Chapter 11. Batch Processing

> *A system cannot be successful if it is too strongly influenced by a single person. Once the initial design is complete and fairly robust, the real test begins as people with many different viewpoints undertake their own experiments.*
>
> —Donald Knuth

Much of this book so far has talked about *requests* and *queries*, and the corresponding *responses* or *results*. This style of data processing is assumed in many modern data systems: you ask for something, or you send an instruction, and the system tries to give you an answer as quickly as possible.

A web browser requesting a page, a service calling a remote API, databases, caches, search indexes, and many other systems work this way. We call these *online systems*. Response time is usually their primary measure of performance, and they often require fault tolerance to ensure high availability.

However, sometimes you need to run a bigger computation or process larger amounts of data than you can do in an interactive request. Maybe you need to train an AI model, or transform lots of data from one form into another, or compute analytics over a very large dataset. We call these tasks *batch processing* jobs, or sometimes *offline systems*.

A batch processing job takes some input data (which is read-only), and produces some output data (which is generated from scratch every time the job runs). It typically does not mutate data in the way a read/write transaction would. The output is therefore *derived* from the input (as discussed in "Systems of Record and Derived Data"): if you don't like the output, you can just delete it, adjust the job logic, and run it again. By treating inputs as immutable and avoiding side effects (such as writing to external databases), batch jobs not only achieve good performance but also have other benefits:

- If you introduce a bug into the code and the output is wrong or corrupted, you can simply roll back to a previous version of the code and rerun the job, and the output will be correct again. Or, even simpler, you can keep the old output in a different directory and simply switch back to it. Most object stores and open table formats (see "Cloud Data Warehouses") support this feature, which is known as *time travel*. Most databases with read-write transactions do not have this property: if you deploy buggy code that writes bad data to the database, then rolling back the code will do nothing to fix the data in the database. The idea of being able to recover from buggy code has been called *human fault tolerance* [1].
- As a consequence of this ease of rolling back, feature development can proceed more quickly than in an environment where mistakes could mean irreversible damage. This principle of *minimizing irreversibility* is beneficial for Agile software development [2].
- The same set of files can be used as input for various different jobs, including monitoring jobs that calculate metrics and evaluate whether a job's output has the expected characteristics (for example, by comparing it to the output from the previous run and measuring discrepancies).
- Batch processing frameworks make efficient use of computing resources. Even though it's possible to batch process data using online data systems such as OLTP databases and applications servers, doing so can be much more expensive in terms of the resources required.

Batch data processing also presents challenges. With most frameworks, output can only be processed by other jobs after the whole job finishes. Batch processing can also be inefficient: any change to input data—even a single byte—means the batch job must reprocess the entire input dataset. Despite these limitations, batch processing has proven useful in a wide range of use cases, which we'll revisit in "Batch Use Cases".

A batch job may take a long time to run: minutes, hours, or even days. Jobs may be scheduled to run periodically (for example, once per day). The primary measure of performance is usually throughput: how much data the job can process per unit time. Some batch systems handle faults by simply aborting and restarting the whole job, while others have fault tolerance so that a job can complete successfully despite some of its nodes crashing.

NOTE

An alternative to batch processing is *stream processing*, in which the job doesn't finish running when it has processed the input, but instead continues watching the input and processes changes in the input shortly after they happen. We will turn to stream processing in Chapter 12.

The boundary between online and batch processing systems is not always clear: a long-running database query looks quite like a batch process. But batch processing also has some particular characteristics that make it a useful building block for building reliable, scalable, and maintainable applications. For example, it often plays a role in *data integration*, i.e., composing multiple data systems to achieve things that one system alone cannot do. ETL, as discussed in "Data Warehousing", is an example of this.

Modern batch processing has been heavily influenced by MapReduce, a batch processing algorithm that was published by Google in 2004 [3], and subsequently implemented in various open source data systems, including Hadoop, CouchDB, and MongoDB. MapReduce is a fairly low-level programming model, and less sophisticated than the parallel query execution engines found, for example, in data warehouses [4, 5]. When it was new, MapReduce was a step forward in terms of the scale of processing that could be achieved on commodity hardware, but now it is largely obsolete, and no longer used at Google [6, 7].

Batch processing today is more often done using frameworks such as Spark or Flink, or data warehouse query engines. Like MapReduce, they rely heavily on sharding (see Chapter 7) and parallel execution, but they have far more sophisticated caching and execution strategies. As these systems have matured, operational concerns have been largely solved, so focus has shifted toward usability. New processing models such as dataflow APIs, query languages, and DataFrame APIs are now widely supported. Job and workflow orchestration has also matured. Hadoop-centric workflow schedulers such as Oozie and Azkaban have been replaced with more generalized solutions such as Airflow, Dagster, and Prefect, which support a wide array of batch processing frameworks and cloud data warehouses.

Cloud computing has grown ubiquitous. Batch storage layers are shifting from distributed filesystems (DFSs) like HDFS, GlusterFS, and CephFS to object storage systems such as S3. Scalable cloud data warehouses like BigQuery and Snowflake are blurring the line between data warehouses and batch processing.

To build an intuition of what batch processing is about, we will start this chapter with an example that uses standard Unix tools on a single machine. We will then investigate how we can extend data processing to multiple machines in a distributed system. We will see that, much like an operating system, distributed batch processing frameworks have a scheduler and a filesystem. We will then explore various processing models that we use to write batch jobs. Finally, we discuss common batch processing use cases.

Batch Processing with Unix Tools

Say you have a web server that appends a line to a log file every time it serves a request. For example, using the nginx default access log format, one line of the log might look like this:

```
216.58.210.78 - - [27/Jun/2025:17:55:11 +0000] "GET /cs
200 3377 "https://martin.kleppmann.com/" "Mozilla/5.0 (
10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
```

(That is actually one line; it’s only broken onto multiple lines here for readability.) There’s a lot of information in that line. In order to interpret it, you need to look at the definition of the log format, which is as follows:

```
$remote_addr - $remote_user [$time_local] "$request"
```

```
    $status $body_bytes_sent "$http_referer" "$http_user_ag
```

So, this one line of the log indicates that on June 27, 2025, at 17:55:11 UTC, the server received a request for the file */css/typography.css* from the client IP address 216.58.210.78. The user was not authenticated, so `$remote_user` is set to a hyphen (`-`). The response status was 200 (i.e., the request was successful), and the response was 3,377 bytes in size. The web browser was Chrome 137, and it loaded the file because it was referenced in the page at the URL *https://martin.kleppmann.com/*.

Though log parsing might seem contrived, it's actually a critical part of many modern technology companies, and is used for everything from ad pipelines to payment processing. Indeed, it was a driving force behind the rapid adoption of MapReduce and the "big data" movement.

Simple Log Analysis

Various tools can take these log files and produce pretty reports about your website traffic, but for the sake of exercise, let's build our own, using basic Unix tools. For example, say you want to find the five most popular pages on your website. You can do this in a Unix shell as follows:

```
cat /var/log/nginx/access.log |
❶

  awk '{print $7}' |
❷

  sort             |
❸

  uniq -c          |
❹

  sort -r -n       |
❺

  head -n 5
❻
```

❶ Read the log file. (Strictly speaking, `cat` is unnecessary here, as the input file could be given directly as an argument to `awk`. However, the linear pipeline is more apparent when written like this.)

2. Split each line into fields by whitespace, and output only the seventh such field from each line, which happens to be the requested URL. In our example line, this request URL is */css/typography.css*.

3. Alphabetically `sort` the list of requested URLs. If some URL has been requested *n* times, then after sorting, the file contains the same URL repeated *n* times in a row.

4. The `uniq` command filters out repeated lines in its input by checking whether two adjacent lines are the same. The `-c` option tells it to also output a counter: for every distinct URL, it reports how many times that URL appeared in the input.

5. The second `sort` sorts by the number (`-n`) at the start of each line, which is the number of times the URL was requested. It then returns the results in reverse (`-r`) order, i.e. with the largest number first.

6. Finally, `head` outputs just the first five lines (`-n 5`) of input, and discards the rest.

The output of that series of commands looks something like this:

```
4189 /favicon.ico
3631 /2016/02/08/how-to-do-distributed-locking.html
2124 /2020/11/18/distributed-systems-and-elliptic-curve
1369 /
 915 /css/typography.css
```

Although the preceding command line likely looks a bit obscure if you're unfamiliar with Unix tools, it is incredibly powerful. It will process gigabytes of log files in a matter of seconds, and you can easily modify the analysis to suit your needs. For example, if you want to omit CSS files from the report, change the `awk` argument to `'$7 !~ /\.css$/ {print $7}'` . If you want to count top client IP addresses instead of top pages, change the `awk` argument to `'{print $1}'` . And so on.

We don't have space in this book to explore Unix tools in detail, but they are very much worth learning about. Surprisingly many data analyses can be done in a few minutes using some combination of `awk` , `sed` , `grep` , `sort` , `uniq` , and `xargs` , and they perform surprisingly well [8].

Chain of Commands Versus Custom Program

Instead of the chain of Unix commands, you could write a simple program to do the same thing. For example, in Python, it might look something like this:

```
from collections import defaultdict

counts = defaultdict(int)
❶


with open('/var/log/nginx/access.log', 'r') as file:
    for line in file:
        url = line.split()[6]
❷

        counts[url] += 1
❸


top5 = sorted(((count, url) for url, count in counts.it
❹


for count, url in top5:
❺

    print(f"{count} {url}")
```

❶ `counts` is a hash table that keeps a counter for the number of times we’ve seen each URL. A counter is zero by default.

❷ From each line of the log, we take the URL to be the seventh whitespace-separated field (the array index is 6 because Python’s arrays are zero-indexed).

❸ Increment the counter for the URL in the current line of the log.

❹ Sort the hash table contents by counter value (descending), and take the top five entries.

❺ Print out those top five entries.

This program is not as concise as the chain of Unix pipes, but it’s fairly readable, and which of the two you prefer is partly a matter of taste. However, besides the superficial syntactic differences between the two, there is a big difference in the execution flow, which becomes apparent if you run this analysis on a large file.

Sorting Versus In-memory Aggregation

The Python script keeps an in-memory hash table of URLs, where each URL is mapped to the number of times it has been seen. The Unix pipeline example does not have such a hash table, but instead relies on sorting a list of URLs in which multiple occurrences of the same URL are simply repeated.

Which approach is better? It depends how many different URLs you have. For most small to mid-sized websites, you can probably fit all distinct URLs, and a counter for each URL, in (say) 1 GB of memory. In this example, the *working set* of the job (the amount of memory to which the job needs random access) depends only on the number of distinct URLs: if there are a million log entries for a single URL, the space required in the hash table is still just one URL plus the size of the counter. If this working set is small enough, an in-memory hash table works fine—even on a laptop.

On the other hand, if the job's working set is larger than the available memory, the sorting approach has the advantage that it can make efficient use of disks. It's the same principle as we discussed in "Log-Structured Storage": chunks of data can be sorted in memory and written out to disk as segment files, and then multiple sorted segments can be merged into a larger sorted file. Mergesort has sequential access patterns that perform well on disks (see "Sequential Versus Random Writes on SSDs").

The `sort` utility in GNU Coreutils (Linux) automatically handles larger-than-memory datasets by spilling to disk, and automatically parallelizes sorting across multiple CPU cores [9]. This means that the simple chain of Unix commands we saw earlier easily scales to large datasets, without running out of memory. The bottleneck is likely to be the rate at which the input file can be read from disk.

A limitation of Unix tools is that they run only on a single machine. Datasets that are too large to fit in memory or local disk present a problem—and that's where distributed batch processing frameworks come in.

Batch Processing in Distributed Systems

The machine that runs our Unix tool example has a number of components that work together to process the log data:

- Storage devices that are accessed through the operating system's filesystem interface.
- A scheduler that determines when processes get to run, and how to allocate CPU resources to them.
- A series of Unix programs whose `stdin` and `stdout` are connected together by pipes.

These same components exist in distributed data processing frameworks. In fact, you can think of a distributed processing framework as a distributed operating system; they have filesystems, job schedulers, and programs that send data to each other through the filesystem or other communication channels.

Distributed Filesystems

The filesystem provided by your operating system is composed of several layers:

- At the lowest level, block device drivers speak directly to the disk, and allow the layers above to read and write raw blocks.
- Above the block layer sits a page cache that keeps recently accessed blocks in memory for faster access.
- The block API is wrapped in a filesystem layer that breaks up large files into blocks, and tracks file metadata such as inodes, directories, and files. ext4 and XFS are two common implementations on Linux, for example.
- Finally, the operating system exposes different filesystems to applications through a common API called the virtual file system (VFS). The VFS is what allows applications to read and write in a standard way regardless of the underlying filesystem.

Distributed filesystems work in much the same way. Files are broken up into blocks, which are distributed across many machines. DFS blocks are typically much larger than local blocks: HDFS (Hadoop Distributed File System) defaults to 128MB, while JuiceFS and many object stores use 4MB blocks—

much larger than ext4’s 4096 bytes. Larger blocks mean less metadata to keep track of, which makes a big difference on petabyte-sized datasets. Larger blocks also lower the overhead of seeking to a block relative to reading it.

Most physical storage devices can’t write partial blocks, so operating systems require writes to use an entire block even if the data doesn’t take up the whole block. Since distributed filesystems have larger blocks and are usually implemented on top of operating system filesystems, they don’t have this requirement. For example, a 900MB file stored with 128MB blocks would have 7 blocks that use 128MB and 1 block that uses 4MB.

DFS blocks are read by making network requests to a machine in the cluster that stores the block. Each machine runs a daemon, exposing an API that allows remote processes to read and write blocks as files on its local filesystem. HDFS refers to these daemons as DataNodes, while GlusterFS calls them glusterfsd processes. We’ll call them *data nodes* in this book.

Distributed filesystems also implement the distributed equivalent of a page cache. Since DFS blocks are stored as files on data nodes, reads and writes go through each data node’s operating system, which includes an in-memory page cache. This keeps frequently read data blocks in-memory on the data nodes. Some distributed filesystems also implement more caching tiers such as the client-side and local-disk caching found in JuiceFS.

Filesystems such as ext4 and XFS keep track of storage metadata including free space, file block locations, directory structures, permission settings, and more. Distributed filesystems also need a way to track file locations spread across machines, permission settings, and so on. Hadoop has a service called the NameNode, which maintains metadata for the cluster. DeepSeek’s 3FS has a metadata service that persists its data to a key-value store such as FoundationDB.

Above the filesystem sits the VFS. A close analogue in batch processing is a distributed filesystem’s protocol. Distributed filesystems must expose a protocol or interface so that batch processing systems can read and write files. This protocol acts as a pluggable interface: any DFS may be used so long as it implements the protocol. For example, Amazon S3’s API has been widely adopted by other storage systems such as MinIO, Cloudflare’s R2, Tigris, Backblaze’s B2, and many others. Batch processing systems with S3 support can use any of these storage systems.

Some DFSs implement POSIX-compliant filesystems that appear to the operating system's VFS like any other filesystem. Filesystem in Userspace (FUSE) or the Network File System (NFS) protocol are often used to integrate into the VFS. NFS is perhaps the most well known distributed filesystem protocol. The protocol was originally developed to allow multiple clients to read and write data on a single server. More recently, filesystems such as AWS's Elastic File System (EFS) and Archil provide NFS-compatible distributed filesystem implementations that are far more scalable. NFS clients still connect to one end point, but underneath, these systems communicate with distributed metadata services and data nodes to read and write data.

DISTRIBUTED FILESYSTEMS AND NETWORK STORAGE

Distributed filesystems are based on the *shared-nothing* principle (see "Shared-Memory, Shared-Disk, and Shared-Nothing Architecture"), in contrast to the shared-disk approach of *Network Attached Storage* (NAS) and *Storage Area Network* (SAN) architectures. Shared-disk storage is implemented by a centralized storage appliance, often using custom hardware and special network infrastructure such as Fibre Channel. On the other hand, the shared-nothing approach requires no special hardware, only computers connected by a conventional datacenter network.

Many distributed filesystems are built on commodity hardware, which is less expensive but has higher failure rates than enterprise-grade hardware. In order to tolerate machine and disk failures, file blocks are replicated on multiple machines. This also allows schedulers to more evenly distribute workloads since it can execute a task on any node that contains a replica of the task's input data. Replication may mean simply several copies of the same data on multiple machines, as in Chapter 6, or an *erasure coding* scheme such as Reed–Solomon codes, which allows lost data to be recovered with lower storage overhead than full replication [10, 11, 12]. The techniques are similar to RAID, which provides redundancy across several disks attached to the same machine; the difference is that in a distributed filesystem, file access and replication are done over a conventional datacenter network without special hardware.

Object Stores

Object storage services such as Amazon S3, Google Cloud Storage, Azure Blob Storage, and OpenStack Swift have become a popular alternative to distributed filesystems for batch processing jobs. In fact, the line between the two is somewhat blurry. As we saw in the previous section and "Databases Backed by Object Storage", Filesystem in Userspace (FUSE) drivers allow users to treat object stores such as S3 as a filesystem. Some DFS implementations such as JuiceFS and Ceph offer both object storage and filesystem APIs. However, their APIs, performance, and consistency guarantees are very different. Care must be taken when adopting such systems to make sure they behave as expected, even if they seem to implement the requisite APIs.

Each object in an object store has a URL such as `s3://my-photo-bucket/2025/04/01/birthday.png`. The host portion of the URL (`my-photo-bucket`) describes the bucket where objects are stored, and the part that follows is the object's *key* (`/2025/04/01/birthday.png` in our example). A bucket has a globally unique name, and each object's key must be unique within its bucket.

Object are read using a `get` call and written using a `put` call. Unlike files on a filesystem, objects are immutable once written. To update an object, it must be fully rewritten using a `put` call, similarly to a key-value store. Azure Blob Storage and S3 Express One Zone support appends, but most other stores do not. There are no file handle APIs with functions like `fopen` and `fseek`.

Objects may look as if they are organised into directories, which is somewhat confusing, since object stores do not have the concept of directories. The path structure is simply a convention, and the slashes are a part of the object's key. This convention allows you to perform something similar to a directory listing by requesting a list of objects with a particular prefix. However, listing objects by prefix is different from a filesystem directory listing in two ways:

- A prefix `list` operation behaves like recursive `ls -R` call on a Unix system: it returns all objects that start with the prefix—objects in subpaths are included.
- Empty directories are not possible: if you were to remove all objects underneath `s3://my-photo-bucket/2025/04/01`, then `01` would

no longer appear when we call `list` on `s3://my-photo-bucket/2025/04`. It is a common practice to create a zero-byte object as a way to represent an empty directory (e.g. creating an empty `s3://my-photo-bucket/2025/04/01` file to keep it present when all child objects are deleted).

DFS implementations often support many common filesystem operations such as hard links, symbolic links, file locking, and atomic renames. Such features are missing from object stores. Linking and locks are typically not supported, while renames are non-atomic; they're accomplished by copying the object to the new key, and then deleting the old object. If you want to rename a directory, you have to individually rename every object within it, since the directory name is a part of the key.

The key-value stores we discussed in Chapter 4 are optimized for small values (typically kilobytes) and frequent, low-latency reads/writes. In contrast, distributed filesystems and object stores are generally optimized for large objects (megabytes to gigabytes) and less frequent, larger reads. Recently, however, object stores have begun to add support for frequent and smaller reads/writes. For example, S3 Express One Zone now offers single-millisecond latency and a pricing model that is more similar to key-value stores.

Another difference between distributed filesystems and object stores is that DFSes such as HDFS allow computing tasks to be run on the machine that stores a copy of a particular file. This allows the task to read that file without having to send it over the network, which saves bandwidth if the executable code of the task is smaller than the file it needs to read. On the other hand, object stores usually keep storage and computation separate. Doing so might use more bandwidth, but modern datacenter networks are very fast, so this is often acceptable. This architecture also allows machine resources such as CPU and memory to be scaled independently of storage since the two are decoupled.

Distributed Job Orchestration

Our operating system analogy also applies to job orchestration. When you execute a Unix batch job, something needs to actually run the `awk`, `sort`, `uniq`, and `head` processes. Data needs to be transferred from one process's output to another process's input, memory must be allocated for each process, instructions from each process must be scheduled fairly and executed on the CPU, memory and I/O boundaries must be enforced, and so on. On a single machine, an operating system's kernel is responsible for such work. In a distributed environment, this is the role of a job orchestrator.

Batch processing frameworks send a request to an orchestrator's scheduler to run a job. Requests to start a job contain metadata such as:

- the number of tasks to execute,
- the amount of memory, CPU, and disk needed for each task,
- a job identifier,
- access credentials,
- job paramaters such as input and output data,
- required hardware details such as GPUs or disk types, and
- where the job's executable code is located.

Orchestrators such as Kubernetes and Hadoop YARN (Yet Another Resource Negotiator) [13] combine this information with cluster metadata to execute the job using the following components:

Task executors

> An executor daemon such as YARN's *NodeManager* or Kubernetes's *kubelet* runs on each node in the cluster. Executors are responsible for running job tasks, sending heartbeats to signal their liveness, and tracking task status and resource allocation on the node. When a task-start request is sent to an executor, it retrieves the job's executable code and runs a command to start the task. The executor then monitors the process until it finishes or fails, at which point it updates the task status metadata accordingly.
>
> Many executors also work with the operating system to provide both security and performance isolation. YARN and Kubernetes both use Linux *cgroups*, for example. This prevents tasks from accessing data without permission, or from negatively affecting the performance of other tasks on the node by using excessive resources.

Resource Manager

> An orchestrator's resource manager stores metadata about each node, including available hardware (CPUs, GPUs, memory, disks, and so

on), task statuses, network location, node status, and other relevant information. Thus, the manager provides a global view of the cluster's current state. The centralized nature of the resource manager can lead to both scalability and availability bottlenecks. YARN uses ZooKeeper and Kubernetes uses etcd to store cluster state (see "Coordination Services").

Scheduler

Orchestrators usually have a centralized scheduler subsystem, which receives requests to start, stop, or check on the status of a job. For example, a scheduler might receive a request to start a job with 10 tasks using a specific Docker image on nodes that have a specific type of GPU. The scheduler uses the information from the request and state of the resource manager to determine which tasks to run on which nodes. The task executors are then informed of their assigned work and begin execution.

Though each orchestrator uses different terminology, you will find these components in nearly all orchestration systems.

NOTE

Scheduling decisions sometimes require application-specific schedulers that can take into account particular requirements, such as auto-scaling read replicas when a certain query threshold is reached. The centralized scheduler and application-specific schedulers work together to determine how to best execute tasks. YARN refers to its sub-schedulers as *ApplicationMasters*, while Kubernetes calls them *operators*.

Resource Allocation

Schedulers have a particularly challenging role in job orchestration: they must figure out how to best allocate the cluster's limited resources amongst jobs with competing needs. Fundamentally, its decisions must balance fairness and efficiency.

Imagine a small cluster with five nodes that has a total of 160 CPU cores available. The cluster's scheduler receives two job requests, each wanting 100 cores to complete its work. What's the best way to schedule the workload?

- The scheduler could decide to run 80 tasks for each job, starting the remaining 20 tasks for each job as earlier tasks complete.
- The scheduler could run all of one job's tasks, and begin running the second job's tasks only when 100 cores are available, a strategy known as *gang scheduling*.
- One job request comes before the other. The scheduler has to decide whether to allocate all 100 cores to that job, or hold some back in anticipation for future jobs.

This is a very simple example, but we already see many difficult trade-offs. In the gang-scheduling scenario, for example, if the scheduler reserves CPU cores until all 100 are available at the same time, nodes will sit idle. The cluster's resource utilization will drop and a deadlock might occur if other jobs also attempt to reserve CPU cores.

On the other hand, if the scheduler simply waits for 100 cores to become available, other jobs might grab the cores in the meantime. The cluster might not have 100 cores available for a very long time, which leads to *starvation*. The scheduler could decide to *preempt* some of the first job's tasks, killing them to make room for the second job. Task preemption decreases cluster efficiency as well, since the killed tasks will need to be restarted later and re-run.

Now imagine a scheduler that must make allocation decisions for hundreds or even millions of such job requests. Finding an optimal solution seems intractable. In fact, the problem is *NP-hard*, which means that it is prohibitively slow to calculate an optimal solution for all but the smallest examples [14, 15].

In practice, schedulers therefore use heuristics to make non-optimal but reasonable decisions. Several algorithms are commonly used, including first-in first-out (FIFO), dominant resource fairness (DRF), priority queues, capacity or quota-based scheduling, and various bin-packing algorithms. The details for such algorithms are beyond the scope of this book, but they're a fascinating area of research.

Scheduling Workflows

The Unix tools example at the start of this chapter involved a chain of several commands, connected by Unix pipes. The same pattern arises in distributed batch processes: often the output from one job needs to become the input to one or more other jobs, and each job may have several inputs that are produced by other jobs. This is called a *workflow* or *directed acyclic graph (DAG)* of jobs.

NOTE

In "Durable Execution and Workflows" we saw workflow engines that offer durable execution of a sequence of steps, typically performing RPCs. In the context of batch processing, "workflow" has a different meaning: it's a sequence of batch processes, each taking input data and producing output data, but normally not making RPCs to external services. Durable execution engines typically process less data per-request than their batch processing counterparts, though the line is somewhat fuzzy.

There are several reasons why a workflow of multiple jobs might be needed:

- If the output of one job needs to become the input to several other jobs, which are maintained by different teams, it's best for the first job to first write its output to a location where all the other jobs can read it. Those consuming jobs can then be scheduled to run every time that data has been updated, or on some other schedule.
- You might want to transfer data from one processing tool to another. For example, a Spark job might output its data to HDFS, then a Python script might trigger a Trino SQL query (see "Cloud Data Warehouses") that does further processing on the HDFS files and outputs to S3.
- Some data pipelines internally require multiple processing stages. For example, if one stage needs to shard the data by one key, and the next stage needs to shard by a different key, the first stage can output data sharded in the way that is required by the second stage.

In the Unix tools example, the pipe that connects the output of one command to the input of another uses only a small in-memory buffer, and doesn't write the data into a file. If that buffer fills up, the producing process needs to wait until the consuming process has read some data from the buffer before it can output more—a form of backpressure. Spark, Flink, and other batch execution engines support a similar model where the output of one task is directly passed to another task (over the network if the tasks are running on different machines).

However, in a workflow it is more usual for one job to write its output to a distributed filesystem or object store, and for the next job to read it from there. This decouples the jobs from each other, allowing them to run at different times. If a job has several inputs, a workflow scheduler typically waits until all of the jobs that produce its inputs have completed successfully before running the job that consumes those inputs.

Schedulers found in orchestration frameworks such as YARN's ResourceManager or Spark's built-in scheduler do not manage entire workflows; they do scheduling on a per-job basis. To handle these dependencies between job executions, various workflow schedulers have been developed, including Airflow, Dagster, and Prefect. Workflow schedulers have management features that are useful when maintaining a large collection of batch jobs. Workflows consisting of 50 to 100 jobs are common in many data pipelines, and in a large organization, many different teams may be running different jobs or workflows that read each other's output across many different systems. Tool support is important for managing such complex dataflows.

Handling Faults

Batch jobs often run for long periods of time. Long-running jobs with many parallel tasks are likely to experience at least one task failure along the way. As discussed in "Hardware and Software Faults" and "Unreliable Networks", there are many reasons why this could happen, including hardware faults (especially on commodity hardware), or network interruptions.

Another reason why a task might not finish running is that the scheduler may intentionally preempt (kill) it. Preemption is particularly useful if you have multiple priority levels: low-priority tasks that are cheaper to run, and high-priority tasks that cost more. Low-priority tasks can run whenever there is spare computing capacity, but they run the risk of being preempted at any moment if a higher-priority task arrives. Such cheaper, low-priority virtual machines are called *spot instances* on Amazon EC2, *spot virtual machines* on Azure, and *preemptible instances* on Google Cloud [16].

Since batch processing is often used for jobs that are not time-sensitive, it is well suited for using low-priority tasks and spot instances to reduce the cost of running jobs. Essentially, those jobs can use spare computing resources that would otherwise be idle, and thereby increase the utilization of the cluster.

However, this also means that those tasks are more likely to be killed by the scheduler: preemptions occur more frequently than hardware faults [17].

Since batch jobs regenerate their output from scratch every time they are run, task failures are easier to handle than in online systems: the system can delete the partial output from the failed execution and schedule it to run again on another machine. It would be wasteful to rerun the entire job due to a single task failure, though. MapReduce and its successors therefore keep the execution of parallel tasks independent from each other, so that they can retry work at the granularity of an individual task [3].

Fault tolerance is trickier when the output of one task becomes the input to another task as part of a workflow. MapReduce solves this by always writing such intermediate data back to the distributed filesystem, and waiting for the writing task to complete successfully before allowing other tasks to read the data. This works, even in an environment where preemption is common, but it means a lot of writes to the DFS, which can be inefficient.

Spark keeps intermediate data in memory or "spills" to local disk, and only writes the final result to the DFS. It also keeps track of how the intermediate data was computed, allowing Spark to recompute it in case it is lost [18]. Flink uses a different approach based on periodically checkpointing a snapshot of tasks [19]. We will return to this topic in "Dataflow Engines".

Batch Processing Models

We have seen how batch jobs are scheduled in a distributed environment. Let us now turn our attention to how batch processing frameworks actually process data. The two most common models are MapReduce and dataflow engines. Although dataflow engines have largely replaced MapReduce in practice, it is useful to understand how MapReduce works, since it influenced many modern batch processing frameworks.

MapReduce and dataflow engines have evolved to support multiple programming models including low-level programmatic APIs, relational query languages, and DataFrame APIs. A variety of options enable application engineers, analytics engineers, business analysts, and even non-technical employees to process company data for various use cases, which we'll discuss in "Batch Use Cases".

MapReduce

The pattern of data processing in MapReduce is very similar to the web server log analysis example in “Simple Log Analysis”:

1. Read a set of input files, and break it up into *records*. In the web server log example, each record is one line in the log (that is, `\n` is the record separator). In Hadoop’s MapReduce, the input file is stored in a distributed filesystem like HDFS or an object store like S3. Various file formats are used, such as Apache Parquet (a columnar format, see “Column-Oriented Storage”) or Apache Avro (a row-based format, see “Avro”).
2. Call the mapper function to extract a key and value from each input record. In the Unix tool example, the mapper function is `awk '{print $7}'` : it extracts the URL (`$7`) as the key, and leaves the value empty.
3. Sort all of the key-value pairs by key. In the log example, this is done by the first `sort` command.
4. Call the reducer function to iterate over the sorted key-value pairs. If there are multiple occurrences of the same key, the sorting has made them adjacent in the list, so it is easy to combine those values without having to keep a lot of state in memory. In the Unix tool example, the reducer is implemented by the command `uniq -c` , which counts the number of adjacent records with the same key.

Those four steps can be performed by one MapReduce job. Steps 2 (map) and 4 (reduce) are where you write your custom data processing code. Step 1 (breaking files into records) is handled by the input format parser. Step 3, the `sort` step, is implicit in MapReduce—you don’t have to write it, because the output from the mapper is always sorted before it is given to the reducer. This sorting step is a foundational batch processing algorithm, which we’ll revisit in “Shuffling Data”.

To create a MapReduce job, you need to implement two callback functions, the mapper and reducer, which behave as follows:

Mapper

The mapper is called once for every input record, and its job is to extract the key and value from the input record. For each input, it may generate any number of key-value pairs (including none). It does not keep any state from one input record to the next, so each record is handled independently.

Reducer

The MapReduce framework takes the key-value pairs produced by the mappers, collects all the values belonging to the same key, and calls the reducer with an iterator over that collection of values. The reducer can produce output records (such as the number of occurrences of the same URL).

In the web server log example, we had a second `sort` command in step 5, which ranked URLs by number of requests. In MapReduce, if you need a second sorting stage, you can implement it by writing a second MapReduce job and using the output of the first job as input to the second job. Viewed like this, the role of the mapper is to prepare the data by putting it into a form that is suitable for sorting, and the role of the reducer is to process the data that has been sorted.

MAPREDUCE AND FUNCTIONAL PROGRAMMING

Though MapReduce is used for batch processing, the programming model comes from functional programming. Lisp introduced *map* and *reduce* (or *fold*) as higher-order functions on lists, and they have made their way into mainstream languages such as Python, Rust, and Java. Many common data processing operations, including those offered by SQL, can be implemented on top of MapReduce. Both functions, and functional programming in general, have important properties that MapReduce benefits from. Map and reduce are composable, which fits nicely with data processing (as we saw in our Unix example). Map is also *embarassingly parallel* (each input is processed independently), which simplifies MapReduce's parallel execution. For reduce, different keys can be processed in parallel.

Implementing a complex processing job using the raw MapReduce APIs is actually quite hard and laborious—for instance, any join algorithms used by the job would need to be implemented from scratch [20]. MapReduce is also quite slow compared to more modern batch processors. One reason is that its file-based I/O prevents job pipelining, i.e., processing output data in a downstream job before the upstream job is complete.

Dataflow Engines

In order to fix some of MapReduce's problems, several new execution engines for distributed batch computations were developed, the most well known of which are Spark [18, 21] and Flink [19]. There are various differences in the way they are designed, but they have one thing in common: they handle an entire workflow as one job, rather than breaking it up into independent subjobs.

Since they explicitly model the flow of data through several processing stages, these systems are known as *dataflow engines*. Like MapReduce, they support a low-level API that repeatedly calls a user-defined function to process one record at a time, but they also offer higher-level operators such as *join* and *group by*. They parallelize work by sharding inputs, and they copy the output of one task over the network to become the input to another task. Unlike in MapReduce, operators need not take the strict roles of alternating map and reduce, but instead can be assembled in more flexible ways.

These dataflow APIs generally use relational-style building blocks to express a computation: joining datasets on the value of some field; grouping tuples by key; filtering by some condition; and aggregating tuples by counting, summing, or other functions. Internally, these operations are implemented using the shuffle algorithms that we discuss in the next section.

This style of processing engine is based on research systems like Dryad [22] and Nephele [23], and it offers several advantages compared to the MapReduce model:

- Expensive work such as sorting need only be performed in places where it is actually required, rather than always happening by default between every map and reduce stage.
- When there are several operators in a row that don't change the sharding of the dataset (such as map or filter), they can be combined into a single task, reducing data copying overheads.
- Because all joins and data dependencies in a workflow are explicitly declared, the scheduler has an overview of what data is required where, so it can make locality optimizations. For example, it can try to place the task that consumes some data on the same machine as the task that produces it, so that the data can be exchanged through a shared memory buffer rather than having to copy it over the network.

- It is usually sufficient for intermediate state between operators to be kept in memory or written to local disk, which requires less I/O than writing it to a distributed filesystem or object store (where it must be replicated to several machines and written to disk on each replica). MapReduce already uses this optimization for mapper output, but dataflow engines generalize the idea to all intermediate state.
- Operators can start executing as soon as their input is ready; there is no need to wait for the entire preceding stage to finish before the next one starts.
- Existing processes can be reused to run new operators, reducing startup overheads compared to MapReduce (which launches a new JVM for each task).

You can use dataflow engines to implement the same computations as MapReduce workflows, and they usually execute significantly faster due to the optimizations described here.

Shuffling Data

We saw that both the Unix tools example at the beginning of the chapter and MapReduce are based on sorting. Batch processors need to be able to sort datasets petabytes in size, which are too large to fit on a single machine. They therefore require a distributed sorting algorithm where both the input and the output is sharded. Such an algorithm is called a *shuffle*.

SHUFFLE IS NOT RANDOM

The term *shuffle* is confusing. When you shuffle a deck of cards, you end up with a random order. In contrast, the shuffle we're talking about here produces a sorted order, with no randomness.

Shuffling is a foundational algorithm for batch processors, where it is used for joins and aggregations. MapReduce, Spark, Flink, Daft, Dataflow, and BigQuery [24] all implement scalable and performant shuffle algorithms in order to handle large datasets. We'll use the shuffle in Hadoop MapReduce [25] for illustration purposes, but the concepts in this section translate to other systems as well.

Figure 11-1 shows the dataflow in a MapReduce job. We assume that the input to the job is sharded, and the shards are labelled *m 1*, *m 2*, and *m 3*. For example, each shard may be a separate file on HDFS or a separate object in an object store, and all the shards belonging to the same dataset are grouped into the same HDFS directory or have the same key prefix in an object store bucket.

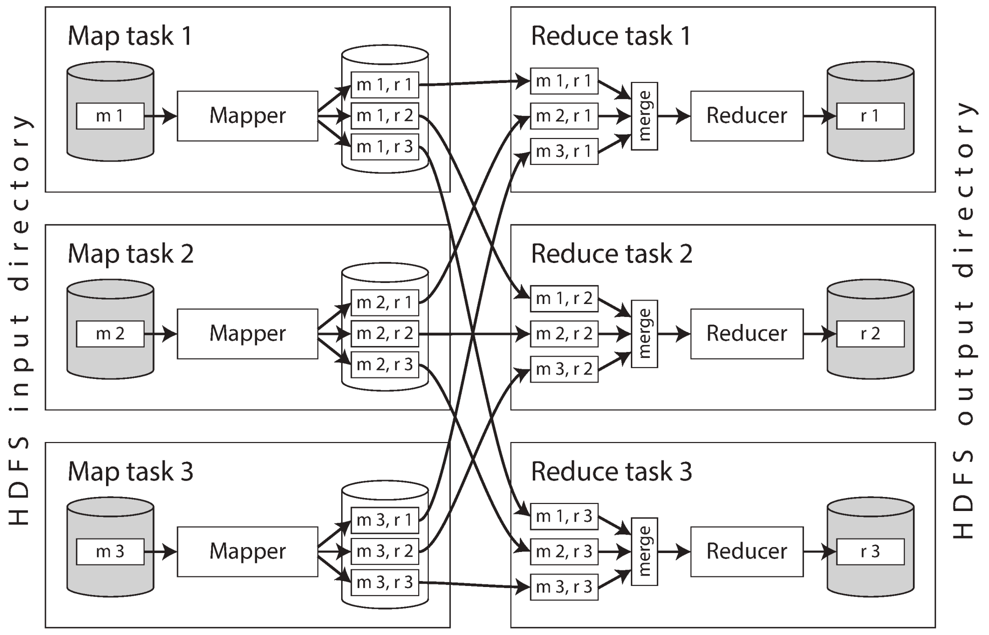

Figure 11-1. A MapReduce job with three mappers and three reducers.

The framework starts a separate map task for each input shard. A task reads its assigned file, passing one record at a time to the mapper callback. The reduce side of the computation is also sharded. While the number of map tasks is determined by the number of input shards, the number of reduce tasks is configured by the job author (it can be different from the number of map tasks).

The output of the mapper consists of key-value pairs, and the framework needs to ensure that if two different mappers output the same key, those key-value pairs end up being processed by the same reducer task. To achieve this, each mapper creates a separate output file on its local disk for every reducer (for example, the file *m 1, r 2* in Figure 11-1 is the file created by mapper 1 containing the data destined for reducer 2). When the mapper outputs a key-value pair, a hash of the key typically determines which reducer file it is written to (similarly to "Sharding by Hash of Key").

While a mapper is writing these files, it also sorts the key-value pairs within each file. This can be done using the techniques we saw in "Log-Structured

Storage”: batches of key-value pairs are first collected in a sorted data structure in memory, then written out as sorted segment files, and smaller segment files are progressively merged into larger ones.

After each mapper finishes, reducers connect to it and copy the appropriate file of sorted key-value pairs to their local disk. Once the reduce task has its share of the output from all of the mappers, it merges these files together, preserving the sort order, mergesort-style. Key-value pairs with the same key are now consecutive, even if they came from different mappers. The reducer function is then called once per-key, each time with an iterator that returns all the values for that key.

Any records output by the reducer function are sequentially written to a file, with one file per reduce task. These files (*r 1*, *r 2*, *r 3* in Figure 11-1) become the shards of the job’s output dataset, and they are written back to the distributed filesystem or object store.

Though MapReduce executes the shuffle step between its map and reduce steps, modern dataflow engines and cloud data warehouses are more sophisticated. Systems such as BigQuery have optimized their shuffle algorithms to keep data in memory and to write data to external sorting services [24]. Such services speed up shuffling and replicate shuffled data to provide resilience.

JOIN and GROUP BY

Let’s look at how sorted data simplifies distributed joins and aggregations. We’ll continue with MapReduce for illustration purposes, though these concepts apply to most batch processing systems.

A typical example of a join in a batch job is illustrated in Figure 11-2. On the left is a log of events describing the things that logged-in users did on a website (known as *activity events* or *clickstream data*), and on the right is a database of users. You can think of this example as being part of a star schema (see “Stars and Snowflakes: Schemas for Analytics”): the log of events is the fact table, and the user database is one of the dimensions.

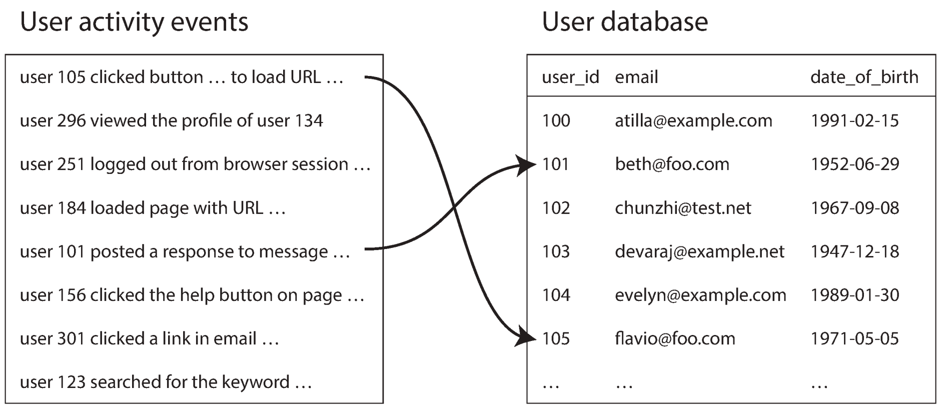
User activity events
user 105 clicked button … to load URL …
user 296 viewed the profile of user 134
user 251 logged out from browser session …
user 184 loaded page with URL …
user 101 posted a response to message …
user 156 clicked the help button on page …
user 301 clicked a link in email …
user 123 searched for the keyword …
User database
user_id email date_of_birth
100 atilla@example.com 1991-02-15
101 beth@foo.com 1952-06-29
102 chunzhi@test.net 1967-09-08
103 devaraj@example.net 1947-12-18
104 evelyn@example.com 1989-01-30
105 flavio@foo.com 1971-05-05
… … …

Figure 11-2. A join between a log of user activity events and a database of user profiles.

If you want to perform an analysis of the activity events that takes into account information from the user database (for example, find out whether certain pages are more popular with younger or older users, using the date of birth field in the user profile), you need to compute a join between these two tables. How would you compute that join, assuming both tables are so large that they have to be sharded?

You can use the fact that in MapReduce, the shuffle brings together all the key-value pairs with the same key to the same reducer, no matter which shard they were on originally. Here, the user ID can serve as the key. You can therefore write a mapper that goes over the user activity events, and emits page view URLs keyed by user ID, as illustrated in Figure 11-3. Another mapper goes over the user database row by row, extracting the user ID as the key and the user's date of birth as the value.

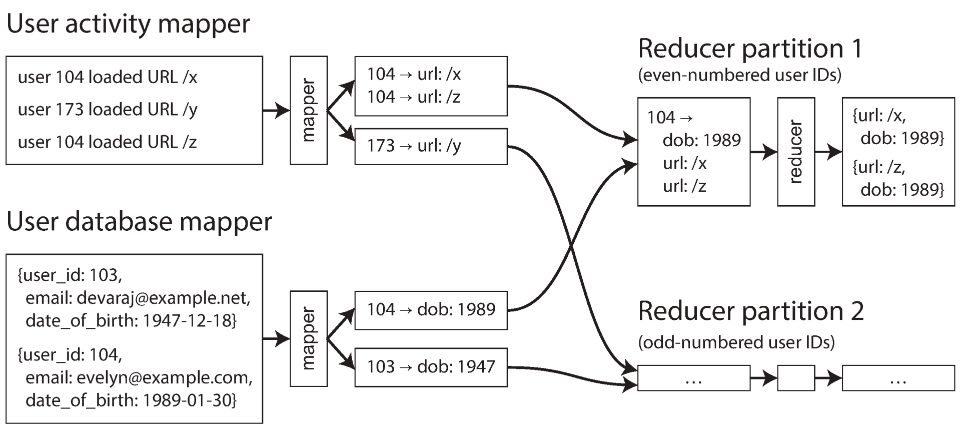

Figure 11-3. A sort-merge join on user ID. If the input datasets are sharded into multiple files, each could be processed with multiple mappers in parallel.

The shuffle then ensures that a reducer function can access a particular user's date of birth and all of that user's page view events at the same time. The MapReduce job can even arrange the records to be sorted such that the reducer always sees the record from the user database first, followed by the activity events in timestamp order—this technique is known as a *secondary sort* [25].

The reducer can then perform the actual join logic easily. The first value is expected to be the date of birth, which the reducer stores in a local variable. It then iterates over the activity events with the same user ID, outputting each viewed URL along with the viewer's date of birth. Since the reducer processes all of the records for a particular user ID in one go, it only needs to keep one user record in memory at any one time, and it never needs to make any requests over the network. This algorithm is known as a *sort-merge join*, since mapper output is sorted by key, and the reducers then merge together the sorted lists of records from both sides of the join.

The next MapReduce job in the workflow can then calculate the distribution of viewer ages for each URL. To do so, the job would first shuffle the data using the URL as key. Once sorted, the reducers would then iterate over all the page views (with viewer birth date) for a single URL, keep a counter for the number of views by each age group, and increment the appropriate counter for each page view. This way you can implement a *group by* operation and aggregation.

Query languages

Over the years, execution engines for distributed batch processing have matured. By now, the infrastructure has become robust enough to store and process many petabytes of data on clusters of over 10,000 machines. As the problem of physically operating batch processes at such scale has been considered more or less solved, attention has turned to improving the programming model.

MapReduce, dataflow engines, and cloud data warehouses have all embraced SQL as the lingua franca for batch processing. It's a natural fit: legacy data warehouses used SQL, data analytics and ETL tools already support SQL, and all developers and analysts know it.

Besides the obvious advantage of requiring less code than handwritten MapReduce jobs, these query language interfaces also allow interactive use, in which you write analytical queries and run them from a terminal or GUI. This style of interactive querying is an efficient and natural way for business analytics, product managers, sales and finance teams, and others to explore data in a batch processing environment. Though not a classic form of batch processing, SQL support has made exploratory queries suitable for distributed batch processing systems.

High-level query languages not only make the humans using the system more productive, but they also improve the job execution efficiency at a machine level. As we saw in "Cloud Data Warehouses", query engines are responsible for converting SQL queries into batch jobs to be executed in a cluster. This translation step from query to syntax tree to physical operators allows the engine to optimize queries. Query engines such as Hive, Trino, Spark, and Flink have cost-based query optimizers that can analyze the properties of join inputs and automatically decide which algorithm would be most suitable for the task at hand. Optimizers might even change the order of joins so that the amount of intermediate state is minimized [19, 26, 27, 28].

While SQL is the most popular general-purpose batch processing query language, other languages remain in use for niche use cases. Apache Pig was a language based on relational operators that allowed data pipelines to be specified step by step, rather than as one big SQL query. DataFrames (see next section) have similar characteristics, and Morel is a more modern language influenced by Pig. Other users have adopted JSON query languages such as jq, JMESPath, or JsonPath.

In "Graph-Like Data Models" we discussed using graphs for modeling data, and using graph query languages to traverse the edges and vertices in a graph. Many graph processing frameworks also support batch computation through query languages such as Apache TinkerPop's Gremlin. We will look at graph processing use cases in more detail in "Batch Use Cases".

BATCH PROCESSING AND CLOUD DATA WAREHOUSES CONVERGE

Historically, data warehouses ran on specialized hardware appliances, and provided SQL analytics queries over relational data. In contrast, batch processing frameworks like MapReduce set out to provide greater scalability and greater flexibility by supporting processing logic written in a general-purpose programming language, allowing it to read and write arbitrary data formats.

Over time, the two have become much more similar. Modern batch processing frameworks now support SQL as a language for writing batch jobs, and they achieve good performance on relational queries by using columnar storage formats such as Parquet and optimized query execution engines (see "Query Execution: Compilation and Vectorization"). Meanwhile, data warehouses have grown more scalable by moving to the cloud (see "Cloud Data Warehouses"), and implementing many of the same scheduling, fault tolerance, and shuffling techniques that distributed batch frameworks do. Many use distributed filesystems as well.

Just as batch processing systems adopted SQL as a processing model, cloud warehouses have adopted alternative processing models such as DataFrames as well (discussed in the next section). For example, Google Cloud BigQuery offers a BigQuery DataFrames library and Snowflake's Snowpark integrates with Pandas. Batch processing workflow orchestrators such as Airflow, Prefect, and Dagster also integrate with cloud warehouses.

Not all batch jobs are easily expressed in SQL, though. Iterative graph algorithms such as PageRank, complex machine learning, and many other tasks are difficult to express in SQL. AI data processing, which includes non-relational and multi-modal data such as images, video, and audio, can also be difficult to do in SQL.

Moreover, cloud data warehouses struggle with certain workloads. Row-by-row computation is less efficient when using column-oriented storage formats. Alternative warehouse APIs or a batch processing system are preferable in such cases. Cloud data warehouses also tend to be more expensive than other batch processing systems. It can be more cost-efficient to run large jobs in batch processing systems such as Spark or Flink instead.

Ultimately, the decision between processing data in batch systems or data warehouses comes down to factors such as cost, convenience, ease of implementation, availability, and so on. Most large enterprises have many data processing systems, which give them flexibility in this decision. Smaller companies often get by with just one.

DataFrames

As data scientists and statisticians began using distributed batch processing frameworks for machine learning use cases, they found existing processing models cumbersome, as they were used to working with the DataFrame data model found in R and Pandas (see "DataFrames, Matrices, and Arrays"). A DataFrame is similar to a table in a relational database: it is a collection of rows, and all the values in the same column have the same type. Instead of writing one big SQL query, users call functions corresponding to relational operators to perform filters, joins, sorting, group by, and other operations.

Originally, DataFrame manipulation typically occurred locally, in memory. Consequently, DataFrames were limited to datasets that fit on a single machine. Data scientists wanted to interact with the large datasets found in batch processing environments using the DataFrame APIs they were used to. Distributed data processing frameworks such as Spark, Flink, and Daft have adopted DataFrame APIs to meet this need. On the other hand, local DataFrames are usually indexed and ordered while distributed DataFrames are generally not [29]. This can lead to performance surprises when migrating to batch frameworks.

DataFrame APIs appear similar to dataflow APIs, but implementations vary. While Pandas executes operations immediately when the DataFrame methods are called, Apache Spark first translates all the DataFrame API calls into a query plan and runs query optimization before executing the workflow on top of its distributed dataflow engine. This allows it to improve performance.

Frameworks such as Daft even support both client and server-side computation. Smaller, in-memory operations are executed on the client while larger datasets and computation are executed on the server. Columnar storage formats such as Apache Arrow offer a unified data model that both client and server-side execution engines can share.

Batch Use Cases

Now that we've seen how batch processing works, let's see how it is applied to a range of different applications. Batch jobs are excellent for processing large datasets in bulk, but they aren't good for low latency use cases. Consequently, you'll find batch jobs wherever there's a lot of data and data freshness isn't important. This might sound limiting, but it turns out that the a significant amount of data processing fits this model:

- Accounting and inventory reconciliation, where companies verify that transactions line up with their bank accounts and inventory, are often done in batch [30].
- In manufacturing, demand forecasting is computed in periodic batch jobs [31].
- Ecommerce, media, and social media companies train their recommendation models using batch jobs [32, 33].
- Many financial systems are batch-based, as well. For example, the United States's banking network runs almost entirely on batch jobs [34].

In the following sections, we'll discuss some of the batch processing use cases you'll find in nearly every industry.

Extract–Transform–Load (ETL)

"Data Warehousing" introduced the idea of ETL and ELT, where a data processing pipeline extracts data from a production database, transforms it, and loads results into a downstream system (we'll use "ETL" in this section to represent both ETL and ELT workloads). Batch jobs are often used for such workloads, especially when the downstream system is a data warehouse.

The parallel nature of batch jobs makes them a great fit for data transformation. Much of data transformation involves "embarrassingly parallel" workloads. Filtering data, projecting fields, and many other common data warehouse transformations can all be done in parallel.

Batch processing environments also come with robust workflow schedulers, which make it easy to schedule, orchestrate, and debug ETL data pipeline jobs. When a failure occurs, schedulers often retry jobs to mitigate transient issues that might occur. A job that fails repeatedly will be marked as failed, which helps developers easily see which job in their data pipeline stopped working. Schedulers like Airflow even come with built-in source, sink, and query operators for MySQL, PostgreSQL, Snowflake, Spark, Flink, and dozens of other popular systems. A tight integration between schedulers and data processing systems simplifies data integration.

We've also seen that batch jobs are easy to troubleshoot and fix when things go awry. This feature is invaluable when debugging data pipelines. Failed files can be easily inspected to see what went wrong, and ETL batch jobs can be fixed and re-run. For example, an input file might no longer contain a field that a transformation batch job intends to use. Data engineers will see that the field is missing, and update the transformation logic or the job that produced the input.

Data pipelines used to be managed by a single data engineering team, as it was considered unfair to ask other teams working on product features to write and manage complex batch data pipelines. Recently, improvements in batch processing models and metadata management have made it much easier for engineers across an organization to contribute to and manage their own data pipelines. *Data mesh* [35, 36], *data contract* [37], and *data fabric* [38] practices provide standards and tools to help teams safely publish their data for consumption by anybody in the organization.

Data pipelines and analytic queries have begun to share not only processing models, but execution engines as well. Many batch ETL jobs now run on the same systems as the analytic queries that read their output. It is not uncommon to see data pipeline transformations and analytic queries both run as SparkSQL, Trino, or DuckDB queries. Such an architecture further blurs the line between application engineering, data engineering, analytics engineering, and business analysis.

Analytics

In "Operational Versus Analytical Systems", we saw that analytic queries (OLAP) often scan over a large number of records, performing groupings and aggregations. It is possible to run such workloads in a batch processing system, alongside other batch processing workloads. Analysts write SQL queries that execute atop a query engine, which reads and writes from a distributed file system or object store. Table metadata such as table to file mappings, names, and types are managed with table formats such as Apache

Iceberg and catalogs such as Unity (see “Cloud Data Warehouses”). This architecture is known as a *data lakehouse* [39].

As with ETL, improvements in SQL query interfaces mean many organizations now use batch frameworks such as Spark for analytics. Such query patterns come in two styles:

- Pre-aggregation queries, where data is rolled up into OLAP cubes or data marts to speed up queries (see “Materialized Views and Data Cubes”). Pre-aggregated data is queried in the warehouse or pushed to purpose-built realtime OLAP systems such as Apache Druid or Apache Pinot. Pre-aggregation normally takes place at a scheduled interval. The workflow schedulers discussed in “Scheduling Workflows” are used to manage these workloads.
- Ad hoc queries that users run to answer specific business questions, investigate user behavior, debug operational issues, and much more. Response times are important for this use case. Analysts run queries iteratively as they get responses and learn more about the data they’re investigating. Batch processing frameworks with fast query execution help reduce waiting times for analysts.

SQL support enables batch processing frameworks to integrate with spreadsheets and data visualization tools such as Tableau, Power BI, Looker, and Apache Superset. For example, Tableau offers SparkSQL and Presto connectors, while Apache Superset supports Trino, Hive, Spark SQL, Presto, and many other systems that ultimately execute batch jobs to query data.

Machine Learning

Machine learning (ML) makes frequent use of batch processing. Data scientists, ML engineers, and AI engineers use batch processing frameworks to investigate data patterns, transform data, and train machine learning models. Common uses include:

- Feature engineering: Raw data is filtered and transformed into data that models can be trained on. Predictive models often need numeric data, so engineers must transform other forms of data (such as text or discrete values) into the required format.
- Model training: The training data is the input to the batch process, and the weights of the trained model are the output.

- Batch inference: A trained model can then be used to make predictions in bulk if datasets are large and realtime results are not required. This includes evaluating the model's predictions on a test dataset.

Batch processing frameworks provide tools explicitly for these use cases. For example, Apache Spark's MLlib and Apache Flink's FlinkML come with a wide variety of feature engineering tools, statistical functions, and classifiers.

Machine learning applications such as recommendation engines and ranking systems also make heavy use of graph processing (see "Graph-Like Data Models"). Many graph algorithms are expressed by traversing one edge at a time, joining one vertex with an adjacent vertex in order to propagate some information, and repeating until some condition is met—for example, until there are no more edges to follow, or until some metric converges.

The *bulk synchronous parallel* (BSP) model of computation [40] has become popular for batch processing graphs. Among others, it is implemented by Apache Giraph [20], Spark's GraphX API, and Flink's Gelly API [41]. It is also known as the *Pregel* model, as Google's Pregel paper popularized this approach for processing graphs [42].

Batch processing is also an integral part of large language model (LLM) data preparation and training. Raw text input data such as websites typically reside in a DFS or object store. This data must be pre-processed to make it suitable for training. Pre-processing steps that are well-suited for batch processing frameworks include:

- Plain text must be extracted from HTML and malformed text must be fixed.
- Low quality, irrelevant, and duplicate documents must be detected and removed.
- Text must be tokenized (split into words) and converted into embeddings, which are numeric representations each word.

Batch processing frameworks such as Kubeflow, Flyte, and Ray are purpose-built for such workloads. OpenAI uses Ray as part of its ChatGPT training process, for example [43]. These frameworks have built-in integrations for LLM and AI libraries such as PyTorch, Tensorflow, XGBoost, and many others. They also offer built-in support for feature engineering, model training, batch inference, and fine tuning (adjusting a foundational model for specific use cases).

Finally, data scientists often experiment with data in interactive notebooks such as Jupyter or Hex. Notebooks are made up of *cells*, which are small chunks of markdown, Python, or SQL. Cells are executed sequentially to produce spreadsheets, graphs, or data. Many notebooks use batch processing via DataFrame APIs or query such systems using SQL.

Serving Derived Data

Batch jobs are often used to build pre-computed or derived datasets such as product recommendations, user-facing reports, and features for machine learning models. These datasets are typically served from a production database, key-value store, or search engine. Regardless of the system used, the pre-computed data needs to make its way from the batch processor's distributed filesystem or object store back into the database that's serving live traffic.

The most obvious choice might be to use the client library for your favorite database directly within a batch job, and to write directly to the database server, one record at a time. This will work (assuming your firewall rules allow direct access from your batch processing environment to your production databases), but it is a bad idea for several reasons:

- Making a network request for every single record is orders of magnitude slower than the normal throughput of a batch task. Even if the client library supports batching, performance is likely to be poor.
- Batch processing frameworks often run many tasks in parallel. If all the tasks concurrently write to the same output database, with a rate expected of a batch process, that database can easily be overwhelmed, and its performance for queries is likely to suffer. This can in turn cause operational problems in other parts of the system [44].
- Normally, batch jobs provide a clean all-or-nothing guarantee for job output: if a job succeeds, the result is the output of running every task exactly once, even if some tasks failed and had to be retried along the way; if the entire job fails, no output is produced. However, writing to an external system from inside a job produces externally visible side effects that cannot be hidden in this way. Thus, you have to worry about the results from partially completed jobs being visible to other systems. If a

task fails and is restarted, it may duplicate output from the failed execution.

A better solution is to have batch jobs push pre-computed datasets to streams such as Kafka topics, which we discuss further in Chapter 12. Search engines like Elasticsearch, realtime OLAP systems like Apache Pinot and Apache Druid, derived datastores like Venice [45], and cloud data warehouses like ClickHouse all have the built-in ability to ingest data from Kafka into their systems. Pushing data through a streaming systems fixes a few of the problems we discussed above:

- Streaming systems are optimized for sequential writes, which make them better suited for the bulk write workload of a batch job.
- Streaming systems can also act as a buffer between the batch job and the production databases. Downstream systems can throttle their read rate to ensure they can continue to comfortably serve production traffic.
- The output of a single batch job can be consumed by multiple downstream systems.
- Streaming systems can serve as a security boundary between batch processing environments and production networks: they can be deployed in a so-called DMZ (demilitarized zone) network that sits between the batch processing network and production network.

Pushing data through streams doesn't inherently solve the all-or-nothing guarantee issue we discussed above. To make this work, batch jobs must send a notification to downstream systems that their job is complete and the data can now be served. Consumers of the stream need to be able to keep data they receive invisible to queries, like an uncommitted transaction with *read committed* isolation (see "Read Committed"), until they are notified that it is complete.

Another pattern that is more common when bootstrapping databases is to build a brand-new database *inside* the batch job and bulk load those files directly into the database from a distributed filesystem, object store, or local filesystem. Many data systems offer bulk import tools such as TiDB's Lightning tool, or Apache Pinot's and Apache Druid's Hadoop import jobs. RocksDB also offers an API to bulk import SSTs from batch jobs.

Building databases in batch and bulk importing the data is very fast, and makes it easier for systems to atomically switch between dataset versions. On the other hand, it can be challenging to incrementally update datasets from batch jobs that build brand-new databases. It’s common to take a hybrid approach in situations where both bootstrapping and incremental loads are needed. Venice, for example, supports hybrid stores that allow for batch row-based updates and full dataset swaps.

Summary

In this chapter, we explored the design and implementation of batch processing systems. We began with the classic Unix toolchain (awk, sort, uniq, etc.), to illustrate fundamental batch processing primitives such as sorting and counting.

We then scaled up to distributed batch processing systems. We saw that batch-style I/O processes immutable, bounded input datasets to produce output data, allowing reruns and debugging without side effects. To process files, we saw that batch frameworks have three main components: an orchestration layer that determines where and when jobs run, a storage layer to persist data, and a computation layer that processes the actual data.

We looked at how distributed filesystems and object stores manage large files through block-based replication, caching, and metadata services, and how modern batch frameworks interact with these systems using pluggable APIs. We also discussed how orchestrators schedule tasks, allocate resources, and handle faults in large clusters. We also compared job orchestrators that schedule jobs with workflow orchestrators that manage the lifecycle of a collection of jobs that run in a dependency graph.

We surveyed batch processing models, starting with MapReduce and its canonical map and reduce functions. Next, we turned to dataflow engines like Spark and Flink, which offer simpler-to-use dataflow APIs and better performance. To understand how batch jobs scale, we covered the shuffle algorithm, a foundational operation that enables grouping, joining, and aggregation.

As batch systems matured, focus shifted to usability. You learned about high-level query languages like SQL and DataFrame APIs, which make batch jobs more accessible and easier to optimize. Query optimizers translate declarative queries into efficient execution plans.

We finished the chapter with common batch processing use cases:

- ETL pipelines, which extract, transform, and load data between different systems using scheduled workflows;
- Analytics, where batch jobs support both pre-aggregated dashboards and ad hoc queries;
- Machine learning, where batch jobs prepare and process large training datasets;
- Populating production-facing systems from batch outputs, often via streams or bulk loading tools, in order to serve the derived data to users.

In the next chapter, we will turn to stream processing, in which the input is *unbounded*—that is, you still have a job, but its inputs are never-ending streams of data. In this case, a job is never complete, because at any time there may still be more work coming in. We shall see that stream and batch processing are similar in some respects, but the assumption of unbounded streams also changes a lot about how we build systems.

FOOTNOTES

REFERENCES

[1] Nathan Marz. How to Beat the CAP Theorem. *nathanmarz.com*, October 2011. Archived at perma.cc/4BS9-R9A4

[2] Molly Bartlett Dishman and Martin Fowler. Agile Architecture. At *O'Reilly Software Architecture Conference*, March 2015.

[3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. At *6th USENIX Symposium on Operating System Design and Implementation* (OSDI), December 2004.

[4] Shivnath Babu and Herodotos Herodotou. Massively Parallel Databases and MapReduce Systems. *Foundations and Trends in Databases*, volume 5, issue 1, pages 1–104, November 2013. doi:10.1561/1900000036

[5] David J. DeWitt and Michael Stonebraker. MapReduce: A Major Step Backwards. Originally published at *databasecolumn.vertica.com*, January 2008. Archived at perma.cc/U8PA-K48V

[6] Henry Robinson. The Elephant Was a Trojan Horse: On the Death of Map-Reduce at Google. *the-paper-trail.org*, June 2014. Archived at perma.cc/9FEM-X787

[7] Urs Hölzle. R.I.P. MapReduce. After having served us well since 2003, today we removed the remaining internal codebase for good. *twitter.com*, September 2019. Archived at perma.cc/B34T-LLY7

[8] Adam Drake. Command-Line Tools Can Be 235x Faster than Your Hadoop Cluster. *aadrake.com*, January 2014. Archived at perma.cc/87SP-ZMCY

[9] `sort` : Sort text files. GNU Coreutils 9.7 Documentation, Free Software Foundation, Inc., 2025.

[10] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The Quantcast File System. *Proceedings of the VLDB Endowment*, volume 6, issue 11, pages 1092–1101, August 2013. doi:10.14778/2536222.2536234

[11] Andrew Wang, Zhe Zhang, Kai Zheng, Uma Maheswara G., and Vinayakumar B. Introduction to HDFS Erasure Coding in Apache Hadoop. *blog.cloudera.com*, September 2015. Archived at archive.org

[12] Andy Warfield. Building and operating a pretty big storage system called S3. *allthingsdistributed.com*, July 2023. Archived at perma.cc/7LPK-TP7V

[13] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. At *4th Annual Symposium on Cloud Computing* (SoCC), October 2013. doi:10.1145/2523616.2523633

[14] Richard M. Karp. Reducibility Among Combinatorial Problems. *Complexity of Computer Computations. The IBM Research Symposia Series*. Springer, 1972. doi:10.1007/978-1-4684-2001-2_9

[15] J. D. Ullman. NP-Complete Scheduling Problems. *Journal of Computer and System Sciences*, volume 10, issue 3, June 1975. doi:10.1016/S0022-0000(75)80008-0

[16] Gilad David Maayan. The complete guide to spot instances on AWS, Azure and GCP. *datacenterdynamics.com*, March 2021. Archived at archive.org

[17] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-Scale Cluster Management at Google with Borg. At *10th European Conference on Computer Systems* (EuroSys), April 2015. doi:10.1145/2741948.2741964

[18] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. At *9th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), April 2012.

[19] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache Flink™: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, volume 38, issue 4, December 2015. Archived at perma.cc/G3N3-BKX5

[20] Mark Grover, Ted Malaska, Jonathan Seidman, and Gwen Shapira. *Hadoop Application Architectures*. O'Reilly Media, 2015. ISBN: 978-1-491-90004-8

[21] Jules S. Damji, Brooke Wenig, Tathagata Das, and Denny Lee. *Learning Spark, 2nd Edition*. O'Reilly Media, 2020. ISBN: 978-1492050049

[22] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. At *2nd European Conference on Computer Systems* (EuroSys), March 2007. doi:10.1145/1272996.1273005

[23] Daniel Warneke and Odej Kao. Nephele: Efficient Parallel Data Processing in the Cloud. At *2nd Workshop on Many-Task Computing on Grids and Supercomputers* (MTAGS), November 2009. doi:10.1145/1646468.1646476

[24] Hossein Ahmadi. In-memory query execution in Google BigQuery. *cloud.google.com*, August 2016. Archived at perma.cc/DGG2-FL9W

[25] Tom White. *Hadoop: The Definitive Guide*, 4th edition. O'Reilly Media, 2015. ISBN: 978-1-491-90163-2

[26] Fabian Hüske. Peeking into Apache Flink's Engine Room. *flink.apache.org*, March 2015. Archived at perma.cc/44BW-ALJX

[27] Mostafa Mokhtar. Hive 0.14 Cost Based Optimizer (CBO) Technical Overview. *hortonworks.com*, March 2015. Archived on archive.org

[28] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. At *ACM International Conference on Management of Data* (SIGMOD), June 2015. doi:10.1145/2723372.2742797

[29] Kaya Kupferschmidt. Spark vs Pandas, part 2 – Spark. *towardsdatascience.com*, October 2020. Archived at perma.cc/5BRK-G4N5

[30] Ammar Chalifah. Tracking payments at scale. *bolt.eu.com*, June 2025. Archived at perma.cc/Q4KX-8K3J

[31] Nafi Ahmet Turgut, Hamza Akyıldız, Hasan Burak Yel, Mehmet İkbal Özmen, Mutlu Polatcan, Pinar Baki, and Esra Kayabali. Demand forecasting at Getir built with Amazon Forecast. *aws.amazon.com.com*, May 2023. Archived at perma.cc/H3H6-GNL7

[32] Jason (Siyu) Zhu. Enhancing homepage feed relevance by harnessing the power of large corpus sparse ID embeddings. *linkedin.com*, August 2023. Archived at archive.org

[33] Avery Ching, Sital Kedia, and Shuojie Wang. Apache Spark @Scale: A 60 TB+ production use case. *engineering.fb.com*, August 2016. Archived at perma.cc/F7R5-YFAV

[34] Edward Kim. How ACH works: A developer perspective — Part 1. *engineering.gusto.com*, April 2014. Archived at perma.cc/F67P-VBLK

[35] Zhamak Dehghani. How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh. *martinfowler.com*, May 2019. Archived at perma.cc/LN2L-L4VC

[36] Chris Riccomini. What the Heck is a Data Mesh?! *cnr.sh*, June 2021. Archived at perma.cc/NEJ2-BAX3

[37] Chad Sanderson, Mark Freeman, B. E. Schmidt. *Data Contracts*. O'Reilly Media, 2025. ISBN: 9781098157623

[38] Daniel Abadi. Data Fabric vs. Data Mesh: What's the Difference? *starburst.io*, November 2021. Archived at perma.cc/RSK3-HXDK

[39] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. At *11th Annual Conference on Innovative Data Systems Research* (CIDR), January 2021.

[40] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, volume 33, issue 8, pages 103–111, August 1990. doi:10.1145/79173.79181

[41] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning Fast Iterative Data Flows. *Proceedings of the VLDB Endowment*, volume 5, issue 11, pages 1268-1279, July 2012. doi:10.14778/2350229.2350245

[42] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. At *ACM International Conference on Management of Data* (SIGMOD), June 2010. doi:10.1145/1807167.1807184

[43] Richard MacManus. OpenAI Chats about Scaling LLMs at Anyscale's Ray Summit. *thenewstack.io*, September 2023. Archived at perma.cc/YJD6-KUXU

[44] Jay Kreps. Why Local State is a Fundamental Primitive in Stream Processing. *oreilly.com*, July 2014. Archived at perma.cc/P8HU-R5LA

[45] Félix GV. Open Sourcing Venice – LinkedIn's Derived Data Platform. *linkedin.com*, September 2022. Archived at archive.org