The web and HTTP/REST based APIs has made the request/response style of interaction so common that we take it for granted -- but it is not the only way of building systems.
Three different kinds of systems:

- Online systems/services
  - a service waits for a request from a client to arrive
  - when one is received, the service tries to handle it as quickly as possible and sends a response back
  - *response time* is the primary metric of performance of a service and availability is often very important
- Offline systems/Batch processing systems
  - A batch processing system takes a large amount of input data, runs a *job* to process it, and produces some output data
  - jobs take a while -- from a few mins to a few days -- so there isnt usuall a user waiting for the job to finish
  - instead batch jobs are scheduled to run periodically
  - *throughput* is the primary performance metric -- the time it takes to crunch through an input dataset of a certain size
- Near-real-time systems/Stream processing
  - Stream processing is between online and offline processing
  - like a batch syhstem, a stream processor consumes inpurs and produces outputs
  - however, a stream processing system operates on evetns shortly after they happen, whereas a batch job operates on a fixed set of input data
  - this difference alloes stream processing systems to have lower latency than the equivalent batch systems -- stream processing builds on batch processing

batch processing is an important building block in our quest to build reliable, scalable, and maintanable applications.
For example, MapReduce was a batch processing algorithm published in 2004
- a farily low level programming model compared to the parallel processing systems developed for data warehouses previously, but it could enable significant advances in the scale of processing on commodity hardware

To begin with, we look at batch processing using Unix tools.

A single mapreduce job is comparable to a single unix process - it takes one ore more inputs and produces one or more outputs

- output files are written once, in a sequential fashion, without modifying any existing part of a file once it has been written

While Unix tools used stdin and stdout as input and output, mapreduce jobs read and write files on a distributed file system.

- in Hadoop's implementation of MapReduce, that filesystem is called HDFS ≡ hadoop distirbuted file system
- various other distributed file systems besides this exist
  - object storage services like Amazon S3, Axure blob storage, are similar inmany ways
  - analogous to HDFS
- in this chapter, we use HDFS as a running example

HDFS is based on shared-nothing principle, as opposed to the shared-disk approach of Network Attached Storage and Storage Area Network architectures
- shared disk storage needs custom hardware; however, shared nothing only needs computers connected by a conventional datacentre network

HDFS cnosists of a daemon process runnin on each machine, exposing a network service that aloows other nodes to access files stored on that machine
- a central servier called the *NameNode* keeps track of which file blocks are stored on which machine
- Thus, HDFS conceptually creates one big filesystem that can use the space on the disks of all machines running the daemon
- file blocks are replicated on multipl machines to tolerate machine and disk failures
  - this replciation can mean simple several copies of the same data on multipl machines, or an erasure coding scheme that allows lost data to be recovered with less overhead than full replication

MapReduce is a programming framework that allows you to write code to process large datasets in a distributed filesystem like HDFS.
- read a set of input files and break it up into records -- for e.g. each line in a file can be a record
- call a mapper function to extract a key and value from each record
  - the scheduler tries to run each mapper on one of the machines that stores a replica of the input file, provided that machine has enough spare ram and cpu resources to run the map task -- *putting the computation near the data* saves copying the input file over the network, reducing network load and increasing locality
  - the number of map tasks is determined by the number of input file blocks
  - in most cases, the application code that should run in the map task is not yet present on the machine that is assigned the task of running it, so the MapReduce frameowkr first copies the code to the appropriate machines
  - then it starts the map task and begins reading the input file, passing one record at a time to the mapper callback
  - the output of the mapper consists of k-v pairs

- sort all k-v pairs by key -- this is implicitly handled by the MapReduce framework
    - the k-v pairs need to be sorted, but the entire dataset is likely too large to fit into memory. So two stage sort
        - first, each map task partitions its output by reducer based on the has of the key
        - second, each of these parititions is written to a sorted file on the mapper's local disk -- similar to how the SSTables and LSM-Trees are written to disk
    - once a mapper finishes reading its input file and writing its sorted output files, the MapReduce scheduler notifies the reducers that they can start fetching the output files from that mapper
    - the reducers connect to each of the mapeprs and downlaod the files of sorted k-v pairs for their partition -- this process of partitioning by reducers, sorting, and copying data partutions from mappers to reducers is called *shuffling*
- call a reducer function to iterate over the sorted k-v pairs
    - the number of reducer tasks is typically configured by the developer
    - to ensure that all k-v pairs with the same key end up at the same reducer, the frameowrk uses a hash of the key to determine which reduce task shoudl receive a particular k-v pair

A single MapReduce job can only handle so a small range of problems
- e.g. if you need a second round of sorting, you need to chain MapReduce jobs = known as a workflow

Workflow schedulers for Hadoop have been developed -- Azkaban, Airflow

Joins

- reducer side joins: sort merge
    - sort by key happens mapper side -- writes sorted files partitioned per reducer
    - reducer gets partitioned files and merges them
    - generic sort that will always work -- no assumptions necessary about the data
- mapper side joins
    - quicker, saves shuffle, but requires you to make assumptions about the shape of the data -- size, how it is partitioned, etc
    - broadcast hash join
        - if one table can fit into memory it can be sent to every mapper where it is stored as an in memory hash table
        - each mapper reads one file from the larger dataset, looks up has table for the join key, and writes out the joined data
    - partition hash join
        - even if a table cannot fit into memory, if both join tables are partitioned on the same key and using the same hash function, then all the data to be

joined together can be brought to the same machine
    * Q: is it advantageous to sort?

Mapreduce drawbacks

- persisting intermediate state between consecutive mapreduce jobs -- happens via file write, which is time consuming
    - A lot of map jobs are redundant -- should just pass the corresponding reduce function logic into the upstream reduce function
- low level APIs are tricky

Dataflow abstractions like Spark help here

- rdd for fault tolerance: it stores the lineage of data so far so that even if a task crashes, it can be recovered by recomputing the data

Hive = SQL query execution engine using map reduce framework on HDFS