CHAPTER

19 Parallel Computing

The activity of a computer must include the proper reacting to a possibly great variety of messages that can be sent to it at unpredictable moments, a situation which occurs in all information systems in which a number of computers are coupled to each other.

— "Cooperating sequential processes,"
E. W. DIJKSTRA, 1965

Parallel computation has become increasingly common. For example, laptops and desktops come with multiple processors which communicate through shared memory. High-end computation is often done using clusters consisting of individual computers communicating through a network.

Parallelism provides a number of benefits:

- High performance—more processors working on a task (usually) means it is completed faster.
- Better use of resources—a program can execute while another waits on the disk or network.
- Fairness—letting different users or programs share a machine rather than have one program run at a time to completion.
- Convenience—it is often conceptually more straightforward to do a task using a set of concurrent programs for the subtasks rather than have a single program manage all the subtasks.
- Fault tolerance—if a machine fails in a cluster that is serving web pages, the others can take over.

Concrete applications of parallel computing include graphical user interfaces (GUI) (a dedicated thread handles UI actions while other threads are, for example, busy doing network communication and passing results to the UI thread, resulting in increased responsiveness), Java virtual machines (a separate thread handles garbage collection which would otherwise lead to blocking, while another thread is busy running the user code), web servers (a single logical thread handles a single client request), scientific computing (a large matrix multiplication can be split across a cluster), and web search (multiple machines crawl, index, and retrieve web pages).

The two primary models for parallel computation are the shared memory model, in which each processor can access any location in memory, and the distributed memory model, in which a processor must explicitly send a message to another processor to access its memory. The former is more appropriate in the multicore setting and the latter is more accurate for a cluster. The questions in this chapter are focused on the shared memory model.

Writing correct parallel programs is challenging because of the subtle interactions between parallel components. One of the key challenges is races—two concurrent instruction sequences access the same address in memory and at least one of them writes to that address. Other challenges to correctness are

- starvation (a processor needs a resource but never gets it; e.g., Problem 19.6 on Page 296),
- deadlock (Thread A acquires Lock $L1$ and Thread B acquires Lock $L2$, following which A tries to acquire $L2$ and B tries to acquire $L1$), and

- livelock (a processor keeps retrying an operation that always fails).

Bugs caused by these issues are difficult to find using testing. Debugging them is also difficult because they may not be reproducible since they are usually load dependent. It is also often true that it is not possible to realize the performance implied by parallelism—sometimes a critical task cannot be parallelized, making it impossible to improve performance, regardless of the number of processors added. Similarly, the overhead of communicating intermediate results between processors can exceed the performance benefits.

The problems in this chapter focus on thread-level parallelism. Problems concerned with parallelism on distributed memory architectures, e.g., cluster computing, are usually not meant to be coded; they are popular design and architecture problems. Problems 20.9 on Page 310, 20.10 on Page 310, 20.11 on Page 311, and 20.17 on Page 316 cover some aspects of cluster-level parallelism.

Parallel computing boot camp

A semaphore is a very powerful synchronization construct. Conceptually, a semaphore maintains a set of permits. A thread calling *acquire()* on a semaphore waits, if necessary, until a permit is available, and then takes it. A thread calling *release()* on a semaphore adds a permit and notifies threads waiting on that semaphore, potentially releasing a blocking acquirer.

```
import threading


class Semaphore():
    def __init__(self, max_available):
        self.cv = threading.Condition()
        self.MAX_AVAILABLE = max_available
        self.taken = 0

    def acquire(self):
        self.cv.acquire()
        while (self.taken == self.MAX_AVAILABLE):
            self.cv.wait()
        self.taken += 1
        self.cv.release()

    def release(self):
        self.cv.acquire()
        self.taken -= 1
        self.cv.notify()
        self.cv.release()
```

Start with an algorithm that **locks aggressively** and is easily seen to be correct. Then **add back concurrency**, while ensuring the **critical** parts are locked. When analyzing parallel code, assume a worst-case thread scheduler. In particular, it may choose to schedule the same thread repeatedly, it may alternate between two threads, it may starve a thread, etc. Try to work at a **higher level of abstraction**. In particular, know the **concurrency libraries**—don't implement your own **semaphores, thread pools, deferred execution**, etc. (You should know how these features are implemented, and implement them if asked to.)

Table 19.1: Top Tips for Concurrency

19.1 Implement caching for a multithreaded dictionary

The program below is part of an online spell correction service. Clients send as input a string, and the service returns an array of strings in its dictionary that are closest to the input string (this array could be computed, for example, using Solution 16.2 on Page 239). The service caches results to improve performance. Critique the implementation and provide a solution that overcomes its limitations.

```
class SpellCheckService:
    w_last = closest_to_last_word = None

    @staticmethod
    def service(req, resp):
        w = req.extract_word_to_check_from_request()
        if w != SpellCheckService.w_last:
            SpellCheckService.w_last = w
            SpellCheckService.closest_to_last_word = closest_in_dictionary(w)
        resp.encode_into_response(SpellCheckService.closest_to_last_word)
```

Hint: Look for races, and lock as little as possible to avoid reducing throughput.

Solution: The solution has a race condition. Suppose clients A and B make concurrent requests, and the service launches a thread per request. Suppose the thread for request A finds that the input string is present in the cache, and then, immediately after that check, the thread for request B is scheduled. Suppose this thread's lookup fails, so it computes the result, and adds it to the cache. If the cache is full, an entry will be evicted, and this may be the result for the string passed in request A. Now when request A is scheduled back, it does a lookup for the value corresponding to its input string, expecting it to be present (since it checked that that string is a key in the cache). However, the cache will return null.

A thread-safe solution would be to synchronize every call to the service. In this case, only one thread could be executing the method and there are no races between cache reads and writes. However, it also leads to poor performance—only one thread can execute the service call at a time.

The solution is to lock just the part of the code that operates on the cached values—specifically, the check on the cached value and the updates to the cached values:

In the program below, multiple threads can be concurrently computing closest strings. This is good because the calls take a long time (this is why they are cached). Locking ensures that the read assignment on a hit and write assignment on completion are atomic.

```
class SpellCheckService:
    w_last = closest_to_last_word = None
    lock = threading.Lock()

    @staticmethod
    def service(req, resp):
        w = req.extract_word_to_check_from_request()
        result = None
        with SpellCheckService.lock:
            if w == SpellCheckService.w_last:
                result = SpellCheckService.closest_to_last_word.copy()
        if result is None:
            result = closest_in_dictionary(w)
            with SpellCheckService.lock:
                SpellCheckService.w_last = w
```

```
                SpellCheckService.closest_to_last_word = result
        resp.encode_into_response(result)
```

Variant: Threads 1 to n execute a method called *critical()*. Before this, they execute a method called *rendezvous()*. The synchronization constraint is that only one thread can execute *critical()* at a time, and all threads must have completed executing *rendezvous()* before *critical()* can be called. You can assume n is stored in a variable n that is accessible from all threads. Design a synchronization mechanism for the threads. All threads must execute the same code. Threads may call *critical()* multiple times, and you should ensure that a thread cannot call *critical()* a $(k+1)$th time until all other threads have completed their kth calls to *critical()*.

19.2 Analyze two unsynchronized interleaved threads

Threads $t1$ and $t2$ each increment an integer variable N times, as shown in the code below. This program yields nondeterministic results. Usually, it prints $2N$ but sometimes it prints a smaller value. The problem is more pronounced for large N. As a concrete example, on one run the program output 1320209 when $N = 1000000$ was specified at the command line.

```
N = 1000000
counter = 0


def increment_thread():
    global counter
    for _ in range(N):
        counter = counter + 1


t1 = threading.Thread(target=increment_thread)
t2 = threading.Thread(target=increment_thread)

t1.start()
t2.start()
t1.join()
t2.join()

print(counter)
```

What are the maximum and minimum values that could be printed by the program as a function of N?

Hint: Be as perverse as you can when scheduling the threads.

Solution: First, note that the increment code is unguarded, which opens up the possibility of its value being determined by the order in which threads that write to it are scheduled by the thread scheduler.

The maximum value is $2N$. This occurs when the thread scheduler runs one thread to completion, followed by the other thread.

When $N = 1$, the minimum value for the count variable is 1: $t1$ reads, $t2$ reads, $t1$ increments and writes, then $t2$ increments and writes. When $N > 1$, the final value of the count variable must be at least 2. The reasoning is as follows. There are two possibilities. A thread, call it T, performs a read-increment-write-read-increment-write without the other thread writing between reads, in

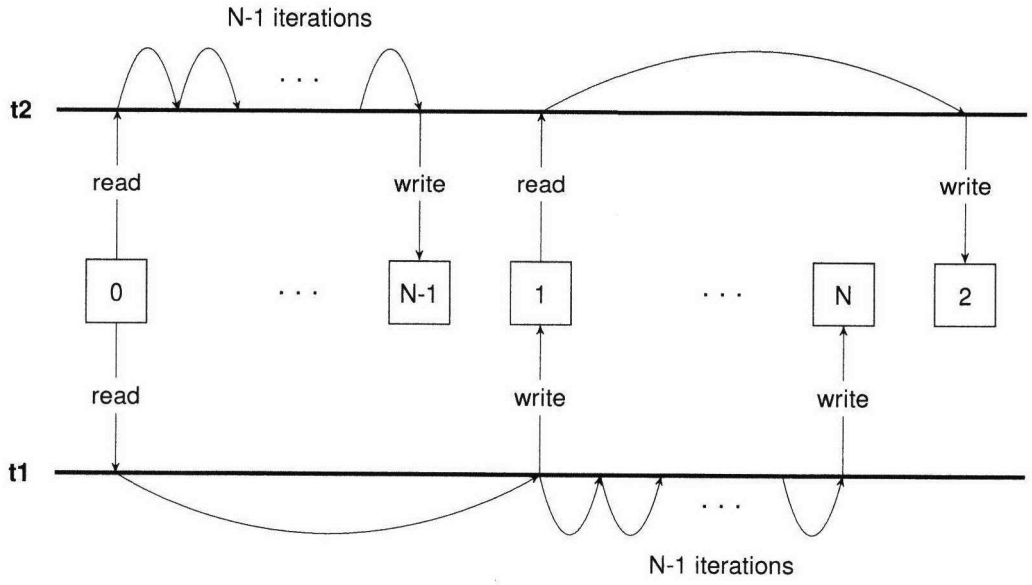

Figure 19.1: Worst-case schedule for two unsynchronized threads.

which case the written value is at least 2. If the other thread now writes a 1, it has not yet completed, so it will increment at least once more. Otherwise, T's second read returns a value of 1 or more (since the other thread has performed at least one write).

The lower bound of 2 is achieved according to the following thread schedule:

- $t1$ loads the value of the counter, which is 0.
- $t2$ executes the loop $N-1$ times.
- $t1$ doesn't know that the value of the counter changed and writes 1 to it.
- $t2$ loads the value of the counter, which is 1.
- $t1$ executes the loop for the remaining $N-1$ iterations.
- $t2$ doesn't know that the value of the counter has changed, and writes 2 to the counter.

This schedule is depicted in Figure 19.1.

19.3 Implement synchronization for two interleaving threads

Thread $t1$ prints odd numbers from 1 to 100; Thread $t2$ prints even numbers from 1 to 100.

Write code in which the two threads, running concurrently, print the numbers from 1 to 100 in order.

Hint: The two threads need to notify each other when they are done.

Solution: A brute-force solution is to use a lock which is repeatedly captured by the threads. A single variable, protected by the lock, indicates who went last. The drawback of this approach is that it employs the busy waiting antipattern: processor time that could be used to execute a different task is instead wasted on useless activity.

Below we present a solution based on the same idea, but one that avoids busy locking by using

```
class OddEvenMonitor(threading.Condition):

    ODD_TURN = True
    EVEN_TURN = False
```

```
    def __init__(self):
        super().__init__()
        self.turn = self.ODD_TURN

    def wait_turn(self, old_turn):
        with self:
            while self.turn != old_turn:
                self.wait()

    def toggle_turn(self):
        with self:
            self.turn ^= True
            self.notify()


class OddThread(threading.Thread):
    def __init__(self, monitor):
        super().__init__()
        self.monitor = monitor

    def run(self):
        for i in range(1, 101, 2):
            self.monitor.wait_turn(OddEvenMonitor.ODD_TURN)
            print(i)
            self.monitor.toggle_turn()


class EvenThread(threading.Thread):
    def __init__(self, monitor):
        super().__init__()
        self.monitor = monitor

    def run(self):
        for i in range(2, 101, 2):
            self.monitor.wait_turn(OddEvenMonitor.EVEN_TURN)
            print(i)
            self.monitor.toggle_turn()
```

19.4 Implement a thread pool

The following program, implements part of a simple HTTP server:

```
SERVERPORT = 8080


def main():
    serversock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    serversock.bind(('', SERVERPORT))
    serversock.listen(5)
    while True:
        sock, addr = serversock.accept()
        process_req(sock)
```

Suppose you find that the program has poor performance because it frequently blocks on I/O. What steps could you take to improve the program's performance? Feel free to use any utilities from the standard library, including concurrency classes.

Hint: Use multithreading, but control the number of threads.

Solution: The first attempt to solve this problem might be to launch a new thread per request rather than process the request itself:

```
SERVERPORT = 8080

serversock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversock.bind(('', SERVERPORT))
serversock.listen(5)
while True:
    sock, addr = serversock.accept()
    threading.Thread(target=process_req, args=(sock, )).start()
```

The problem with this approach is that we do not control the number of threads launched. A thread consumes a nontrivial amount of resources, such as the time taken to start and end the thread and the memory used by the thread. For a lightly-loaded server, this may not be an issue but under load, it can result in exceptions that are challenging, if not impossible, to handle.

The right trade-off is to use a *thread pool*. As the name implies, this is a collection of threads, the size of which is bounded. A thread pool can be implemented relatively easily using a blocking queue, i.e., a queue which blocks the writing thread on a put until the queue is empty. However, since the problem statement explicitly allows us to use library routines, we can use

```
SERVERPORT = 8080
NTHREADS = 2

executor = concurrent.futures.ThreadPoolExecutor(max_workers=NTHREADS)
serversock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversock.bind(('', SERVERPORT))
serversock.listen(5)
while True:
    sock, addr = serversock.accept()
    executor.submit(process_req, sock)
```

19.5 Deadlock

When threads need to acquire multiple locks to enter a critical section, deadlock can result. As an example, suppose both $T1$ and $T2$ need to acquire locks L and M. If $T1$ first acquires L, and then $T2$ then acquires M, they end up waiting on each other forever.

Identify a concurrency bug in the program below, and modify the code to resolve the issue.

```
class Account:

    _global_id = 0

    def __init__(self, balance):
        self._balance = balance
        self._id = Account._global_id
        Account._global_id += 1
        self._lock = threading.RLock()
```

```
    def get_balance(self):
        return self._balance

    @staticmethod
    def transfer(acc_from, acc_to, amount):
        th = threading.Thread(target=acc_from._move, args=(acc_to, amount))
        th.start()

    def _move(self, acc_to, amount):
        with self._lock:
            if amount > self._balance:
                return False
            acc_to._balance += amount
            self._balance -= amount
            print('returning True')
            return True
```

Solution: Suppose $U1$ initiates a transfer to $U2$, and immediately afterwards, $U2$ initiates a transfer to $U1$. Since each transfer takes place in a separate thread, it's possible for the first thread to lock $U1$ and then the second thread to be scheduled in and take the lock $U2$. The program is now deadlocked—each of the two threads is waiting for the lock held by the other thread.

One solution is to have a global lock which is acquired by the transfer method. The drawback is that it blocks transfers that are unrelated, e.g., $U3$ cannot transfer to $U4$ if there is a pending transfer from $U5$ to $U6$.

The canonical way to avoid deadlock is to have a global ordering on locks and acquire them in that order. Since accounts have a unique integer id, the update below is all that is needed to solve the deadlock.

```
lock1 = self._lock if self._id < acc_to._id else acc_to._lock
lock2 = acc_to._lock if self._id < acc_to._id else self._lock
# Does not matter if lock1 equals lock2: since recursive_mutex locks
# are reentrant, we will re-acquire lock2.
with lock1, lock2:
```

19.6 The readers-writers problem

Consider an object s which is read from and written to by many threads. (For example, s could be the cache from Problem 19.1 on Page 291.) You need to ensure that no thread may access s for reading or writing while another thread is writing to s. (Two or more readers may access s at the same time.)

One way to achieve this is by protecting s with a mutex that ensures that two threads cannot access s at the same time. However, this solution is suboptimal because it is possible that a reader $R1$ has locked s and another reader $R2$ wants to access s. Reader $R2$ does not have to wait until $R1$ is done reading; instead, $R2$ should start reading right away.

This motivates the first readers-writers problem: protect s with the added constraint that no reader is to be kept waiting if s is currently opened for reading.

Implement a synchronization mechanism for the first readers-writers problem.

Hint: Track the number of readers.

Solution: We want to keep track of whether the string is being read from, as well as whether the string is being written to. Additionally, if the string is being read from, we want to know the number of concurrent readers. We achieve this with a pair of locks—a read lock and a write lock—and a read counter locked by the read lock.

A reader proceeds as follows. It locks the read lock, increments the counter, and releases the read lock. After it performs its reads, it locks the read lock, decrements the counter, and releases the read lock. A writer locks the write lock, then performs the following in an infinite loop. It locks the read lock, checks to see if the read counter is 0; if so, it performs its write, releases the read lock, and breaks out of the loop. Finally, it releases the write lock. As in Solution 19.3 on Page 293, we use wait-notify primitives to avoid busy waiting.

```
# LR and LW are class attributes in the RW class.
# They serve as read and write locks. The integer
# variable read_count in RW tracks the number of readers.
class Reader(threading.Thread):

    def run(self):
        while True:
            with RW.LR:
                RW.read_count += 1


            print(RW.data)
            with RW.LR:
                RW.read_count -= 1
                RW.LR.notify()
            do_something_else()


class Writer(threading.Thread):

    def run(self):
        while True:
            with RW.LW:
                done = False
                while not done:
                    with RW.LR:
                        if RW.read_count == 0:
                            RW.data += 1
                            done = True
                        else:
                            # use wait/notify to avoid busy waiting
                            while RW.read_count != 0:
                                RW.LR.wait()
            do_something_else()
```

19.7 The readers-writers problem with write preference

Suppose we have an object s as in Problem 19.6 on the preceding page. In the solution to Problem 19.6 on the facing page, a reader $R1$ may have the lock; if a writer W is waiting for the lock and then a reader $R2$ requests access, $R2$ will be given priority over W. If this happens often enough, W will starve. Instead, suppose we want W to start as soon as possible.

This motivates the second readers-writers problem: protect s with "writer-preference", i.e., no writer, once added to the queue, is to be kept waiting longer than absolutely necessary.

Implement a synchronization mechanism for the second readers-writers problem.

Hint: Force readers to acquire a write lock.

Solution: We want to give writers the preference. We achieve this by modifying Solution 19.6 on the previous page to have a reader start by locking the write lock and then immediately releasing it. In this way, a writer who acquires the write lock is guaranteed to be ahead of the subsequent readers.

Variant: The specifications to Problems 19.6 on Page 296 and 19.7 on the previous page allow starvation—the first may starve writers, the second may starve readers. The third readers-writers problem adds the constraint that neither readers nor writers should starve. Implement a synchronization mechanism for the third readers-writers problem.

19.8 Implement a Timer class

Consider a web-based calendar in which the server hosting the calendar has to perform a task when the next calendar event takes place. (The task could be sending an email or a Short Message Service (SMS).) Your job is to design a facility that manages the execution of such tasks.

Develop a timer class that manages the execution of deferred tasks. The timer constructor takes as its argument an object which includes a run method and a string-valued name field. The class must support—(1.) starting a thread, identified by name, at a given time in the future; and (2.) canceling a thread, identified by name (the cancel request is to be ignored if the thread has already started).

Hint: There are two aspects—data structure design and concurrency.

Solution: The two aspects to the design are the data structures and the locking mechanism.

We use two data structures. The first is a min-heap in which we insert key-value pairs: the keys are run times and the values are the thread to run at that time. A dispatch thread runs these threads; it sleeps from call to call and may be woken up if a thread is added to or deleted from the pool. If woken up, it advances or retards its remaining sleep time based on the top of the min-heap. On waking up, it looks for the thread at the top of the min-heap—if its launch time is the current time, the dispatch thread deletes it from the min-heap and executes it. It then sleeps till the launch time for the next thread in the min-heap. (Because of deletions, it may happen that the dispatch thread wakes up and finds nothing to do.)

The second data structure is a hash table with thread ids as keys and entries in the min-heap as values. If we need to cancel a thread, we go to the min-heap and delete it. Each time a thread is added, we add it to the min-heap; if the insertion is to the top of the min-heap, we interrupt the dispatch thread so that it can adjust its wake up time.

Since the min-heap is shared by the update methods and the dispatch thread, we need to lock it. The simplest solution is to have a single lock that is used for all read and writes into the min-heap and the hash table.

19.9 Test the Collatz conjecture in parallel

In Problem 12.11 on Page 176 and its solution we introduced the Collatz conjecture and heuristics for checking it. In this problem, you are to build a parallel checker for the Collatz conjecture.

Specifically, assume your program will run on a multicore machine, and threads in your program will be distributed across the cores. Your program should check the Collatz conjecture for every integer in $[1, U]$ where U is an input to your program.

Design a multi-threaded program for checking the Collatz conjecture. Make full use of the cores available to you. To keep your program from overloading the system, you should not have more than n threads running at a time.

Hint: Use multithreading for performance—take care to minimize threading overhead.

Solution: Heuristics for pruning checks on individual integers are discussed in Solution 12.11 on Page 176. The aim of this problem is implementing a multi-threaded checker. We could have a master thread launch n threads, one per number, starting with $1, 2, \ldots, x$. The master thread would keep track of what number needs to be processed next, and when a thread returned, it could re-assign it the next unchecked number.

The problem with this approach is that the time spent executing the check in an individual thread is very small compared to the overhead of communicating with the thread. The natural solution is to have each thread process a subrange of $[1, U]$. We could do this by dividing $[1, U]$ into n equal sized subranges, and having Thread i handle the ith subrange.

The heuristics for checking the Collatz conjecture take longer on some integers than others, and in the strategy above there is the potential of a situation arising where one thread takes much longer to complete than the others, which leads to most of the cores being idle.

A good compromise is to have threads handle smaller intervals, which are still large enough to offset the thread overhead. We can maintain a work-queue consisting of unprocessed intervals, and assigning these to returning threads.

```
# Performs basic unit of work
def worker(lower, upper):
    for i in range(lower, upper + 1):
        assert collatz_check(i, set())
    print('(%d,%d)' % (lower, upper))


# Checks an individual number
def collatz_check(x, visited):
    if x == 1:
        return True
    elif x in visited:
        return False
    visited.add(x)
    if x & 1:  # odd number
        return collatz_check(3 * x + 1, visited)
    else:  # even number
        return collatz_check(x >> 1, visited)  # divide by 2


    # Uses the library thread pool for task assignment and load balancing
    executor = concurrent.futures.ProcessPoolExecutor(max_workers=NTHREADS)
    with executor:
        for i in range(N // RANGESIZE):
            executor.submit(worker, i * RANGESIZE + 1, (i + 1) * RANGESIZE)
```

Part III

Domain Specific Problems

CHAPTER

20 Design Problems

Don't be fooled by the many books on complexity or by the many complex and arcane algorithms you find in this book or elsewhere. Although there are no textbooks on simplicity, simple systems work and complex don't.

— *"Transaction Processing: Concepts and Techniques,"* J. GRAY, 1992

You may be asked in an interview how to go about creating a set of services or a larger system, possibly on top of an algorithm that you have designed. These problems are fairly open-ended, and many can be the starting point for a large software project.

In an interview setting when someone asks such a question, you should have a conversation in which you demonstrate an ability to think creatively, understand design trade-offs, and attack unfamiliar problems. You should sketch key data structures and algorithms, as well as the technology stack (programming language, libraries, OS, hardware, and services) that you would use to solve the problem.

The answers in this chapter are presented in this context—they are meant to be examples of good responses in an interview and are not comprehensive state-of-the-art solutions.

We review patterns that are useful for designing systems in Table 20.1. Some other things to keep in mind when designing a system are implementation time, extensibility, scalability, testability, security, internationalization, and IP issues.

Table 20.1: System design patterns.

Design principle	Key points
Algorithms and Data Structures	Identify the basic algorithms and data structures
Decomposition	Split the functionality, architecture, and code into manageable, reusable components.
Scalability	Break the problem into subproblems that can be solved relatively independently on different machines. Shard data across machines to make it fit. Decouple the servers that handle writes from those that handle reads. Use replication across the read servers to gain more performance. Consider caching computation and later look it up to save work.

Decomposition

Good decompositions are critical to successfully solving system-level design problems. Functionality, architecture, and code all benefit from decomposition.

For example, in our solution to designing a system for online advertising (Problem 20.15 on Page 315), we decompose the goals into categories based on the stake holders. We decompose the architecture itself into a front-end and a back-end. The front-end is divided into user management, web page design, reporting functionality, etc. The back-end is made up of middleware, storage, database, cron services, and algorithms for ranking ads. The design of TEX (Problem 20.6 on Page 307) and Connexus (Problem 20.14 on Page 314) also illustrate such decompositions.

Decomposing code is a hallmark of object-oriented programming. The subject of design patterns is concerned with finding good ways to achieve code-reuse. Broadly speaking, design patterns are grouped into creational, structural, and behavioral patterns. Many specific patterns are very natural—strategy objects, adapters, builders, etc., appear in a number of places in our codebase. Freeman *et al.*'s *"Head First Design Patterns"* is, in our opinion, the right place to study design patterns.

Scalability

In the context of interview questions parallelism is useful when dealing with scale, i.e., when the problem is too large to fit on a single machine or would take an unacceptably long time on a single machine. The key insight you need to display is that you know how to decompose the problem so that

- each subproblem can be solved relatively independently, and
- the solution to the original problem can be efficiently constructed from solutions to the subproblems.

Efficiency is typically measured in terms of central processing unit (CPU) time, random access memory (RAM), network bandwidth, number of memory and database accesses, etc.

Consider the problem of sorting a petascale integer array. If we know the distribution of the numbers, the best approach would be to define equal-sized ranges of integers and send one range to one machine for sorting. The sorted numbers would just need to be concatenated in the correct order. If the distribution is not known then we can send equal-sized arbitrary subsets to each machine and then merge the sorted results, e.g., using a min-heap. Details are given in Solution 20.9 on Page 310.

The solutions to Problems 20.8 on Page 308 and 20.17 on Page 316 also illustrate the use of parallelism.

Caching is a great tool whenever computations are repeated. For example, the central idea behind dynamic programming is caching results from intermediate computations. Caching is also extremely useful when implementing a service that is expected to respond to many requests over time, and many requests are repeated. Workloads on web services exhibit this property. Solution 19.1 on Page 291 sketches the design of an online spell correction service; one of the key issues is performing cache updates in the presence of concurrent requests. Solution 19.9 on Page 299 shows how multithreading combines with caching in code which tests the Collatz hypothesis.

20.1 Design a spell checker

Designing a good spelling correction system can be challenging. We discussed spelling correction in the context of edit distance (Problem 16.2 on Page 239). However, in that problem, we only computed the Levenshtein distance between a pair of strings. A spell checker must find a set of words that are closest to a given word from the entire dictionary. Furthermore, the Levenshtein

distance may not be the right distance function when performing spelling correction—it does not take into account the commonly misspelled words or the proximity of letters on a keyboard.

How would you build a spelling correction system?

Hint: Start with an appropriate notion of distance between words.

Solution: The basic idea behind most spelling correction systems is that the misspelled word's Levenshtein distance from the intended word tends to be very small (one or two edits). Hence, if we keep a hash table for all the words in the dictionary and look for all the words that have a Levenshtein distance of 2 from the text, it is likely that the intended word will be found in this set. If the alphabet has m characters and the search text has n characters, we need to perform $O(n^2m^2)$ hash table lookups. More precisely, for a word of length n, we can pick any two characters and change them to any other character in the alphabet. The total number of ways of selecting any two characters is $n(n-1)/2$, and each character can be changed to one of $(m-1)$ other chars. Therefore, the number of lookups is $n(n-1)(m-1)^2/2$.

The intersection of the set of all strings at a distance of two or less from a word and the set of dictionary words may be large. It is important to provide a ranked list of suggestions to the users, with the most likely candidates are at the beginning of the list. There are several ways to achieve this:

- Typing errors model—often spelling mistakes are a result of typing errors. Typing errors can be modeled based on keyboard layouts.
- Phonetic modeling—a big class of spelling errors happen when the person spelling it knows how the words sounds but does not know the exact spelling. In such cases, it helps to map the text to phonemes and then find all the words that map to the same phonetic sequence.
- History of refinements—often users themselves provide a great amount of data about the most likely misspellings by first entering a misspelled word and then correcting it. This historic data is often immensely valuable for spelling correction.
- Stemming—often the size of a dictionary can be reduced by keeping only the stemmed version of each word. (This entails stemming the query text.)

20.2 Design a solution to the stemming problem

When a user submits the query "computation" to a search engine, it is quite possible he might be interested in documents containing the words "computers", "compute", and "computing" also. If you have several keywords in a query, it becomes difficult to search for all combinations of all variants of the words in the query.

Stemming is the process of reducing all variants of a given word to one common root, both in the query string and in the documents. An example of stemming would be mapping $\{computers, computer, compute\}$ to *compute*. It is almost impossible to succinctly capture all possible variants of all words in the English language but a few simple rules can get us most cases.

Design a stemming algorithm that is fast and effective.

Hint: The examples are suggestive of general rules.

Solution: Stemming is a large topic. Here we mention some basic ideas related to stemming—this is in no way a comprehensive discussion on stemming approaches.

Most stemming systems are based on simple rewrite rules, e.g., remove suffixes of the form "es", "s", and "ation". Suffix removal does not always work. For example, wolves should be stemmed to wolf. To cover this case, we may have a rule that replaces the suffix "ves" with "f".

Most rules amount to matching a set of suffixes and applying the corresponding transformation to the string. One way of efficiently performing this is to build a finite state machine based on all the rules.

A more sophisticated system might have exceptions to the broad rules based on the stem matching some patterns. The Porter stemmer, developed by Martin Porter, is considered to be one of the most authoritative stemming algorithms in the English language. It defines several rules based on patterns of vowels and consonants.

Other approaches include the use of stochastic methods to learn rewrite rules and n-gram based approaches where we look at the surrounding words to determine the correct stemming for a word.

20.3 Plagiarism detector

Design an efficient algorithm that takes as input a set of text files and returns pairs of files which have substantial commonality.

Hint: Design a hash function which can incrementally hash $S[i, i+k-1]$ for $i = 0, 1, 2, \ldots$.

Solution: We will treat each file as a string. We take a pair of files as having substantial commonality if they have a substring of length k in common, where k is a measure of commonality. (Later, we will have a deeper discussion as to the validity of this model.)

Let l_i be the length of the string corresponding to the ith file. For each such string we can compute $l_i - k + 1$ hash codes, one for *each* k length substring.

We insert these hash codes in a hash table G, recording which file each code corresponds to, and the offset the corresponding substring appears at. A collision indicates that the two length-k substrings are potentially the same.

Since we are computing hash code for each k length substring, it is important for efficiency to have a hash function which can be updated incrementally when we delete one character and add another. Solution 6.13 on Page 80 describes such a hash function.

In addition, it is important to have many slots in G, so that collisions of unequal strings is rare. A total of $\sum_{i=1}^{|S|}(l_i - k + 1)$ strings are added to the hash table. If k is small relative to the string length and G has significantly fewer slots than the total number of characters in the strings, then we are certain to have collisions.

If it is not essential to return an exact answer, we can save storage by only considering a subset of substrings, e.g., those whose hash codes have 0s in the last b bits. This means that on average we consider $\frac{1}{2^b}$ of the total set of substrings (assuming the hash function does a reasonable job of spreading keys).

The solution presented above can lead to many false positives. For example, if each file corresponds to an HTML page, all pages with a common embedded script of length k or longer will show up as having substantial overlap. We can account for this through preprocessing, e.g., parsing the documents and removing headers. (This may require multiple passes with some manual inspection driving the process.) This solution will also not work if, for example, the files are similar in that they are the exact same program, but with ids renamed. (It is however, resilient to code blocks being

moved around.) By normalizing ids, we can cast the problem back into one of finding common substrings.

The approach we have just described is especially effective when the number of files is very large and the files are spread over many servers. In particular, the map-reduce framework can be used to achieve the effect of a single G spread over many servers.

20.4 Pair users by attributes

You are building a social network where each user specifies a set of attributes. You would like to pair each user with another unpaired user that specified the same set of attributes.

You are given a sequence of users where each user has a unique 32-bit integer key and a set of attributes specified as strings. When you read a user, you should pair that user with another previously read user with identical attributes who is currently unpaired, if such a user exists. If the user cannot be paired, you should add him to the unpaired set.

Hint: Map sets of attributes to strings.

Solution: First, we examine the algorithmic core of this problem. Later we will consider implementation details, particularly those related to scale.

A brute-force algorithm is to compare each new user's attributes with the attributes of users in the unpaired set. This leads to $O(n^2)$ time complexity, where n is the number of users.

To improve the time complexity, we need a way to find users who have the same attributes as the new user. A hash table whose keys are subsets of attributes and values are users is the perfect choice. This leaves us with the problem of designing a hash function which is suitable for subsets of attributes.

If the total number of possible attributes is small, we can represent a subset of attributes with a bit array, where each index represents a specific attribute. Specifically, we store a 1 at an index if that attribute is present. We can then use any hash function for bit arrays. The time complexity to process the n users is a hash lookup per user, i.e., $O(nm)$, where m is the number of distinct attributes. The space complexity is also $O(nm)$—n entries, each of size m.

If the set of possible attributes is large, and most users have a small subset of those attributes, the bit vector representation for subsets is inefficient from both a time and space perspective. A better approach to represent sparse subsets is directly record the elements. To ensure equal subsets have equal hash codes, we need to represent the subsets in a unique way. One approach is to sort the elements. We can view the sorted sequence of attributes as a single string, namely the concatenation the individual elements. We then use a hash function for strings. For example, if the possible attributes are {USA, Senior, Income, Prime Customer}, and a user has attributes {USA, Income}, we represent his set of attributes as the string "Income,USA".

The time complexity of this approach is $O(M)$, where M is the sum of the sizes of the attribute sets for all users.

Now we consider implementation issues. Suppose the network is small enough that a single machine can store it. For such a system, you can use database with Users table, Attributes table, and a join table between them. Then for every attribute you will find matches (or no match) and based on criteria you can decide if user could join a group or start a new one. If you do not like to use a database and want to reinvent the wheel, you still can use similar approach and get a list

of matching user IDs for every attribute. To do it use reverse indexes, with a string hash for quick lookup in storage. Assuming arrays are returned sorted there is an easy way to merge arrays for multiple attributes and make a decision about matching to a group / starting new group.

Now suppose the network is too large to fit on a single machine. Because of number of users it makes sense to process problem on multiple machines—each will store subset of attributes and may return results as user IDs. We will need to make two merge operations:

- Identical searches—when a single attribute lookup was performed on many machines. We will need to merge returned sorted IDs into big sorted list of IDs.
- Searches by different attributes—we will need to merge those attributes if they all are present in all lists (or if the match criteria is above X%, e.g. 3/4 are matching).

Since in all likelihood we do not need to perform real-time matching we could use the Consumer-Producer pattern: pick up users from a queue and perform a search, thereby limit the number of simultaneous requests.

20.5 Design a system for detecting copyright infringement

YouTV.com is a successful online video sharing site. Hollywood studios complain that much of the material uploaded to the site violates copyright.

Design a feature that allows a studio to enter a set V of videos that belong to it, and to determine which videos in the YouTV.com database match videos in V.

Hint: Normalize the video format and create signatures.

Solution: If we replaced videos everywhere with documents, we could use the techniques in Solution 20.3 on Page 304, where we looked for near duplicate documents by computing hash codes for each length-k substring.

Videos differ from documents in that the same content may be encoded in many different formats, with different resolutions, and levels of compression.

One way to reduce the duplicate video problem to the duplicate document problem is to re-encode all videos to a common format, resolution, and compression level. This in itself does not mean that two videos of the same content get reduced to identical files—the initial settings affect the resulting videos. However, we can now "signature" the normalized video.

A trivial signature would be to assign a 0 or a 1 to each frame based on whether it has more or less brightness than average. A more sophisticated signature would be a 3 bit measure of the red, green, and blue intensities for each frame. Even more sophisticated signatures can be developed, e.g., by taking into account the regions on individual frames. The motivation for better signatures is to reduce the number of false matches returned by the system, and thereby reduce the amount of time needed to review the matches.

The solution proposed above is algorithmic. However, there are alternative approaches that could be effective: letting users flag videos that infringe copyright (and possibly rewarding them for their effort), checking for videos that are identical to videos that have previously been identified as infringing, looking at meta-information in the video header, etc.

Variant: Design an online music identification service.

20.6 Design TEX

The TEX system for typesetting beautiful documents was designed by Don Knuth. Unlike GUI based document editing programs, TEX relies on a markup language, which is compiled into a device independent intermediate representation. TEX formats text, lists, tables, and embedded figures; supports a very rich set of fonts and mathematical symbols; automates section numbering, cross-referencing, index generation; exports an API; and much more.

How would you implement TEX?

Hint: There are two aspects—building blocks (e.g., fonts, symbols) and hierarchical layout.

Solution: Note that the problem does not ask for the design of TEX, which itself is a complex problem involving feature selection, and language design. There are a number of issues common to implementing any such program: programming language selection, lexing and parsing input, error handling, macros, and scripting.

Two key implementation issues specific to TEX are a specifying fonts and symbols (e.g., `A`, `b`, f, $\sum$, $\oint$, $\uplus$), and assembling a document out of components.

Focusing on the second aspect, a reasonable abstraction is to use a rectangular bounding box to describe components. The description is hierarchical: each individual symbol is a rectangle, lines and paragraphs are made out of these rectangles and are themselves rectangles, as are section titles, tables and table entries, and included images. A key algorithmic problem is to assemble these rectangles, while preserving hard constraints on layout, and soft constraints on aesthetics. See also Problem 16.11 on Page 254 for an example of the latter.

Turning our attention to symbol specification, the obvious approach is to use a 2D array of bits to represent each symbol. This is referred to as a bit-mapped representation. The problem with bit-mapped fonts is that the resolution required to achieve acceptable quality is very high, which leads to huge documents and font-libraries. Different sizes of the same symbol need to be individually mapped, as do italicized and bold-face versions.

A better approach is to define symbols using mathematical functions. A reasonable approach is to use a language that supports quadratic and cubic functions, and elementary graphics transformations (rotation, interpolation, and scaling). This approach overcomes the limitations of bit-mapped fonts—parameters such as aspect ratio, font slant, stroke width, serif size, etc. can be programmed.

Other implementation issues include enabling cross-referencing, automatically creating indices, supporting colors, and outputting standard page description formats (e.g., PDF)).

Donald Knuth's book *"Digital Typography"* describes in great detail the design and implementation of TEX.

20.7 Design a search engine

Keyword-based search engines maintain a collection of several billion documents. One of the key computations performed by a search engine is to retrieve all the documents that contain the keywords contained in a query. This is a nontrivial task in part because it must be performed in a few tens of milliseconds.

Here we consider a smaller version of the problem where the collection of documents can fit within the RAM of a single computer.

Given a million documents with an average size of 10 kilobytes, design a program that can efficiently return the subset of documents containing a given set of words.

Hint: Build on the idea of a book's index.

Solution: The predominant way of doing this is to build inverted indices. In an inverted index, for each word, we store a sequence of locations where the word occurs. The sequence itself could be represented as an array or a linked list. Location is defined to be the document ID and the offset in the document. The sequence is stored in sorted order of locations (first ordered by document ID, then by offset). When we are looking for documents that contain a set of words, what we need to do is find the intersection of sequences for each word. Since the sequences are already sorted, the intersection can be done in time proportional to the aggregate length of the sequences. We list a few optimizations below.

- *Compression*—compressing the inverted index helps both with the ability to index more documents as well as memory locality (fewer cache misses). Since we are storing sorted sequences, one way of compressing is to use delta compression where we only store the difference between the successive entries. The deltas can be represented in fewer bits.
- *Caching*—the distribution of queries is usually fairly skewed and it helps a great deal to cache the results of some of the most frequent queries.
- *Frequency-based optimization*—since search results often do not need to return every document that matches (only top ten or so), only a fraction of highest quality documents can be used to answer most of the queries. This means that we can make two inverted indices, one with the high quality documents that stays in RAM and one with the remaining documents that stays on disk. This way if we can keep the number of queries that require the secondary index to a small enough number, then we can still maintain a reasonable throughput and latency.
- *Intersection order*—since the total intersection time depends on the total size of sequences, it would make sense to intersect the words with smaller sets first. For example, if we are looking for "USA GDP 2009", it would make sense to intersect the lists for GDP and 2009 before trying to intersect the sequence for USA.

We could also build a multilevel index to improve accuracy on documents. For a high priority web page, we can decompose the page into paragraphs and sentences, which are indexed individually. That way the intersections for the words might be within the same context. We can pick results with closer index values from these sequences. See the sorted array intersection problem 13.1 on Page 182 and digest problem 12.6 on Page 168 for related issues.

20.8 Implement PageRank

The PageRank algorithm assigns a rank to a web page based on the number of "important" pages that link to it. The algorithm essentially amounts to the following:

(1.) Build a matrix A based on the hyperlink structure of the web. Specifically, $A_{ij} = \frac{1}{d_j}$ if page j links to page i; here d_j is the number of distinct pages linked from page j.

(2.) Find X satisfying $X = \epsilon[\mathbf{1}] + (1-\epsilon)AX$. Here ϵ is a constant, e.g., $\frac{1}{7}$, and $[\mathbf{1}]$ represents a column vector of 1s. The value $X[i]$ is the rank of the ith page.

The most commonly used approach to solving the above equation is to start with a value of X, where each component is $\frac{1}{n}$ (where n is the number of pages) and then perform the following

iteration: $X_k = \epsilon[\mathbf{1}] + (1 - \epsilon)AX_{k-1}$. The iteration stops when the X_k converges, i.e., the difference from X_k to X_{k+1} is less than a specified threshold.

Design a system that can compute the ranks of ten billion web pages in a reasonable amount of time.

Hint: This must be performed on an ensemble of machines. The right data structures will simplify the computation.

Solution: Since the web graph can have billions of vertices and it is mostly a sparse graph, it is best to represent the graph as an adjacency list. Building the adjacency list representation of the graph may require a significant amount of computation, depending upon how the information is collected. Usually, the graph is constructed by downloading the pages on the web and extracting the hyperlink information from the pages. Since the URL of a page can vary in length, it is often a good idea to represent the URL by a hash code.

The most expensive part of the PageRank algorithm is the repeated matrix multiplication. Usually, it is not possible to keep the entire graph information in a single machine's RAM. Two approaches to solving this problem are described below.

- Disk-based sorting—we keep the column vector X in memory and load rows one at a time. Processing Row i simply requires adding $A_{i,j}X_j$ to X_j for each j such that $A_{i,j}$ is not zero. The advantage of this approach is that if the column vector fits in RAM, the entire computation can be performed on a single machine. This approach is slow because it uses a single machine and relies on the disk.
- Partitioned graph—we use n servers and partition the vertices (web pages) into n sets. This partition can be computed by partitioning the set of hash codes in such a way that it is easy to determine which vertex maps to which machine. Given this partitioning, each machine loads its vertices and their outgoing edges into RAM. Each machine also loads the portion of the PageRank vector corresponding to the vertices it is responsible for. Then each machine does a local matrix multiplication. Some of the edges on each machine may correspond to vertices that are owned by other machines. Hence the result vector contains nonzero entries for vertices that are not owned by the local machine. At the end of the local multiplication it needs to send updates to other hosts so that these values can be correctly added up. The advantage of this approach is that it can process arbitrarily large graphs.

PageRank runs in minutes on a single machine on the graph consisting of the multi-million pages that constitute Wikipedia. It takes roughly 70 iterations to converge on this graph. Anecdotally, PageRank takes roughly 200 iterations to converge on the graph consisting of the multi-billion pages that constitute the World-Wide Web.

When operating on a graph the scale of the webgraph, PageRank must be run on many thousands of machines. In such situations, it is very likely that a machine will fail, e.g., because of a defective power supply. The widely used Map-Reduce framework handles efficient parallelization as well as fault-tolerance. Roughly speaking, fault-tolerance is achieved by replicating data across a distributed file system, and having the master reassign jobs on machines that are not responsive.

20.9 Design TeraSort and PetaSort

Modern datasets are huge. For example, it is estimated that a popular social network contains over two trillion distinct items.

How would you sort a billion 1000 byte strings? How about a trillion 1000 byte strings?

Hint: Can a trillion 1000 byte strings fit on one machine?

Solution: A billion 1000 byte strings cannot fit in the RAM of a single machine, but can fit on the hard drive of a single machine. Therefore, one approach is to partition the data into smaller blocks that fit in RAM, sort each block individually, write the sorted block to disk, and then combine the sorted blocks. The sorted blocks can be merged using for example Solution 10.1 on Page 134. The UNIX `sort` program uses these principles when sorting very large files, and is faster than direct implementations of the merge-based algorithm just described.

If the data consists of a trillion 1000 byte strings, it cannot fit on a single machine—it must be distributed across a cluster of machines. The most natural interpretation of sorting in this scenario is to organize the data so that lookups can be performed via binary search. Sorting the individual datasets is not sufficient, since it does not achieve a global ordering—lookups entail a binary search on each machine. The straightforward solution is to have one machine merge the sorted datasets, but then that machine will become the bottleneck.

A solution which does away with the bottleneck is to first reorder the data so that the ith machine stores strings in a range, e.g., Machine 3 is responsible for strings that lie between *daily* and *ending*. The range-to-machine mapping R can be computed by sampling the individual files and sorting the sampled values. If the sampled subset is small enough, it can be sorted by a single machine. The techniques in Solution 11.8 on Page 153 can be used to determine the ranges assigned to each machine. Specifically, let A be the sorted array of sampled strings. Suppose there are M machines. Define $r_i = iA[n/M]$, where n is the length of A. Then Machine i is responsible for strings in the range $[r_i, r_{i+1})$. If the distribution of the data is known *a priori*, e.g., it is uniform, the sampling step can be skipped.

The reordering can be performed in a completely distributed fashion by having each machine route the strings it begins with to the responsible machines.

After reordering, each machine sorts the strings it stores. Consequently queries such as lookups can be performed by using R to determine which individual machine to forward the lookup to.

20.10 Implement distributed throttling

You have n machines ("crawlers") for downloading the entire web. The responsibility for a given URL is assigned to the crawler whose ID is Hash(URL) mod n. Downloading a web page takes away bandwidth from the web server hosting it.

Implement crawling under the constraint that in any given minute your crawlers do not request more than b bytes from any website.

Hint: Use a server to coordinate the crawl.

Solution: This problem, as posed, is ambiguous.

- Since we usually download one file in one request, if a file is greater than b bytes, there is no way we can meet the constraint of serving fewer than b bytes every minute, unless we can work with the lower layers of the network stack such as the transport layer or the network layer. Often the system designer could look at the distribution of file sizes and conclude that this problem happens so infrequently that we do not care. Alternatively, we may choose to download no more than the first b bytes of any file.
- Given that the host's bandwidth is a resource for which there could be contention, one important design choice to be made is how to resolve a contention. Do we let requests get served in first-come first-served order or is there a notion of priority? Often crawlers have a built-in notion of priority based on how important the document is to the users or how fresh the current copy is.

One way of doing this could be to maintain a permission server with which each crawler checks to see if it is okay to hit a particular host. The server can keep an account of how many bytes have been downloaded from the server in the last minute and not permit any crawler to hit the server if we are already close to the quota. If we do not care about priority, then we can keep the interface synchronous where a server requests permission to download a file and it immediately gets approved or denied. If we care about priorities, then the server may enqueue the request and inform the crawler when the file is available for download. The queues at the permission server may be based on priorities.

If the permission server becomes a bottleneck, we can use multiple permission servers such that the responsibility of a given host is decided by applying a hash function to the host name and assigning it to a particular server based on the hash code.

20.11 Design a scalable priority system

Maintaining a set of prioritized jobs in a distributed system can be tricky. Applications include a search engine crawling web pages in some prioritized order, as well as event-driven simulation in molecular dynamics. In both cases the number of jobs is in the billions and each has its own priority.

Design a system for maintaining a set of prioritized jobs that implements the following API: (1.) insert a new job with a given priority; (2.) delete a job; (3.) find the highest priority job. Each job has a unique ID. Assume the set cannot fit into a single machine's memory.

Hint: How would you partition jobs across machines? Is it always essential to operate on the highest priority job?

Solution: If we have enough RAM on a single machine, the most simple solution would be to maintain a min-heap where entries are ordered by their priority. An additional hash table can be used to map jobs to their corresponding entry in the min-heap to make deletions fast.

A more scalable solution entails partitioning the problem across multiple machines. One approach is to apply a hash function to the job ids and partition the resulting hash codes into ranges, one per machine. Insert as well as delete require communication with just one server. To do extract-min, we send a lookup minimum message to all the machines, infer the min from their responses, and then delete it.

At a given time many clients may be interested in the highest priority event, and it is challenging to distribute this problem well. If many clients are trying to do this operation at the same time, we

may run into a situation where most clients will find that the min event they are trying to extract has already been deleted. If the throughput of this service can be handled by a single machine, we can make one server solely responsible for responding to all the requests. This server can prefetch the top hundred or so events from each of the machines and keep them in a heap.

In many applications, we do not need strong consistency guarantees. We want to spend most of our resources taking care of the highest priority jobs. In this setting, a client could pick one of the machines at random, and request the highest priority job. This would work well for the distributed crawler application. It is not suited to event-driven simulation because of dependencies.

Another other issue to consider is resilience: if a node fails, all list of work on that node fails as well. It is better to have nodes to contain overlapped lists and the dispatching node in this case will handle duplicates. The lost of a node shouldn't result in full re-hashing—the replacement node should handle only new jobs. Consistent hashing can be used to achieve this.

A front-end caching server can become a bottleneck. This can be avoided by using replication, i.e., multiple servers which duplicate each other. There could be possible several ways to coordinate them: use non-overlapping lists, keep a blocked job list, return a random job from the jobs with highest priority.

20.12 Create photomosaics

A photomosaic is built from a collection of images called "tiles" and a target image. The photomosaic is another image which approximates the target image and is built by juxtaposing the tiles. Quality is defined by human perception.

Design a program that produces high quality mosaics with minimal compute time.

Hint: How would you define the distance between two images?

Solution: A good way to begin is to partition the image into $s \times s$-sized squares, compute the average color of each such image square, and then find the tile that is closest to it in the color space. Distance in the color space can be the $L2$-distance over the Red-Green-Blue (RGB) intensities for the color. As you look more carefully at the problem, you might conclude that it would be better to match each tile with an image square that has a similar structure. One way could be to perform a coarse pixelization (2×2 or 3×3) of each image square and finding the tile that is "closest" to the image square under a distance function defined over all pixel colors. In essence, the problem reduces to finding the closest point from a set of points in a k-dimensional space.

Given m tiles and an image partitioned into n squares, then a brute-force approach would have $O(mn)$ time complexity. You could improve on this by first indexing the tiles using an appropriate search tree. You can also run the matching in parallel by partitioning the original image into subimages and searching for matches on the subimages independently.

20.13 Implement Mileage Run

Airlines often give customers who fly frequently with them a "status". This status allows them early boarding, more baggage, upgrades to executive class, etc. Typically, status is a function of miles flown in the past twelve months. People who travel frequently by air sometimes want to take

a round trip flight simply to maintain their status. The destination is immaterial—the goal is to minimize the cost-per-mile (cpm), i.e., the ratio of dollars spent to miles flown.

Design a system that will help its users find mileage runs.

Hint: Partition the implied features into independent tasks.

Solution: There are two distinct aspects to the design. The first is the user-facing portion of the system. The second is the server backend that gets flight-price-distance information and combines it with user input to generate the alerts.

We begin with the user-facing portion. For simplicity, we illustrate it with a web-app, with the realization that the web-app could also be written as a desktop or mobile app. The web-app has the following components: a login page, a manage alerts page, a create an alert page, and a results page. For such a system we would like defer to a single-sign-on login service such as that provided by Google or Facebook. The management page would present login information, a list of alerts, and the ability to create an alert.

One reasonable formulation of an alert is that it is an origin city, a target cpm, and optionally, a date or range of travel dates. The results page would show flights satisfying the constraints. Note that other formulations are also possible, such as how frequently to check for flights, a set of destinations, a set of origins, etc.

The classical approach to implement the web-app front end is through dynamically generated HTML on the server, e.g., through Java Server Pages. It can be made more visually appealing and intuitive by making appropriate use of cascaded style sheets, which are used for fonts, colors, and placements. The UI can be made more efficient through the use of Javascript to autocomplete common fields, and make attractive date pickers.

Modern practice is to eschew server-side HTML generation, and instead have a single-page application, in which Javascript reads and writes JavaScript Object Notation (JSON) objects to the server, and incrementally updates the single-page based. The AngularJS framework supports this approach.

The web-app backend server has four components: gathering flight data, matching user-generated alerts to this data, persisting data and alerts, and generating the responses to browser initiated requests.

Flight data can be gathered via "scraping" or by subscribing to a flight data service. Scraping refers to extraction of data from a website. It can be quite involved—some of the issues are parsing the results from the website, filling in form data, and running the Javascript that often populates the actual results on a page. Selenium is a Java library that can programmatically interface to the Firefox browser, and is appropriate for scraping sites that are rich in Javascript. Most flight data services are paid. ITA software provides a very widely used paid aggregated flight data feed service. The popular Kayak site provides an Extensible Markup Language (XML) feed of recently discovered fares, which can be a good free alternative. Flight data does not include the distance between airports, but there are websites which return the distance between airport codes which can be used to generate the cpm for a flight.

There are a number of common web application frameworks—essentially libraries that handle many common tasks—that can be used to generate the server. Java and Python are very commonly used for writing the backend for web applications.

Persistence of data can be implemented through a database. Most web application frameworks provide support for automating the process of reading and writing objects from and to a database. Finally, web application frameworks can route incoming HTTP requests to appropriate code—this is through a configuration file matching URLs to methods. The framework provides convenience methods for accessing HTTP fields and writing results. Frameworks also provide HTTP templating mechanisms, wherein developers intersperse HTML with snippets of code that dynamically add content to the HTML.

Web application frameworks typically implement cron functionality, wherein specified functions are executed at a regular interval. This can be used to periodically scrape data and check if the condition of an alert is matched by the data.

Finally, the web app can be deployed via a platform-as-a-service such as Amazon Web Services and Google App Engine.

20.14 Implement Connexus

How would you design Connexus, a system by which users can share pictures? Address issues around access control (public, private), picture upload, organizing pictures, summarizing sets of pictures, allowing feedback, and displaying geographical and temporal information for pictures.

Hint: Think about the UI, in particular UI widgets that make for an engaging product.

Solution: There are three aspects to Connexus. The first is the server backend, used to store images, and meta-data, such as author, comments, hashtags, GPS coordinates, etc., as well as run `cron` jobs to identify trending streams. The technology for this is similar to Mileage Run (Solution 20.13 on the previous page), with the caveat that a database is not suitable for storing large binary objects such as images, which should be stored on a local or remote file system. (The database will hold references to the file.)

The web UI is also similar to Mileage Run, with a login page, a management page, and pages for displaying images. Images can be grouped based on a concept of a stream, with comment boxes annotating streams. Facebook integration would make it easier to share links to streams, and post new images as status updates. Search capability and discussion boards also enhance the user experience.

Some UI features that are especially appealing are displaying images by location on a zoomable map, slider UI controls to show subsets of images in a stream based on the selected time intervals, a file upload dialog with progress measures, support for multiple simultaneous uploads, and drag-and-drop upload. All these UI widgets are provided by, for example, the jQuery-UI Javascript library. (This library also makes the process of creating autocompletion on text entry fields trivial.)

Connexus is an application that begs for a mobile client. A smartphone provides a camera, location information, and push notifications. These can be used to make it easier to create and immediately upload geo-tagged images, find nearby images, and be immediately notified on updates and comments. The two most popular mobile platforms, iOS and Android, have APIs rich in UI widgets and media access. Both can use serialization formats such as JSON or protocol buffers to communicate with the server via remote procedure calls layered over HTTP.

20.15 Design an online advertising system

Jingle, a search engine startup, has been very successful at providing a high-quality Internet search service. A large number of customers have approached Jingle and asked it to display paid advertisements for their products and services alongside search results.

Design an advertising system for Jingle.

Hint: Consider the stakeholders separately.

Solution: Reasonable goals for such a system include

- providing users with the most relevant ads,
- providing advertisers the best possible return on their investment, and
- minimizing the cost and maximizing the revenue to Jingle.

Two key components for such a system are:

- The front-facing component, which advertisers use to create advertisements, organize campaigns, limit when and where ads are shown, set budgets, and create performance reports.
- The ad-serving system, which selects which ads to show on the searches.

The front-facing system can be a fairly conventional web application, i.e., a set of web pages, middleware that responds to user requests, and a database. Key features include:

- User authentication—a way for users to create accounts and authenticate themselves. Alternatively, use an existing single sign-on login service, e.g., Facebook or Google.
- User input—a set of form elements to let advertisers specify ads, advertising budget, and search keywords to bid on.
- Performance reports—a way to generate reports on how the advertiser's money is being spent.
- Customer service—even the best of automated systems require occasional human interaction, e.g., ways to override limits on keywords. This requires an interface for advertisers to contact customer service representatives, and an interface for those representatives to interact with the system.

The whole front-end system can be built using, for example, HyperText Markup Language (HTML) and JavaScript. A commonly used approach is to use a LAMP stack on the server-side: Linux as the OS, Apache as the HTTP server, MySQL as the database software, and PHP for the application logic.

The ad-serving system is less conventional. The ad-serving system would build a specialized data structure, such as a decision tree, from the ads database. It chooses ads from the database of ads based on their "relevance" to the search. In addition to keywords, the ad-serving systems can use knowledge of the user's search history, how much the advertiser is willing to pay, the time of day, user locale, and type of browser. Many strategies can be envisioned here for estimating relevance, such as, using information retrieval or machine learning techniques that learn from past user interactions.

The ads could be added to the search results by embedding JavaScript in the results page. This JavaScript pulls in the ads from the ad-serving system directly. This helps isolate the latency of serving search results from the latency of serving ad results.

There are many more issues in such a system: making sure there are no inappropriate images using an image recognition API; using a link verification to check if keywords really corresponds to a real site; serving up images from a content-delivery network; and having a fallback advertisement to show if an advertisement cannot be found.

20.16 Design a recommendation system

Jingle wants to generate more page views on its news site. A product manager has the idea to add to each article a sidebar of clickable snippets from articles that are likely to be of interest to someone reading the current article.

Design a system that automatically generates a sidebar of related articles.

Hint: This problem can be solved with various degrees of algorithmic sophistication: none at all, simple frequency analysis, or machine learning.

Solution: The key technical challenge in this problem is to come up with the list of articles—the code for adding these to a sidebar is trivial.

One suggestion might be to add articles that have proved to be popular recently. Another is to have links to recent news articles. A human reader at Jingle could tag articles which he believes to be significant. He could also add tags such as finance, sports, and politics, to the articles. These tags could also come from the HTML meta-tags or the page title.

We could also provide randomly selected articles to a random subset of readers and see how popular these articles prove to be. The popular articles could then be shown more frequently.

On a more sophisticated level, Jingle could use automatic textual analysis, where a similarity is defined between pairs of articles—this similarity is a real number and measures how many words are common to the two. Several issues come up, such as the fact that frequently occurring words such as "for" and "the" should be ignored and that having rare words such as "arbitrage" and "diesel" in common is more significant than having say, "sale" and "international".

Textual analysis has problems, such as the fact that two words may have the same spelling but completely different meanings (anti-virus means different things in the context of articles on acquired immune deficiency syndrome (AIDS) and computer security). One way to augment textual analysis is to use collaborative filtering—using information gleaned from many users. For example, by examining cookies and timestamps in the web server's log files, we can tell what articles individual users have read. If we see many users have read both A and B in a single session, we might want to recommend B to anyone reading A. For collaborative filtering to work, we need to have many users.

20.17 Design an optimized way of distributing large files

Jingle is developing a search feature for breaking news. New articles are collected from a variety of online news sources such as newspapers, bulletin boards, and blogs, by a single lab machine at Jingle. Every minute, roughly one thousand articles are posted and each article is 100 kilobytes.

Jingle would like to serve these articles from a data center consisting of a 1000 servers. For performance reasons, each server should have its own copy of articles that were recently added. The data center is far away from the lab machine.

Design an efficient way of copying one thousand files each 100 kilobytes in size from a single lab server to each of 1000 servers in a distant data center.

Hint: Exploit the data center.

Solution: Assume that the bandwidth from the lab machine is a limiting factor. It is reasonable to first do trivial optimizations, such as combining the articles into a single file and compressing this file.

Opening 1000 connections from the lab server to the 1000 machines in the data center and transferring the latest news articles is not feasible since the total data transferred will be approximately 100 gigabytes (without compression).

Since the bandwidth between machines in a data center is very high, we can copy the file from the lab machine to a single machine in the data center and have the machines in the data center complete the copy. Instead of having just one machine serve the file to the remaining 999 machines, we can have each machine that has received the file initiate copies to the machines that have not yet received the file. In theory, this leads to an exponential reduction in the time taken to do the copy.

Several additional issues have to be dealt with. Should a machine initiate further copies before it has received the entire file? (This is tricky because of link or server failures.) How should the knowledge of machines which do not yet have copies of the file be shared? (There can be a central repository or servers can simply check others by random selection.) If the bandwidth between machines in a data center is not a constant, how should the selections be made? (Servers close to each other, e.g., in the same rack, should prefer communicating with each other.)

Finally, it should be mentioned that there are open source solutions to this problem, e.g., Unison and BitTorrent, which would be a good place to start.

20.18 Design the World Wide Web

Design the World Wide Web. Specifically, describe what happens when you enter a URL in a browser address bar, and press return.

Hint: Follow the flow of information.

Solution: At the network level, the browser extracts the domain name component of the URL, and determines the IP address of the server, e.g., through a call to a Domain Name Server (DNS), or a cache lookup. It then communicates using the HTTP protocol with the server. HTTP itself is built on top of TCP/IP, which is responsible for routing, reassembling, and resending packets, as well as controlling the transmission rate.

The server determines what the client is asking for by looking at the portion of the URL that comes after the domain name, and possibly also the body of the HTTP request. The request may be for something as simple a file, which is returned by the webserver; HTTP spells out a format by which the type of the returned file is specified. For example, the URL http://go.com/imgs/abc.png may encode a request for the file whose hierarchical name is imgs/abc.png relative to a base directory specified at configuration to the web server.

The URL may also encode a request to a service provided by the web server. For example, http://go.com/lookup/flight?num=UA37,city=AUS is a request to the lookup/flight service, with an argument consisting of two attribute-value pair. The service could be implemented in many ways, e.g., Java code within the server, or a Common Gateway Interface (CGI) script written in Perl. The service generates a HTTP response, typically HTML, which is then returned to the browser. This response could encode data which is used by scripts running in the browser. Common data formats include JSON and XML.

The browser is responsible for taking the returned HTML and displaying it on the client. The rendering is done in two parts. First, a parse tree (the DOM) is generated from the HTML, and then a rendering library "paints" the screen. The returned HTML may include scripts written in JavaScript. These are executed by the browser, and they can perform actions like making requests and updating the DOM based on the responses—this is how a live stock ticker is implemented. Styling attributes (CSS) are commonly used to customize the look of a page.

Many more issues exist on both the client and server side: security, cookies, HTML form elements, HTML styling, and handlers for multi-media content, to name a few.

20.19 Estimate the hardware cost of a photo sharing app

Estimate the hardware cost of the server hardware needed to build a photo sharing app used by every person on the earth.

Hint: Use variables to denote quantities and costs, and relate them with equations. Then fill in reasonable values.

Solution: The cost is a function of server CPU and RAM resources, storage, and bandwidth, as well as the number and size of the images that are uploaded each day. We will create an estimate based on unit costs for each of these. We assume a distributed architecture in which images are spread ("sharded") across servers.

Assume each user uploads i images each day with an average size of s bytes, and that each image is viewed v times. After d days, the storage requirement is $isdN$, where N is the number of users. Assuming $v \gg 1$, i.e., most images are seen many times, the server cost is dominated by the time to serve the images. The servers are required to serve up Niv images and $Nivs$ bytes each day. Assuming a server can handle h HTTP requests per second and has an outgoing bandwidth of b bytes per second, the number of required servers is $\max(Niv/Th, Nivs/Tb)$, where T is the number of seconds in a day.

Reasonable values for N, i, s, and v are 10^{10}, 10, 10^5, and 100. Reasonable values for h and b are 10^4 and 10^8. There are approximately 10^5 seconds in a day. Therefore the number of servers required is $\max((10^{10} \times 10 \times 100)/(10^5 \times 10^4), (10^{10} \times 10 \times 100 \times 10^5)/(10^5 \times 10^8)) = 10^5$. Each server would cost \$1000, so the total cost would be of the order of 100 million dollars.

Storage costs are approximately \$0.1 per gigabyte, and we add $Nis = 10^{10} \times 10 \times 10^5$ bytes each day, so each day we need to add a million dollars worth of storage.

The above calculation leaves out many costs, such as electricity, cooling, and network. It also neglects computations such as computing trending data and spam analysis. Furthermore, there is no measure of the cost of redundancy, such as replicated storage, or the ability to handle nonuniform loads. Nevertheless, it is a decent starting point. What is remarkable is the fact that the entire world can be connected through images at a very low cost—pennies per person.