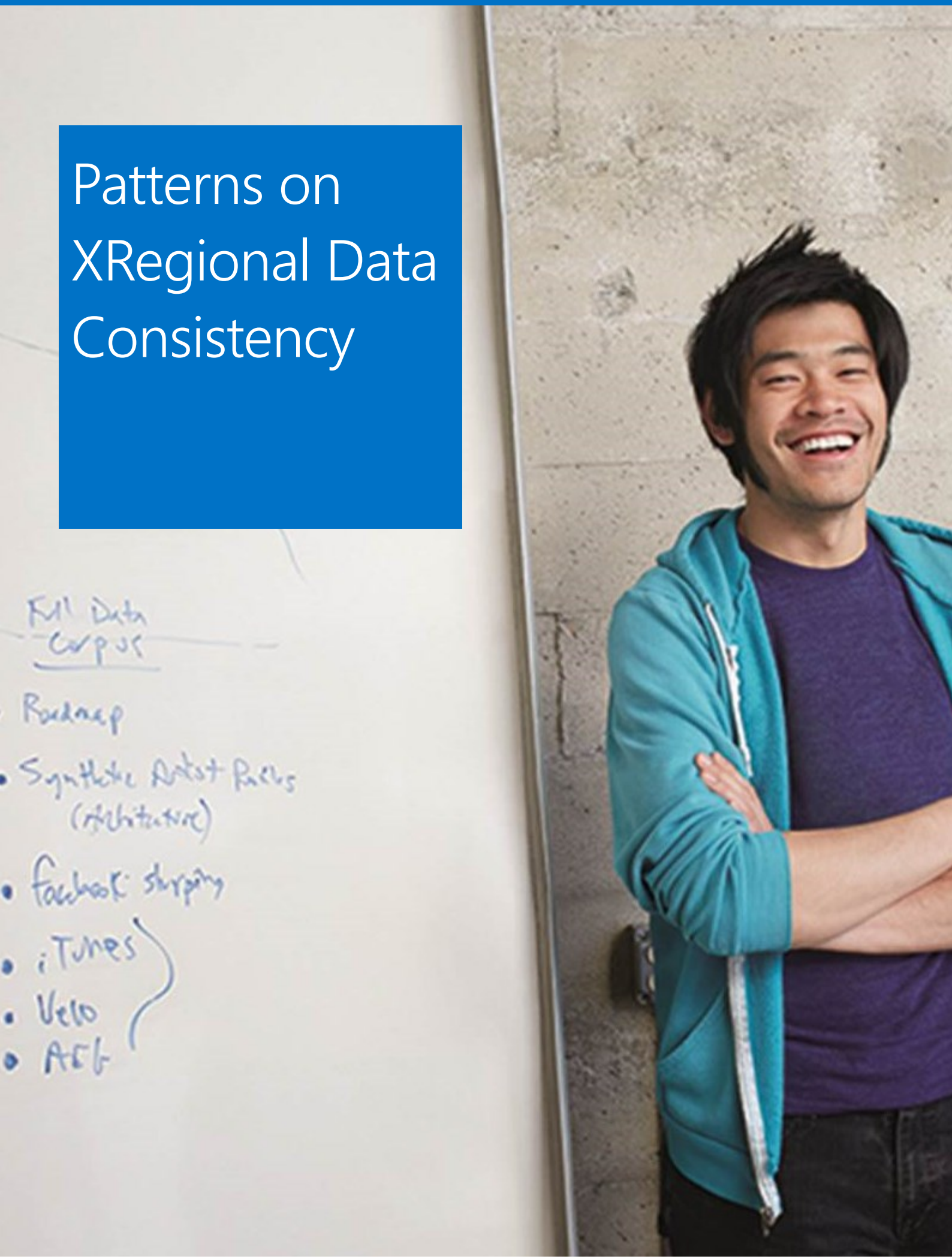
Patterns on XRegional Data Consistency



Contents

The problem

High-scale Microsoft Azure solutions require sophisticated deployment models. If you want to go multi-national or provide 100-percent uptime to your customers, you must think cross-regionally and deploy your services in multiple Azure regions. While deploying the copy of an executable code such as a website or a cloud service may seem straightforward, the issue of data replication and consistency is a key challenge. Dealing with data consistency across multiple independent nodes demands some forethought and may bring you to full redesign of your application.

There is no "silver bullet" to all consistency problems; however, we observe certain design patterns and approaches used in different projects with high scale requirements. Examples of such projects include a huge, multi-national sporting event, like the Winter Olympic Games, or a viral marketing campaign run by one of the largest beer companies. For each of these systems, one Azure region is just not enough. If there were an outage in that region, the application would become unavailable. In addition, placing the application closer to end users gives them a better experience in terms of latency and performance.

A scalable application is naturally a distributed application. Data consistency in distributed systems is an intriguing technology challenge. We recommend that you read Werner Vogel's blog posts on eventual consistency as a common solution to this challenge: http://www.allthingsdistributed.com/2008/12/eventually_consistent.html.

Introducing XRegional

We crafted a simple framework, called *XRegional*, that demonstrates the approach described in this white paper. The framework can be used out of the box with data stores like Azure Table Storage or Document Db, and we plan to extend it to support Microsoft SQL Azure. If your project requires you to use another storage platform, you still can borrow ideas or—better yet—contribute by extending the code. The framework is located at https://github.com/stascode/XRegional.

If you're wondering why we would create this kind of framework when many Azure data services have geo-redundant options out of the box, note that those options are often limited to a specific geographic region (such as West Europe and North Europe, not West Europe and Southeast Asia). There also is no SLA on how fast your data will catch up and become consistent.

Plus, our approach is aimed primarily at the *one writer/multiple readers* scenario, where writes occur strictly in one region (usually called *primary*) but reads occur in all regions (Figure 1). The pattern allows for failover to another write location, though, in case the primary write region experiences an outage. There is also an additional constraint: The approach deals with full entity (or document—from here on, *entity* refers to both Azure Table Storage entity and Document Db document) state updates, not partial updates.

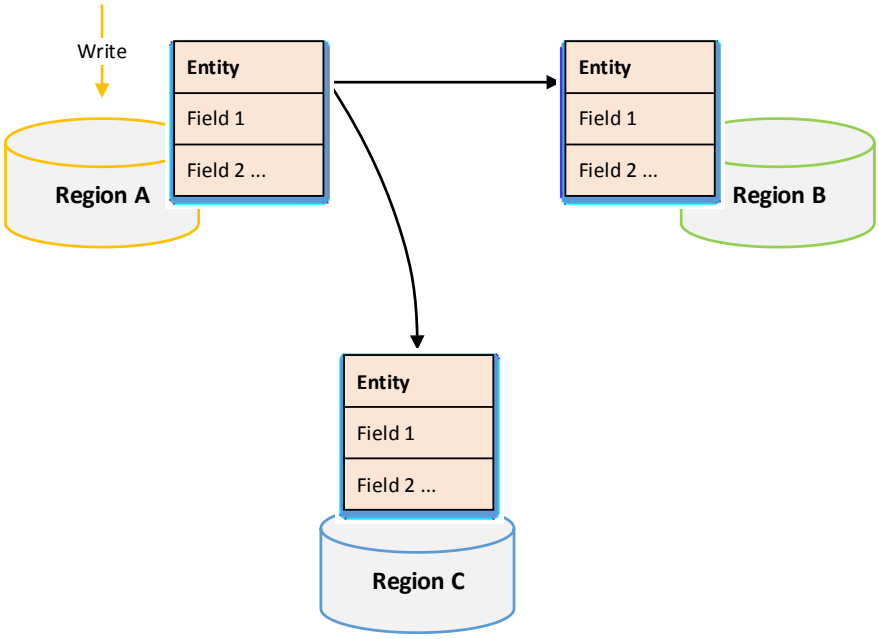


Figure 1. One writer/multiple readers with full state updates

On the other hand, you might redesign your data structure such that the final entity structure consists of multiple data parts—each independent of the other and updated solely by a region it represents (Figure 2).

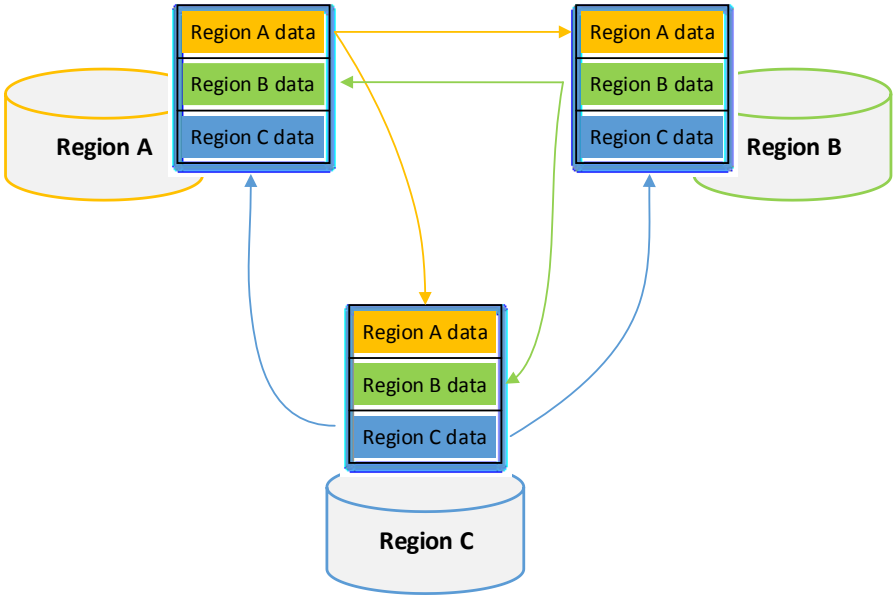


Figure 2. Entity with structure composed of independent, "region-specific" parts

To explain our focus on NoSQL stores, we believe that while SQL database technology is exceptional at giving you the highest level of data consistency and transaction atomicity, it forces you into a certain mindset limited by the richness of that specific technology. Simply stated, it often falls short when it comes to distributed systems. Yes, there are options like AlwaysOn and others that help to mitigate consistency issues, but they all are about strict ACID behavior (Atomicity, Consistency, Isolation, and Durability).

On the flip side, we've seen that real high-scale projects tend to be built on top of NoSQL technology like Azure Table Storage and, recently, Document Db. NoSQL scales; it does not

confine you to a specific data schema; and it gives you more freedom. However, with this freedom comes the responsibility to design your application differently and put enough thought into the foundation of that design.

The solution

The pattern is shown in Figure 3. The primary region is where a write operation occurs. After the writer has persisted data to the primary region's data store, it immediately places that data in the Azure queue, which is regularly polled by the secondary region's worker. As soon as the worker fetches data from the queue, it persists that data to the data store of its region. As soon as the primary region is updated, the data becomes temporarily inconsistent across the regions. However, in a short while, the other regions catch up and the data comes to the consistent state. So with this approach, we introduce the classic eventual consistency to our system.

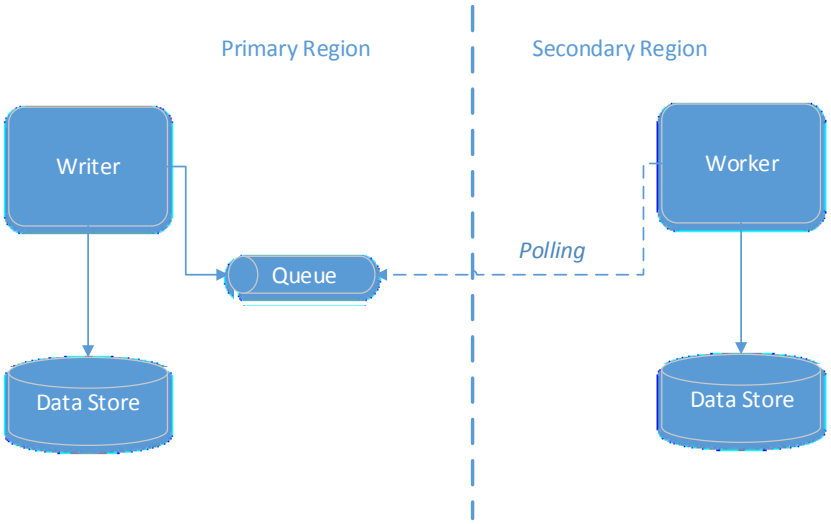


Figure 3. The "queue/worker" pattern

A key benefit of this design is that it scales. It is resilient to intermittent connection losses between the regions—messages will stack up in the queue until the connection is recovered. It also allows you to perform load leveling on the data store by amending the number of the worker's instances/threads to provide a balance on dealing with the queue. Moreover, if you want another secondary region, you just add another queue.

This picture may seem straightforward, but there are a number of challenges to be dealt with. How do you handle an update sequence of the same entity? For our framework, we chose Azure Storage Queue as a simple and fast technology, but it comes with no guarantee on ordered delivery of messages. Even if the "order delivery" problem is solved (see the Some Considerations section below), you still have to take in mind the worker polling the queue in an eternal loop. A well-designed system should have resiliency in place, so one instance of the worker is not enough. In the Azure world, the worker would be deployed as a Cloud Service multi-instance worker role or a regularly scheduled Azure WebSites web job. Each of these instances or jobs would compete with the others for messages in the queue.

You might argue that a continuous web job would fix the problem because it runs only one instance with a failover capability, but one instance is limited to parallelization. In other words, it is not scalable (or limitedly scalable)!

So there are unordered messages in the queue and multiple instances of the worker, each one racing to fetch the next message from the queue and then persist it in the data store. Now think what could happen if the writer enqueues updates to the same entity in order A, B, and C, but the worker instances fetch them in a different order, such as B, C, and A. (Here updates A, B, and C represent the full entity state, not partial updates, as discussed above). Chances are that the entity's final state in the secondary region would become A. That is, the data would become inconsistent across regions.

Enabling consistency

Here's how we solved the problem of consistency: We added a number field called *Version* to the entity data structure. Each time the writer in the primary region updates an entity, it correspondingly increases its version by one. We've been strategic to choose Int64 as the data type for the Version field because it should have enough capacity for the majority of scenarios (Figure 4).

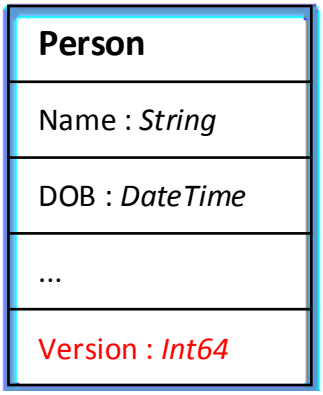


Figure 4. Example of a Person entity with the Version field

Processing messages in the secondary region entails the following operation:

1. Fetch the next message from the queue.
2. Compare the dequeued entity's version value with the version of the same entity already stored in the data store.
3. If the dequeued version is less than the existing one, discard the message because there is a more recent version of the entity in the data store.
4. If the dequeued version is greater than that in the data store, update the data store with the dequeued entity.

It gets tricky if you imagine this same procedure executing in parallel for multiple messages and you can't put a lock on the row in the data store. (Neither Table Storage nor Document Db allows locking.) This is where optimistic concurrency comes in: Every time you read an entity from Table Storage or Document Db, you get back an ETag, in addition to the payload. The ETag is an obscure value, and its only purpose is to give you a way to deal with

concurrency. The ETag's value changes every time the corresponding underlying entity is updated. If you attach the same ETag value as it was read to your update request, the request will succeed if the underlying entity has stayed intact; it will fail if the underlying entity has been updated by another thread/process/worker since the last read operation.

The ETag comes from the HTTP protocol space, where a client can issue an HTTP request with an If-Match header condition, and if that condition fails, the server returns HTTP status code 412. And what happens if you attach the ETag, but the write operation fails with code 412? You just retry the operation from Step 2 through Step 4. You can find more on the ETag at http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.24.

Now let's see the refined write operation performed by the worker, as shown in Figure 5.

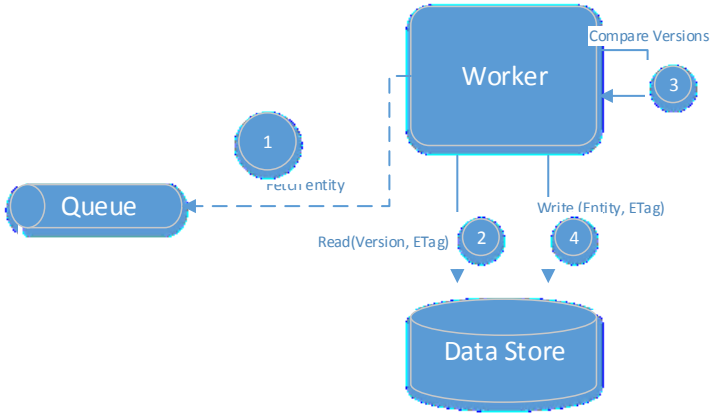


Figure 5. Writing scheme in the secondary region

1. Fetch the next message from the queue.
2. Read the underlying entity's version and ETag from the data store. (You have the entity's ID or key from Step 1.)
3. Compare the dequeued version with the underlying entity's version value. If it is less, discard the message and go to Step 1.
4. Otherwise, try to write the dequeued entity with the attached ETag. If the data store returns HTTP code 412, retry the attempt from Step 2.

One question still remains: What if the entity does not yet exist in the data store? You actually would get HTTP code 404 from Table Storage or Document Db on Step 2 because that read request resulted in no match. Apparently, you may safely continue and persist the dequeued entity without attaching any ETags. But what if another instance of the worker has persisted this entity (its other version) in between the read and write operation? Luckily, in this case, both Table Storage and Document Db return HTTP code 409 (Conflict), giving you the chance to fix the issue and try again. As such, the entire operation would start from Step 2 again.

Most importantly, this algorithm is **idempotent**, which means that it has no side effects and can safely be executed repeatedly. Ultimately, the data store will always have only the last version of the entity, and because it is idempotent, it scales out.

Our discussion so far has focused on the secondary region, but what about the writer in the primary region? How do you deal with multiple parallel instances of the writer trying to

update the same entity? There is no one-size-fits-all answer, but you still have the power of ETags and optimistic concurrency. You may want to keep the ETag while a user updates the entity, and then warn her that the entity has been changed since she started the updates. This gives the user a chance to retry the attempt. In our framework, we implemented a *brutal* writer—that is, the writer cares only about correctly increasing the version value and persisting the updated entity. If the entity has been changed in between, the writer just repeats the write operation, increasing the version by one again.

Finally, you may have noticed that there are no transactional boundaries in the primary region on the write operation to the data store and subsequent writes to the queue. Because you cannot promote a distributed transaction across Azure Storage components and/or other Azure services, you should mitigate potential disruptions between these write operations. The mitigation strategy is to deploy all services within one datacenter, enclosing them in a single large fault domain. You can also go further and use the same Azure Storage account for both the queue and the data store. In addition, you should use retry policies over the queue to avoid errors due to intermittent networking issues. It is important to note, though, that you should employ a monitoring subsystem and error logging to detect such issues promptly. In a real project, we had a monitoring system that would send an immediate alert so that we could look into the problem and potentially retry the entire write operation.

The XRegional Framework: A closer look

Our sample XRegional Framework is located on Github: https://github.com/stascode/XRegional. It currently supports both Azure Table Storage and Document Db according to the approach described above. As with any code library, XRegional has a specific terminology and design, which are discussed in this section.

XRegional introduces the simple concept of *gateway*, which is used in the primary region to send data to other Azure regions. There is one interface, *IGatewayWriter*, and two implementations, *GatewayQueueWriter* and *GatewayMultiQueueWriter*, which send data respectively to one queue or multiple queues. *GatewayQueueWriter* is smart enough to store a message to Azure Blob storage if it exceeds the 64 KB limit, passing a message-pointer instead. A message is zipped and passed as an array of bytes for maximum efficiency. Conversely, *GatewayQueueReader* is used in the secondary region and simply dequeues the next message from the gateway queue, calling either a callback to subsequently process the message or an "on-error" callback to provide the opportunity to process the poisonous message. Instead of *GatewayQueueReader*, you would use *TableGatewayQueueProcessor* for Azure Tables and *DocdbGatewayQueueProcessor* for Document Db because those two classes implement writing to the secondary region's data store for you.

To start executing writes to either an Azure Table or Document Db collection, you would use *SourceTable* or *SourceCollection*, respectively. Both implement the brutal writer approach noted earlier. Their counterparts, *TargetTable* and *TargetCollection*, implement the write operation following the ETag rules in the secondary region.

Figure 6 shows how all of these classes work together:

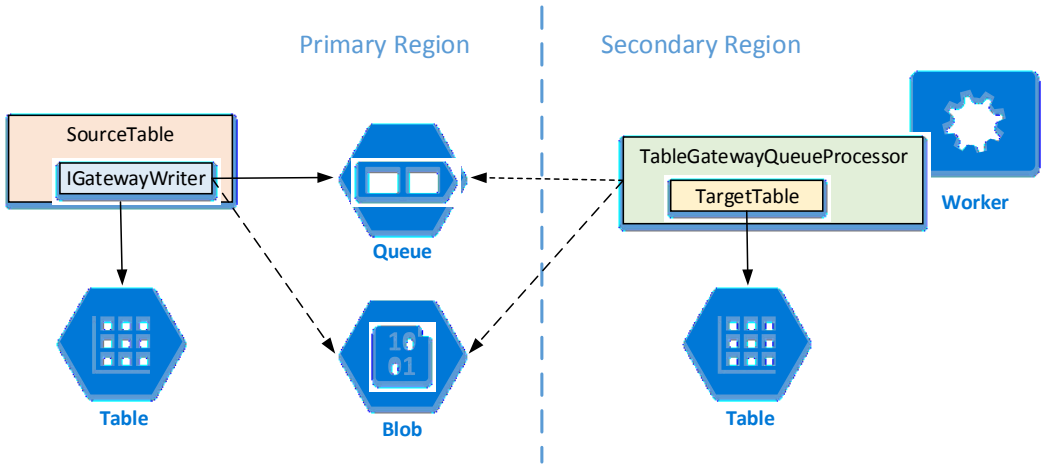


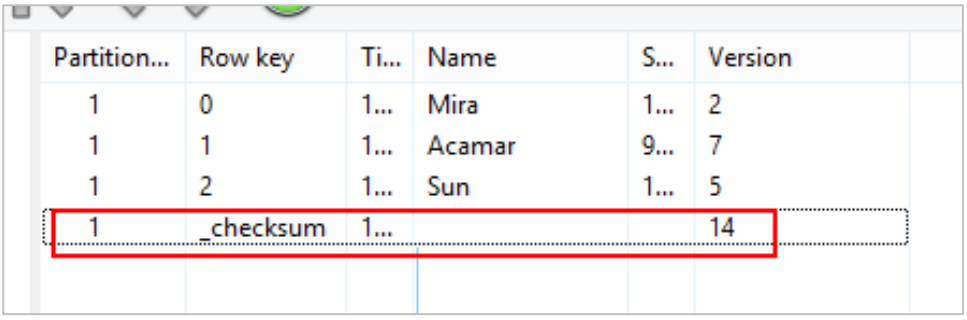
Figure 6. XRegional classes scheme

Remember that the best way to learn the code is to see it, test it, and play with it. We recommend that you take a look, run some tests, and adjust and apply the code for your needs.

Some considerations

Although this white paper discusses the standard approach to the XRegional Framework, it is in no way dogmatic, and you can make adjustments to best fit your situation. The first obvious modification is another queuing technology, such as Azure Service Bus. If you use a Service Bus topic as the queue, you get the benefits of having just one topic and multiple subscribers (regions). Service Bus also provides good options for the guaranteed delivery of messages. However, keep in mind that those options come at a cost—and not just in money. As your options grow more sophisticated, you limit the scalability of your system because ordered delivery becomes a performance constraint in the design of your application.

Table Storage allows you to persist entities in a batch that executes in a single transaction. A batch can contain entities only with the same partition key. In one of our projects, we used this as an advantage for quick verification of data consistency across all regions. We simply stored an additional *checksum* entity whose version was the sum of all entities' versions within the same table and partition key. This entity was updated in a single batch, in one transaction with other entities. To see if the data was consistent across Azure regions, we just checked this special entity's version value in all regions—which saved a lot of time. Figure 7 shows an example of the special checksum row that has the row key value as *_checksum*, with its version value being the sum of all entities within that partition.



Partition...	Row key	Ti...	Name	S...	Version
1	0	1...	Mira	1...	2
1	1	1...	Acamar	9...	7
1	2	1...	Sun	1...	5
1	_checksum	1...			14

Figure 7. Example of the checksum row