CHAPTER

Big Data, data streaming, and the mobile cloud

12

Advances in processor, storage, software, and networking technologies enable us to store and process massive amounts of data for the benefit of humanity, for profit, for entertainment, for nefarious schemes, or simply because we can. One can talk about "democratization of data" as scientists and decision makers, journalists and health care providers, artists and engineers, neophytes and domain experts attempt to extract knowledge from data.

This chapter covers three of the most exciting and demanding classes of cloud applications: Big Data, data streaming, and mobile cloud computing. Big Data is a reality; every day, we generate 2.5 quintillion, 2.5×10^{18}, bytes of data. This colossal volume of data is collected every day by devices ranging from inexpensive sensors in mobile phones to the detectors of the Large Hadron Collider, by online services offered by Google and Amazon, or by devices connected by the Internet of Things.

Big Data is a defining challenge for cloud computing; a significant amount of the data collected every day is stored and processed on computer clouds. Big Data and data streaming applications require low latency, scalability, versatility, and a high degree of fault tolerance. Achieving these qualities at scale is extremely challenging.

It makes sense to define a more comprehensive concept of scale for the applications discussed in this chapter. The "scale" in this context means millions of servers operating in concert in large data centers, a very large number of diverse applications, tens of millions of users, and a range of performance metrics reflecting the views of users, on one hand, and those of the CSPs, on the other hand. The scale has a *disruptive* effect: It changes how we design and engineer such systems and broadens the range of problems that can be solved and of the applications that can only run on computer clouds.

Scale amplifies the unanticipated benefits, as well as the dreaded nightmares, of system designers. Even a slight improvement of the individual server performance and/or of the algorithms for resource management could lead to huge cost savings and rave reviews. At the same time, the failure of one of the billions of hardware and software components can be amplified, propagate throughout the entire system, and have catastrophic consequences.

Several important lessons when engineering large-scale systems for Big Data storage and processing are: (a) Prepare for the unexpected because low probability events occur and can cause major disruptions; (b) It is utterly unreasonable to assume that strict performance guarantees can be offered at scale; and (c) It is unfeasible to build fault-free systems beyond a certain level of system complexity; understanding this truth motivated the next best solution, the development of design principle for fault-tolerant systems.

A realistic alternative to the applications discussed in this chapter is to develop tail-tolerant techniques for the distributions of performance metrics, such as response time latency. This means to understand why a performance metric has a heavy-tail distribution, detect the events leading to such an undesirable state as early as feasible, and take the necessary actions to limit its effects. Another design

Cloud Computing. https://doi.org/10.1016/B978-0-32-385277-7.00019-1

principle is that a reasonably accurate result and fast response time are preferable to a delayed best result.

The defining attributes of Big Data are analyzed in Section 12.1. The next sections discuss how Big Data is stored and processed. High-capacity datastores and databases are necessary to store the very large volume of data. Scaling data warehouses and databases poses its own challenges. The Mesa datastore and Spanner and F1 databases developed at Google are discussed in Section 12.2, while another class of Big Data applications combining mathematical modeling with simulation and measurements, dynamic, data-driven applications (DDAS), are discussed in Section 12.3.

Clouds host several classes of data streaming applications, ranging from content delivery data streaming to applications consuming a continuous stream of events. Such applications are discussed in Sections 12.4, 12.5, and 12.6.

Mobile devices, such as smartphones, tablets, laptops, and wearable devices, are ubiquitous and indispensable for life in a modern society. Mobile devices are in a symbiotic relationship with computer clouds, and their users benefit from the democratization of data processing. The user of a mobile device has access to the vast amounts of computing cycles and storage available on computer clouds. Mobile cloud computing enables execution of mobile applications on mobile devices and on computer clouds. Mobile devices act as producers and consumers of data stored on clouds and shared with others.

Section 12.7 is an introduction to mobile computing and its applications, while Section 12.8 covers energy efficiency of mobile computing. Section 12.9 analyzes the effects of latency and presents alternative mobile computing models, including Cloudlets.

Scale allows us to add mission-critical applications demanding very high availability, a topic discussed in Section 12.10. Scale amplifies variability, often causing heavy-tail distributions of critical performance metrics as is the case of latency discussed in Section 12.11. Lastly, edge computing and Markov decision processes are analyzed in Section 12.12.

12.1 Big Data

Some of the defining characteristics of Big Data are the three "Vs," volume, velocity, and variety, along with persistency. Volume is self-explanatory, and velocity means that responses to queries and data analysis requests have to be provided very fast. Variety recognizes the wide range of data sources and data formats. Persistency means that data has a lasting value; it is not ephemeral.

Big Data covers a wide spectrum of data including user-generated content and machine-generated data. Some of the data is highly structured, as in the case of patient records in health care, insurance claims, or mortgage documents. Others are raw data from sensors, log files, or data generated by social media.

Big Data has affected the organization of database systems. The traditional relational databases are unable to satisfy some of these requirements, and NoSQL databases proved to be better suited for many cloud applications. A database *schema* is a way to logically group objects such as tables, views, stored procedures, etc. A schema can be viewed as a container of objects. One can assign a user login permission to a single schema so that the user can only access the objects they are authorized to.

For decades, the database community used *schema-on-write*. First, one defines a schema, then writes the data, and data comes back according to the original schema when reading. The alternative, *schema-*

on-read loads data as-is, and a user-defined filter extracts data for processing. Schema-on-read has several advantages:

- Often, data is a shared asset among individuals with differing roles and differing interests who want to get different insights from that data. Schema-on-read can present data in a schema that is best adapted to the queries being issued.
- It is not necessary to develop a super-schema that covers all datasets when multiple data sets are consolidated.

An insightful discussion of the state of research on databases [3] starts by acknowledging that "Big Data requirements will cause massive disruptions to the ways that we design, build, and deploy data management solutions." The report continues by identifying three major causes for these disruptions: "...it has become much cheaper to generate a wide variety of data, due to inexpensive storage, sensors, smart devices, social software, multiplayer games, and the emerging IoT ... it has become much cheaper to process large amounts of data, due to advances in multicore CPUs, solid state storage, inexpensive cloud computing, and open source software ... not just database administrators and developers, but many more types of people have become intimately involved in the process of generating, processing, and consuming data..."

Big Data revolutionized computing. Chapter 11 presented specialized frameworks for Big Data processing such as MapReduce and Hadoop. The myriad system software components reviewed in Chapter 4, including Pig, Hive, Spark, and Impala, are essential elements of a more effective infrastructure for processing unstructured or semi-structured data. The evidence of the effort to mold the cloud infrastructure to the requirements of Big Data is overwhelming. In spite of their diversity Big Data workloads have a few common traits:

- Data is immutable; consequently, widely used storage systems for Big Data such as HDFS only allow *append* operations.
- Jobs such as MapReduce are deterministic; therefore fault tolerance can be ensured by recomputations.
- The same operations are carried out on different segments of data on different servers; replicating programs is less expensive than replicating data.
- It is feasible to identify a *working set*, a subset of data frequently used within a time window and keep this working set in the memory of the servers of a large cluster. Systems such as Spark [533] and Tachyon [302] exploit this idea to dramatically improve performance.
- *Locality,* the proximity of the data to the location where it is processed is critical for the performance. Supporting data locality is a main objective of scheduling algorithms, including delay scheduling [532] and data-aware scheduling [492], as we have seen in Chapter 9.

The cloud hardware infrastructure also faces a number of challenges. Bridging the gap between the processor speed and the communication latency and bandwidth is a major challenge. This challenge is greatly amplified by response-time constrains when Big Data is processed on the cloud. Addressing this challenge requires prompt adoption of faster networks, such as InfiniBand, and of full bisection bandwidth networks between servers. Remote direct memory access capabilities can also help bridge this gap.

Nonvolatile random-access memories, specialized processors such as GPUs and field-programmable gate arrays (FPGAs), and application-specific integrated circuits (ASICs) contribute to the effort to de-

Table 12.1 Capacity and bandwidth of hard disks (HDD), solid-state disks (SDD), and memory of a modern server according to http://www.dell.com/us/business/p/servers. The network bandwidth is 1.25 GB/sec.

Media	Capacity (TB)	Bandwidth GB/sec
HDD (x12)	12–36	0.2–2
SDD (x4)	1–4	1–4
Memory	0.128–0.512	10–100

velop a scalable infrastructure. The storage technology has dramatically improved, as summarized in Table 12.1.

Some of the enduring challenges posed by Big Data are:

1. Develop a scalable data infrastructures capable to respond promptly to the timing constraints of an application.
2. Develop effective means to accommodate diversity in data management systems.
3. Support comprehensive end-to-end processing and knowledge extraction from data.
4. Develop convenient interfaces for a layman involved in data collection and data analysis.

The next sections address the scalability challenges faced by the development of large data storage systems and by processing very large volumes of data.

12.2 Data warehouses and Google databases for Big Data

A *data warehouse* is a central repository for the important data of an enterprise; it is a core enterprise software component required by a range of applications related to so-called *business intelligence*. Data from multiple operational systems are uploaded and used for predictive analysis and for the detection of hidden patterns in the data. Data models developed using Statistical Learning Theory allow enterprises to optimize their operations and maximize their profits.

Scaling data repositories is very challenging due to the low-latency, scalability, versatility, availability, and fault-tolerance requirements. Google realized early on the need to develop a coherent storage architecture for the massive amounts of data required by their cloud services and by AdWords.[1] This storage architecture reflects the realization that "developers do not ask simple questions of the data, change their data access patterns frequently and use APIs that hide storage requests while expecting uniformity of performance, strong availability and consistent operations, and visibility into distributed storage requests" [173].

Two of the most popular Google data stores, BigTable and Megastore, discussed in Sections 7.11 and 7.12, respectively, were designed and developed early on to support Google's cloud computing services. A major complaint about BigTable is the lack of support for cross-row transactions. Megas-

[1] AdWords consists of hundreds of applications supporting Google's advertising services.

tore supports schematized semi-relational tables and synchronous replication. At least 300 applications within Google use Megastore, including Gmail, Picasa, Calendar, Android Market, and App Engine.

From the experience with the BigTable storage system, the developers of Google's storage software stack learned that it is hard to share distributed storage. Another lesson is that distributed transactions are the only realistic option to guarantee the low latency requited for a high-volume transaction processing system. They also learned that end-user latency matters and that the application complexity is reduced if an application is close to its data [173]. Another important lesson when developing a large-scale system is that making strong assumptions about applications is not advisable.

These lessons led to the development of Colossus, the successor of the GFS as a next-generation cluster-level file system, the Mesa warehouse, and Spanner and F1 databases, discussed in this section. Colossus is built for real-time services and supports virtually all web services, from Gmail, Google Docs, and YouTube, to Google Cloud Storage service offered to third-party developers. The system allows client-driven replication and encoding. The automatically shared metadata layer of Colossus enables availability analysis. To reduce cost Colossus data is typically encoded with Reed–Solomon codes.

Mesa—a scalable data warehouse. The system developed at Google [217] is an example of a data warehouse designed to support measurement data for the multibillion-dollar advertising business of the organization. The system is expected to support near-real-time data processing and be highly available and scalable. The extremely high availability is ensured by geo-replication.[2] Mesa is able to handle petabytes of data, respond to billions of queries accessing trillions of rows of data per day, and update millions of rows per second. The system complexity reflects the stringent requirements including the support for:

- Complex queries such as "How many ad clicks were there for a particular advertiser matching the keyword "fig" during the first week of December between 11:00 AM and 2:00 PM that were displayed on google.com for users in a specific geographic location using a mobile device?" [217].
- Multidimensional data with *dimensional* attributes, called keys, and *measure* attributes, called *values*.
- Atomic updates, consistency, and correctness. Multiple data views defined for different performance metrics are affected by a single user action and all must be consistent. The BigTable storage system does not support atomicity, while the Megastore system provides consistency across geo-replicated data; see Sections 7.11 and 7.12, respectively.
- Availability - planned downtime is not allowed, and unplanned downtime should never be experienced.
- Scalability - accommodates a very large volume of data and a large user population.
- Near-real-time performance - support live customer queries, reports, and updates; allows queries to multiple data views from multiple data centers.
- Flexibility and ability to support new features.

Logical and physical data organization in Mesa. The system stores data using tables with a very large key, K, and value, V, spaces. These spaces are represented by tuples of columns of items of identical

[2] The term *geo-replication* used in the title of the paper [217] means that the Mesa system runs at multiple sites concurrently. The term used in [218] for this strategy supporting high availability is *multihoming*.

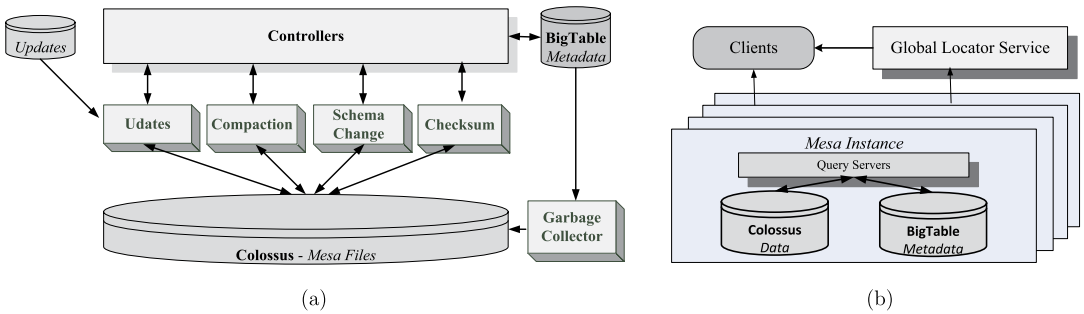

FIGURE 12.1

Mesa instance. (a) The controller/worker subsystem. Controllers interact with four types of workers: update, compaction, schema change, and checksum. The data and metadata are stored on Colossus and BigTable, respectively. (b) The query subsystem of a Mesa instance interacts with the clients and with the Global Location Service.

data type, e.g., integers, strings, or floating-point numbers. The data is horizontally partitioned and replicated.

The table structure and the *aggregation function* $F : V \times V \mapsto V$ are specified by a *table schema*. Function F is associative and often commutative. To maximize the throughput, updates, each consisting of at most one aggregated value for every (table name, key) pair, are applied in batches.

Updates are applied by *version number*, are atomic, and the next update can only be applied after the previous one has finished. The time associated with a version is the time when the version was generated. A query has also a version number and a predicate P on key K.

A *delta* is a pre-aggregation of versioned data consisting of a set of rows corresponding to the set of keys for a range of versions $[V_1, V_2]$ with $V_1 \leq V_2$. Deltas can be aggregated by merging row keys and aggregating values accordingly, e.g.,

$$[V_1, V_2] \,\&\, [V_2 + 1, V_3] \;\rightarrow\; [V_1, V_3]. \tag{12.1}$$

Deltas are immutable, and the rows in one are stored in sorted order in files of limited size. A row consists of multiple row blocks; each row block is transposed and compressed. Each table has one or more *table indexes,* and each table index has a copy of the data sorted according to the index order.

Mesa limits the time a version can be queried to reduce the amount of space. Older versions can be deleted, and queries for such versions are rejected. For example, updates can be aggregated into base $B \geq 0$ with version $[0, B]$, and any updates with $[V_1, V_2]$ with $0 \leq V_1 \leq V_2 \leq B$ can be deleted.

Mesa instances. One Mesa instance runs at every site and consists of two subsystems, the *update/maintenance* and the *query* subsystem shown in Fig. 12.1. The pool of workers of the first subsystem operate on data stored in *Colossus*. *Workers* load updates, carry out table compaction, apply schema changes, and run table checksums under supervision of *controllers* that determine the work to be done and manage metadata stored on BigTable.

To scale, the controller is *sharded* by table; a shard is a horizontal partition of data in a data store and each shard is held on a separate physical storage device. The controller maintains separate queues of work for each type of worker and redistributes the workload of a slow worker to another worker in

the same pool. Workers poll the controller for additional work when idle and notify the controller upon completion of a task. Work requiring global coordination, including schema change and checksums, is initiated by components outside the controller.

The query subsystem includes a pool of query servers that process client queries. A query server looks up BigTable for metadata, determines the files where the data are stored, performs on-the-fly aggregation of data, and converts data from the internal format to the client protocol format. Multiple queries acting upon the same tables are assigned to a group of servers to reduce access time and optimize the system performance.

Mesa instances are running at multiple sites to support a high level of availability. A *committer* coordinates updates at all sites. The committer assigns a new version number to batches of updates and publishes the metadata for the update to the *versions* database, a globally replicated and consistent data store using the Paxos algorithm. Controllers detect the availability of new updates by listening to the changes in the versions database and then assign the work to update workers.

Mesa reads 30 to 60 MB compressed data, updates 3 to 6 million distinct rows, and adds some 3×10^5 thousand new rows per second. It executes more than 500 million queries and returns $(1.7–3.2) \times 10^{12}$ rows per day. Updates arrive in batches about every five minutes, with median and 95th-percentile commit times of 54 seconds and 211 seconds, respectively.

Spanner—a globally distributed database. Scaling traditional databases is not without major challenges. Spanner [117] is a distributed database replicating data on many sites across the globe. Some applications replicate their data across three to five data centers running Spanner in one geographic area. Other applications spread their data over a much larger area, e.g., F1, presented later in this section, maintains five replicas of data around the US. Spanner has been used by many Google applications since its release in 2011. The first user of Spanner was the F1 system.

The data model of each application using Spanner is layered on top of the directory-bucketed key-value mappings supported by the distributed database. Spanner supports consistent backups, atomic schema updates, and consistent MapReduce execution and provides externally consistent reads and writes and globally consistent reads across the database at a time stamp. This is possible because Spanner assigns globally meaningful commit time stamps to transactions reflecting the serialization order, even though transactions may be distributed. Serialization order satisfies *external consistency*: The commit time stamp of transaction T_1 is lower than that of transaction T_2 when T_1 commits before T_2 starts.

Spanner applications can control the number and the location of data centers where the replicas reside, as well as read and write latency. The read and write latencies are determined by how far data is from its users and, respectively, how far are the replicas from one another. The system is organized in *zones*, units of administrative deployment and of physical isolation similar with AWS zones.

Spanner organization. A *zonemaster* is in charge of several thousand *spanservers* in each zone. The clients use *location proxies* to find the spanservers able to serve their data. A *placement driver* handles the migration of data across zones with latencies of minutes, and the *universe master* maintains the state of all zones. A spanserver serves data to the clients. The spanserver implements a Paxos state machine per tablet and manages 100–1 000 tablets. A *tablet* is a data structure implementing a bag of the mapping

$$(key : string, aimestamp : int64) \rightarrow string. \tag{12.2}$$

Table 12.2 Spanner latency and throughput for write, read-only, and snapshot read. The mean and standard deviation over 10 runs reported in [117].

	Latency in ms.			Throughput in Kops/sec.		
Replicas	*Write*	*Read-only*	*Snapshot read*	*Write*	*Read-only*	*Snapshot read*
1	14.4 ± 1.0	1.4 ± 0.1	1.3 ± 0.1	4.1 ± 0.5	10.9 ± 0.4	13.5 ± 0.1
3	13.9 ± 0.6	1.3 ± 0.1	1.2 ± 0.1	2.2 ± 0.5	13.8 ± 3.2	38.5 ± 0.3
5	14.4 ± 0.4	1.4 ± 0.05	1.3 ± 0.4	2.8 ± 0.3	25.3 ± 5.2	50.0 ± 1.1

A great deal of attention is paid to replication and concurrency control. A set of replicas is called a *Paxos group*. The system administrators control the number and types of replicas and the geographic placement of them. Applications control the manner the data is replicated.

Each spanserver implements a *lock table* for concurrency control. Every replica has a leader. Every Paxos write is logged twice, once in the tablet's log and once in the Paxos log. The $(key, value)$ mapping state is stored in the tablet. At each site, the local spanserver software stack includes a site *participant leader* communicating with its peers at other sites. This leader controls a *transaction manager* and manages the lock table.

The system stores all tablets in Colossus. The implementation of the Paxos algorithm is optimized. The algorithm is pipelined to reduce latency. The leaders of the Paxos algorithm discussed in Section 10.13 are long-lived, about 10 seconds.

Spanner *directories*, also called *buckets*, are sets of contiguous keys sharing a common prefix. A directory is the smallest data unit whose placement can be specified by an application. A background task called *moved* moves data directory-by-directory between the Paxos groups. This task is also used to add or remove replicas to/from Paxos groups.

Spanner transactions. The database supports read-write transactions, read-only transactions, and snapshot reads. A *read–write* transaction implements a standalone *write*, and the concurrency control is pessimistic. A *read-only* transaction benefits from snapshot isolation.[3] A *snapshot read* is a lock-free *read* in the past, at a time stamp specified by the client or at a time stamp chosen by the system before a time stamp upper bound specified by the client.

The system supports atomic schema-change transactions. The transaction is assigned a time stamp t in the future. The time stamp is registered during the prepare phase so that the schema changes on thousands of servers can complete with minimal disruption to other concurrent activity. Reads and writes, implicitly depending on the schema, are synchronize with any registered schema-change and may proceed if their time stamps precede time t; otherwise, they must block until schema changes.

Some of the results of a micro-benchmark reported in [117] are shown in Table 12.2. The data was collected on time-shared systems with spanservers running in each zone on 4 cores AMD Barcelona 2200MHz servers with 4GB RAM. The two-phase commit scalability was also assessed: it increases from 17.0 ± 1.4 ms for one participant to 30.0 ± 3.7 for 10 participants, to 71.4 ± 7.6 for 100, and 150.5 ± 11.0 ms for 200.

TrueTime. A two-phase locking is used for transactional reads and writes. An elaborate process for time stamp management based on the TrueTime is used. The *TrueTime* API enables the system to

[3] Snapshot isolation is a guarantee that all reads made in a transaction see a consistent snapshot of the database.

support consistent backups, atomic schema updates, and other desirable features. This API represents time as a *TTinterval* with the starting and ending times of type *TTstamp*. A *TTinterval* has bounded time uncertainty. Three methods using t of type *TTstamp* as argument are supported

$TT.now()$—returns a $TTinterval : [earliest, latest]$.
$TT.after(t)$—returns *true* if time t has definitely past.
$TT.before(t)$—returns *true* if time t has definitely not arrived.

TrueTime guarantees that for an invocation

$$tt = TT.now(), tt.earliest \leq t_{abs}(e_{now}) \leq tt.latest, \tag{12.3}$$

where e_{now} is the invocation event. Each data center will have one *time master,* and each server will have its own *timeslave* daemon interacting with several time masters to reduce the probability of errors.

TrueTime guarantees correctness of concurrency control and supports externally consistent transactions, lock-free read-only transactions, and nonblocking reads in the past. It guarantees that a whole-database audit at time t sees the effects of every transaction committed up to time t. The results show that refining clock uncertainty associated with jitter or skew allows building distributed systems with much stronger time semantics.

F1-scaling a traditional SQL database. A traditional database design shares the goals with the design of data warehouses, i.e., scalability, availability, consistence, usability, and latency hiding. Google system developers realized that scaling up their sharded MySQL implementation supporting online-transaction processing (OLTP) and online analytical processing (OLAP) systems was not feasible. Instead, a distributed SQL database called F1 was developed. F1 uses Spanner and stores data on Colossus File System (CFS).

The F1 database [447] has been used since 2012 for the AdWords advertising ecosystem at Google. F1 inherits Spanner's scalability, synchronous replication, strong consistency, and ordering properties and adds to them: (1) distributed SQL queries; (2) secondary indexes transactionally consistent; (3) asynchronous schema changes; (4) optimistic transactions; and (5) automatic change history recording and publishing.

Fig. 12.2 illustrates the F1 organization. Users interact with F1 using a *client* library. F1 servers are colocated in each data centers along with Spanner servers. Multiple Spanner instances run at each site along with multiple CFS instances. CFS is not a globally replicated service, and Spanner instances communicate only with local CFS instances. Storing data locally reduces the latency. The system is scalable, and its throughput can be increased by adding additional F1 and Spanner servers. The commit latencies are in the range 50–150 ms due to synchronous data replication across multiple data centers.

F1 has a logical Relational Database Management System schema with some extensions, including explicit table hierarchy and columns with Protocol Buffer data types. F1 stores each child table clustered with, and interleaved within, the rows from its parent table. Table columns contain structured data types based on the schema and binary encoding format of Google's open-source Protocol Buffer library.

F1 supports *nonblocking schema changes* in spite of several challenges including:

- The scale of the system.
- High availability and tight latency constraints.
- The requirement to continue queries and transaction processing while schemas change.

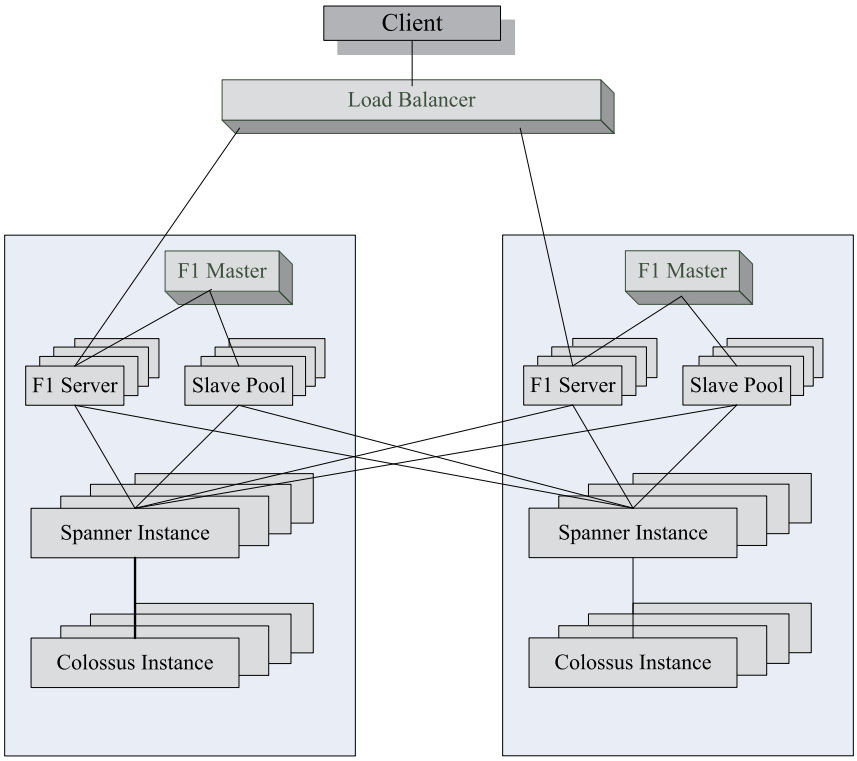

FIGURE 12.2

F1 architecture. A load balancer distributes the workload to F1 server instances at every site which, in turn, interact with Spanner instances at all sites where F1 is running. F1 data is stored on Colossus at the same site. The shared slave pool execute parts of distributed query plans on behalf of regular F1 servers. F1 master monitors the health of slave pool processes and communicates to F1 server the list of available slaves.

- Each F1 servers has a copy of the schema in local memory for efficiency reasons; thus it is impractical to require atomical schema updates for all servers.

The schema change algorithm requires that at most two different schema are active at any time; one can be the current schema, the other the next schema. A server cannot use a schema after its lease expires. A schema change is divided in multiple phases such that consecutive pairs of phases are mutually compatible.

F1 transactions. F1 supports ACID transactions discussed in Section 7.3 and is required by systems such as financial systems and AdWords. The three types of F1 transactions built on top of Spanner transactions are:

1. Snapshot transactions—read-only transactions with snapshot semantics used by SQL queries and by MapReduce. Snapshot isolation is a guarantee that all reads made in a transaction see a consistent

snapshot of the database. The transaction itself will successfully commit only if no updates it has made conflict with any concurrent updates made since that snapshot.
2. Pessimistic transactions—map to the same Spanner transaction type.
3. Optimistic transactions—have an arbitrarily long lockless read phase, followed by a short write phase, and are the default transactions used by the clients.

Optimistic transactions have a number of benefits:

- Are long-lasting.
- Can be retried transparently by an F1 server.
- The state is kept on the client and is immune to server failure.
- The time stamp for a read can be used by a client for a speculative write that can only succeed if no other writes occurred after the read.

Optimistic transactions have two drawbacks, insertion phantoms and low throughput for high contention. Insertion phantoms occur when one transaction selects a set of rows, another transaction inserts rows that meet the same criteria, and, then, different results are produced when the first transaction re-executes the query.

The default locking in F1 is at the row-level, though concurrency levels can be changed in the schema. By default, tables are change-tracked unless the schema shows that the some tables or columns have opted out. Every transaction creates one or more ChangeBatch Protocol Buffers, including the primary key and before and after values of changed columns for each updated row.

12.3 Dynamic data-driven applications

There are many applications, some involving Big Data, which combine mathematical modeling with simulation and measurements. For example, *large-scale-dynamic-data* refers to data captured by sensing instruments and control in engineered, natural, and societal systems. Some sensing instruments capture very large volumes of data; this is the case of the Large Hadron Collider at CERN, discussed in Section 3.15. A special case of such systems, Dynamic Data-Driven Application Systems (DDDAS), is discussed in this section. Such applications can benefit from dataflow architectures and programming models, such as FreshBreeze, developed by Professor Jack Dennis at MIT [140,141,305].

Dynamic Data-Driven Application Systems. Efforts to analyze, understand, and predict the behavior of complex systems often require a dynamic feedback loop. Sometimes, the simulation of a system uses measurement data to refine the system model through successive iterations or to speedup the simulation. Alternatively, the accuracy of a measurement process can be iteratively improved by feeding the data to the system model and then using the results to control the measurement process. In all these cases, the computation side and the instrumentation have to work in concert. The mathematical modeling, the simulation algorithms, the control system, the sensors, and the computing infrastructure of a DDDAS system should support an optimal logical and physical dataflow through a feedback loop.

Some DDDAS applications discussed in http://www.1dddas.org/activities/2016-pi-meeting are: adaptive stream mining, modeling of nanoparticle self-assembly processes, dynamic integration of motion and neural data to capture human behavior, real-time assessment and control of electric microgrids, optimization, and health monitoring of large-scale structural systems.

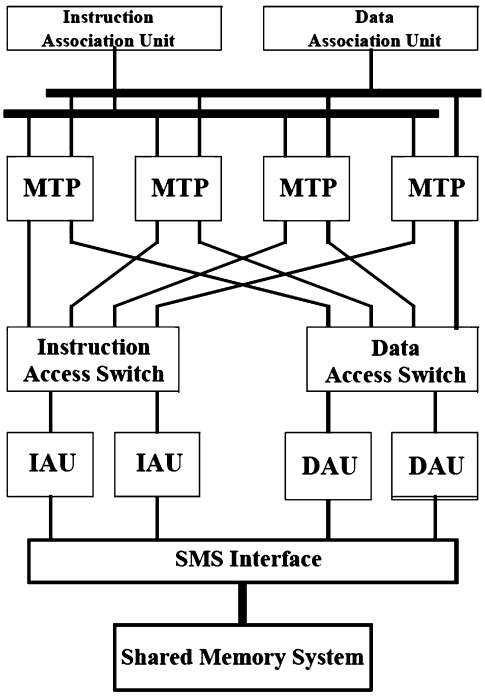

FIGURE 12.3

A FreshBreeze system consists of multithreaded processors (MTPs), a shared memory system (SMS), instruction and data access units (IAU) and (DAUs), and instruction and data switches [140].

The FreshBreeze multiprocessor architecture and FreshBreeze model of thread execution. A chip architecture guided by modular programming principles described in [140] is well-suited for DDDAS applications, as we shall see in the discussion of the Mahali project. The system is designed to satisfy the software design principles discussed in Section 3.11. One of the distinctive features of the architecture is to prohibit memory updates and use a cycle-free heap. Objects retrieved from memory are immutable, and the allocation, release, and garbage collection of fixed-size chunks of memory are implemented by hardware mechanisms. This eliminates entirely the challenges posed by the cache coherence.

System organization is depicted in Fig. 12.3. The system consists of several multithreaded scalar processors (MTPs), a shared memory system (SMS) organized as a collection of fixed-size chunks of 1 024 bits, and an interconnection network. An MTP supports up to four execution threads involving integer and floating-point operations. The MTPs communicate with Instruction Access Units (IAU) and Data Access Units (DAUs) to access chunks containing instructions and data, respectively. The access units have multiple slots of size equal to the size of memory chunks and maintain chunk usage data used by LRU algorithms for purging data to the SMS. The SMS performs automatically all memory updates including garbage collection.

An array is represented by a tree of chunks with element values at level 0. The number of tree levels is determined by the largest index value with a defined element, up to a maximum of eight levels. A collection of chunks containing pointers to other chunks forms a *heap*. The FreshBreeze execution model allows only the creation of *cycle-free* heaps. Many applications use also stream data types, unending series of values of uniform type. A Fresh Breeze stream is represented by a sequence of chunks, each chunk containing a group of stream elements. A chunk may include a reference to the chunk holding the next group of stream elements. As an alternative, an auxiliary chunk may contain "current" and "next" references.

The FreshBreeze execution model is discussed in [141]. A *master thread* spawns *slave threads* and initializes a *join point* providing the slave with a join ticket, similar to the return address of a method. Any slave thread may be a master for a group of slaves. The master does not continue after spawning the slaves, and there is no interaction between the master and the slaves or among the slaves other than the contribution of each to the continuation thread. A program can generate an arbitrary hierarchy of concurrent threads corresponding to the available application parallelism.

A join point is a special entry in the local data segment of the master thread that provides space for a record of master thread status and for the result produced by the slave. Only one slave can be given a join ticket to the same join point to avoid race conditions. Several instructions are used to access a join point:

1. *Spawn*—sets a flag, stores a joint ticket in the slave's local data segment, and starts slave execution.
2. *EnterJoinResult*—allows the slave to save the result in the join point and then quit.
3. *ReadJoinValue*—returns the join value if it is available or suspends the master if the join value has not yet been entered.

The operation of a thread is deterministic because any heap operation either reads data or creates private data, and operations at a join point are independent of the order of slave thread arrival.

The parallelization of a dot product of two vectors discussed next illustrates an application of the FreshBreeze execution model [305]. First, the vectors are converted to tree-based memory chunks. The vectors are split into 16-element segments and organized as a tree structure. The leaf chunks hold the actual values, while the internal nodes of the tree store chunks holding handles of chunks that point to other chunks.

The *TraverseVector* thread takes the roots of two trees-of-chunks of vectors A and B as inputs and checks the depth of the tree to see if it is a leaf node. If not a leaf node, then it recursively spawns *TraverseVector* threads taking the root handles of the next level as inputs. At the leaf level, a *Compute* thread is spawned to compute the dot product of 16 elements and return the result sum to the *Sync* chunk. The continuation *Reduce* thread adds all partial results in the *Sync* chunk filled by lower level *Compute/Reduce* threads. Then, the *Reduce* thread returns the handle of the *Sync* chunk to the upper level *Sync* chunk until reaching the root level *Sync* chunk as the final result.

Space Weather Monitoring. The Mahali[4] space weather monitoring project at MIT captures the quintessence of DDDAS applications. The project uses multicore mobile devices, such as phones and tablets, to form a global space weather monitoring network. Multifrequency GPS sensor data is processed on a cloud to reconstruct the structure of the space environment and its dynamic changes.

[4] "Kila Mahali" means "everywhere" in the Swahili language; see https://mahali.mit.edu/.

The core ideas of the project are: (1) leverage the entire ionosphere as a sensor for ground-based and space-based phenomena; and (2) take advantage of mobile technology as a game changer for observatories.

12.4 Data streaming

Data streaming is *the transfer of data at a steady high-speed rate, with low and well-controlled latency.* The data volume in data streaming is high and decisions have to be made in real-time. High-definition television (HDTV) is a ubiquitous application of data streaming.

Cloud data streaming services support content distribution from many organizations, e.g., AWS hosts Netflix and Google cloud hosts YouTube. Clouds support a variety of other data streaming applications in addition to hosting content providers. For example, AWS supports several streaming data platforms, including Apache Kafka, Apache Flume, Apache Spark Streaming, and Apache Storm.

Data streaming versus batch processing. According to AWS: "Streaming data ... is generated continuously by thousands of data sources, which typically send in the data records simultaneously, and in small sizes (order of Kilobytes). Streaming data includes a wide variety of data such as log files generated by customers using your mobile or web applications, ecommerce purchases, in-game player activity, information from social networks, financial trading floors, or geospatial services, and telemetry from connected devices or instrumentation in data centers. This data needs to be processed sequentially and incrementally on a record-by-record basis or over sliding time windows, and used for a wide variety of analytics including correlations, aggregations, filtering, and sampling."

There are important differences between streaming and batch data processing:

1. Streaming processes individual records or microbatches, rather than large batches.
2. Streaming processes only the most recent data or data over a rolling time window, rather than the entire, or a large segment of a, data set.
3. Streaming requires latency of milliseconds, rather than minutes or hours.
4. Streaming provides simple response functions, aggregates, and rolling metrics, rather than carrying out complex analytics.
5. It can be hard to reason about the global state in data streaming because different nodes may be processing data that arrived at different times, while in batch processing the system state is well defined and can be used to checkpoint and later restart the computation.

These differences suggest that the data streaming programming models and the APIs will be different from the ones for batch processing discussed in Chapters 4 and 11. It is therefore necessary to develop new data processing models for cloud data streaming services. Such models should be simple, yet effective.

Spark Streaming. Spark Streaming system along with a data streaming model, called D-Streams, are discussed in [534]. D-Streams model offers a high-level functional programming API, strong consistency, and efficient fault recovery. This model, prototyped as Spark Streaming, achieves fault tolerance by replication; there are two processing nodes, the upstream backup and a downstream node.

To address the latency concerns, the Spark Streaming system relies on a storage abstraction, *Resilient Distributed Dataset* (RDD), to rebuild lost data without replication. A *parallel recovery* mech-

anism involve all cluster nodes working in concert to reconstruct lost data. The system divides time in short intervals, stores the input data received during each interval in RDD, and then processes data via deterministic parallel computations that may involve MapReduce and other frameworks. A D-Stream is a collection of RDDs.

Spark Streaming provides an API similar to DryadLINQ in the Scala language and supports stateless operations acting independently in each time interval, as well as aggregation over time window. To recover in the case of node failures, D-Streams and RDDs track their lineage using a dependency graph. The system supports:

1. Stateless transformations available in batch frameworks, such as *map, reduce, groupBy*, and *join,* and provides two types of operators: (i) stateless and statefull transform operators, the first act independently on each time interval and the second share data among intervals; and (ii) output operators that save data, e.g., store RDDs on HDFS.
2. New statefull operations: (i) windowing - a window groups records from a range of past intervals into one RDD; (ii) incremental aggregation over a window; and (iii) time-skewed joins when a stream is combined with RDDs.

Spark Streaming uses an inefficient upstream backup approach, does not support finer checkpointing because it creates checkpoints every minute, and depends on application idempotency and system slack for recovery.

Zeitgeist and MillWheel. Google's Zeitgeist,[5] a system used to track trends in web queries, is a typical application of data streaming. The system builds a historical model of each query and continually identifies queries that spike or dip. Zeitgeist processing pipeline includes a window counter with query searches as input, a model calculator and a spike/dip detector, and an anomaly notification engine. System buckets records arriving in one-second intervals and then compares the actual traffic for each time bucket to the expected traffic predicted by the model. An example of Zeitgeist aggregation of one-second buckets of (key, value, time stamp) query triplets given in [15] is shown in Fig. 12.4.

Google MillWheel framework developed for building fault-tolerant and scalable data streaming systems supports persistent state [15] and addresses some of the limitations of Spark Streaming and other data streaming systems. Zeitgeist system provided the initial requirements for MillWheel stream processing services. Some of these requirements are:

- Immediate data availability for data processing services.
- Exposure of persistent state abstraction to user applications.
- Graceful handling of out-of-order data.
- Exactly once delivery of records.
- Constant latency as the system scales up.
- Monotonically increasing low watermarking of data time stamps.

[5] Zeitgeist is a German word literary translated as *time mind* and used with the sense of *the spirit of the time*. This concept, attributed to Georg Wilhelm Friedrich Hegel, an important figure of the German idealism philosophy school, constitutes the dominant ideas in the society at a particular time.

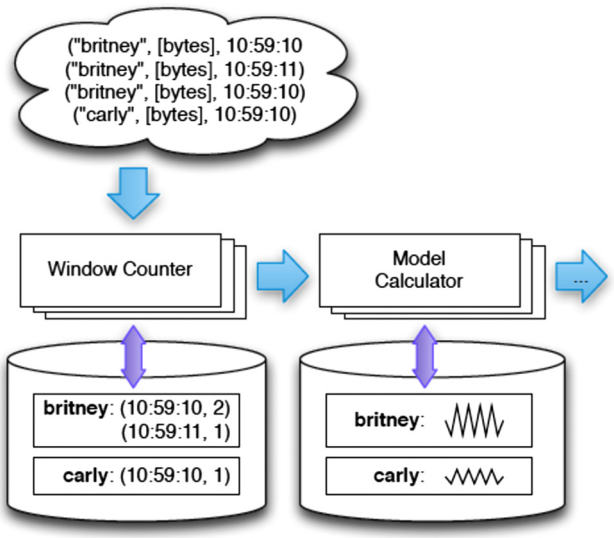

FIGURE 12.4

Zeitgeist aggregation of one-second buckets of (key, value, time stamp) query triplets [15].

MillWheel users express the logic of their data streaming applications as a directed graph where the stream of data records is delivered along the graph edges. A node or an edge in the arbitrary topology can fail at any time without affecting the correctness of the result. Record delivery is idempotent.[6]

The data structures used by MillWheel as input and output are (key, value, time stamp) triplets. The *key* and the *value* can be any string. The *time stamp* is typically close to the wall time of the event, though it can be assigned an arbitrary value by MillWheel. Input data trigger computations invoking either user-defined actions, or MillWheel primitives.

The key extraction function uses a user-assigned key for the aggregation and the comparison among records. For applications such as Zeitgeist, the key could be the text of the query. MillWheel uses the concept of *low-water mark* to bound the time stamp of future records. Waiting for the low-water mark allows computations to reflect a more complete picture of the data. *Timers* are programmatic hooks that trigger at a specific wall time or at a low-water mark value for a particular key.

A user-defined computation subscribes to input streams and publishes output streams, and the system guarantees delivery of both streams. Computation is run in the context of a specific key. The processing steps for an incoming record are:

1. Check for duplication and discard a previously seen record.
2. Run user code and identify pending changes to the timers, state, and production.
3. Commit pending changes to a backing store.

[6] An idempotent operation can be carried out repeatedly without affecting the results.

4. Acknowledge senders.
5. Send pending downstream productions.

There are two types of productions, strong and weak. Strong productions support handling of inputs that are not necessarily ordered or deterministic. Checkpointing of *strong productions* is done before delivery as a single atomic write; the state modification is done at the same time as the atomic write. *Weak productions* are generated when the application semantics allows the user to disable the strong production option.

The computations are pipelined and may run on different hosts of the MillWheel cluster. To record the persistent state, the system uses either BigTable or other databases supporting single-row updates. A replicated master balances the load and distributes it based on lexicographic analysis of the record keys.

Measurements conducted on the system report low latency and scalability, well within the targets set at Google for stream processing. A median record delay of 3.6 milliseconds and a 95th-percentile latency of 30 milliseconds are reported for an experiment on 200 CPUs.

MillWheel is not suitable for monolithic computations whose checkpointing would interfere with the dynamic load balancing necessary to ensure low latency.

Caching strategies for data streaming. Routers of a Content-Centric Network (CCN)—see Section 6.12—use several replacement policies. LRU (Least Recently Used) keeps in cache *the most recently used content,* while LFU (Least Frequently Used) uses *cache request history to keep in cache highly used content*.

There are obvious limitations of both LRU and LFU. The former ignores the history, e.g., items highly used in the past are kept in cache in spite of a new patterns of access, while the latter ignores the popularity of an item. The combinations of the two, the Least Recently/Frequently Used replacement strategies, support tradeoffs by specifying a weight for each request that decays exponentially over time. The sLRFU (streaming Least Recently/Frequently Used) introduced in [419] aims to maximize cache. The novelties introduced by sLRFU are:

1. Partitions a cache of size C into an LFU-managed area using a sliding window of size k and an LRU-managed area of size $C - k$.
2. Estimates the top-k most popular data item in a sliding window of requests, keeps the frequently used ones in cache, and discards the old ones. k is set dynamically based on bounds given by the streaming algorithm.
3. For every request:

   - Increases the reference counter for the data item referenced by the request and decreases the counter for the request falling out of the window.
   - If the data requested is not in cache, it recovers the data and delivers it.
   - Using the last N requests, possibly rearranges the top-k most popular elements in cache (adding and/or removing content from the list) and complements the available spaces in cache with the $C - k$ most recently requested elements that are not already in cache.

Simulation results reported in [419] show that sLRFU has a hit rate of 70%, compared with a baseline LRU of 65%.

12.5 A dataflow model for data streaming

A more sophisticated programming model for data streaming was developed at Google. Alphabet, the holding company consisting of Google's core businesses and future-looking and finance activities, earns billions of dollars from advertising embedded in data streaming. That's why Google is interested to support an effective computational model for event-time correlations. The Dataflow model described in [16] is based on MillWheel [15] and FlumeJava [90].

Instead of the widely used terminology distinguishing batch processing from streaming, the two types of data processing are identified by a characterization of the input data: *bounded* for batch processing and *unbounded* for streaming. The implications of this dichotomy reflect what the execution engine is expected to do. In the bounded case, the engine processes a data set of known contents and size. In the unbounded case the engine processes a dynamic data set where one never knows if the set is complete because new records are continually added and old ones are retracted.

Dataflow SDK motivation for developing a new model is that grooming unbounded data sets to look like bounded ones does not allow an optimal balance among correctness, latency, and cost. It is thus necessary to refresh, simplify, and have flexible programming models for unbounded datasets.

The shortcomings of previous models are perfectly well illustrated by Google applications. For example, assume that a streaming video provider wants to bill advertisers placing their adds along with the video stream for the amount of advertising watched. This requires the ability to identify who watched a video stream and each add and for how long.

Existing systems did not provide exactly once semantics, do not scale well, are not fault tolerant, and have high latency, while others such as MillWheel or Spark Streaming do not have high-level programming support for event-time correlations. The model introduced in [16] "allows for calculation of event-time ordered results, windowed by features of the data themselves, over an unbounded, unordered data source, with correctness, latency, and cost tunable across a broad spectrum of combinations."

It also "Decomposes pipeline implementation across four related dimensions, providing clarity, composability, and flexibility: *What* results are being computed. *Where* in event time they are being computed. *When* in processing time they are materialized. *How* earlier results relate to later refinements." The design of the system was guided by several principles:

- Never rely on the notion of completeness.
- Encourage clarity of implementation.
- Support data analysis in the context it was collected.

The new model uses *windowing* to split a dataset into groups of events to be processed together, an essential concept for unbounded data processing. Windowing is almost always time-based and the windows can be static/fixed or sliding. A *sliding* window has a size and a sliding period, e.g., a two-hour window starting every 10 minutes. *Sessions* are sets of windows related by a data attribute, e.g., key for key-value datasets.

An *event* causes the change of the system state, for example, the arrival of a new record in case of data streaming. Typically, the *time skew*, the difference between the event time and the event processing time, varies during processing.The distinction between the event time and the processing time of the event is illustrated in Fig. 12.5.

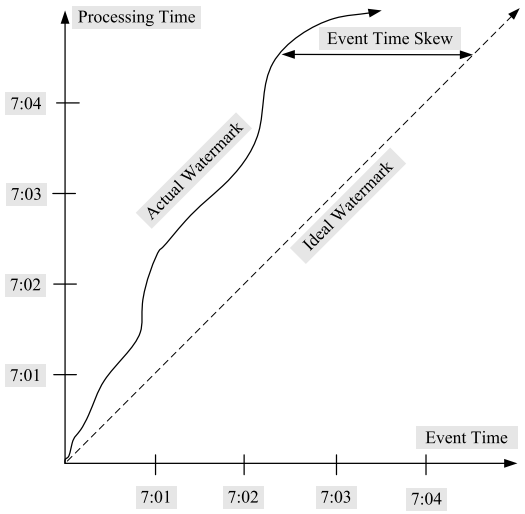

FIGURE 12.5

Event time skew. *Event time* is the wall clock time when the event occurred; this time never changes. *Event processing time* is the time when the event was observed during the processing pipeline; this time changes as the event flows through the processing pipeline. For example, an event occurring at 7:01 is processed at 7:02.

Dataflow SDK uses the $ParDo$ and $GroupByKey$ operations of FlumeJava, discussed in Section 10.15, and defines a new operation $GroupByKeyAndWindow$. The entities flowing through the pipeline are four-tuples $(key, value, eventtime, window)$.

Several operations involving windows are defined. *Window assignment* replicates the object in every window; the $(key, value)$ pairs are duplicated in windows overlapping the time stamp of an event. *Window merging* is a more complex operation because it involves the six steps in Fig. 12.6 [16]. In this example, the window size is 30 minutes, and there are four events, three with key k_1 and one with key k_2.

In Step 2, four-tuples $(key; value; eventtime; window)$ are transformed as three-tuples $(key; value; window)$. The $GroupByKey$ combines the three events with k_1 in Step 3. Overlapping windows $[13:02, 13:32]$ and $[13:20, 13:50]$ for k_1 are merged as $[13:02, 13:50]$ in Step 4. Then, in Step 5, the two k_1 events with values v_1 and v_4 in the same window $[13:02, 13:50]$ are grouped together. Finally, in Step 6, time stamps are added.

Watermarks used in MillWheel to trigger processing of the events in a window cannot by themselves ensure correctness; sometimes late data is missed. At the same time, a watermark may delay pipeline processing. The Dataflow system uses *triggers* to determine the time when the results of groupings are emitted as panes.[7]

The system has several refinement mechanisms to control how several panes are related to one another. Window contents are discarded once triggered, provided that later pipelines stages expect

[7] A pane is a well-defined area within a window for the display of, or interaction with, a part of that window's application or output.

$$(k_1, v_1, 13{:}02, [0, \infty)),$$
$$(k_2, v_2, 13{:}14, [0, \infty)),$$
$$(k_1, v_3, 13{:}57, [0, \infty)),$$
$$(k_1, v_4, 13{:}20, [0, \infty))$$

$$\downarrow \; AssignWindows(Sessions(30m))$$

$$(k_1, v_1, 13{:}02, [13{:}02, 13{:}32)),$$
$$(k_2, v_2, 13{:}14, [13{:}14, 13{:}44)),$$
$$(k_1, v_3, 13{:}57, [13{:}57, 14{:}27)),$$
$$(k_1, v_4, 13{:}20, [13{:}20, 13{:}50))$$

$$\downarrow \; DropTimestamps$$

$$(k_1, v_1, [13{:}02, 13{:}32)),$$
$$(k_2, v_2, [13{:}14, 13{:}44)),$$
$$(k_1, v_3, [13{:}57, 14{:}27)),$$
$$(k_1, v_4, [13{:}20, 13{:}50))$$

$$\downarrow \; GroupByKey$$

$$(k_1, [(v_1, [13{:}02, 13{:}32)),$$
$$(v_3, [13{:}57, 14{:}27)),$$
$$(v_4, [13{:}20, 13{:}50))]),$$
$$(k_2, [(v_2, [13{:}14, 13{:}44))])$$

$$\downarrow \; MergeWindows(Sessions(30m))$$

$$(k_1, [(v_1, [\mathbf{13{:}02}, \mathbf{13{:}50})),$$
$$(v_3, [13{:}57, 14{:}27)),$$
$$(v_4, [\mathbf{13{:}02}, \mathbf{13{:}50}))]),$$
$$(k_2, [(v_2, [13{:}14, 13{:}44))])$$

$$\downarrow \; GroupAlsoByWindow$$

$$(k_1, [([\mathbf{v_1}, \mathbf{v_4}], [13{:}02, 13{:}50)),$$
$$([\mathbf{v_3}], [13{:}57, 14{:}27))]),$$
$$(k_2, [([\mathbf{v_2}], [13{:}14, 13{:}44))])$$

$$\downarrow \; ExpandToElements$$

$$(k_1, [v_1, v_4], \mathbf{13{:}50}, [13{:}02, 13{:}50)),$$
$$(k_1, [v_3], \mathbf{14{:}27}, [13{:}57, 14{:}27)),$$
$$(k_2, [v_2], \mathbf{13{:}44}, [13{:}14, 13{:}44))$$

FIGURE 12.6

Six window-merging steps: 1. AssignWindow; 2. Drop time stamps; 3. GroupByKey; 4. MergeWindows—based on windowing strategy; 5. GroupAlsoByWindow—for each key group values by window; 6. ExpandToElements—expand per-key, per-window groups into (key; $value$; $eventtime$; $window$) tuples, with new per-window time stamps.

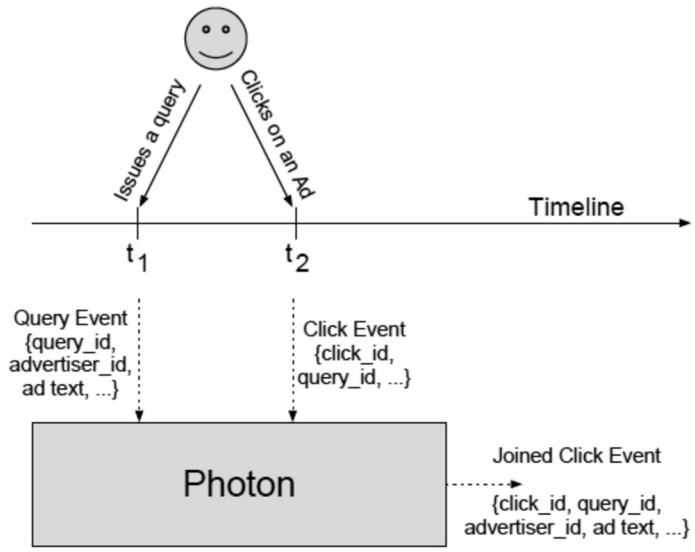

FIGURE 12.7

The primary and the secondary stream events in Photon are the query stream and ad clickstream events. At time t_1, a web server responding to a query also serves an ad. The user clicks on the ad at time t_2, and then the query event and the click event are combined into a joined click event [24].

the values of the triggers to be independent. The window contents are also discarded when the user expresses no interest in later events. The contents of a window can be saved in persistent storage when needed for refining the system state.

12.6 Joining multiple data streams

Applications such as advertising, IP network management, and telephone fraud detection require ability to correlate in real-time events occurring in separate high-speed data streams. For example, Google search engine delivers ads along with answers to queries. The Photon system [24] developed for Google Advertising System produces joint logs for the query data stream and the ad click data stream. The joint logs are used for billing the advertisers. A typical Photon scenario is illustrated in Fig. 12.7.

Photon processes millions of events per minute with an average end-to-end latency of less than 10 seconds. The system joins 99.9999% events within a few seconds and 100% events within a few hours. Proton designers had to address a set of challenges, including:

- Each click on an ad should be processed once and only once. This means: at-most-once semantics at any point of time, near-exact semantics in real-time, and exactly once semantics, eventually. If a click is missed, Google loses money; if the click is processed multiple times, the advertisers are overcharged.

- Automated data center-level fault tolerance. Manual recovery takes too long. Photon instances in multiple data centers will attempt to join the same input event, but must coordinate their output to guarantee that each input event is joined at-most-once.
- Latency constraints. Low latency is required by advertisers because it helps optimize their marketing campaigns. Load balancers redirect a user request to the closest running server, where it is processed without interacting with any other server.
- Scalability. The event rates are very high now and are expected to increase in the future; therefore, the system has to scale up to satisfy latency constraints.
- Combine ordered and unordered streams. While the query stream events are ordered by time stamps, the events in the clickstream may occur at any time and are not sorted. Indeed, the user may click on the ad long after the results of the query are displayed.
- Delays due to the system scale. The servers generating query and click events are distributed over the entire world. The volume of query logs is much greater than that of the click log; thus the query logs can be delayed relative to the click logs.

Photon organization and operation. *IdRegistry* stores the critical state shared among running Photon instances all over the world. This critical state includes the *eventId* of all events joined in the last N days. Before writing a joined event, each instance checks whether the *eventId* already exists in the *IdRegistry*; if so, it skips processing the event, otherwise it adds the event to *IdRegistry*.

IdRegistry is implemented using the Paxos protocol discussed in Section 10.13. An in-memory key-value store based on PaxosDB is consistently replicated across multiple data centers to ensure availability in case of the failure of one or more data centers. All operations are carried out as read–modify–write transactions to ensure that writing to the *IdRegistry* is carried out if and only if the event is not already in.

An *eventId* uniquely identifies the server, the process on that server that generated the event, and the time the event was generated

$$EventId = (ServerIP, ProcessId, Timestamp). \tag{12.4}$$

Events with different Ids can be processed independently of each other. This allows the *EventId space* of the *IdRegistry* to be partitioned into disjoint shards. *EventId*'s from separate shards are managed by separate *IdRegistry* servers.

Identical Photon pipelines run at multiple data centers around the world. A Photon pipeline has three major components, shown in Fig. 12.8:

1. *Dispatcher* - reads the stream of clicks and feeds them to the joiner.
2. *EventStore* - supports efficient query lookup.
3. *Joiner* - generates joined output logs.

The joining process involves several steps:

1. The *Dispatcher* monitors the logs, and, when new events are detected, it looks-up the *IdRegistry* to determine if the *clickId* is already recorded.
   - If already recorded, skip processing the click.
   - Else, the dispatcher sends the event to the joiner and waits for a reply. To guarantee at-least-once semantics, the dispatcher resends it until it gets a positive acknowledgment from the joiner.

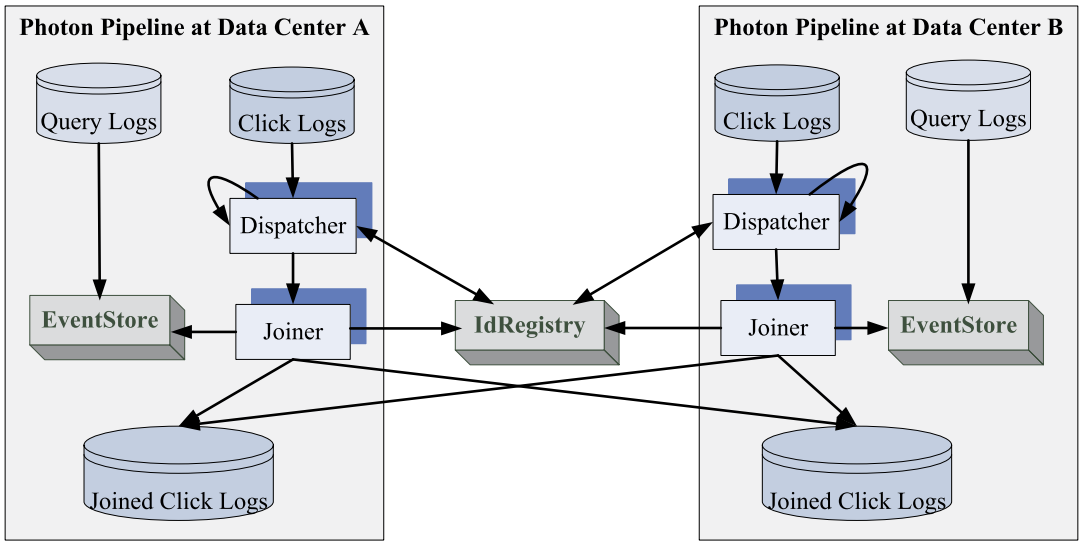

FIGURE 12.8

Photon pipeline organization. The pipeline at each site includes a *Dispatcher*, an *EventStore,* and a *Joiner* operating on query logs, click logs, and joined click logs. *IdRegistry* stores the critical state shared among running Photon instances all over the world.

2. The *Joiner* extracts *queryId* and carries out an *EventStore* lookup to locate the corresponding query. If the query is

   - Found: then the joiner attempts to register the *clickId* in the *IdRegistry*.
     - If the *clickId* is in the *IdRegistry,* the *Joiner* assumes that the join has already been done.
     - If not, the *clickId* is recorded in the *IdRegistry,* and the event is recorded in the joint event log.
   - Not found: then the *Joiner* sends a failure response top the dispatcher; this will cause a retry.

In the US, there are five replicas of the *IdRegistry* in data centers in three geographical regions up to 100 ms apart in round-trip latency. The other components in the pipelines are deployed in two geographically distant regions on the east and west coasts. According to [24]: "during the peak periods, Photon can scale up to millions of events per minute,... each day, Photon consumes terabytes of foreign logs (e.g. clicks), and tens of terabytes of primary logs (e.g. queries), ...more than a thousand *IdRegistry* shards run in each data center,... each data center employs thousands of *Dispatchers* and *Joiners*, and hundreds of *CacheEventStore* and *LogsEventStore* workers."

12.7 Mobile computing and applications

Mobile devices are in a symbiotic relationship with computer clouds. "Mobile cloud computing at its simplest, refers to an infrastructure where both the data storage and data processing happen

outside of the mobile device. Mobile cloud applications move the computing power and data storage away from mobile phones and into the cloud, bringing applications and mobile computing to not just smartphone users but a much broader range of mobile subscribers," according to http://www.mobilecloudcomputingforum.com/.

Mobile devices are producers, as well as consumers, of data stored on the cloud. They also take advantage of cloud resources for computationally intensive tasks. Mobile devices benefit from this symbiotic relationship; their reliability is improved as data and the applications are backed up on the cloud. Computer clouds extend the limited physical resources of smartphones, tablets, and laptops: (i) The processing power and the storage capacity—an ubiquitous applications of mobile devices is to capture still images or videos, upload to a computer cloud, and make them available via cloud services such as Flickr, Youtube, or Facebook; and (ii) Battery life—migrating mobile game components to a cloud can save 27% of the energy consumption for computer games and 45% for a chess game [124]. Clouds also extend the utility of mobile devices providing access to vast amounts of information stored on their servers. There are also major benefits due to the integration of mobile devices in the cloud ecosystem [438]:

- Mobile devices capture images and videos rich in content, rather than simple scalar data such as temperatures. Such information can be used to understand the extent of catastrophic events, such as earthquakes or forest fires.
- There is power in numbers—cloud sourcing amplifies the power of sensing and can be used for applications related to security, rapid service discovery, etc.
- Near-real-time data consistency is important in disaster relief scenarios. For example, the aftershocks of an earthquake often trigger major structural damage to buildings. Images and/or videos taken before and after the event help assess the extendit of the damage and identify buildings requiring immediate evacuation.
- Opportunistic information gathering. For example, anti-lock braking devices on cars transmit their GPS coordinates on each activation, enabling maintenance crews to identify the slick spots on roads.
- Computer vision algorithms running on clouds can use images captured by mobile devices to locate lost children in crowds, estimate crowd sizes, etc.

Applications of mobile cloud computing. Many mobile applications consist of a lightweight front-end component running on the mobile device and a back-end data-intensive and computationally intensive component running on a cloud. There are countless examples of such ubiquitous applications of mobile cloud computing in areas as diverse as health care, learning, electronic commerce, mobile gaming, mobile location services, and search.

An important application of mobile healthcare is patient record storage and retrieval and medical image sharing using computer clouds and mobile devices. Other applications in this area are [148]: health monitoring services where patients are monitored using wireless services; emergency management involving the coordination of emergency vehicles; and wearable systems for monitoring the vital signs of outpatients after surgery.

There are many education tools helping students learn and understand subjects as diverse as anatomy, computer science, or engineering, or the arts. Though some progress has been made, the potential of mobile learning is not fully exploited at this time. Classroom tools using AI algorithms to identify relevant questions posed by students during lectures and enabling instructors to interact with the students in classes with large enrollments are yet to be developed.

Mobile gaming has the potential of generating large earnings for cloud service providers. The engine requiring large computing resources can be offloaded to a cloud, while gamers only interact with the screen interface on their mobile devices. For example, to maximize energy savings, the Maui system [124] partitions the application codes at runtime based on the costs of network communication and the CPU power and the energy consumption of the mobile device. Browsers running on mobile devices are often used for keyword-, voice-, or tag-based searching of large debases available on computer clouds.

All applications of mobile cloud computing face challenges related to communication and computing. The low bandwidth, the service availability, and the network heterogeneity pose serious problems on the communication side. The security, the efficiency of data access, and the offloading overhead are top concerns on the computing side.

Mobile devices are exposed to a range of threats, including viruses, worms, Trojan horses, and ransomware. Such threats are amplified because mobile devices have limited power resources, and it is impractical to continually run virus-detection software. Moreover, once files located on the mobile device are infected, the infection propagates to the cloud when the files are automatically uploaded. At the same time the integration of GPS is a source of concern for privacy as the number of location-based services (LBS) increases. Cloud data security, integrity, and authentication, along with digital rights management are also sources of concern.

Enhancing the efficiency of data access requires a delicate balance between local and remote operations and the amount of data exchanged between the mobile device and the cloud. The number of I/O operations executed on the cloud in response to a mobile device request should be minimized to reduce the access time and the cost. The memory and storage capacity of the mobile device should be used to increase the speed of data access, reduce latency, and improve energy efficiency of mobile devices.

The future of mobile cloud computing. As the technology advances, mobile devices will be equipped with more advanced functional units, including high-resolution cameras, barometers, light sensors, etc. Augmented reality and mobile gaming are emerging as important mobile cloud computing applications. Augmented reality could become a reality with new mobile devices and fast access to computer clouds. Composition of real-time traffic maps from collective traffic data sensing, monitoring environmental pollution, and traffic and pollution management in smart cities are only a few of the potential future applications of mobile cloud computing. A recent survey [505] analyzes applications, the solutions to the challenges they pose, and the future solutions:

- Code and computation offloading—currently based on static partitioning and dynamic profiling is expected to be automated in the future.
- Task-oriented mobile services—currently provided by Mobile-Data-as-a-Service, Mobile-Computing-as-a-Service, Mobile-Multimedia-as-a-Service, and Location-based Services—are expected to be replaced by human-centric mobile services.
- Elasticity and scalability—components of the resource allocation and scheduling are expected to be validated by algorithms using valid traffic VM migration models.
- Cloud service pricing—currently based on auctions, and bidding is expected to be replaced by empirical validation and optimization algorithms.

Unquestionably, inclusion of mobile devices into the cloud ecosystem has countless benefits. It opens new avenues for learning, increased productivity, and entertainment, but can also have less desirable effects since it can overwhelm us with information. Herb Simon reflected on the effects of

information overload [448] "What information consumes is rather obvious: it consumes the attention of its recipients. Hence a wealth of information creates a poverty of attention and a need to allocate that attention efficiently among the overabundance of information sources that might consume it."

12.8 Energy efficiency of mobile computing

Mobile computing benefits from virtually infinite resources available on computer clouds provided that mobile devices have network access to these islands of plentiful computing and storage resources. All cloud users have to transfer data to/from the cloud, but there is a significant dissimilarity between the cloud users connected to computer clouds via landlines and using stationary devices and users of mobile devices connected via cellular or wireless local area networks (WLANs). The first are concerned with the time and the cost of transferring massive amounts of data, while for mobile cloud users the key issue is the energy consumption for communication.

Battery technology is lagging behind the needs of modern mobile devices. The amount of energy stored in a battery is growing by only about 5% annually. Increasing the battery size is not an option because devices tend to be increasingly lighter. Moreover, the power of small mobile devices without active cooling is limited to about three watts [367].

An analysis of the critical factors regarding mobile devices energy consumption reported in [347] is discussed in this section. As expected, the computing-to-communication ratio is the critical factor for balancing local processing and computation offloading. The analysis shows that not only the amount of transferred data, but also the traffic pattern is important. Indeed, sending a larger amount of data in a single burst consumes less energy than sending a sequence of small packets.

The analysis of energy used by mobile computing uses several parameters: E_{cloud} and E_{local}, the quantitative characterization of the energy for a computation on the cloud and locally, respectively; D - the amount of data to be transferred; C - the amount of computation expressed as CPU cycles. C_{eff} and D_{eff} are the efficiencies of computing and device specific data transfer, respectively. C_{eff} is measured in CPU cycles per Joule and represents the computations carried out with a given energy.

Dynamic voltage and frequency scaling affects only slightly the power and performance of the CPU during execution, thus C_{eff}. For example, the N810 Nokia processor requires 0.8 W and has $C_{eff} = 480$ when running at 400 MHz and needs only 0.3 W and has a $C_{eff} = 510$ at 165 MHz. This is also true for the more performant N900 Nokia processor; when running at 600 MHz, the power required and the C_{eff} are 0.9 W and 650 cycles/J, respectively, while at 250 MHz the same parameters are 0.4 W and 700 cycles/J, respectively.

D_{eff} is measured in bytes per Joule and represents the amount of data that can be transferred with a given energy. D_{eff} is affected by the traffic pattern. The time the network interface is active affects the energy consumption for communication. Typically, the power consumption for activation and deactivation of a cellular network interface is larger than that of a wireless interface. The larger the clock rate, the larger the power consumption and D_{eff}. For example, for N810, the power consumption and D_{eff} at 400 Mhz are 1.5 W and 390 KB/J, respectively, and decrease to 1.1 W and 310 KB/J, respectively, at 165 MHz.

It makes sense to offload a computation to a cloud if

$$E_{cloud} < E_{local}, \tag{12.5}$$

with

$$E_{cloud} = \frac{D}{D_{eff}} \quad \text{and} \quad E_{local} = \frac{C}{C_{eff}}. \tag{12.6}$$

The condition expressed by Eq. (12.5) becomes:

$$\frac{C}{D} > \frac{C_{eff}}{D_{eff}}. \tag{12.7}$$

According to [347], a rule of thumb is that "offloading computation is beneficial when the workload needs to perform more than 1 000 cycles of computation for each byte of data."

The CPU cycles per byte ratios measured on a system with an ARM Cortex-A8 core running at 720 Mz for several applications are: 330 for gzip ACII compression; 1 300 for x264 VBR encoding; 1 900 for CBR encoding; 2 100 for htm12text on wikipedia.org; and 5 900 for htm12text on en.wikipedia.org.

Several conclusions were drawn from the experiments reported in [347]:

- Energy consumption of a mobile device is affected by the end-to-end chain involved in each transaction thus, the server-side resource management is important.
- Higher performance typically contributes to better energy efficiency.
- Simple models able to guide design decisions for increasing energy efficiency should be developed.
- Automated decisions on whether a computation should be uploaded to a cloud to maximize energy efficiency of mobile devices should be built into mobile cloud computing middleware.
- Latencies associated with wireless communication are critical for interactive workloads.

It is unlikely we will see dramatic improvements of the energy storage capacity for mobile devices batteries in the future. The need for increasingly more sophisticated software for energy optimization should motivate the research in this challenging area because new mobile cloud computing data and computation-intensive applications are developed every day. At the same time, the other critical limitations of mobile cloud computing applications, communication speed, and effectiveness are improving because WLAN speed is steadily increasing, along with the efficiency of mobile device antenna.

12.9 Alternative mobile cloud computing models

In Section 12.7, we have seen that mobile device access to the large concentration of resources in cloud computing data centers is supported by wide-area networks (WANs), cellular networks, and wireless networks. In this section, we examine the limitations due to the communication latency and, to some extent, of the bandwidth and discuss alternative mobile cloud computing architectures.

Latency effects. Network security, energy efficiency, and network manageability, along with bandwidth, are the main sources of concern for networking companies and for networking research. Unfor-

tunately, virtually all methods to address these concerns have a negative side effect, *they increase the end-to-end communication latency*. Indeed, the techniques to increase energy efficiency discussed in Section 12.8 include reduction of the time the network interface is active, delaying data transmission until large data blocks can be sent, and turning on the transceivers of mobile devices for short periods of time to receive and to acknowledge data packets buffered at a base station.

The speed of the fastest wireless LAN (802.11n) and of the wireless Internet HSPDA (High-Speed Downlink Packet Access) technologies are 400 Mbps and 2 Mbps, respectively, and the corresponding transmission delays for a 4 Mbyte Jpeg image are 80 versus 16 msec, respectively. The range of Internet latencies is 30–300 msec. For example, the mean Berkeley to Trondheim-Norway and Berkeley to Canberra-Australia latencies are 197 and 174 msec, respectively, while Pittsburgh to Hong Kong and Pittsburgh to Seattle are 223 and 83.9 msec, respectively, as measured in 2008–2009 [437]. The current Internet latencies between selected pairs of points on the globe can be found at https://www.internetweathermap.com/map.

The latency effect differs from application to application. For example, the subjective effects of the latency, L, for a particular application, GNU Manipulation Program (GIMP) for Virtual Network Computing communication software, the graphical desktop sharing system that uses the Remote Frame Buffer protocol are: crisp when $L < 150$ msec; noticeable- to-annoying when $150 < L < 1\,000$ msec; annoying when $1 < L < 2$ sec; unacceptable when $2 < L < 5$ sec; and unusable when $L > 5$ sec [437].

Cloudlets. A possible solution to reduce end-to-end latency mimics the model supporting wireless communication where access points are scattered throughout campuses of large organizations and in cities. Micro data centers or "clouds in a box," called *cloudlets*, are placed in the proximity of regions with a high concentrations of mobile systems. A cloudlet could be a cluster of multicore processors with a fast interconnect and a high-bandwidth wireless LAN. The resources available on such cloudlets pale in comparison with the ones on a cloud; the cloudlets only assist the mobile devices for data and computationally intensive tasks since they are not permanent repositories of data. A mobile device could connect to a cloudlet and upload code to the cloudlet.

There are obvious benefits, as well as problems, with this solution proposed in [437]. The main benefit is the short end-to-end communication delay for mobile devices in the proximity of the cloudlet. On the other hand, the cost and the maintenance of cloudlets able to support a wide range of mobile users are of concern. It is safe to assume that hardware costs will decline as the processor technology evolves and that the processor bandwidth and reliability will increase. The software maintenance effort can be reduced if self-management techniques are developed. A range of business and technical challenges including cloudlet sizing must be addressed before this solution gets sufficient traction.

The solution for software organization proposed in [437] is to have a permanent cloudlet host software environment and a transient guest software environment. A transient customization configures the system for running an application, and, once the run is complete, a cleanup phase clears the way for running the next one. A VM encapsulates the transient guest environment.

The VM can be created on the mobile device, run locally until the VM needs additional resources, then stopped, its state saved in a file, and the file sent to a cloudlet in the vicinity of the mobile device where the VM is restarted. The problem with this straightforward solution is that the latency can be increased when the footprint of the VM migrated to the cloudlet is substantial.

Another solution is the *dynamic VM synthesis* in which the mobile device delivers to the cloudlet only a small overlay. This solution assumes that cloudlet already has the base VM used to derive the

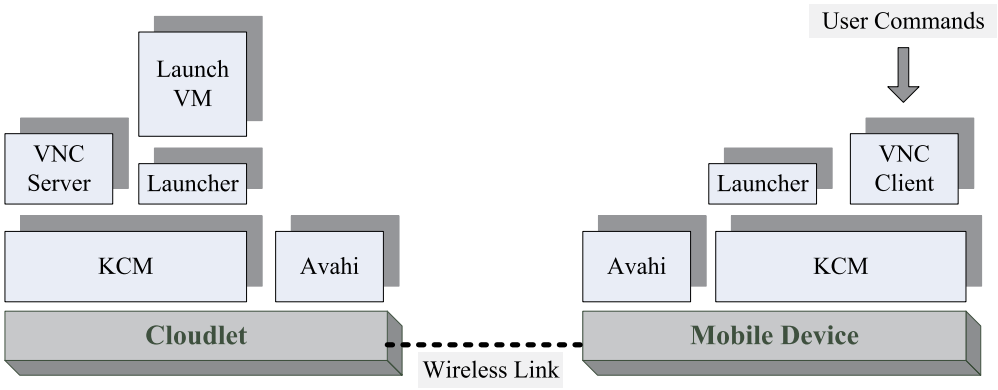

FIGURE 12.9

Kimberley organization. Kimberley Control Manager (KCM) runs on both the cloudlet and the mobile device. The two communicate over a wireless link using the Virtual Network Computing (VNC) communication software and Avahi [437].

small overlay; therefore the cloudlet environment can start execution immediately without the need to contact a cloud or other cloudlets. The assumption that a relatively small set of base VMs will suffice for a large range of applications may prove to be overly optimistic.

Kimberley—a proof of concept cloudlet system [437]. The cloudlet infrastructure consists of a desktop running Ubuntu Linux and the mobile device is a Nokia N810 tablet running Maemo 4.0 Linux. The cloudlet uses a hosted hypervisor for Linux called VirtualBox and a tool with three components, *baseVM, instal-script,* and *resume-script* to create the VM overlays for any OS compatible with the components of the tool.

First, the *baseVM* is launched, then the *instal-script* in the guest OS is executed, and finally the *resume-script* in the guest OS launches the application. The VM encapsulates the application, the so-called *launchVM* can be activated without the need to reboot, and finally this VM is compressed and encrypted.

Kimberley organization, depicted in Fig. 12.9, shows the software components running on the cloudlet and the mobile device. Communication through the wireless link is supported by Virtual Network Computing (VNC) communication software and Avahi,[8] a free zero-configuration[9] for automatic networking implementation supporting multicast DNS/DNS-SD service discovery.

KCM abstracts the service discovery, including browsing and publishing in Linux using Avahi. A secure TCP tunnel using SSL is established between KCM instances. After the establishment of

[8] Avahi is the Latin name of the woolly lemur primates indigenous to Madagascar.

[9] Zero-configuration networking (zeroconf) creates a TCP/IP computer network based on automatic assignment of numeric network addresses for networked devices, automatic distribution and resolution of computer hostnames, and automatic location of network services. It is used to interconnect computers or network peripherals.

the secure TCP tunnel, the authentication using the Simple Authentication and Security Layer (SASL) framework is carried out. Then, the KCM on the cloudlet fetches the VM overlay from the mobile device KCM, decrypts and decompresses the VM overlay, and applies the overlay to the base VM.

12.10 System availability at scale (R)

As the cloud user population grows and the scale of the data center infrastructure expands, the concerns regarding system availability are amplified. Very high system availability and correctness are critical for data streaming processing systems. Such systems are particularly vulnerable because the likelihood of missed events is very high when the resources required for event processing suddenly become unavailable.

When we think about availability, a 99% figure of merit seems quite high. There is another way to look at this figure: *A 99% availability translates to 22 hours of downtime per quarter, and this is not acceptable for mission-critical systems.* Google aims to achieve 99.9999–99.99999% availability for their data streaming systems [218]. How can this be achieved? An answer to this question is the topic of this section.

Multiple failure scenarios are possible. Individual servers may fail, all servers in one rack may be affected by a power failure, or a power failure may force the entire data center infrastructure to shut down, though such events are rare. The interconnection network may fail and partition the network, or public networks connecting a data center to the Internet or private networks connecting the data center with other data centers of the same CSP may fail.

Workloads can be migrated to functioning systems in case of partial failures affecting a subset of resources, and then users may experience slower communication and extended execution times in such cases. Often, it takes some time before the cause of a partial failure is found and corrective measures are taken. Also, events are missed when partial failures affect data streaming.

Shutdown of servers, networks, or the entire data center may be planned for hardware or software updates or maintenance. Such events are announced in advance, and in such cases, the shutdown is graceful and without data loss or other unpleasant consequences for the cloud users. The unplanned shutdown are the ones the CSPs are concerned about. The most consequential are the data center-level failures.

While current systems manage quite well the failure of individual servers, data center-level failures, though seldom occurring, have catastrophic consequences especially for *single-hommed* software systems, the ones running *only* at the site affected by the failure. Less affected are the *failover-based* software systems. Such systems only run at one site, but checkpoints are created periodically and sent to backup data centers. When the primary data center fails, the last checkpoints of the critical systems are used to restart processing at the backup site. Data streaming is affected in this case as events are likely to be missed.

Checkpointing can be done asynchronously or synchronously. In the first case, all events are processed exactly once during the restart if the events in progress are drained before checkpointing, and only when the shutdown was planned. Synchronous checkpointing can, in principle, capture all events even in case of unplanned shutdowns, but their processing is considerably more complex. For planned

shutdowns, they require each pipeline to generate a checkpoint and block until the checkpoint is replicated.

Virtually all CSPs have multiple data centers distributed across several regions, and a basic assumption for handling data center-level failures is that the probability of a simultaneous failure of data centers in multiple regions is extremely low. Critical software systems are *multi-homed,* the workload is dynamically distributed among these centers, and, when one of them fails, its share of the workload is reassigned to the ones still running.

The design of multi-homed systems is not without its own challenges. First, the global state of the system must be available to all sites. The size of the metadata reflecting the global state should be minimized. Communication delays of tens of milliseconds and limited global communication bandwidth do not make determination of the global state an easy task, even when a Paxos-bases commit is used to update the global state.

Second, a data streaming pipeline is implemented as a network with multiple processing nodes. Checkpointing has to capture the state of all nodes, as well as metadata such as the state of the input and output queues of pending and completed node work. Clustering groups nodes together and recording only the global state of the cluster helps. Clustering increases the checkpointing efficiency and shrinks the checkpoint size.

Handling the same input events at several sites can be optimized so that the checkpoint includes only one copy of the event. This is actually done for the events recorded by logs in the system discussed next. Such primary events may require database lookup, and if the database data does not change, then the state of the primary event should not record the information from the database.

Lastly, guaranteeing exactly once semantics for multihomed systems is nontrivial because pipelines may fail while producing output; multiple sites may process the same events concurrently. When the global state is stored in a shared memory, updating global state can be atomic. Idempotence can help achieving the desired semantics. If multiple sites update the same record, this guarantees the desired semantics. If idempotence is not achievable, a two-phase commit can be used to write the results produced by the streaming system to the output.

Data streaming services collect data generated by user interactions and expect consistency. Data streaming consistency means that, if a state at time t includes an event e, then any future state at time $t + \tau$ must also include event e and an observer should see consistent outcomes if more than one are generated.

The concern for data center-level failures forced Google to run multiple copies of critical systems at multiple data centers. Several large-scale Google systems run hot in multiple data centers, including the F1 database [447], discussed in Section 12.2, and the Photon system [24], discussed in Section 12.6.

An infrastructure supporting ads management at Google is discussed in [218]. The strategy for supporting availability and consistency for the streaming system is to log the events caused by user interactions in multiple data centers. Local logs are then collected by a *log collection service* and are sent to several locations designated as *log data centers*. Consistency of these logs is critical and requires *exactly once* event processing semantics.

Once a system is developed for a single site, it is very hard to adapt to multisite operations, especially when consistency among sites is a strong requirement. An *ab initio* design as a multihome is the optimal solution for mission critical systems. Running a system at multiple sites transfers the burden to solve the very hard problem caused by data center-level failures to the infrastructure. This burden falls to

the users in traditional systems designed to run at one site only. Indeed, the users have to take periodic checkpoints and transfer them to backup sites.

Multihoming is challenging for system designers and adds to the resource costs. Additional resources are needed to process the workload normally processed by the failing data center and to catch up after delays. The spare capacity has to be reserved ahead of time and should be ready to accept the additional workload.

12.11 Scale and latency (R)

The "scale" of the cloud has altered our ideas on how to compute efficiently and has generated new challenges in virtually all areas of computer science and computer engineering, from computer architecture to security, via software engineering and machine learning. A fair number of models and algorithms for parallel processing have been adapted to the cloud environment. Frameworks for Big Data processing, resource management for large-scale systems, scheduling algorithms for latency-critical workloads discussed in Chapters 11, 4, 9, 5, and 12 offer only a limited window into the new computing landscape. Several critically important ideas for the future of cloud computing are discussed in this section.

Latency, the time elapsed from the instant when an action is initiated to its completion, has always been an important measure of quality of service for interactive applications, but its relevance and impact has been substantially amplified in the age of online searches and web-based electronic commerce. There are quite a few latency-critical applications of cloud computing now, and their number is likely to explode when a large number of cyber–physical systems of the IoT will invade our working and living space.

The latency has three major components: communication time, waiting time, and service time. Only the latter two are discussed in this section because the communication speed is not controlled by the cloud service provider, and in practice it has little impact on latency. A heavy-tail distribution of latency due to the latter two components is undesirable; it affects the user's experience and, ultimately, the ability of the service provider to compete.

Heavy-tail distributions. A random variable X with the cumulative distribution function $F_X(x)$ is said to have a heavy right tail if

$$\lim_{x\to\infty} e^{\lambda x} Pr\,[X > x] = \infty, \quad \forall\, \lambda > 0. \tag{12.8}$$

This definition can also be expressed in terms of the tail-distribution function

$$\bar{F}_X(x) \equiv Pr\,[X > x] \tag{12.9}$$

and implies that $MF(t)$, the moment generating function of $F_X(x)$, is infinite for $t > 0$. The Lévy distribution is an example of a heavy-tail distribution. Its probability density function (PDF) over the domain $x \geq \mu$ and cumulative distribution function (CDF) are, respectively,

$$f_X(x;\mu,c) = \sqrt{\frac{c}{2\pi}} \times \frac{e^{-\frac{c}{2(x-\mu)}}}{(x-\mu)^{3/2}} \quad \text{and} \quad F_X(x;\mu,c) = erfc\left(\sqrt{\frac{c}{2(x-\mu)}}\right), \tag{12.10}$$

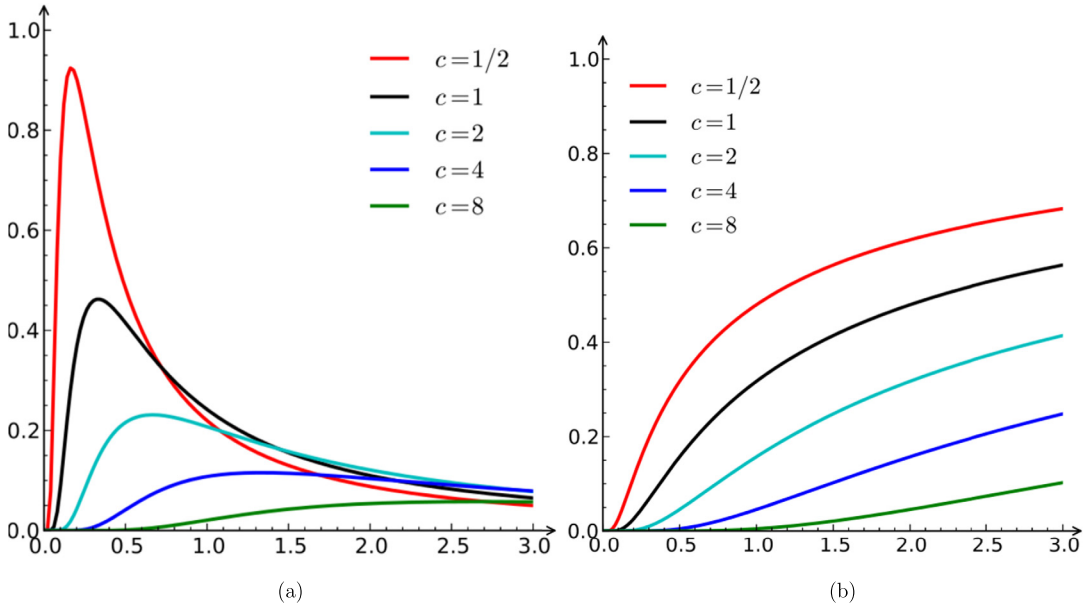

FIGURE 12.10

(a) The probability density function of the unshifted Lévy distribution, $\mu = 0$; (b) The cumulative distribution function.

with μ being the location parameter, c being the scale parameter, and $erfc(y)$ being the complementary error function. Fig. 12.10 displays the PDF and the CDF of the Lévy distribution. In probability theory and statistics, a scale parameter is a special kind of numerical parameter of a parametric family of probability distributions. The larger the scale parameter, the more spread out is the distribution function.

Query processing at Google. The analysis of query processing at Google gives us some insight into the reasons why latency has a heavy-tail distribution and how this can be managed. A query fans out into thousands of requests to a large number of servers where cached data resides. Some of these servers contain images, others videos, web data, blogs, books, news, and many other bits of data that could possibly answer the query.

The individual-leaf-request finishing times measured from the root node to the leafs of the query fan-out tree show the effect of the heavy-tail distribution [130]. As the limit x in the latency cumulative distribution function increases from 50 to 95 and then to 99 percentile, the latency increases:

- When a single randomly selected leaf node of the tree is observed the latency increases from 1 to 5, and then to 10 milliseconds, respectively.
- When we request that 95% of all leafs finish execution the latency increases from 12 to 32, and then to 70 milliseconds, respectively.
- When we request that all nodes finish execution the latency increases from 40 to 87, and then to 140 milliseconds, respectively.

These measurements show that the latency increases with the number of leafs of the fan-out tree; it also increases as we wait for more leaves to finish. The extent of the latency tail is significant, the difference between 95 and 99 percentile is dramatic, it doubles or nearly doubles for the three cases examined, 10 versus 5, 70 versus 32, and 140 versus 87, mimicking the law of diminishing returns.

Google is able to address the problems posed by the heavy-tail latency distribution [130]: "The system updates the results of the query interactively as the user types predicting the most likely query based on the prefix typed so far, performing the search and showing the results within a few tens of milliseconds." How this is done is discussed in this section.

A brief analysis of the factors affecting the variability of the query response time helps understanding how this variability can be limited. Resource sharing is unavoidable and contention for system resources translates into longer response time for the losers. There are actors working behind the scenes carrying out system management and maintenance functions. For example, daemons are themselves users of resources and competitors of production workloads; they can occasionally cause an increase of the response time. Data migration, data replication, load balancing, garbage collection, and log compaction, are all activities competing for system resources.

Multiple layers of queuing in the hierarchical infrastructure increase the waiting time and this can be addressed by priority queuing and techniques discussed next. Optimization of resource utilization and energy saving mechanisms contribute to latency. For example, servers are switched to a power-saving mode when the workload dips. The latency can increase in this case because it takes some time before servers wakeup and are able to absorb a spike in demand. Dynamic frequency and voltage scaling, a mechanism to adapt the power consumption to the workload intensity, could also contribute to latency variability.

Several techniques can reduce the effects of component-level variability on the heavy-tail query latency distribution. These techniques are:

1. Define different service classes and use priority queuing to minimize waiting for latency-critical workloads.
2. Keep the server queues short allowing requests from latency-critical workloads to benefit from their high priority. For example, Google storage servers keep few operations outstanding, instead of maintaining a queue. Thus, high-priority requests are not delayed, while earlier requests from low-priority tasks are served.
3. Reduce head-of-line blocking. When a long-running task cannot be preempted, other tasks waiting for the same resource are blocked, a phenomenon called head-of-line blocking.[10] The solution is to split a long-running task into a number of shorter tasks and/or to use time-slicing, to allocate the resource to task for brief periods of time. For example, Google's Web search system uses time-slicing to prevent computationally expensive queries to add to the latency.
4. Limit the disruption caused by background activities. Some of the behind-the-scene activities such as garbage collection or log compaction require multiple resources. Their continuous running in the background could lead to increased latency of many high-priority queries over extended periods of time. It is more beneficial to allow such background activities to proceed in synchronous bursts of

[10] For an intuitive scenario of head-of-line blocking, imagine a slow-moving truck on a one-lane, winding mountain road. It is very likely that a large number of cars will be stuck behind the truck.

Table 12.3 Read latency in ms with requests tied 1 ms [130] for a latency-critical application. (Left) Mostly idle system. (Right) System running a background job in addition to the latency-critical application.

Limit (%)	No hedge	Hedge-tied	No hedge	Hedge-tied
50	19	16	24	19
90	38	29	54	38
99	67	42	108	67
99.9	98	61	159	108

concurrent utilization of several servers, thus affecting only the latency-critical tasks over the time of the burst.

A very important realization is that *the heavy-tail distribution of latency cannot be eliminated; the only alternative is to develop tail-tolerant techniques for masking long latencies*. Two types of tail-tolerant techniques are used at Google: (i) *within request, short-term*, acting in milliseconds; and (ii) *cross-request, long-term*, acting at a scale of seconds.

The former technique works well for replicated data, e.g., for distributed files systems with data striped and replicated across multiple storage servers and for read-only datasets. For example, the spelling-correction service benefits from this mechanisms because the model is updated once a day and handles a very high rate of requests, in the hundreds of thousands per second. Cross-request techniques aim to increase parallelism and prevent imbalance either by creating micro-partitions, by replicating the items likely to cause imbalance, or by latency-induced probation.

Hedged and *tied* requests are short-term tail-tolerant techniques. In both cases, the client issues multiple replicas of the request to increase the chance of a prompt reply. Hedged requests are separated by a short time interval; the client issues the requests, accepts the first answer, and then notifies the other servers canceling the request. The client should wait a fraction, say 90% to 95% of the average response time, before sending the replica of the request to limit the additional workload and to avoid duplication.

Requests are tied when each replica includes the address of other servers wishing to reply to the request. In this case, the servers receiving the request communicate with one another; the first server able to start processing the request sends a canceling message to the other servers at the time it starts execution. If the input queue of all recipients of the request is empty, then all can start processing at about the same time and canceling messages criss-cross the network that are unable to prevent work duplication. This undesirable situation is prevented if the client waits a time equal to twice the average message delay before sending a replica of the request.

Hedged requests are very effective. For example, a Google benchmark reads the key value of 1 000 key-value pairs stored in a large BigTable distributed over 100 servers. Hedged requests sent 10 ms after the original ones reduced the 99 percentile latency dramatically, from 1 700 to 74 ms, while sending only 2% more requests. Another Google benchmark shows the effect of the queries tied one millisecond to an idle cluster and the other query on a cluster running a batch job in the background. The BigTable data is not cached, and each file chunk has three replicas on different storage servers. Table 12.3 shows read latencies with no hedge and with hedged tied requests.

The results in Table 12.3 show first that hedged and tied requests work well not only when the system is lightly loaded but also at higher system loads. This also implies that the overhead of this mechanisms is low and adds only a minute workload to the system. These results also show the extent of the heavier tail; the difference in latency between 99% and 99.9% is significant for both the lightly and the heavily loaded system.

Micro-partitions, replication of items likely to cause imbalance, and latency-induced probation are long-term tail-tolerant techniques. Perfect load balancing in a large-scale system is practically unachievable for many reasons. These reasons include the difference in server performance, the dynamic nature of the workloads, and the impossibility of having an accurate picture of the global system state. A fairly intuitive approach is a fine-grained partitioning of resources on every server, thus the name micro-partitions.

These "virtual servers" are more nimble and able to process fine-grained units of work more quickly. Error recovery is also less painful since less work is lost in case of errors. For a long time, immovable data replication has been a method of choice for improved performance, and it is also widely used at Google. Intermediate servers in a hierarchically organized system can identify slow-responding servers and avoid sending them latency-critical tasks, while continuing to monitor their behavior.

12.12 Edge computing and Markov decision processes (R)

Edge computing is a distributed computing framework that brings enterprise applications closer to data sources, such as IoT devices or local edge servers. Edge computing harnesses growing in-device computing capability to provide deep insights and predictive analysis in near-real time. Edge computing and mobile edge computing on 5G networks enables more comprehensive data analysis, faster response times, and improved customer experiences.

Edge computing is effective as data is processed and analyzed closer to the point where it is created. Data does not traverse a network to be processed and latency is significantly reduced. Proximity to data source has obvious benefits: improved response times and lower bandwidth availability. It is estimated that, in 2025, 75% of enterprise data will be processed at the edge, instead of only 10% today.

The *follow-me cloud* [468] and the *mobile edge-cloud* are variations on the theme of cloudlets discussed in Section 12.9 that were conceived to reduce the end-to-end latency for cloud access. The mobile edge-cloud concept allows mobile devices to carry out computationally intensive tasks on stationary servers located in the small data centers distributed across the network and connected directly to base stations at the edge of the network.

The new twist is to support *dynamic service placement*, in other words, to allow computations initiated by a mobile device in motion to migrate from one mobile edge-cloud server to another following the movement of the mobile device [483,506]. Optimal service migration policies pose very challenging problems due to the uncertainty of the mobile device movements and the likely nonlinearity of the migration and communication costs. One method to address these challenges is to formulate the migration problem in terms of the Markov decision process discussed next.

Markov Decision Processes. Markov decision processes are *discrete-time* stochastic control processes used for a variety of optimization problems where the outcome is partially random and partially under the control of the decision maker. Markov decision processes extend Markov chains with choice and motivations; *actions* allow choices, and *rewards* provide motivation for actions.

The state of the process in slot t is s_t, and the decision maker may choose any action $a_t \in \mathcal{A}(s_t)$ available in that state. As a result of action a_t, the system moves to a new state s' and provides the reward $\mathcal{R}_{a_t}(s_t, s')$. *The next state s' depends only on the current state s_t and the action taken.* The probability that the system moves to state s' is given by the transition function $p_{a_t}(s_t, s')$.

A Markov decision process is a 5-tuple: $(\mathcal{S}, \mathcal{A}, \mathcal{P}(\cdot,\cdot), \mathcal{R}(\cdot,\cdot), \gamma)$ with

- $\mathcal{S}$—a finite set of system states;
- $\mathcal{A}$—a finite set of actions; $\mathcal{A}_{s_t} \in \mathcal{A}$ is the finite set of actions available in state $s_t \in \mathcal{S}$ in time slot t.
- $\mathcal{P}$—the set of transition probabilities; $p_{a_t}(s_t, s') = Pr(s(t+1) = s' | s_t, a_t)$—the probability that action a_t in state s_t in time slot t will lead to state s' in time slot $t+1$.
- $\mathcal{R}_{a_t}(s_t, s')$—the immediate reward after the transition from state s_t to state s'.
- $\gamma \in [0, 1]$—a discount factor representing the difference in importance between present and future rewards.

The goal is to optimize a policy π maximizing a cumulative function of the random rewards, e.g., the expected discounted sum of rewards over an infinite time horizon is

$$\sum_{t=0}^{\infty} \gamma^t \mathcal{R}_{a_t}(s_t, s_{t+1}). \tag{12.11}$$

To calculate the optimal policy given $\mathcal{P}$ and $\mathcal{R}$, the state transitions and, respectively, the rewards, one needs two arrays $\mathcal{V}(s)$ and $\pi(s)$ indexed by state, the value, and policies, respectively. The first array contains values and the second actions:

$$\pi(s) = \arg\max_{\alpha} \left\{ \sum_{s'} \mathcal{P}_{\alpha}(s, s') \left(\mathcal{R}_{\alpha}(s, s') + \gamma \mathcal{V}(s') \right) \right\} \tag{12.12}$$

$$\mathcal{V}(s) = \sum_{s'} \mathcal{P}_{\pi(s)}(s, s') \left(\mathcal{R}_{\pi(s)}(s, s') + \gamma \mathcal{V}(s') \right). \tag{12.13}$$

In the value induction proposed by Bellman, the calculation of $\pi(s)$ is substituted in the calculation of $\mathcal{V}(s)$:

$$\mathcal{V}_{i+1}(s) = \max_{\alpha} \left\{ \sum_{s'} \mathcal{P}_{\alpha}(s, s') \left(\mathcal{R}_{\alpha}(s, s') + \gamma V_i(s') \right) \right\}. \tag{12.14}$$

A Markov decision process can be solved by linear programming or by dynamic programming.

Migration decisions and cost in mobile cloud edge. The solution proposed in [483,506] assumes that:

1. The user's location in each slot is the same and changes from one slot to the next are according to the Markovian model discussed in this section; see Fig. 12.11.
2. The set of all possible locations, $\mathcal{L}$, can be represented as a 2D vector, and the distance between two locations $l_1, l_2 \in \mathcal{L}$ is given by $|| l_1 - l_2 ||$.
3. The migration time is very small and will be neglected.

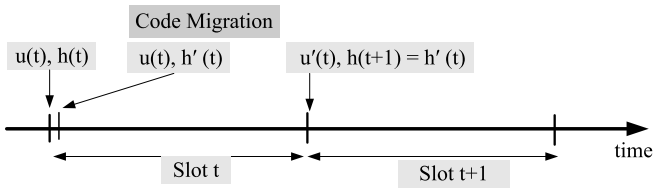

FIGURE 12.11

Time slots for the edge cloud. At the very beginning of slot t the user and service locations are $u(t)$ and $h(t)$, respectively. The service migrates to $h'(t)$, as soon as slot t starts and during the slot t the system operates with $u(t)$ and $h'(t)$. At the beginning of slot $t+1$, the service location is $h(t+1) = h'(t)$.

The following notations are used in [506]:

- $u(t)$—users's location in slot t.
- $h(t)$—service location in slot t.
- $d(t)$—the distance between the user and the service; $d(t) = || u(t) - h(t) ||$ in slot t.
- N—the maximum allowed distance between the user and the service, $N = \max d(t)$.
- $s(t) = (u(t), h(t))$—the initial system state at the beginning of slot t. The initial state is $s(0) = s_0$.
- $c_m(x)$—the migration cost is a nondecreasing function of x, the distance between the two edge cloud servers, $x = || h(t) - h'(t) ||$.
- $c_d(x)$—the transmission cost is a nondecreasing function of x, the distance between the edge cloud server and the user x, with $x = || u(t) - h'(t) ||$. Initially, $c_d(0) = 0$.
- π—the policy used for control decision based on $s(t)$.
- $a_\pi(s(t))$—the control action taken under policy π when the system is in state $s(t)$.
- $\mathcal{C}_{a_\pi}$—the sum of migration and transmission costs incurred by a control $a_\pi(s(t))$ in slot t. $\mathcal{C}_{a_\pi} = c_m(|| h(t) - h'(t) ||) + c_d(|| u(t) - h'(t) ||)$.
- γ—discount factor, $0 < \gamma < 1$.
- $\mathcal{V}_\pi$—expected discount under policy π.

The Markov decision process controller makes a decision at the beginning of each slot. The decision could be:

1. Do not migrate the service; then the cost is $c_m(x) = 0$.
2. Migrate the service from location $h(t)$ to location $h'(t) \in \mathcal{L}$; $c_m(x) > 0$.

Given a policy π, its long-term expected discounted sum cost is

$$\mathcal{V}_\pi(s_0) = \lim_{t\to\infty} \mathbb{E}\left\{\sum_{\tau=0}^{t} \gamma^\tau \mathcal{C}_{a_\pi}(s(\tau)) \mid s(0) = s_0\right\}. \tag{12.15}$$

An optimal control policy is one that minimizes $\mathcal{V}_\pi(s_0)$ starting from any initial state

$$\mathcal{V}^*(s_0) = \min_\pi \mathcal{V}_\pi(s_0), \quad \forall s_0. \tag{12.16}$$

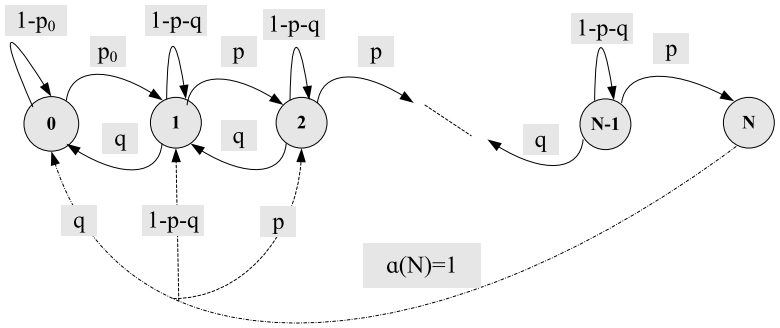

FIGURE 12.12

Markov decision process state transitions assuming a 1-D mobility model for the edge cloud. The mobile device moves one step to the left or right with probability r_1 and stays in the same location with probability $1-2r_1$, thus $p=q=r_1$ and $p_0=2r_1$. Migration occurs in slot t if and only if $d(t)\geq N$. The actions in slot t with distance $d(t)$ are $\alpha(d(t))=\alpha(N)$ for $d(t)>N$. This implies that we need only to study the states where $d(t)\in[0,N]$. After an action $\alpha(N)$, the system moves to states 0, 1, and 2 with probabilities q, $1-p-q$, and p, respectively.

A stationary policy is given by Bellman's equation written as

$$\mathcal{V}^*(s_0)=\min_{\alpha}\left\{\mathcal{C}_\alpha(s_0)+\gamma\sum_{s_1\in\mathcal{L}\times\mathcal{L}}\mathcal{P}_{\alpha(s_0,s_1}\mathcal{V}^*(s_1)\right\}, \tag{12.17}$$

with $\mathcal{P}_{\alpha(s_0,s_1)}$ being the transition probability from state $s'(0)=s_0=\alpha(s_0)$ to $s(1)=s_1$. The intermediate state $s'(t)$ has no randomness when $s(t)$ and $\alpha(\cdot)$ are given.

A proposition reflecting the intuition that it is not optimal to migrate the service to a location that is farther away from the current location of the mobile device has the following intuitive implication, which simplifies the search for optimal policy [506]:

Proposition. *If $c_m(x)$ and $c_d(x)$ are constants and*

$$c_m(0)<c_m(x)\quad and\quad c_d(0)<c_d(x)\quad for\quad x>0, \tag{12.18}$$

then the current user location is not optimal.

Fig. 12.12 shows the system transition model when the transition probabilities are p_0, p, and q, assuming a uniform 1-D mobility model. In this model $u(t)$, $h(t)$, and $h'(t)$ are scalars. h' is the new service location chosen such that:

$$||\,h(t)-h'(t)\,||=|\,d(t)-d'(t)\,|\quad\text{and}\quad||\,u(t)-h'(t)\,||=d'(t). \tag{12.19}$$

The migration occurs along the shortest path connecting $u(t)$, $h(t)$, and $h'(t)$.

An operating procedure is discussed in [506], and workload scheduling for edge clouds is presented in [483].

12.13 Bootstrapping techniques for data analytics (R)

The size of the data sets used for data analytics, as well as the complexity and the diversity of queries posed by impatient users, often without training in statistics, continually increase. In many instances, e.g., in case of exploratory queries, it is desirable to provide good enough, yet prompt answers, rather than perfect answers after a long delay.

This is only possible by limiting the search to a subset of data, but in such instances, the user expects an estimation of answer's quality along with the answer. The quality of the answer is context dependent, and a general solution is far from trivial. Bootstrapping techniques discussed in this section can be applied to a broad range of applications for estimating the quality of such approximations. Given a set F and a random variable U, the bootstrapping methods are based on the *bootstrap substitution principle*

> To determine the probability distribution of $U \equiv u(Y, F)$ with $Y = \{Y_1, Y_2, \ldots, Y_n\}$ random samples of F are taken and then F is replaced by a fitted model $\hat{F}$.

Thus, we make the approximation

$$Pr\{u(Y, F) \leq u \mid F\} \approx Pr\{u(Y^*, \hat{F}) \leq u\hat{F}\}. \tag{12.20}$$

The superscript $*$ distinguishes random variables and related quantities sampled from a probability model that has been fitted to the data. Sometimes, $u(Y, F) = T - \theta$ with T as the estimator of the parameter $\theta \equiv t(F)$, and more sophisticated cases involve transformations of T. Often, $T = t(\tilde{F})$, and $\tilde{F}$ is an empirical distribution function of the data values.

While bootstrapping can produce reasonably accurate results, there are also instances when the results are unreliable. Some of such instances discussed in depth in [86] are:

1. Inconsistency of the bootstrap method when the model, the statistic, and the resampling fail to approximate the required properties, regardless of the sample size.
2. Incorrect resampling model in case of nonhomogeneous data when the random variation of data is incorrectly modeled.
3. Nonlinearity of statistic T as the good properties of the bootstrap are associated with an accurate linear approximation $T \approx \theta + n^{-1} \sum_i l(T_i)$, with $l(y)$ the influence function of $t(\cdot)$ at (y, F).

A key idea is to construct a proxy to ground truth for a sample smaller than that of the complete observed data set and compare the bootstrap's results with this proxy. The bootstrapping techniques discussed in this section are based on [275], which states "existing diagnostic methods target only specific bootstrap failure modes, are often brittle or difficult to apply, and generally lack substantive empirical evaluations ...this paper presents a general bootstrap performance diagnostic which does not target any particular bootstrap failure mode but rather directly and automatically determines whether or not the bootstrap is performing satisfactorily."

The bootstrap method. Let P be an unknown distribution, $\theta(P)$ be some parameter of P, and $\mathcal{D}$ the set of n independent identically distributed (i.i.d.) sampled data points $\mathcal{D} = \{X_1, X_2, \ldots, X_n\}$ from P.

Let $\mathbb{P} = n^{-1} \sum_{i=1}^{n} \delta_{X_i}$ be the empirical distribution of data. We wish to construct the estimate $\hat{\theta}(\mathcal{D})$ of $\theta(P)$ and then create $\xi(P, n)$, an assessment of the quality of $\hat{\theta}(\mathcal{D})$, consisting of a summary of the distribution Q_n of some quantity $u(\mathcal{D}, P)$.

Both P and Q_n are unknown; thus the estimate $\xi(P, n)$, called the *ground truth* in this discussion, cannot be computed directly; it can be approximated by $\xi(\mathbb{P}_n, n)$ using Monte Carlo procedure. The following steps are carried out repeatedly:

1. Form simulated data sets $\mathcal{D}^*$ of size n consisting of i.i.d. sampled points from $\mathbb{P}_n$;
2. Compute $u(\mathcal{D}^*, \mathbb{P}_n)$ for the simulated data set $\mathcal{D}^*$;
3. Form the empirical distribution $\mathbb{Q}_n$ of the computed values of u;
4. Return the desired summary of this distribution.

The final bootstrap output will be a real-valued $\xi(\mathbb{Q}_n, n)$. The assessment $\xi(P, n)$ could compute:

1. The bias, the expectation of

$$u(\mathcal{D}, P) = \hat{\theta}(\mathcal{D}) - \theta(P). \tag{12.21}$$

2. A confidence interval based on the distribution of

$$u(\mathcal{D}, P) = n^{1/2}[\hat{\theta}(\mathcal{D}) - \theta(P)]. \tag{12.22}$$

3. Simply

$$u(\mathcal{D}, P) = \hat{\theta}(\mathcal{D}). \tag{12.23}$$

Given the estimator, the data generating distribution P, and n, the size of the data set, we want to determine if the output of the bootstrap procedure is *sufficiently close* to the ground truth. The formulation *sufficiently close* avoids a precise expression of accuracy, giving the procedure a degree of generality and allowing its use for a range of applications with own accuracy requirements.

In practice, we can observe only one set with n data points rather than many independent sets of size n. The solution is to randomly sample $p \in \mathbb{N}$ disjoint subsets of the dataset $\mathcal{D}$, each of size $b \leq \lfloor n/p \rfloor$. Then, to approximate the distribution of Q_b, we use the set of values of u calculated for each subset. This distribution yields an approximation of the ground truth, $\xi(P, b)$ for data sets of size b. Then, to determine if the bootstrap performs as expected on sample size b, we run the bootstrap on each of the p subsets and compare the p bootstrap outputs to the ground truth.

Carrying out this procedure for a single sample size b is insufficient, the bootstrap performance may be acceptable for small sample size, but it may get worse as the sample size increases or, conversely, be mediocre for small sizes, but improve as the sample size increases. Thus, it is necessary to compare the distribution of bootstrap outputs for a range of sample sizes, $b_1, b_2, \ldots, b_k$, $b_k \leq \lfloor n/p \rfloor$. If the distribution of the bootstrap outputs converges monotonically for all smaller sample sizes $b_1, b_2, \ldots, b_k$, we conclude that the bootstrap is performing satisfactorily for size n.

Convergence criteria are based on the relative deviation of the absolute value and the size of the standard deviation of the bootstrap output from the ground truth. The pseudocode of the Bootstrap Performance Diagnostic (BPD) algorithm illustrates these steps.

BPD Algorithm—Bootstrap Performance Diagnostic [275]

Input: $\mathcal{D} = \{X_1, \ldots, X_n\}$: observed data
- u: quantity whose distribution is summarized to yield estimator quality assessments
- ξ: assessment of estimator quality
- p: number of disjoint subsamples used to compute ground truth approximations
- $b_1, \ldots, b_k$: increasing sequence of subsample sizes for which ground truth approximations
- are computed with $b_k \leq \lfloor n/p \rfloor$ (e.g., $b_i = \lfloor n/(p2^{k-i}) \rfloor$ with $k = 3$.
- $c_1 \geq 0$: tolerance for decreases in absolute relative deviation of mean bootstrap output
- $c_2 \geq 0$: tolerance for decreases in relative standard deviation of bootstrap output
- $c_3 \geq 0, \alpha \in [0, 1]$: desired probability that bootstrap output at sample size n has absolute
- relative deviation from ground truth less than or equal to c3 (e.g., c3 = 0:5; = 0:95)

Output: *true* if the bootstrap is deemed to be performing satisfactorily, and *false* otherwise

$\mathbb{P}_n \rightarrow n^{-1} \sum_{i=1}^{n} \delta_{X_i}$

for $i \leftarrow 1$ **to** k **do**
- $\mathcal{D}_{i1}, \ldots, \mathcal{D}_{ip} \rightarrow$ random disjoint subsets of $\mathcal{D}$, each containing b_i data points
- **for** $j \leftarrow 1$ **to** p **do**
- $u_{ij} \leftarrow u(\mathcal{D}, \mathbb{P}_n)$
- $\xi^*_{ij} \leftarrow bootstrap(\xi, u, b_i, \mathcal{D}_{ij})$
- **end**
- // Compute ground truth approximation for sample size b_i
- $\mathbb{Q}_{b_i} \leftarrow \sum_{j=1}^{p} \delta_{u_{ij}}$
- $\tilde{\xi}_i \leftarrow \xi(\mathbb{Q}_{b_i}, b_i)$
- // Compute absolute relative deviation of mean and relative standard deviation
- // of bootstrap outputs for sample size b_i
- $\Delta_i \leftarrow | \frac{mean\{\tilde{\xi}^*_{i1}, \ldots, \tilde{\xi}^*_{ip}\} - \tilde{\xi}_i}{\tilde{\xi}_i} |$ $\sigma_i \leftarrow | \frac{stddev\{\tilde{\xi}^*_{i1}, \ldots, \tilde{\xi}^*_{ip}\} - \tilde{\xi}_i}{\tilde{\xi}_i} |$

end

return true *if all of the following hold, and* false *otherwise*

$$\Delta_{i+1} < \Delta_i \quad \text{OR} \quad \Delta_{i+1} \leq c_1, \forall i = 1, \ldots, k \tag{12.24}$$

$$\sigma_{i+1} < \sigma_i \quad \text{OR} \quad \sigma_{i+1} \leq c_2, \forall i = 1, \ldots, k \tag{12.25}$$

$$\frac{\#\left(j \in \{1, \ldots, p\} : \ | \frac{\tilde{\xi}^*_{kj} - \tilde{\xi}_k}{\tilde{\xi}_k} | \leq c_3\right)}{p} \geq \alpha \tag{12.26}$$

This algorithm generates a confidence interval with coverage $\alpha \in [0, 1]$.[11] A false positive, deciding that an approximation is satisfactory when it is not, is less desirable than a false negative, rejecting

[11] Confidence intervals offer a guarantee of the quality of a result. A procedure is said to generate confidence intervals with a specified coverage $\alpha \in [0, 1]$ if, on a proportion exactly α of the set of experiments, the procedure generates an interval that includes the answer. For example, a 95% confidence interval $[a, b]$ means that in 95% of the experiments, the result will be in $[a, b]$.

an approximation when in fact it is a satisfactory one. Eq. (12.26) reflects this conservative approach. The absolute value of the deviation of a quantity γ from γ_0 is defined as $|\gamma - \gamma_0|/|\gamma_0|$, and an approximation is satisfactory if the output of the bootstrap run on a data set of size n has an absolute value of the relative deviation from the ground truth of at most c_3 with a probability $\alpha \in [0, 1]$.

Choosing the sample sizes close together or choosing c_1 or c_2 too small will cause a larger number of false negatives. It is recommended to use an exponential distribution of sample sizes to ensure a meaningful comparisons of bootstrap performance for consecutive values b_i, b_{i+1} in the set $\{b_1, \ldots, b_i, b_{i+1}, \ldots, b_k\}$.

The process discussed in this section requires a substantial amount of data, but this does not seem to be a problem in the age of Big Data. For example, according to [275] when $p = 100$ and $b_k = 1\,000$ then $n \geq 10^{15}$ and if b_k increases, $b_k = 10\,000$, $n \geq 10^6$. Processing such large data sets requires significant resources.

Simulation experiments for several distributions including *Normal(0; 1), Uniform(0; 10), StudentT(1.5), StudentT(3), Cauchy(0; 1), 0.95Normal(0; 1)+ 0.05Cauchy(0; 1)*, and *0.99Normal(0; 1) + 0.01Cauchy(104; 1)* are reported in [275]. The results of these experiments show that the diagnostic performs well across a range of data generating distributions and estimators. Moreover, its performance improves with the size of the sample data sets.

In summary, given $\mathcal{S}$ a random sample from $\mathcal{D}$, the subsamples generated by disjoint partitions of $\mathcal{S}$ are also mutually independent random samples from $\mathcal{D}$. The procedure must be carried out using a sequence of samples of increasingly larger size, $b_1, \ldots, b_i, \ldots, b_k$. It is necessary to ensure that the error decreases while increasing the sample size and that the error is sufficiently small for the largest sample.

12.14 Approximate query processing (R)

The advantages of executing a query on a data sample of rather than on an entire dataset are quite perspicuous, and this idea has been applied to sampling relational databases since the 1970s. Different versions of this technique have been investigated since its first use. Approximate query processing and sampling-based approximate query processing have become popular enough to warrant the acronyms, AQP and S-AQP, respectively, along with the realization that approximate answers are most useful if accompanied by accuracy estimates.

Let θ be a query on a dataset $\mathcal{D}$ and the desired answer to it be $\theta(\mathcal{D})$. Simple random sampling with plug-in estimation is often used for approximate query processing. This method generates a sample with replacement $\mathcal{S} \subset \mathcal{D}$ of cardinality $n = |\mathcal{S}| \leq |\mathcal{D}|$ and produces a *sample estimate result* $\theta(\mathcal{S})$ instead of computing $\theta(\mathcal{D})$ with the *sampling error* $\epsilon = \theta(\mathcal{S} - \mathcal{D})$ and the *sampling error distribution* $Dist(\epsilon)$.

The emergence of Big Data processed on large clusters in the cloud and the requirement of near real-time response have increased the demand for high-quality error estimates. Such error estimates can be reported to the users enabling them to judge the impact of errors on their specific applications and/or to application developers to decide if the sampling methods are adequate. These estimates can also be used to correlate the errors with the sample size necessary for the accuracy–response time tradeoffs.

Two methods for producing close-form estimates error bars are based on the Central Limit Theorem (CLT) and on Hoeffding bounds [220]. An error bar is a line segment through a point on a graph,

parallel to one of the axes, that represents the uncertainty or error of the corresponding coordinate of the point. Informally, CLT states that the sum of a large number of independent random variables has a normal distribution. More precisely, if $\{X_1, \ldots, X_n\}$ is a sequence of i.i.d. random variables with $E[X_i] = \mu$ and $Var[X_i] = \sigma^2 < \infty$ and $S_n = 1/n \sum_{i=1}^{n} X_i$ is the sample average, then the random variables $\sqrt{n}(S_n - \mu)$ converge to a normal distribution, $N(0, \sigma^2)$, as n approaches infinity:

$$\sqrt{n}\left[\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right) - \mu\right] \xrightarrow{d} N(0, \sigma^2). \tag{12.27}$$

The derivation of Hoeffding bounds starts with the estimation of the mean

$$\mu = \frac{1}{m}\sum_{i=1}^{m} v(i), \tag{12.28}$$

with v as a real-valued function defined on the set $\mathcal{S} = \{1, 2, \ldots, m\}$ and $m > 1$ as a fixed integer. Let $L_1, L_2, \ldots, L_n$ and $L'_1, L'_2, \ldots, L'_n$ be random samples from the set $\mathcal{S}$ with and without replacement, respectively. Given $n > 1$, let $\bar{Y}_n$ and $\bar{Y}'_n$ be two estimators for μ be defined as

$$\bar{Y}_n = \frac{1}{n}\sum_{i=1}^{n} v(L_i) \quad \text{and} \quad \bar{Y}'_n = \frac{1}{\min(n,m)}\sum_{i=1}^{\min(n,m)} v(L'_i). \tag{12.29}$$

These estimators are unbiased if $E[\bar{Y}_n] = E[\bar{Y}'_n]$. Hoeffding showed that, for any $n \geq m$ and convex function f, $E[f(\bar{Y}'_n)] \leq E[f(\bar{Y}_n)]$. It follows that when $f(x) = x^2 - \mu$,

$$Var[f(\bar{Y}'_n)] \leq Var[f(\bar{Y}_n)]. \tag{12.30}$$

These results are useful to bound the probability that the estimates deviate from μ more than a given amount. If the only information available before sampling is that

$$a \leq v(i) \leq b, \qquad 1 \leq i \leq m, \tag{12.31}$$

Hoeffding established the following bounds for the estimation error:

$$P\{|\, \bar{Y}_n - \mu \,| \geq t\} \leq 2e^{-2nt^2/(b-a)^2} \quad \text{and} \quad P\{|\, \bar{Y}'_n - \mu \,| \geq t\} \leq 2e^{-2n't^2/(b-a)^2} \tag{12.32}$$

for $t > 0$, $n \geq 1$ and

$$n' = \begin{cases} n & \text{if } n < m; \\ +\infty & \text{if } n \geq m. \end{cases} \tag{12.33}$$

Hoeffding bounds overestimate the error and increase the computational effort. The sample size for a system using Hoeffding bounds are one to two orders of magnitude larger than for CLT-based or bootstrap-based methods, but their accuracy is significantly higher. Experiments to estimate the sample sizes for achieving different levels of relative errors carried out on tens of terabytes of data are reported in [12] and reproduced in Fig. 12.13.

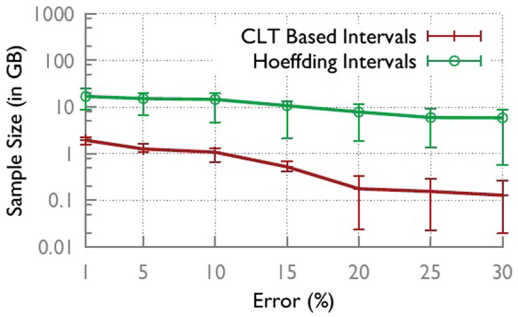

FIGURE 12.13

Sample size for CLT- and Hoeffding-based estimation for error bars [12].

Investigation of nonparametric bootstrap, closed-form estimation of variance, and large deviation bounds confirms that all the techniques have different failure modes. A benchmark, consisting of 100 different samples of some 10^6 rows, reported by [12], shows that only queries with COUNT, SUM, AVG, and VARIANCE aggregates are amenable to closed-form error estimation. All aggregates are amenable to the bootstrap, and 43.21% of the queries over one dataset and 62.79% of the queries over another can only be approximated using bootstrap-based error estimation methods. As expected, queries involving MAX and MIN are very sensitive to rare large or small values, respectively. In one data set, these two functions involved 2.87% and 33.35% of all queries, respectively. Bootstrap error estimation fails for 86.17% of these queries.

BEC: Bootstrap Error Computation for a SELECT query on $\mathcal{S}$

```
SELECT foo(col_S), ξ̂(resample_error) AS error
FROM (
.    SELECT foo(col_S) AS resample_answer
.    FROM S TABLE_SAMPLE POISSONIZED (100)
.      UNION ALL
.    SELECT foo(col_S) AS resample_answer
.    FROM S TABLE_SAMPLE POISSONIZED (100)
.      UNION ALL
. . . .
.      UNION ALL
.    SELECT foo(col_S) AS resample_answer
.    FROM S TABLE_SAMPLE POISSONIZED (100)
)
```

The diagnostic algorithm discussed in Section 12.13 is computationally intensive, thus impractical in many cases. A technique to quickly determine if a technique will work well for a particular query based on symmetrically centered confidence intervals was proposed in [12]. The workflow for a large-scale distributed approximate query processing proposed has several steps:

Logical Plan (LP)—a query is compiled into an LP consisting of three procedure to compute: (1) an approximative answer $\theta(\mathcal{S})$; (2) error $\bar{\xi}$; (3) diagnostic tests;
Physical Plan (PP)—initiates a DAG of tasks involving these procedures;
Data Storage Layer—distributes samples to a set of servers and manages cached data.

The system supports Poissonized resampling, thus allowing a straightforward implementation of the bootstrap error. For example, for a simple query of the form *SELECT foo(col_S) FROM* $\mathcal{S}$, the bootstrap error is computed by using the BEC pseudocode in the box.

An important source of inefficiency of the bootstrap method is the execution of the same query on various samples of the data. *Scan consolidation* can eliminate this source of inefficiency. As a first step of this process, the Logical Plan is optimized by extending the resampling operations. Each tuple in the sample $\mathcal{S}$ is extended with a set of 100 independent weights $w_1, w_2, \ldots, w_{100}$ drawn from a Poisson(1) distribution. The sample $\mathcal{S}$ is partitioned into multiple sets of $a = 50$, $b = 100$ and $c = 200$ MB, and three sets of weights, $D_{a1}, \ldots, D_{a100}$, $D_{b1}, \ldots, D_{b100}$ and $D_{c1}, \ldots, D_{c100}$, are associated with each row to create 100 resamples of each set.

The logical plan is rewritten for further optimization. After finding the longest set of consecutive operators that do not change the statistical properties of the set of columns that are being finally aggregated,[12] the custom Poissonized resampling operator is inserted right before the first nonpassthrough operator in the query graph. The subsequent aggregate operators are modified to compute a set of resample aggregates by appropriately scaling the corresponding aggregation column with the weights associated with every tuple. Further optimization of the cache management is used to improve the performance of the procedure discussed in [12].

12.15 Further readings

Several references, including [3,302,532,533], discuss defining characteristics of Big Data applications. Insights into Google's storage architecture for Big Data can be found in [173], and several systems, including Mesa, Spanner, and F1, are presented in [217], [117], and [447].

Data analytics is the subject of [528], and [104] analyzes interactive analytical processing in Big Data systems. [208] covers in-memory performance of Big Data. Continuous pipelines are discussed in [142]. The Starfish system for Big Data analytics is covered in [234], and [252] analyzes enterprise use of Big Data. Several papers including [86] and [275] cover bootstrapping techniques, and [12] analyzes approximate query processing. Hoeffding bounds are discussed in [220]. Several references such as [140,141,305] cover Dynamic Data-Driven Application Systems (DDDAS) and the FreshBreeze system.

Spark Streaming system is discussed in [534], and [15] covers the MillWheel framework developed at Google for building fault-tolerant and scalable data streaming systems. [419] presents caching strategies for data streaming. [24] covers Google's Photon system and system scalability, and the performance of large-scale system is the topic of [218]. The problems posed by the heavy-tail distribution of latency are analyzed in [130].

[12] These so-called *passthrough operators* could be scans, filters, projections, and so on.

Mobile devices and applications are covered in the literature, including [124,148,437,438,505]. Energy efficiency of mobile computing is analyzed in [347]. The use of mobile devices for space weather monitoring is discussed in [387]. The Follow-Me cloud and edge cloud computing are the subjects of [468] and [483,506].

12.16 Exercises and problems

Problem 1. Read [183] and analyze the benefits and the problems related to *dataspaces*. Research the literature for potential applications of dataspaces to data management of data-intensive applications in science and engineering.

Problem 2. Discuss the possible solution for stabilizing cloud services mentioned in [182] inspired by BGP routing [209,491].

Problem 3. Discussing the bootstrap method presented in Section 12.13, reference [275] states: "Ideally, we might approximate $\xi(P, n)$ for a given value of n by observing many independent datasets, each of size n. ...in practice we only observe a single set of n data points, rendering this approach an unachievable ideal. To surmount this difficulty, our diagnostic, the BPD Algorithm, executes this ideal procedure for dataset sizes smaller than n. That is, for a given $p \in \mathbb{N}$ and $b \leq \lceil n/p \rceil$ we randomly sample p disjoint subsets of the observed dataset $\mathcal{D}$, each of size b. For each subset, we compute the value of u; the resulting collection of u values approximates the distribution Q_b, in turn yielding a direct approximation of $\xi(P, b)$ the ground truth value for the smaller dataset size b. Additionally, we run the bootstrap on each of the p subsets of size b, and comparing the distribution of the resulting p bootstrap outputs to our ground truth approximation, we can determine whether or not the bootstrap performs acceptably well at sample size b." (1) Estimate the bootstrap speedup function of n, p, b; ignore the time to compute Δ_i, σ_i. (2) Examine the simulation results in [275] and discuss the impact of the sample size.

Problem 4. Read [12] and discuss the merits and the shortcomings of the three techniques for estimating the sample distribution using only a single sample: nonparametric bootstrap, close-form estimation, and large deviation bounds.

Problem 5. What is a key extraction function, and what role does it play in MillWheel? Give an example of such a key extraction function in Zeitgeist.

Problem 6. In systems with very high fan-out, a request may exercise an untested code path, causing crashes or extremely long delays on thousands of servers simultaneously. How can this problem be prevented?

Problem 7. Consider a system where each server typically responds in 10 ms but with a 99th-percentile latency of one second. If a user request must collect responses from 100 such servers in parallel, then 63% of user requests will take more than one second [130]. Assuming that there are 1 000 servers and the user request needs responses from 1 000 servers running in parallel, what is the probability that response latency will be larger than one second? What if, instead of 1 000 individual requests, the user request needs data from 2 000 servers running in parallel?

Problem 8. Research the power consumption of processors used in mobile devices and their energy efficiency. Rank the components of a mobile device in terms of power consumption. Establish a set of guidelines to minimize the power consumption of mobile applications.

Problem 9. What is the main benefit of Algorithm 1 for finding the optimal policy of a Markov Decision Process in [505] compared with the standard approaches?