遍历排序后的中间数据，对于每一个唯一的中间 key 值，Reduce worker 程序将这个 key 值和它相关的中间 value 值的集合传递给用户自定义的 Reduce 函数。Reduce 函数的输出被追加到所属分区的输出文件。
7. 当所有的 Map 和 Reduce 任务都完成之后，master 唤醒用户程序。在这个时候，在用户程序里的对 MapReduce 调用才返回。

在成功完成任务之后，MapReduce 的输出存放在 R 个输出文件中（对应每个 Reduce 任务产生一个输出文件，文件名由用户指定）。一般情况下，用户不需要将这 R 个输出文件合并成一个文件－他们经常把这些文件作为另外一个 MapReduce 的输入，或者在另外一个可以处理多个分割文件的分布式应用中使用。

3.2 Master 数据结构

Master 持有一些数据结构，它存储每一个 Map 和 Reduce 任务的状态（空闲、工作中或完成），以及 Worker 机器(非空闲任务的机器)的标识。

Master 就像一个数据管道，中间文件存储区域的位置信息通过这个管道从 Map 传递到 Reduce。因此，对于每个已经完成的 Map 任务，master 存储了 Map 任务产生的 R 个中间文件存储区域的大小和位置。当 Map 任务完成时，Master 接收到位置和大小的更新信息，这些信息被逐步递增的推送给那些正在工作的 Reduce 任务。

3.3 容错

因为 MapReduce 库的设计初衷是使用由成百上千的机器组成的集群来处理超大规模的数据，所以，这个库必须要能很好的处理机器故障。

3.3.1 worker 故障

master 周期性的 ping 每个 worker。如果在一个约定的时间范围内没有收到 worker 返回的信息，master 将把这个 worker 标记为失效。所有由这个失效的 worker 完成的 Map 任务被重设为初始的空闲状态，之后这些任务就可以被安排给其他的 worker。同样的，worker 失效时正在运行的 Map 或 Reduce 任务也将被重新置为空闲状态，等待重新调度。

当 worker 故障时，由于已经完成的 Map 任务的输出存储在这台机器上，Map 任务的输出已不可访问了，

因此必须重新执行。而已经完成的 Reduce 任务的输出存储在全局文件系统上，因此不需要再次执行。

当一个 Map 任务首先被 worker A 执行，之后由于 worker A 失效了又被调度到 worker B 执行，这个“重新执行”的动作会被通知给所有执行 Reduce 任务的 worker。任何还没有从 worker A 读取数据的 Reduce 任务将从 worker B 读取数据。

MapReduce 可以处理大规模 worker 失效的情况。比如，在一个 MapReduce 操作执行期间，在正在运行的集群上进行网络维护引起 80 台机器在几分钟内不可访问了，MapReduce master 只需要简单的再次执行那些不可访问的 worker 完成的工作，之后继续执行未完成的任务，直到最终完成这个 MapReduce 操作。

3.3.2 master 失败

一个简单的解决办法是让 master 周期性的将上面描述的数据结构（alex 注：指 3.2 节）的写入磁盘，即检查点（checkpoint）。如果这个 master 任务失效了，可以从最后一个检查点（checkpoint）开始启动另一个 master 进程。然而，由于只有一个 master 进程，master 失效后再恢复是比较麻烦的，因此我们现在的实现是如果 master 失效，就中止 MapReduce 运算。客户可以检查到这个状态，并且可以根据需要重新执行 MapReduce 操作。

3.3.3 在失效方面的处理机制

（alex 注：原文为”semantics in the presence of failures”）

当用户提供的 Map 和 Reduce 操作是输入确定性函数（即相同的输入产生相同的输出）时，我们的分布式实现在任何情况下的输出都和所有程序没有出现任何错误、顺序的执行产生的输出是一样的。

我们依赖对 Map 和 Reduce 任务的输出是原子提交的来完成这个特性。每个工作中的任务把它的输出写到私有的临时文件中。每个 Reduce 任务生成一个这样的文件，而每个 Map 任务则生成 R 个这样的文件（一个 Reduce 任务对应一个文件）。当一个 Map 任务完成的时，worker 发送一个包含 R 个临时文件名的完成消息给 master。如果 master 从一个已经完成的 Map 任务再次接收到到一个完成消息，master 将忽略这个消息；否则，master 将这 R 个文件的名字记录在数据结构里。

当 Reduce 任务完成时，Reduce worker 进程以原子的方式把临时文件重命名为最终的输出文件。如果同一个 Reduce 任务在多台机器上执行，针对同一个最终的输出文件将有多个重命名操作执行。我们依赖底层文件系统提供的重命名操作的原子性来保证最终的文件系统状态仅仅包含一个 Reduce 任务产生的数据。

使用 MapReduce 模型的程序员可以很容易的理解他们程序的行为，因为我们绝大多数的 Map 和 Reduce 操作是确定性的，而且存在这样的一个事实：我们的失效处理机制等价于一个顺序的执行的操作。当 Map 或/和 Reduce 操作是不确定性的时候，我们提供虽然较弱但是依然合理的处理机制。当使用非确定操作的时候，一个 Reduce 任务 R1 的输出等价于一个非确定性程序顺序执行产生时的输出。但是，另一个 Reduce 任务 R2

的输出也许符合一个不同的非确定顺序程序执行产生的 R2 的输出。

考虑 Map 任务 M 和 Reduce 任务 R1、R2 的情况。我们设定 e(Ri)是 Ri 已经提交的执行过程（有且仅有一个这样的执行过程）。当 e(R1)读取了由 M 一次执行产生的输出，而 e(R2)读取了由 M 的另一次执行产生的输出，导致了较弱的失效处理。

3.4 存储位置

在我们的计算运行环境中，网络带宽是一个相当匮乏的资源。我们通过尽量把输入数据(由 GFS 管理)存储在集群中机器的本地磁盘上来节省网络带宽。GFS 把每个文件按 64MB 一个 Block 分隔，每个 Block 保存在多台机器上，环境中就存放了多份拷贝(一般是 3 个拷贝)。MapReduce 的 master 在调度 Map 任务时会考虑输入文件的位置信息，尽量将一个 Map 任务调度在包含相关输入数据拷贝的机器上执行；如果上述努力失败了，master 将尝试在保存有输入数据拷贝的机器附近的机器上执行 Map 任务(例如，分配到一个和包含输入数据的机器在一个 switch 里的 worker 机器上执行)。当在一个足够大的 cluster 集群上运行大型 MapReduce 操作的时候，大部分的输入数据都能从本地机器读取，因此消耗非常少的网络带宽。

3.5 任务粒度

如前所述，我们把 Map 拆分成了 M 个片段、把 Reduce 拆分成 R 个片段执行。理想情况下，M 和 R 应当比集群中 worker 的机器数量要多得多。在每台 worker 机器都执行大量的不同任务能够提高集群的动态的负载均衡能力，并且能够加快故障恢复的速度：失效机器上执行的大量 Map 任务都可以分布到所有其他的 worker 机器上去执行。

但是实际上，在我们的具体实现中对 M 和 R 的取值都有一定的客观限制，因为 master 必须执行 O(M+R)次调度，并且在内存中保存 O(M*R)个状态（对影响内存使用的因素还是比较小的：O(M*R)块状态，大概每对 Map 任务/Reduce 任务 1 个字节就可以了）。

更进一步，R 值通常是由用户指定的，因为每个 Reduce 任务最终都会生成一个独立的输出文件。实际使用时我们也倾向于选择合适的 M 值，以使得每一个独立任务都是处理大约 16M 到 64M 的输入数据（这样，上面描写的输入数据本地存储优化策略才最有效），另外，我们把 R 值设置为我们想使用的 worker 机器数量的小的倍数。我们通常会用这样的比例来执行 MapReduce：M=200000，R=5000，使用 2000 台 worker 机器。

3.6 备用任务

影响一个 MapReduce 的总执行时间最通常的因素是“落伍者”：在运算过程中，如果有一台机器花了很长的时间才完成最后几个 Map 或 Reduce 任务，导致 MapReduce 操作总的执行时间超过预期。出现“落伍者”的原因非常多。比如：如果一个机器的硬盘出了问题，在读取的时候要经常的进行读取纠错操作，导致读取

数据的速度从 30M/s 降低到 1M/s。如果 cluster 的调度系统在这台机器上又调度了其他的任务，由于 CPU、内存、本地硬盘和网络带宽等竞争因素的存在，导致执行 MapReduce 代码的执行效率更加缓慢。我们最近遇到的一个问题是由于机器的初始化代码有 bug，导致关闭了的处理器的缓存：在这些机器上执行任务的性能和正常情况相差上百倍。

我们有一个通用的机制来减少“落伍者”出现的情况。当一个 MapReduce 操作接近完成的时候，master 调度备用（backup）任务进程来执行剩下的、处于处理中状态（in-progress）的任务。无论是最初的执行进程、还是备用（backup）任务进程完成了任务，我们都把这个任务标记成为已经完成。我们调优了这个机制，通常只会占用比正常操作多几个百分点的计算资源。我们发现采用这样的机制对于减少超大 MapReduce 操作的总处理时间效果显著。例如，在 5.3 节描述的排序任务，在关闭掉备用任务的情况下要多花 44%的时间完成排序任务。

4 技巧

虽然简单的 Map 和 Reduce 函数提供的基本功能已经能够满足大部分的计算需要，我们还是发掘出了一些有价值的扩展功能。本节将描述这些扩展功能。

4.1 分区函数

MapReduce 的使用者通常会指定 Reduce 任务和 Reduce 任务输出文件的数量（R）。我们在中间 key 上使用分区函数来对数据进行分区，之后再输入到后续任务执行进程。一个缺省的分区函数是使用 hash 方法(比如，hash(key) mod R)进行分区。hash 方法能产生非常平衡的分区。然而，有的时候，其它的一些分区函数对 key 值进行的分区将非常有用。比如，输出的 key 值是 URLs，我们希望每个主机的所有条目保持在同一个输出文件中。为了支持类似的情况，MapReduce 库的用户需要提供专门的分区函数。例如，使用“hash(Hostname(urlkey)) mod R”作为分区函数就可以把所有来自同一个主机的 URLs 保存在同一个输出文件中。

4.2 顺序保证

我们确保在给定的分区中，中间 key/value pair 数据的处理顺序是按照 key 值增量顺序处理的。这样的顺序保证对每个分成生成一个有序的输出文件，这对于需要对输出文件按 key 值随机存取的应用非常有意义，对在排序输出的数据集也很有帮助。

4.3 Combiner 函数

在某些情况下，Map 函数产生的中间 key 值的重复数据会占很大的比重，并且，用户自定义的 Reduce 函数满足结合律和交换律。在 2.1 节的词数统计程序是个很好的例子。由于词频率倾向于一个 zipf 分布(齐夫分

布)，每个 Map 任务将产生成千上万个这样的记录<the,1>。所有的这些记录将通过网络被发送到一个单独的 Reduce 任务，然后由这个 Reduce 任务把所有这些记录累加起来产生一个数字。我们允许用户指定一个可选的 combiner 函数，combiner 函数首先在本地将这些记录进行一次合并，然后将合并的结果再通过网络发送出去。

Combiner 函数在每台执行 Map 任务的机器上都会被执行一次。一般情况下，Combiner 和 Reduce 函数是一样的。Combiner 函数和 Reduce 函数之间唯一的区别是 MapReduce 库怎样控制函数的输出。Reduce 函数的输出被保存在最终的输出文件里，而 Combiner 函数的输出被写到中间文件里，然后被发送给 Reduce 任务。

部分的合并中间结果可以显著的提高一些 MapReduce 操作的速度。附录 A 包含一个使用 combiner 函数的例子。

4.4 输入和输出的类型

MapReduce 库支持几种不同的格式的输入数据。比如，文本模式的输入数据的每一行被视为是一个 key/value pair。key 是文件的偏移量，value 是那一行的内容。另外一种常见的格式是以 key 进行排序来存储的 key/value pair 的序列。每种输入类型的实现都必须能够把输入数据分割成数据片段，该数据片段能够由单独的 Map 任务来进行后续处理(例如，文本模式的范围分割必须确保仅仅在每行的边界进行范围分割)。虽然大多数 MapReduce 的使用者仅仅使用很少的预定义输入类型就满足要求了，但是使用者依然可以通过提供一个简单的 Reader 接口实现就能够支持一个新的输入类型。

Reader 并非一定要从文件中读取数据，比如，我们可以很容易的实现一个从数据库里读记录的 Reader，或者从内存中的数据结构读取数据的 Reader。

类似的，我们提供了一些预定义的输出数据的类型，通过这些预定义类型能够产生不同格式的数据。用户采用类似添加新的输入数据类型的方式增加新的输出类型。

4.5 副作用

在某些情况下，MapReduce 的使用者发现，如果在 Map 和/或 Reduce 操作过程中增加辅助的输出文件会比较省事。我们依靠程序 writer 把这种“副作用”变成原子的和幂等的[3]。通常应用程序首先把输出结果写到一个临时文件中，在输出全部数据之后，在使用系统级的原子操作 rename 重新命名这个临时文件。

如果一个任务产生了多个输出文件，我们没有提供类似两阶段提交的原子操作支持这种情况。因此，对于会产生多个输出文件、并且对于跨文件有一致性要求的任务，都必须是确定性的任务。但是在实际应用过程中，这个限制还没有给我们带来过麻烦。

[3] 幂等的指一个总是产生相同结果的数学运算

4.6 跳过损坏的记录

有时候，用户程序中的 bug 导致 Map 或者 Reduce 函数在处理某些记录的时候 crash 掉，MapReduce 操作无法顺利完成。惯常的做法是修复 bug 后再次执行 MapReduce 操作，但是，有时候找出这些 bug 并修复它们不是一件容易的事情；这些 bug 也许是在第三方库里边，而我们手头没有这些库的源代码。而且在很多时候，忽略一些有问题的记录也是可以接受的，比如在一个巨大的数据集上进行统计分析的时候。我们提供了一种执行模式，在这种模式下，为了保证保证整个处理能继续进行，MapReduce 会检测哪些记录导致确定性的 crash，并且跳过这些记录不处理。

每个 worker 进程都设置了信号处理函数捕获内存段异常（segmentation violation）和总线错误（bus error）。在执行 Map 或者 Reduce 操作之前，MapReduce 库通过全局变量保存记录序号。如果用户程序触发了一个系统信号，消息处理函数将用“最后一口气”通过 UDP 包向 master 发送处理的最后一条记录的序号。当 master 看到在处理某条特定记录不止失败一次时，master 就标志着条记录需要被跳过，并且在下次重新执行相关的 Map 或者 Reduce 任务的时候跳过这条记录。

4.7 本地执行

调试 Map 和 Reduce 函数的 bug 是非常困难的，因为实际执行操作时不但是分布在系统中执行的，而且通常是在好几千台计算机上执行，具体的执行位置是由 master 进行动态调度的，这又大大增加了调试的难度。为了简化调试、profile 和小规模测试，我们开发了一套 MapReduce 库的本地实现版本，通过使用本地版本的 MapReduce 库，MapReduce 操作在本地计算机上顺序的执行。用户可以控制 MapReduce 操作的执行，可以把操作限制到特定的 Map 任务上。用户通过设定特别的标志来在本地执行他们的程序，之后就可以很容易的使用本地调试和测试工具（比如 gdb）。

4.8 状态信息

master 使用嵌入式的 HTTP 服务器（如 Jetty）显示一组状态信息页面，用户可以监控各种执行状态。状态信息页面显示了包括计算执行的进度，比如已经完成了多少任务、有多少任务正在处理、输入的字节数、中间数据的字节数、输出的字节数、处理百分比等等。页面还包含了指向每个任务的 stderr 和 stdout 文件的链接。用户根据这些数据预测计算需要执行大约多长时间、是否需要增加额外的计算资源。这些页面也可以用来分析什么时候计算执行的比预期的要慢。

另外，处于最顶层的状态页面显示了哪些 worker 失效了，以及他们失效的时候正在运行的 Map 和 Reduce 任务。这些信息对于调试用户代码中的 bug 很有帮助。

4.9 计数器

MapReduce 库使用计数器统计不同事件发生次数。比如，用户可能想统计已经处理了多少个单词、已经索引的多少篇 German 文档等等。

为了使用这个特性，用户在程序中创建一个命名的计数器对象，在 Map 和 Reduce 函数中相应的增加计数器的值。例如：

```
Counter* uppercase;
uppercase = GetCounter("uppercase");
map(String name, String contents):
  for each word w in contents:
    if (IsCapitalized(w)):
      uppercase->Increment();
    EmitIntermediate(w, "1");
```

这些计数器的值周期性的从各个单独的 worker 机器上传递给 master（附加在 ping 的应答包中传递）。master 把执行成功的 Map 和 Reduce 任务的计数器值进行累计，当 MapReduce 操作完成之后，返回给用户代码。

计数器当前的值也会显示在 master 的状态页面上，这样用户就可以看到当前计算的进度。当累加计数器的值的时候，master 要检查重复运行的 Map 或者 Reduce 任务，避免重复累加（之前提到的备用任务和失效后重新执行任务这两种情况会导致相同的任务被多次执行）。

有些计数器的值是由 MapReduce 库自动维持的，比如已经处理的输入的 key/value pair 的数量、输出的 key/value pair 的数量等等。

计数器机制对于 MapReduce 操作的完整性检查非常有用。比如，在某些 MapReduce 操作中，用户需要确保输出的 key value pair 精确的等于输入的 key value pair，或者处理的 German 文档数量在处理的整个文档数量中属于合理范围。

5 性能

本节我们用在一个大型集群上运行的两个计算来衡量 MapReduce 的性能。一个计算在大约 1TB 的数据中进行特定的模式匹配，另一个计算对大约 1TB 的数据进行排序。

这两个程序在大量的使用 MapReduce 的实际应用中是非常典型的 — 一类是对数据格式进行转换，从一种表现形式转换为另外一种表现形式；另一类是从海量数据中抽取少部分的用户感兴趣的数据。

5.1 集群配置

所有这些程序都运行在一个大约由 1800 台机器构成的集群上。每台机器配置 2 个 2G 主频、支持超线程的 Intel Xeon 处理器，4GB 的物理内存，两个 160GB 的 IDE 硬盘和一个千兆以太网卡。这些机器部署在一个两层的树形交换网络中，在 root 节点大概有 100-200GBPS 的传输带宽。所有这些机器都采用相同的部署（对等部署），因此任意两点之间的网络来回时间小于 1 毫秒。

在 4GB 内存里，大概有 1-1.5G 用于运行在集群上的其他任务。测试程序在周末下午开始执行，这时主机的 CPU、磁盘和网络基本上处于空闲状态。

5.2 GREP

这个分布式的 grep 程序需要扫描大概 10 的 10 次方个由 100 个字节组成的记录，查找出现概率较小的 3 个字符的模式（这个模式在 92337 个记录中出现）。输入数据被拆分成大约 64M 的 Block（M=15000），整个输出数据存放在一个文件中（R=1）。

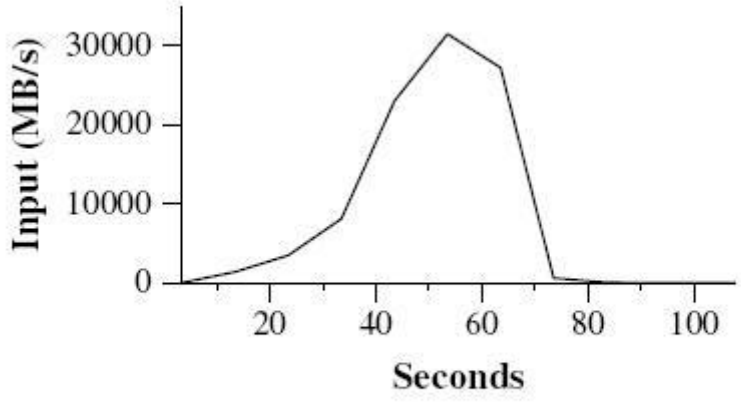


Figure 2: Data transfer rate over time

图 2 显示了这个运算随时间的处理过程。其中 Y 轴表示输入数据的处理速度。处理速度随着参与 MapReduce 计算的机器数量的增加而增加，当 1764 台 worker 参与计算的时，处理速度达到了 30GB/s。当 Map 任务结束的时候，即在计算开始后 80 秒，输入的处理速度降到 0。整个计算过程从开始到结束一共花了大概 150 秒。这包括了大约一分钟的初始启动阶段。初始启动阶段消耗的时间包括了是把这个程序传送到各个 worker 机器上的时间、等待 GFS 文件系统打开 1000 个输入文件集合的时间、获取相关的文件本地位置优化信息的时间。

5.3 排序

排序程序处理 10 的 10 次方个 100 个字节组成的记录（大概 1TB 的数据）。这个程序模仿 TeraSort benchmark[10]。

排序程序由不到 50 行代码组成。只有三行的 Map 函数从文本行中解析出 10 个字节的 key 值作为排序的 key，并且把这个 key 和原始文本行作为中间的 key/value pair 值输出。我们使用了一个内置的恒等函数作为 Reduce 操作函数。这个函数把中间的 key/value pair 值不作任何改变输出。最终排序结果输出到两路复制的 GFS 文件系统（也就是说，程序输出 2TB 的数据）。

如前所述，输入数据被分成 64MB 的 Block（M=15000）。我们把排序后的输出结果分区后存储到 4000 个文件（R=4000）。分区函数使用 key 的原始字节来把数据分区到 R 个片段中。

在这个 benchmark 测试中，我们使用的分区函数知道 key 的分区情况。通常对于排序程序来说，我们会增加一个预处理的 MapReduce 操作用于采样 key 值的分布情况，通过采样的数据来计算对最终排序处理的分区点。

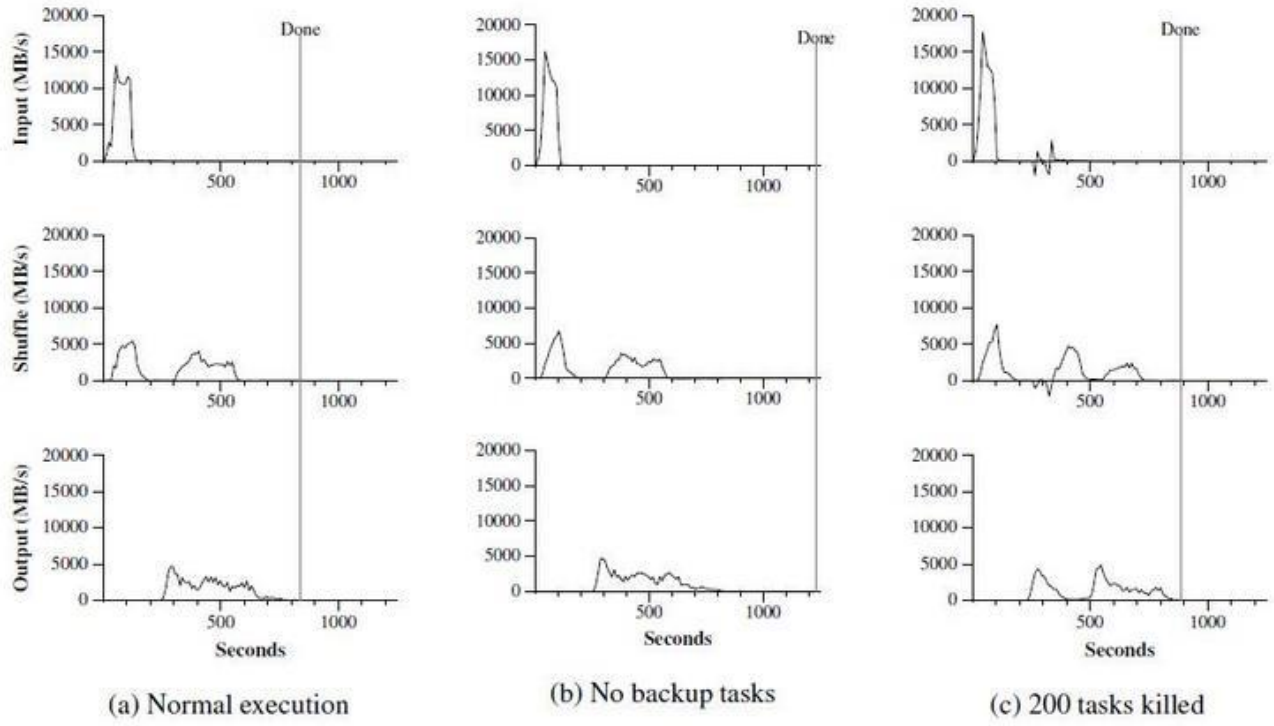


Figure 3: Data transfer rates over time for different executions of the sort program

图三（a）显示了这个排序程序的正常执行过程。左上的图显示了输入数据读取的速度。数据读取速度峰值会达到 13GB/s，并且所有 Map 任务完成之后，即大约 200 秒之后迅速滑落到 0。值得注意的是，排序程序输入数据读取速度小于分布式 grep 程序。这是因为排序程序的 Map 任务花了大约一半的处理时间和 I/O 带宽把中间输出结果写到本地硬盘。相应的分布式 grep 程序的中间结果输出几乎可以忽略不计。

左边中间的图显示了中间数据从 Map 任务发送到 Reduce 任务的网络速度。这个过程从第一个 Map 任务完成之后就开始缓慢启动了。图示的第一个高峰是启动了第一批大概 1700 个 Reduce 任务（整个 MapReduce 分布到大概 1700 台机器上，每台机器 1 次最多执行 1 个 Reduce 任务）。排序程序运行大约 300 秒后，第一批启动的 Reduce 任务有些完成了，我们开始执行剩下的 Reduce 任务。所有的处理在大约 600 秒后结束。

左下图表示 Reduce 任务把排序后的数据写到最终的输出文件的速度。在第一个排序阶段结束和数据开始写入磁盘之间有一个小的延时，这是因为 worker 机器正在忙于排序中间数据。磁盘写入速度在 2-4GB/s 持续一段时间。输出数据写入磁盘大约持续 850 秒。计入初始启动部分的时间，整个运算消耗了 891 秒。这个速度和 TeraSort benchmark[18]的最高纪录 1057 秒相差不多。

还有一些值得注意的现象：输入数据的读取速度比排序速度和输出数据写入磁盘速度要高不少，这是因为我们的输入数据本地化优化策略起了作用 — 绝大部分数据都是从本地硬盘读取的，从而节省了网络带宽。排序速度比输出数据写入到磁盘的速度快，这是因为输出数据写了两份（我们使用了 2 路的 GFS 文件系统，写入复制节点的原因是为了保证数据可靠性和可用性）。我们把输出数据写入到两个复制节点的原因是因为这是底层文件系统的保证数据可靠性和可用性的实现机制。如果底层文件系统使用类似容错编码[14](erasure coding)的方式而不是复制的方式保证数据的可靠性和可用性，那么在输出数据写入磁盘的时候，就可以降低网络带宽的使用。

5.4 高效的 backup 任务

图三（b）显示了关闭了备用任务后排序程序执行情况。执行的过程和图 3（a）很相似，除了输出数据写磁盘的动作在时间上拖了一个很长的尾巴，而且在这段时间里，几乎没有什么写入动作。在 960 秒后，只有 5 个 Reduce 任务没有完成。这些拖后腿的任务又执行了 300 秒才完成。整个计算消耗了 1283 秒，多了 44%的执行时间。

5.5 失效的机器

在图三（c）中演示的排序程序执行的过程中，我们在程序开始后几分钟有意的 kill 了 1746 个 worker 中的 200 个。集群底层的调度立刻在这些机器上重新开始新的 worker 处理进程（因为只是 worker 机器上的处理进程被 kill 了，机器本身还在工作）。

图三（c）显示出了一个“负”的输入数据读取速度，这是因为一些已经完成的 Map 任务丢失了（由于相应的执行 Map 任务的 worker 进程被 kill 了），需要重新执行这些任务。相关 Map 任务很快就被重新执行了。整个运算在 933 秒内完成，包括了初始启动时间（只比正常执行多消耗了 5%的时间）。

6 经验

我们在 2003 年 1 月完成了第一个版本的 MapReduce 库，在 2003 年 8 月的版本有了显著的增强，这包括了输入数据本地优化、worker 机器之间的动态负载均衡等等。从那以后，我们惊喜的发现，MapReduce 库能广泛应用于我们日常工作中遇到的各类问题。它现在在 Google 内部各个领域得到广泛应用，包括：

1. 大规模机器学习问题

2. Google News 和 Froogle 产品的集群问题
3. 从公众查询产品（比如 Google 的 Zeitgeist）的报告中抽取数据。
4. 从大量的新应用和新产品的网页中提取有用信息（比如，从大量的位置搜索网页中抽取地理位置信息）。
5. 大规模的图形计算。

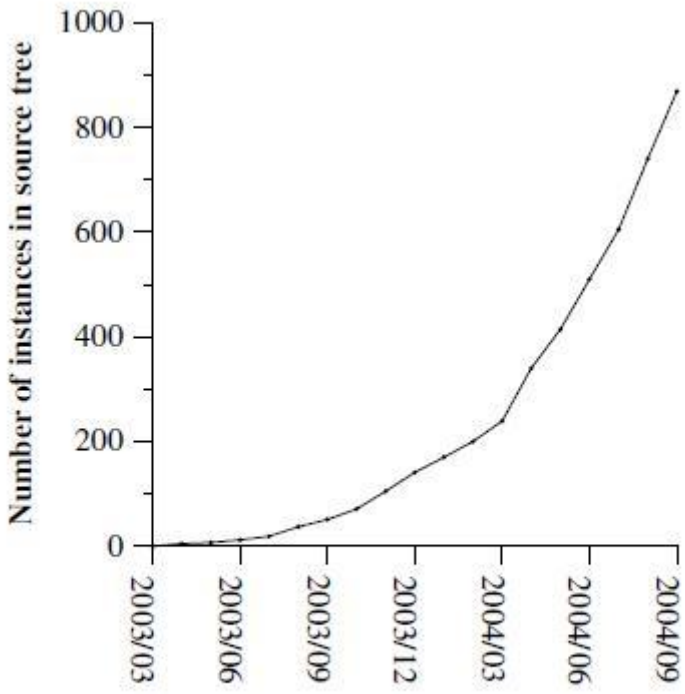


Figure 4: MapReduce instances over time

图四显示了在我们的源代码管理系统中，随着时间推移，独立的 MapReduce 程序数量的显著增加。从 2003 年早些时候的 0 个增长到 2004 年 9 月份的差不多 900 个不同的程序。MapReduce 的成功取决于采用 MapReduce 库能够在不到半个小时时间内写出一个简单的程序，这个简单的程序能够在上千台机器的组成的集群上做大规模并发处理，这极大的加快了开发和原形设计的周期。另外，采用 MapReduce 库，可以让完全没有分布式和/或并行系统开发经验的程序员很容易的利用大量的资源，开发出分布式和/或并行处理的应用。

任务数	29423
平均任务完成时间	634 秒
使用的机器时间	79,186 天
读取的输入数据	3,288TB
产生的中间数据	758TB
写出的输出数据	193TB
每个 job 平均 worker 机器数	157
每个 job 平均死掉 work 数	1.2
每个 job 平均 map 任务	3,351
每个 job 平均 reduce 任务	55
map 唯一实现	395
reduce 的唯一实现	296
map/reduce 的 combiner 实现	426

表 1：MapReduce2004 年 8 月的执行情况

在每个任务结束的时候，MapReduce 库统计计算资源的使用状况。在表 1，我们列出了 2004 年 8 月份 MapReduce 运行的任务所占用的相关资源。

6.1 大规模索引

到目前为止，MapReduce 最成功的应用就是重写了 Google 网络搜索服务所使用到的 index 系统。索引系统的输入数据是网络爬虫抓取回来的海量的文档，这些文档数据都保存在 GFS 文件系统里。这些文档原始内容[4]的大小超过了 20TB。索引程序是通过一系列的 MapReduce 操作（大约 5 到 10 次）来建立索引。使用 MapReduce（替换上一个特别设计的、分布式处理的索引程序）带来这些好处：

实现索引部分的代码简单、小巧、容易理解，因为对于容错、分布式以及并行计算的处理都是 MapReduce 库提供的。比如，使用 MapReduce 库，计算的代码行数从原来的 3800 行 C++代码减少到大概 700 行代码。

MapReduce 库的性能已经足够好了，因此我们可以把在概念上不相关的计算步骤分开处理，而不是混在一起以期减少数据传递的额外消耗。概念上不相关的计算步骤的隔离也使得我们可以很容易改变索引处理方式。比如，对之前的索引系统的一个小更改可能要耗费好几个月的时间，但是在使用 MapReduce 的新系统上，这样的更改只需要花几天时间就可以了。

索引系统的操作管理更容易了。因为由机器失效、机器处理速度缓慢、以及网络的瞬间阻塞等引起的绝大部分问题都已经由 MapReduce 库解决了，不再需要操作人员的介入了。另外，我们可以通过在索引系统集群中增加机器的简单方法提高整体处理性能。

7 相关工作

很多系统都提供了严格的编程模式，并且通过对编程的严格限制来实现并行计算。例如，一个结合函数

[4] raw contents，我认为就是网页中的剔除 html 标记后的内容、pdf 和 word 等有格式文档中提取的文本内容等

可以通过把 N 个元素的数组的前缀在 N 个处理器上使用并行前缀算法，在 log N 的时间内计算完[6，9，13][5]。MapReduce 可以看作是我们结合在真实环境下处理海量数据的经验，对这些经典模型进行简化和萃取的成果。更加值得骄傲的是，我们还实现了基于上千台处理器的集群的容错处理。相比而言，大部分并发处理系统都只在小规模的集群上实现，并且把容错处理交给了程序员。

Bulk Synchronous Programming[17]和一些 MPI 原语[11]提供了更高级别的并行处理抽象，可以更容易写出并行处理的程序。MapReduce 和这些系统的关键不同之处在于，MapReduce 利用限制性编程模式实现了用户程序的自动并发处理，并且提供了透明的容错处理。

我们数据本地优化策略的灵感来源于 active disks[12,15]等技术，在 active disks 中，计算任务是尽量推送到数据存储的节点处理[6]，这样就减少了网络和 IO 子系统的吞吐量。我们在挂载几个硬盘的普通机器上执行我们的运算，而不是在磁盘处理器上执行我们的工作，但是达到的目的一样的。

我们的备用任务机制和 Charlotte System[3]提出的 eager 调度机制比较类似。Eager 调度机制的一个缺点是如果一个任务反复失效，那么整个计算就不能完成。我们通过忽略引起故障的记录的方式在某种程度上解决了这个问题。

MapReduce 的实现依赖于一个内部的集群管理系统，这个集群管理系统负责在一个超大的、共享机器的集群上分布和运行用户任务。虽然这个不是本论文的重点，但是有必要提一下，这个集群管理系统在理念上和其它系统，如 Condor[16]是一样。

MapReduce 库的排序机制和 NOW-Sort[1]的操作上很类似。读取输入源的机器（map workers）把待排序的数据进行分区后，发送到 R 个 Reduce worker 中的一个进行处理。每个 Reduce worker 在本地对数据进行排序（尽可能在内存中排序）。当然，NOW-Sort 没有给用户自定义的 Map 和 Reduce 函数的机会，因此不具备 MapReduce 库广泛的实用性。

River[2]提供了一个编程模型：处理进程通过分布式队列传送数据的方式进行互相通讯。和 MapReduce 类似，River 系统尝试在不对等的硬件环境下，或者在系统颠簸的情况下也能提供近似平均的性能。River 是通过精心调度硬盘和网络的通讯来平衡任务的完成时间。MapReduce 库采用了其它的方法。通过对编程模型进行限制，MapReduce 框架把问题分解成为大量的“小”任务。这些任务在可用的 worker 集群上动态的调度，这样快速的 worker 就可以执行更多的任务。通过对编程模型进行限制，我们可用在工作接近完成的时候调度备用任务，缩短在硬件配置不均衡的情况下缩小整个操作完成的时间（比如有的机器性能差、或者机器被某些操作阻塞了）。

BAD-FS[5]采用了和 MapReduce 完全不同的编程模式，它是面向广域网（alex 注：wide-area network）的。

---

[5] 注：完全没有明白作者在说啥，具体参考相关 6、9、13 文档

[6] 即靠近数据源处理

不过，这两个系统有两个基础功能很类似。（1）两个系统采用重新执行的方式来防止由于失效导致的数据丢失。（2）两个都使用数据本地化调度策略，减少网络通讯的数据量。

TACC[7]是一个用于简化构造高可用性网络服务的系统。和 MapReduce 一样，它也依靠重新执行机制来实现的容错处理。

8 结束语

MapReduce 编程模型在 Google 内部成功应用于多个领域。我们把这种成功归结为几个方面：首先，由于 MapReduce 封装了并行处理、容错处理、数据本地化优化、负载均衡等等技术难点的细节，这使得 MapReduce 库易于使用。即便对于完全没有并行或者分布式系统开发经验的程序员而言；其次，大量不同类型的问题都可以通过 MapReduce 简单的解决。比如，MapReduce 用于生成 Google 的网络搜索服务所需要的数据、用来排序、用来数据挖掘、用于机器学习，以及很多其它的系统；第三，我们实现了一个在数千台计算机组成的大型集群上灵活部署运行的 MapReduce。这个实现使得有效利用这些丰富的计算资源变得非常简单，因此也适合用来解决 Google 遇到的其他很多需要大量计算的问题。

我们也从 MapReduce 开发过程中学到了不少东西。首先，约束编程模式使得并行和分布式计算非常容易，也易于构造容错的计算环境；其次，网络带宽是稀有资源。大量的系统优化是针对减少网络传输量为目的的：本地优化策略使大量的数据从本地磁盘读取，中间文件写入本地磁盘、并且只写一份中间文件也节约了网络带宽；第三，多次执行相同的任务可以减少性能缓慢的机器带来的负面影响（alex 注：即硬件配置的不平衡），同时解决了由于机器失效导致的数据丢失问题。

9 感谢

Josh Levenberg has been instrumental in revising and extending the user-level MapReduce API with a number of new features based on his experience with using MapReduce and other people’s suggestions for enhancements. MapReduce reads its input from and writes its output to the Google File System [8]. We would like to thank Mohit Aron, Howard Gobioff, Markus Gutschke, David Kramer, Shun-Tak Leung, and Josh Redstone for their work in developing GFS. We would also like to thank Percy Liang and Olcan Sercinoglu for their work in developing the cluster management system used by MapReduce. Mike Burrows, Wilson Hsieh, Josh Levenberg, Sharon Perl, Rob Pike, and Debby Wallach provided helpful comments on earlier drafts of this paper.The anonymous OSDI reviewers, and our shepherd, Eric Brewer, provided many useful suggestions of areas where the paper could be improved. Finally, we thank all the users of MapReduce within Google’s engineering organization for providing helpful feedback, suggestions, and bug reports.

10 参考资料

[1] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau,David E. Culler, Joseph M. Hellerstein, and David A. Patterson.High-performance sorting on networks of workstations.In Proceedings of the 1997 ACM SIGMOD InternationalConference on Management of Data, Tucson,Arizona, May 1997.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, NoahTreuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River:Making the fast case common. In Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS ’99), pages 10.22, Atlanta, Georgia, May 1999.

[3] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems, 1996. [4] Luiz A. Barroso, Jeffrey Dean, and Urs H ̈olzle. Web search for a planet: The Google cluster architecture. IEEE Micro, 23(2):22.28, April 2003.

[5] John Bent, Douglas Thain, Andrea C.Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed file system. In Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI, March 2004.

[6] Guy E. Blelloch. Scans as primitive parallel operations.IEEE Transactions on Computers, C-38(11), November 1989.

[7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In Proceedings of the 16th ACM Symposium on Operating System Principles, pages 78. 91, Saint-Malo, France, 1997.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In 19th Symposium on Operating Systems Principles, pages 29.43, Lake George, New York, 2003. To appear in OSDI 2004 12

[9] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, Euro-Par’96. Parallel Processing, Lecture Notes in Computer Science 1124, pages 401.408. Springer-Verlag, 1996.

[10] Jim Gray. Sort benchmark home page. http://research.microsoft.com/barc/SortBenchmark/.

[11] William Gropp, Ewing Lusk, and Anthony Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge, MA, 1999.

[12] L. Huston, R. Sukthankar, R.Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In Proceedings of the 2004

yore

USENIX File and Storage Technologies FAST Conference, April 2004.

[13] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. Journal of the ACM, 27(4):831.838, 1980.

[14] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. Journal of the ACM, 36(2):335.348, 1989.

[15] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. IEEE Computer, pages 68.74, June 2001.

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. Concurrency and Computation: Practice and Experience, 2004.

[17] L. G. Valiant. A bridging model for parallel computation. Communications of the ACM, 33(8):103.111, 1997.

[18] Jim Wyllie. Spsort: How to sort a terabyte quickly. http://alme1.almaden.ibm.com/cs/spsort.pdf.

11 附录 A-单词频率统计

本节包含了一个完整的程序，用于统计在一组命令行指定的输入文件中，每一个不同的单词出现频率。

```
#include “mapreduce/mapreduce.h”
// User’s map function
class WordCounter : public Mapper {
  public:
    virtual void Map(const MapInput& input) {
      const string& text = input.value();
      const int n = text.size();
      for (int i = 0; i < n; ) {
        // Skip past leading whitespace
        while ((i < n) && isspace(text[i]))
          i++;

        // Find word end
        int start = i;
```

```
      while ((i < n) && !isspace(text[i]))
        i++;
      if (start < i)
        Emit(text.substr(start,i-start),” 1″ );
    }
  }
};

REGISTER_MAPPER(WordCounter);

// User’s reduce function
class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }

    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};

REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);
```

```
MapReduceSpecification spec;

// Store list of input files into "spec"
for (int i = 1; i < argc; i++) {
  MapReduceInput* input = spec.add_input();
  input->set_format("text");
  input->set_filepattern(argv[i]);
  input->set_mapper_class("WordCounter");
}

// Specify the output files:
// /gfs/test/freq-00000-of-00100
// /gfs/test/freq-00001-of-00100
// …
MapReduceOutput* out = spec.output();
out->set_filebase("/gfs/test/freq");
out->set_num_tasks(100);
out->set_format("text");
out->set_reducer_class("Adder");

// Optional: do partial sums within map
// tasks to save network bandwidth
out->set_combiner_class("Adder");

// Tuning parameters: use at most 2000
// machines and 100 MB of memory per task
spec.set_machines(2000);
spec.set_map_megabytes(100);
spec.set_reduce_megabytes(100);
```

```
  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();

  // Done: 'result' structure contains info
  // about counters, time taken, number of
  // machines used, etc.
  return 0;
}
```

Google Bigtable 中文版[1]

1 摘要

Bigtable 是一个分布式的结构化数据存储系统，它被设计用来处理海量数据：通常是分布在数千台普通服务器上的 PB 级的数据。

Google 的很多项目使用 Bigtable 存储数据，包括 Web 索引、Google Earth、Google Finance。这些应用对 Bigtable 提出的要求差异非常大，无论是在数据量上（从 URL 到网页到卫星图像）还是在响应速度上（从后端的批量处理到实时数据服务）。尽管应用需求差异很大，但是，针对 Google 的这些产品，Bigtable 还是成功的提供了一个灵活的、高性能的解决方案。

本论文描述了 Bigtable 提供的简单的数据模型。利用这个模型，用户可以动态的控制数据的分布和格式。我们还将描述 Bigtable 的设计和实现。

2 介绍

在过去两年半时间里，我们设计、实现并部署了一个分布式的结构化数据存储系统 — 在 Google，我们称之为 Bigtable。Bigtable 的设计目的是可靠的处理 PB 级别的数据，并且能够部署到上千台机器上。Bigtable 已经实现了下面的几个目标：**适用性广泛、可扩展、高性能和高可用性**。

Bigtable 已经在超过 60 个 Google 的产品和项目上得到了应用，包括 Google Analytics、Google Finance、Orkut、Personalized Search、Writely 和 Google Earth。这些产品对 Bigtable 提出了迥异的需求，有的需要高吞吐量的批处理，有的则需要及时响应，快速返回数据给最终用户。它们使用的 Bigtable 集群的配置也有很大的差异，有的集群只有几台服务器，而有的则需要上千台服务器、存储几百 TB 的数据。

在很多方面，Bigtable 和数据库很类似：它使用了很多数据库的实现策略。并行数据库【14】和内存数据库【13】已经具备可扩展性和高性能，但是 Bigtable 提供了一个和这些系统完全不同的接口。Bigtable 不支持完整的关系数据模型；与之相反，Bigtable 为客户提供了简单的数据模型，利用这个模型，客户可以动态控制数据的分布和格式[2]，用户也可以自己推测[3]底层存储数据的位置相关性[4]。数据的下标是行和列的名字，名字可以是任意的字符串。Bigtable 将存储的数据都视为字符串，但是 Bigtable 本身不去解析这些字符串，客户

---

[1] 译者 alex，原文地址 http://blademaster.ixiezi.com/

[2] 也就是对 BigTable 而言，数据是没有格式的。用数据库领域的术语说，就是数据没有 Schema，用户自己去定义 Schema

[3] 英文为 reason about

[4] 位置相关性可以这样理解，比如树状结构，具有相同前缀的数据的存放位置接近。在读取的时候，可以把这些数据一次读取出来

程序通常会在把各种结构化或者半结构化的数据串行化到这些字符串里。通过仔细选择数据的模式，客户可以控制数据的位置相关性。最后，可以通过 BigTable 的模式参数来控制数据是存放在内存中、还是硬盘上。

第 3 节描述关于数据模型更多细节方面的东西；

第 4 节概要介绍了客户端 API；

第 5 节简要介绍了 BigTable 底层使用的 Google 的基础框架；

第 6 节描述了 BigTable 实现的关键部分；

第 7 节描述了我们为了提高 BigTable 的性能采用的一些精细的调优方法；

第 8 节提供了 BigTable 的性能数据；

第 9 节讲述了几个 Google 内部使用 BigTable 的例子；

第 10 节是我们在设计和后期支持过程中得到一些经验和教训；

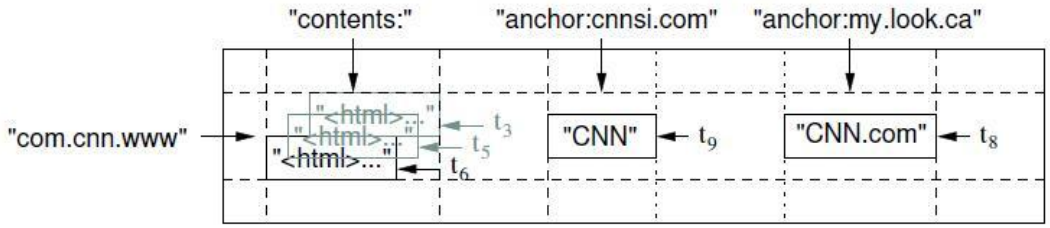
最后，在第 11 节列出我们的相关研究工作，第 12 节是我们的结论。

3 数据模型

Bigtable 是一个稀疏的、分布式的、持久化存储的多维度排序 Map[5]。Map 的索引是行关键字、列关键字以及时间戳；Map 中的每个 value 都是一个未经解析的 byte 数组。

```
(row:string, column:string,time:int64)->string
```

我们在仔细分析了一个类似 Bigtable 的系统的种种潜在用途之后，决定使用这个数据模型。我们先举个具体的例子，这个例子促使我们做了很多设计决策；假设我们想要存储海量的网页及相关信息，这些数据可以用于很多不同的项目，我们姑且称这个特殊的表为 Webtable。在 Webtable 里，我们使用 URL 作为行关键字，使用网页的某些属性作为列名，网页的内容存在“contents:”列中，并用获取该网页的时间戳作为标识[6]，如图一所示。



图表 1 一个存储 Web 网页的例子的表的片断

[5] Map 由 key 和 value 组成，后面我们直接使用 key 和 value，不再另外翻译了

[6] 即按照获取时间不同，存储了多个版本的网页数据

行名是一个反向 URL。contents 列族存放的是网页的内容，anchor 列族存放引用该网页的锚链接文本[7]。CNN 的主页被 Sports Illustrator 和 MY-look 的主页引用，因此该行包含了名为“anchor:cnnsi.com”和“anchhor:my.look.ca”的列。每个锚链接只有一个版本[8]；而 contents 列则有三个版本，分别由时间戳 t3，t5，和 t6 标识。

3.1 行

表中的行关键字可以是任意的字符串（目前支持最大 64KB 的字符串，但是对大多数用户，10-100 个字节就足够了）。对同一个行关键字的读或者写操作都是原子的（不管读或者写这一行里多少个不同列），这个设计决策能够使用户很容易的理解程序在对同一个行进行并发更新操作时的行为。

Bigtable 通过行关键字的字典顺序来组织数据。表中的每个行都可以动态分区。每个分区叫做一个”Tablet”，Tablet 是数据分布和负载均衡调整的最小单位。这样做的结果是，当操作只读取行中很少几列的数据时效率很高，通常只需要很少几次机器间的通信即可完成。用户可以通过选择合适的行关键字，在数据访问时有效利用数据的位置相关性，从而更好的利用这个特性。举例来说，在 Webtable 里，通过反转 URL 中主机名的方式，可以把同一个域名下的网页聚集起来组织成连续的行。具体来说，我们可以把 maps.google.com/index.html 的数据存放在关键字 com.google.maps/index.html 下。把相同的域中的网页存储在连续的区域可以让基于主机和域名的分析更加有效。

3.2 列族

列关键字组成的集合叫做“列族“，列族是访问控制的基本单位。存放在同一列族下的所有数据通常都属于同一个类型（我们可以把同一个列族下的数据压缩在一起）。列族在使用之前必须先创建，然后才能在列族中任何的列关键字下存放数据；列族创建后，其中的任何一个列关键字下都可以存放数据。根据我们的设计意图，一张表中的列族不能太多（最多几百个），并且列族在运行期间很少改变。与之相对应的，一张表可以有无限多个列。

列关键字的命名语法如下：列族：限定词。 列族的名字必须是可打印的字符串，而限定词的名字可以是任意的字符串。比如，Webtable 有个列族 language，language 列族用来存放撰写网页的语言。我们在 language 列族中只使用一个列关键字，用来存放每个网页的语言标识 ID。Webtable 中另一个有用的列族是 anchor；这个列族的每一个列关键字代表一个锚链接，如图一所示。Anchor 列族的限定词是引用该网页的站点名；Anchor 列族每列的数据项存放的是链接文本。

---

[7] HTML 的 Anchor

[8] 注意时间戳标识了列的版本，t9 和 t8 分别标识了两个锚链接的版本

访问控制、磁盘和内存的使用统计都是在列族层面进行的。在我们的 Webtable 的例子中，上述的控制权限能帮助我们管理不同类型的应用：我们允许一些应用可以添加新的基本数据、一些应用可以读取基本数据并创建继承的列族、一些应用则只允许浏览数据（甚至可能因为隐私的原因不能浏览所有数据）。

3.3 时间戳

在 Bigtable 中，表的每一个数据项都可以包含同一份数据的不同版本；不同版本的数据通过时间戳来索引。Bigtable 时间戳的类型是 64 位整型。Bigtable 可以给时间戳赋值，用来表示精确到毫秒的“实时”时间；用户程序也可以给时间戳赋值。如果应用程序需要避免数据版本冲突，那么它必须自己生成具有唯一性的时间戳。数据项中，不同版本的数据按照时间戳倒序排序，即最新的数据排在最前面。

为了减轻多个版本数据的管理负担，我们对每一个列族配有两个设置参数，Bigtable 通过这两个参数可以对废弃版本的数据自动进行垃圾收集。用户可以指定只保存最后 n 个版本的数据，或者只保存“足够新”的版本的数据（比如，只保存最近 7 天的内容写入的数据）。

在 Webtable 的举例里，contents:列存储的时间戳信息是网络爬虫抓取一个页面的时间。上面提及的垃圾收集机制可以让我们只保留最近三个版本的网页数据。

4 API

Bigtable 提供了建立和删除表以及列族的 API 函数。Bigtable 还提供了修改集群、表和列族的元数据的 API，比如修改访问权限。

```
 // Open the table
Table *T = OpenOrDie(“/bigtable/web/webtable”);
// Write a new anchor and delete an old anchor
RowMutation r1(T, “com.cnn.www”);
r1.Set(“anchor:www.c-span.org”, “CNN”);
r1.Delete(“anchor:www.abc.com”);
Operation op;
Apply(&op, &r1);
```

图表 2 Writing to Bigtable.

客户程序可以对 Bigtable 进行如下的操作：写入或者删除 Bigtable 中的值、从每个行中查找值、或者遍历表中的一个数据子集。图 2 中的 C++代码使用 RowMutation 抽象对象进行了一系列的更新操作。（为了保持示例代码的简洁，我们忽略了一些细节相关代码）。调用 Apply 函数对 Webtable 进行了一个原子修改操作：它

为 www.cnn.com 增加了一个锚点，同时删除了另外一个锚点。

```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
    printf("%s %s %lld %s\n",
    scanner.RowName(),
    stream->ColumnName(),
    stream->MicroTimestamp(),
    stream->Value());
}
```

图表 3 Reading from Bigtable.

图 3 中的 C++代码使用 Scanner 抽象对象遍历一个行内的所有锚点。客户程序可以遍历多个列族，有几种方法可以对扫描输出的行、列和时间戳进行限制。例如，我们可以限制上面的扫描，让它只输出那些匹配正则表达式*.cnn.com 的锚点，或者那些时间戳在当前时间前 10 天的锚点。

Bigtable 还支持一些其它的特性，利用这些特性，用户可以对数据进行更复杂的处理。首先，Bigtable 支持单行上的事务处理，利用这个功能，用户可以对存储在一个行关键字下的数据进行原子性的读-更新-写操作。虽然 Bigtable 提供了一个允许用户跨行批量写入数据的接口，但是，Bigtable 目前还不支持通用的跨行事务处理。其次，Bigtable 允许把数据项用做整数计数器。最后，Bigtable 允许用户在服务器的地址空间内执行脚本程序。脚本程序使用 Google 开发的 Sawzall【28】数据处理语言。虽然目前我们基于的 Sawzall 语言的 API 函数还不允许客户的脚本程序写入数据到 Bigtable，但是它允许多种形式的数据转换、基于任意表达式的数据过滤、以及使用多种操作符的进行数据汇总。

Bigtable 可以和 MapReduce【12】一起使用，MapReduce 是 Google 开发的大规模并行计算框架。我们已经开发了一些 Wrapper 类，通过使用这些 Wrapper 类，Bigtable 可以作为 MapReduce 框架的输入和输出。

5 BigTable 构件

Bigtable 是建立在其它的几个 Google 基础构件上的。BigTable 使用 Google 的分布式文件系统(GFS)【17】存储日志文件和数据文件。BigTable 集群通常运行在一个共享的机器池中，池中的机器还会运行其它的各种

各样的分布式应用程序，BigTable 的进程经常要和其它应用的进程共享机器。BigTable 依赖集群管理系统来调度任务、管理共享的机器上的资源、处理机器的故障、以及监视机器的状态。

BigTable 内部存储数据的文件是 Google SSTable 格式的。SSTable 是一个持久化的、排序的、不可更改的 Map 结构，而 Map 是一个 key-value 映射的数据结构，key 和 value 的值都是任意的 Byte 串。可以对 SSTable 进行如下的操作：查询与一个 key 值相关的 value，或者遍历某个 key 值范围内的所有的 key-value 对。从内部看，SSTable 是一系列的数据块（通常每个块的大小是 64KB，这个大小是可以配置的）。SSTable 使用块索引（通常存储在 SSTable 的最后）来定位数据块；在打开 SSTable 的时候，索引被加载到内存。每次查找都可以通过一次磁盘搜索完成：首先使用二分查找法在内存中的索引里找到数据块的位置，然后再从硬盘读取相应的数据块。也可以选择把整个 SSTable 都放在内存中，这样就不必访问硬盘了。

BigTable 还依赖一个高可用的、序列化的分布式锁服务组件，叫做 Chubby【8】。一个 Chubby 服务包括了 5 个活动的副本，其中的一个副本被选为 Master，并且处理请求。只有在大多数副本都是正常运行的，并且彼此之间能够互相通信的情况下，Chubby 服务才是可用的。当有副本失效的时候，Chubby 使用 Paxos 算法【9,23】来保证副本的一致性。Chubby 提供了一个名字空间，里面包括了目录和小文件。每个目录或者文件可以当成一个锁，读写文件的操作都是原子的。Chubby 客户程序库提供对 Chubby 文件的一致性缓存。每个 Chubby 客户程序都维护一个与 Chubby 服务的会话。如果客户程序不能在租约到期的时间内重新签订会话的租约，这个会话就过期失效了[9]。当一个会话失效时，它拥有的锁和打开的文件句柄都失效了。Chubby 客户程序可以在文件和目录上注册回调函数，当文件或目录改变、或者会话过期时，回调函数会通知客户程序。

Bigtable 使用 Chubby 完成以下的几个任务：

1. 确保在任何给定的时间内最多只有一个活动的 Master 副本；
2. 存储 BigTable 数据的自引导指令的位置（参考 5.1 节）；
3. 查找 Tablet 服务器，以及在 Tablet 服务器失效时进行善后（5.2 节）；
4. 存储 BigTable 的模式信息（每张表的列族信息）；
5. 以及存储访问控制列表。

如果 Chubby 长时间无法访问，BigTable 就会失效。最近我们在使用 11 个 Chubby 服务实例的 14 个 BigTable 集群上测量了这个影响。由于 Chubby 不可用而导致 BigTable 中的部分数据不能访问的平均比率是 0.0047%（Chubby 不能访问的原因可能是 Chubby 本身失效或者网络问题）。单个集群里，受 Chubby 失效影响最大的百分比是 0.0326%[10]。

[9] 又用到了 lease。原文是：A client’s session expires if it is unable to renew its session lease within the lease expiration time.

[10] 原文是：The percentage for the single cluster that was most affected by Chubby unavailability was 0.0326%.

6 介绍

Bigtable 包括了三个主要的组件：链接到客户程序中的库、一个 Master 服务器和多个 Tablet 服务器。针对系统工作负载的变化情况，BigTable 可以动态的向集群中添加（或者删除）Tablet 服务器。

Master 服务器主要负责以下工作：为 Tablet 服务器分配 Tablets、检测新加入的或者过期失效的 Table 服务器、对 Tablet 服务器进行负载均衡、以及对保存在 GFS 上的文件进行垃圾收集。除此之外，它还处理对模式的相关修改操作，例如建立表和列族。

每个 Tablet 服务器都管理一个 Tablet 的集合（通常每个服务器有大约数十个至上千个 Tablet）。每个 Tablet 服务器负责处理它所加载的 Tablet 的读写操作，以及在 Tablets 过大时，对其进行分割。

和很多 Single-Master 类型的分布式存储系统【17.21】类似，客户端读取的数据都不经过 Master 服务器：客户程序直接和 Tablet 服务器通信进行读写操作。由于 BigTable 的客户程序不必通过 Master 服务器来获取 Tablet 的位置信息，因此，大多数客户程序甚至完全不需要和 Master 服务器通信。在实际应用中，Master 服务器的负载是很轻的。

一个 BigTable 集群存储了很多表，每个表包含了一个 Tablet 的集合，而每个 Tablet 包含了某个范围内的行的所有相关数据。初始状态下，一个表只有一个 Tablet。随着表中数据的增长，它被自动分割成多个 Tablet，缺省情况下，每个 Tablet 的尺寸大约是 100MB 到 200MB。

6.1 Tablet 的位置

我们使用一个三层的、类似 B+树[10]的结构存储 Tablet 的位置信息(如图 4)。

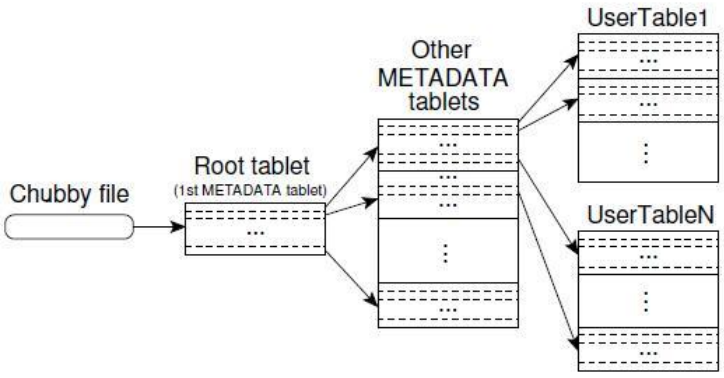


Figure 4: Tablet location hierarchy.

第一层是一个存储在 Chubby 中的文件，它包含了 Root Tablet 的位置信息。Root Tablet 包含了一个特殊的 METADATA 表里所有的 Tablet 的位置信息。METADATA 表的每个 Tablet 包含了一个用户 Tablet 的集合。Root Tablet 实际上是 METADATA 表的第一个 Tablet，只不过对它的处理比较特殊 — Root Tablet 永远不会被分割 — 这就保证了 Tablet 的位置信息存储结构不会超过三层。

在 METADATA 表里面，每个 Tablet 的位置信息都存放在一个行关键字下面，而这个行关键字是由 Tablet 所在的表的标识符和 Tablet 的最后一行编码而成的。METADATA 的每一行都存储了大约 1KB 的内存数据。在一个大小适中的、容量限制为 128MB 的 METADATA Tablet 中，采用这种三层结构的存储模式，可以标识 2^34 个 Tablet 的地址（如果每个 Tablet 存储 128MB 数据，那么一共可以存储 2^61 字节数据）。

客户程序使用的库会缓存 Tablet 的位置信息。如果客户程序没有缓存某个 Tablet 的地址信息，或者发现它缓存的地址信息不正确，客户程序就在树状的存储结构中递归的查询 Tablet 位置信息；如果客户端缓存是空的，那么寻址算法需要通过三次网络来回通信寻址，这其中包括了一次 Chubby 读操作；如果客户端缓存的地址信息过期了，那么寻址算法可能需要最多 6 次网络来回通信才能更新数据，因为只有在缓存中没有查到数据的时候才能发现数据过期[11]。尽管 Tablet 的地址信息是存放在内存里的，对它的操作不必访问 GFS 文件系统，但是，通常我们会通过预取 Tablet 地址来进一步的减少访问的开销：每次需要从 METADATA 表中读取一个 Tablet 的元数据的时候，它都会多读取几个 Tablet 的元数据。

在 METADATA 表中还存储了次级信息[12]，包括每个 Tablet 的事件日志（例如，什么时候一个服务器开始为该 Tablet 提供服务）。这些信息有助于排查错误和性能分析。

6.2 Tablet 分配

在任何一个时刻，一个 Tablet 只能分配给一个 Tablet 服务器。Master 服务器记录了当前有哪些活跃的 Tablet 服务器、哪些 Tablet 分配给了哪些 Tablet 服务器、哪些 Tablet 还没有被分配。当一个 Tablet 还没有被分配、并且刚好有一个 Tablet 服务器有足够的空闲空间装载该 Tablet 时，Master 服务器会给这个 Tablet 服务器发送一个装载请求，把 Tablet 分配给这个服务器。

BigTable 使用 Chubby 跟踪记录 Tablet 服务器的状态。当一个 Tablet 服务器启动时，它在 Chubby 的一个指定目录下建立一个有唯一性名字的文件，并且获取该文件的独占锁。Master 服务器实时监控着这个目录（服务器目录），因此 Master 服务器能够知道有新的 Tablet 服务器加入了。如果 Tablet 服务器丢失了 Chubby 上的独占锁 — 比如由于网络断开导致 Tablet 服务器和 Chubby 的会话丢失 — 它就停止对 Tablet 提供服务。（Chubby 提供了一种高效的机制，利用这种机制，Tablet 服务器能够在不增加网络负担的情况下知道它是否还持有锁）。只要文件还存在，Tablet 服务器就会试图重新获得对该文件的独占锁；如果文件不存在了，那么 Tablet 服务器就不能再提供服务了，它会自行退出[13]。当 Tablet 服务器终止时（比如，集群的管理系统将运行该 Tablet 服务器的主机从集群中移除），它会尝试释放它持有的文件锁，这样一来，Master 服务器就能尽快把 Tablet 分配到其它的 Tablet 服务器。

---

[11] 其中的三次通信发现缓存过期，另外三次更新缓存数据）(假设 METADATA 的 Tablet 没有被频繁的移动

[12] secondary information

[13] so it kills itself

Master 服务器负责检查一个 Tablet 服务器是否已经不再为它的 Tablet 提供服务了，并且要尽快重新分配它加载的 Tablet。Master 服务器通过轮询 Tablet 服务器文件锁的状态来检测何时 Tablet 服务器不再为 Tablet 提供服务。如果一个 Tablet 服务器报告它丢失了文件锁，或者 Master 服务器最近几次尝试和它通信都没有得到响应，Master 服务器就会尝试获取该 Tablet 服务器文件的独占锁；如果 Master 服务器成功获取了独占锁，那么就说明 Chubby 是正常运行的，而 Tablet 服务器要么是宕机了、要么是不能和 Chubby 通信了，因此，Master 服务器就删除该 Tablet 服务器在 Chubby 上的服务器文件以确保它不再给 Tablet 提供服务。一旦 Tablet 服务器在 Chubby 上的服务器文件被删除了，Master 服务器就把之前分配给它的所有的 Tablet 放入未分配的 Tablet 集合中。为了确保 Bigtable 集群在 Master 服务器和 Chubby 之间网络出现故障的时候仍然可以使用，Master 服务器在它的 Chubby 会话过期后主动退出。但是不管怎样，如同我们前面所描述的，Master 服务器的故障不会改变现有 Tablet 在 Tablet 服务器上的分配状态。

当集群管理系统启动了一个 Master 服务器之后，Master 服务器首先要了解当前 Tablet 的分配状态，之后才能够修改分配状态。Master 服务器在启动的时候执行以下步骤：

1. Master 服务器从 Chubby 获取一个唯一的 Master 锁，用来阻止创建其它的 Master 服务器实例；
2. Master 服务器扫描 Chubby 的服务器文件锁存储目录，获取当前正在运行的服务器列表；
3. Master 服务器和所有的正在运行的 Tablet 表服务器通信，获取每个 Tablet 服务器上 Tablet 的分配信息；
4. Master 服务器扫描 METADATA 表获取所有的 Tablet 的集合。

在扫描的过程中，当 Master 服务器发现了一个还没有分配的 Tablet，Master 服务器就将这个 Tablet 加入未分配的 Tablet 集合等待合适的时机分配。

可能会遇到一种复杂的情况：在 METADATA 表的 Tablet 还没有被分配之前是不能够扫描它的。因此，在开始扫描之前（步骤 4），如果在第三步的扫描过程中发现 Root Tablet 还没有分配，Master 服务器就把 Root Tablet 加入到未分配的 Tablet 集合。这个附加操作确保了 Root Tablet 会被分配。由于 Root Tablet 包括了所有 METADATA 的 Tablet 的名字，因此 Master 服务器扫描完 Root Tablet 以后，就得到了所有的 METADATA 表的 Tablet 的名字了。

保存现有 Tablet 的集合只有在以下事件发生时才会改变：建立了一个新表或者删除了一个旧表、两个 Tablet 被合并了、或者一个 Tablet 被分割成两个小的 Tablet。Master 服务器可以跟踪记录所有这些事件，因为除了最后一个事件外的两个事件都是由它启动的。Tablet 分割事件需要特殊处理，因为它是由 Tablet 服务器启动。在分割操作完成之后，Tablet 服务器通过在 METADATA 表中记录新的 Tablet 的信息来提交这个操作；当分割操作提交之后，Tablet 服务器会通知 Master 服务器。如果分割操作已提交的信息没有通知到 Master 服务器（可能两个服务器中有一个宕机了），Master 服务器在要求 Tablet 服务器装载已经被分割的子表的时候会发

现一个新的 Tablet。通过对比 METADATA 表中 Tablet 的信息，Tablet 服务器会发现 Master 服务器要求其装载的 Tablet 并不完整，因此，Tablet 服务器会重新向 Master 服务器发送通知信息。

6.3 Tablet 服务

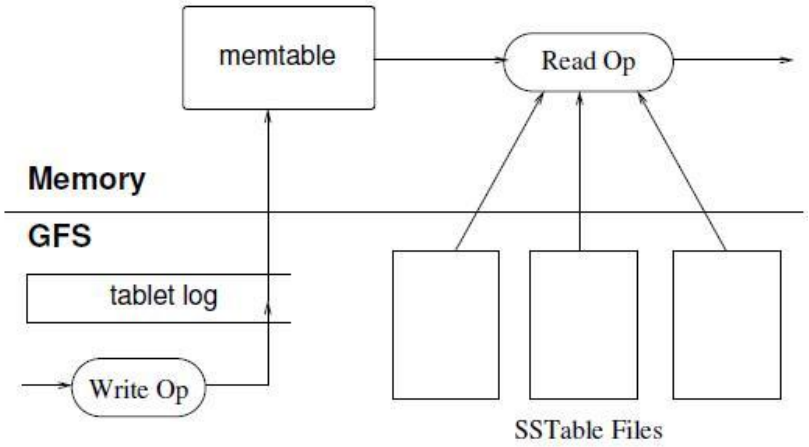


Figure 5: Tablet Representation

如图 5 所示，Tablet 的持久化状态信息保存在 GFS 上。更新操作提交到 REDO 日志中[14]。在这些更新操作中，最近提交的那些存放在一个排序的缓存中，我们称这个缓存为 memtable；较早的更新存放在一系列 SSTable 中。为了恢复一个 Tablet，Tablet 服务器首先从 METADATA 表中读取它的元数据。Tablet 的元数据包含了组成这个 Tablet 的 SSTable 的列表，以及一系列的 Redo Point[15]，这些 Redo Point 指向可能含有该 Tablet 数据的已提交的日志记录。Tablet 服务器把 SSTable 的索引读进内存，之后通过重复 Redo Point 之后提交的更新来重建 memtable。

当对 Tablet 服务器进行写操作时，Tablet 服务器首先要检查这个操作格式是否正确、操作发起者是否有执行这个操作的权限。权限验证的方法是通过从一个 Chubby 文件里读取出来的具有写权限的操作者列表来进行验证（这个文件几乎一定会存放在 Chubby 客户缓存里）。成功的修改操作会记录在提交日志里。可以采用批量提交方式[16]来提高包含大量小的修改操作的应用程序的吞吐量【13，16】。当一个写操作提交后，写的内容插入到 memtable 里面。

当对 Tablet 服务器进行读操作时，Tablet 服务器会作类似的完整性和权限检查。一个有效的读操作在一个由一系列 SSTable 和 memtable 合并的视图里执行。由于 SSTable 和 memtable 是按字典排序的数据结构，因此可以高效生成合并视图。

---

[14] 原文：Updates are committed to a commit log that stores redo records

[15] 原文：a set of redo points

[16] 原文：group commit

当进行 Tablet 的合并和分割时，正在进行的读写操作能够继续进行。

6.4 空间收缩[17]

随着写操作的执行，memtable 的大小不断增加。当 memtable 的尺寸到达一个门限值的时候，这个 memtable 就会被冻结，然后创建一个新的 memtable；被冻结住 memtable 会被转换成 SSTable，然后写入 GFS[18]。Minor Compaction 过程有两个目的：shrink [19]Tablet 服务器使用的内存，以及在服务器灾难恢复过程中，减少必须从提交日志里读取的数据量。在 Compaction 过程中，正在进行的读写操作仍能继续。

每一次 Minor Compaction 都会创建一个新的 SSTable。如果 Minor Compaction 过程不停滞的持续进行下去，读操作可能需要合并来自多个 SSTable 的更新；否则，我们通过定期在后台执行 Merging Compaction 过程合并文件，限制这类文件的数量。Merging Compaction 过程读取一些 SSTable 和 memtable 的内容，合并成一个新的 SSTable。只要 Merging Compaction 过程完成了，输入的这些 SSTable 和 memtable 就可以删除了。

合并所有的 SSTable 并生成一个新的 SSTable 的 Merging Compaction 过程叫作 Major Compaction。由非 Major Compaction 产生的 SSTable 可能含有特殊的删除条目，这些删除条目能够隐藏在旧的、但是依然有效的 SSTable 中已经删除的数据[20]。而 Major Compaction 过程生成的 SSTable 不包含已经删除的信息或数据。Bigtable 循环扫描它所有的 Tablet，并且定期对它们执行 Major Compaction。Major Compaction 机制允许 Bigtable 回收已经删除的数据占有的资源，并且确保 BigTable 能及时清除已经删除的数据[21]，这对存放敏感数据的服务是非常重要。

7 优化

上一章我们描述了 Bigtable 的实现，我们还需要很多优化工作才能使 Bigtable 到达用户要求的高性能、高可用性和高可靠性。本章描述了 Bigtable 实现的其它部分，为了更好的强调这些优化工作，我们将深入细节。

7.1 局部性群组

客户程序可以将多个列族组合成一个局部性群族。对 Tablet 中的每个局部性群组都会生成一个单独的

---

[17] Compactions 这个词挺简单，但是在这节里面挺难翻译的。应该是空间缩减的意思，但是似乎又不能完全概括它在上下文中的意思，在下文中不进行翻译。

[18] 我们称这种 Compaction 行为为 Minor Compaction

[19] shrink 是数据库用语，表示空间收缩

[20] 令人费解啊，原文是 SSTables produced by non-major compactions can contain special deletion entries that suppress deleted data in older SSTables that are still live.

[21] 实际是回收资源。数据删除后，它占有的空间并不能马上重复利用；只有空间回收后才能重复使用。

SSTable。将通常不会一起访问的列族分割成不同的局部性群组可以提高读取操作的效率。例如，在 Webtable 表中，网页的元数据（比如语言和 Checksum）可以在一个局部性群组中，网页的内容可以在另外一个群组：当一个应用程序要读取网页的元数据的时候，它没有必要去读取所有的页面内容。

此外，可以以局部性群组为单位设定一些有用的调试参数。比如，可以把一个局部性群组设定为全部存储在内存中。Tablet 服务器依照惰性加载的策略将设定为放入内存的局部性群组的 SSTable 装载进内存。加载完成之后，访问属于该局部性群组的列族的时候就不必读取硬盘了。这个特性对于需要频繁访问的小块数据特别有用：在 Bigtable 内部，我们利用这个特性提高 METADATA 表中具有位置相关性的列族的访问速度。

7.2 压缩

客户程序可以控制一个局部性群组的 SSTable 是否需要压缩；如果需要压缩，那么以什么格式来压缩。每个 SSTable 的块（块的大小由局部性群组的优化参数指定）都使用用户指定的压缩格式来压缩。虽然分块压缩浪费了少量空间[22]，但是，我们在只读取 SSTable 的一小部分数据的时候就不必解压整个文件了。很多客户程序使用了“两遍”的、可定制的压缩方式。第一遍采用 Bentley and McIlroy’s 方式[6]，这种方式在一个很大的扫描窗口里对常见的长字符串进行压缩；第二遍是采用快速压缩算法，即在一个 16KB 的小扫描窗口中寻找重复数据。两个压缩的算法都很快，在现在的机器上，压缩的速率达到 100-200MB/s，解压的速率达到 400-1000MB/s。

虽然我们在选择压缩算法的时候重点考虑的是速度而不是压缩的空间，但是这种两遍的压缩方式在空间压缩率上的表现也是令人惊叹。比如，在 Webtable 的例子里，我们使用这种压缩方式来存储网页内容。在一次测试中，我们在一个压缩的局部性群组中存储了大量的网页。针对实验的目的，我们没有存储每个文档所有版本的数据，我们仅仅存储了一个版本的数据。该模式的空间压缩比达到了 10:1。这比传统的 Gzip 在压缩 HTML 页面时 3:1 或者 4:1 的空间压缩比好的多；“两遍”的压缩模式如此高效的原因是由于 Webtable 的行的存放方式：从同一个主机获取的页面都存在临近的地方。利用这个特性，Bentley-McIlroy 算法可以从来自同一个主机的页面里找到大量的重复内容。不仅仅是 Webtable，其它的很多应用程序也通过选择合适的行名来将相似的数据聚簇在一起，以获取较高的压缩率。当我们在 Bigtable 中存储同一份数据的多个版本的时候，压缩效率会更高。

7.3 通过缓存提高读操作的性能

为了提高读操作的性能，Tablet 服务器使用二级缓存的策略。扫描缓存是第一级缓存，主要缓存 Tablet 服务器通过 SSTable 接口获取的 Key-Value 对；Block 缓存是二级缓存，缓存的是从 GFS 读取的 SSTable 的

[22] 相比于对整个 SSTable 进行压缩，分块压缩压缩率较低

Block。对于经常要重复读取相同数据的应用程序来说，扫描缓存非常有效；对于经常要读取刚刚读过的数据附近的数据的应用程序来说，Block 缓存更有用（例如，顺序读，或者在一个热点的行的局部性群组中随机读取不同的列)。

7.3.1 Bloom 过滤器[23]

如 6.3 节所述，一个读操作必须读取构成 Tablet 状态的所有 SSTable 的数据。如果这些 SSTable 不在内存中，那么就需要多次访问硬盘。我们通过允许客户程序对特定局部性群组的 SSTable 指定 Bloom 过滤器【7】，来减少硬盘访问的次数。我们可以使用 Bloom 过滤器查询一个 SSTable 是否包含了特定行和列的数据。对于某些特定应用程序，我们只付出了少量的、用于存储 Bloom 过滤器的内存的代价，就换来了读操作显著减少的磁盘访问的次数。使用 Bloom 过滤器也隐式的达到了当应用程序访问不存在的行或列时，大多数时候我们都不需要访问硬盘的目的。

7.3.2 Commit 日志的实现

如果我们把对每个 Tablet 的操作的 Commit 日志都存在一个单独的文件的话，那么就会产生大量的文件，并且这些文件会并行的写入 GFS。根据 GFS 服务器底层文件系统实现的方案，要把这些文件写入不同的磁盘日志文件时[24]，会有大量的磁盘 Seek 操作。另外，由于批量提交[25]中操作的数目一般比较少，因此，对每个 Tablet 设置单独的日志文件也会给批量提交本应具有的优化效果带来很大的负面影响。为了避免这些问题，我们设置每个 Tablet 服务器一个 Commit 日志文件，把修改操作的日志以追加方式写入同一个日志文件，因此一个实际的日志文件中混合了对多个 Tablet 修改的日志记录。

使用单个日志显著提高了普通操作的性能，但是将恢复的工作复杂化了。当一个 Tablet 服务器宕机时，它加载的 Tablet 将会被移到很多其它的 Tablet 服务器上：每个 Tablet 服务器都装载很少的几个原来的服务器的 Tablet。当恢复一个 Tablet 的状态的时候，新的 Tablet 服务器要从原来的 Tablet 服务器写的日志中提取修改操作的信息，并重新执行。然而，这些 Tablet 修改操作的日志记录都混合在同一个日志文件中的。一种方法新的 Tablet 服务器读取完整的 Commit 日志文件，然后只重复执行它需要恢复的 Tablet 的相关修改操作。使用这种方法，假如有 100 台 Tablet 服务器，每台都加载了失效的 Tablet 服务器上的一个 Tablet，那么，这个日志文件就要被读取 100 次（每个服务器读取一次）。

为了避免多次读取日志文件，我们首先把日志按照关键字（table，row name，log sequence number）排序。排序之后，对同一个 Tablet 的修改操作的日志记录就连续存放在了一起，因此，我们只要一次磁盘 Seek 操作、

---

[23] Bloom，又叫布隆过滤器，什么意思？请参考 Google 黑板报 http://googlechinablog.com/2007/07/bloom-filter.html

[24] 原文：different physical log files

[25] 原文：group commit

之后顺序读取就可以了。为了并行排序，我们先将日志分割成 64MB 的段，之后在不同的 Tablet 服务器对段进行并行排序。这个排序工作由 Master 服务器来协同处理，并且在一个 Tablet 服务器表明自己需要从 Commit 日志文件恢复 Tablet 时开始执行。

在向 GFS 中写 Commit 日志的时候可能会引起系统颠簸，原因是多种多样的（比如，写操作正在进行的时候，一个 GFS 服务器宕机了；或者连接三个 GFS 副本所在的服务器的网络拥塞或者过载了）。为了确保在 GFS 负载高峰时修改操作还能顺利进行，每个 Tablet 服务器实际上有两个日志写入线程，每个线程都写自己的日志文件，并且在任何时刻，只有一个线程是工作的。如果一个线程的在写入的时候效率很低，Tablet 服务器就切换到另外一个线程，修改操作的日志记录就写入到这个线程对应的日志文件中。每个日志记录都有一个序列号，因此，在恢复的时候，Tablet 服务器能够检测出并忽略掉那些由于线程切换而导致的重复的记录。

7.3.3 Tablet 恢复提速

当 Master 服务器将一个 Tablet 从一个 Tablet 服务器移到另外一个 Tablet 服务器时，源 Tablet 服务器会对这个 Tablet 做一次 Minor Compaction。这个 Compaction 操作减少了 Tablet 服务器的日志文件中没有归并的记录，从而减少了恢复的时间。Compaction 完成之后，该服务器就停止为该 Tablet 提供服务。在卸载 Tablet 之前，源 Tablet 服务器还会再做一次（通常会很快）Minor Compaction，以消除前面在一次压缩过程中又产生的未归并的记录。第二次 Minor Compaction 完成以后，Tablet 就可以被装载到新的 Tablet 服务器上了，并且不需要从日志中进行恢复。

7.3.4 利用不变性

我们在使用 Bigtable 时，除了 SSTable 缓存之外的其它部分产生的 SSTable 都是不变的，我们可以利用这一点对系统进行简化。例如，当从 SSTable 读取数据的时候，我们不必对文件系统访问操作进行同步。这样一来，就可以非常高效的实现对行的并行操作。memtable 是唯一一个能被读和写操作同时访问的可变数据结构。为了减少在读操作时的竞争，我们对内存表采用 COW(Copy-on-write)机制，这样就允许读写操作并行执行。

因为 SSTable 是不变的，因此，我们可以把永久删除被标记为“删除”的数据的问题，转换成对废弃的 SSTable 进行垃圾收集的问题了。每个 Tablet 的 SSTable 都在 METADATA 表中注册了。Master 服务器采用“标记-删除”的垃圾回收方式删除 SSTable 集合中废弃的 SSTable【25】，METADATA 表则保存了 Root SSTable 的集合。

最后，SSTable 的不变性使得分割 Tablet 的操作非常快捷。我们不必为每个分割出来的 Tablet 建立新的 SSTable 集合，而是共享原来的 Tablet 的 SSTable 集合。

8 性能评估

为了测试 Bigtable 的性能和可扩展性，我们建立了一个包括 N 台 Tablet 服务器的 Bigtable 集群，这里 N 是可变的。每台 Tablet 服务器配置了 1GB 的内存，数据写入到一个包括 1786 台机器、每台机器有 2 个 IDE 硬盘的 GFS 集群上。我们使用 N 台客户机生成工作负载测试 Bigtable。（我们使用和 Tablet 服务器相同数目的客户机以确保客户机不会成为瓶颈。） 每台客户机配置 2GZ 双核 Opteron 处理器，配置了足以容纳所有进程工作数据集的物理内存，以及一张 Gigabit 的以太网卡。这些机器都连入一个两层的、树状的交换网络里，在根节点上的带宽加起来有大约 100-200Gbps。所有的机器采用相同的设备，因此，任何两台机器间网络来回一次的时间都小于 1ms。

Tablet 服务器、Master 服务器、测试机、以及 GFS 服务器都运行在同一组机器上。每台机器都运行一个 GFS 的服务器。其它的机器要么运行 Tablet 服务器、要么运行客户程序、要么运行在测试过程中，使用这组机器的其它的任务启动的进程。

R 是测试过程中，Bigtable 包含的不同的列关键字的数量。我们精心选择 R 的值，保证每次基准测试对每台 Tablet 服务器读/写的数据量都在 1GB 左右。

在序列写的基准测试中，我们使用的列关键字的范围是 0 到 R-1。这个范围又被划分为 10N 个大小相同的区间。核心调度程序把这些区间分配给 N 个客户端，分配方式是：只要客户程序处理完上一个区间的数据，调度程序就把后续的、尚未处理的区间分配给它。这种动态分配的方式有助于减少客户机上同时运行的其它进程对性能的影响。我们在每个列关键字下写入一个单独的字符串。每个字符串都是随机生成的、因此也没有被压缩[26]。另外，不同列关键字下的字符串也是不同的，因此也就不存在跨行的压缩。随机写入基准测试采用类似的方法，除了行关键字在写入前先做 Hash，Hash 采用按 R 取模的方式，这样就保证了在整个基准测试持续的时间内，写入的工作负载均匀的分布在列存储空间内。

序列读的基准测试生成列关键字的方式与序列写相同，不同于序列写在列关键字下写入字符串的是，序列读是读取列关键字下的字符串（这些字符串由之前序列写基准测试程序写入）。同样的，随机读的基准测试和随机写是类似的。

扫描基准测试和序列读类似，但是使用的是 BigTable 提供的、从一个列范围内扫描所有的 value 值的 API。由于一次 RPC 调用就从一个 Tablet 服务器取回了大量的 Value 值，因此，使用扫描方式的基准测试程序可以减少 RPC 调用的次数。

随机读（内存）基准测试和随机读类似，除了包含基准测试数据的局部性群组被设置为“in-memory”，因此，读操作直接从 Tablet 服务器的内存中读取数据，不需要从 GFS 读取数据。针对这个测试，我们把每台

[26] 参考第 7 节的压缩小节

Tablet 服务器存储的数据从 1GB 减少到 100MB，这样就可以把数据全部加载到 Tablet 服务器的内存中了。

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

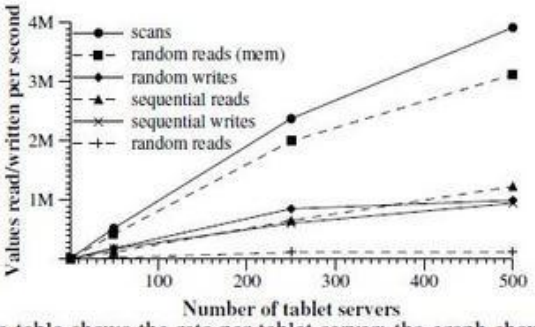


Figure 6: Number of 1000-byte values read/written per second. The table shows the rate per tablet server; the graph shows the aggregate rate.

图 6 中有两个视图，显示了我们的基准测试的性能；图中的数据和曲线是读/写 1000-byte value 值时取得的。图中的表格显示了每个 Tablet 服务器每秒钟进行的操作的次数；图中的曲线显示了每秒种所有的 Tablet 服务器上操作次数的总和。

8.1 单个 Tablet 服务器的性能

我们首先分析下单个 Tablet 服务器的性能。随机读的性能比其它操作慢一个数量级或以上[27]。 每个随机读操作都要通过网络从 GFS 传输 64KB 的 SSTable 到 Tablet 服务器，而我们只使用其中大小是 1000 byte 的一个 value 值。Tablet 服务器每秒大约执行 1200 次读操作，也就是每秒大约从 GFS 读取 75MB 的数据。这个传输带宽足以占满 Tablet 服务器的 CPU 时间，因为其中包括了网络协议栈的消耗、SSTable 解析、以及 BigTable 代码执行；这个带宽也足以占满我们系统中网络的链接带宽。大多数采用这种访问模式 BigTable 应用程序会减小 Block 的大小，通常会减到 8KB。

内存中的随机读操作速度快很多，原因是，所有 1000-byte 的读操作都是从 Tablet 服务器的本地内存中读取数据，不需要从 GFS 读取 64KB 的 Block。

随机和序列写操作的性能比随机读要好些，原因是每个 Tablet 服务器直接把写入操作的内容追加到一个 Commit 日志文件的尾部，并且采用批量提交的方式，通过把数据以流的方式写入到 GFS 来提高性能。随机写和序列写在性能上没有太大的差异，这两种方式的写操作实际上都是把操作内容记录到同一个 Tablet 服务器的 Commit 日志文件中。

序列读的性能好于随机读，因为每取出 64KB 的 SSTable 的 Block 后，这些数据会缓存到 Block 缓存中，后续的 64 次读操作直接从缓存读取数据。

扫描的性能更高，这是由于客户程序每一次 RPC 调用都会返回大量的 value 的数据，所以，RPC 调用的消耗基本抵消了。

[27] 原文：by the order of magnitude or more

8.2 性能提升

随着我们将系统中的 Tablet 服务器从 1 台增加到 500 台，系统的整体吞吐量有了梦幻般的增长，增长的倍率超过了 100。比如，随着 Tablet 服务器的数量增加了 500 倍，内存中的随机读操作的性能增加了 300 倍。之所以会有这样的性能提升，主要是因为这个基准测试的瓶颈是单台 Tablet 服务器的 CPU。

尽管如此，性能的提升还不是线性的。在大多数的基准测试中我们看到，当 Tablet 服务器的数量从 1 台增加到 50 台时，每台服务器的吞吐量会有一个明显的下降。这是由于多台服务器间的负载不均衡造成的，大多数情况下是由于其它的程序抢占了 CPU。 我们负载均衡的算法会尽量避免这种不均衡，但是基于两个主要原因，这个算法并不能完美的工作：一个是尽量减少 Tablet 的移动导致重新负载均衡能力受限（如果 Tablet 被移动了，那么在短时间内 — 一般是 1 秒内 — 这个 Tablet 是不可用的），另一个是我们的基准测试程序产生的负载会有波动[28]。

随机读基准测试的测试结果显示，随机读的性能随 Tablet 服务器数量增加的提升幅度最小（整体吞吐量只提升了 100 倍，而服务器的数量却增加了 500 倍）。这是因为每个 1000-byte 的读操作都会导致一个 64KB 大的 Block 在网络上传输。这样的网络传输量消耗了我们网络中各种共享的 1GB 的链路，结果导致随着我们增加服务器的数量，每台服务器上的吞吐量急剧下降。

9 实际应用

# of tablet servers	# of clusters
0 .. 19	259
20 .. 49	47
50 .. 99	20
100 .. 499	50
> 500	12

Table 1: Distribution of number of tablet servers in Bigtable clusters.

截止到 2006 年 8 月，Google 内部一共有 388 个非测试用的 Bigtable 集群运行在各种各样的服务器集群上，合计大约有 24500 个 Tablet 服务器。表 1 显示了每个集群上 Tablet 服务器的大致分布情况。这些集群中，许多用于开发目的，因此会有一段时期比较空闲。通过观察一个由 14 个集群、8069 个 Tablet 服务器组成的集群组，我们看到整体的吞吐量超过了每秒 1200000 次请求，发送到系统的 RPC 请求导致的网络负载达到了 741MB/s，系统发出的 RPC 请求网络负载大约是 16GB/s。

[28] 原文：the load generated by our benchmarks shifts around as the benchmark progresses

Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency-sensitive?
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
Google Earth	0.5	64%	8	7	2	33%	Yes
Google Earth	70	–	9	8	3	0%	No
Orkut	9	–	0.9	8	5	1%	Yes
Personalized Search	4	47%	6	93	11	5%	Yes

Table 2: Characteristics of a few tables in production use. *Table size* (measured before compression) and *# Cells* indicate approximate sizes. *Compression ratio* is not given for tables that have compression disabled.

表 2 提供了一些目前正在使用的表的相关数据。一些表存储的是用户相关的数据，另外一些存储的则是用于批处理的数据；这些表在总的大小、 每个数据项的平均大小、从内存中读取的数据的比例、表的 Schema 的复杂程度上都有很大的差别。本节的其余部分，我们将主要描述三个产品研发团队如何使用 Bigtable 的。

9.1 Google Analytics

Google Analytics 是用来帮助 Web 站点的管理员分析他们网站的流量模式的服务。它提供了整体状况的统计数据，比如每天的独立访问的用户数量、每天每个 URL 的浏览次数；它还提供了用户使用网站的行为报告，比如根据用户之前访问的某些页面，统计出几成的用户购买了商品。

为了使用这个服务，Web 站点的管理员只需要在他们的 Web 页面中嵌入一小段 JavaScript 脚本就可以了。这个 Javascript 程序在页面被访问的时候调用。它记录了各种 Google Analytics 需要使用的信息，比如用户的标识、获取的网页的相关信息。Google Analytics 汇总这些数据，之后提供给 Web 站点的管理员。

我们粗略的描述一下 Google Analytics 使用的两个表。Row Click 表（大约有 200TB 数据）的每一行存放了一个最终用户的会话。行的名字是一个包含 Web 站点名字以及用户会话创建时间的元组。这种模式保证了对同一个 Web 站点的访问会话是顺序的，会话按时间顺序存储。这个表可以压缩到原来尺寸的 14%。

Summary 表（大约有 20TB 的数据）包含了关于每个 Web 站点的、各种类型的预定义汇总信息。一个周期性运行的 MapReduce 任务根据 Raw Click 表的数据生成 Summary 表的数据。每个 MapReduce 工作进程都从 Raw Click 表中提取最新的会话数据。系统的整体吞吐量受限于 GFS 的吞吐量。这个表的能够压缩到原有尺寸的 29%。

9.2 Google Earth

Google 通过一组服务为用户提供了高分辨率的地球表面卫星图像，访问的方式可以使通过基于 Web 的 Google Maps 访问接口（maps.google.com），也可以通过 Google Earth 定制的客户端软件访问。这些软件产品允许用户浏览地球表面的图像：用户可以在不同的分辨率下平移、查看和注释这些卫星图像。这个系统使用

一个表存储预处理数据，使用另外一组表存储用户数据。

数据预处理流水线使用一个表存储原始图像。在预处理过程中，图像被清除，图像数据合并到最终的服务数据中。这个表包含了大约 70TB 的数据，所以需要从磁盘读取数据。图像已经被高效压缩过了，因此存储在 Bigtable 后不需要再压缩了。

Imagery 表的每一行都代表了一个单独的地理区域。行都有名称，以确保毗邻的区域存储在了一起。Imagery 表中有一个列族用来记录每个区域的数据源。这个列族包含了大量的列：基本上市每个列对应一个原始图片的数据。由于每个地理区域都是由很少的几张图片构成的，因此这个列族是非常稀疏的。

数据预处理流水线高度依赖运行在 Bigtable 上的 MapReduce 任务传输数据。在运行某些 MapReduce 任务的时候，整个系统中每台 Tablet 服务器的数据处理速度是 1MB/s。

这个服务系统使用一个表来索引 GFS 中的数据。这个表相对较小（大约是 500GB），但是这个表必须在保证较低的响应延时的前提下，针对每个数据中心，每秒处理几万个查询请求。 因此，这个表必须在上百个 Tablet 服务器上存储数据，并且使用 in-memory 的列族。

9.3 个性化查询

个性化查询（www.google.com/psearch）是一个双向服务；这个服务记录用户的查询和点击，涉及到各种 Google 的服务，比如 Web 查询、图像和新闻。用户可以浏览他们查询的历史，重复他们之前的查询和点击；用户也可以定制基于 Google 历史使用习惯模式的个性化查询结果。

个性化查询使用 Bigtable 存储每个用户的数据。每个用户都有一个唯一的用户 id，每个用户 id 和一个列名绑定。一个单独的列族被用来存储各种类型的行为（比如，有个列族可能是用来存储所有的 Web 查询的）。每个数据项都被用作 Bigtable 的时间戳，记录了相应的用户行为发生的时间。个性化查询使用以 Bigtable 为存储的 MapReduce 任务生成用户的数据图表。这些用户数据图表用来个性化当前的查询结果。

个性化查询的数据会复制到几个 Bigtable 的集群上，这样就增强了数据可用性，同时减少了由客户端和 Bigtable 集群间的“距离”造成的延时。个性化查询的开发团队最初建立了一个基于 Bigtable 的、“客户侧”的复制机制为所有的复制节点提供一致性保障。现在的系统则使用了内建的复制子系统。

个性化查询存储系统的设计允许其它的团队在它们自己的列中加入新的用户数据，因此，很多 Google 服务使用个性化查询存储系统保存用户级的配置参数和设置。在多个团队之间分享数据的结果是产生了大量的列族。为了更好的支持数据共享，我们加入了一个简单的配额机制[29]限制用户在共享表中使用的空间；配额也为使用个性化查询系统存储用户级信息的产品团体提供了隔离机制。

[29] quota，参考 AIX 的配额机制

10 经验教训

在设计、实现、维护和支持 Bigtable 的过程中，我们得到了很多有用的经验和一些有趣的教训。

一个教训是，我们发现，很多类型的错误都会导致大型分布式系统受损，这些错误不仅仅是通常的网络中断、或者很多分布式协议中设想的 fail-stop 类型的错误[30]。比如，我们遇到过下面这些类型的错误导致的问题：内存数据损坏、网络中断、时钟偏差、机器挂起、扩展的和非对称的网络分区[31]、我们使用的其它系统的 Bug（比如 Chubby）、GFS 配额溢出、计划内和计划外的硬件维护。我们在解决这些问题的过程中学到了很多经验，我们通过修改协议来解决这些问题。比如，我们在我们的 RPC 机制中加入了 Checksum。我们在设计系统的部分功能时，不对其它部分功能做任何的假设，这样的做法解决了其它的一些问题。比如，我们不再假设一个特定的 Chubby 操作只返回错误码集合中的一个值。

另外一个教训是，我们明白了在彻底了解一个新特性会被如何使用之后，再决定是否添加这个新特性是非常重要的。比如，我们开始计划在我们的 API 中支持通常方式的事务处理。但是由于我们还不会马上用到这个功能，因此，我们并没有去实现它。现在，Bigtable 上已经有了很多的实际应用，我们可以检查它们真实的需求；我们发现，大多是应用程序都只是需要单个行上的事务功能。有些应用需要分布式的事务功能，分布式事务大多数情况下用于维护二级索引，因此我们增加了一个特殊的机制去满足这个需求。新的机制在通用性上比分布式事务差很多，但是它更有效（特别是在更新操作的涉及上百行数据的时候），而且非常符合我们的“跨数据中心”复制方案的优化策略。

还有一个具有实践意义的经验：我们发现系统级的监控对 Bigtable 非常重要（比如，监控 Bigtable 自身以及使用 Bigtable 的客户程序）。比如，我们扩展了我们的 RPC 系统，因此对于一个 RPC 调用的例子，它可以详细记录代表了 RPC 调用的很多重要操作。这个特性允许我们检测和修正很多的问题，比如 Tablet 数据结构上的锁的内容、在修改操作提交时对 GFS 的写入非常慢的问题、以及在 METADATA 表的 Tablet 不可用时，对 METADATA 表的访问挂起的问题。关于监控的用途的另外一个例子是，每个 Bigtable 集群都在 Chubby 中注册了。这可以帮助我们跟踪所有的集群状态、监控它们的大小、检查集群运行的我们软件的版本、监控集群流入数据的流量，以及检查是否有引发集群高延时的潜在因素。

对我们来说，最宝贵的经验是简单设计的价值。考虑到我们系统的代码量（大约 100000 行生产代码[32]），以及随着时间的推移，新的代码以各种难以预料的方式加入系统，我们发现简洁的设计和编码给维护和调试带来的巨大好处。这方面的一个例子是我们的 Tablet 服务器成员协议。我们第一版的协议很简单：Master 服

---

[30] fail-stop failture，指一旦系统 fail 就 stop，不输出任何数据；fail-fast failture，指 fail 不马上 stop，在短时间内 return 错误信息，然后再 stop

[31] extended and asymmetric network partitions，不明白什么意思。partition 也有中断的意思，但是我不知道如何用在这里

[32] non-test code

务器周期性的和 Tablet 服务器签订租约，Tablet 服务器在租约过期的时候 Kill 掉自己的进程。不幸的是，这个协议在遇到网络问题时会大大降低系统的可用性，也会大大增加 Master 服务器恢复的时间。我们多次重新设计这个协议，直到它能够很好的处理上述问题。但是，更不幸的是，最终的协议过于复杂了，并且依赖一些 Chubby 很少被用到的特性。我们发现我们浪费了大量的时间在调试一些古怪的问题[33]，有些是 Bigtable 代码的问题，有些是 Chubby 代码的问题。最后，我们只好废弃了这个协议，重新制订了一个新的、更简单、只使用 Chubby 最广泛使用的特性的协议。

11 相关工作

Boxwood【24】项目的有些组件在某些方面和 Chubby、GFS 以及 Bigtable 类似，因为它也提供了诸如分布式协议、锁、分布式 Chunk 存储以及分布式 B-tree 存储。Boxwood 与 Google 的某些组件尽管功能类似，但是 Boxwood 的组件提供更底层的服务。Boxwood 项目的目的是提供创建类似文件系统、数据库等高级服务的基础构件，而 Bigtable 的目的是直接为客户程序的数据存储需求提供支持。

现在有不少项目已经攻克了很多难题，实现了在广域网上的分布式数据存储或者高级服务，通常是“Internet 规模”的。这其中包括了分布式的 Hash 表，这项工作由一些类似 CAN【29】、Chord【32】、Tapestry【37】和 Pastry【30】的项目率先发起。这些系统的主要关注点和 Bigtable 不同，比如应对各种不同的传输带宽、不可信的协作者、频繁的更改配置等；另外，去中心化和 Byzantine 灾难冗余[34]也不是 Bigtable 的目的。

就提供给应用程序开发者的分布式数据存储模型而言，我们相信，分布式 B-Tree 或者分布式 Hash 表提供的 Key-value pair 方式的模型有很大的局限性。Key-value pair 模型是很有用的组件，但是它们不应该是提供给开发者唯一的组件。我们选择的模型提供的组件比简单的 Key-value pair 丰富的多，它支持稀疏的、半结构化的数据。另外，它也足够简单，能够高效的处理平面文件；它也是透明的（通过局部性群组），允许我们的使用者对系统的重要行为进行调整。

有些数据库厂商已经开发出了并行的数据库系统，能够存储海量的数据。Oracle 的 RAC【27】使用共享磁盘存储数据（Bigtable 使用 GFS），并且有一个分布式的锁管理系统（Bigtable 使用 Chubby）。IBM 并行版本的 DB2【4】基于一种类似于 Bigtable 的、不共享任何东西的架构[35]【33】。每个 DB2 的服务器都负责处理存储在一个关系型数据库中的表中的行的一个子集。这些产品都提供了一个带有事务功能的完整的关系模型。

---

[33] obscure corner cases

[34] Byzantine，即拜占庭式的风格，也就是一种复杂诡秘的风格。Byzantine Fault 表示：对于处理来说，当发错误时处理器并不停止接收输出，也不停止输出，错就错了，只管算，对于这种错误来说，这样可真是够麻烦了，因为用户根本不知道错误发生了，也就根本谈不上处理错误了。在多处理器的情况下，这种错误可能导致运算正确结果的处理器也产生错误的结果，这样事情就更麻烦了，所以一定要避免处理器产生这种错误。

[35] a shared-nothing architecture

Bigtable 的局部性群组提供了类似于基于列的存储方案在压缩和磁盘读取方面具有的性能；这些以列而不是行的方式组织数据的方案包括 C-Store【1，34】、商业产品 Sybase IQ【15，36】、SenSage【31】、KDB+【22】，以及 MonetDB/X100【38】的 ColumnDM 存储层。另外一种在平面文件中提供垂直和水平数据分区、并且提供很好的数据压缩率的系统是 AT&T 的 Daytona 数据库【19】。局部性群组不支持 Ailamaki 系统中描述的 CPU 缓存级别的优化【2】。

Bigtable 采用 memtable 和 SSTable 存储对表的更新的方法与 Log-Structured Merge Tree【26】存储索引数据更新的方法类似。这两个系统中，排序的数据在写入到磁盘前都先存放在内存中，读取操作必须从内存和磁盘中合并数据产生最终的结果集。

C-Store 和 Bigtable 有很多相似点：两个系统都采用 Shared-nothing 架构，都有两种不同的数据结构，一种用于当前的写操作，另外一种存放“长时间使用”的数据，并且提供一种机制在两个存储结构间搬运数据。两个系统在 API 接口函数上有很大的不同：C-Store 操作更像关系型数据库，而 Bigtable 提供了低层次的读写操作接口，并且设计的目标是能够支持每台服务器每秒数千次操作。C-Store 同时也是个“读性能优化的关系型数据库”，而 Bigtable 对读和写密集型应用都提供了很好的性能。

Bigtable 也必须解决所有的 Shared-nothing 数据库需要面对的、类型相似的一些负载和内存均衡方面的难题（比如，【11，35】）。我们的问题在某种程度上简单一些：

1. 我们不需要考虑同一份数据可能有多个拷贝的问题，同一份数据可能由于视图或索引的原因以不同的形式表现出来；
2. 我们让用户决定哪些数据应该放在内存里、哪些放在磁盘上，而不是由系统动态的判断；
3. 我们的系统中没有复杂的查询执行或优化工作。

12 结论

我们已经讲述完了 Bigtable，Google 的一个分布式的结构化数据存储系统。Bigtable 的集群从 2005 年 4 月开始已经投入使用了，在此之前，我们花了大约 7 人年设计和实现这个系统。截止到 2006 年 4 月，已经有超过 60 个项目使用 Bigtable 了。我们的用户对 Bigtable 提供的高性能和高可用性很满意，随着时间的推移，他们可以根据自己的系统对资源的需求增加情况，通过简单的增加机器，扩展系统的承载能力。

由于 Bigtable 提供的编程接口并不常见，一个有趣的问题是：我们的用户适应新的接口有多难？新的使用者有时不太确定使用 Bigtable 接口的最佳方法，特别是在他们已经习惯于使用支持通用事务的关系型数据库的接口的情况下。但是，Google 内部很多产品都成功的使用了 Bigtable 的事实证明了，我们的设计在实践中行之有效。

我们现在正在对 Bigtable 加入一些新的特性，比如支持二级索引，以及支持多 Master 节点的、跨数据中

心复制的 Bigtable 的基础构件。我们现在已经开始将 Bigtable 部署为服务供其它的产品团队使用，这样不同的产品团队就不需要维护他们自己的 Bigtable 集群了。随着服务集群的扩展，我们需要在 Bigtable 系统内部处理更多的关于资源共享的问题了【3，5】。

最后，我们发现，建设 Google 自己的存储解决方案带来了很多优势。通过为 Bigtable 设计我们自己的数据模型，是我们的系统极具灵活性。另外，由于我们全面控制着 Bigtable 的实现过程，以及 Bigtable 使用到的其它的 Google 的基础构件，这就意味着我们在系统出现瓶颈或效率低下的情况时，能够快速的解决这些问题。

13 Acknowledgements

We thank the anonymous reviewers, David Nagle, and our shepherd Brad Calder, for their feedback on this paper.The Bigtable system has benefited greatly from the feedback of our many users within Google. In addition,we thank the following people for their contributions to Bigtable: Dan Aguayo, Sameer Ajmani, Zhifeng Chen,Bill Coughran, Mike Epstein, Healfdene Goguen, Robert Griesemer, Jeremy Hylton, Josh Hyman, Alex Khesin,

Joanna Kulik, Alberto Lerner, Sherry Listgarten, Mike Maloney, Eduardo Pinheiro, Kathy Polizzi, Frank Yellin,and Arthur Zwiegincew.

14 References

[1] ABADI, D. J., MADDEN, S. R., AND FERREIRA, M. C. Integrating compression and execution in columnoriented database systems. Proc. of SIGMOD (2006).

[2] AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND SKOUNAKIS, M. Weaving relations for cache performance.In The VLDB Journal (2001), pp. 169-180.

[3] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: A new facility for resource management in server systems. In Proc. of the 3rd OSDI (Feb. 1999), pp. 45-58.

[4] BARU, C. K., FECTEAU, G., GOYAL, A., HSIAO, H., JHINGRAN, A., PADMANABHAN, S., COPELAND,G. P., AND WILSON, W. G. DB2 parallel edition. IBM Systems Journal 34, 2 (1995), 292-322.

[5] BAVIER, A., BOWMAN, M., CHUN, B., CULLER, D., KARLIN, S., PETERSON, L., ROSCOE, T., SPALINK, T., AND WAWRZONIAK, M. Operating system support for planetary-scale network services. In Proc. of the 1st NSDI(Mar. 2004), pp. 253-266.

[6] BENTLEY, J. L., AND MCILROY, M. D. Data compression using long common strings. In Data Compression Conference (1999), pp. 287-295.

yore

[7] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. CACM 13, 7 (1970), 422-426.

[8] BURROWS, M. The Chubby lock service for looselycoupled distributed systems. In Proc. of the 7th OSDI (Nov. 2006).

[9] CHANDRA, T., GRIESEMER, R., AND REDSTONE, J.Paxos made live ? An engineering perspective. In Proc. of PODC (2007).

[10] COMER, D. Ubiquitous B-tree. Computing Surveys 11, 2 (June 1979), 121-137.

[11] COPELAND, G. P., ALEXANDER, W., BOUGHTER, E. E., AND KELLER, T. W. Data placement in Bubba. In Proc. of SIGMOD (1988), pp. 99-108.

[12] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In Proc. of the 6th OSDI (Dec. 2004), pp. 137-150.

[13] DEWITT, D., KATZ, R., OLKEN, F., SHAPIRO, L., STONEBRAKER, M., AND WOOD, D. Implementation techniques for main memory database systems. In Proc. of SIGMOD (June 1984), pp. 1-8.

[14] DEWITT, D. J., AND GRAY, J. Parallel database systems: The future of high performance database systems. CACM 35, 6 (June 1992), 85-98.

[15] FRENCH, C. D. One size ts all database architectures do not work for DSS. In Proc. of SIGMOD (May 1995), pp. 449-450.

[16] GAWLICK, D., AND KINKADE, D. Varieties of concurrency control in IMS/VS fast path. Database Engineering Bulletin 8, 2 (1985), 3-10.

[17] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In Proc. of the 19th ACM SOSP (Dec.2003), pp. 29-43.

[18] GRAY, J. Notes on database operating systems. In Operating Systems ? An Advanced Course, vol. 60 of Lecture Notes in Computer Science. Springer-Verlag, 1978.

[19] GREER, R. Daytona and the fourth-generation language Cymbal. In Proc. of SIGMOD (1999), pp. 525-526.

[20] HAGMANN, R. Reimplementing the Cedar file system using logging and group commit. In Proc. of the 11th SOSP (Dec. 1987), pp. 155-162.

[21] HARTMAN, J. H., AND OUSTERHOUT, J. K. The Zebra striped network file system. In Proc. of the 14th SOSP(Asheville, NC, 1993), pp. 29-43.

[22] KX.COM. kx.com/products/database.php. Product page.

[23] LAMPORT, L. The part-time parliament. ACM TOCS 16,2 (1998), 133-169.

yore

[24] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In Proc. of the 6th OSDI (Dec. 2004), pp. 105-120.

[25] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine. CACM 3, 4 (Apr. 1960), 184-195.

[26] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). Acta Inf. 33, 4 (1996), 351-385.

[27] ORACLE.COM. www.oracle.com/technology/products/database/clustering/index.html. Product page.

[28] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. Scientific Programming Journal 13, 4 (2005), 227-298.

[29] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In Proc. of SIGCOMM (Aug. 2001), pp. 161-172.

[30] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for largescale peer-to-peer systems. In Proc. of Middleware 2001(Nov. 2001), pp. 329-350.

[31] SENSAGE.COM. sensage.com/products-sensage.htm. Product page.

[32] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In Proc. of SIGCOMM (Aug. 2001), pp. 149-160.

[33] STONEBRAKER, M. The case for shared nothing. Database Engineering Bulletin 9, 1 (Mar. 1986), 4-9.

[34] STONEBRAKER,M., ABADI, D. J., BATKIN, A., CHEN, X., CHERNIACK, M., FERREIRA, M., LAU, E., LIN, A., MADDEN, S., O'NEIL, E., O'NEIL, P., RASIN, A., TRAN, N., AND ZDONIK, S. C-Store: A columnoriented DBMS. In Proc. of VLDB (Aug. 2005), pp. 553-564.

[35] STONEBRAKER, M., AOKI, P. M., DEVINE, R., LITWIN, W., AND OLSON, M. A. Mariposa: A new architecture for distributed data. In Proc. of the Tenth ICDE(1994), IEEE Computer Society, pp. 54-65.

[36] SYBASE.COM. www.sybase.com/products/databaseservers/sybaseiq. Product page.

[37] ZHAO, B. Y., KUBIATOWICZ, J., AND JOSEPH, A. D. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, CS Division, UC Berkeley, Apr. 2001.

[38] ZUKOWSKI, M., BONCZ, P. A., NES, N., AND HEMAN, S. MonetDB/X100 ?A DBMS in the CPU cache. IEEE Data Eng. Bull. 28, 2 (2005), 17-22.

yore 2017-09-05