Data Pipeline Scalability

Learning objectives

In this topic, we will go over the following concepts:

- Understand the requirement of having scalable data pipelines.
- Discuss principles for building scalable software and data pipelines.
- Factors to consider when building scalable data pipelines.

Table of contents

1. Scalability in software

Scalability in software refers to an application or system's ability to continue functioning at optimal levels regardless of increased workloads. This scalability is normally achieved through the provisioning of computing resources or the scaling of the application's capabilities and functionality.

In the context of data pipelines, we can define scalability as the capacity to retain functionality in response to changes in volume, velocity, or variety of the incoming data.

Data pipelines typically ingest data that drive business-critical processes or real-time insights. Think of the live traffic features on Google Maps. Moreover, data pipelines are generally set up to be automated and, as a result, should be able to respond to changes in load without human intervention. These two characteristics of data pipelines mean that we have to be able to respond to change quickly to return to homeostasis.

In this section, we list the principles crucial for creating scalable software and provide a few practices that you can implement to achieve scalability.

Quite logically, if the amount of time it takes for processes to execute decreases, then cumulatively, throughput and by extension scalability will increase. This is easier said than done, and it might not be immediately obvious where to change your application to decrease processing time.

1.1. Caching

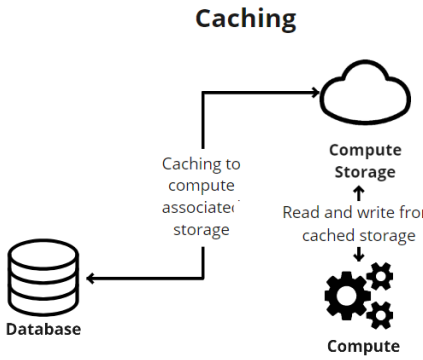

Figure 1: Caching is the process of storing a copy of the data close to the computing resources for quick access.

Traditionally, when working with SQL or NoSQL databases, applications had to fetch data from database servers, use the data in calculations and transformations, and discard the data as soon as the calculations were complete. This meant that every time calculations needed to be performed, data had to be fetched again. Caching is the process of storing a copy of the data within a local environment. For example, when reading data from a database to perform processing on, instead of retrieving data from the database every time, we store a local copy of the required data on a virtual machine doing the processing or on blob storage with low latency. Instead of fetching the data repeatedly, caching data reduces the overhead in transferring it. Within the context of scaling for data pipelines, this will decrease processing time for each data pipeline, thus, increasing throughput.

1.2. Collocation

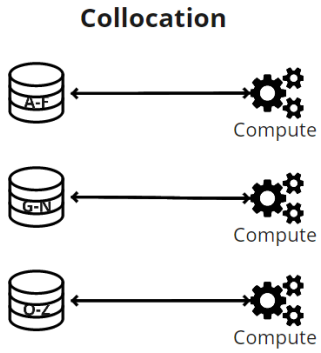

Figure 2: Collocation ensures that the data to be processed are located close to the computational resources performing the processing.

Collocation is the concept of keeping the data physically close to the computing resources that will be performing the transformations, not fetching and discarding it as described above. This is not just fetching the full dataset and storing it locally, but ensuring that the correct data partition is located close to the computational resources that will be performing the processing when doing parallel processing. This also means keeping data that need to be processed or aggregated together on the same partition and, therefore, processed on the same virtual machine within a cluster. Keeping the correct data and code nearby reduces the data transfer overhead. Similar to caching, by reducing the data transfer latency, we are decreasing processing times, which helps to increase throughput and scalability.

1.3. Pooling

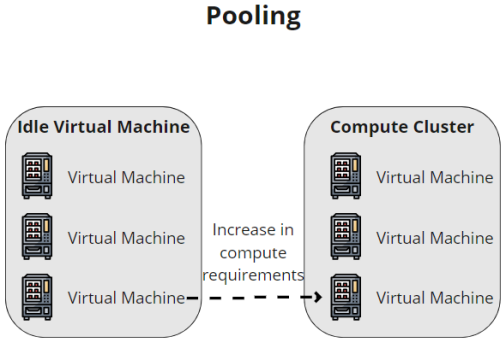

Figure 3: Pooling involves having some resources on standby to start processing as soon as the load increases.

Pooling is the centralisation of quickly accessible resources. Instead of virtual machines allocated to specific clusters, a larger pool of idle virtual machines is available and ready to attach to a cluster and start processing as soon as required. Having a pool from which to allocate virtual machines to clusters allows for more resources to be available overall, reduces the time taken to provision resources, and may also reduce cost. Depending on the infrastructure used to run a pipeline, pooling can also be applied to speed the pipeline up or reduce cost. In contrast to caching and collocation, pooling directly influences the ability of data pipelines to scale.

> Did you know?
>
> Performing processing on big data may require lots of computational resources. It's not always practical to use larger and larger virtual machines, but virtual machines can be organised into *clusters* to accommodate the increased processing demand. Clusters are a collection of virtual machines, enabling massively parallel processing.

1.4. Writing efficient code

Not all code is written equally, and it's important to ensure all code runs quickly and efficiently. The choice of algorithm can have a large impact on processing speed, and so will the use of high-performance libraries or changes in memory management. Other ways to improve code efficiency involve checking for:

- Unnecessary code or code performing redundant processing.
- Unused variables.
- Inefficient use of conditionals and loops.
- Suboptimal memory and storage usage.
- Use of reusable components.
- Effective error handling throughout the application.
- Code practices for the tools and languages you are using.
- Optimal data access and data management.

Keeping the above in mind will overall decrease the processing times within data pipelines, increasing throughput. Writing efficient code is a responsibility of both the data engineering and data science teams.

1.5. Using parallelism and concurrency

Arguably, one of the most important practices in improving scalability is increasing the concurrency of the code and processing. Recent improvements in processing power and parallelisation enable the scaling of applications beyond anything previously possible.

Scaling out may be as simple as provisioning more servers to execute code and run your application (assuming the application was written using a compatible language and framework). However, some bottlenecks may not be apparent upfront and may slow down your application as soon as the load increases.

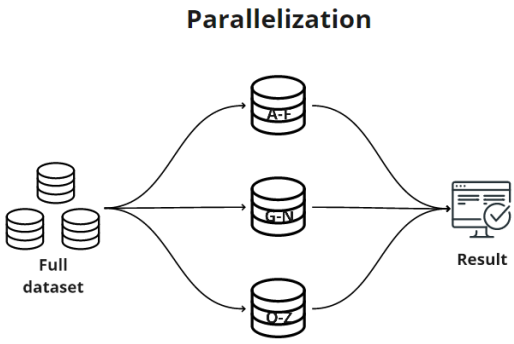

Figure 4: Parallelisation involves breaking the dataset down into chunks that can be processed.

Parallelisation is appropriate when we have *lots* of data on which the same transformation or processes need to be performed. In this case, it is possible to have a fleet of virtual machines (a.k.a. a cluster) performing the same transformation on different subsets of the data. Changing tasks from executing serially to in parallel has the possibility of greatly reducing processing times. Most data pipelining tools have parallelisation built in, and using parallel big data tools (for example, Apache Spark) can also facilitate this. Tools like Apache Spark make use of distributed computing, a process in which the components of an application are distributed across the various virtual machines in a cluster. This can be done either at the application code level (by splitting the actual application code across different virtual machines) or at the data level (by performing the same transformations on all machines but on different subsets of the dataset).

When building an application, it is important to partition it. The modularity increases the flexibility when scaling out is

required. Partitioning makes it simpler to parallelise parts of the application and swap them out, as they are implemented in a parallel way.

The level of parallelism will depend on the type of application, which is based on if the applicationkeeps track of state or not.

> ðŸ’¡ Remember
>
> **State** refers to the current mode or condition of an application. Being stateful means that the applicationâ€™s state is being tracked, and if the application terminates, it is possible to resume where it was terminated. This contrasts with a stateless application, where the internal state is not being tracked, and you will have to restart from the beginning if the application is interrupted.

A stateless application can be scaled out by simply running more instances of the code. This is true because each instance of the code does not depend on other instances or internal states being tracked.

For example, for a data pipeline that ingests data from a source system, one pipeline run will not be dependent on the next run. This means that our pipeline is stateless and each run can be executed independently and scaled by just adding more nodes/clusters to execute the pipeline.

Stateful applications may require some communication between parallel runs. This needs to be handled carefully to avoid bottlenecks. We can achieve this by having data partitioned so that each partition deals with a subset of the data or work that does not have to interact or exchange data with other partitions. Scaling is more difficult since states must be propagated from current nodes to scaled (added) nodes on a cluster.

When building concurrent applications, we need to ensure that we do not cause bottlenecks by having serial steps in our applications. We also need to ensure that we do not have parallel execution that could lead to data corruption. Some practices that can help:

- When writing to a database, the database will be locked, preventing other users, pipelines, or parallel pipeline instances from writing to that specific database. Therefore, locks should be held for as short as possible.
- Do not have any resources (or others) in contention in the critical path. Have asynchronous processing or scheduling for work that may have levels of contention.
- Design for concurrency upfront. This enables an understanding of where the serial sections and possible pain points are within the pipeline and where they may be throughout the development process.

2. Design architecture and requirements

Implementing the first two principles introduced above will greatly increase scalability. Still, it is also quite necessary to know to which extent it is required, to stop efforts on scaling pipelines before getting to the point of diminishing returns. When designing an application, functional requirements are often known and provided upfront, for example, you need to build a data pipeline that transforms data from one source and writes the data to a database. We are typically not provided with non-functional requirements: performance, load, and scalability requirements.

For data pipelines, it is not always possible to know the non-functional requirements upfront, and these can change throughout the lifecycle of a pipeline. However, it is quite important to know the extent to which non-functional requirements may change over time and to design the application knowing that the data sources may change or the load may increase. One way this can be designed is through loosely coupled data pipelines. Loose coupling involves having well-defined contact points. These contact points, in turn, allow for services or processing steps to be changed out as requirements change.

Imagine building a pipeline that uses batch architecture but may change to streaming in the future. It will be essential to design the pipeline with well-defined contact points in mind, allowing the change from a batch application to a streaming application when needed.

Once the requirements are established, and to what level scalability is required for the specific application, we can start designing the application. We will address this in the next section.

To build a scalable application, the required architecture needs to be addressed in the initial design to build a scalable application. This is where the difference between scaling up and scaling out becomes apparent. If scaling is not considered earlier on, you will likely design an application that can easily scale up but not easily scale out.

Designing upfront means knowing the full set of requirements as stated above and incorporating them into the application. We also need to consider the complete application and not just view it as individual components. For example, only considering and optimising the ingestion components whilst neglecting to optimise data processing and output components could greatly speed up ingestion, but it will lead to a suboptimal solution â€“ local optimum â€“ with the overall pipeline processing time not increasing significantly. Instead, all components should be improved in concert to find a global optimum where we aim to improve the overall performance. An application or data pipeline will only be as fast as its slowest components, meaning it's very important to think about the application in its entirety upfront.

Some things to consider:

- How the data will arrive â€“ the origin and type of data will determine the velocity of the data. The timeliness and requirements for processing will determine the speed at which the data has to be processed.
- Choose the correct database for storage â€“ some NoSQL databases or cloud implementation of SQL databases provide superior ability to deal with a change in load.
- Use asynchronous processing, which allows you to provide a buffer, so you can fill during high loads and then process after scaling or at times of reduced load.
- Choose the correct processing tools â€“ using distributed processing tools such as Apache Spark or Apache Flink can provide upfront optimisation and scalability.
- Utilise scalable and serverless functions where possible, which theoretically can scale infinitely.

3. Factors influencing ease of scalability

Having had a look at the principles required to build scalable applications, we can now look at the factors that influence how complex it will be to build a scalable data pipeline.

We broadly divide these factors into three categories:

- The nature of the input data sources.
- The processing that has to be performed within the data pipeline.
- The specific cloud technology chosen to build the data pipeline on.

Generally, we have to ensure that these three components are balanced. If one of the three cause a bottleneck or delay, the overall pipeline efficiency will be reduced.

3.1. Data sources and scalability

The type of data flowing through the data pipeline will influence on the scalability, for example, streaming vs. batch. Generally, when data are ingested using a batch loading process, it should not have rapid change velocity but rather set times for when it gets ingested. As such, when working with batch loads, we know when the load will increase and we typically know the size of the data that will be ingested. As a result, we can provision infrastructure accordingly. However, if a batch process has not been designed well, it will not be able to scale effectively, and when the volume of a batch process changes, we may face some scaling challenges.

To make the above point clearer, let's consider an example. Say we design a batch process that ingests a file every hour. The batch process is authored in Python code and runs on a local computer. In this scenario, if there is a sudden increase in load, the pipeline will likely not be able to accommodate the change in load because of the following factors:

1. The program is written in Python, which does not natively support parallel processing.
2. It's running on a local computer and the scaling capability is limited to the machine's physical hardware.

When the data load increases for this data pipeline, each pipeline run will start to get longer, up to the point where runs will start overlapping and each run will either lockout subsequent runs from the database or cause other pipelines to fail.

In contrast to batch, streaming data is much less predictable. The volume or velocity of streaming data may change quite rapidly. Fortunately, the streaming architecture is designed to accommodate the possibility that the volume or velocity of the data may change at a moment's notice.

Let's use an example to explain this. Imagine we designed a streaming data pipeline that received data from a website that monitors disease prevalence globally. Our pipeline and application first went live in January 2020. The data we were initially interested in are the coordinates of the reported sick person. We built a data pipeline that ingested data from the website, using a message broker and Apache Flink for stream processing (more on processing later). Our data pipelines performed quite well at first, being able to cope with the load we initially estimated based on global disease rates. In March 2020, disease rates started to increase globally, which increased the volume and velocity of data. In our data pipeline, this caused the message queues to become flooded. Since we implemented a message queue that asynchronously ingests the data, we were not too concerned. We hypothesised that we could catch up on the processing once the load dissipates (a great feature of having a streaming data source). Within a couple of days, we realised that the load would not dissipate, and as a result, we had to scale. Luckily, message broker services scale quite effectively. We just have to provide more computational resources to process more messages, scaling out to deal with the increase in load. Thus, we could accommodate the increase in load after scaling, effectively track global disease rates, and be one of the first pandemic tracking applications during the COVID-19 pandemic.

Apart from the differences between streaming and batch loads, the specific nature of a data source also influences the need for a data pipeline to scale and how important scaling will be in the design of the data pipeline. For example, when designing a system that ingests data from IoT sensors that measure air temperature on the moon, it is expected that both the volume and velocity will remain constant. The only case where either of those will change is when there is a requirement for a higher granularity of data, ingesting data more frequently, or installing more sensors (which should happen less if the sensors are on the moon).

In contrast to and continuing the website traffic example, most applications and data sources involving human engagement

can change rapidly in volume or velocity and should be designed accordingly.

3.2. Downstream processing and scalability

The previous section showed the importance of designing your application to allow for quick ingestion of data. The next big consideration is to design the processing in such a way that it supports scaling effectively.

Within data pipelines, processing can mean simply parsing the data into a more appropriate format, for example, ingesting data in a JSON format and transforming it into a table format ready to be loaded into a relational database. Alternatively, processing can mean training and deploying a live machine learning model, which could involve any of these:

1. Formatting and cleaning the input dataset.
2. Training the model and creating a step to predict outcomes for new incoming data points.
3. Monitoring and retraining the models based on specific conditions (for example, change in performance metrics or a retraining schedule).

The first scenario will likely not require many computational resources and can probably be run on a serverless function that can theoretically scale infinitely. The second scenario is much more complex and will require setting up computing resources for developers to experiment and build models. The processing required for the end result is also more demanding, and we will likely have to provision a cluster that can scale based on demand to accommodate the change in load.

3.3. Cloud technologies for scalability

Finally, after considering all of the above principles and practices, choosing your toolbelt to build a data pipeline will impact how robust and scalable it will be. We prefer not to focus on specific tools but rather on classes of tools to solve certain problems. In this section, we will note some of the best tools currently available. Below we provide three common scenarios and toolsets available to solve each.

> âš™ï¸ A note on tools
>
> Only once the requirements have been drawn up and a high-level architecture has been designed for the solution should tools be selected or investigated. Tools change rapidly and may have new features every day, meaning you do not want to paint yourself into a corner by designing your solution around a specific tool or feature within a tool. Having the correct principles and practices in place will be more imperative than the tools used to build the solution.

3.3.1. Scenario 1: Ingest a daily weather report from an external API and output data to a database for analytical use

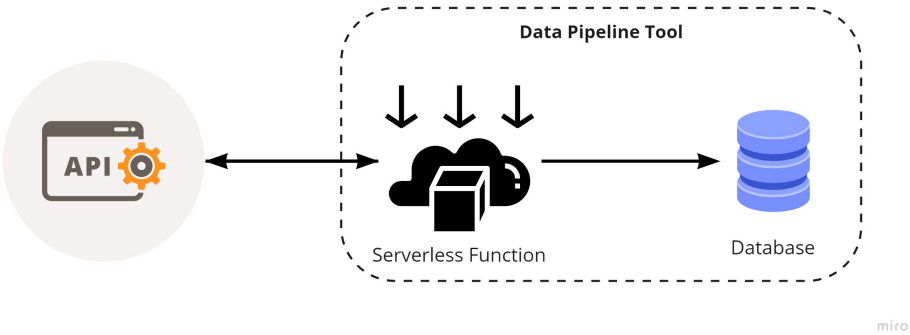

Figure 5. A simple data pipeline to retrieve data from an API and write it back to a database.

This scenario takes data from one source and outputs it to a sink location with minimal transformation. There is a lot less to consider with minimal transformation. The source system is an API, which in this instance, we can assume is stable and well documented and easy to interact with. The database we are writing to will be a relational database, given that the data will be used for analytical purposes.

Drawing on the principles above, we must ensure that the application is sufficiently decoupled, processing times are reduced, and data can be processed in parallel. To decouple the data pipeline, we have to decide on the optimal number of steps required in the data pipeline.

In this case, it would be either two or three pipeline steps, being the ingestion step (retrieving data from the API), a transformation step (getting data into the correct format for the database), and a loading step into the database. For simplicity, we assume that the data will be in the correct format when it comes from the API.

To ensure reduced processing times and sufficient parallel processing, we have to choose tools that allow for this. In the first

stage of our pipeline, calling the API, we can use serverless functions to interact with the API, which is highly parallelisable. We can use a data pipelining tool to orchestrate the API calls on a predefined frequency. Data pipelines typically also have functionality built-in to scale automatically. Finally, the data pipelining tool will also facilitate writing to the database. Most dedicated data pipelines have connectors to most types of databases available

If you anticipate that your calls will overlap, you can implement an asynchronous layer in the middle that will receive the messages from the API and then serve as a temporary location from where they can be processed serially.

3.3.2. Scenario 2: Receiving live data from IoT devices measuring atmospheric temperature for data science use

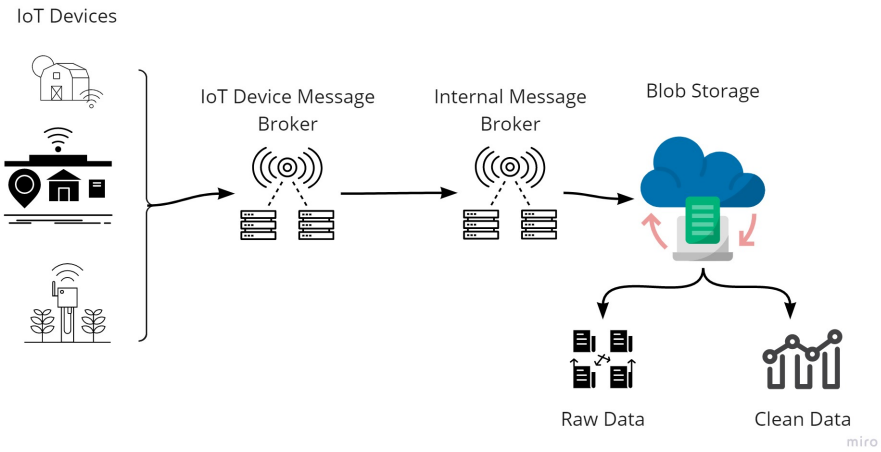

Figure 6. A simple pipeline to ingest live data from IoT devices and output back to a central data lake.

In this scenario, we are ingesting data from a live data source, which has additional considerations. We are also writing data for data science purposes, which means that we cannot perform any transformations on the data and need to provide the data in a raw format for data scientists to use.

How does this impact scalability? Similar to the previous scenario, we use the above principles to guide our implementation.

We can identify two separate steps: ingesting data from the IoT devices and outputting the data back for use by data scientists. In the first step, we need to consider how the data will be coming into our environment. This can be through querying an API to retrieve the data over HTTP or using IoT device-specific protocols such as MQTT. Both of these will be when you are in charge of the IoT devices. Sometimes you may have an external team maintaining the IoT devices, and they will interface with the devices. They will then store the data in a location you can pick it up from. This, again, could be through interfacing with an API, SFT, or even a message broker. Let's assume we have such a team, and they provide the data to us through a message broker.

In this case, to ensure scalability, we would implement a second message broker to consume from the IoT device message broker topics. Message brokers can, in many cases, be integrated directly with other sinks, such as blob storage or bespoke data lake solutions. In this scenario, we use blob storage as a storage location from where data scientists can utilise the data. We will organise the data into a data lake within the blob storage. This has significant benefits for data scientists, who will be able to access raw and cleaned data from there, while also allowing them the freedom to write back to the data lake.

3.3.3. Scenario 3: Retrieving user shopping patterns and serving recommendations back to users on an e-commerce platform.

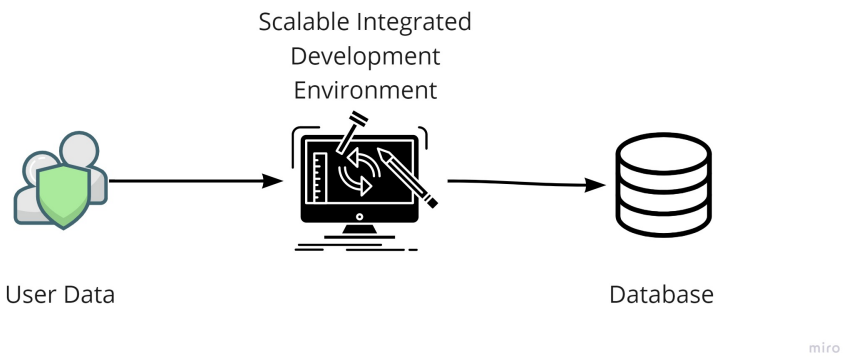

Figure 7. A simple pipeline to perform data science modelling and output a recommendation back to a user.

In this scenario, we are getting user behaviour data, we need to perform some data science, and then output the data from where it can be sent to the user. We will assume that this is a batch process and it does not happen in real-time.

In this case, we have three steps: ingesting the data, performing the data science, and outputting the data from where it can be displayed to the user. Fortunately for us, since this is an e-commerce platform, all the generated data will already be within our environments. This means user statistics, click rates, and other data that will be useful in a data science model will already be in a data lake or database, ready for us to use in our pipelines.

Something we have not dealt with yet is allowing other teams to work scalably within our data pipelines. For data science, we have to create a highly scalable environment, seeing that data science requirements may change quite rapidly. Solutions include having scalable clusters that can run code and notebooks, for example, Amazon EMR or Databricks. These tools allow for scaling rapidly and have a flexible user interface that is easy to interact with. Finally, we will output back to a relational database, which the front-end team can then pick up and display back to the end user. This is all orchestrated through orchestration pipelines.

4. Conclusion

In this topic, we had a look deep into the principles that should be applied when building scalable data pipelines. These principles are broadly transferable between any tools available for constructing data pipelines. We also discussed three possible scenarios and the tools you would consider to solve the challenges associated with each.

5. Resources

- Parallel pipelines in Azure Data Factory
- Building IoT data pipelines
- Scaling through loosely coupled pipelines
- Data pipeline design considerations