Using S3 Intermediate Storage for Efficient Horizontal Scalability

Author: Peter Gaivoronski

Claps: 7

Date: Aug 25, 2021

When we create web applications, scalability is usually at the top of our concerns list. Having a scalable application allows us to easily handle traffic spikes when users flood our resources with requests. We can avoid costly outages and provide a good experience for our users despite the growing popularity of our services. Yet what do we really mean by scalability? When most people talk about scalability, what they are referring to is scaling potential. The scaling potential of an application can be described as the number of degrees of freedom along which it can potentially increase its processing capacity. If we have a high scaling potential, that means we can increase our processing capacity along many degrees of freedom, where each componentâ€™s processing capacity can be increased independently of other components.

Scalability can be defined as being either vertical or horizontal in nature. Vertical scalability has to do with increasing the processing power of a set of centralized nodes so that they can do more work with the additional resources. Horizontal scalability has to do with increasing the number of nodes to more broadly distribute work across a decentralized network, while keeping node size fairly constant.

Which should we favor? Vertical scalability, while appealing and simple to do in principle, has serious limitations to its ability to grow past a certain point. A useful metaphor for a vertically scalable process is a skyscraper, shown in figure 1. A skyscraper can fit many people in a small area and is therefore a highly efficient building in a square footage sense. However, it is not efficient in terms of construction costs or maintenance. It also cannot be taller than a certain height due to technology limitations. We also cannot add floors to an existing skyscraper without creating major disruptions and incurring extreme expenses. Horizontal scalability, in its perfect form, is only limited by the number of resources that can be devoted to processing the task, which in modern cloud-based environments can be a very substantial number. If vertical scalability is building skyscrapers, then horizontal scalability is building small towns. A 200 story building, if it could be built, would only be able to contain around 12,500 people. A typical small town can fit up to 25,000 people, about twice as many as our biggest possible skyscraper, according to the official definition of a small town. Each building in our small town is cheap to build and maintain separately when compared to our skyscraper, and the town can easily grow further into a large town to accommodate more people if there is such a need, while the skyscraper cannot. Multiple skyscrapers can be constructed if necessary, but not only is this a form of horizontal scalability, it is also more expensive to add a skyscraper than to add the same amount of capacity in small buildings to our small town, as long as we have the appropriate land available.

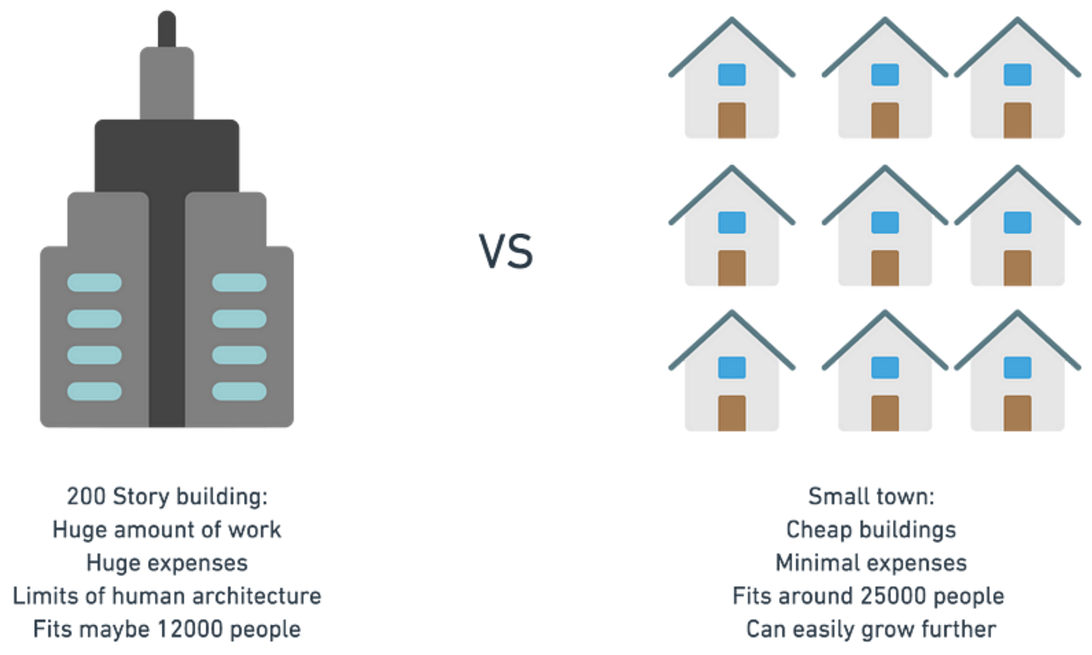


Figure 1: Real life scalability concerns

Horizontal scalability, therefore, is the preferred option for almost any scenario where scale is important. We can achieve horizontal scalability by decoupling each step in our system in such a way that steps can be parallelized independently of each other. The more logically distinct operations we can separate, the more potential for scalability exists in our code. There are many ways to accomplish this goal. We can logically split our code and databases into controlled â€œregionsâ€� such as â€œaccounts codeâ€� and â€œemail codeâ€� that can be owned by different teams and can run on different hardware. We can shard our data and workloads such that we are using a mathematical function to distribute our data and computations into a fixed number of distinct buckets. These kinds of divisions are hard to make because they require human thought and ingenuity, but once made permanent they provide a basis for separating code to have a higher potential for scalability through distributed processing and distributed ownership.

What if we could separate things in a more systematic way? It turns out that we can create a general framework for splitting any process into many subprocesses that are functionally independent from each other, creating more scalability potential at every split. We can analyze any code in a semi-autonomous way which does not require much human ingenuity and split it into components that can be run and scaled independently. We can look at code as a series of data operations, as in figure 2. Each step of the code starts with some data input, performs some sort of operation on that data, then outputs the result somewhere else. Pretty much all non-trivial applications can be summed up as doing a series of operations that receive data from some previous operation, perform some work on that data, and then pass that data along to the next operation. Furthermore, operations can be seen as abstractions of a series of smaller, child operations, which in turn call their own child operations, until the most basic operations are reached, as illustrated in figure 3. This type of analysis can feasibly be done by an automated process that can create such graphs for any existing codebase or system.

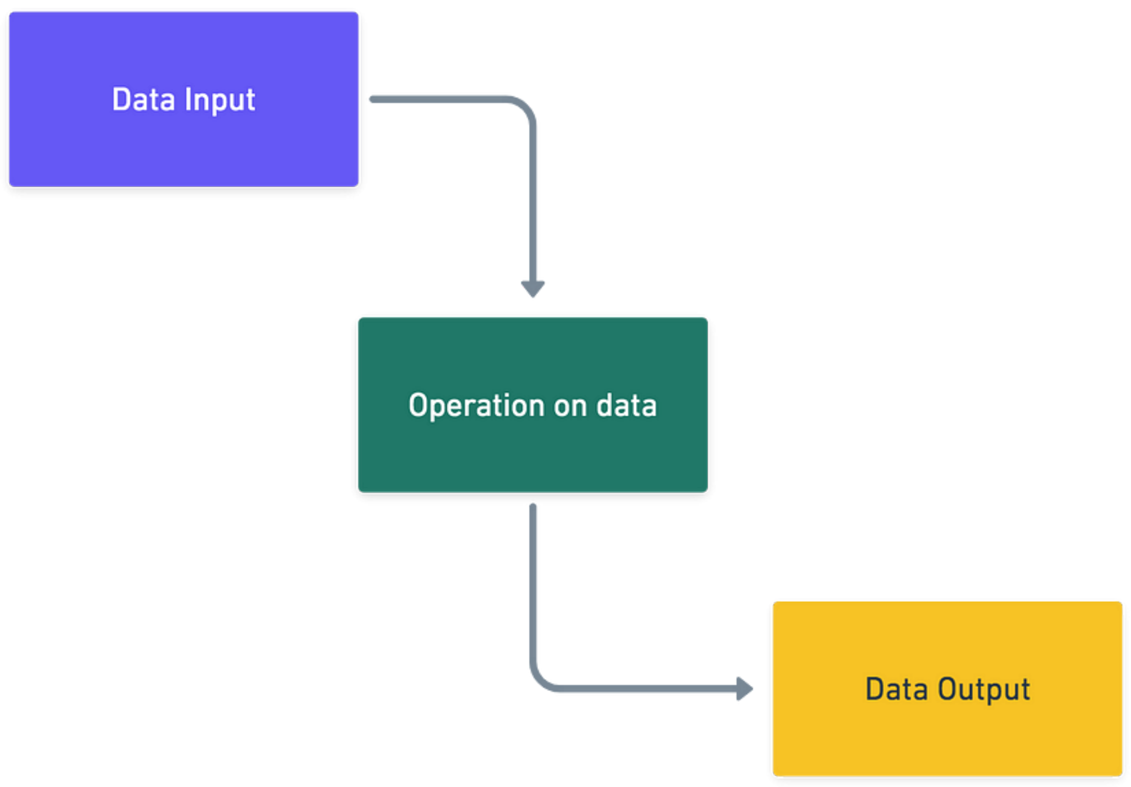


Figure 2: Code as a series of data operations

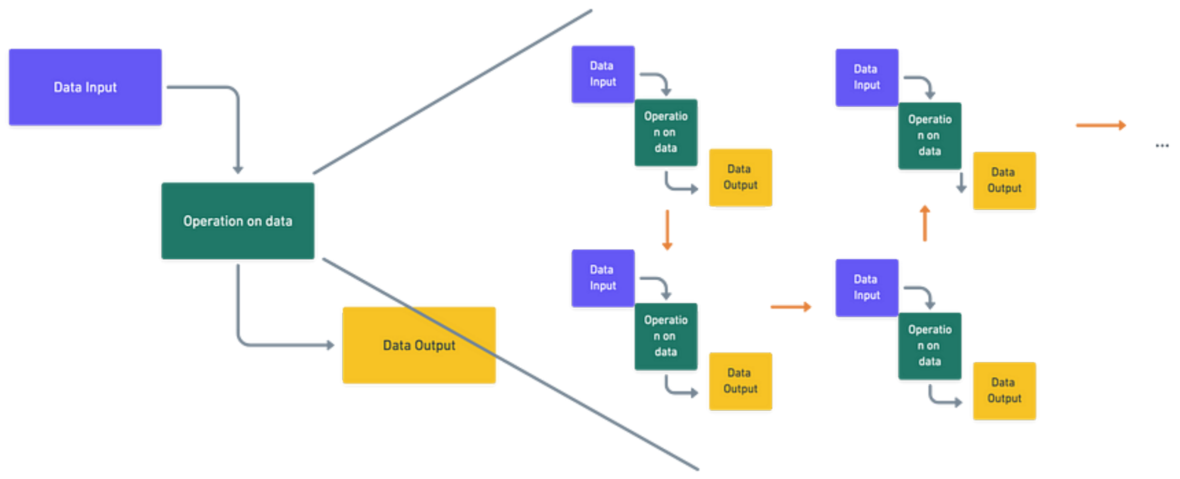


Figure 3: The analysis can be applied as a fractal

To create systematic scalability, therefore, we can analyze our code and figure out where these data transitions are most crucial, e.g. where we would gain the most from storing the intermediate state before moving on to the next step. Letâ€™s look at an example. Say Widget Co makes 1 million widgets per day and has a database row per widget made. It has to pull a report that gets all their widgets constructed last week, perform statistical analysis on them, and figure out which widgets took longer than some percentile to analyze process bottlenecks. Their current process looks like the following: Fetch all widget rows, calculate results on all widget rows, filter down to just the slow widgets, construct a CSV report of the results.The process is slow and the single huge box performing this calculation is unable to go any faster even with many forks performing separate database operations/calculations. How do we fix it?

First, we must analyze the major data transitions that are happening in this process. There are three large components that we can immediately see, shown in figure 4.

- First, there is a process that fetches raw widget data out of the database, turns this raw data into processable entities, and stores these entities in memory.
- Second, there is a process that takes these entities and performs statistical analysis on them, storing the resulting aggregates in memory.
- Third, there is a process that uses the aggregates to construct the CSV rows of the report and outputs the final report somewhere as the final goal of the task. We can see that if we were to store the intermediate states between these three processes, we can greatly increase the scalability potential of this system.

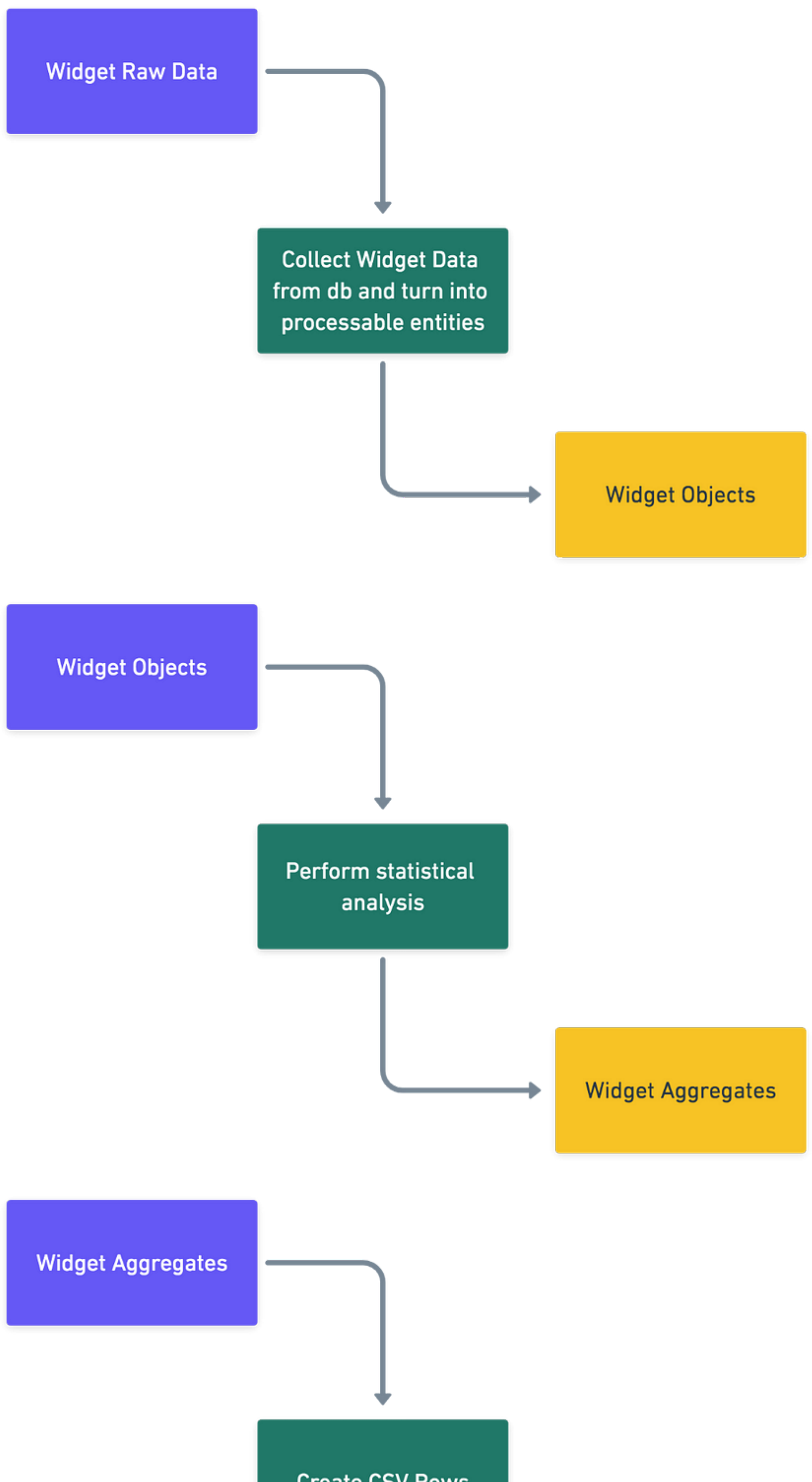
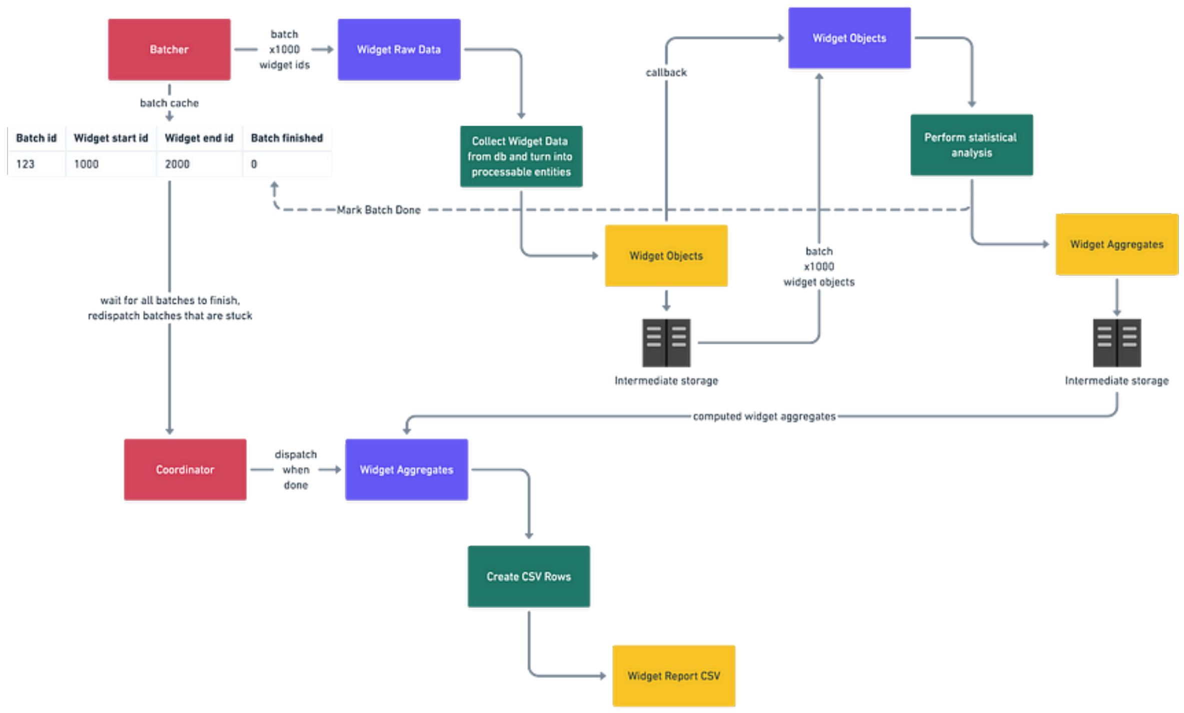

Widget Raw Data
Collect Widget Data from db and turn into processable entities
Widget Objects
Widget Objects
Perform statistical analysis
Widget Aggregates
Widget Aggregates
Create CSV Rows

Figure 4: The sub processes composing our process

Here we can begin with batch processing, which could be defined as splitting our workload into â€œbatches,â€� or work sets that have a maximum fixed size. We can maintain the state of each batch independently in a batch cache, using a batch id as the key and the batch state as the value. We would create a batcher process that simply reads 1000 row ids out of the database at a time and assigns these rows to a batch id via the batch cache. The batcher can be parallelized if necessary, but even a single batcher process can be very fast at creating batches for sub-billion scale, since it only has to read the entity references from our centralized data store and cut them into lists of 1000. These batches would be stored in the batch cache with at a minimum a batch id, job id, batch widget ids, and batch status. We then move onto a MapReduce operation, which consists of mappers, or workers that filter and sort data, and reducers, or workers that perform summary operations on the resulting data. Our mapper would read the raw rows out of the database and transform them into workable objects, storing an easily loadable version of these objects in batched form in our intermediate storage. Our reducer would then load the batched objects, perform statistical aggregation on them, and output the aggregate to our intermediate storage. An automated process can feasibly generate the code for mappers and reducers necessary for a given problem.

We then require a coordinator that monitors the batches in the batch cache and dispatches to a final CSV worker when all the batches are complete. The CSV worker is a reducer that loads all the batch aggregates from storage, combines them into CSV elements, then outputs the CSV. This can be either one worker generating the whole CSV or multiple workers to generate sub-CSVs, then combine them together in another reducer. We have just followed a systematic process of (1) analyzing the process to identify sub processes, (2) storing the intermediate state between those sub processes, and (3) combining the results of these sub processes through a reducer framework to convert a single monolithic process into a series of steps that can be independently scaled, as illustrated in figure 5. The only fuzzy part of this approach is the intermediate storage. How are we going to store the data between the steps in an efficient manner?



Batch id	Widget start id	Widget end id	Batch finished
123	1000	2000	0

Figure 5: Systematically scaling the process by breaking it down into components

Potential lossy solutions for data storage include tasks/queues and caching. Tasks can hold the state from the previous process as the input for the next process. For example, if we render an email in one task, we can put the full HTML payload as the task argument to the next process. The problem is that task brokers, which hold the tasks that are meant to be executed, are generally centralized and their storage is generally ephemeral for performance reasons. So at some point, we will need a lot of task brokers to hold all our data. If a broker goes down, the tasks are not guaranteed to survive. Tasks that run out of retries also disappear.

As for caching, we can implement a temporary key value storage to hold the intermediate state. For example, we can render an email in one task, and then put the full HTML payload into a cache so the next process can pick it up. The problem is that caches are designed to be fixed size, so they can, should, and will kick out any stale data once they get past their size maximum (or that data expires). This means we need more and more caches as we grow, and we still have to deal with cached data randomly disappearing due to expiration/eviction. This can create back-pressure in the system because process 1 has to re-run if process 2 that depends on process 1 has no cache to start with, etc.

Potential lossless solutions include various types of permanent disk storage. We would ideally like to use a cheap, permanent, extremely easy to scale storage such as disk space to store our intermediate data. But, just one disk isnâ€™t really enough. We need many disks â€” to store all our data, enable fast reads by splitting up data, and for redundancy. We can start with a series of raid 0 systems (to increase read/write speed) that have some sort of shard split for the data and connect to other systems that serve as replicas via a different type of raid, then put a SQLite, NFS, or maybe even HTTP interface on top of that so we can query for the data when we need to read it back out. Figure 6 illustrates the basic layout.

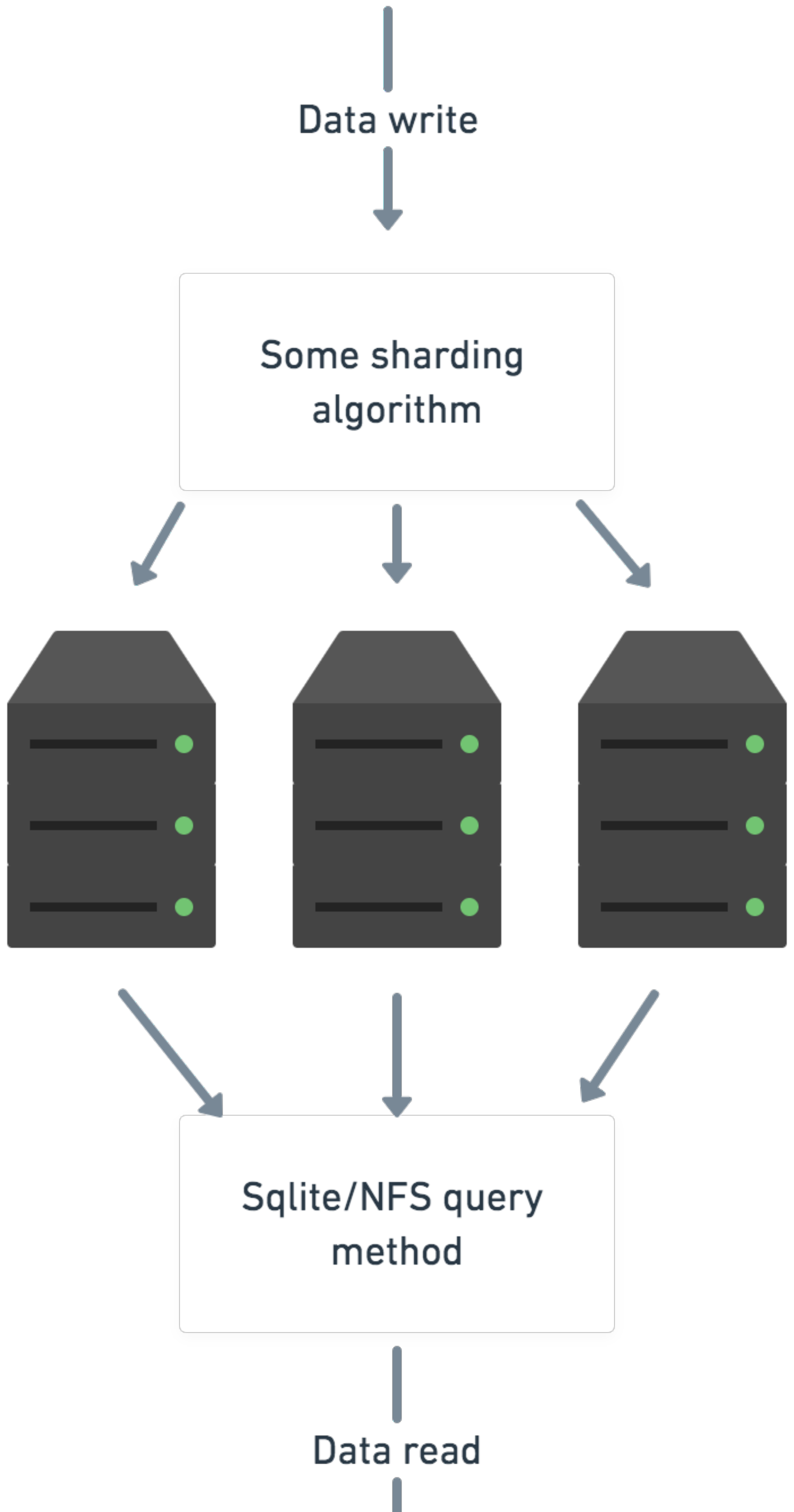

Data write
Some sharding algorithm
Sqlite/NFS query method
Data read

Figure 6: Proposed lossless storage solution

If this type of system seems familiar, thatâ€™s because it should be. We have effectively just reinvented a prototype version of Amazon S3 as our ultimate scalability solution. It turns out that S3 (and similar services at other cloud providers) already has the fast writes, fast reads, unlimited capacity, and data redundancy (and even versioning and lifecycle policies) that we need for our intermediate data storage solution. S3 has incredibly low latency considering its throughput. In our production environment the amount of time to read out roughly 3â€“5 megabyte-size files into a worker is around 500â€“750ms. Each file corresponds to a fully rendered email sending batch. S3 supports an extremely large number of operations per prefix key, so if we can structure our bucket prefixes in the correct manner, we can get effectively unlimited throughput. We can send 3,500 PUT/COPY/POST/DELETE or 5,500 GET/HEAD requests per second per prefix in an Amazon S3 bucket. There are no limits to the number of prefixes that we can have in our bucket.

To recap, if we want to create a system where we can take any arbitrary workload and split it in a systematic way to give it higher scalability potential, we need to think of that workload as a series of subprocesses that are effectively just operations on data. We can split our workload by taking the output of one subprocess and feeding it in as inputs to another subprocess, as shown in figure 7. We need to store that intermediate data in a high-throughput, massive, lossless, preferably low-latency environment. While we can build our own storage solution (and may need to if our latency requirements are extreme), S3 happens to be extremely well suited to these criteria in the average case.

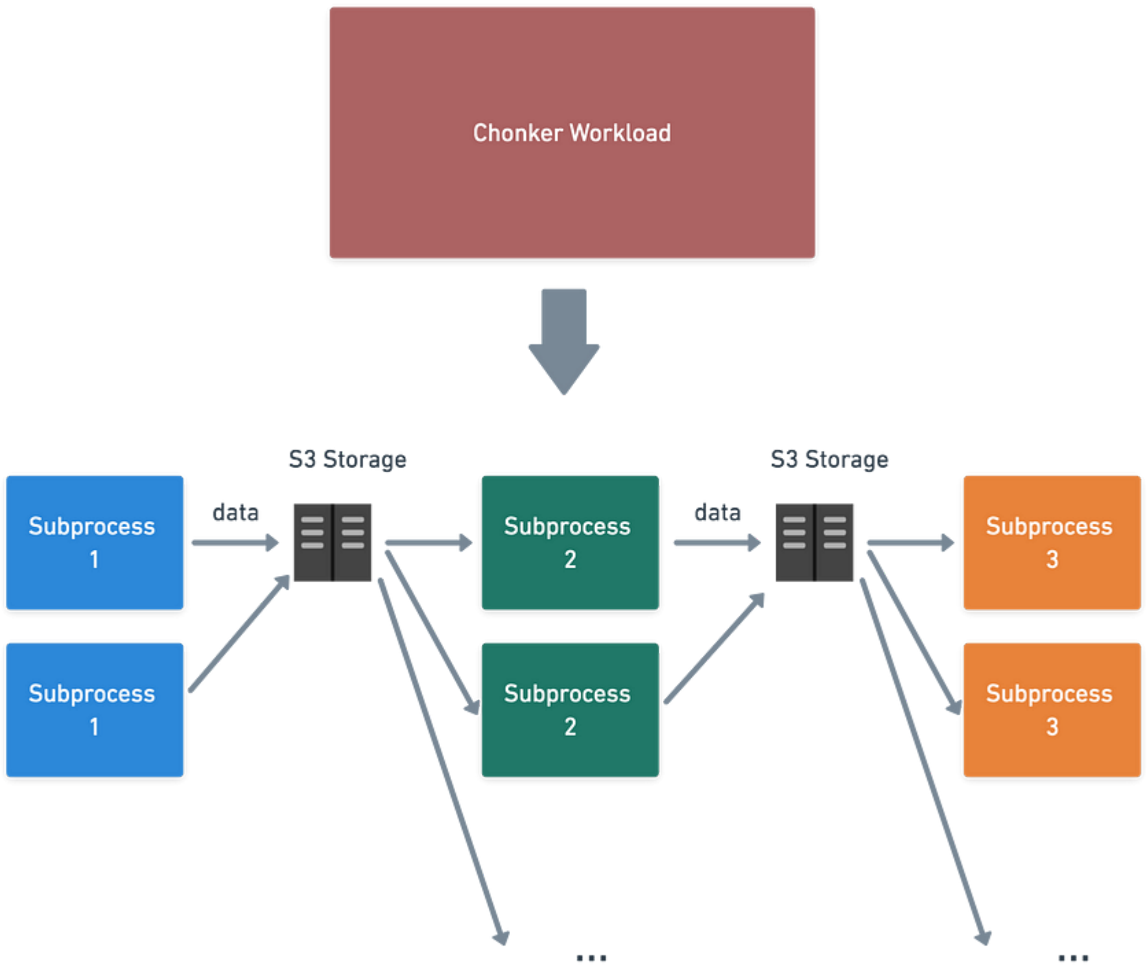


Figure 7: Systematic scaling

As a case study, we will look at a project we completed internally to increase sending scalability. In our sending pipeline, sending an email to a batch of recipients consists of two operations, rendering and sending. Rendering is generally very expensive because a vast array of information has to be processed in complex templates to generate some end-goal HTML that represents the emailâ€™s content. Each individual email recipient can have a slightly different output HTML depending on the specifics of the template, the underlying customer information, the data sources involved, etc. The scalability problem we were facing was that for email sending, rendering and sending took place in the same task, so any direct email sends that failed would restart the entire batch. Each batch contains hundreds of recipients and we would have to re-render certain recipients before sending again. Therefore any downstream sending outages could easily spiral into huge queue backups as we struggled to re-render everything to clear the queue for new sends to come in.

Following a similar logic to the process described previously, the proposed solution was to split rendering and sending into separate tasks so that rendering was not affected by sending backups. The biggest problem was in the storage, as shown in figure 8. Where would we store the intermediate renders, if each recipientâ€™s render is slightly different from the next one, and each can be a massive blob of HTML? When combined, it would be easily multiple megabytes per batch of storage, which could easily result in terabytes of storage when scaled up to normal volume. At first we thought about using tasks as intermediate storage, but then gave up after realizing the number and size of brokers as well as operational complexity we would need to accomplish this.

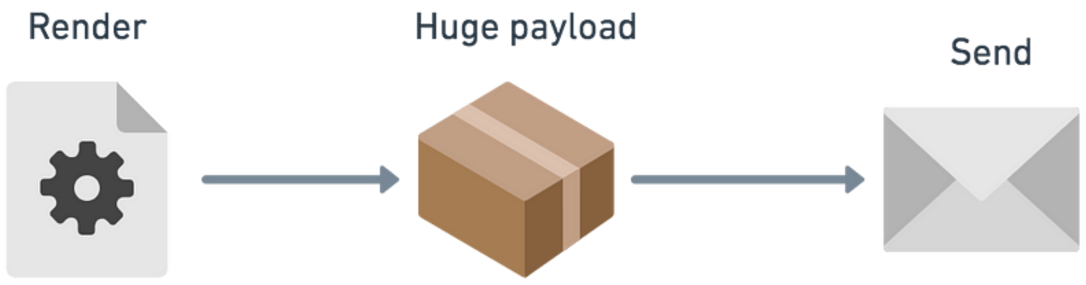


Figure 8: The basic problem

The solution, illustrated in figure 9, was that we created a service boundary where the rendering team would own all the rendering and use S3 keys structured as /core-bucket/[message_id]/[batch_id]/â€¦ to store the compressed artifacts and metadata files that signified when an artifact has started to process and when it finished. As the sending team, we would own the sending side of the boundary, where we used the rendering teamâ€™s methods to pull the artifact from S3 once it was considered ready. The task hop was structured as follows: First, the email dispatch system creates batches to be sent for the message. These are scheduled into the rendering system with an assigned priority and send-by date. Rendering takes over and uses internal dispatchers and executors to complete payloads in the most efficient manner. Once a batch is complete, rendering calls a callback function owned by the sending team. As an initial implementation, this directly creates a sending task that uses the batch information to pull the pre-rendered artifact from S3, construct the email payloads and send the messages.

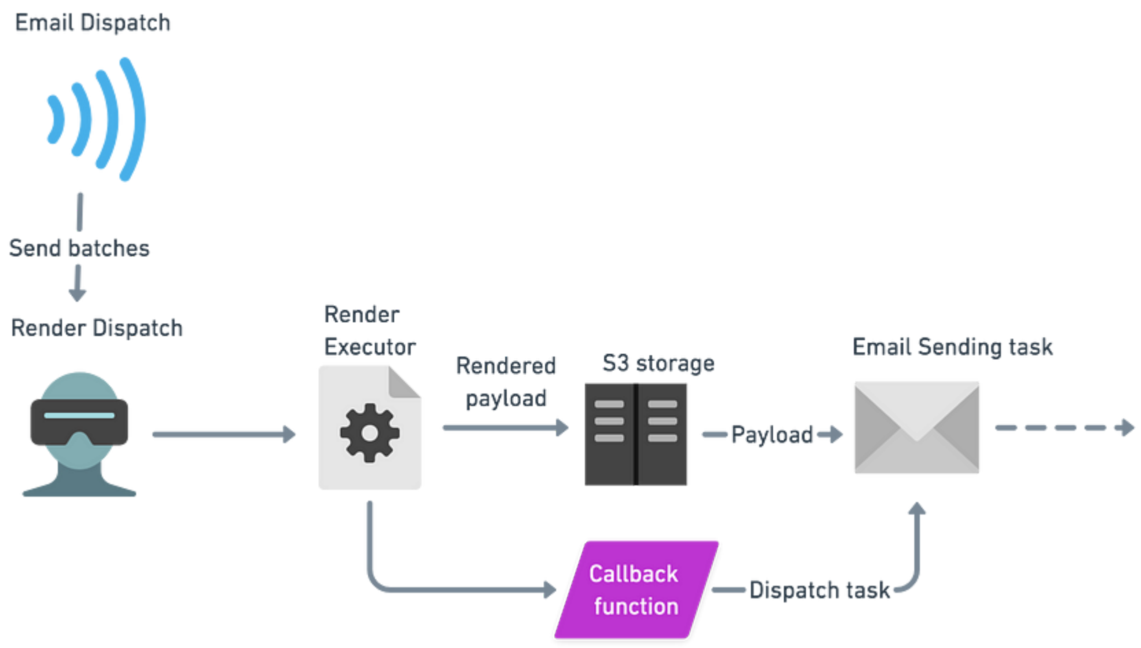


Figure 9: The solution

From our load testing, we determined that S3â€™s latency and throughput were an extremely attractive option. The sending tasks themselves went from spending over half their time on rendering to spending roughly 750ms out of 23 seconds, or about 3% of their time on pulling the artifact with fully rendered HTML. We were able to hit our scale targets and, most importantly, accomplished the goal of separating rendering from sending. Sending could now retry endlessly in sending outage scenarios without worrying about the extra time spent re-rendering email payloads.

Splitting the workload in this case was not just a scalability win, however. We were able to create a tight service boundary around template rendering, even though rendering happens in between processes owned by the sending team (scheduling, dispatch, sending). Intermediate storage allows these kinds of splits to happen, because each process separated via storage becomes independent of the other processes, so ownership can be assigned separately and teams do not have to own the whole thing. The rendering team was also able to get much better control over when, where, and how rendering happened, whereas before they were dependent on the organization of sending boxes. They can now create their own complex priority systems that allow them to manage their workloads in a way that makes sense given their deep knowledge of rendering.

We were also able to simplify our sending queues because our tasks became more homogeneous. This is because most of our sending queue divisions were mostly based on the different rendering patterns that a sending task could go through and had nothing to do with sending. By removing the complexity of rendering from the sending task, we were able to make all sending tasks roughly equal in terms of processing time and intensity. In addition, the first-come first-serve nature of the callback publishing process effectively randomized the ordering of batches. Combined with fast processing and homogeneity this allowed us to put all message sending into just a single queue as opposed to more than six previously, which in turn simplified our autoscaling logic.

Most processes that are hard to scale can be broken down into much easier-to-scale sub processes as long as there is some form of intermediate storage that can hold the results of one subprocess and feed them to the next subprocess. This allows the subprocesses to be scaled and operated independently of each other, which makes it easy to identify and address bottlenecks and create service boundaries. For most processes that value throughput more than latency, S3 is a built-in solution for AWS that allows a very simple intermediate storage implementation. Other cloud providers have similar solutions that should work equally well with their infrastructure.

For our use cases, S3 has other characteristics that make it useful as an intermediate storage medium. It is cheap to store data and gets cheaper the more data we put into it, which means we can store lots of intermediate storage without huge cost concerns. It has lifecycle policies that can automatically delete or archive data once we donâ€™t need it anymore, such as when a job is completed. Everything can be accessed via web API so it is easy to build inspection tools on intermediate state to see where something might be going wrong, even long after the process has finished running. This is extremely useful because it can be transformed into new use cases, such as allowing end users to see the intermediate results of a process. One example is that end users can now potentially see a historical fully rendered template at the time of sending for a particular customer. Finally, it has storage tiers with automated lifecycle movements that can make storing gigantic amounts of data even cheaper at the cost of greater latency. In general when constructing complex systems we want to use as many pre-built and battle tested components as possible to speed up implementation. So it works out nicely when an existing storage system is effective for semi-automated scaling of processes for the vast majority of cases.