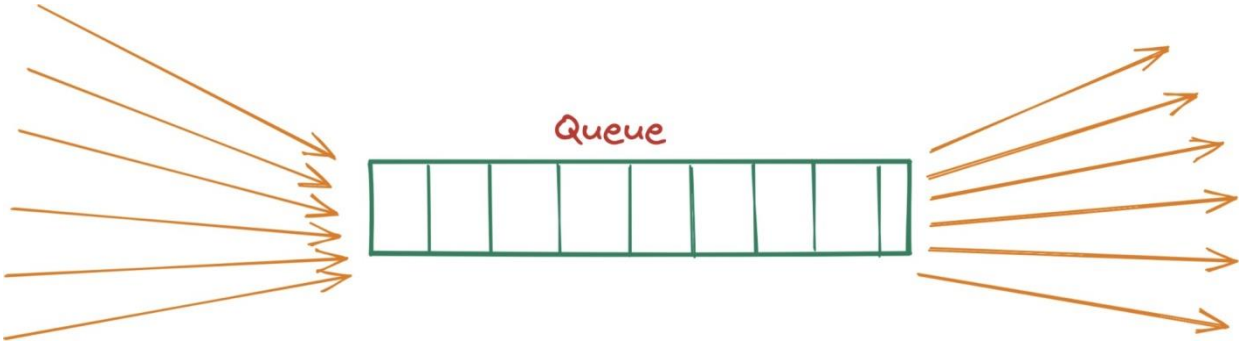
Design a Messaging Queue with High Throughput (non-distributed)

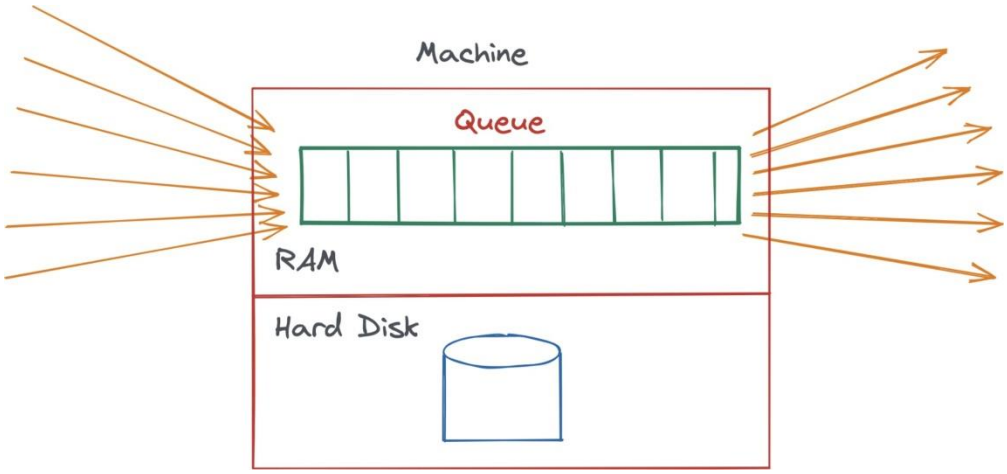


In this question, we design a queue in a single machine. This queue can handle things like:

- A Rood of tasks from a website like Facebook
- A stream of logs from a consumer app.

These use cases require a lot of throughput.

On one hand, this problem might seem trivial to you. After all, we just need to implement the data structure right? Initialize a Queue object and that's it.



You're right about that. If we simply initialize a queue in memory and read/write to it, that will be quite fast.

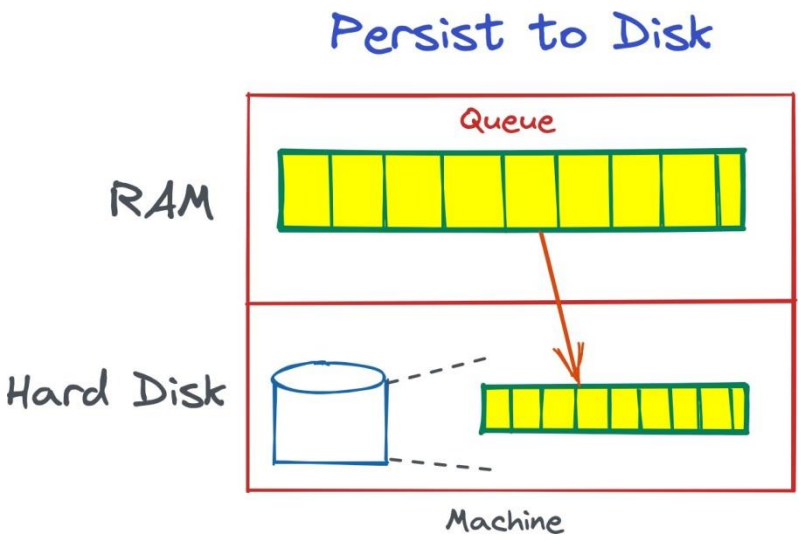
Why? Because in-memory reads/writes are quick. The problem comes when we have to write this to disk. So far, we don't have any requirement to write to disk, so everything is hunky dory.

Making our Queue Persistent

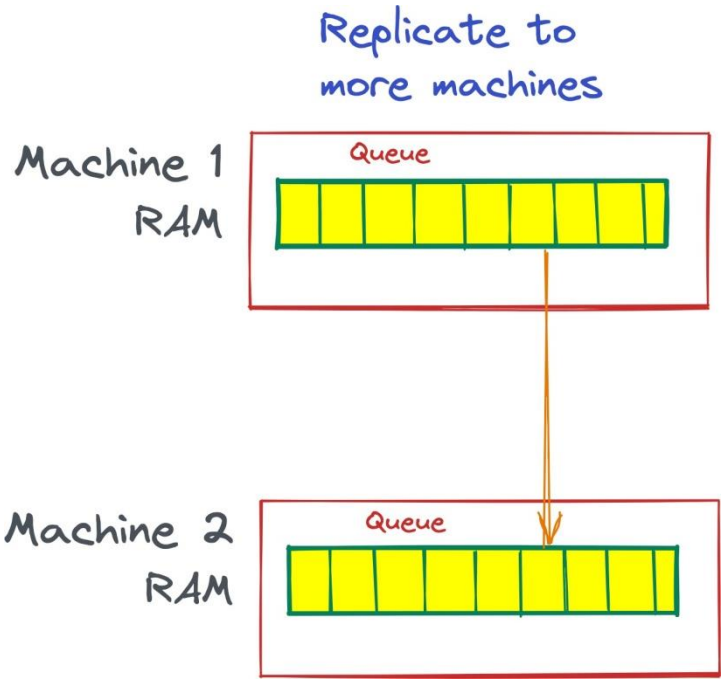
With our setup above, there is a problem - if the machine goes down or restarts, we will lose the entire queue. RAM goes away if the computer restarts. This is something we should handle, because failures are very common in systems. We need to safeguard against this.

There are 2 ways to deal with this:

1. Persist the queue to disk - even if the machine goes down, the queue can be reloaded from the hard drive.



2. Replicate the queue across multiple machines - if one machine goes down, the replica machine will still have the queue.



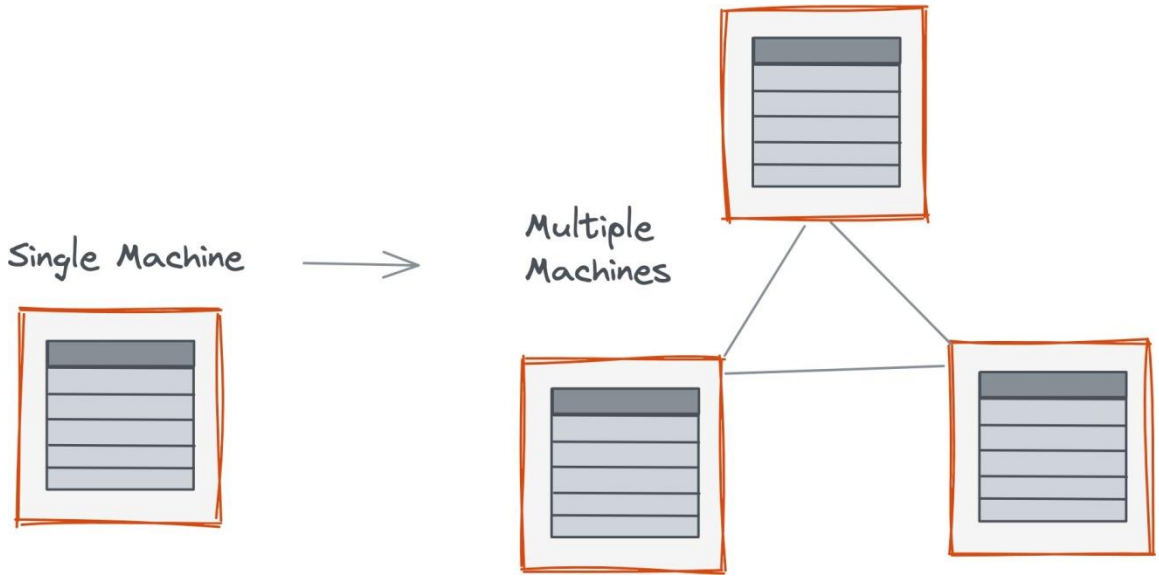
Both these approaches are used by queues in the industry. For the interview, you should know both!

We will start with Persisting to Disk. We will handle replication later on, when we distribute this queue to multiple machines.

As you may have guessed already, we are working towards designing a distributed queue - something like RabbitMQ or Amazon SQS.

Queue on a Single Machine

Let's 1rst start with a Messaging Queue on a single machine. Once the single machine model is clear, it will be easier to scale it horizontally. This is not very diIerent from how we scale a key-value store or a distributed hash table.



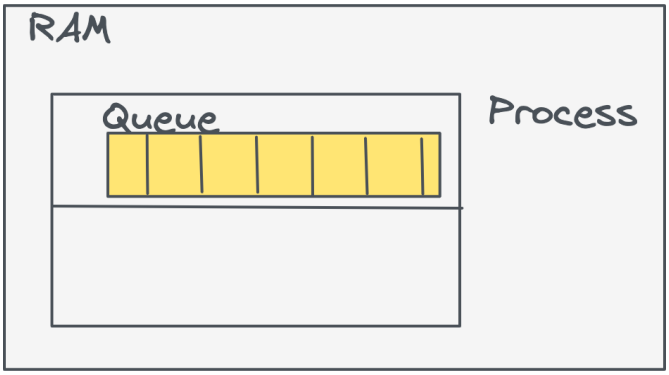
We implement it locally on a single machine, and then we 1gure out how to shard it across many machines. A distributed queue uses similar concepts.

Let's look at how a queue is implemented on a single machine.

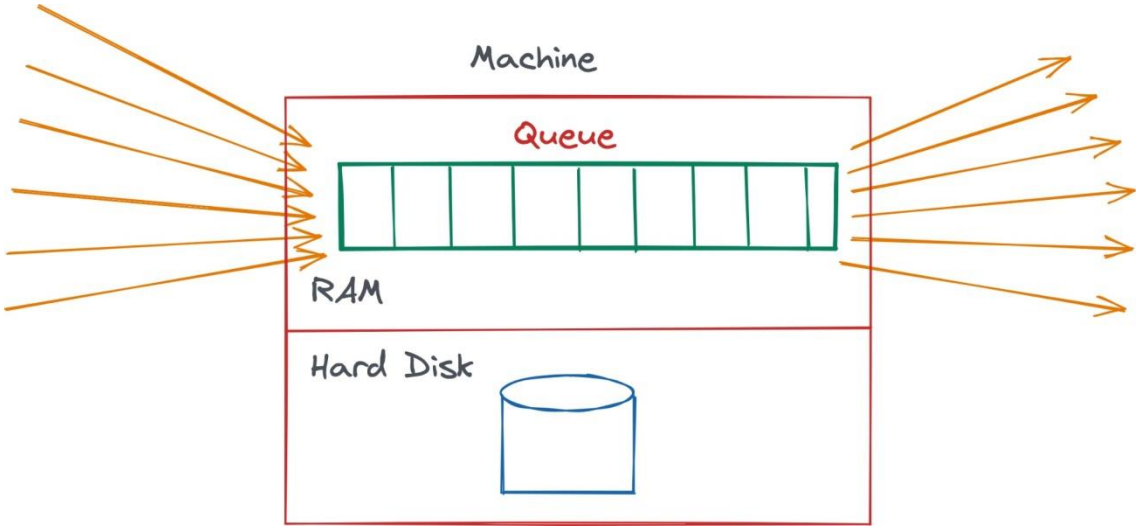
If you were to implement a queue using your local IDE, how would you do it? Well that's simple, you would probably make a Queue object in Java (or a List in Python). Something like this:

```
public class QueueManager<A> {
     Queue<A> queue = new Queue<>();

     // constructor

     // getter

     // setter
}
```

This will make an in-memory queue that will live in your program's Heap memory (in RAM).



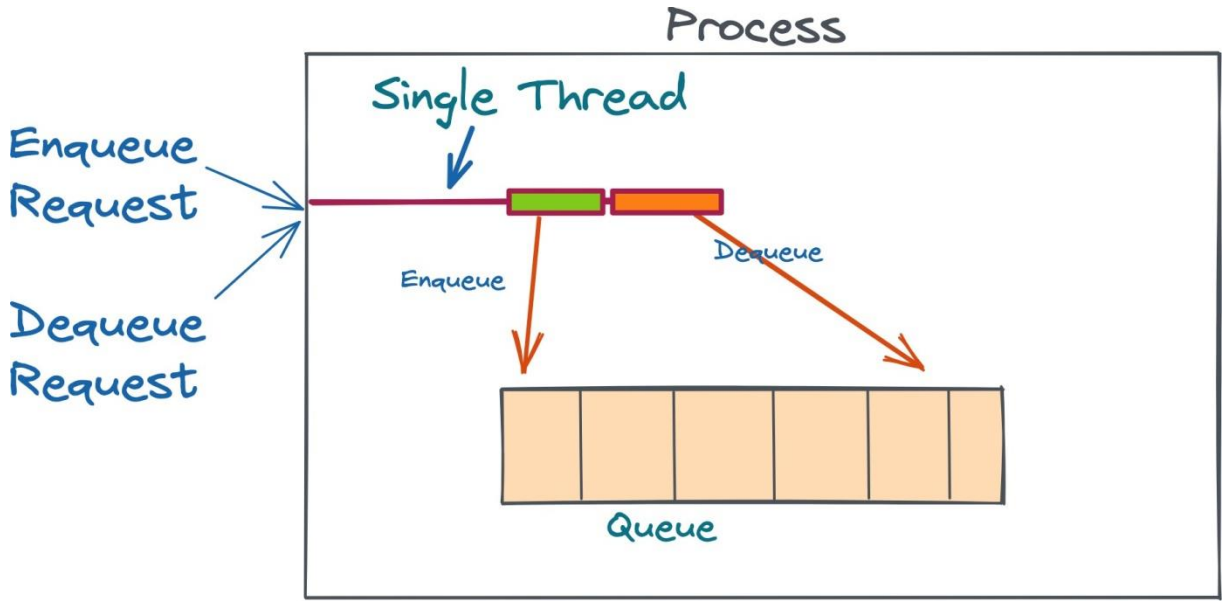
Heap space can be expanded as you add more data into the Queue, eventually taking all the RAM available.



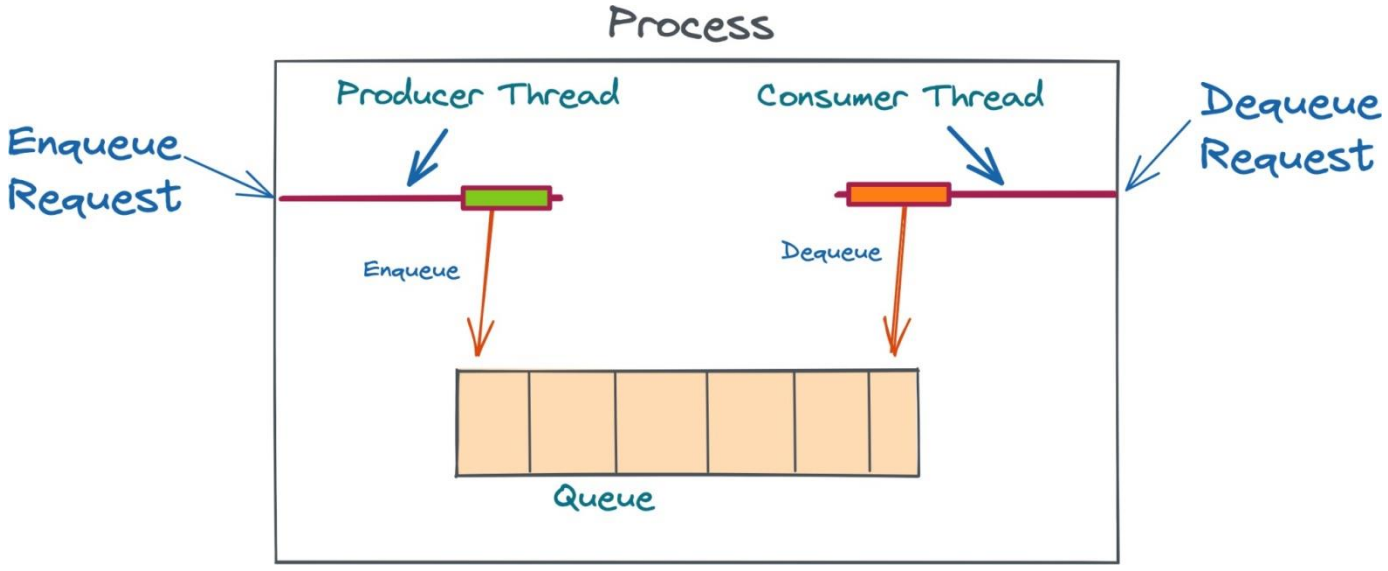
Reads and Writes will be very fast - because you read and write from memory.

Thread Design in our Queue

Let's go a bit deeper now - how do we structure threads and processes in this Queue? So far, we have a single process and a single thread. This thread does both - reads and writes. Or in this case, enqueue and dequeue.

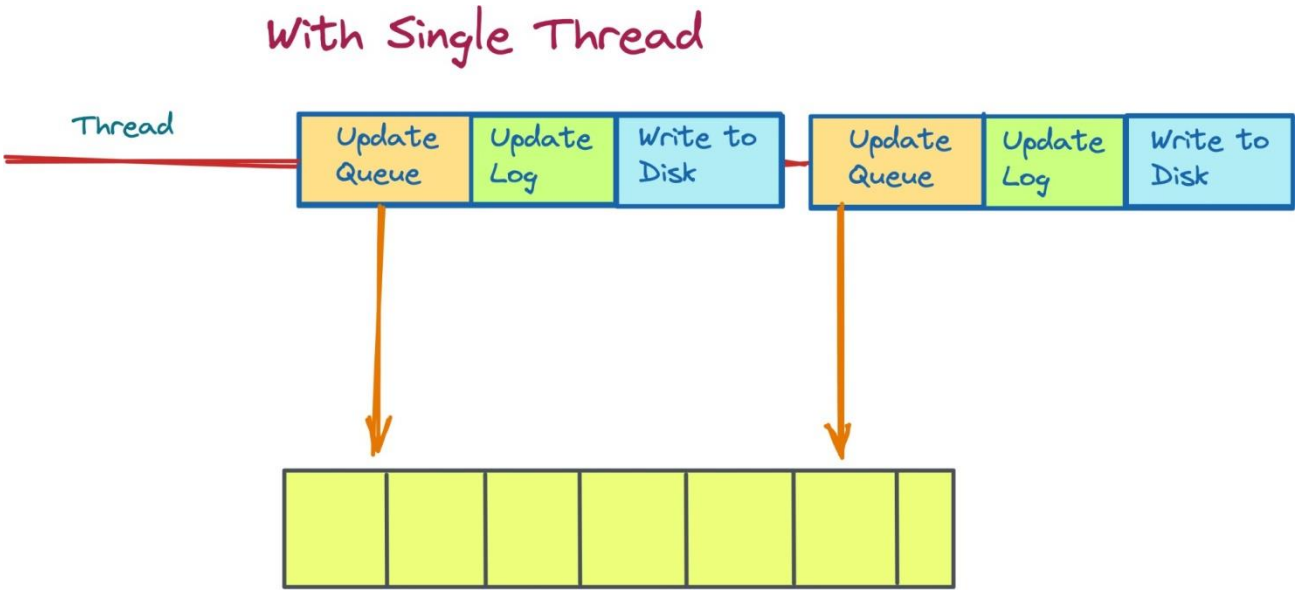


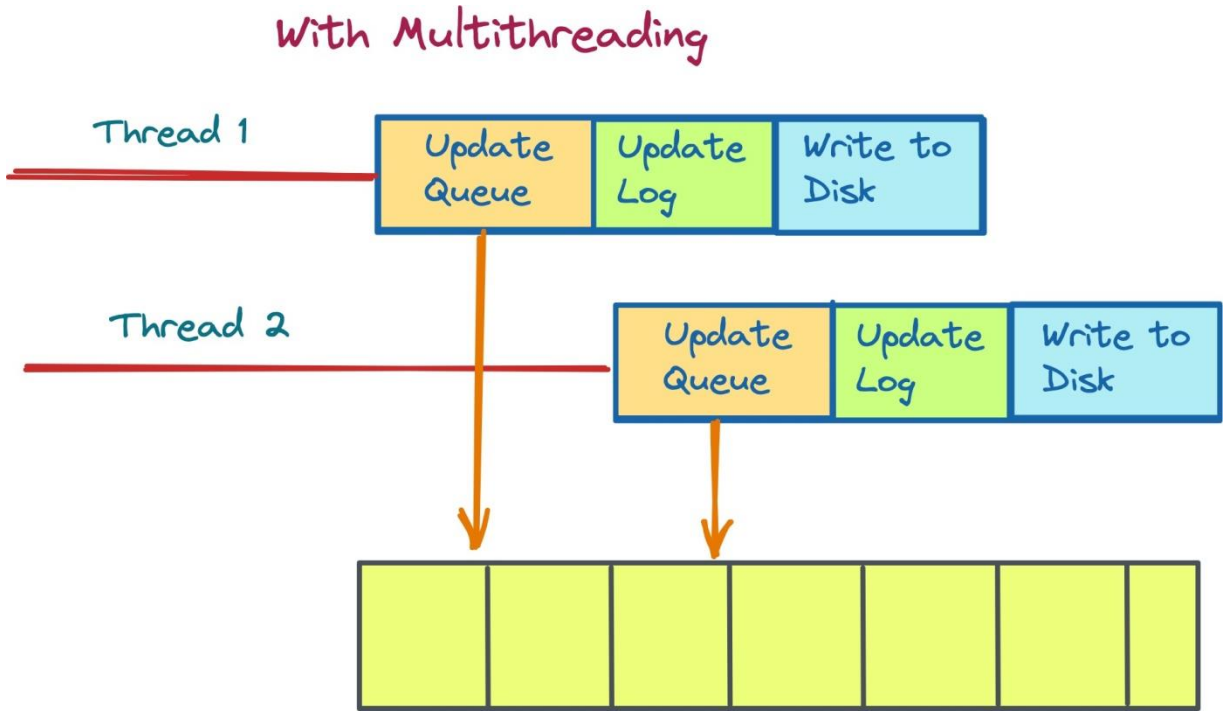
If we run two threads - one for read and one for write, it might be better, because reads and writes don't have to wait for each other. This way, one enqueue and one dequeue can happen at the same time.



In the way our Queue is structured right now, read and write operations are simple - simply modifying the single queue. But in reality, the read/write operation can involve

multiple steps - logging, persisting to disk, etc. For those use cases, having separate threads for reading and writing might be bene1cial, so that a read thread doesn't have to wait while a write thread is doing those other operations.



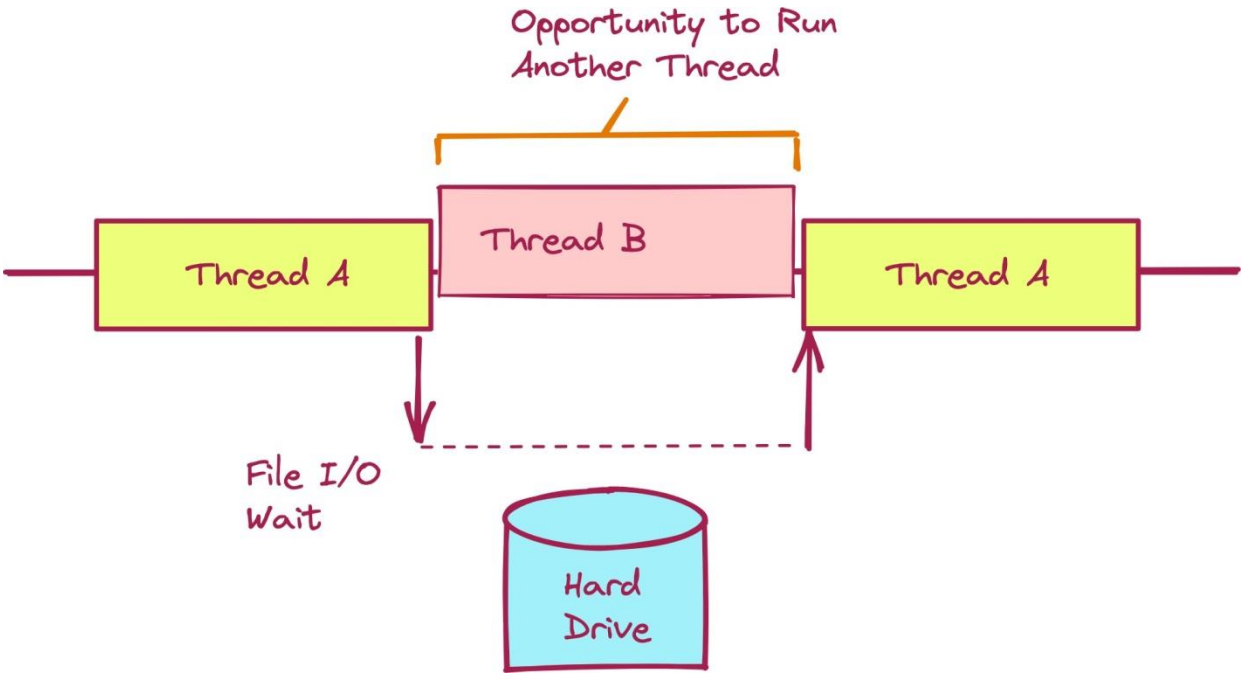


This becomes especially bene1cial if we have multiple processing cores. More on that below.

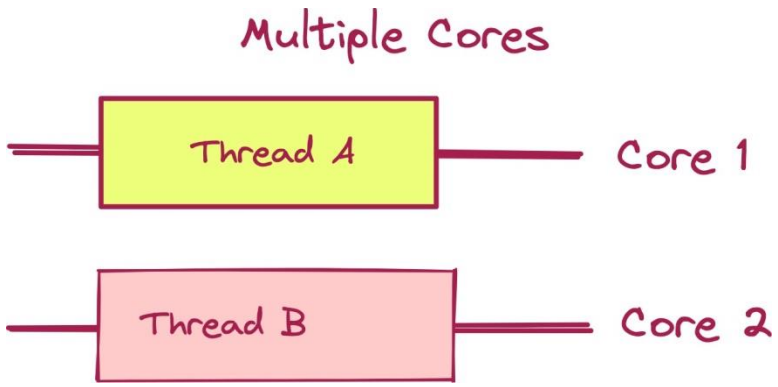
How does it differ with multiple cores?

If we have multiple cores, that's like having multiple CPUs, so two threads/processes can actually run in parallel.

In a single CPU core, threads give you the *illusion* of running in parallel. They run another thread when one thread is waiting for something. For example,



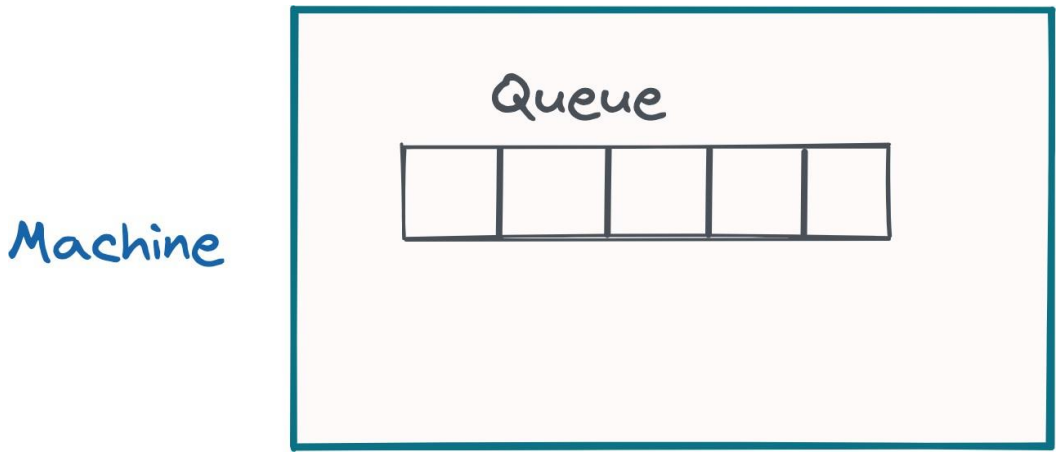
With multiple cores, you can actually run two threads in parallel.



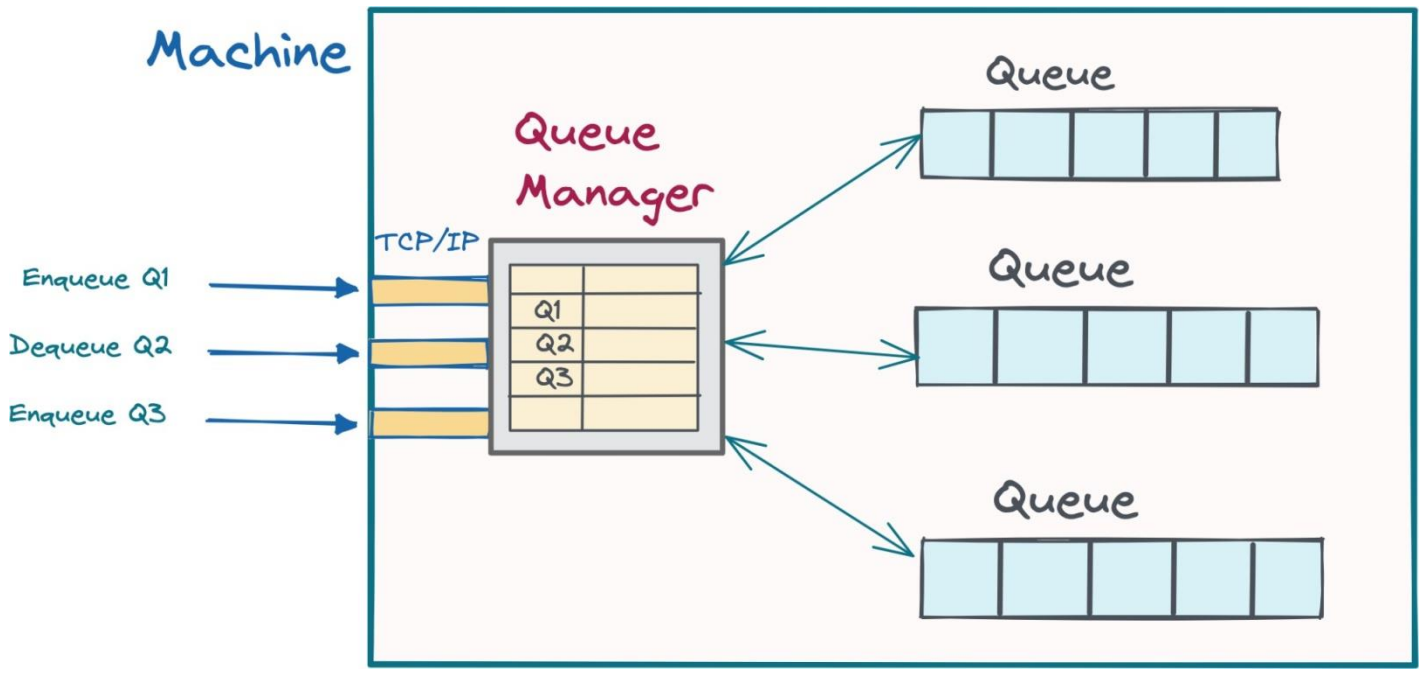
Why did we bring this up? Generally, it is good to know this concept in interviews. Adding parallelism is especially useful when you have multiple cores.

Multiple Queues on one machine

Ok so this is what our queue looks like right now:



You can have multiple queues in the machine. For this, you can add a QueueManager that initializes diIerent Queue objects and keeps track of each of them.



This Queue Manager can also maintain connections and serve as the interface to the queue. A client can send a request asking for an item in a particular queue - getItem(Q3), and the manager can call the Q3's dequeue function and get the item and send it to the client.

This way, we're able to create multiple queues in the single machine with high throughput. Since we're storing everything in memory, read/write operations are very fast.

Note:
Keep in mind that this may not be exactly how Queues are implemented in RabbitMQ in reality. The point of the system design question is to show how you would implement it from scratch, not to repeat the implementations of these well known systems.

Ok, so now we have a high throughput well functioning queueing system that can accept messages and give them quickly. Awesome. However, there are some ways we can improve this.

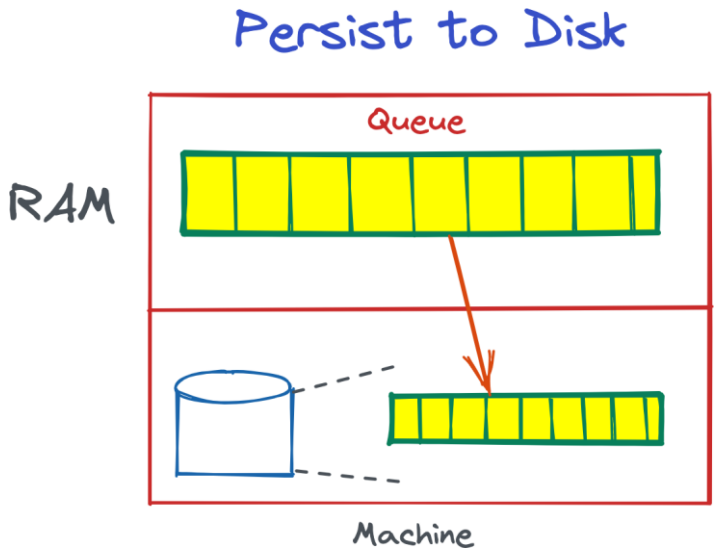
How can we improve this? Well for one, there is a big problem with this system. Can you spot it?

What if we want persistence?

We have a big problem with Data loss. In a Queue, our clients rely on us to not lose data. If the data is not super important, then it's ok. Or, if the data is backed up somewhere, then it's ok too. However, in our queue, because we are storing messages only in memory, as soon as the system restarts or if the process crashes (a common occurrence), our entire queue is lost.

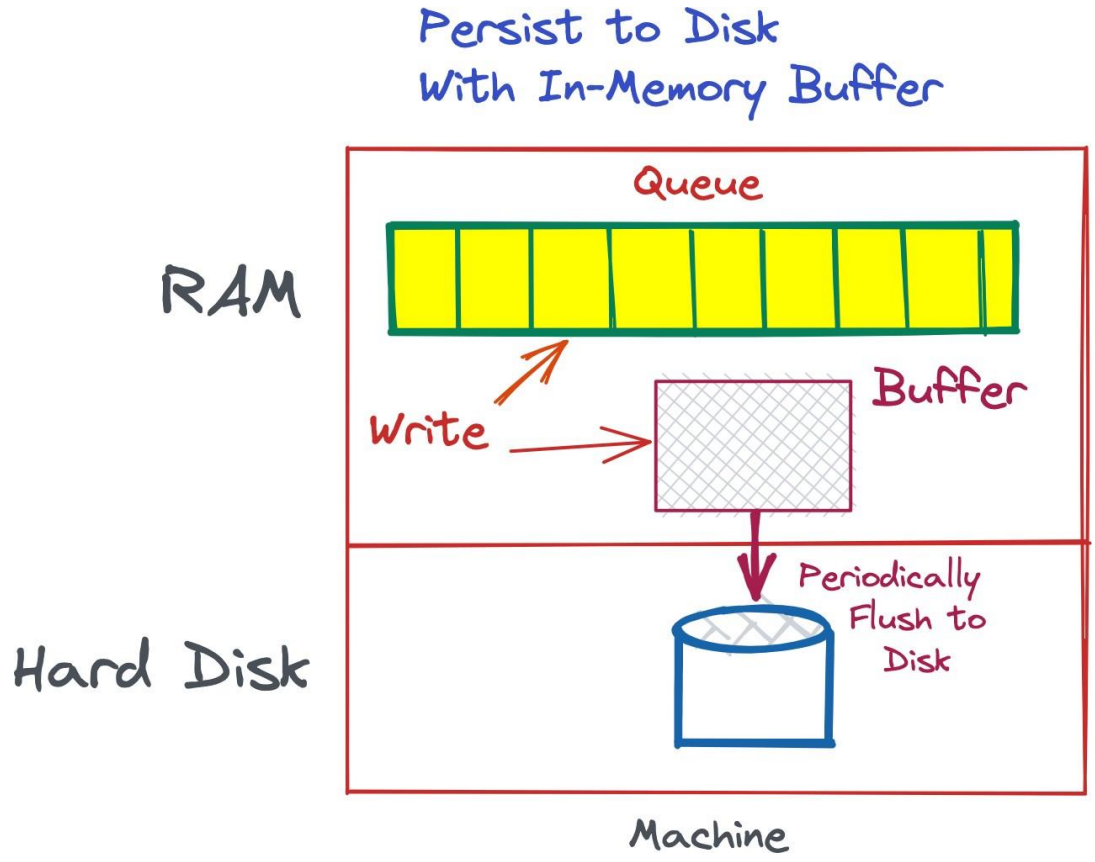
So, we need to add persistence to this queue. This is a natural next question the interviewer will ask.

Thankfully, adding persistence is a simple matter of backing up our queue to disk.



How can we back up our queue to hard drive? Well the solution is quite simple.

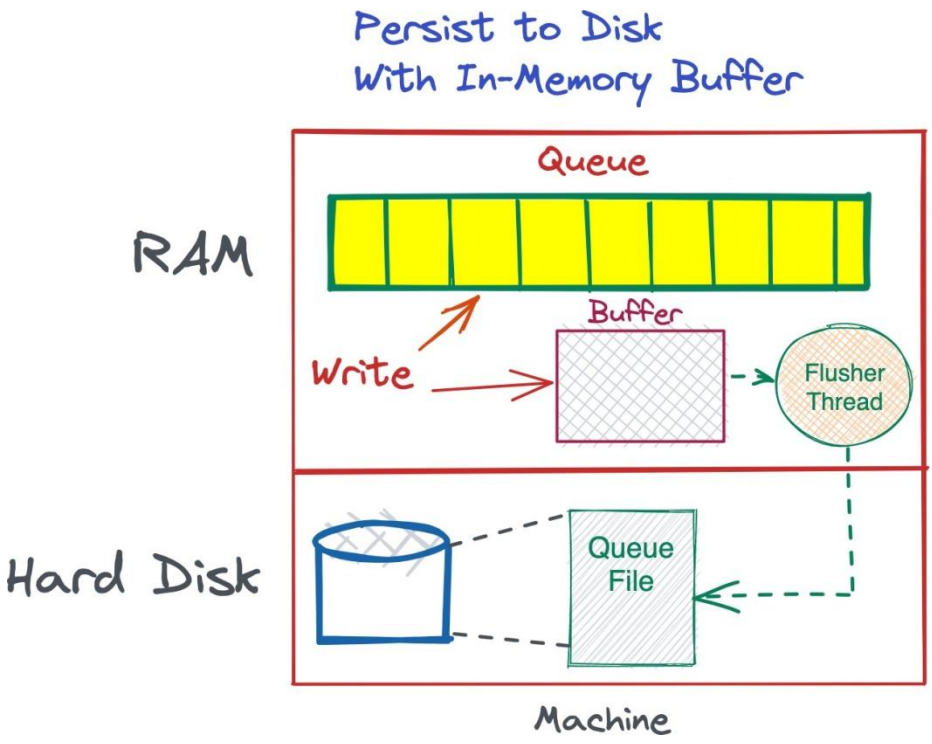
We can write new data to an in-memory buIer and periodically Rush the data to hard disk.



Why Rush the data periodically? Because we want to minimize the number of writes to disk. The more we can collect data and Rush together, the less load on our program.

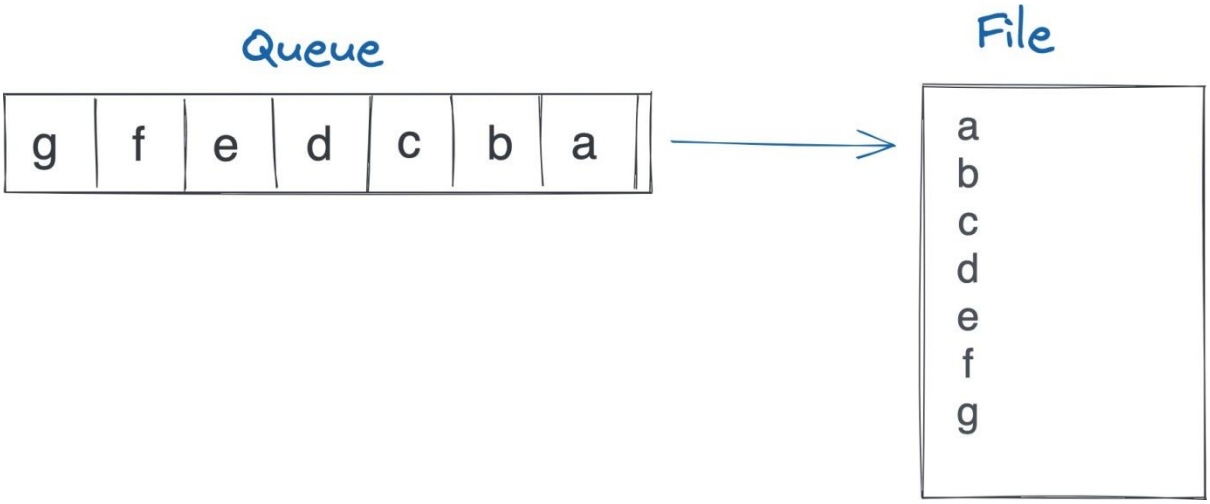
Depending on how often we want the data to be Rushed, we can set a Rush interval. At the interval, someone needs to take the buIer and write it's contents to disk. We can have a separate thread do this.

Each queue can have it's own 1le on disk.



On the disk, we have used a simple 1le to store queue contents.

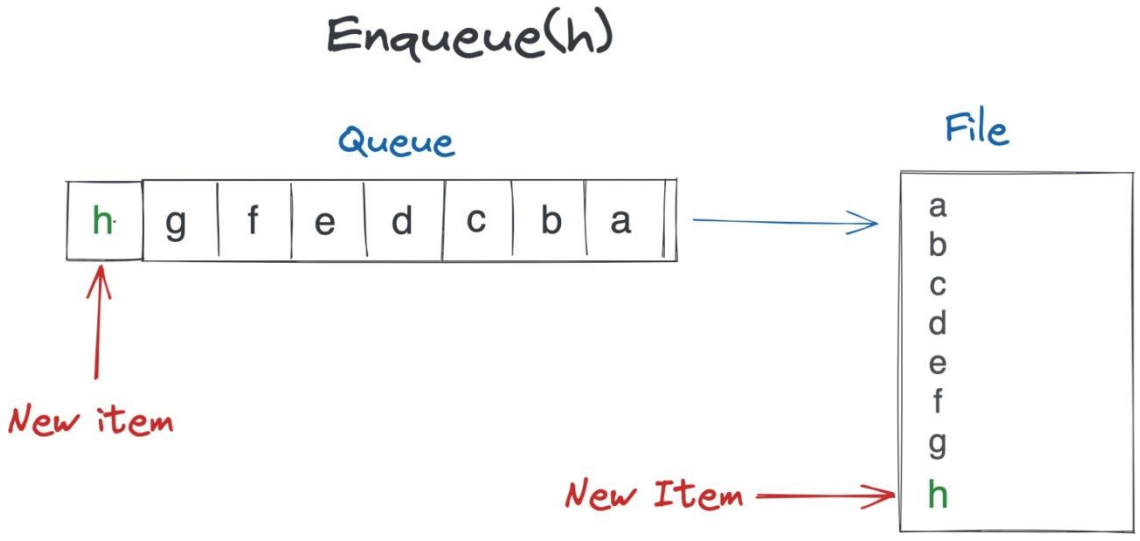
How is the queue represented in the file?



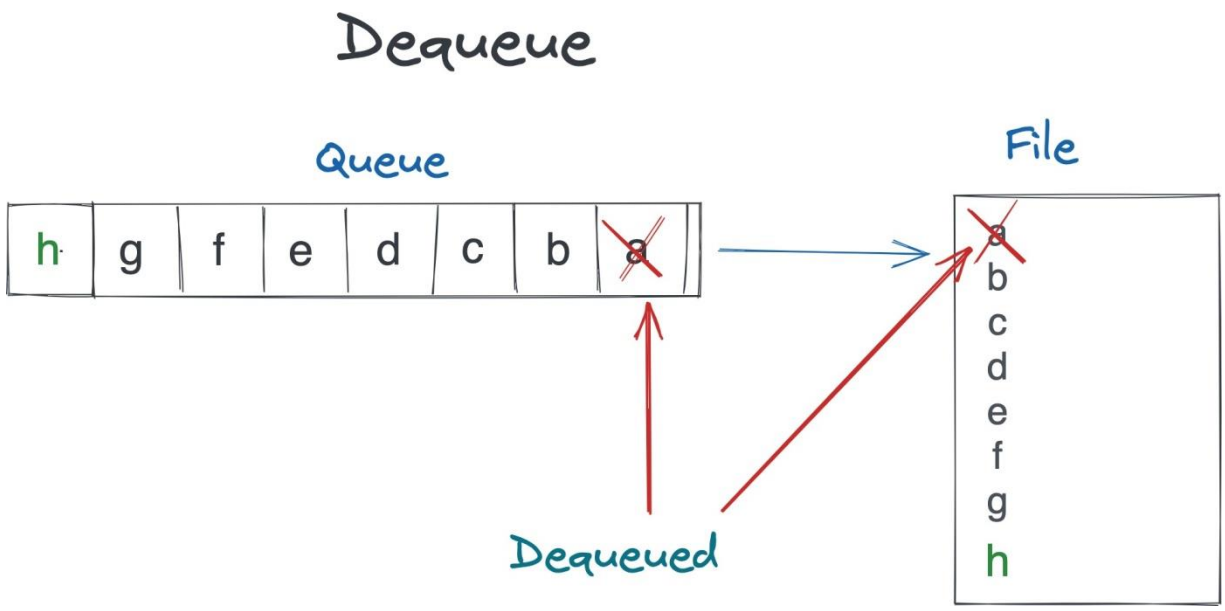
Every line is an item in the queue. We have put single characters in the queue. In reality, Queue items will more likely be long lines, or even JSON.

Instead of using a 1le, some people might use a simple DB - like SQLite. A 1le seems to be quite simple and well suited to this task though, so we'll pick this. Feel free to pick anything else.

When items are added to the queue, we append them to the end of the 1le.

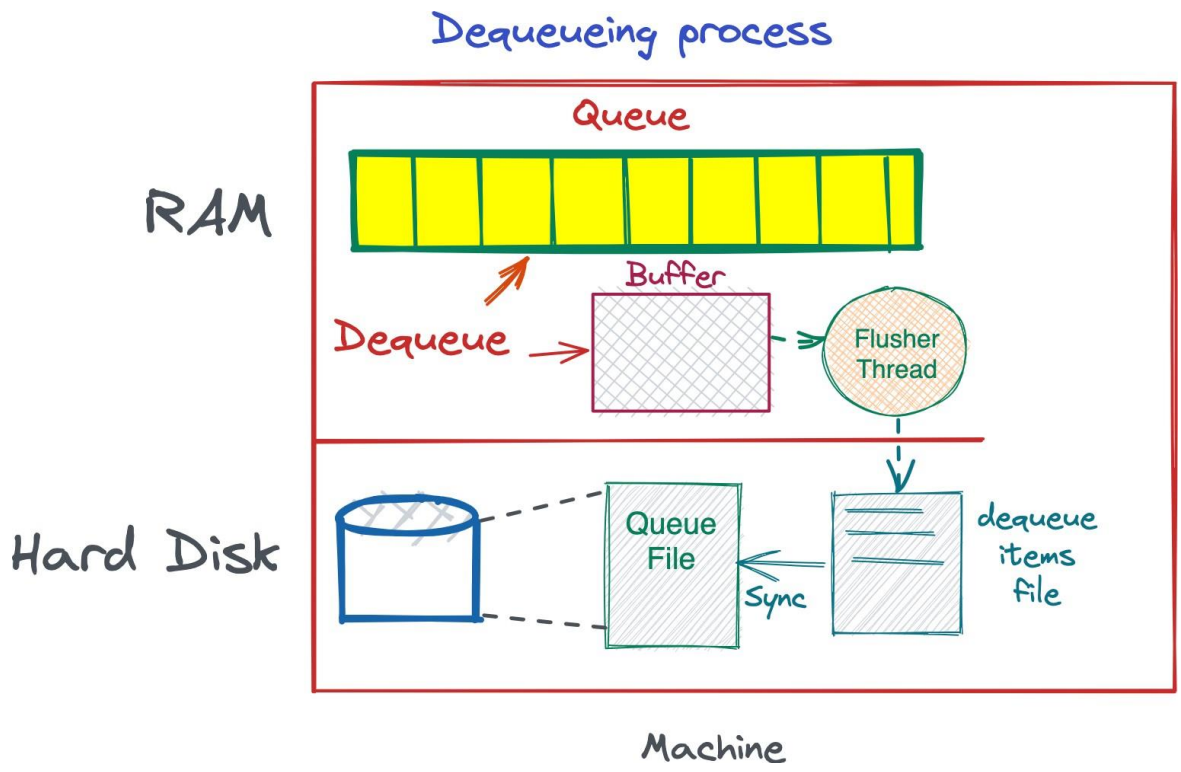


When items are deleted from the queue, we need to delete them from the beginning of the 1le.



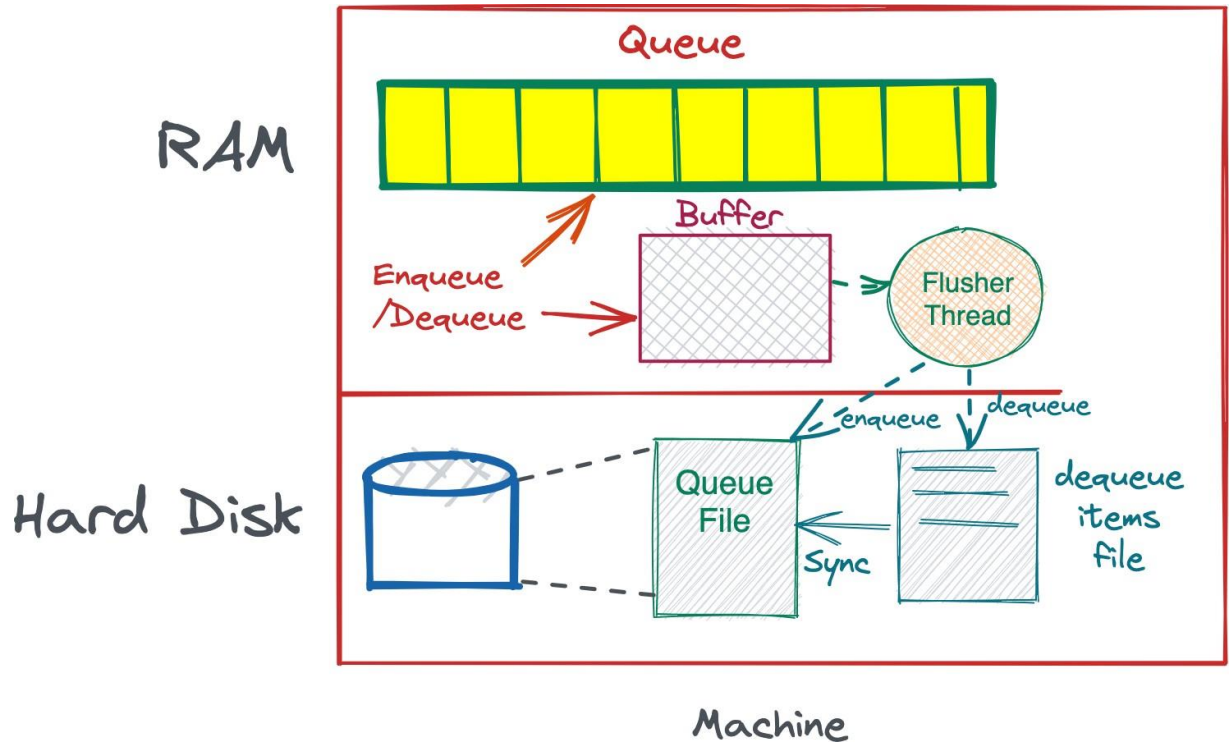
How can we make file deletions faster?

Now, removing items from the start of the 1le is more expensive because we have to rewrite the entire 1le. It makes sense to do this less often. To remove items from the queue's 1le, we can 1rst add it to another 1le of dequeued items. We can periodically Rush these items from the front of the 1le.

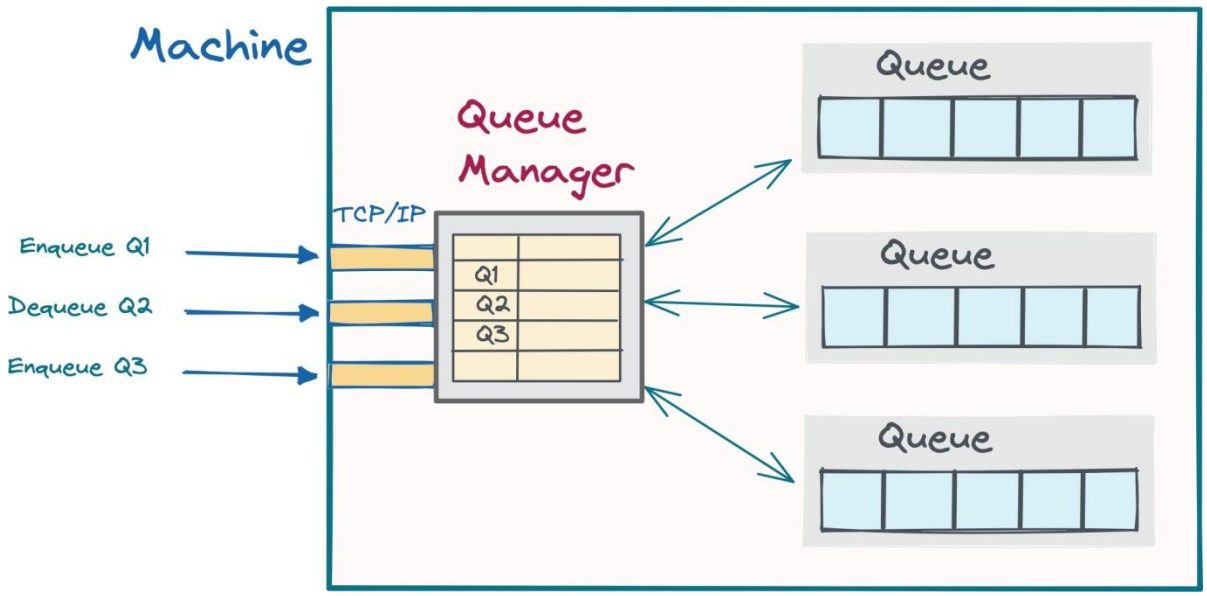


Here is another consideration. Theoretically, as long as we keep track of which items are deleted, we don't need to delete from the beginning of the Queue 1le frequently. If we can back up this buIer to separate 1les, so that we have a log of deletions, we can perform this deletion very less frequently. This would especially be useful if the 1le is large, so that we can avoid the large 1le from being re-written.

So now our persistent, high throughput queue looks like the following:



With Queue Manager and Multiple Queues:



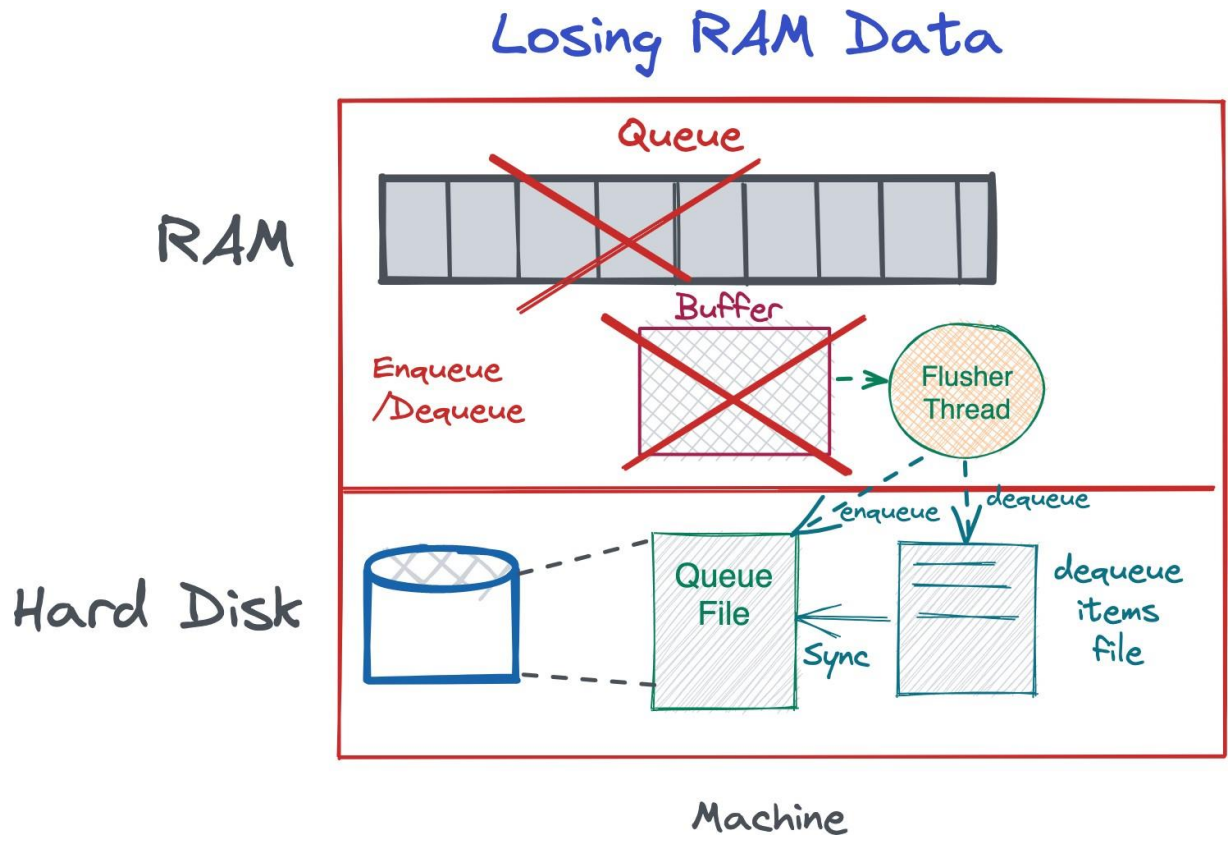
How can we make it fault tolerant?

Let's do a fault tolerance audit for this system. There's two scenarios we want to look at - the machine restarting and the machine going down, along with the hard disk.

Scenario 1: Machine restarts or process crashes

If the machine restarts and we lose the RAM Queue, we will still have the Queue persisted in the 1le.

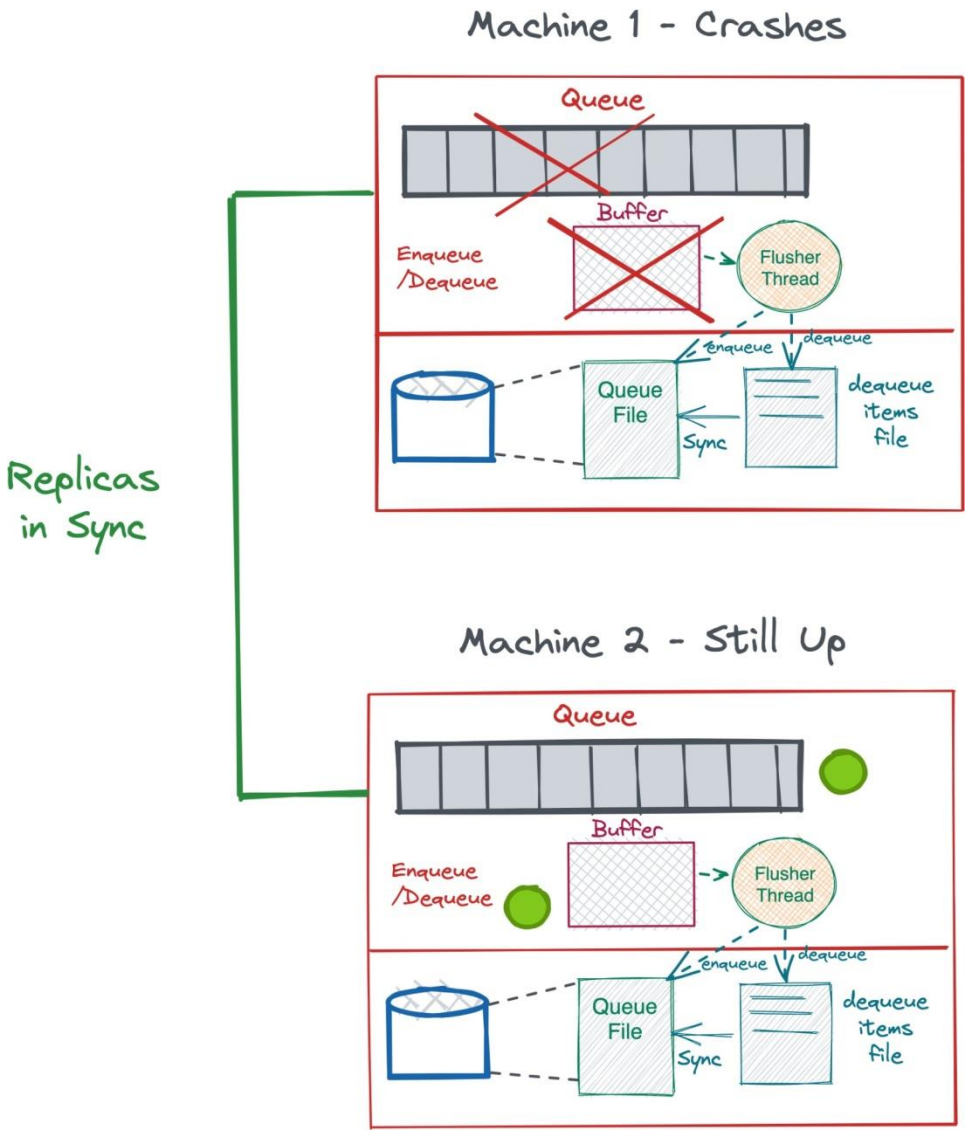
However, any unRushed items from the buIer might go missing. So, let's say we Rush the buIer every 50ms, then there is potential for 50ms worth of queue items going missing (in the worst case).



This may or may not be tolerable, but let's assume for now that this is not tolerable, since we want to design for high reliability. Since we want to maintain high throughput,

it doesn't make sense to Rush more often, and even with a smaller Rush interval, there will always be the possibility of losing that smaller interval, e.g, 5 ms. Writing to disk every time is the only safe way here, but that will be quite slow. So what do we do?

The answer for such scenarios is to add replication. If we replicate this queue into another machine, then that small loss will be very unlikely to happen, because the other machine's queue will still be up.



Scenario 2: Machine crashes along with the hard disk

This also solves our second problem - what if the hard disk fails. If we replicate to another machine, we will always have a backup queue saved in that machine's hard disk, so no problem if the machine fails.

Now you may ask, what if both machines fail at the exact same time? Well that scenario is highly unlikely, a lot more unlikely than a single machine failing. But if our data is extremely crucial that we want to safeguard even against that, then we can do a couple things:

1. Increase our replicas to three. Three replicas is the standard number of replicas cloud 1le systems use. Google File System uses that number by default. This reduces our likelihood even more.
2. Place the replicas in diIerent locations connected to diIerent power supplies. This ensures that if one power supply fails, they are not failing at the same time. Placing them at diIerent geographic locations is an excellent way to do this.

What if the Queue becomes too big for memory?

If the queue becomes too big - it's because the consumer is slow and the producer is producing a lot of data, the default way of managing this is to add another machine and partition the queue.

However, if we want to restrict to one machine, we can also move part of the queue into the disk to free up RAM space. This is very similar to what we do with caches - swapping out lesser used pages into disk.

We can swap out pages of the queue to disk and the producer can then continue writing to the queue.

Design a Distributed Messaging Queue - like RabbitMQ or Amazon SQS

This is a very popular question. Designing a distributed queue is actually quite simple once you understand the basics.

If you haven't already, please take a look at the non-distributed version of a high throughput queue. We can use that design and scale it up to implement a distributed version. This is also a good pattern to follow if you are confused about implementing a distributed system- 1gure out the single machine model, and then apply sharding and replication to distribute it across diIerent machines.

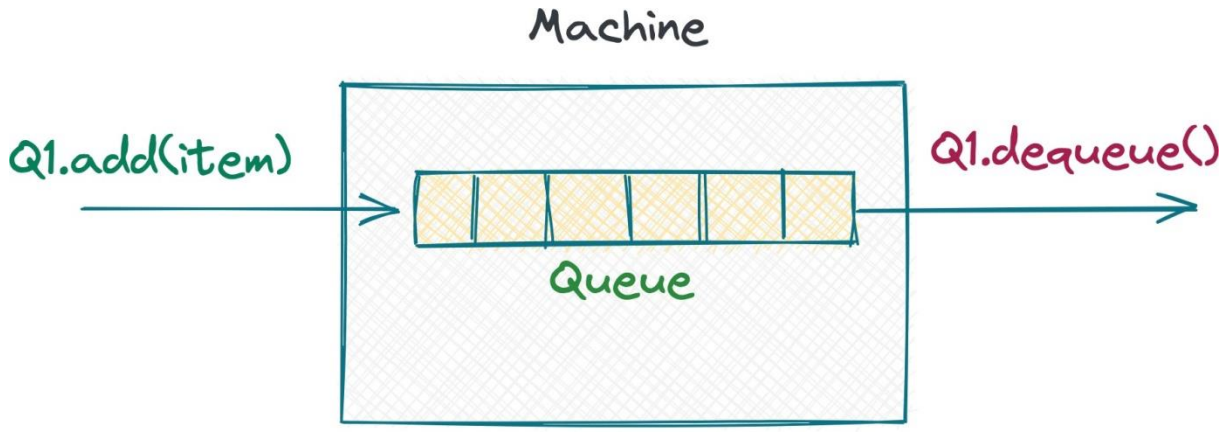
Let's take a look at the requirements of this queue:

Requirements:

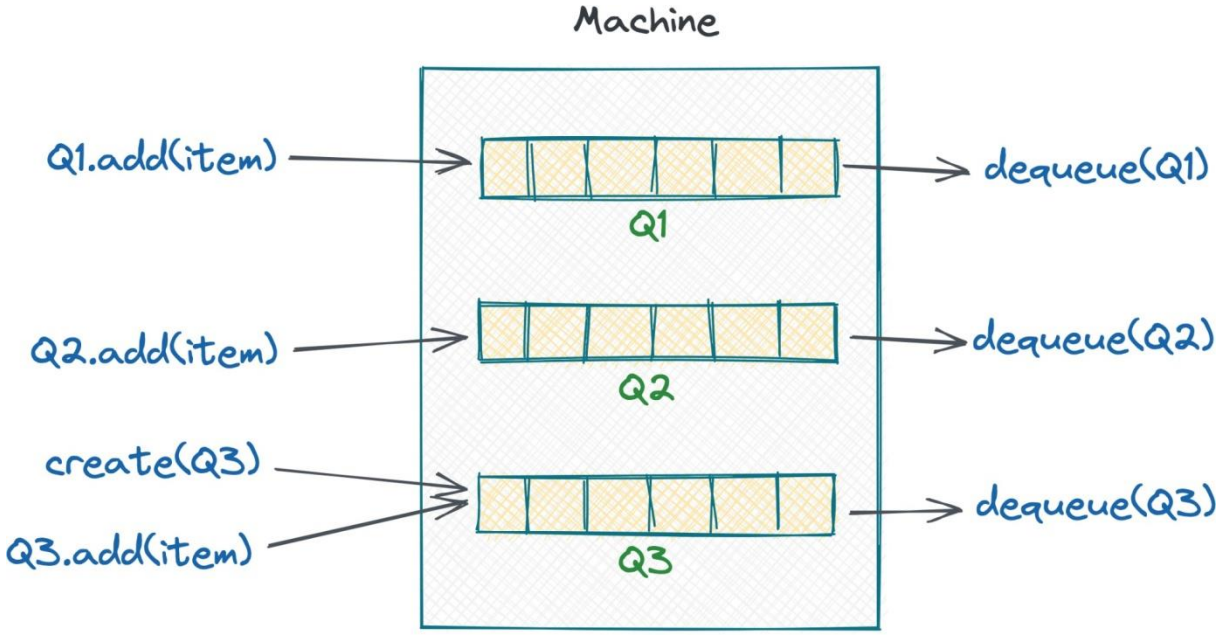
1. High Throughput
2. The queue should have persistence as much as we can without compromising on high throughput
3. Support for multiple consumers and producers
4. Once an item is taken from a queue, it can be deleted. Only one consumer can access one item

Let's see what we have so far from the single-machine implementation.

We can add and remove items from a created queue.



We can create multiple queues and add and remove items from each one of them by specifying the queue id or the queue name.



Using this as a building block, we can 1gure out how to do this on multiple machines.

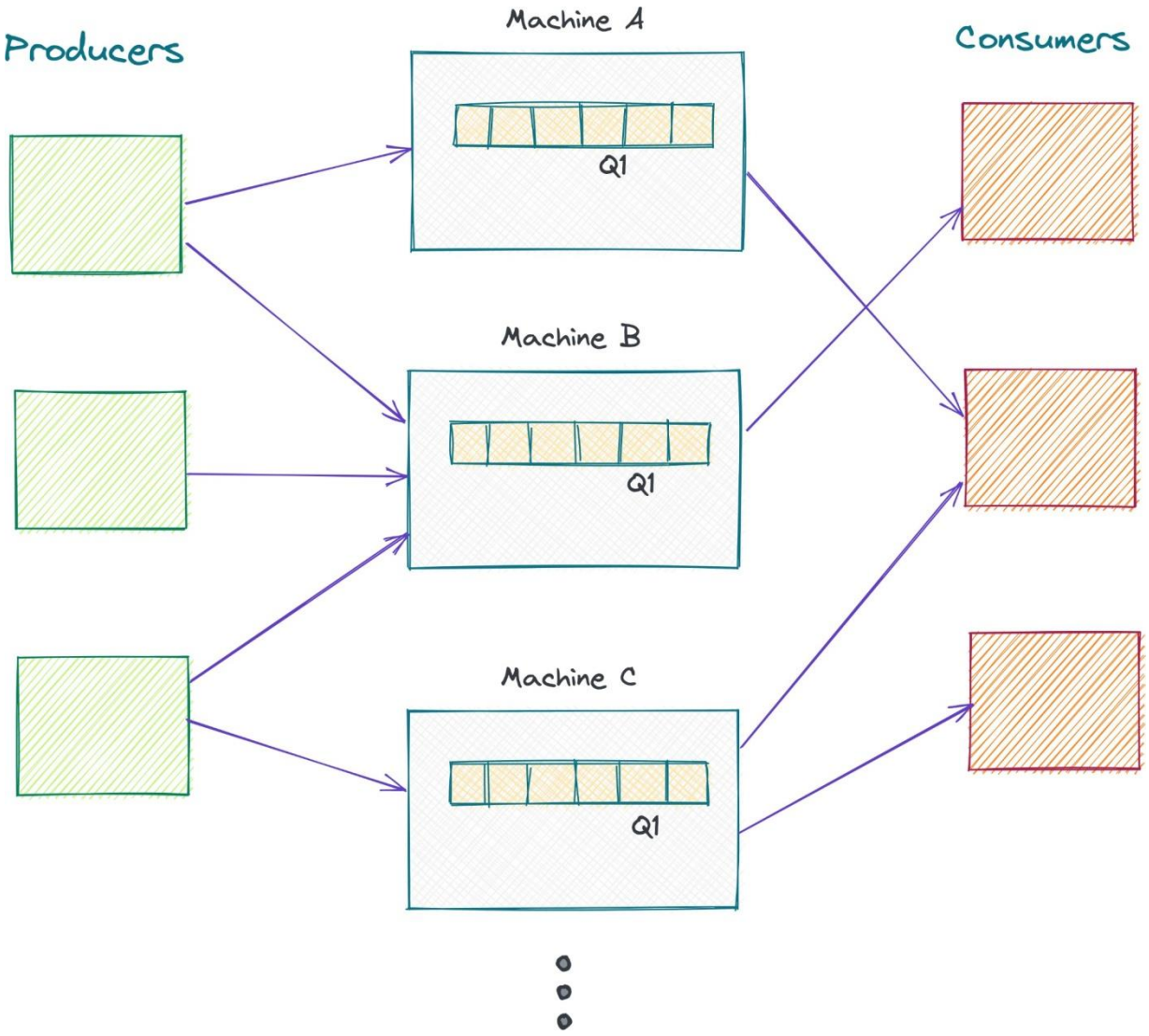
High Throughput Distributed Queue Scenario

Let's now say we have 3 machines and one queue to create and manage. We have a large volume of data, so we want this queue to be sharded. How can we manage this?

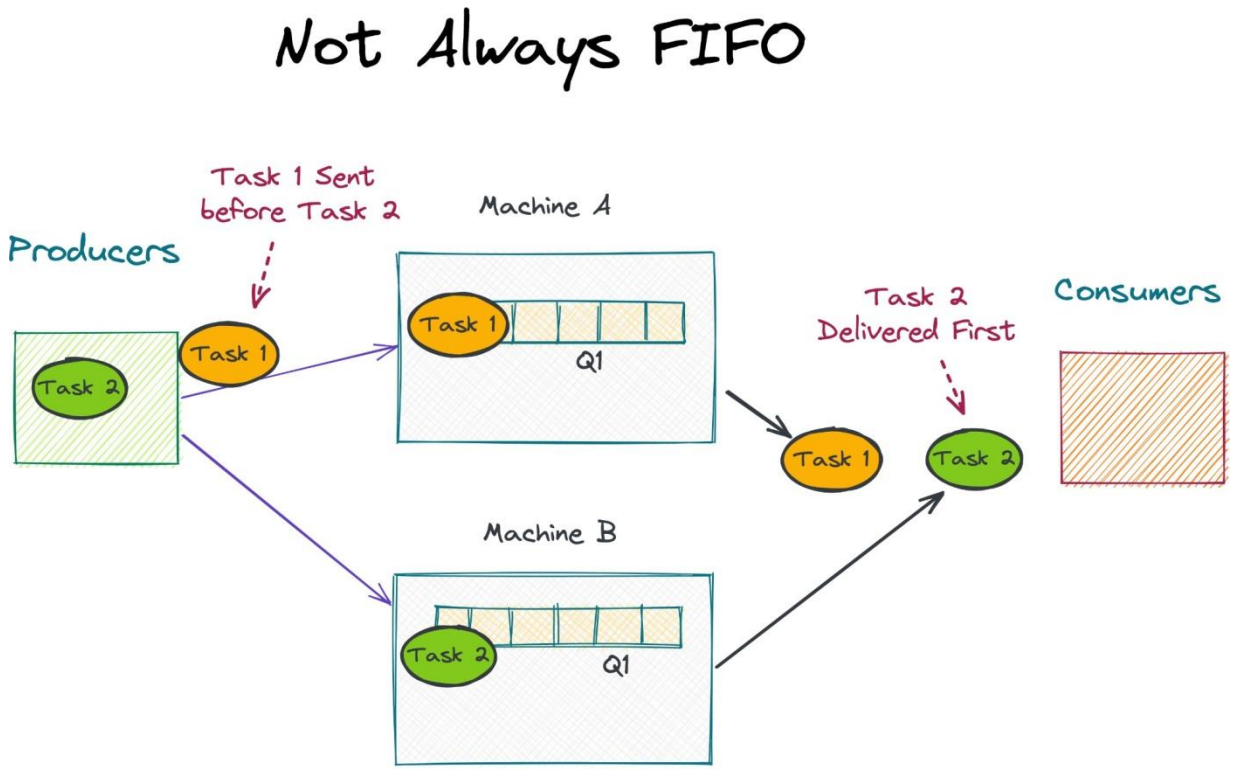
We have a lot of producers writing to this queue and a lot of consumers pulling from the queue. How do we handle the high throughput?

Well the logical thing to do will be to partition the queue into 3 separate queues, one on each machine. Each producer can write to any partition, and consumers can consume from any partition.

For huge workloads (think Facebook scale), you can partition this queue over 100 machines and achieve a lot of scale.



There is one problem though - this implementation will not be strictly FIFO for the entire queue. It will only be FIFO within each partition. For most use cases, this is ok. For example, if we're using it as a task queue or as a noti1cations queue, it is ok if two tasks are out-of-order within a small time frame.

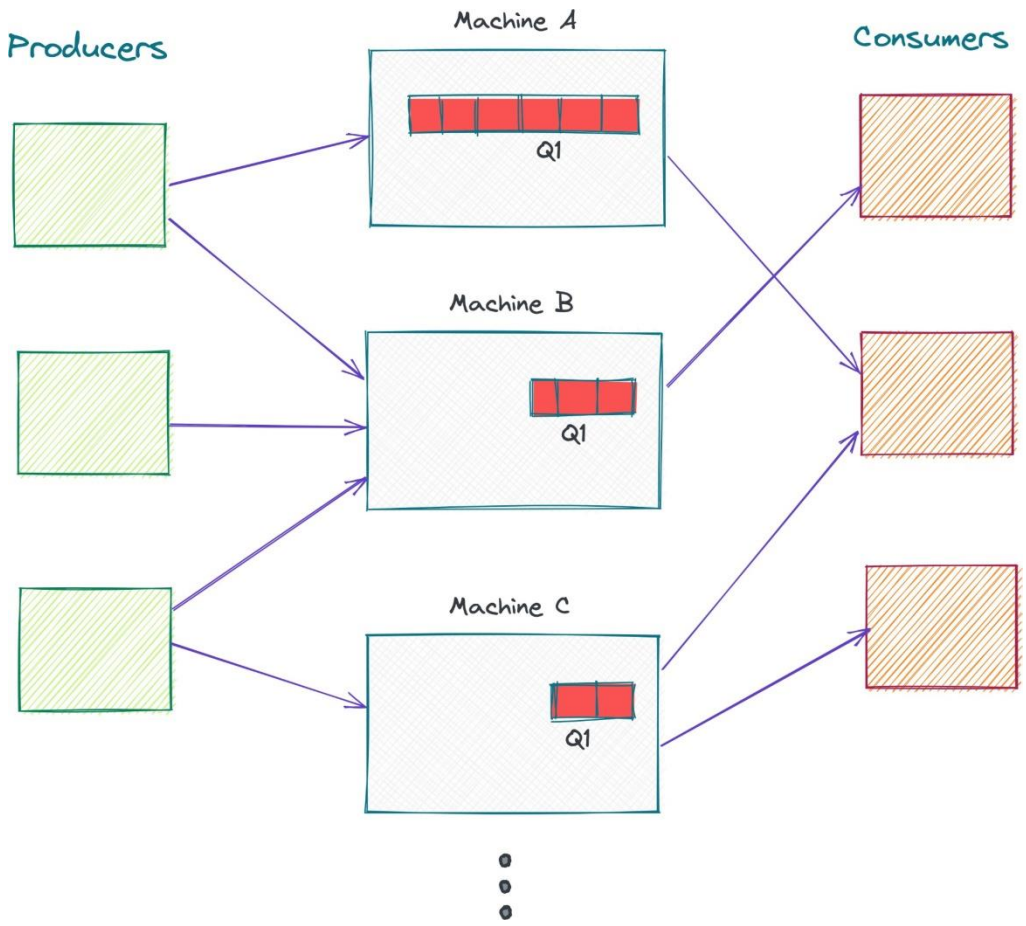


We should try to keep all partitions evenly occupied. That will be ideal for the queue's performance because we don't want to overload one partition.

With X machines, our throughput should be X times the throughput of a single machine.

How to keep the Distributed Queue evenly balanced?

Keeping the queue balanced means writing and reading evenly from all the machines. If one machine gets starved of items, then it's readers will be starved as well.



In the Queue above, readers of Machine B and C will soon get starved because Machine A is getting a higher rate of writes.

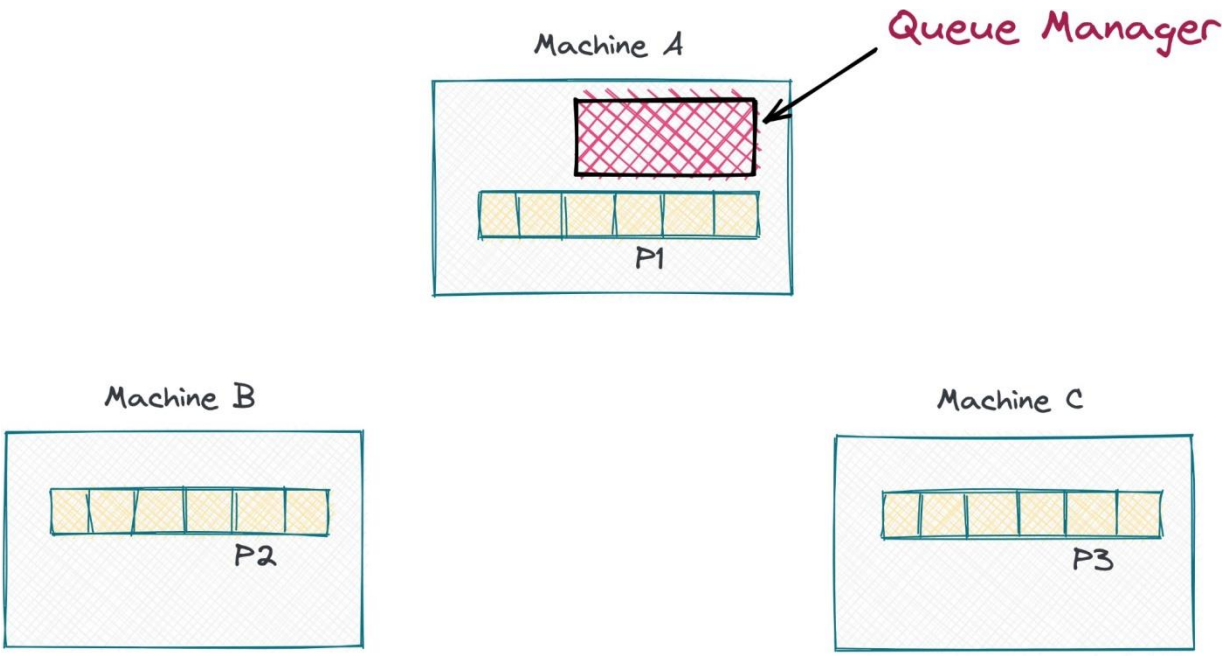
The challenge really becomes writing and reading evenly from all partitions. So we need some sort of a balancing approach, or some sort of **coordinator**.

Coordinating between the Queue partitions

Let's see how to coordinate a queue between 3 machines.

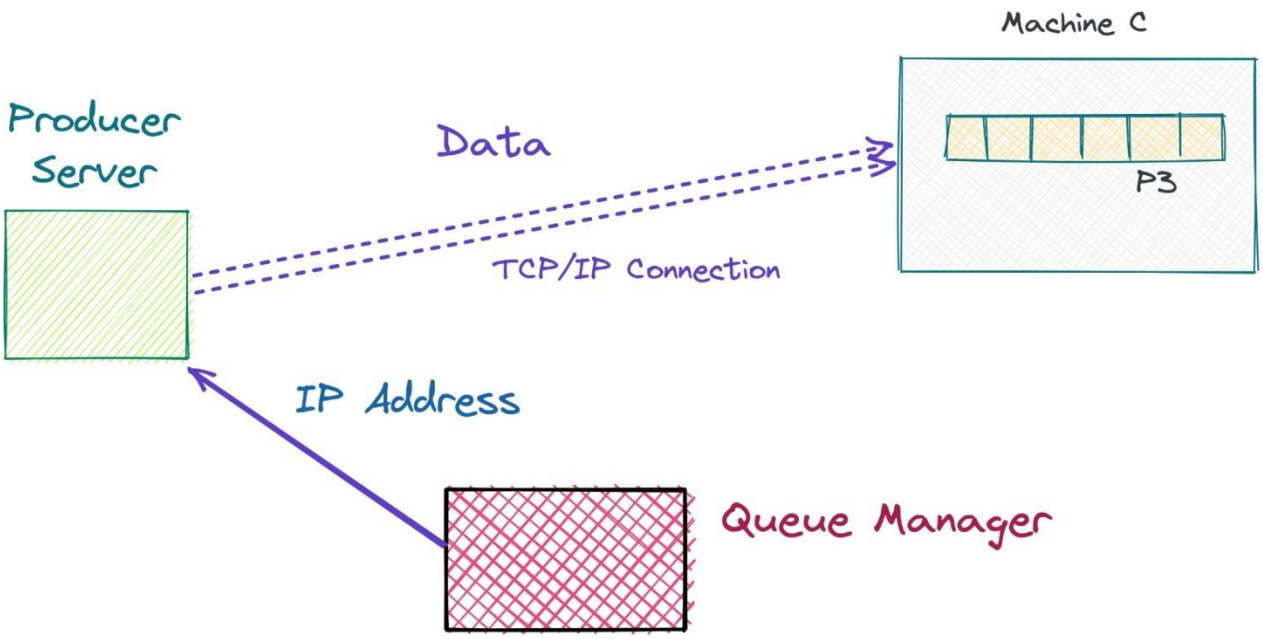
Let's say we are creating a Queue Q1. We create 3 partitions - P1, P2 and P3 - each on a diIerent machine. We will need some sort of **Queue Manager** that keeps track of the partitions.

Let's say that we have such a program running on one of the three machines:

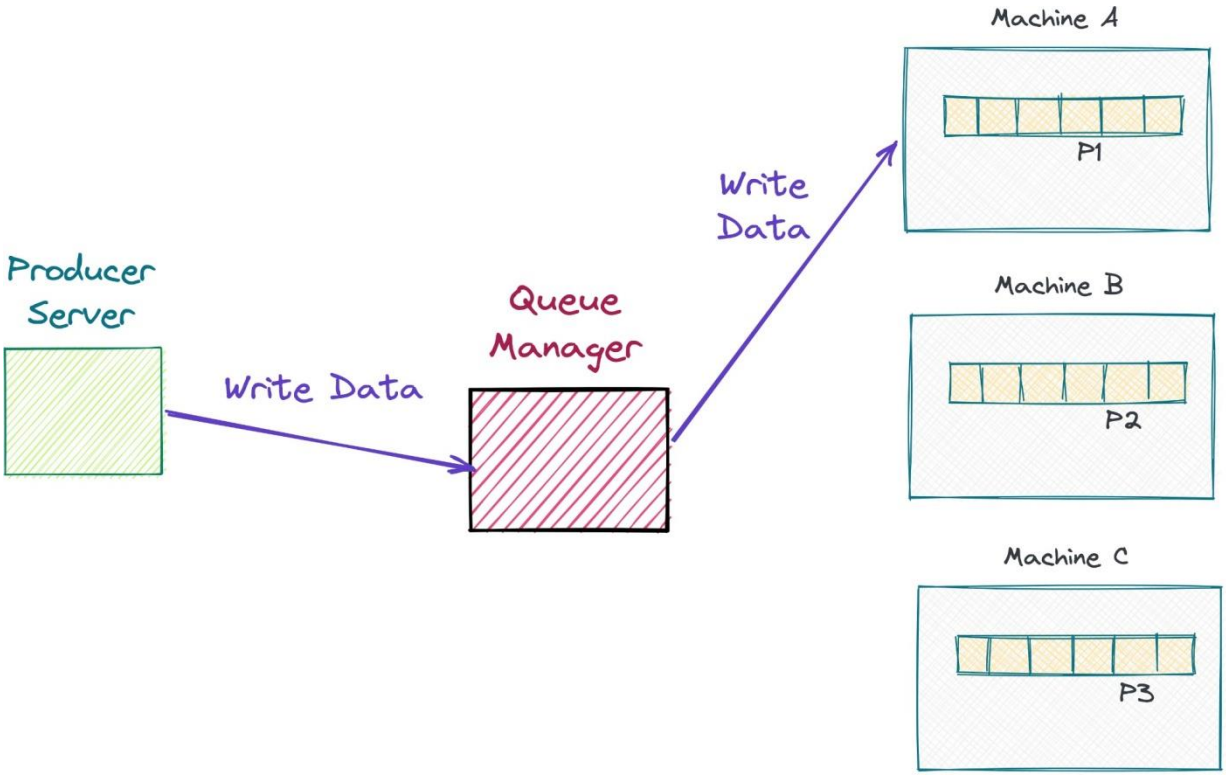


Try to 1gure this out yourself before reading our solution.

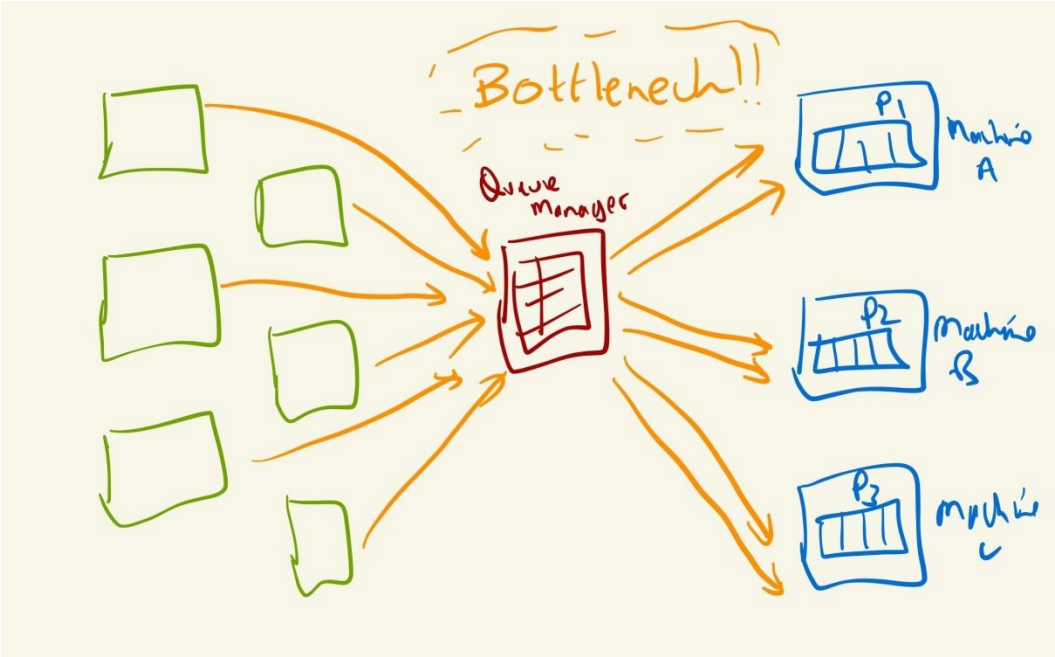
When a producer needs to write to the Queue, it asks the Queue Manager for the IP address of a machine it can write to. Let's say the Queue Manager returns the IP address of Machine C (where P3 is located). The Producer can now establish a persistent TCP/IP connection and start writing to P3.



Now you may ask, why is the producer connecting directly with the Partition Machine? Why doesn't it just pass along the message to the Queue Manager, and the Queue Manager can send it to a partition, like this:



This will make things simple - just hand it oI to the Queue Manager and the Queue manager can take care of the rest. The problem with this approach is that the Queue Manager becomes a bottleneck. All of the data has to go through it, and there's only so much throughput this one machine can handle.

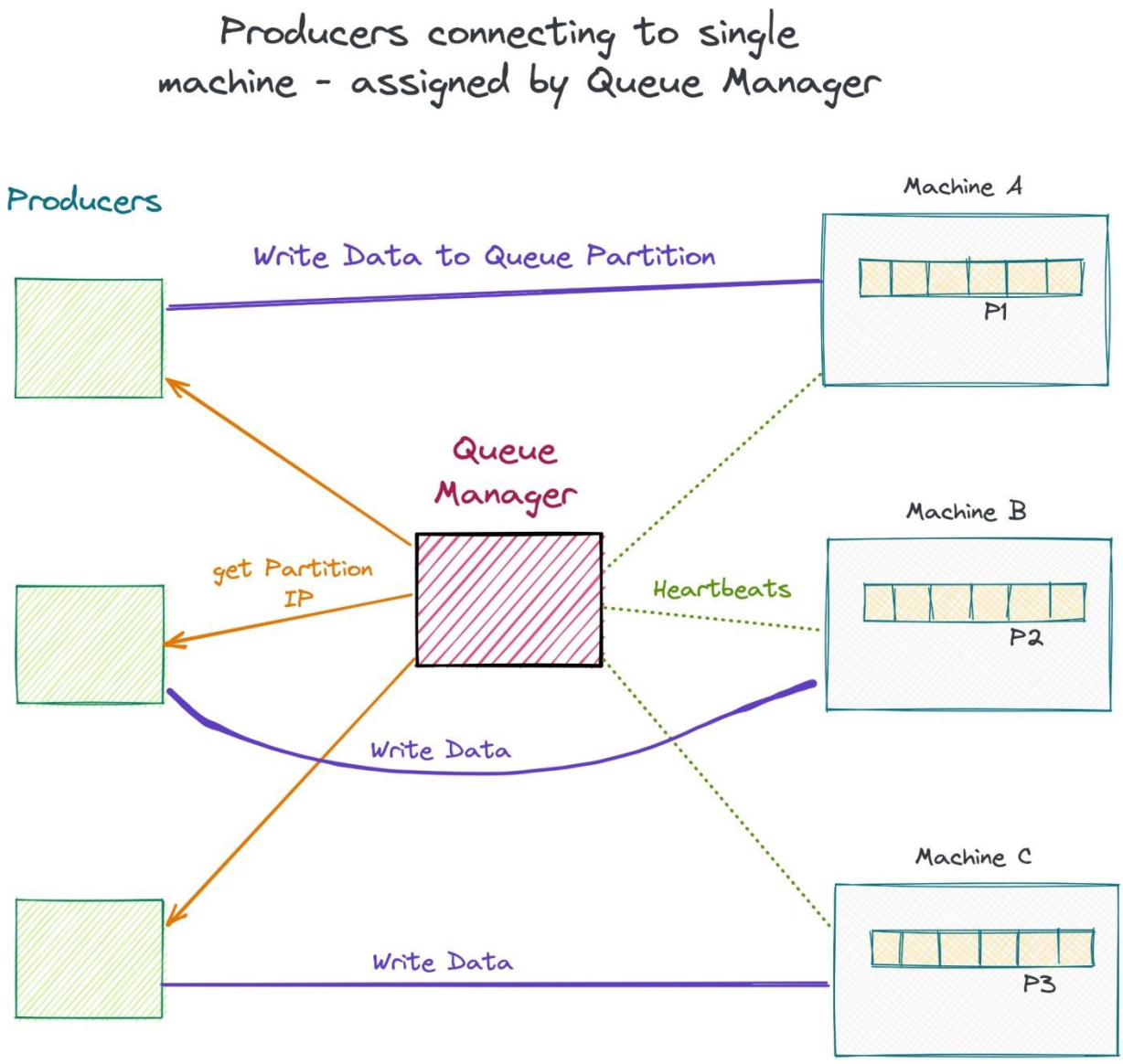


As the number of producers increases, this might slow down the entire queue. Then you would have to add more queue managers to handle more writes. Instead of all this, the general pattern is to directly let the producers connect to the machine. This way the writes are decentralized, and that makes it more scalable horizontally. This is also the pattern used in most distributed 1le systems for writing data. For example, Google File System uses this exact pattern.

So now we've established that the producers will directly connect to the partition machine and write the data.

So far we had one producer. Now, let's say one more producer wants to write to the Queue. The Queue Manager can send this producer to a diIerent Machine so that the previous partition is not hogged and other partitions are also written to.

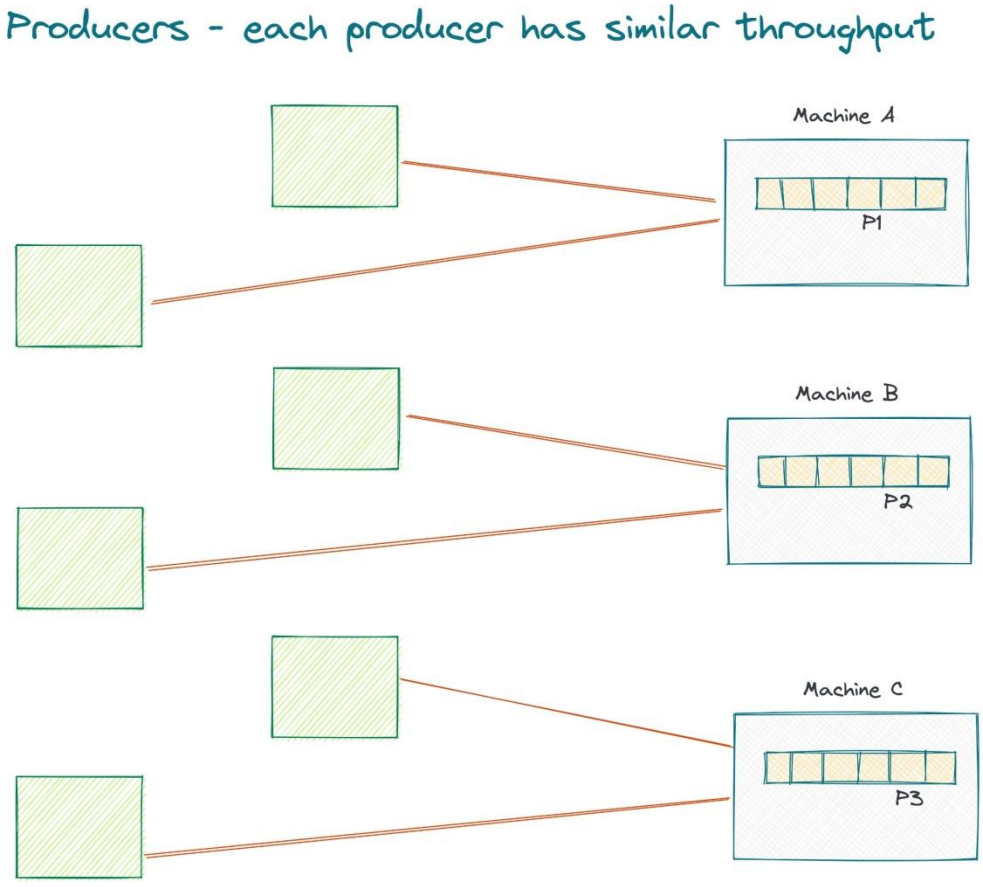
This way, the Queue Manager keeps sending new producers to diIerent partitions, distributing the throughput across partitions.



As you can see, we also have **heartbeat** messages going from the Queue Manager to each partition, regularly communicating the partition's health with the Queue Manager. If the partition goes down or becomes overcrowded in relation to the other partitions, the Queue manager can assign less producers to the partition.

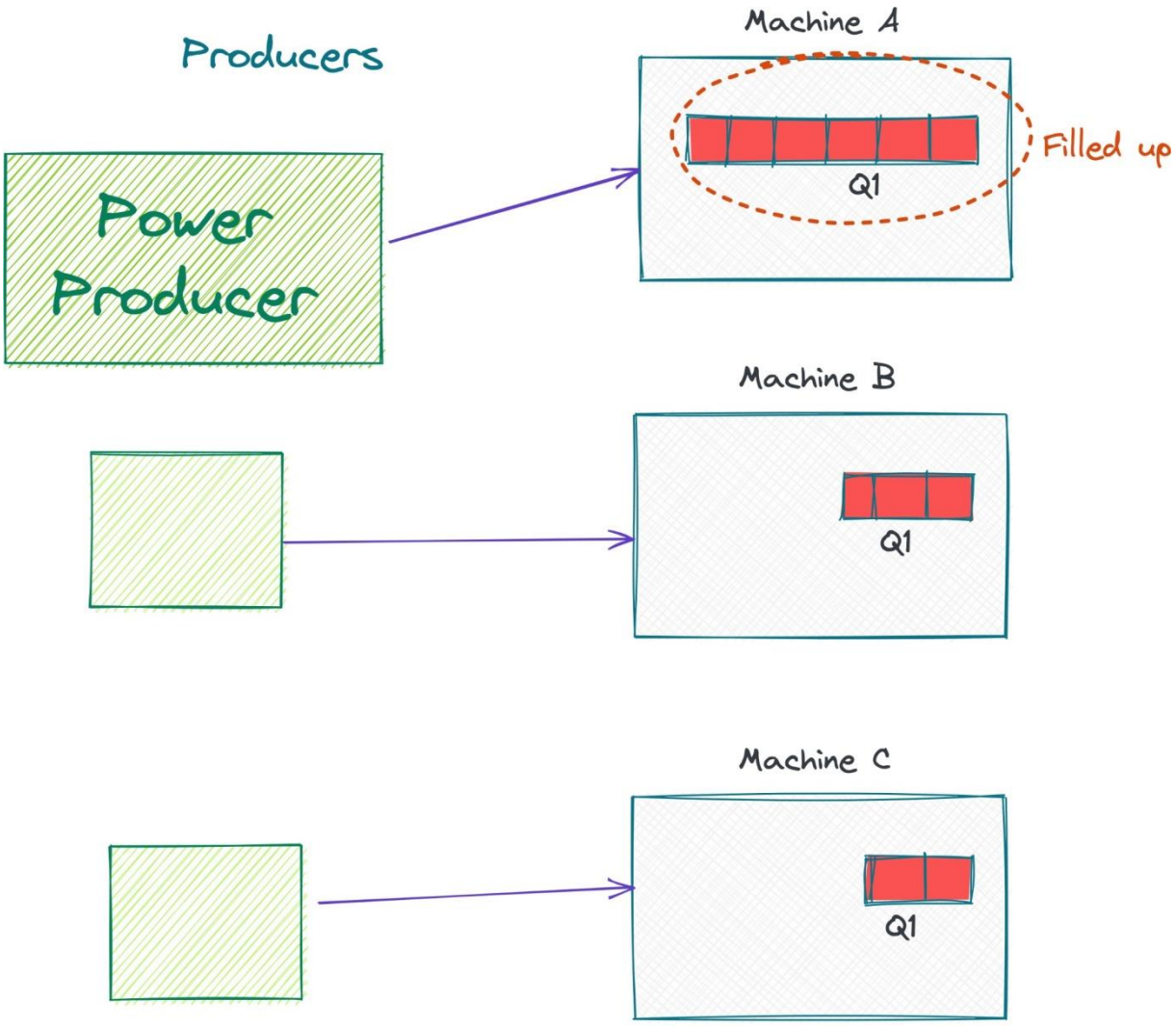
This above setup has a Raw though - can you spot it? Spend some time thinking about it.

Here it is: This model of assigning a single partition per producer works well if the producers are homogeneous and more in number than the queue partitions. That way, each partition will get a similar amount load, and we don't need to do any load balancing.



For load that doesn't have a lot of bursts, and which has a lot of producer servers, this will work well. If there is even one "power producer" that is producing a lot of data at a high rate, one partition will be hogged up.

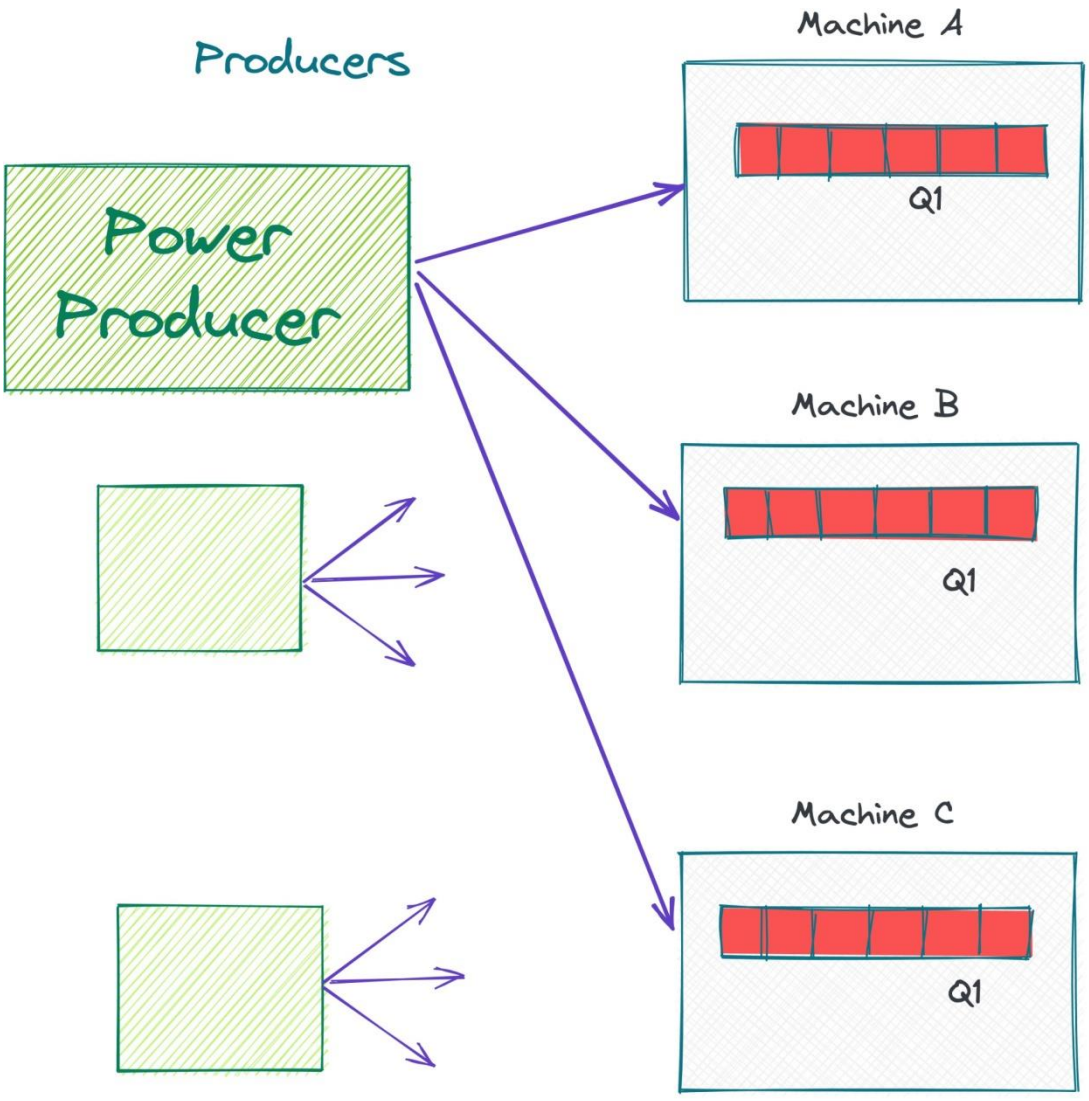
This model breaks down as soon if we have diIerent producers producing diIerent quantities of data.



As we can see above, the power producer 1lls up one partition quicker. When consumers read the queue randomly, the 1lled up partition's items are at a disadvantage,

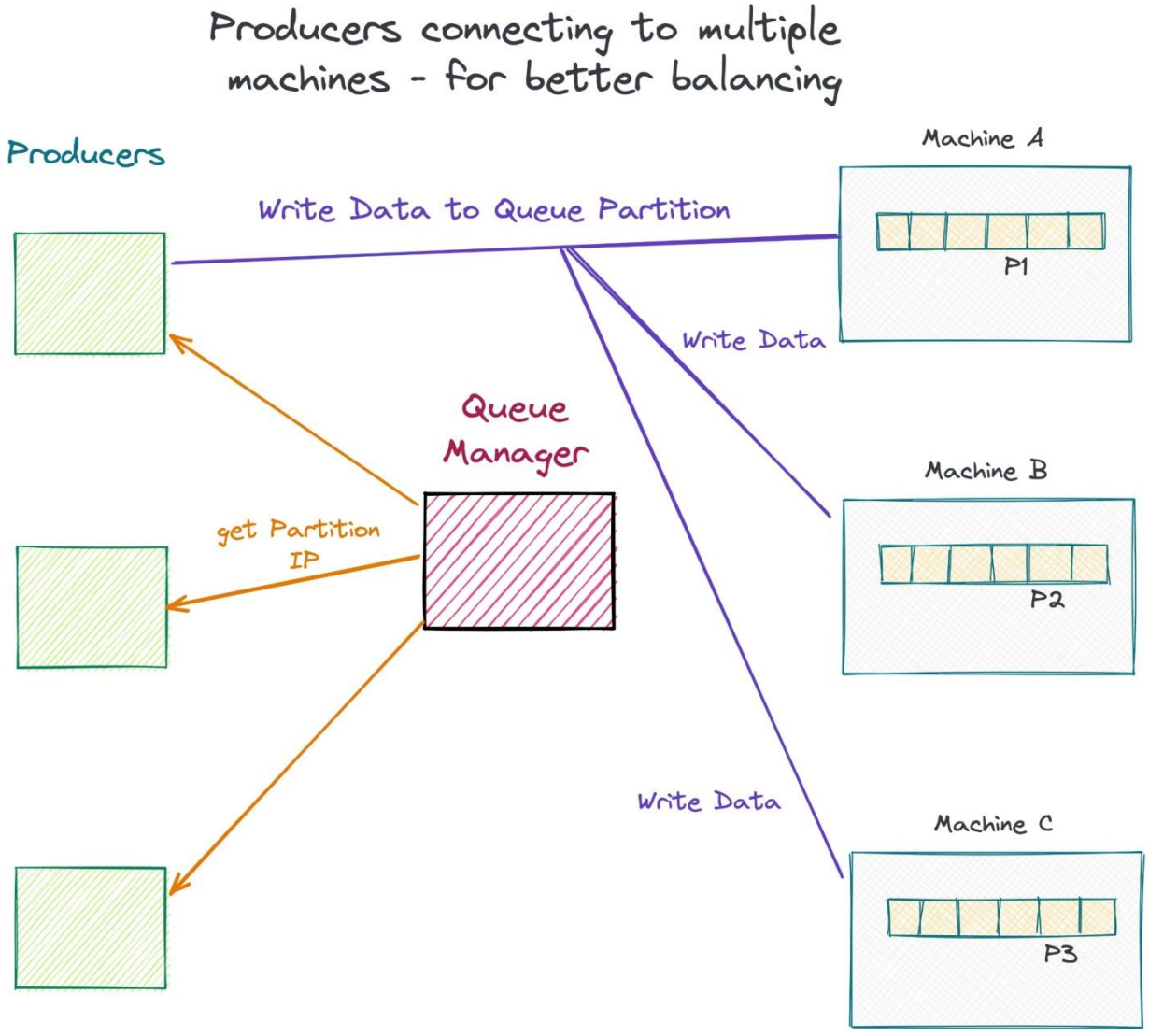
because they will be read much later - after the items in other slower partitions have been read. This reduces the FIFO properties of the queue.

What can we do to solve this? Well, one solution is for each producer to write to multiple partitions. Each producer can do a round-robin write to diIerent partitions.



When the producer connects, the Queue manager can give it the IPs of all three partitions. The producer can then connect to all three machines and write to all 3

machines in a round robin fashion - once to Machine A, then to Machine B, then to C, and so on.



Now, you might ask, if I am writing code on the producer, do I need to write a loop to pick one partition and write, then pick another partition, etc.? No, that will be handled by the Queue's client library.

For example, RabbitMQ has client libraries that the producer machine will install. Let's say the producer is a Web server and the web server needs to write lots of JSON objects

to the queue. The web server will install your Queue's (MyQueue) client library. This library will have functions to connect to the queue and write to it. For example:

```
Connection queueConnect = MyQueueManager.get("My JSON Queue");
queueConnect.enqueue("{id:33435, data: {...}}");
```

The enqueue function and the MyQueueManager library should handle round robin writes to diIerent partitions. All of this is abstracted away from the end user aka the producer.

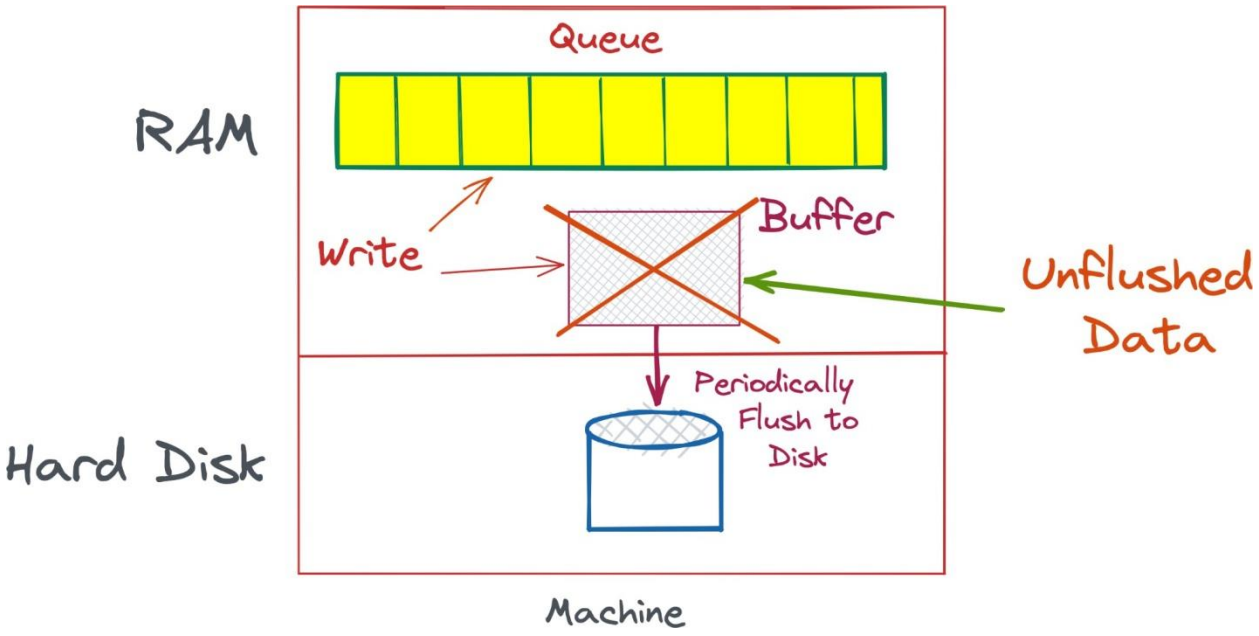
So now we have seen two approaches for writing to the Queue - assign one partition to a producer and assign multiple partitions to the producer. In reality, diIerent situations might deem diIerent approaches to be more suitable.

If we have 1000 producers and only 3 machines, then it might make sense to assign one partition per producer, because evenly spreading so many producers might result in good load balancing anyway.

In our library, we can give a con1gurable property for this. The developer can con1gure it according to their situation and customize the load balancing.

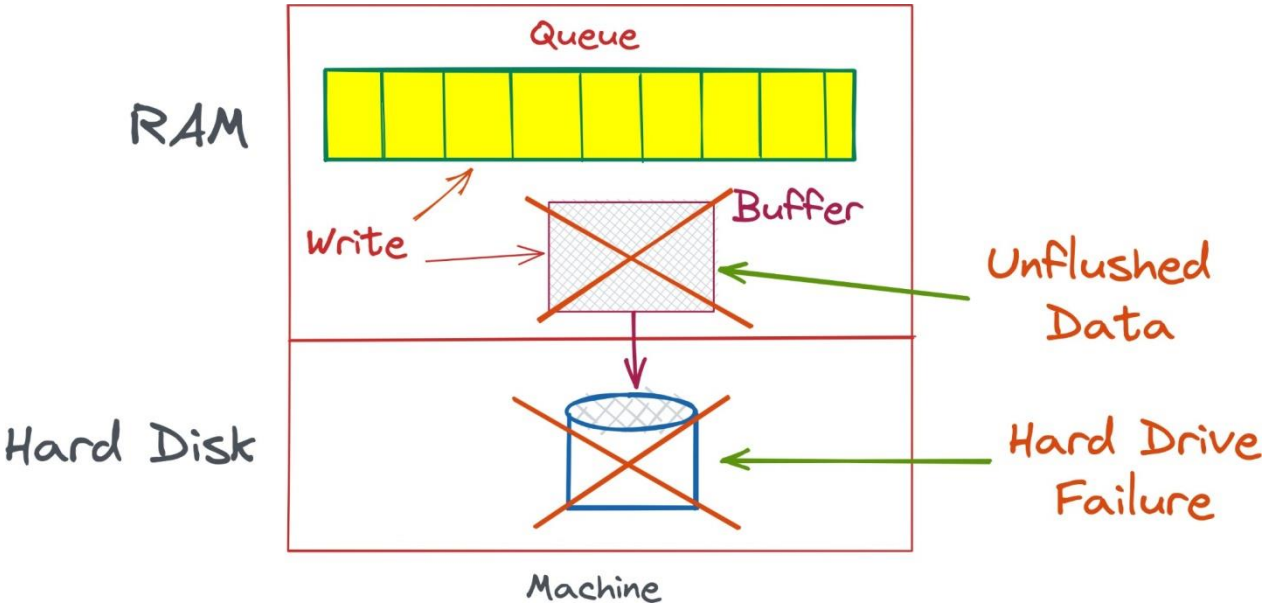
How do we make sure the system is fault tolerant?

From the discussion in the non-distributed version, we know that till the data Rushes to disk, it is susceptible to being lost if the process crashes or if the machine goes down.

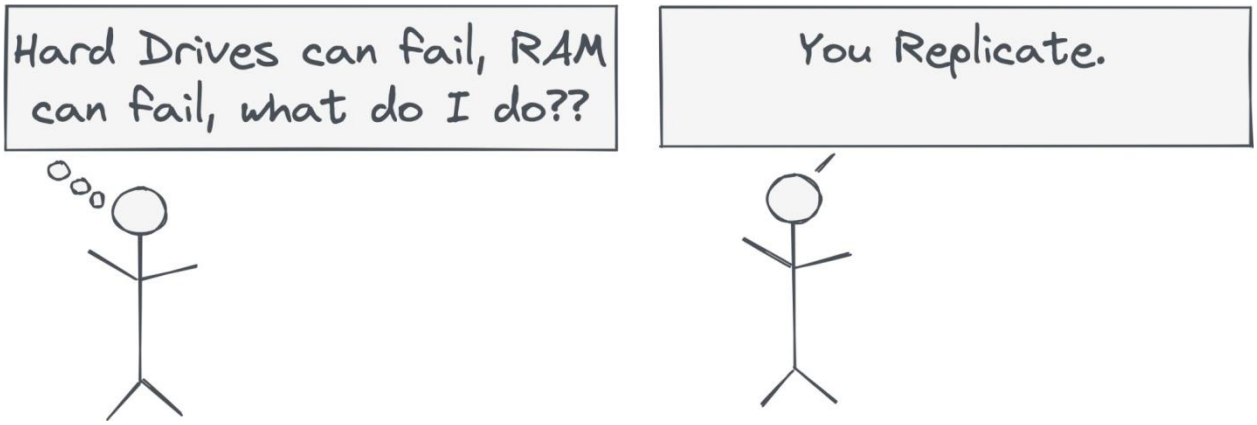


This is usually small - the size of the Rush interval - usually less than a second. However, this is still a major cause of concern because in a messaging queue, we want zero data loss. If you are sending a task to be queued and the messaging queue loses that task, we have suddenly lost the task. This is not acceptable in most use cases. Imagine you sending a message to a friend, and that message just disappearing into thin air without you ever knowing or being informed. That is what will happen if our queue loses data.

Now, if a hard drive fails, we have suddenly lost all contents in a queue.

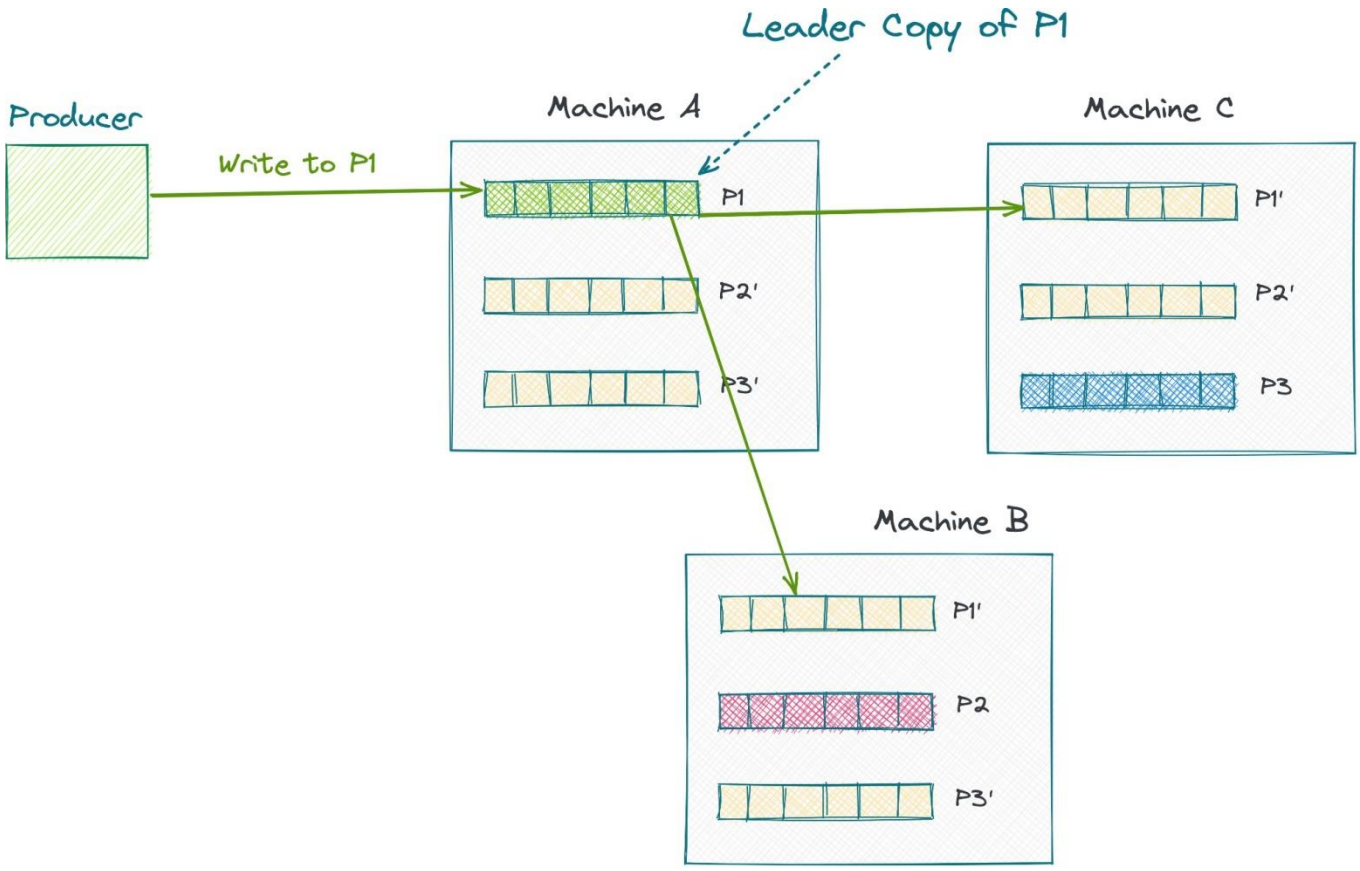


This is not desirable at all. If the drive fails, all the data is not recoverable. Assuming the process also crashes when the drive fails, the entire queue is lost, including the data in the RAM. How do we save ourselves from this horrible fate?



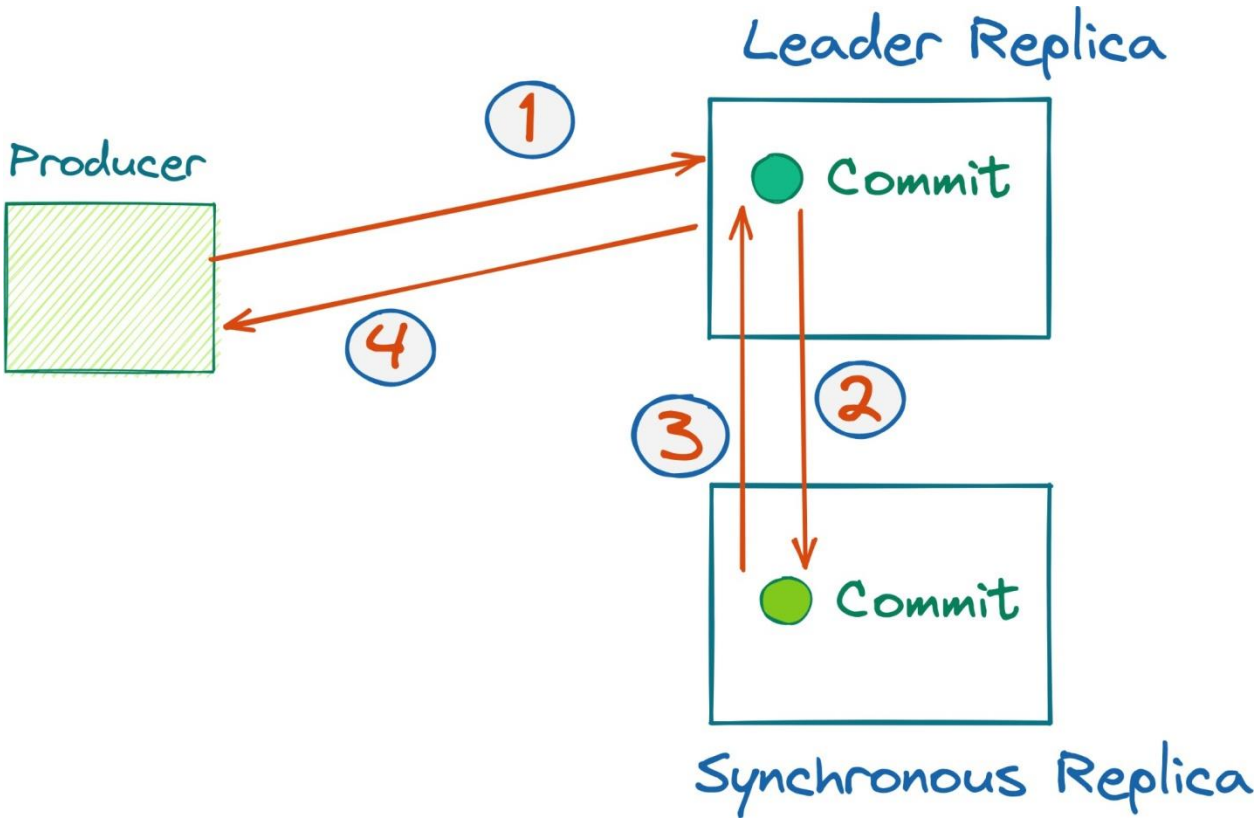
We turn to a time tested method of replication. That is the only way to safeguard against total machine failure.

This means that we need to replicate each partition into multiple machines. Here is what that would look like:



Each partition has a leader machine. All reads and writes to that partition go to that leader machine . Note that in a Queue, a “Read” is also eIectively a write because the item has to be dequeued. So we cannot take advantage of replication to increase read throughput as we do in replicated databases - where read replicas are able to increase our read throughput. This might be diIerent in a streaming platform like Kafka where a read does not delete the queue.

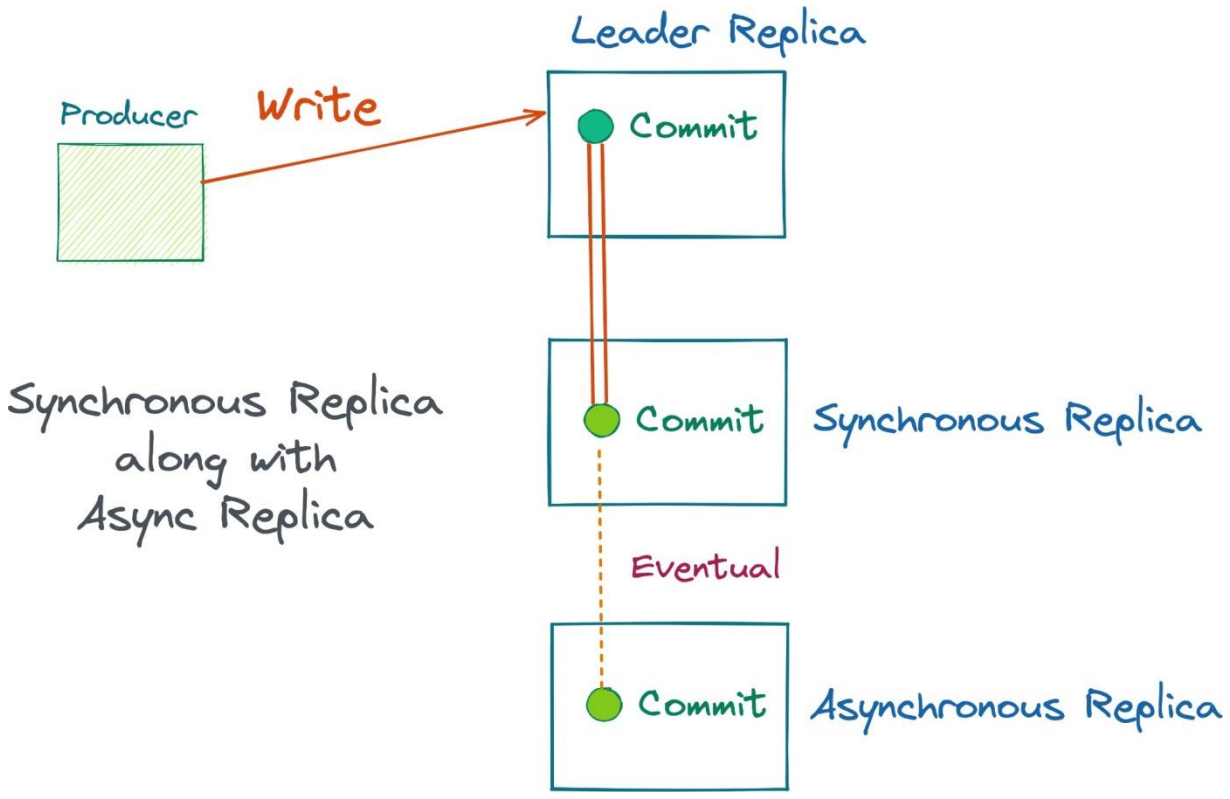
Let’s look at how exactly the replication happens. As we can see, there is a leader machine for each partition. The producer will connect with the leader machine and send data. The leader will write data to it’s partition and to all it’s **synchronous** replicas. Synchronous replicas are those who are updated synchronously with the leader replica. Let’s say we have 1 synchronous replica. This ensures that the write is written to 2 machines right away.



Before we send an acknowledgement to the producer that the write is done, it is written in 2 machines. This might take longer than writing to one machine, but it ensures that data loss will be very hard.

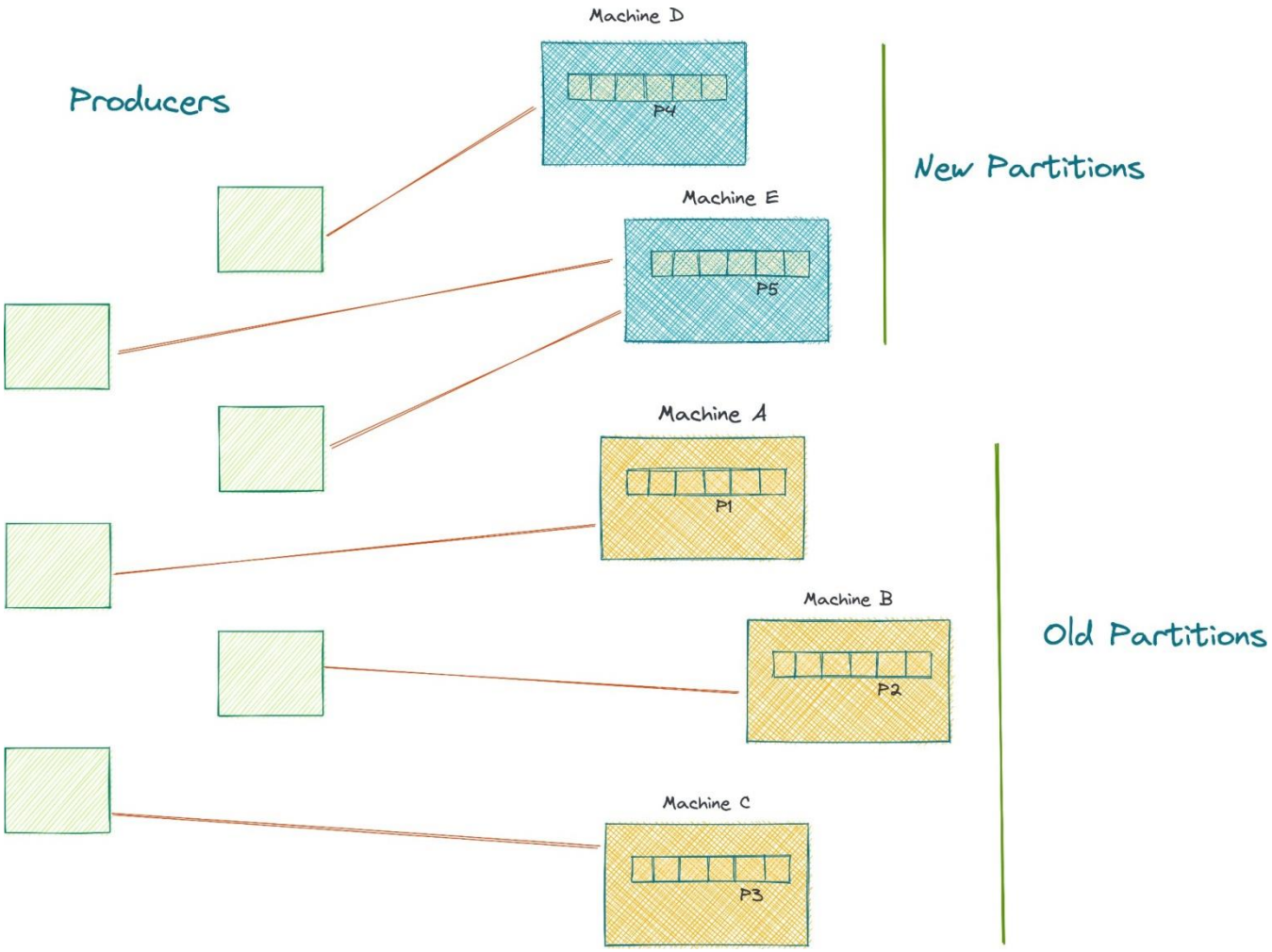
In order to lose data, now 2 machines have to go down at the same time, which is much less likely. To make this even less likely, the administrator can also do things like ensuring the two machines are connected to diIerent power supplies or diIerent network routers - so that they don't have a common cause of failure.

We can also have normal replicas or **asynchronous** replicas, which are synchronized after the main replicas returns an ack to the producer. These replicas are faster for the throughput, because the write is propagated asynchronously. This is what the replication looks like now after adding the asynchronous replicas.



As we can see, we have a replication factor of 3 here - 2 synchronous and 1 asynchronous replica. You can adjust this as per the data needs. For example, if the data is not too critical, like user behavior logs, it's probably ok to lose small amounts of data once in a while, so you can do 1 synchronous replica and 1 async replica. It's up to the developer to adjust this according to business needs and costs.

After adding fault tolerance, this queueing system can now scale to many machines. We can auto scale the queue and add more partitions as load increases. To add another partition to a new machine, we simply create a new partition and point producers to it.



Design a Publish/Subscribe System - like Kafka

In Progress

Design API for Amazon.com

Design a Logging System for Facebook

Design Twitter's Analytics System

Design Capacity Planning for Reddit.com

Design Top K Apps/Amazon Best Sellers

"Which DB will you use? NoSQL or MySQL?"

Design Uber Surge Pricing - a Stream Processing System

"Do you need to add a Queue to your backend?"

..More questions to come..

Interview Concepts

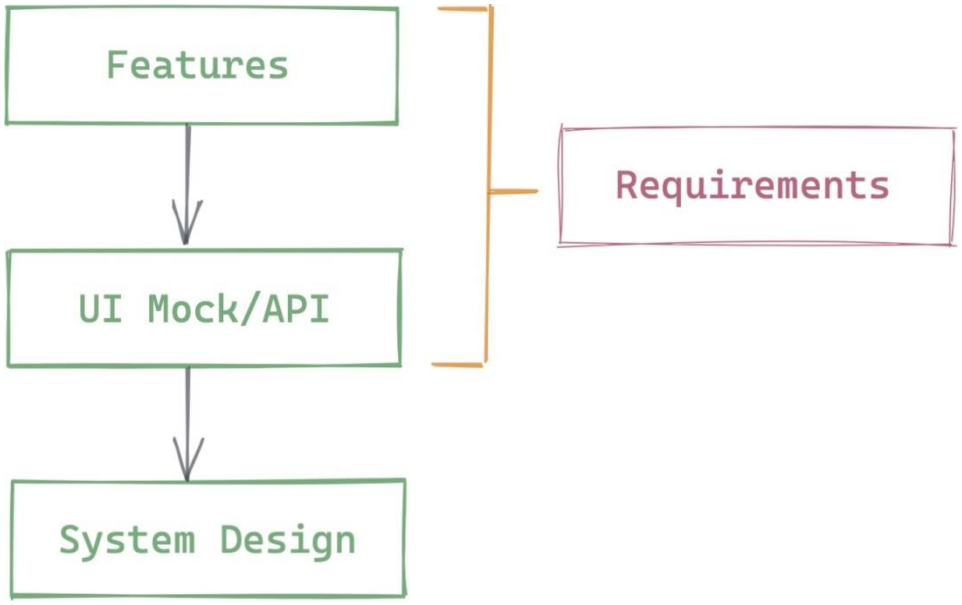
Discussions vs Exams

With algorithms, if you've seen a question before, you can easily ace the interview - it's an exam. If you do the same with SD, it won't work.

For example, let's say you know the high-level design for Facebook's backend. You present it. That is not enough to impress the interviewer . They know you're just repeating what you saw on a popular site. They will probe you further - until they 1nd something you're struggling with. Then they will cross question you. That's when they see your skills. It's a discussion, not an exam.

Waterfall Approach

As much as possible, I recommend following a waterfall approach. What does that mean? The waterfall approach means that one step leads to the other.



First, you decide what features of this system are. You try to narrow down features as much as possible - there's only so many features you can implement and go in depth. Those features directly lead to a UI mock, or an API. And that API or UI mock leads to the system we need to design.

1. Collaborate with the interviewer. Deploy the Suggestive Approach.
2. The interviewer can interrupt at any point and lead you in a diIerent direction. For example, let's say you're done de1ning features, and you start to de1ne an API. Maybe this interviewer doesn't want an API. They tell you to jump to the design. That's ok, you have to be adaptable. Remember, System Design Interviews are a Tree, not a Line. The interviewer can interrupt at any point and take you in another direction.
3. Before you start with the next step, Con1rm it with the interviewer. This ensures that you both are on the same page. It also gives the interviewer an opportunity to adjust the course if they want to.

We are still in the process of writing this book. Please consider subscribing for new updates □ : https://systemdesign.org/subscribe