CHAPTER

Cloud data storage

7

The volume of data generated by human activities is growing at a breathtaking rate. More data was generated in the last two years than in the entire human history before that. Computer clouds provide vast amounts of storage demanded by many applications using these data. Several data sources contribute to the massive amounts of data stored on a cloud. A variety of sensors feed streams of data to cloud applications or simply generate content. An ever-increasing number of cloud-based services collect detailed data about their services and information about the users of these services.

Big Data, discussed in depth in Chapter 12, reflects the reality that many applications use data sets so large that local computers, or even small-to-medium scale data centers, do not have the capacity to store and process such data. Big Data growth can be viewed as a three-dimensional phenomenon as it: (i) implies an increased volume of data; (ii) requires increased processing speed to process more data and produce more results; and (iii) involves a diversity of data sources and data types.

A network-centric data storage model is particularly useful for mobile devices with limited power reserves and local storage, now able to save and access large audio and video files on computer clouds. Billions of Internet-connected mobile and stationary devices access data stored on computer clouds.

Cloud data storage and processing are intimately tied to one another. Data analytics uses large volumes of data collected by an organization to optimize its businesses. In-depth data analysis allows an organization to reach a larger population of customers, identify the strengths of the products or shortcomings in the organization, save energy, and, last but not least, protect the environment.

Applications in many areas of science, including genomics, structural biology, high energy physics, astronomy, meteorology, and the study of the environment, carry out complex analysis of data sets often on the order of terabytes.[1] As a result, file systems, such as Btrfs, XFS, ZFS, exFAT, NTFS, HFS Plus, and ReFS, support disk formats with theoretical volume sizes of several exabytes.

While we emphasize the advantages of a concentration of resources, we have to be acutely aware that a cloud is a large-scale distributed system with a very large number of components that must work in concert. Management of a large collection of storage systems poses significant challenges and requires novel approaches to storage system design. Effective data replication and storage management strategies are critical for cloud applications.

Sophisticated strategies to reduce the access time and to support multimedia access are necessary to satisfy the timing requirements of data streaming and content delivery. Data replication allows concurrent access to data from multiple processors and decreases the chances of data loss. Maintaining consistency among multiple copies of data records increases the complexity of data management software and could negatively affect the storage system performance if data is frequently updated.

[1] Terabyte, 1 TB $= 10^{12}$ bytes; Petabyte, 1 PB $= 10^{15}$ bytes; Exabyte, 1 EX $= 10^{18}$ bytes; Zettabyte, 1 ZB $= 10^{21}$ bytes.

Cloud Computing. https://doi.org/10.1016/B978-0-32-385277-7.00014-2

Nowadays, large-scale systems are built with off-the-shelf components, while the distributed file systems of the past used custom-designed reliable components. The storage system design philosophy has shifted from performance-at-any-cost to reliability-at-the-lowest-possible-cost. This shift is evident in the evolution of ideas from the file systems of the 1980s, such as the Network File System (NFS), the Andrew File System (AFS), and the Sprite File System (SFS), to the Google File System (GFS), the Megastore, and the Colossus [173], developed during the last two decades.

The discussion of cloud storage starts with a review of storage technology in Sections 7.1 and 7.2, followed by an overview of storage models in Section 7.3. Evolution of file systems from distributed file system to parallel file systems, then to the file systems capable of handling massive amounts of data, is presented in Sections 7.4 and 7.5, where we discuss distributed file systems, General Parallel File Systems, and Google File System, respectively.

A locking service, Chubby, based on the Paxos algorithm, is presented in Section 7.7. The mismatch between relational database systems and cloud requirements in Section 7.8 is followed by a discussion of NoSQL databases in Section 7.9 and of transaction processing systems in Section 7.10. Sections 7.11 and 7.12 analyze the BigTable and the Megastore systems, respectively. Storage reliability at scale, data center disk locality, and database provenance are discussed in Sections 7.13, 7.14, and 7.15, respectively.

7.1 Dynamic random access memories and hard disk drives

During the last decades, the storage technology has evolved at an accelerated pace, and the volume of data stored every year has constantly increased [236]. Though it pales in comparison to the evolution of processor technology, the evolution of storage technology is astounding. A 2003 study [359] shows that during the 1980–2003 period the storage density of hard disk drives (HDD) has increased by four orders of magnitude from about 0.01 Gb/in^2 to about 100 Gb/in^2. During the same period the prices have fallen by five orders of magnitude to about 1 cent/Mbyte. HDD densities were projected to climb to 1 800 Gb/in^2 by 2016, up from 744 Gb/in^2 in 2011. Solid-state drives (SSDs) can support hundreds of thousands of I/O operations per second (IOPS).

Dynamic Random Access Memory (DRAM). The density of DRAM (Dynamic Random Access Memory) devices increased from about 1 Gb/in^2 in 1990 to 100 Gb/in^2 in 2003. DRAM cost tumbled from \$80/MB to less than \$1/MB during the same period. In April 2020, TSMC (Taiwan Semiconductor Manufacturing Company) started mass production of integrated circuits using 5 nm lithografic process.

Recent advancements in storage technology have a broad impact on the storage systems used for cloud computing. The capacity of NAND flash-based devices outpaced DRAM capacity growth, and the cost per gigabyte has significantly declined. Manufacturers of storage devices are investing in competing solid-state technologies such as Phase-Change Memory. While solid-state memories are based on electron charge, another fundamental property of an electron, its spin, is used to store information. *Spintronics*, an acronym for spin transport electronics, promises storage media based on antiferromagnetic materials insensitive to perturbations by stray fields and with much shorter switching times [518].

While the density of storage devices has increased and the cost has decreased dramatically, the access time has improved only slightly. Performance of I/O subsystems has not kept pace with the

performance of processors, and this performance gap affects multimedia, scientific and engineering, and other applications that process increasingly large volumes of data.

Storage systems face substantial pressure because the volume of data generated has increased exponentially during the last decades. In the 1980s and 1990s, data was primarily generated by humans, nowadays, machines generate data at an unprecedented rate. Mobile devices, such as smartphones and tablets recording static images and movies have limited local storage capacity and rely on transferring the data to cloud storage. Sensors, surveillance cameras, and digital medical imaging devices generate data at a high rate and dump it on storage systems accessible via the Internet.

Online digital libraries, eBooks, and digital media, along with reference data, add to the demand for massive amounts of storage. The term reference data is used for infrequently used data, such as archived copies of medical or financial records, customer account statements, etc.

As the volume of data increases, new methods and algorithms for data mining that require powerful computing systems are being developed. Only a concentration of resources could provide the CPU cycles, along with the vast storage capacity, necessary when performing such intensive computations and when accessing the very large volume of data.

The rapid technological advancements have changed the balance between the initial investment in the storage devices and the system management costs. Now, the cost of storage management is the dominant element of the total cost of a storage system. This effect favors the centralized storage strategy supported by a cloud; indeed, a centralized approach can automate some of the storage management functions such as replication and backup and, thus, reduce substantially the storage management cost.

Hard disk drives (HDDs) are ubiquitous secondary storage media for general-purpose computers. An HDD is a nonvolatile random-access data storage device consisting of one or more rotating platters coated with magnetic material. Magnetic heads mounted on a moving actuator arm read and write data to the surface of the platters.

A typical HDD has a spindle motor and an actuator that positions the read/write head assembly across the spinning disks. Rotation speed of platters in today's HDDs ranges from 4 200 rpm for energy-efficient portable devices, to 15 000 rpm for high-performance servers. HDDs for desktop computers and laptops are 3.5-inch and 2.5-inch, respectively.

HDDs are characterized by capacity and performance. The capacity is measured in Megabytes (MB), Terabytes (TB), or Gigabytes (GB). The average access time is the most relevant HDD performance indicator. The access time includes the *seek time*, the time for the arm to reach to the cylinder/track, and the *search time*, the time to locate the record on a track. HDD technology has improved dramatically since the disk first introduced by IBM in 1956, as shown in Table 7.1.

7.2 Solid-state disks

Solid-state disks are solid-state persistent storage devices using integrated circuit assemblies as memory. SSD interfaces are compatible with the block I/O of HDDs, thus they can replace the traditional disks. SSDs do not have moving parts, are typically more resistant to physical shock, run silently, and have shorter access time and lower latency than HDDs.

Lower-priced SSDs use triple-level or multi-level cell (MLC) flash memory, slower and less reliable than single-level cell (SLC) flash memory. MLC to SLC ratios of persistence, sequential write, sequen-

Table 7.1 Evolution of hard disk drive technologies from 1956 to 2016. The improvement ranges from astounding ratios such as 650×10^6 to one, 300×10^6 to one, and 2.7×10^6 to one for density, price, and capacity, respectively, to a modest 200 to one for average access time and 11 to one for MTBF, the mean time between failures.

Parameter	1956	2016
Capacity	3.75 MB	10 TB
Average access time	≈ 600 msec	2.5–10 ms
Density	200 bits/sq. inch	1.3 TB sq. inch
Average life span	≈ 2 000 hours/MTBF	≈ 22 500 hours/MTBF
Price	$9 200/MB	$0.032/GB
Weight	910 Kg	62 g
Physical volume	1.9 m^3	34 cm^3

tial read, and price are 1 : 10, 1 : 3, 1 : 1, and 1 : 1.3, respectively. Most SSDs use MLC NAND-based flash memory, a nonvolatile memory that retains data when power is lost.

SLC NAND I/O operation latency is: 25 μsec to fetch a 4 KB page from the array to the I/O buffer on a read, 250 μsec to commit a 4 KB page from the I/O buffer to the array on a write, and 2 msec to erase a 256 KB block. When multiple NAND devices operate in parallel, SSD bandwidth scales up and high latencies can be hidden, provided that the load is evenly distributed between NAND devices and sufficient outstanding operations are pending. Most SSD manufacturers use nonvolatile NAND due to lower cost compared with DRAM and to the ability to retain data without a constant power supply.

Solid-state hybrid disks (SSHDs) combine the features of SSDs and HDDs; they include a large HDD and an SSD cache to improve performance of frequently accessed data. The SSD cost-per-byte is reduced by about 50% every year, but it must be reduced by up to three orders of magnitude to be competitive with HDD.

Table 7.2 shows that SSD capacity has increased by a factor of five million from 1991 to 2018, and the price per GB has decreased by a factor of 555 000, while the access times has decreased 11 times for read and 38 times for write.

Enterprise flash drives (EFDs) are SSDs with a higher set of specifications, designed for applications requiring high I/O performance (IOPS), reliability, energy efficiency, and consistent performance.

An EFD includes a controller, an embedded processor critical for EFD performance that executes firmware-level code. Some of the functions performed by the controller are: (i) bad block mapping; (ii) read and write caching; (iii) encryption; (iv) crypto-shredding; (v) error detection and correction; (vi) garbage collection; (vi) read scrubbing and read disturb management; and (vii) wear leveling.

The performance scales with the number of parallel NAND flash chips. A single NAND chip is relatively slow, due to the narrow (8/16 bit) asynchronous I/O interface, and additional high latency of basic I/O operations. Multiple NAND devices operate in parallel and the bandwidth scales. High latencies can be hidden, as long as enough outstanding operations are pending and the load is evenly distributed between devices.

NVM Express (NVMe) is an optimized, high-performance scalable host controller interface for non-volatile memory (NVM) technologies; see Fig. 7.1. Typical SAS (Serial Attached SCSI) devices

Table 7.2 SSD Evolution. Leftmost column—SSD parameters, capacity, bandwidth, latency, and cost. Middle column—characteristics, high price, pick performer of enterprise SSDs. Right column—the same parameters for consumer SSDs.

Parameter	Enterprise	Available for Consumers
Capacity	100 TB in 2018 DC100 (20 MB in 1991)	8 TB in 2020
Sequential read speed	15 GB/sec in 2019 (49.3 MB/sec in 2007)	6.795 GB/sec in 2020
Sequential write speed	15.2 GB/sec in 2019 (80.0 MB/sec in 2007)	4.397 GB/sec in 2020
IOPS (I/O operations)	2 500 000 in 2019 (79 in 2007)	736 000 read & 700 000 write in 2020
Access time	45 nsec read & 13 nsec write in 2020 (500 nsec in 2007)	45 nsec read & 13 nsec write in 2020
Price	$ 0.10/GB in 2020 ($ 50 000/GB in 1991)	$ 0.10/GB in 2020

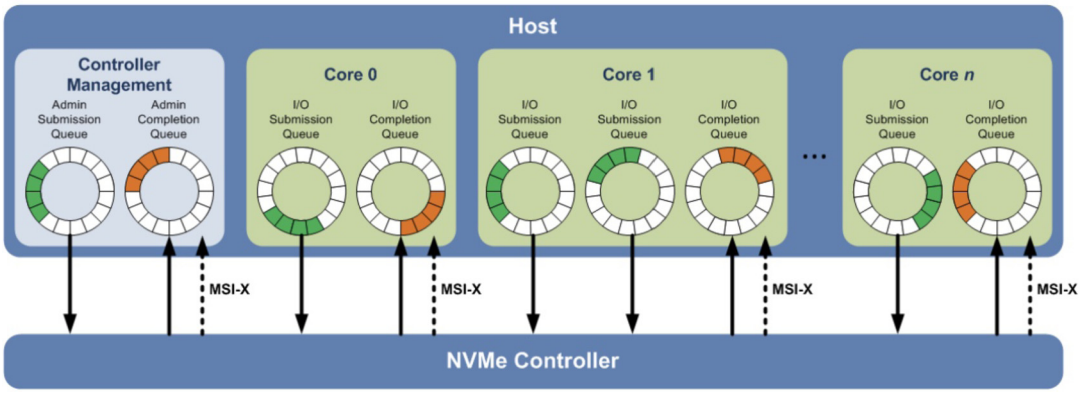

FIGURE 7.1

NVMe controller allows each core to have multiple queues for outstanding operations and to hide the latency of individual operations.

support up to 256 commands; and SATA (Serial AT Attachment) devices support up to 32 commands in a single queue.

NVMe supports 64K commands per queue and up to 64K queues. These queues are designed such that I/O commands and responses to those commands operate on the same processor core and can take advantage of the parallel processing capabilities of multi-core processors. Each application or thread can have its own independent queue, so no I/O locking is required (https://nvmexpress.org/wp-content/uploads/NVMe_Overview.pdf).

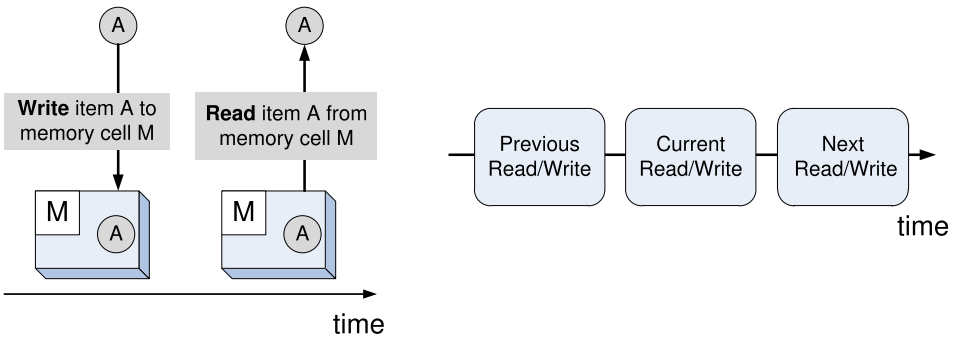

FIGURE 7.2

Semantics of read/write coherence and before-or-after atomicity.

The standard form factor for an SSD is 2.5-inch, which fits inside the drive bay of most laptop or desktop computers. M.2 is a form factor specification for internally mounted SSDs, more expensive per GB than other devices. Samsung 970 EVO Plus M2 NVMe SS has impressive performance: 3 430 GB/sec and 2 813 GB/sec for sequential read and write, respectively, and 278 077 and 190 185 for random IOPS read and write, respectively, according to https://www.samsung.com/semiconductor/minisite/ssd.

7.3 Storage models, file systems, and databases

Storage models describe the layout of a data structure in a physical storage; a data model captures the most important logical aspects of a data structure in a database. Physical storage can be a local disk, a removable media, or storage accessible via the network.

Two abstract models of storage are commonly used: cell storage and journal storage. *Cell storage* assumes that the storage consists of cells of the same size and that each object fits in one cell. This model reflects the physical organization of several storage media; the primary memory of a computer is organized as an array of memory cells and a secondary storage device, e.g., a disk, is organized in sectors or blocks read and written as a unit. Read/write coherence and before-or-after atomicity are two highly desirable properties of any storage model and, in particular, of cell storage; see Fig. 7.2.

Journal storage is a fairly elaborate organization for storing composite objects such as records consisting of multiple fields. Journal storage consists of a manager and a cell storage where the entire history of a variable is maintained, rather than just the current value. The user does not have direct access to the cell storage, instead it can request the journal manager to: (i) start a new action; (ii) read the value of a cell; (iii) write the value of a cell; (iv) commit an action; and (v) abort an action. The

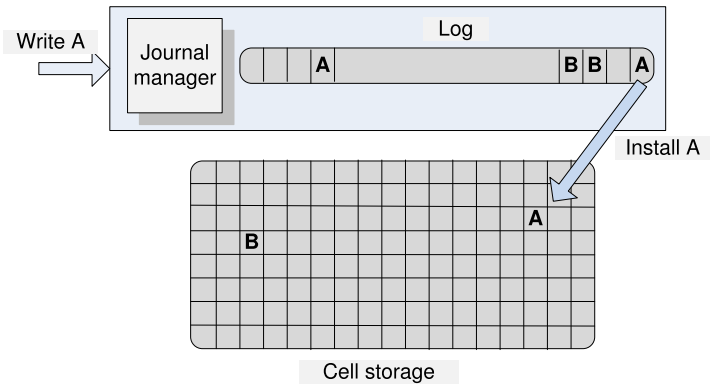

FIGURE 7.3

A log contains the entire history of all variables; the log is stored on a nonvolatile media of a journal storage. If the system fails after the new value of a variable is stored in the log, but before the value is stored in the cell memory, the value can be recovered from the log. If the system fails while writing the log, the cell memory is not updated. This guarantees that all actions are all-or-nothing. Two variables **A** and **B** in the log and the cell storage are shown. A new value of **A** is written first to the log and then installed on cell memory at the unique address assigned to **A**.

journal manager translates user requests into commands sent to the cell storage: (i) read a cell; (ii) write a cell; (iii) allocate a cell; and (iv) deallocate a cell.

The log of a storage system contains a history of all variables in a cell storage. Information about the updates of each data item forms a record appended to the end of the log. A log provides authoritative information about the outcome of an action involving the cell storage; the cell storage can be reconstructed using the log which can be easily accessed, we only need a pointer to the last record.

An all-or-nothing action first records the action in a log in journal storage and then installs the change in the cell storage by overwriting the previous version of a data item; see Fig. 7.3. The log is always kept on non-volatile storage, e.g., disk, and the considerably larger cell storage resides typically on nonvolatile memory, but can be held in memory for real-time access or using a write-through cache.

A file system consists of a collection of directories and each directory provides information about a set of files. High-performance systems can choose among three classes of file systems: Network File Systems (NFS), Storage Area Networks (SAN), and Parallel File Systems (PFS). Network file systems are very popular and have been used for some time, but do not scale well and have reliability problems; an NFS server could be a single point of failure.

Advances in networking technology allow separation of storage systems from computational servers; the two can be connected by an SAN. SANs offer additional flexibility and allow cloud servers to deal with nondisruptive changes in the storage configuration. Moreover, the storage in an SAN can be *pooled* and then allocated based on the needs of the servers; pooling requires additional software and hardware support and represents another advantage of a centralized storage system. An SAN-based implementation of a file system can be expensive because each node must have a Fibre Channel adapter to connect to the network.

Parallel file systems are scalable, are capable of distributing files across a large number of nodes, and provide a global naming space. In a parallel data system, several I/O nodes serve data to all computational nodes; the system includes also a metadata server that contains information about the data stored in the I/O nodes. The interconnection network of a parallel file system could be a SAN.

Databases and database management systems. Most cloud applications do not interact directly with the file systems but through an application layer that manages a database. A database is a collection of logically related records. The software that controls the access to the database is called a Data Base Management System (DBMS). The main functions of a DBMS are: enforce data integrity, manage data access and concurrency control, and support recovery after a failure.

A DBMS supports a query language, a dedicated programming language used to develop database applications. Several database models, including the navigational model of the 1960s, the relational model of the 1970s, the object-oriented model of the 1980s, and the NoSQL model of the first decade of the 2000s, reflect the limitations of the hardware available at the time and the requirements of the most popular applications of each period.

Cloud databases. Most cloud applications are data-intensive, test the limitations of existing cloud storage infrastructure, and demand database management systems capable of supporting rapid application development and a short time-to-market. Cloud applications require low latency, scalability, high availability, and demand a consistent view of data. These requirements cannot be satisfied simultaneously by existing database models; for example, relational databases are easy to use for application development but do not scale well.

As its name implies, the NoSQL model does not support SQL as a query language and may not guarantee the ACID, Atomicity, Consistency, Isolation, and Durability properties of traditional databases. It usually guarantees an eventual consistency for transactions limited to a single data item. The NoSQL model is useful when the structure of the data does not require a relational model and the amount of data is very large. Several types of NoSQL databases have emerged in the last few years. Based on the manner the NoSQL databases store the data, we recognize several types such as key-value stores, BigTable implementations, document store databases, and graph databases.

Replication, used to ensure fault tolerance of large-scale systems built with commodity components, requires mechanisms to guarantee that replicas are consistent with one another. This is yet another example of increased complexity of modern computing and communication systems when the software has to support desirable properties of the physical systems. Section 7.7 contains an in-depth analysis of a service implementing a consensus algorithm to guarantee that replicated objects are consistent.

Many cloud applications support online transaction processing and have to guarantee the correctness of the transactions. Transactions consist of multiple actions; for example, the transfer of funds from one account to another requires withdrawing funds from one account and crediting it to another. The system may fail during or after each one of the actions, and steps to ensure correctness must be taken. Correctness of a transaction means that the result should be guaranteed to be the same as if the actions were applied one after another regardless of the order. More stringent conditions must sometimes be observed; for example, banking transactions must be processed in the order they are issued, the so-called *external time consistency*. To guarantee correctness, a transaction processing system supports *all-or-nothing atomicity*, discussed in Section 10.12.

7.4 Distributed file systems; the precursors

The first distributed file systems were developed in the 1980s by software companies and universities. The systems covered are: NFS, developed by Sun Microsystems in 1984; AFS, developed at Carnegie Mellon University as part of the Andrew project; and SFS, developed by John Osterhout's group at U.C. Berkeley as a component of the Unix-like distributed operating system called Sprite. Other systems developed at about the same time are Locus [500], Apollo [299], and Remote File System (RFS) [41]. Main concerns in the design of these systems are scalability, performance, and security; see Table 7.3.

In 1980s, many organizations, including research centers, universities, financial institutions, and design centers, considered that networks of workstations are an ideal environment for their operations. Diskless workstations were appealing due to reduced hardware costs and lower maintenance and system administration costs. Soon, it became obvious that a distributed file system could be very useful for the management of a large number of workstations, and Sun Microsystems, one of the main promoters of a distributed systems based on workstations, proceeded to develop the NFS in early 1980s.

Network File System. NFS was the first widely used distributed file system; the development of this application based on the client-server model was motivated by the need to share a file system among a number of clients interconnected by a local area network.

A majority of workstations were running under UNIX; thus, many design decisions for the NFS were influenced by the design philosophy of the UNIX File System (UFS). It is not surprising that the NFS designers aimed to: (i) provide the same semantics as a local UFS to ensure compatibility with existing applications and facilitate easy integration into existing UFS; (ii) ensure that the system will be widely used, thus support clients running on different operating systems; and (iii) accept a modest performance degradation due to remote access over a network with a bandwidth of several Mbps.

UFS has three important characteristics enabling extension from local to remote file management:

1. Layered design provides the necessary flexibility of the file system. Layering allows separation of concerns and minimization of the interaction among the modules necessary to implement the system. The addition of the vnode layer allowed UNIX file system to treat uniformly local and remote file access.
2. Hierarchical design supports file system scalability; it allows grouping of files into special files called directories, supports multiple levels of directories and collections of directories and files, the so-called file systems. The hierarchical file structure is reflected by the file-naming convention.
3. Metadata supports a systematic rather than an ad hoc design philosophy of the file system. The inodes contain information about individual files and directories and are kept on persistent media together with the data. Metadata includes the file owner, the access rights, the creation time or the time of the last modification of the file, and the file size, as well as information about the structure of the file and the persistent storage device cells where the data is stored. Metadata also supports device independence, a very important objective due to the very rapid pace of storage technology development.

The *logical organization* of a file reflects the data model, the view of the data from the perspective of the application. The *physical organization* reflects the storage model and describes the manner the file is stored on a given storage media. The layered design allows UFS to separate the concerns for the physical file structure from those for the logical one.

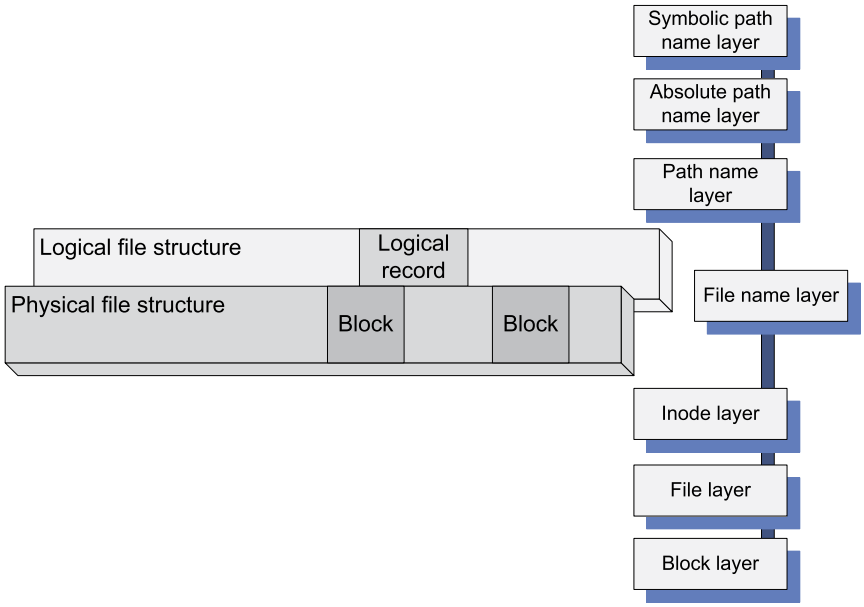

FIGURE 7.4

UFS layered design separates the physical file structure from the logical one. The lower three layers, block, file, and inode, are related to the physical file structure, while the upper three layers, path name, absolute path name, and symbolic path name, reflect the logical organization. The file name layer mediates between the two groups.

Recall that a file is a linear array of cells stored on a persistent storage device; the file pointer identifies a cell used as a starting point for a read or write operation. This linear array is viewed by an application as a collection of logical records; the file is stored on a physical device as a set of physical records, or blocks, of size dictated by the physical media.

The lower three layers of the UFS hierarchy, the block, the file, and the inode layer, reflect the physical organization. The block layer allows the system to locate individual blocks on the physical device; the file layer reflects the organization of blocks into files; and the inode layer provides the metadata for the objects (files and directories). The upper three layers, the path name, the absolute path name, and symbolic path name layer, reflect the logical organization. The file name layer mediates between the machine-oriented and the user-oriented views of the file system; see Fig. 7.4.

Several control structures maintained by the kernel of the operating systems support the file handling by a running process; these structures are maintained in the user area of the process address space and can only be accessed in kernel mode. To access a file, a process must first establish a connection with the file system by opening the file; at that time, a new entry is added to the file description table, and the metainformation is brought in to another control structure, the open file table.

A *path* specifies the location in a file system of a file or directory; a *relative path* specifies this location relative to the current/working directory of the process, while a *full path*, also called an absolute

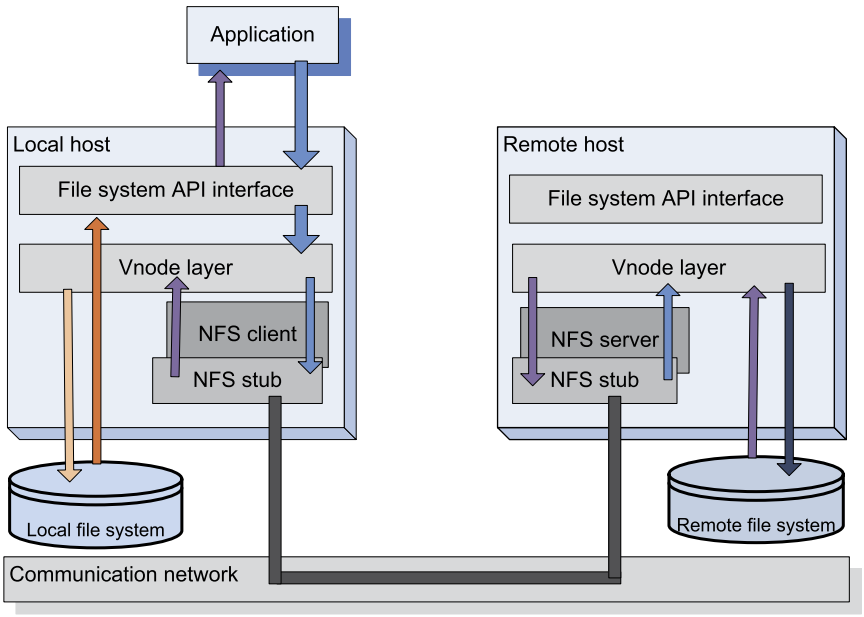

FIGURE 7.5

The NFS client-server interaction. The vnode layer implements file operation in a uniform manner, regardless of whether the file is local or remote. An operation targeting a local file is directed to the local file system, while one for a remote file involves NFS; an NSF client packages the relevant information about the target and the NFS server passes it to the vnode layer on the remote host, which, in turn, directs it to the remote file system.

path, specifies file location independently of the current directory, typically relative to the root directory. A local file is uniquely identified by a *file descriptor* (fd), generally, an index in the open file table.

NFS is based on the client–server paradigm. The client runs on the local host, while the server is at the site of the remote file system, and they interact by means of Remote Procedure Calls (RPCs). NFS uses a vnode layer to distinguish between operations on local and remote files; see Fig. 7.5. The API interface of the local file system distinguishes file operations on a local file from the ones on a remote file and, in the latter case, invokes the RPC client. Fig. 7.6 shows the API for a UNIX file system and the calls made by the RPC client in response to API calls issued by a user program for a remote file system, as well as some of the actions carried out by the NFS server in response to an RPC call.

A remote file is uniquely identified by a file handle rather than a file descriptor; a file handle is a 32-byte internal name, a combination of a file system identification, an inode number, and a generation number. A file handle allows a host to locate the remote file system and the file on that system. A generation number allows the system to reuse inode numbers and ensures a correct semantics when multiple clients operate on the same remote file.

While many RPC calls, such as *Read*, are idempotent, an action is idempotent if repeating it several times has the same effect as if the action was executed only once. Communication failures could sometimes lead to an unexpected behavior. Indeed, if the network fails to deliver the response to a *Read* RPC,

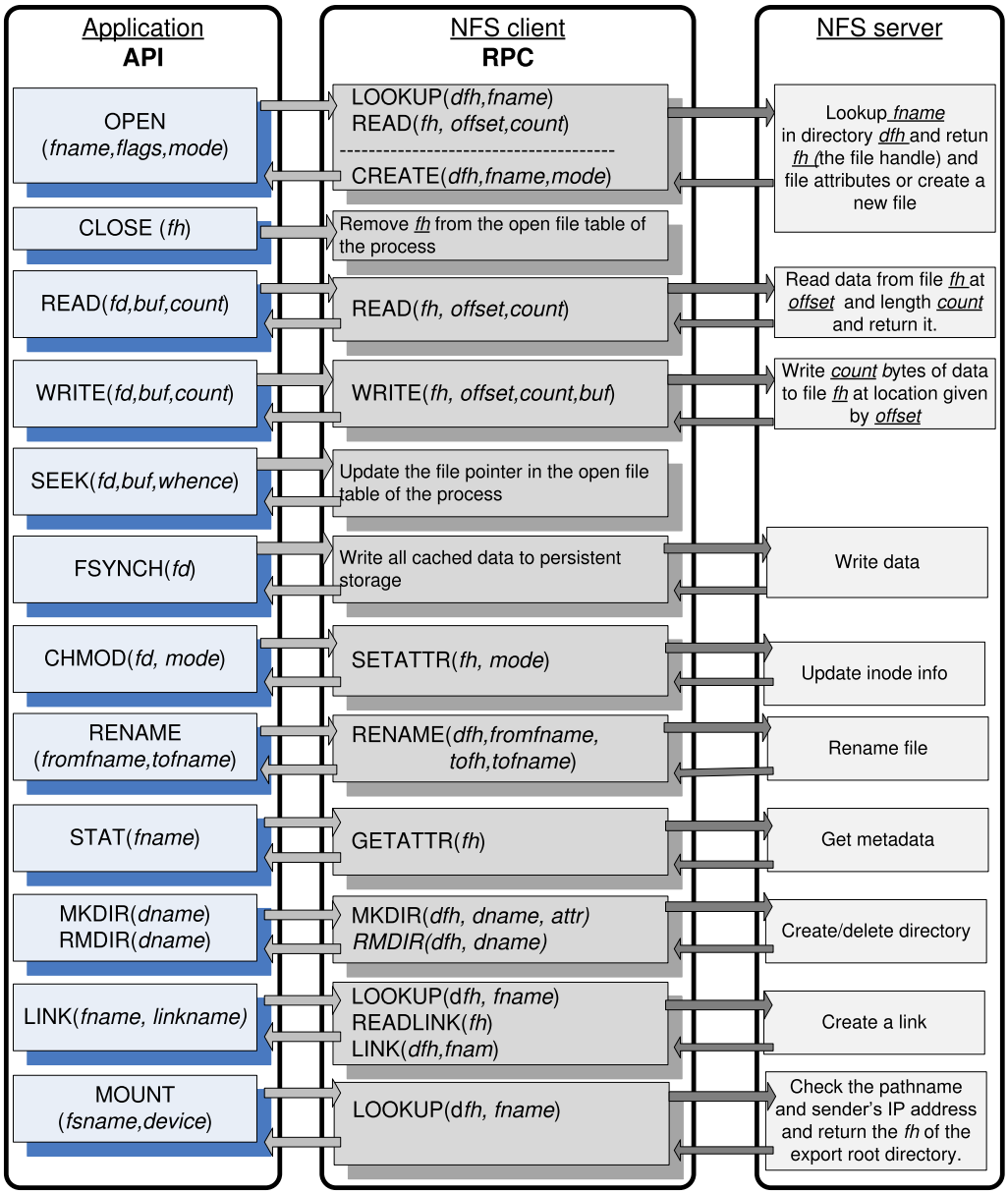

FIGURE 7.6

The API of UFS and the corresponding RPCs issued by an NFS client to the NFS server. The actions of the server in response to an RPC issued by the NFS client are too complex to be fully described here. *fd* stands for file descriptor, *fh* for file handle, *fname* for file name, *dname* for directory name, *dfh* for the directory where the file handle can be found, *count* for the number of bytes to be transferred, *buf* for the buffer to transfer the data to/from, and *device* for the device where the file system is located.

then the call can be repeated without any side effects. By contrast, when the network fails to deliver the response to the *Rmdir* RPC, the second call returns an error code to the user if the call was successful the first time; if the server fails to execute the first call, then the second call returns normally. Note also that there is no *Close* RPC because this action only makes changes to the open file data structure of the process and does not affect the remote file.

NFS has undergone significant transformations over the years; it has evolved from Version 2 [433], discussed in this section, to Version 3 [392] in 1994, and then to Version 4 [393] in 2000; see Section 7.16.

Andrew File System. AFS is a distributed file system developed in the late 1980s at Carnegie Mellon University in collaboration with IBM [358]. System designers envisioned a very large number of workstations interconnected with a relatively small number of servers. It was anticipated that each individual at CMU would have an Andrew workstation, thus the system would connect up to 10 000 workstations.

The set of trusted servers in AFS form a structure called Vice. The workstation OS, 4.2BSD UNIX, intercepts file system calls and forwards them to a user-level process called Venus which caches files from Vice and stores modified copies of files back on the servers they came from. Reading and writing operations are performed directly on the cached copy of the file and bypass Venus. Only when a file is opened or closed does Venus communicate with Vice.

The emphasis of the AFS design is on performance, security, and simple management of the file system [245]. The local disk of a workstation acts as a persistent cache ensuring scalability and reducing the response time. The master copy of a file residing on one of the servers is updated only when the file is modified. This strategy reduces the server load and improves the system performance.

Another major objective of the AFS design is improved security. The communications between clients and servers are encrypted, and all file operations require secure network connections. When a user signs in to a workstation, the password is used to obtain security tokens from an authentication server; these tokens are then used every time a file operation requires a secure network connection.

The AFS uses Access Control Lists (ACLs) to allow control sharing of the data. An ACL specifies the access rights of an individual user or of a group of users. A set of tools support the management of ACLs. Another facet of the effort to reduce the user involvement in the file management is *location transparency*. Files could be accessed from any location and could be moved automatically, or at the request of system administrators, without user's involvement and inconvenience. The relatively small number of servers reduces drastically the efforts related to system administration because operations, such as backups, affect only the servers, while workstations can be added, removed, or moved from one location to another without administrative intervention.

Sprite Network File System. SFS is a component of the Sprite network operating system [237]. SFS supports non-write-through caching of files on the client, as well as the server systems [363]. Processes running on all workstations enjoy the same semantics for file access as if they would run on a single system; this is possible due to a cache consistency mechanism that flushes portions of the cache and disables caching for shared files opened for read-write operations.

Caching not only hides the network latency but also reduces the server utilization and improves the performance by reducing the responser time. A file access request made by a client process could be satisfied at various levels. First, the request is directed to the local cache; if not satisfied there, it is passed to the local file system of the client. If it cannot be satisfied locally, then the request is sent to the remote server; if the request cannot be satisfied by the remote server's cache, then it is sent to the file system running on the server.

Sprite system design decisions were influenced by resources available when a typical workstation had a 1 to 2 MIPS processor and 4 to 14 Mbytes of physical memory. The main-memory caches allowed diskless workstations to be integrated in the system and enabled the development of unique caching mechanisms and policies for both clients and servers. The results of a file-intensive benchmark reported by [363] show that SFS was 30 to 35% faster than either NFS or AFS.

The file cache is organized as a collection of 4K blocks; a cache block has a virtual address consisting of a unique file identifier supplied by the server and a block number in the file. Virtual addressing allows the clients to create new blocks without the need to communicate with the server; file servers map virtual addresses to physical disk addresses. Note also that the page size of the virtual memory in Sprite is also 4K. The size of the cache available to an SFS client or a server system changes dynamically function of the needs. This is possible because the Sprite operating system ensures an optimal sharing of the physical memory between file caching by SFS and virtual memory management.

The file system and the virtual memory manage separate sets of physical memory pages and maintain a time-of-last-access for each block or page, respectively. Virtual memory uses a version of the clock algorithm [362] to implement a Least Recently Used (LRU) page replacement algorithm, and the file system implements a strict LRU order since it knows the time of each read and write operation. Whenever the file system or the virtual memory management experiences a file cache miss or a page fault, it compares the age of its oldest cache block or page, respectively, with the age of the oldest one of the other system; the oldest cache block or page is forced to release the real memory frame.

An important design decision of SFS was to delay write-backs; this means that a block is first written to cache, and the writing to the disk is delayed for a time on the order of tens of seconds. This strategy speeds up writing and also avoids writing when the data is discarded before the time to write it to the disk. The obvious drawback of this policy is that data can be lost in the case of a system failure. A write-through is the alternative to the delayed write-back; it guarantees reliability because the block is written to the disk as soon as it is available on the cache, but it increases the time for a write operation.

Most network file systems guarantee that, once a file is closed, the server will have the newest version on persistent storage. As far as concurrency is concerned, we distinguish sequential write-sharing, when a file cannot be opened simultaneously for reading and writing by several clients, from concurrent write-sharing, when multiple clients can modify the file at the same time. Sprite allows both concurrency modes and delegates cache consistency to the servers. In case of concurrent write-sharing, the client cashing for the file is disabled; all reads and writes are carried out through the server.

Table 7.3 presents a comparison of caching, writing strategy, and consistency of NFS, AFS [358], Sprite [237], Locus [500], Apollo [299], and RFS [41].

7.5 General parallel file system

Once the distributed file systems became ubiquitous, the natural next step in the file systems evolution was supporting parallel access. Parallel file systems allow multiple clients to read and write concurrently from the same file. Support for parallel I/O is essential for the performance of many applications [339]. Early supercomputers such as Intel Paragon took advantage of parallel file systems to support data-intensive applications.

Concurrency control is a critical issue for parallel file systems. Several semantics for handling shared and concurrent file access are possible. One option is to have a shared file pointer; in this case,

Table 7.3 A comparison of several network file systems [358].

File system	Cache size and location	Writing policy	Consistency guarantees	Cache validation
NFS	Fixed, memory	On close or 30 sec. delay	Sequential	On open, with server consent
AFS	Fixed, disk	On close	Sequential	When modified server asks client
SFS	Variable, memory	30 sec. delay	Sequential, concurrent	On open, with server consent
Locus	Fixed, memory	On close	Sequential, concurrent	On open, with server consent
Apollo	Variable, memory	Delayed or on unlock	Sequential	On open, with server consent
RFS	Fixed, memory	Write-through	Sequential, concurrent	On open, with server consent

successive reads issued by different clients advance the file pointer. Another semantics is to allow each client to have its own file pointer.

IBM developed the General Parallel File System [440] in early 2000s as a successor of TigerShark multimedia file system [230]. GPFS emulates closely the behavior of a general-purpose POSIX system running on a single system. This parallel file system was designed for optimal performance of large clusters and can support a file system of up to 4 petabytes consisting of up to 4 096 disks of 1 TB each.

The maximum file size is $(2^{63} - 1)$ bytes. A file consists of blocks of equal size, ranging from 16 KB to 1 MB, striped across several disks. The system could support not only very large files but also a very large number of files. GPFS directories use the *extensible hashing* techniques to access a file.

A hash function is applied to the file name; then, the n low-order bits of the hash value give the block number of the directory where the file information can be found with n, a function of the number of files in the directory. Extensible hashing is used to add a new directory block. The system maintains user data, file metadata, such as last modified time, and file system metadata, such as allocation maps. Metadata, such as file attributes and data block addresses, is stored in inodes and in indirect blocks.

Reliability is a major concern in a system with many physical components. To recover from system failures, GPFS records all metadata updates in a write-ahead log file. *Write-ahead* means that updates are written to persistent storage only after the log records have been written. For example, when a new file is created, a directory bloc must be updated, and an inode for the file must be created. These records are transferred from cache to disk after the log records have been written. When the system ends up in an inconsistent state, the directory bloc is written, and then, if the I/O node fails before writing the inode, the log file allows the system to recreate the inode record.

The log files are maintained by each I/O node for each file system it mounts; thus, any I/O node is able to initiate recovery on behalf of a failed node. Disk parallelism is used to reduce the access time; multiple I/O read requests are issued in parallel, and data is pre-fetched in a buffer pool.

Data striping allows concurrent access and improves performance but can have unpleasant side effects. Indeed, when a single disk fails, a large number of files are affected. To reduce the impact of such undesirable events, the system attempts to mask a single disk failure or the failure of the access path to a disk. The system uses RAID devices with the stripes equal to the block size and dual-attached

RAID controllers. To further improve the fault tolerance of the system, GPFS data files and metadata are replicated on two different physical disks.

Consistency and performance, critical for any distributed file system, are difficult to balance; support for concurrent access improves the performance but faces serious challenges for maintaining consistency. GPFS consistency and synchronization are ensured by a distributed locking mechanism; a *central lock manager* grants *lock tokens* to *local lock managers* running in each I/O node. Lock tokens are also used by the cache management system.

Lock granularity has important implications for the performance of a file system, and GPFS uses a variety of techniques for different types of data. *Byte-range tokens* are used for read and write operations to data files as follows: The first node attempting to write to a file acquires a token covering the entire file, $[0, \infty]$. This node is allowed to carry out all reads and writes to the file without any need for permission until a second node attempts to write to the same file; then, the range of the token given to the first node is restricted. More precisely, if the first node writes sequentially at offset fp_1 and the second one at offset $fp_2 > fp_1$, then the ranges of the two tokens are $[0, fp_2]$ and $[fp_2, \infty]$, respectively, and the two nodes can operate concurrently without the need for further negotiations. Byte-range tokens are rounded to block boundaries.

Byte-range token negotiations among nodes use the *required range* and the *desired range* for the offset and for the length of the current and the future operations, respectively. The *data-shipping*, an alternative to byte-range locking, allows fine-grain data sharing. In this mode, the file blocks are controlled by the I/O nodes in a round-robin manner. A node forwards a read or write operation to the node controlling the target block, the only one allowed to access the file.

A *token manager* maintains the state of all tokens; it creates and distributes tokens, collects tokens once a file is closed, and downgrades/upgrades tokens when additional nodes request access to a file. Token management protocols attempt to reduce token manager load; for example, when a node wants to revoke a token it sends messages to all other nodes holding the token and forwards the reply to the token manager.

Access to metadata is synchronized; for example, when multiple nodes write to the same file, the file size and the modification dates are updated using a *shared write lock* to access an inode. One of the nodes assumes the role of a *metanode* and all updates are channeled through it, the file size and the last update time are determined by the metanode after merging the individual requests. The same strategy is used for updates of the indirect blocks. GPFS global data, such as ACLs (Access Control Lists), quotas, and configuration data, are updated using the distributed locking mechanism.

GPFS uses *disk maps* for the management of the disk space. The GPFS block size can be as large as 1 MB, and a typical block size is 256 KB. A block is divided into 32 sub-blocks to reduce disk fragmentation for small files, thus the block map has 32 bits to indicate if a subbloc is free or used. The system disk map is partitioned into n regions, and each disk map region is stored on a different I/O node; this strategy reduces the conflicts and allows multiple nodes to allocate disk space at the same time. An *allocation manager* running on one of the I/O nodes is responsible for actions involving multiple disk map regions. For example, it updates free space statistics and helps with deallocation by sending periodically hints of the regions used by individual nodes.

A detailed discussion of system utilities and of the lessons learned from the deployment of the file system at several installations in 2002 can be found in [440]; the documentation of the GPFS is available from [250].

7.6 Google file system

Google File System, developed in late 1990s, uses thousands of storage systems built from inexpensive commodity components to provide petabytes of storage to a large user community with diverse needs [197]. Thus, it should not be surprising that a main concern of GFS designers was reliability of a system exposed to hardware failures, system software errors, application errors, and, last but not least, human errors. The system was designed after a careful analysis of file characteristics and access models. Some of the most important aspects of this analysis reflected in the GFS design are:

a. Scalability and reliability are critical features of the system; they must be considered from the beginning, rather than at a later design stage.
b. The vast majority of files range in size from a few GB to hundreds of TB.
c. The most common operation is to append to an existing file; random write operations to a file are extremely infrequent.
d. Sequential read operations are the norm.
e. Users process the data in bulk and are less concerned with the response time.
f. The consistency model should be relaxed without placing an additional burden on the application developers to simplify system implementation.

This analysis led to several major design decisions:

1. Segment a file in large chunks.
2. Implement an atomic file *append* operation allowing multiple applications operating concurrently to append to the same file.
3. Build the cluster around a high-bandwidth rather than low-latency interconnection network. Separate the flow of control from the data flow; schedule the high-bandwidth data flow by pipelining the data transfer over TCP connections to reduce the response time. Exploit network topology by sending data to the closest node in the network.
4. Eliminate client site caching; caching increases the overhead for maintaining consistency among cached copies at multiple client sites and is not likely to improve performance.
5. Ensure consistency by channeling critical file operations through a master controlling the entire system.
6. Minimize master's involvement in file access operations to avoid hotspot contention and to ensure scalability.
7. Support efficient checkpointing and fast recovery mechanisms.
8. Support efficient garbage-collection mechanisms.

GFS files are collections of fixed-size segments called *chunks*; at the time of file creation, each chunk is assigned a unique *chunk handle*. A chunk consists of 64 KB blocks and each block has a 32 bit checksum. Chunks are stored on Linux files systems and are replicated on multiple sites; a user may change the number of the replicas, from the standard value of three to any desired value. The chunk size is 64 MB; this choice is motivated by the desire to optimize the performance for large files and to reduce the amount of metadata maintained by the system.

A large chunk size increases the likelihood that multiple operations will be directed to the same chunk, thus it reduces the number of requests to locate the chunk, and, at the same time, it allows an application to maintain a persistent network connection with the server where the chunk is located.

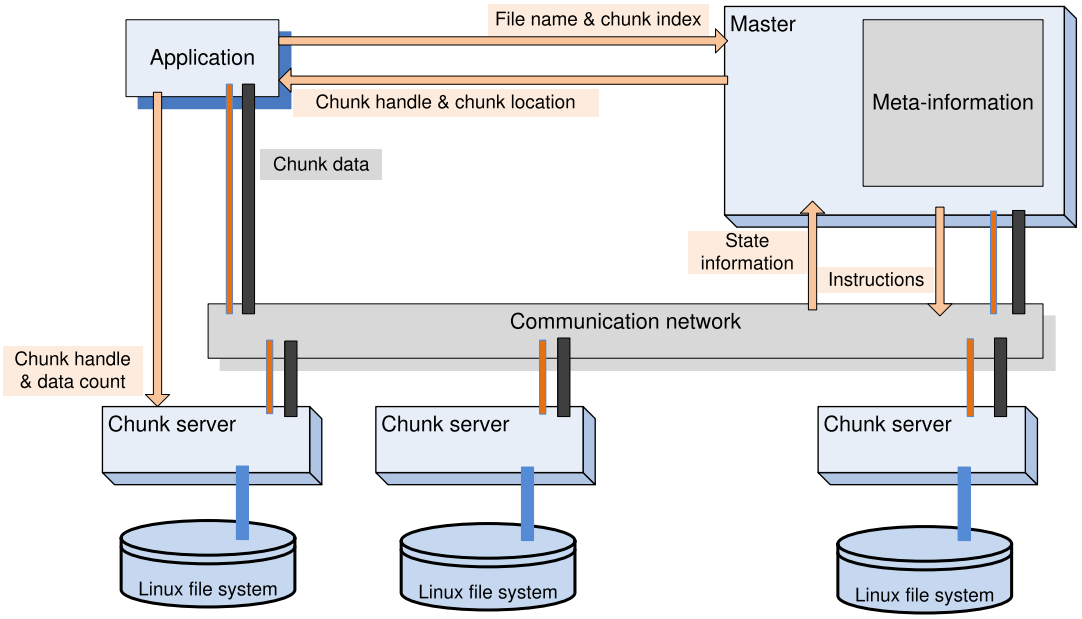

FIGURE 7.7

GFS cluster architecture. The *master* maintains state information about all system components and controls a number of *chunk servers*. A chunk server runs under Linux and uses metadata provided by the master to communicate directly with an application. The data flow is decoupled from the control flow. The data and the control paths are shown separately, data paths with thick lines and the control paths with thin lines. Arrows show the flow of control between an application, the master, and the chunk servers.

Space fragmentation occurs infrequently because the chunk of a small file and the last chunk of a large file are only partially filled.

The architecture of a GFS cluster is illustrated in Fig. 7.7. The *master* controls a large number of *chunk servers*; it maintains metadata such as the file names, access control information, the location of all the replicas for every chunk of each file, and the state of individual chunk servers. Some of the metadata is stored in persistent storage, e.g., the *operation log* records the file namespace, as well as the file-to-chunk-mapping. Chunks location is stored only in the control structure of master's memory and is updated at the system start up, or when a new chunk server joins the cluster. This strategy allows the master to have up-to-date information about the location of the chunks.

System reliability is a major concern, and the operation log maintains a historical record of metadata changes enabling the master to recover in case of a failure. As a result, such changes are atomic and are not made visible to the clients until they have been recorded on multiple replicas on persistent storage. To recover from a failure, the master replays the operation log. To minimize the recovery time, the master periodically checkpoints its state and, at recovery time, it replays only the log records after the last checkpoint.

Each chunk server is a commodity Linux system. A chunk server receives instructions from the master and responds with status information. For file read or write operations an application sends to the master the file name, the chunk index, and the offset in the file. The master responds with the chunk handle and the location of the chunk. Then the application *communicates directly* with the chunk server to carry out the desired file operation.

The consistency model is very effective and scalable. Operations, such as file creation, are atomic and are handled by the master. To ensure scalability, the master has a minimal involvement in file mutations, operations such as *write* or *append,* that occur frequently. In such cases the master grants a lease for a particular chunk to one of the chunk servers called the *primary*; then, the primary creates a serial order for the updates of that chunk.

When a data for a *write* straddles the chunk boundary, two operations are carried out, one for each chunk. The steps for a *write* request illustrate a process that buffers data and decouples the control flow from the data flow for efficiency:

i. The client contacts the master, which assigns a lease to one of the chunk servers for the a particular chunk, if no lease for that chunk exists; then, the master replies with the id of the primary as well as secondary chunk servers holding replicas of the chunk. The client caches this information.
ii. The client sends the data to all chunk servers holding replicas of the chunk; each one of the chunk servers stores the data in an internal LRU buffer and then sends an acknowledgment to the client.
iii. The client sends the *write* request to the primary chunk server once it has received the acknowledgments from all chunk servers holding replicas of the chunk. The primary chunk server identifies mutations by consecutive sequence numbers.
iv. The primary chunk server sends the *write* requests to all secondaries.
v. Each secondary chunk server applies the mutations in the order of the sequence number and then sends an acknowledgment to the primary one.
vi. Finally, after receiving the acknowledgments from all secondaries, the primary informs the client.

The system supports an efficient checkpointing procedure based on *copy-on-write* to construct system snapshots. A lazy garbage-collection strategy is used to reclaim the space after a file deletion. As a first step, the file name is changed to a hidden name and this operation is time stamped. The master periodically scans the namespace and removes metadata for files with a hidden name older than a few days. This mechanism enables a user who deleted files by mistake to recover the files with little effort.

Periodically, chunk servers exchange with the master the list of chunks stored on each one of them; the master supplies them with the identity of orphaned chunks, whose metadata has been deleted, and such chunks are then deleted. Even when control messages are lost, a chunk server will carry out the house cleaning at the next *heartbeat* exchange with the master. Each chunk server maintains in core the checksums for the locally stored chunks to guarantee data integrity. *CloudStore* is an open source C++ implementation of GFS. CloudStore allows client access from C++, Java, and Python.

7.7 Locks; Chubby—a locking service

Locks support the implementation of reliable storage for loosely coupled distributed systems. Locks enable controlled access to shared storage and ensure atomicity of *read* and *write* operations. Consensus protocols are critically important for the design of reliable distributed storage systems. The election of

a leader or master from a group of data servers requires an effective consensus protocol because the master plays an important role in the management of a distributed storage system. For example, in GFS, the master maintains state information about all systems components.

Locking and the election of a master can be done using a version of the Paxos algorithm for asynchronous consensus. The algorithm guarantees safety without any timing assumptions, a necessary condition in a large-scale system when communication delays are unpredictable. Nevertheless, the algorithm must use clocks to ensure liveliness and to avoid the impossibility of reaching consensus with a single faulty process [175]. Coordination and consensus using Paxos are discussed in depth in Sections 11.4 and 10.13, respectively.

Distributed systems experience communication problems, such as lost messages, messages out of sequence, or corrupted messages. There are solutions for handling these undesirable phenomena; for example, one can use virtual time, i.e., sequence numbers, to ensure that messages are processed in an order consistent with the time they were sent by all processes involved, but this complicates the algorithms and increases the processing time.

Advisory locks are based on the assumption that all processes play by the rules; advisory locks do not have any effect on processes that circumvent the locking mechanisms and access the shared objects directly. *Mandatory locks* block access to the locked objects to all processes that do not hold the locks, regardless if they use locking primitives or not.

Locks held for a very short time are called *fine-grained*, while *coarse-grained* locks are held for a longer time. Some operations such as accessing the name of the lock, whether the lock is shared or held in exclusivity, the generation number of the lock require meta-information about the lock. The metainformation is sometimes aggregated into an opaque byte-string called a *sequencer*. The question how to most effectively support a locking and consensus component of a large-scale distributed system demands several design decisions.

A first decision is whether the locks should be mandatory or advisory. Mandatory locks have the obvious advantage of enforcing access control; a traffic analogy is that a mandatory lock is like a drawbridge: Once it is up, all traffic is forced to stop. An advisory lock is like a stop sign: Those who obey the traffic laws will stop, but some may not. The disadvantages of mandatory locks are added overhead and less flexibility. Once a data item is locked, even a high-priority task related to maintenance or recovery cannot access the data unless it forces the application holding the lock to terminate. This is a very significant problem in large-scale systems where partial system failures are likely.

A second design decision is whether the system should be based on fine-grained or coarse-grained locking. *Fine-grained locks* allow more application threads to access shared data, but generate a larger workload for the lock server. Moreover, when the lock server fails for a period of time, a larger number of applications are affected. Advisory locks and *coarse-grain locks* seem to be a better choice for a system expected to scale to a very large number of nodes distributed in data centers interconnected via wide area networks with a higher communication latency. A third design decision is how to support a systematic approach to locking. Two alternatives come to mind: (i) delegate the implementation of the consensus algorithm to the clients and provide a library of functions needed for this task; and (ii) create a locking service implementing a version of the asynchronous Paxos algorithm and provide a library to be linked with an application client to support service calls. Forcing application developers to invoke calls to a Paxos library is more cumbersome and more prone to errors than the service alternative. Of course, the lock service itself has to be scalable to support a potentially heavy load.

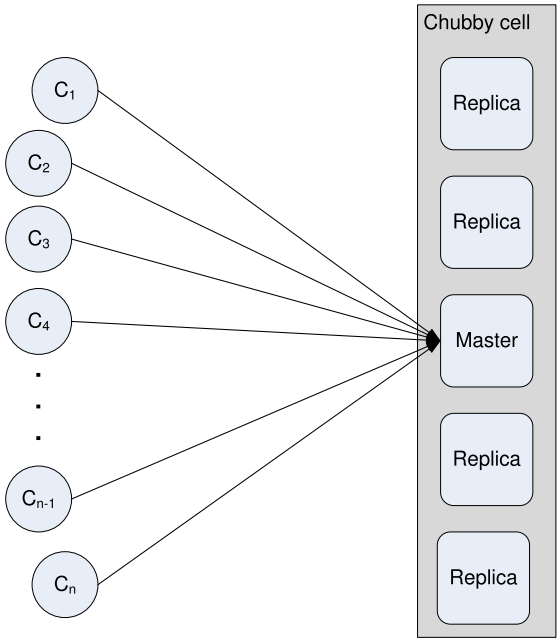

FIGURE 7.8

A Chubby cell consisting of five replicas, one of them is elected as the master. Clients $c_1, c_2, \ldots, c_n$ communicate with the master using RPCs.

Another consideration when making this choice is flexibility: the ability of the system to support a variety of applications. A name service comes to mind as many cloud applications *read* and *write* small files. To allow atomic file operations, the names of small files should be included in the namespace of the service. The choice should also consider the performance, if a service can be optimized and if clients can be allowed to cache control information. Lastly, the overhead and resources for reaching consensus should be considered. Again, the service alternative seems more advantageous because it needs fewer replicas for high availability.

In 2000 Google decided to use advisory locks and coarse-grained locks and started developing Chubby [81], a lock service. The service has been used by several Google systems including GFS discussed in Section 7.6 and BigTable presented in Section 7.11. A Chubby cell typically serves one data center. The cell server in Fig. 7.8 includes several *replicas*; the standard number of replicas is five. To reduce the probability of correlated failures, the servers hosting replicas are distributed across the campus of a data center.

Chubby replicas use a distributed consensus protocol to elect a new *master* when the current one fails. A master is elected by a majority, as required by the asynchronous Paxos algorithm, accompanied by the commitment that a new master will not be elected for a period of time, namely, the *master lease.* A session is a connection between a client and the cell server maintained over a period of time. Data cached by a client, locks acquired, and handles of all files locked by the client are only valid for the

duration of the session. Clients use RPCs to request services from the master. The master responds without consulting replicas when receiving a *read* request, propagates the request to all replicas, and waits for a reply from a majority of replicas before responding to a *write* request.

The client interface of the system is similar, yet simpler, than the one supported by the Unix file system; in addition, it includes notification for events related to file or system status. A client can subscribe to events such as: file contents modification, change or addition of a child node, master failure, lock acquired, conflicting lock requests, and invalid file handle. Files and directories of Chubby service are organized in a tree structure and use a naming scheme similar to Unix. Each file has a *file handle* similar to the file descriptor.

The master of a cell periodically writes a snapshot of its database to a GFS file server. Every file or directory can act as a lock. To *write* to a file, the client must be the only one holding the file handle, while multiple clients may hold the file handle to *read* from the file. Handles are created by a call to *open()* function and destroyed by a call to *close()*. Other calls supporting the service are *GetContentsAndStat()*, to get the file data and meta-information, and *SetContents*, *Delete()*.

Several calls allow the client to acquire and release locks. Some applications may decide to create and manipulate a sequencer with calls to: *SetSequencer()* for associating a sequencer with a handle; *GetSequencer()* for obtaining the sequencer associated with a handle; or *CheckSequencer()* for checking the validity of a sequencer.

The sequence of calls *SetContents()*, *SetSequencer()*, *GetContentsAndStat()*, and *CheckSequencer()* can be used by an application for the election of a master. In this process all candidate threads attempt to open a lock file, call it *lfile*, in exclusive mode. The one that succeeds to acquire the lock for *lfile*, becomes the master, writes its identity in *lfile*, creates a sequencer for the lock of *lfile*, call it *lfseq*, and passes it to the server. The other threads read the *lfile* and discover that they are replicas. Periodically, they check the sequencer *lfseq* to determine if the lock is still valid. The example illustrates the use of Chubby as a name server; in fact, this is one of the most frequent uses of the system.

Chubby locks and Chubby files are stored in a replicated database. The architecture of these replicas shows that the stack consists of: (i) the Chubby component implementing the Chubby protocol for communication with the clients; and (ii) the active components writing log entries and files to the local storage of the replica; see Fig. 7.9.

An *atomicity log* for a transaction processing system allows a crash recovery procedure to undo all-or-nothing actions that did not complete or to finish all-or-nothing actions that committed but did not record all of their effects. Each replica maintains its own copy of the log; a new log entry is appended to the existing log, and the Paxos algorithm is executed repeatedly to ensure that all replicas have the same sequence of log entries.

The next element of the stack is responsible for the maintenance of a fault-tolerant database, in other words, that all local copies are consistent. Fault tolerance is a property enabling a system to continue operating properly in the event of the failure of one or more faults of some of its components. The database consists of the actual data, or the *local snapshot* in Chubby speak, and a *replay log* to allow recovery in case of failure. The state of the system is also recorded in the database.

Paxos algorithm is used to reach consensus on sets of values, e.g., the sequence of entries in a replicated log. To ensure that the algorithm succeeds, in spite of the occasional failure of a replica, the following three phases of the algorithm are executed repeatedly:

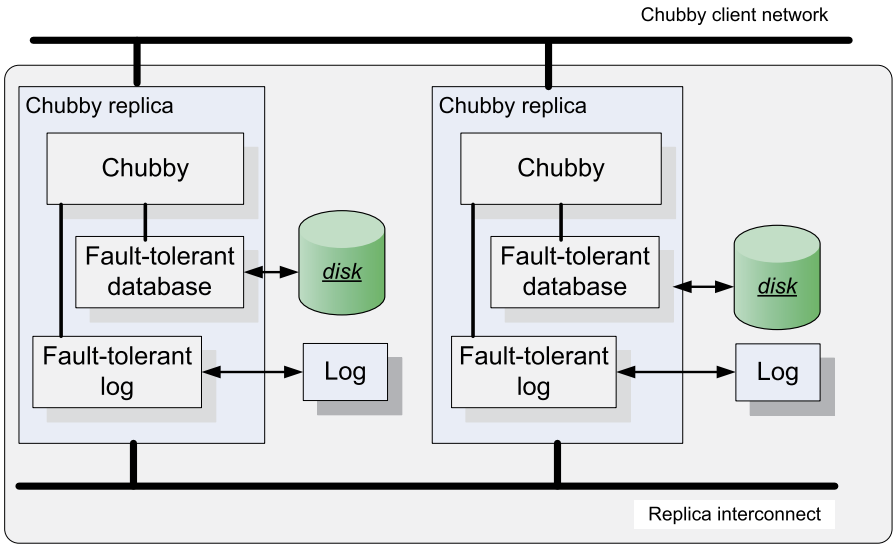

FIGURE 7.9

Chubby replica architecture. Chubby implements the communication protocol with clients. The system includes a component to transfer files to a fault-tolerant database and a fault-tolerant log component to write log entries. The fault-tolerant log uses the Paxos algorithm to achieve consensus. Each replica has its own local file system; replicas communicate with one another using a dedicated interconnect and communicate with the clients through a client network.

i. Elect a replica to be the master/coordinator. When a master fails, several replicas may decide to assume the role of a master. To ensure that the result of the election is unique, each replica generates a sequence number larger than any sequence number it has seen, in the range $(1, r)$ where r is the number of replicas. Then it broadcasts a *propose* message with this sequence number. The replicas that have not seen a higher sequence number broadcast a *promise* reply and declare that they will reject proposals from other candidate masters. The replica who sent the *propose* message is elected as the master if the number of respondents represent a majority of replicas.
ii. The master broadcasts to all replicas an *accept* message, including the value it has selected and waits for replies, either *acknowledge* or *reject*.
iii. Consensus is reached when the majority of the replicas send the *acknowledge* message; then, the master broadcasts the *commit* message.

Paxos algorithm implementation is far from trivial: While the algorithm can be expressed as a few tens of lines of pseudocode, its actual implementation could be several thousand lines of C++ code [92]. Moreover, the practical use of the algorithm cannot ignore the wide variety of failure modes, including algorithm errors and bugs in its implementation, and testing a software system of a few thousands lines of codes is challenging.

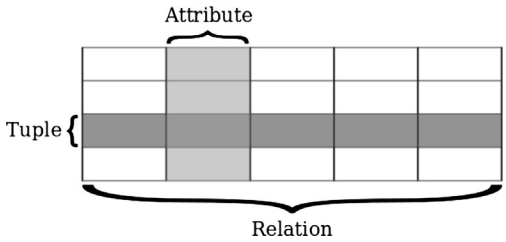

FIGURE 7.10

Data organization in relational databases.

7.8 RDBMS—cloud mismatch

Before the age of cloud computing, several data models were widely used: the *hierarchical* model for strictly hierarchical relations, the *network* model for many-to-many relationships, and, the most ubiquitous of all, the *relational* model (RDBMS) of Edgar F. Codd from IBM Almaden Research Center. Codd proposed a set of 12 rules for the model [111], but his rules are usually condensed into one sentence: *present data as relations and provide relational operators to manipulate data.*

The relational model organizes data into *tables/relations* representing one *entity type*. Tables consists of *columns/attributes* and rows, called records/tuples; a unique *key* identifies each row. Rows represent instances of a type of entity and columns represent values attributed to instance; see Fig. 7.10.

Structured Query Language (SQL) is a special-purpose language for managing structured data in a relational database system and the centerpiece of this storage technology. SQL has three components: a data definition language, a data manipulation language, and a data control language. Oracle, MySQL, IBM DB2, PostgresSQL, Hive, and Microsoft's SQL Server, Access, and Azure SQL Database are widely used relational database systems.

The set of possible values for a given attribute is described by a *domain,* and *constraints* can restrict the types and range of values of an attribute. Indices, created on any combination of attributes of a relation, usually implemented via B+ trees,[2] R-trees,[3] and bitmaps, provide data access. Relational database queries are expressed using *relational algebra* (a set of algebraic structures with a well-defined semantics) and *relational calculus*.

RDBMS instances reference one another and form a graph, but relational schemas are based on relational algebras linking heterogeneous tuples. RDBMS applications are often written in object-oriented (OO) languages, which encapsulate objects hiding their internal representation. Type system differences between RDBMS and OO languages represent a major mismatch between relational semantics and OO languages. Operator semantics and scalar types are another facet of this mismatch. Further-

[2] A B+ tree is an m-arry tree (a rooted tree in which each node has no more than m children), with a variable, but often large, number of children per node. A B+ tree can be viewed as a B-tree in which each node contains only keys, and an additional level is added at the bottom with linked leaves. In a block-oriented storage context, e.g., filesystems, B+ trees, retrieve data efficiently.

[3] R-trees are data structures used for spatial access methods, i.e., for indexing multidimensional information such as geographical coordinates or polygons. R-trees group nearby objects and represent them with a minimum bounding rectangle (thus, the R in the name) in the next higher level of the tree.

more, relational models prohibit pointers, while OO languages use extensively pointers to reference objects.

RDBMS transactions are much larger than operations by classes in OO languages. Moreover, RDBMS transactions are dynamically bounded sets of arbitrary data manipulations, whereas the granularity of transactions in an OO language is typically on the level of individual assignments to primitive-typed fields. All these facts contribute to the so-called *object-relational impedance mismatch* [427].

The values in a relational tuple cannot contain their own structure, like a nested record. In-memory data structures support much richer representations than relations, so this mismatch does not affect in-memory data representation. In-memory data structures must be translated to a relational representation before being stored on the disk. This mismatch is one of the main reasons for the transition to NoSQL cloud databases.

A critical requirement for cloud databases is *scalability,* the ability to store and process efficiently huge amounts of data distributed over a large set of storage devices. We distinguish two types of scalability, horizontal and vertical; in the former additional resources are provided by a larger number of storage servers, while in the latter additional resources are provided by more powerful servers.

RDBMS are not designed to run effectively on a large number of storage servers without a single point of failure. A case in point, the Microsoft SQL server runs on a cluster-aware file system storing state information on a highly available disk system, a single point of failure.

In Section 3.17, we have seen that the CAP theorem states that a distributed system cannot guarantee consistency, availability, and partition tolerance at the same time. RDBMS systems guarantee consistency, and sharding[4] makes them tolerant to partitioning, therefore they cannot guarantee availability. That's why a standard RDBMS cannot scale very well: It is not able to guarantee availability.

Cloud computing brought along the demand for storing unstructured or semistructured data, scalability, and effectiveness, thus the need for a new database model, the NoSQL model discussed next.

7.9 NoSQL databases

Convenience prevailed when naming this new database model. An accurate description of this model, possibly *NO-Relational-database,* lacked appeal, so the name NoSQL was rapidly adopted by the community. But this name is misleading.

Michael Stonebreaker, the 2014 Turing Award winner for major contributions to the theory and practice of database systems, writes, "blinding performance depends on removing overhead. Such overhead has nothing to do with SQL, but instead revolves around traditional implementations of ACID transactions, multi-threading, and disk management" [458].

NoSQL refers to a set of databases developed in late 1990s that do not rely on the relational model. NoSQL databases differ from each other, run well on large clusters, and are open source. NoSQL schema is defined implicitly by the way application code utilizes the database. NoSQL does not reflect an advance in storing technology, rather a response to practical needs to efficiently access very large datasets stored on large computer clusters [75]. In the new model RDBMS names changed: a *partition* is a *shard*, a *table* is a *document root element*, a *row* is an *aggregate/record,*

[4] A shard is a horizontal partitioning of a database, a row in a table structured data.

and a *column* is an *attribute/field/property*. There is no stand-alone query language for NoSQL databases.

NoSQL distribution models. Scalability is an essential attribute of a cloud database, therefore the ability to store very large volumes of data on a large number of storage servers is a major requirement for NoSQL databases. The mean time to failure in a large-scale storage systems built with off-the-shelf components is rather short, therefore the distribution model should support effective replication required for database availability and sharding. Replication stores data on multiple servers, while sharding partitions the data.

The distribution model is based on an aggregate data model that eliminates the relations and transactions of the relational model in favor of more complex data structures allowing lists and other record structures to be nested. The aggregate structures are naturally replicated and sharded; storing relevant data on the same storage server minimizes the number of servers accessed to respond to a query.

The NoSQSL distribution models are: (i) sharding; (ii) master–slave replication; and (iii) peer-to-peer (P2P) replication. *Sharding* is based on horizontal scalability; the data is distributed to the servers aiming to balance the workload. The aggregates are stored together if accessed together; the aggregate distribution to storage servers may change over time in response to access pattern changes. Sharding optimizes the writes.

Master–slave replication designates one server as master and all other storage servers as slaves. The master is responsible for processing updates and is the authoritative data source. Servers are synchronized with the master. A drawback of this distribution model is inconsistency as updates take time to propagate from the master to the slaves. This distribution model is optimal for read-intensive database applications and horizontal scaling because new slaves can be added easily.

P2P replication. P2P systems are the most democratic; servers have the same functionality and accept write operations; and loosing one or a few servers may affect performance but not system functionality. The price to pay is potential lack of consistency; a *write-write conflict* is unavoidable when a record is updated on two different servers at the same time. These models can be combined, for example, a P2P and sharding combination is appealing to column-family databases.

NoSQL database types. Four flavors can be distinguished:

1. *Key-value.* This type of NoSQL databases model data as an index key and a value, similar with hash tables. Access to database is via primary key. Modeling relationships among different sets of data, correlating data with different sets of keys, searching based on the value of the key-value pair, and operations on sets are not recommended for key-value databases.
2. *Document databases.* Similar to key-value, but the value associated with a key contains structured or semistructured data. These databases enjoy a number of useful features such as: (i) ability to query data in the document without having to retrieve the whole document by key; (ii) support nested documents within the value field. Document databases should not be used for complex transactions spanning different operations because atomicity is only guaranteed within a single document, not across documents.
3. *Column-family databases.* Such databases store large sparse tables with a very large number of rows and only a few columns. They store data in column families as rows. Several columns are associated with a row key for data often accessed together. Super columns consist of a map of columns; there are also super column families.

4. *Graph databases.* In graph databases, the vertices represent entities, and the directional edges represent relationships among the entities. The graphs are meant to be stored with a single organizational structure and then interpreted in different ways based on the relationships. Scaling graph databases is different than for aggregate-oriented models, and sharding is difficult because the nodes are relationship-oriented rather than aggregate-oriented. Horizontal scaling is ineffective for graph databases, only vertical scaling is suitable.

Rating of several NoSQL databases. Recall from Section 3.17 that PACELC theorem states that, in case of (P), network partitioning, one has to choose between (A), availability, and (C), consistency. Else, (E), even when the system is running normally in absence of partitions, one has to choose between latency, and availability. According to [2] the choices made by several NoSQL databases are:

1. *PA/EL*—if a partition occurs, give up consistency for availability, and under normal operation give up consistency for lower latency. Examples: default versions of DynamoDB, Cassandra, Riak, and Cosmos DB.
2. *PC/EC*—to achieve consistency pay availability and latency costs. Examples: fully ACID systems such as VoltDB/H-Store, Megastore, and MySQL Cluster.
3. PA/EC—guarantees reads and writes to be consistent. Example: MongoDB.
4. *Tradeoffs between C/A during P, and L/C during E.* For example, Cosmos DB supports five tunable consistency levels and never violates the specified consistency level.

Summary. Cloud stores such as *document stores* and NoSQL databases are designed to scale well, do not exhibit a single point of failure, have built-in support for consensus-based decisions, and support partitioning and replication as basic primitives. Systems such as *SimpleDB* discussed in Section 2.2, *CouchDB* (see http://couchdb.apache.org/), or *Oracle NoSQL database* [379] are very popular, though they provide less functionality than traditional databases. The *key-value* data model is very popular. Several such systems including Voldemort, Redis, Scalaris, and Tokyo cabinet are discussed in [89].

The *soft-state* approach in the design of NoSQL allows data to be inconsistent and transfers the task of implementing only the subset of the ACID properties required by a specific application to the application developer. The NoSQL systems ensure that data will be *eventually consistent* at some future point in time, instead of enforcing consistency at the time when a transaction is "committed."

It was suggested to associate NoSQL databases with the BASE acronym reflecting their relevant properties, Basically Available, Soft state, and Eventually consistent, whereas traditional databases are characterized by ACID properties; see Section 7.3. Data partitioning among multiple storage servers and data replication are also tenets of the NoSQL philosophy; they increase availability, reduce the response time, and enhance scalability.

7.10 Data storage for online transaction processing systems

Many cloud services are based on Online Transaction Processing (OLTP) and operate under tight latency constraints. Moreover, OLTP applications have to deal with extremely high data volumes and are expected to provide reliable services for very large communities of users. It did not take very long for organizations heavily involved in cloud computing, such as Google and Amazon, eCommerce companies, such as *eBay*, and social media networks such as Facebook, Twitter, or LinkedIn, to discover that traditional relational databases are not able to handle the massive amount of data and the real-time demands of online applications critical for their business model.

The search for alternate models to store the data on a cloud is motivated by the need to decrease the latency by caching frequently used data in memory on dedicated servers, rather than fetching it repeatedly. Distributing data to a large number of servers allows multiple transactions to occur at the same time and decreases the response time. The relational schema is of little use for OLTP applications, and conversion to key-value databases seems a much better approach. Of course, such systems do not store meaningful metadata information, unless they use extensions that cannot be exported easily.

Reducing the response time is a major concern of OLTP system designers. The term *memcaching* refers to a general purpose distributed memory system that caches objects in main memory. The system is based on a very large hash table distributed across many servers. A *memcached* system is based on a client–server architecture and runs under several operating systems, including Linux, Unix, Mac OS X, and Windows. The servers maintain a key–value associative array. The API allows clients to add entries to the array and to query it; a key can be up to 250 bytes long, and a value can be not larger than 1 MB. A *memcached* system uses the LRU cache replacement strategy. Scalability is the other major concern for cloud OLTP applications and implicitly for datastores. There is a distinction between *vertical scaling*, where the data and the workload are distributed to systems that share resources, such as cores/processors, disks, and possibly RAM, and *horizontal scaling*, where the systems do not share either the primary or secondary storage [89].

The overhead of OLTP systems is due to four sources with equal contributions: logging, locking, latching, and buffer management. Logging is expensive because traditional databases require transaction durability, thus every write to the database can only be completed after the log has been updated. To guarantee atomicity, transactions lock every record, and this requires access to a lock table.

Many operations require multithreading, and the access to shared data structures, such as lock tables, demands short-term latches[5] for coordination. The breakdown of the instruction count for these operations in existing DBMS is: 34.6% buffer management, 14.2% latching, 16.3% locking, 11.9% logging, and 16.2% for hand-coded optimization [228].

Today, OLTP databases could exploit the vast amounts of resources of modern computing and communication systems to store the data in main memory rather than rely on disk-resident B-trees and heap files, locking-based concurrency control, and the support for multithreading optimized for the computer technology of past decades [228]. Logless, single threaded, and transactionless databases could replace the traditional ones for some cloud applications.

Data replication is critical not only for system reliability and availability but also for its performance. In an attempt to avoid catastrophic failures due to power blackouts, natural disasters, or other causes (see also Section 1.4), many companies have established multiple data centers located in various geographic regions. Thus, data replication must be done over a wide area network. This could be quite challenging especially for log data, metadata, and system configuration information due to larger communication delays and an increased probability of communication failures. Several strategies are possible, some based on master–slave configurations, other based on homogeneous replica groups.

Master–slave replication can be asynchronous or synchronous. In the first case, the master replicates write-ahead log entries to at least one slave, and each slave acknowledges appending the log record as soon as the operation is done, while in the second case, the master must wait for the acknowledgments from all slaves before proceeding. Homogeneous replica groups enjoy shorter latency and higher

[5] A latch is a counter that triggers an event when it reaches zero; for example, a master thread initiates a counter with the number of worker threads and waits to be notified when all of them have finished.

availability than master–slave configurations; any member of the group can initiate mutations which propagate asynchronously.

In summary, the "one-size-fits-all" approach in the traditional storage system design is replaced by a flexible one, tailored to the specific requirements of the applications. Sometimes, the data management of a cloud computing environment integrates multiple databases. For example, Oracle integrates its NoSQL database with the HDFS discussed in Section 11.7, with the Oracle Database, and with the Oracle Exadata. Another approach, discussed in Section 7.12, partitions the data and guarantees full ACID semantics within a partition, while supporting eventual consistency among partitions.

7.11 BigTable

BigTable is a distributed storage system developed by Google to store massive amounts of data and to scale up to thousands of servers [94]. The system uses GFS discussed in Section 7.6 to store user data and system information. BigTable uses the Chubby distributed lock service to guarantee atomic *read* and *write* operations. Directories and files in Chubby's namespace are used as locks. Client applications written in C++ can add/delete values, search for a subset of data, and lookup for data in a row.

BigTable is based on a simple and flexible data model and allows an application developer to exercise control on the data format and layout. It also reveals data locality information to the application clients. Column keys identify units of access control called *column families,* including data of the same type. A column key consists of a string defining the family name, a set of printable characters, and an arbitrary string as qualifier. A row key is an arbitrary string of up to 64 KB, and a row range is partitioned into *tablets* serving as units for load balancing. Any *read* or *write* row operation is atomic, even when it affects more than one column. Time stamps used to index different versions of the data in a cell are 64-bit integers. The interpretation of time stamps can be defined by the application, while the default is the time of an event in microseconds.

The organization of a BigTable—see Fig. 7.11—shows a sparse, distributed, multidimensional map for an email application. The system consists of three major components: a library linked to application clients to access the system, a master server, and a large number of tablet servers. The master server controls the entire system, it assigns tablets to tablet servers and balances the load among them, manages garbage collection, and handles table and column family creation and deletion.

Internally, the space management is ensured by a three-level hierarchy: the *root tablet* whose location is stored in a Chubby file, points to entries in the second element, the *metadata* tablet which, in turn, points to *user* tablets, collections of locations of user's tablets. An application client searches through this hierarchy to identify the location of its tablets and then caches addresses for further use.

BigTable performance reported in [94] is summarized in Table 7.4 which shows the number of random and sequential *read* and *write* and scan operations for 1 000 bytes, when the number of servers increases from 1 to 50, then to 250, and finally to 500. Locking prevents the system from achieving a linear speedup, but the BigTable performance is remarkable due to a fair number of optimizations. For example, the number of scans on 500 tablet servers is $7\,843 \times 500$ instead of $15\,385 \times 500$. It is reported that only 12 clusters use more than 500 tablet servers, while some 259 clusters use between 1 and 19 tablet servers.

BigTable is used by a variety of applications including Google Earth, Google Analytics, Google Finance, and web crawlers. For example, Google Earth uses two tables, one for preprocessing and

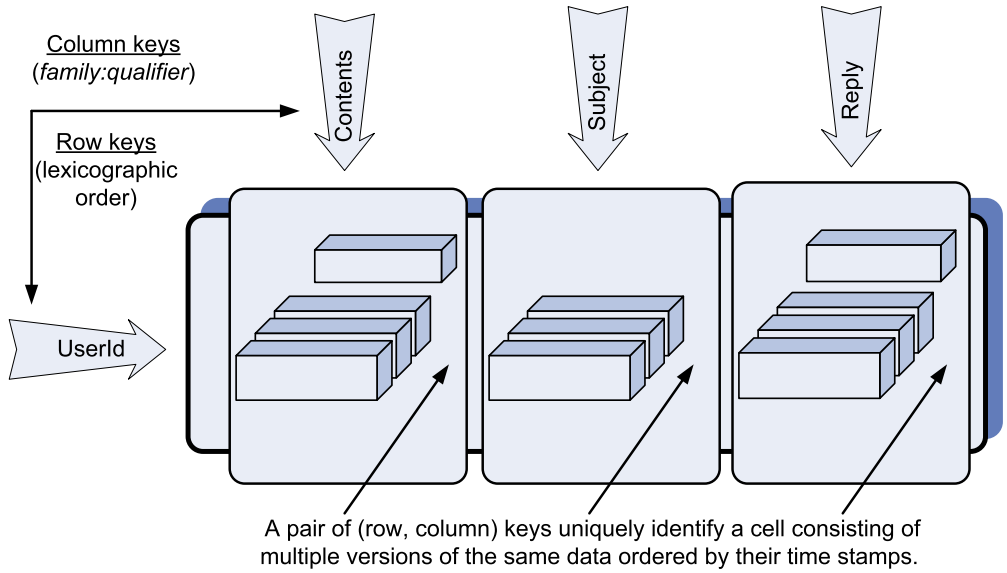

FIGURE 7.11

BigTable example; the organization of an email application as a sparse, distributed, multidimensional map. The slice of the BigTable shown consists of a row with the *UserId* key and three *family* columns; the *Contents* key identifies the cell holding the contents of emails received, the one with the *Subject* key identifies the subject of emails, and the one with the *Reply* key identifies the cell holding the replies; the version of records in each cell are ordered according to their time stamps. The row keys of this BigTable are ordered lexicographically; a column key is obtained by concatenating the *family* and the *qualifier* fields. Each value is an uninterpreted array of bytes.

Table 7.4 BigTable performance; the number of operations per tablet server.

Number of tablet servers	Random read	Sequential read	Random write	Sequential write	Scan
1	1 212	4 425	8 850	8 547	15 385
50	593	2 463	3 745	3 623	10 526
250	479	2 625	3 425	2 451	9 524
500	241	2 469	2 000	1 905	7 843

one for serving client data. The preprocessing table stores raw images; the table is stored on disk as it contains some 70 TB of data. Each row of data consists of a single imagery; adjacent geographic segments are stored in rows in close proximity to one another. The column family is very sparse; it contains a column for every raw image. The preprocessing stage relies heavily on MapReduce to clean and consolidate the data for the serving phase. The serving table is stored on GFS and is "only" 500 GB, and it is distributed across several hundred tablet servers that maintain in-memory column families; this organization enables the serving phase of Google Earth to provide a fast response time to tens of thousands of queries per second.

Google Analytics provides aggregate statistics such as the number of visitors of a web page per day. To use this service, web servers embed a JavaScript code in their web pages to record information every time a page is visited. The data is collected in a *raw click* BigTable of some 200 TB with a row for each end-user session. A *summary* table of some 20 TB contains predefined summaries for a website.

7.12 Megastore

Megastore is a scalable storage for online services. The system, distributed over several data centers, has a very large capacity and is highly available. Megastore is widely used internally at Google; in 2011, it had a capacity of 1PB, handled some 23 billion transactions daily, 3 billion *write*, and 20 billion *read* transactions [43].

The basic design philosophy of the system is to partition the data into *entity groups* and replicate each partition independently in data centers located in different geographic areas. The system supports full ACID semantics within each partition and provides limited consistency guarantees across partitions; see Fig. 7.12. Megastore supports only those traditional database features that allow the system to scale well and do not affect drastically the response time.

Another distinctive feature of the system is the use of the Paxos consensus algorithm discussed in Section 10.13 for replicating primary user data, metadata, and system configuration information across data centers and for locking. The version of the Paxos algorithm used by Megastore does not require a single master, but instead any node can initiate *read* and *write* operations to a write-ahead log replicated to a group of symmetric peers.

Entity groups are application-specific and store together logically related data; for example, an email account could be an entity group for an email application. Data should be carefully partitioned to avoid excessive communication between entity groups. Sometimes, it is desirable to form multiple entity groups as is the case of blogs [43].

This middle ground between traditional and NoSQL databases taken by the Megastore designers is also reflected by the data model. The data model is declared in a *schema* consisting of a set of *tables* that are composed of *entries*, each entry being a collection of named and typed *properties*. The unique primary key of an entity in a table is created as a composition of entry properties. A Megastore table can be a *root* or a *child* table; each *child entity* must reference a special entity, called *root entity* in its root table. An entity group consists of a primary entity and all the entities that reference it.

The system makes extensive use of BigTable. Entities from different Megastore tables can be mapped to the same BigTable row without collisions. This is possible because the BigTable column name is a concatenation of the Megastore table name and the name of a property. A BigTable row for the root entity stores the transaction and all metadata for the entity group. Multiple versions of the data with different time stamps can be stored in a cell as we have seen in Section 7.11.

Megastore takes advantage of this feature to implement *multiversion concurrency control*. When a mutation of a transaction occurs, this mutation is recorded along with its time stamp, rather than marking the old data as obsolete and adding the new version. This strategy has a couple of several advantages: *Read* and *write* operations can proceed concurrently, and a *read* always returns the last fully updated version.

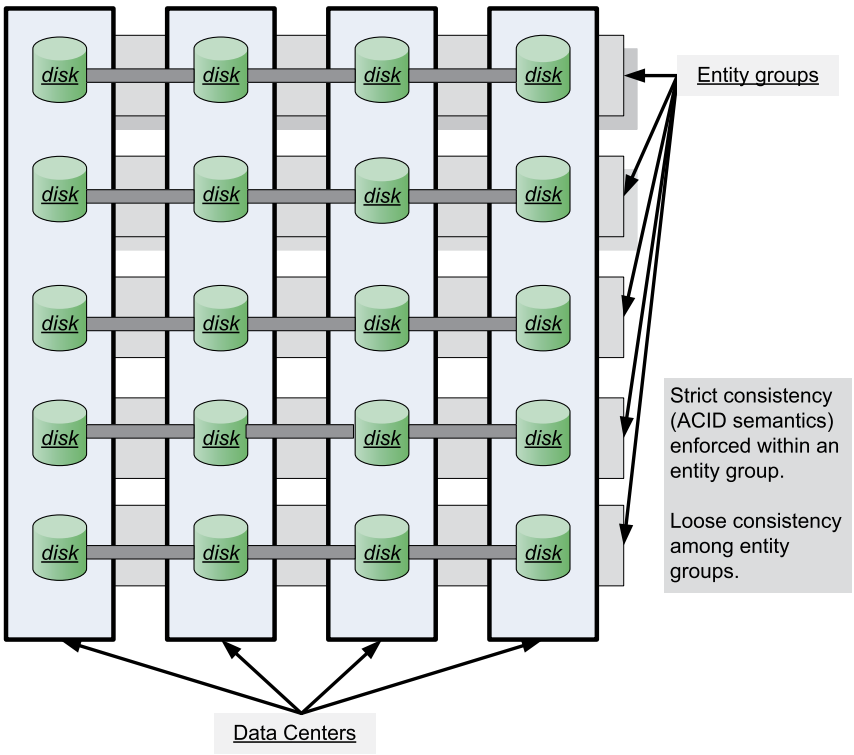

FIGURE 7.12

Megastore organization. Data is partitioned into *entity groups*; full ACID semantics within each partition and limited consistency guarantees across partitions are supported. A partition is replicated across data centers in different geographic areas.

A *write* transaction involves several steps: (1) get the time stamp and the log position of the last committed transaction; (2) gather the *write* operations in a log entry; (3) use the consensus algorithm to append the log entry and then commit; (4) update the BigTable entries; and (5) cleanup.

7.13 Storage reliability at scale

Building reliable systems with unreliable components is a major challenge in system design identified and studied early on by John von Neumann [497]. This challenge is greatly amplified, on one hand, by the scale of the cloud computing infrastructure and by the use of off-the-shelf components that reduces the infrastructure cost and, on the other hand, by the latency constrains of many cloud applications.

Even though the mean time to failure of individual components can be on the order of month or years, it is unavoidable to witness a small, but significant, number of server and network components

that are failing at any given time. Data losses cannot be tolerated, thus the failure of storage devices is a major concern. It is left to the software to mask the failures of storage devices and avoid data loss.

Dynamo and DynamoDB. Amazon developed two database systems to support reliability at scale. Dynamo, a highly available key-value storage system, has been solely used by AWS core services for in-house applications since 2007 [133]. In 2012 DynamoDB, a NoSQL database service for latency-sensitive applications that need consistent access at any scale, was opened to AWS user community. Dynamo and DynamoDB use a similar data model, but Dynamo had a multimaster design requiring the client to resolve version conflicts, whereas DynamoDB uses synchronous replication across multiple data centers for high durability and availability.

DynamoDB is a fully managed database service designed to provide an "always-on" experience. It supports both document and key-value store models and has been used for mobile, web, gaming, IoT, advertising, real-time analytics, and other applications. DynamoDB stores data on SSDs to support latency-sensitive applications; typical requests take milliseconds to complete. DynamoDB allows developers to specify the throughput capacity required for specific tables within their database using the *provisioned throughput* feature to deliver predictable performance at any scale. The service is integrated with other AWS services, e.g., it offers integration with Hadoop via Elastic MapReduce.

Design objectives. Dynamo's primary concern is high availability where updates are not rejected even in the wake of network partitions or server failures. Dynamo has to deliver predictive performance, in addition to reliability and scalability. Services supported by Dynamo have stringent latency requirements, and this precludes supporting ACID properties. Indeed, data stores providing ACID guarantees tend to exhibit poor availability.

The most significant design considerations regard data replication and measures to increase availability in the wake of failures. Strong consistency and high data availability cannot be achieved simultaneously. Availability can be increased by optimistic replication, allowing changes to propagate to replicas in the background, while disconnected work is tolerated.

In traditional data stores, writes may be rejected if the data store cannot reach all, or a majority of, the replicas at a given time. This approach is not tolerated by many AWS applications. As a consequence, rather than implementing conflict resolution during writes and keeping the read complexity simple, Dynamo increases the complexity of conflict resolution of the read operations.

Dynamo supports simple read and write operations to data items uniquely identified by a key. The *get(key)* operation locates object replicas associated with the key in the storage system and returns a single object or a list of objects with conflicting versions along with a context. The *put(key, context, object)* operation determines where the replicas of the object should be placed based on the associated key and writes replicas to the secondary storage.

The *context* encodes system metadata about the object that is opaque to the caller and includes information such as the version of the object. Context information is stored along with the object so that the system can verify the validity of the context object supplied in the *put* request. The main techniques used to achieve Dynamo's design objectives are:

1. *Incremental scalability* ensured by consistent hashing.
2. *High write availability* based on the use of vector clocks with reconciliation.
3. *Handling temporary failures* using sloppy quorum and hinted handoff. This provides high availability and durability guarantees when some of the replicas are not available.

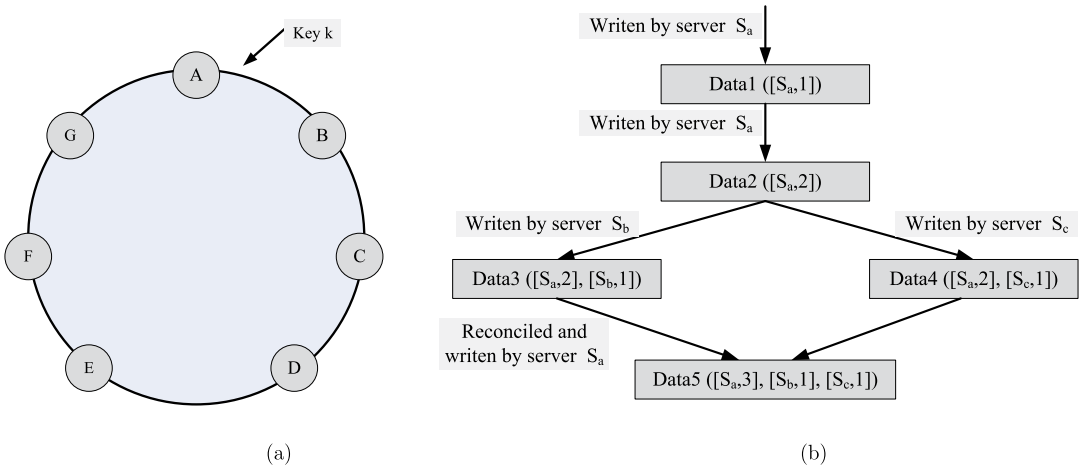

FIGURE 7.13

(a) Dynamo servers are organized as a ring. Ring nodes B, C, and D store keys in range (A, B), including the key, k. (b) Evolution of an object in time using vector clocks.

4. *Permanent failure recovery* based on anti-entropy and Merkle trees.[6] This technique synchronizes divergent replicas in the background.
5. *Gossip-based membership protocol and failure detection*. This technique preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Scaling, load balancing, and replication. Data partitioning scheme based on *consistent hashing* is designed to support incremental system scaling. The output of a hash function is treated as a ring, and each node in the system is assigned a random value representing its position on the ring.

Consistent hashing reduces the number of keys to be remapped when a hash table is resized. On average, only K/n keys need to be remapped, with K being the number of keys and n being the number of slots. In most traditional hash tables, a change in the number of slots causes nearly all keys to be remapped because the mapping between the keys and the slots is defined by a modular operation.

A data item identified by a key is assigned to a storage server by hashing the data item key to yield its position on the ring and then walking the ring clockwise to find the first node with a position larger than the position of the item. Each storage server is responsible for the ring region between itself and its predecessor in the ring; see Fig. 7.13(a). In this example, node B replicates key k at nodes C and D, in addition to storing it locally. Node D will store keys in the ranges $(A, B]$, $(B, C]$, and $(C, D]$.

Instead of mapping a storage server to a single point on the ring, the system uses the concept of "virtual nodes" and assigns it to multiple points on the ring. A physical server is mapped to multiple nodes of the ring. This form of virtualization supports:

[6] A Merkle or hash tree is a tree in which every leaf node is labelled with the cryptographic hash of a data block, and every non-leaf node is labelled with the cryptographic hash of the labels of its child nodes. Hash trees allow efficient and secure verification of the contents of large data structures.

1. Load balancing. When a storage server is unavailable, its load is dispersed among available servers. When the server comes back again, it is added to the system and accepts a load roughly equivalent to the load of other servers.
2. System heterogeneity. The number of virtual nodes a physical server is mapped to depends on its capacity.

A data item is replicated at N servers. Each key, k, is assigned to a coordinator charged with the replication of the data items in its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the N-1 clockwise successor nodes in the ring. Each node is responsible for the ring region between it and its N-th predecessor. The *preference list* is the list of all nodes responsible for storing a particular key.

Eventual consistency. This strategy allows *updates to be propagated to all replicas asynchronously*. A versioning system allows multiple versions of a data object to be present in the data store at the same time. The result of each modification is a new and immutable data version. New versions often subsume the older ones, and the system can use syntactic reconciliation to determine the authoritative version.

The system uses *vector clocks*, lists of (node, counter) pairs, to capture causality of each version of a data object. When a client updates an object, it must specify the version it is updating by passing the context it obtained from an earlier read operation, which contains the vector clock information. System failures combined with concurrent updates lead to conflicting versions of an object and version branching. Given two versions of the same object, the first is an ancestor of the second and can be forgotten if the counters on the first object's clock are less than or equal to all of the nodes in the second clock; otherwise, the two changes are in conflict and require reconciliation.

Fig. 7.13(b) illustrates the versioning mechanism for the following sequence of events. Data is written by server S_a, and object *Data1* with associated clock $[S_a, 1]$ is created. The same server S_a writes again, and object *Data2* with associated clock $[S_a, 2]$ is created. *Data2* is a descendent of *Data1* and overwrites it. There may be replicas of *Data1* at servers that have not yet seen *Data2*. Then, the same client updates the object, and server S_b handles the request; a new object data *Data3* and its associated clock $[(S_a, 2), (S_b, 1)]$ are created.

A different client reads *Data2* and tries to update it; this time, server S_c handles her request. A new object *Data4*, a descendent of *Data2*, with version clock $[(S_a, 2), (S_c, 1)]$ is created. Upon receiving *Data4* and its clock, a server aware of *Data1* or *Data2* could determine that both are overwritten by the new data and can be collected garbage.

A node aware of *Data3* and *Data4* will see that there is no causal relation between them because there are changes not reflected in each other. Both versions of the data must be kept and presented to a client for semantic reconciliation. If a client reads both *Data3* and *Data4,* its context will be $[(S_a, 2), (S_b, 1), (S_c, 1)]$, the summary of the virtual clocks of both data objects. If the client performs a reconciliation and the write request is handled by server S_a, then the vector clock of the new data, *Data5*, will be $[(S_a, 3), (S_b, 1), (S_c, 1)]$. The size of the vector clock grows, but in practice this growth is limited.

Sloppy quorum for handling failures. During server failures and network partitions a strict quorum membership is enforced by traditional systems. This conflicts with the durability requirement, and in Dynamo all read and write operations are performed on the first N *healthy nodes* from the preference list. In this *sloppy quorum*, the healthy nodes may not always be the first N nodes encountered while walking the consistent hashing ring.

For example, to maintain the desired availability and durability guarantees, when node A in Fig. 7.13(a) is unreachable during a write operation, a replica that would normally have been sent to A will be sent to D. The metadata of this replica will include a hint indicating the intended recipient of the replica.

Replica synchronization in case of permanent failures. Replica inconsistencies can be detected faster and with a minimum of data transfer using Merkle trees.[7] This allows each branch of the tree to be checked independently without the need to download the entire tree. For example, if the hash values of the root of two trees are equal, then the values of the leaf nodes in the tree are equal, and the nodes require no synchronization.

Anti-entropy is a process of comparing the data of all replicas and updating each replica to the newest version; Merkle trees are used for anti-entropy. The key range is the set of keys covered by a virtual node. Each node maintains a separate Merkle tree for each key range, and two nodes exchange the root of the Merkle tree corresponding to the key ranges that they host in common to compare whether the keys within a key range are up-to-date.

Gossip-based node addition to or removal from the ring. The node receiving the request writes the change and its time of issue to persistent store. A gossip-based protocol propagates membership changes and maintains an eventually consistent view of the membership. Each node contacts a peer chosen at random every second, and the two reconcile their persisted membership change histories.

For example, consider the case when a new node Q is added between nodes A and B to the ring in Fig. 7.13(a). Now node Q will be storing keys in the ranges $(F, G]$, $(G, A]$ and $(A, Q]$ freeing nodes B, C, and D from storing the keys in these ranges. Upon confirmation from Q, nodes B, C, and D will transfer the appropriate set of keys to it. When a node is removed from the system, the reallocation of keys proceeds as a reverse process.

7.14 Disk locality versus data locality in computer clouds

Locality is critical for the performance of computing systems. Recall that a sequence of references is said to have spatial locality if the items referenced within a short time interval are close in space, e.g., they are at nearby memory addresses or nearby sectors on a disk. A sequence exhibits temporal locality if accesses to the same item are clustered in time.

Locality has major implications for memory hierarchies. The performance of a processor is highly dependent on the ability to access code and data in the cache rather than memory. Virtual memory can only be effective if the code and the data exhibit spatial and temporal locality so that page faults occur infrequently. Optimizing locality in cloud computing is a challenging problem, and a fair amount of effort has been devoted to algorithms and systems designed to increase the fraction of the tasks of a job enjoying locality. Locality improves the performance of cloud applications for two main reasons: (i) disk bandwidth is larger than the network bandwidth, and the off-rack communication bandwidth is oversubscribed and affects the off-rack disk access; and (ii) the better performance of I/O-intensive

[7] A Merkle tree is a hash tree where leaves are hashes of the values of individual keys. Parent nodes higher in the tree are hashes of their respective children.

applications, when data is stored locally, is due to the lower latency and the higher bandwidth of a local disk versus the latency and the bandwidth of a remote disk.

An important question for cloud resource management is whether *disk locality* is important. Intuitively, we expect the answer to the question whether tasks should be dispatched to the cluster nodes, where their input data resides would be a resounding "Yes." A slightly different view on the subject of disk locality is expressed in a paper with the blunt title "Disk-Locality in Datacenter Computing Considered Irrelevant" [23].

Maybe we tried to solve the wrong problem, and instead of focusing on disk locality, we should focus on data locality. In other words, we should look at the local memory as a "data cache" and make sure that data is stored in the local memory rather than the local disk of the processor where the task is scheduled to run. Data transfer through the network may still be necessary, but why store it on the disk and then load it in memory? Several arguments support this thesis:

a. Networking technology improves at a faster pace than hard disk technology. Switches with aggregate link speeds of 40 Gbps and 100 Gbps are available today. Rates of 10 and 25 Gbps of server network interfaces will be available soon.
b. The bandwidth available to applications will increase as data centers adopt bisection topologies for their interconnects [52]. Recall that the bisection bandwidth is the sum of the bandwidths of the minimal number of links that are cut when splitting the system into two parts.
c. The latency of a read access to a local disk is only slightly lower than the latency of a read to a disk in the same rack; local disk access is about 8% faster [52]. Access to a disk in a different rack will not increase drastically due to faster networks.
d. Though the cost of solid-state disk technology is dropping at a impressive rate, i.e., 50% per year, solid-state disks are unlikely to replace hard disk drives any time soon due to the sheer volume of data stored on computer clouds. To be competitive with HDDs, the cost-per-byte of SSD should be reduced by up to three orders of magnitude.
e. Accessing data in local memory is two orders of magnitude faster than reading from a local disk. An increasing number of applications use the processor memory distributed across the nodes of large clusters to cache the data, rather than access the HDDs.
f. There is at least a two orders of magnitude discrepancy between the capacity of the disks and the memory of today's clusters, thus it seems reasonable to use processor memory as a cache for the much larger volume of data stored on the disks.

It makes sense to use processor memory as a cache for large data sets used as the input of an application if and only if: (1) the size of the application's input is much larger than the memory size; and (2) the applications exhibit locality and have a modestly sized working set, in other words, if they access frequently only a relatively small fraction of data blocks.

Hadoop tasks running at a data center with some 3 000 machines interconnected by three-tiered networks are investigated in [23]. Most Facebook jobs are data-intensive and spend most of their execution time reading input data. The analysis of job traces led to the conclusion that there is an order of magnitude discrepancy between the input size and the memory size and that some 75% of the data blocks are accessed only once.

The analysis also shows that workloads have a heavy tail distribution of block access. Moreover, 96% of the data inputs can fit into a fraction of the total cluster memory for a majority of jobs. Some

64% of all jobs investigated perform well under a least frequently used (LFU) replacement policy applied to all their tasks.

The *task progress report*, $\mathbb{T}$, a metric for the effect of locality on the duration of a task, is defined as the ratio

$$\mathbb{T} = \frac{data\_read + data\_written}{task\_duration}. \tag{7.1}$$

The results of measurements comparing node-local versus rack-local tasks show that 85% of the jobs have $0.9 \leq \mathbb{T} \leq 1.1$ and only 4% of jobs have $\mathbb{T} \leq 0.7$.

Data compression reduces the pressure on the data center disk space. Tasks read compressed data and uncompress it before processing. In spite of a network oversubscribed by a factor of ten at Facebook, data compression leads to very good results; running a task off-rack is only 1.2 times to 1.4 times slower compared to on-rack execution. The log analysis of Hadoop jobs suggests that prefetching data blocks could be beneficial because a large fraction of data blocks are accessed only once.

7.15 Database provenance

Data *provenance* or *lineage* describes the origins and the history of data and adds value to data by explaining how it was obtained. The lineage of a tuple $\mathcal{T}$ in the result of a query is the set of items contributing to produce $\mathcal{T}$. The lineage is important for the extract-transform-load process and for incrementally adding and updating a database.

Before the Internet era the information in databases was trusted, and it was assumed that the organization maintaining the database was trustworthy and that every effort to ensure data veracity was made. This assumption is no longer true; there is no centralized control over data integrity because the Internet allows data to be indiscriminately created, copied, moved around, and combined. Establishing data provenance is necessary for all databases and is critical for cloud databases as the data owners relinquish control of their data to CSPs.

Data provenance has been practiced by the scientific and engineering community for some time, long before the disruptive effects of data democratization brought about by the Internet. Data collected by scientific experiments contains information about the experimental setup and the settings of measuring instruments for each batch of data. Ensuring that experiments can be replicated has always been an essential aspect of scientific integrity. The same requirements apply to engineering when test data, e.g., data collected during the testing of a new aircraft, include lineage information.

Data provenance could explain *why* an output record was produced, describing in detail *how* the record was produced, and/or explaining *where* output data cames from. The why-, how-, and where-provenance are analyzed in [105] and discussed briefly in this section.

The *witness* of a database record is the subset of database records ensuring that the record is the output of a query. The *why-provenance* includes information about the witnesses to a query. The number of witnesses can be exponentially large. To limit this number, the concept of *witness bases* of tuple $\mathcal{T}$ in query $\mathcal{Q}$ on database $\mathcal{D}$ is defined as the particular set of witnesses that can be calculated efficiently from $\mathcal{Q}$ and $\mathcal{D}$.

The *why-provenance* of an output tuple $\mathcal{T}$ is the witness basis of $\mathcal{I}$ according to $\mathcal{Q}$. The witness basis depends on the structure of the query, and it is sensitive to the query formulation. We now take a short

Table 7.5 Two queries Q and Q' of instance I are equivalent under R for different pairs of A and B elements of the three tuples t, t', t''.

	Output of $Q'(I)$		Output of $Q(I)$	
R	A	B	A	B
t:	1	2	1	2
t':	1	3	1	3
t'':	4	2	4	2

detour for introducing *Datalog conjunctive query* notations before presenting an example showing that the why-provenance is sensitive to query rewriting.

Datalog is a declarative logic programming language. Query evaluation in Datalog is based on first order logic, thus it is sound and complete. A Datalog program includes *facts* and *rules*. A rule consists of two elements, the head and the body, separated by the ":-" symbol. A rule should be understood as: *"head" if it is known that "body"*. For example,

- the facts in the left box below mean: (1) Y is in relation R with X; (2) Z is in relation R with Y.
- the rules in the right box mean: (1) Y is in relation P with X if it is known that Y is in relation R with X; and (2) Y is in relation P with X if it is known that Z is in relation R with X AND rule P is satisfied, i.e., Y is in relation P with Z.

```
R(X,Y).
R(Y,Z).
```

```
P(X,Y) :- R(X,Y).
P(X,Y) :- R(X,Z), P(Z,Y)
```

Consider now an instance I defining relation R and three tuples t, t', t'', two queries Q, Q', and the output of the two queries $Q(I)$, $Q'(I)$. The two queries are

$$\begin{aligned} &Q : Ans(x, y) : -R(x, y) \\ &Q' : Ans(x, y) : -R(x, y), R(x, z). \end{aligned} \tag{7.2}$$

Table 7.5 shows that queries Q and Q' are equivalent since they produce the same result. Table 7.6 shows that the why-provenance is sensitive to query rewriting. There exists a subset of the witness basis invariant under equivalent queries. This subset, called *minimal witness basis*, includes all *minimal witnesses* in the witness basis. A witness is minimal if none of its proper subinstances is also a witness in the witness basis. For example, $\{t\}$ is a minimal witness for the output tuple $(1, 2)$ in Table 7.6, but $\{t, t'\}$ is not a minimal witness since $\{t\}$ is a sub-instance of it and it is a witness to $(1, 2)$. Hence, the minimal witness basis is $\{t\}$ in this case.

The *why-provenance* describes the source tuples that witness the existence of an output tuple in the result of the query, but it does not show how an output tuple is derived according to the query. The how-provenance is more general than the why-provenance and it is also sensitive to query formulation as shown in Table 7.7. The how-provenance of the tuple $(1, 2)$ is t in the output of Q and $t^2 + t \cdot t'$ in the output of Q'.

Table 7.6 The why-provenance of the two equivalent queries Q and Q' are different for the three tuples t, t' and t''.

Instance I			Output of $Q(I)$			Output of $Q'(I)$		
R	A	B	A	B	*why*	A	B	*why*
t:	1	2	1	2	$\{\{t\}\}$	1	2	$\{\{t\},\{t,t''\}\}$
t':	1	3	1	3	$\{\{t'\}\}$	1	3	$\{\{t'\},\{t,t''\}\}$
$(t'')^2$:	4	2	4	2	$\{\{(t'')^2\}\}$	4	2	$\{\{(t'')^2\}\}$

Table 7.7 The how-provenance of the two equivalent queries Q and Q' are different for the three tuples t, t' and t''.

Instance I			Output of $Q(I)$			Output of $Q'(I)$		
R	A	B	A	B	*how*	A	B	*how*
t:	1	2	1	2	t	1	2	$t^2 + t \cdot t'$
t':	1	3	1	3	t'	1	3	$(t')^2 + t \cdot t'$
t'':	4	2	4	2	t''	4	2	$(t'')^2$

The where-provenance describes the relationship between source and output locations, while why-provenance describes the relationship between source and output tuples. The where-provenance is also sensitive to query formulation [105].

7.16 History notes and further readings

A 1989 survey of distributed file systems can be found in [436]. NFS Versions 2, 3, and 4 are defined in RFCs 1094, 1813, and 3010, respectively. NFS Version 3 added a number of features including: support for 64-bit file sizes and offsets, support for asynchronous writes on the server, additional file attributes in many replies, and a *READDIRPLUS* operation. These extensions allowed the new version to handle files larger than 2 GB, to improve performance, and to get file handles and attributes along with file names when scanning a directory. NFS Version 4 borrowed a few features from the Andrew file system. WebNFS is an extension of NFS Version 2 and 3; it enables operations through firewalls and is easier integrated into Web browsers.

AFS was further developed as an open-source system by IBM under the name OpenAFS in 2000. Locus [500] was initially developed at UCLA in the early 1980s, and its development was continued by Locus Computing Corporation. Apollo [299] was developed at Apollo Computer Inc, established in 1980, and acquired in 1989 by HP. RFS (Remote File System) [41] was developed at Bell Labs in the mid-1980s. Documentation of a current version of GPFS and an analysis of caching strategy are discussed [250] and [440].

Several DBMS generations based on different models have been developed over the years. In 1968 IBM released the Information Management System (IMS) for the IBM 360 computers. IMS was based on the so-called *navigational model* supporting manual navigation in a linked data set where the data is organized hierarchically. In the RDBMS model, introduced by Codd, related records are linked together

and can be accessed using a unique *key*. Codd also introduced a *tuple calculus* as a basis for a query model for a RDBMS; this led to the development of the Structured Query Language.

In 1973, the Ingres research project at UC Berkeley developed a relational database management system. Several companies, including Sybase, Informix, NonStop SQL, and Ingres, were established to create SQL RDBMS commercial products based on the ideas generated by the Ingres project. IBM's DB2 and SQL/DS dominated the RDBMS market for mainframes during the last part of the 1980s. Oracle Corporation, founded in 1977, was also involved in the development of RDBMS.

ACID properties of database transactions were defined by Jim Gray in 1981 [206], and the term ACID was introduced in [227]. The object-oriented programming ideas of 1980s led to the development of Object-Oriented Data Base Management Systems (OODBMS) where the information is packaged as objects. Ideas developed by several research projects, including Encore-Ob/Server at Brown University, Exodus at the University of Wisconsin-Madison, Iris at HP, ODE at Bell Labs, and the Orion project at MCC-Austin, helped the development of several OODBMS commercial products [271].

NoSQL database management systems emerged in the 2000s. They do not follow the RDBMS model, do not use SQL as a query language, may not give ACID grantees, and have a distributed, fault-tolerant architecture.

Further readings. A 2011 article in the journal *Science* [236] discusses the volume of information stored, processed, and transferred through the networks. [359] is a comprehensive study of the storage technology up to 2003. Evolution of storage technology is presented in [251]. Network File System Versions 2, 3, and 4 are discussed in [433], [392], and [393], respectively. [358] and [245] provide a wealth of information about the Andrew File System, while [237] and [363] discuss in detail the Sprite File System. Other file systems such as Locus, Apollo, and the Remote File System (RFS) are discussed in [500], [299], and [41], respectively. The recovery in the Calypso file system is analyzed in [143]. The Lustre file system is analyzed in [378].

The General Parallel File System (GPFS) developed at IBM and its precursor, the TigerShark multimedia file system, are presented in [440] and [230]. A good source for information about the Google Files System is [197]. Main memory OLTP recovery is covered in [324]. The development of Chubby is covered by [81]. NoSQL databases are analyzed in several papers including [458], [228], and [89]. BigTable and Megastore developed at Google are discussed in [94] and [43]. An evaluation of distributed data store is reported [78].

Attaching cloud storage to a campus grid is the subject of [150]. A cost analysis of storage in enterprise is discussed in [406], and [455] is an insightful discussion of main-memory OLTP databases. [496] presents VMware storage.

7.17 Exercises and problems

Problem 1. Analyze the reasons for the introduction of storage area networks (SANs) and their properties. *Hint:* read [359].

Problem 2. Block virtualization simplifies the storage management tasks in SANs. Provide solid arguments in support of this statement. *Hint:* read [359].

Problem 3. Analyze the advantages of memory-based checkpointing. *Hint:* read [261].

Problem 4. Discuss the security of distributed file systems including SUN NFS, Apollo Domain, Andrew, IBM AIX, RFS, and Sprite. *Hint:* read [436].

Problem 5. The designers of the Google file system (GFS) have reexamined the traditional choices for a file system. Discuss the four observations regarding these choices that have guided the design of GFS. *Hint:* read [197].

Problem 6. In his seminal paper on the virtues and limitations of the transaction concept [206], Jim Gray analyzes logging and locking. Discuss the main conclusions of his analysis.

Problem 7. Michael Stonebreaker argues that "blinding performance depends on removing overhead..." Discuss his arguments regarding the NoSQL concept. *Hint:* read [458].

Problem 8. Discuss the Megastore data model. *Hint:* read [43].

Problem 9. Discuss the use of locking in the BigTable. *Hint:* read [94] and [81].