CHAPTER 5

Splitting the Monolith

We've discussed what a good service looks like, and why smaller servers may be better for us. We also previously discussed the importance of being able to evolve the design of our systems. But how do we handle the fact that we may already have a large number of codebases lying about that don't follow these patterns? How do we go about decomposing these monolithic applications without having to embark on a big-bang rewrite?

The monolith grows over time. It acquires new functionality and lines of code at an alarming rate. Before long it becomes a big, scary giant presence in our organization that people are scared to touch or change. But all is not lost! With the right tools at our disposal, we can slay this beast.

It's All About Seams

We discussed in Chapter 3 that we want our services to be highly cohesive and loosely coupled. The problem with the monolith is that all too often it is the opposite of both. Rather than tend toward cohesion, and keep things together that tend to change together, we acquire and stick together all sorts of unrelated code. Likewise, loose coupling doesn't really exist: if I want to make a change to a line of code, I may be able to do that easily enough, but I cannot deploy that change without potentially impacting much of the rest of the monolith, and I'll certainly have to redeploy the entire system.

In his book *Working Effectively with Legacy Code* (Prentice-Hall), Michael Feathers defines the concept of a *seam*—that is, a portion of the code that can be treated in isolation and worked on without impacting the rest of the codebase. We also want to identify seams. But rather than finding them for the purpose of cleaning up our codebase, we want to identify seams that can become service boundaries.

So what makes a good seam? Well, as we discussed previously, bounded contexts make excellent seams, because by definition they represent cohesive and yet loosely coupled boundaries in an organization. So the first step is to start identifying these boundaries in our code.

Most programming languages provide namespace concepts that allow us to group similar code together. Java's `package` concept is a fairly weak example, but gives us much of what we need. All other mainstream programming languages have similar concepts built in, with JavaScript very arguably being an exception.

Breaking Apart MusicCorp

Imagine we have a large backend monolithic service that represents a substantial amount of the behavior of MusicCorp's online systems. To start, we should identify the high-level bounded contexts that we think exist in our organization, as we discussed in Chapter 3. Then we want to try to understand what bounded contexts the monolith maps to. Let's imagine that initially we identify four contexts we think our monolithic backend covers:

Catalog
: Everything to do with metadata about the items we offer for sale

Finance
: Reporting for accounts, payments, refunds, etc.

Warehouse
: Dispatching and returning of customer orders, managing inventory levels, etc.

Recommendation
: Our patent-pending, revolutionary recommendation system, which is highly complex code written by a team with more PhDs than the average science lab

The first thing to do is to create packages representing these contexts, and then move the existing code into them. With modern IDEs, code movement can be done automatically via refactorings, and can be done incrementally while we are doing other things. You'll still need tests to catch any breakages made by moving code, however, especially if you're using a dynamically typed language where the IDEs have a harder time of performing refactoring. Over time, we start to see what code fits well, and what code is *left over* and doesn't really fit anywhere. This remaining code will often identify bounded contexts we might have missed!

During this process we can use code to analyze the dependencies between these packages too. Our code should represent our organization, so our packages representing the bounded contexts in our organization should interact in the same way the real-life organizational groups in our domain interact. For example, tools like Structure 101 allow us to see the dependencies between packages graphically. If we spot things that

look wrong—for example, the warehouse package depends on code in the finance package when no such dependency exists in the real organization—then we can investigate this problem and try to resolve it.

This process could take an afternoon on a small codebase, or several weeks or months when you're dealing with millions of lines of code. You may not need to sort all code into domain-oriented packages before splitting out your first service, and indeed it can be more valuable to concentrate your effort in one place. There is no need for this to be a big-bang approach. It is something that can be done bit by bit, day by day, and we have a lot of tools at our disposal to track our progress.

So now that we have our codebase organized around these seams, what next?

The Reasons to Split the Monolith

Deciding that you'd like a monolithic service or application to be smaller is a good start. But I would strongly advise you to chip away at these systems. An incremental approach will help you learn about microservices as you go, and will also limit the impact of getting something wrong (and you will get things wrong!). Think of our monolith as a block of marble. We could blow the whole thing up, but that rarely ends well. It makes much more sense to just chip away at it incrementally.

So if we are going to break apart the monolith a piece at a time, where should we start? We have our seams now, but which one should we pull out first? It's best to think about where you are going to get the most benefit from some part of your codebase being separated, rather than just splitting things for the sake of it. Let's consider some drivers that might help guide our chisel.

Pace of Change

Perhaps we know that we have a load of changes coming up soon in how we manage inventory. If we split out the warehouse seam as a service now, we could change that service faster, as it is a separate autonomous unit.

Team Structure

MusicCorp's delivery team is actually split across two geographical regions. One team is in London, the other in Hawaii (some people have it easy!). It would be great if we could split out the code that the Hawaii team works on the most, so it can take full ownership. We'll explore this idea further in Chapter 10.

Security

MusicCorp has had a security audit, and has decided to tighten up its protection of sensitive information. Currently, all of this is handled by the finance-related code. If

we split this service out, we can provide additional protections to this individual service in terms of monitoring, protection of data at transit, and protection of data at rest—ideas we'll look at in more detail in Chapter 9.

Technology

The team looking after our recommendation system has been spiking out some new algorithms using a logic programming library in the language Clojure. The team thinks this could benefit our customers by improving what we offer them. If we could split out the recommendation code into a separate service, it would be easy to consider building an alternative implementation that we could test against.

Tangled Dependencies

The other point to consider when you've identified a couple of seams to separate is how entangled that code is with the rest of the system. We want to pull out the seam that is least depended on if we can. If you can view the various seams you have found as a directed acyclical graph of dependencies (something the package modeling tools I mentioned earlier are very good at), this can help you spot the seams that are likely going to be harder to disentangle.

This brings us to what is often the mother of all tangled dependencies: the database.

The Database

We've already discussed at length the challenges of using databases as a method of integrating multiple services. As I made it pretty clear earlier, I am not a fan! This means we need to find seams in our databases too so we can split them out cleanly. Databases, however, are tricky beasts.

Getting to Grips with the Problem

The first step is to take a look at the code itself and see which parts of it read to and write from the database. A common practice is to have a repository layer, backed by some sort of framework like Hibernate, to bind your code to the database, making it easy to map objects or data structures to and from the database. If you have been following along so far, you'll have grouped our code into packages representing our bounded contexts; we want to do the same for our database access code. This may require splitting up the repository layer into several parts, as shown in Figure 5-1.

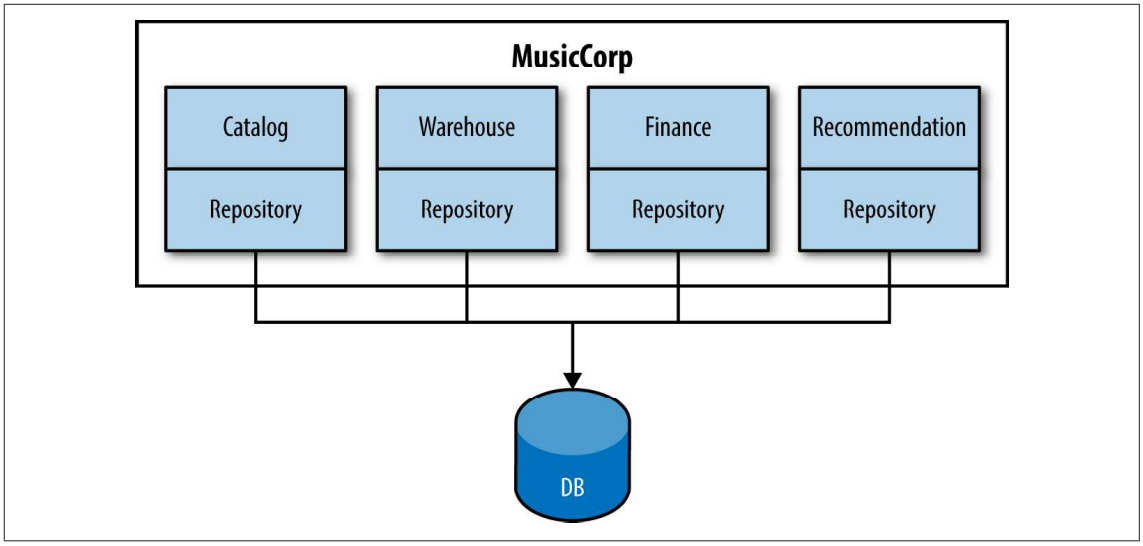

Figure 5-1. Splitting out our repository layers

Having the database mapping code colocated inside the code for a given context can help us understand what parts of the database are used by what bits of code. Hibernate, for example, can make this very clear if you are using something like a mapping file per bounded context.

This doesn't give us the whole story, however. For example, we may be able to tell that the finance code uses the ledger table, and that the catalog code uses the line item table, but it might not be clear that the database enforces a foreign key relationship from the ledger table to the line item table. To see these database-level constraints, which may be a stumbling block, we need to use another tool to visualize the data. A great place to start is to use a tool like the freely available SchemaSpy, which can generate graphical representations of the relationships between tables.

All this helps you understand the coupling between tables that may span what will eventually become service boundaries. But how do you cut those ties? And what about cases where the same tables are used from multiple different bounded contexts? Handling problems like these is not easy, and there are many answers, but it is doable.

Coming back to some concrete examples, let's consider our music shop again. We have identified four bounded contexts, and want to move forward with making them four distinct, collaborating services. We're going to look at a few concrete examples of problems we might face, and their potential solutions. And while some of these examples talk specifically about challenges encountered in standard relational databases, you will find similar problems in other alternative NOSQL stores.

Example: Breaking Foreign Key Relationships

In this example, our catalog code uses a generic line item table to store information about an album. Our finance code uses a ledger table to track financial transactions. At the end of each month we need to generate reports for various people in the organization so they can see how we're doing. We want to make the reports nice and easy to read, so rather than saying, "We sold 400 copies of SKU 12345 and made $1,300," we'd like to add more information about what was sold (i.e., "We sold 400 copies of Bruce Springsteen's *Greatest Hits* and made $1,300"). To do this, our reporting code in the finance package will reach into the line item table to pull out the title for the SKU. It may also have a foreign key constraint from the ledger to the line item table, as we see in Figure 5-2.

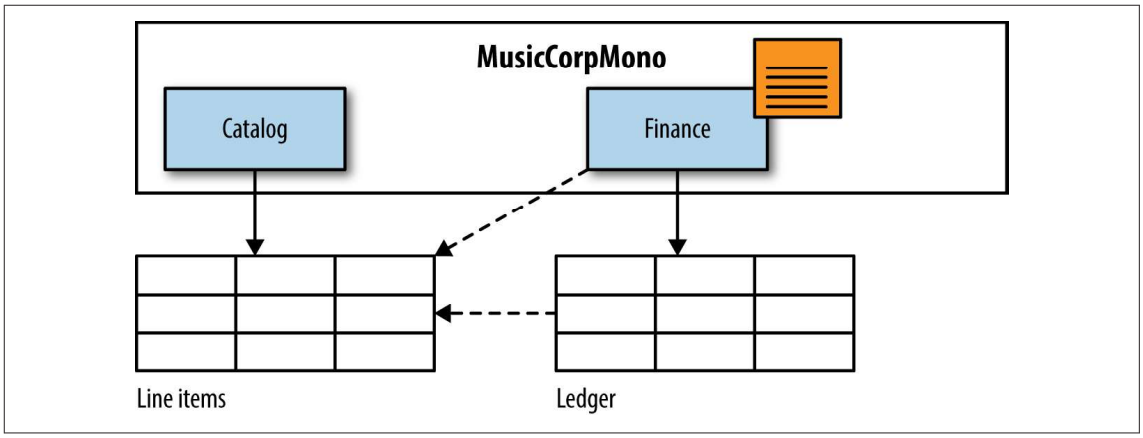

Figure 5-2. Foreign key relationship

So how do we fix things here? Well, we need to make a change in two places. First, we need to stop the finance code from reaching into the line item table, as this table really belongs to the catalog code, and we don't want database integration happening once catalog and finance are services in their own rights. The quickest way to address this is rather than having the code in finance reach into the line item table, we'll expose the data via an API call in the catalog package that the finance code can call. This API call will be the forerunner of a call we will make over the wire, as we see in Figure 5-3.

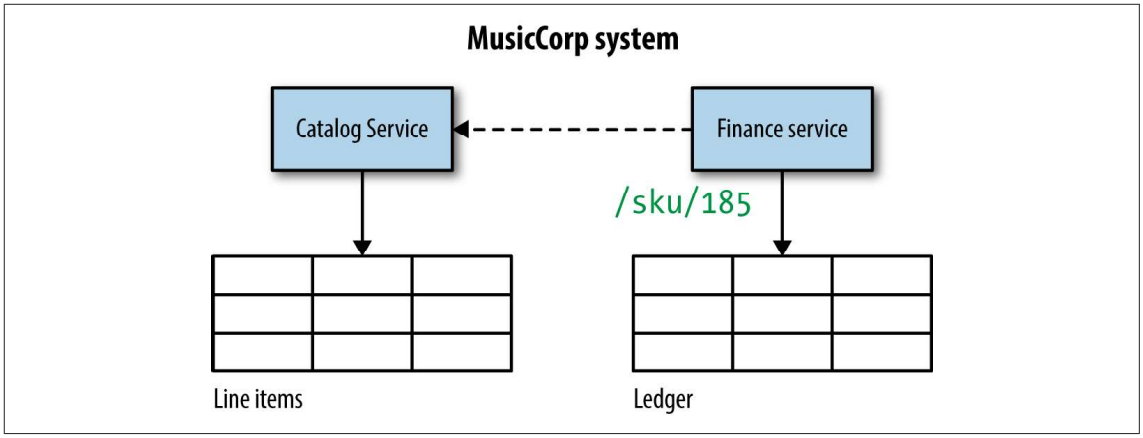

Figure 5-3. Post removal of the foreign key relationship

At this point it becomes clear that we may well end up having to make two database calls to generate the report. This is correct. And the same thing will happen if these are two separate services. Typically concerns around performance are now raised. I have a fairly easy answer to those: how fast does your system need to be? And how fast is it now? If you can test its current performance and know what good performance looks like, then you should feel confident in making a change. Sometimes making one thing slower in exchange for other things is the right thing to do, especially if *slower* is still perfectly acceptable.

But what about the foreign key relationship? Well, we lose this altogether. This becomes a constraint we need to now manage in our resulting services rather than in the database level. This may mean that we need to implement our own consistency check across services, or else trigger actions to clean up related data. Whether or not this is needed is often not a technologist's choice to make. For example, if our order service contains a list of IDs for catalog items, what happens if a catalog item is removed and an order now refers to an invalid catalog ID? Should we allow it? If we do, then how is this represented in the order when it is displayed? If we don't, then how can we check that this isn't violated? These are questions you'll need to get answered by the people who define how your system should behave for its users.

Example: Shared Static Data

I have seen perhaps as many country codes stored in databases (shown in Figure 5-4) as I have written `StringUtils` classes for in-house Java projects. This seems to imply that we plan to change the countries our system supports way more frequently than we'll deploy new code, but whatever the real reason, these examples of shared static data being stored in databases come up a lot. So what do we do in our music shop if all our potential services read from the same table like this?

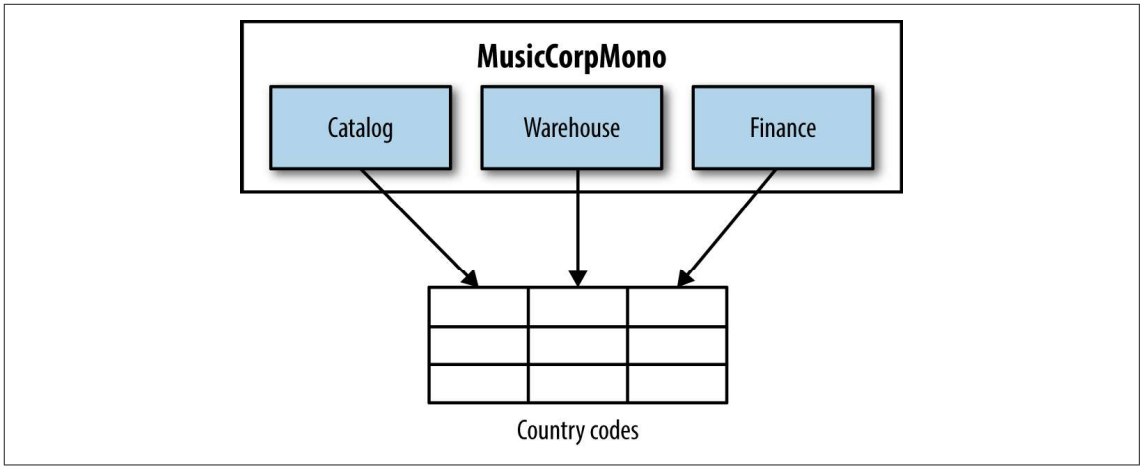

Figure 5-4. Country codes in the database

Well, we have a few options. One is to duplicate this table for each of our packages, with the long-term view that it will be duplicated within each service also. This leads to a potential consistency challenge, of course: what happens if I update one table to reflect the creation of Newmantopia off the east coast of Australia, but not another?

A second option is to instead treat this shared, static data as code. Perhaps it could be in a property file deployed as part of the service, or perhaps just as an enumeration. The problems around the consistency of data remain, although experience has shown that it is far easier to push out changes to configuration files than alter live database tables. This is often a very sensible approach.

A third option, which may well be extreme, is to push this static data into a service of its own right. In a couple of situations I have encountered, the volume, complexity, and rules associated with the static reference data were sufficient that this approach was warranted, but it's probably overkill if we are just talking about country codes!

Personally, in most situations I'd try to push for keeping this data in configuration files or directly in code, as it is the simple option for most cases.

Example: Shared Data

Now let's dive into a more complex example, but one that can be a common problem when you're trying to tease apart systems: shared mutable data. Our finance code tracks payments made by customers for their orders, and also tracks refunds given to them when they return items. Meanwhile, the warehouse code updates records to show that orders for customers have been dispatched or received. All of this data is displayed in one convenient place on the website so that customers can see what is going on with their account. To keep things simple, we have stored all this information in a fairly generic customer record table, as shown in Figure 5-5.

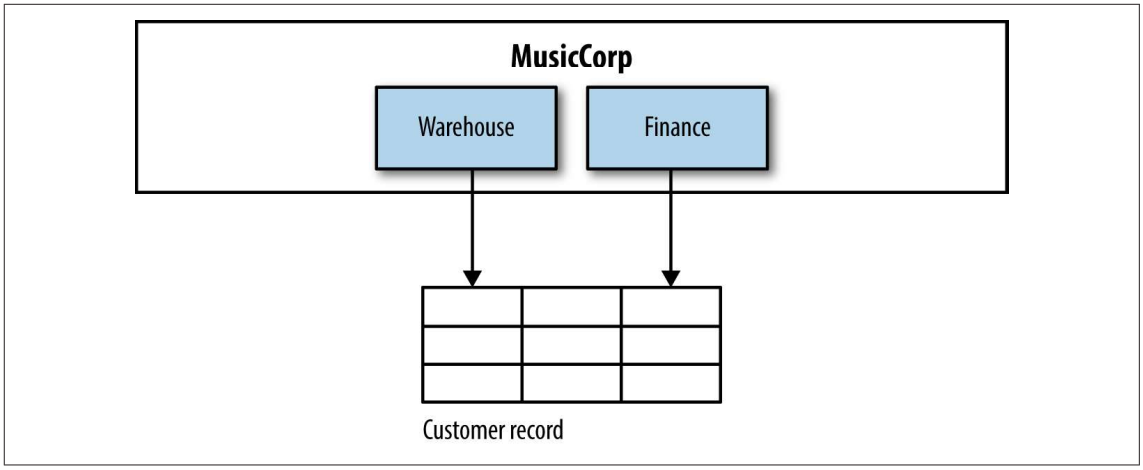

Figure 5-5. Accessing customer data: are we missing something?

So both the finance and the warehouse code are writing to, and probably occasionally reading from, the same table. How can we tease this apart? What we actually have here is something you'll see often—a domain concept that isn't modeled in the code, and is in fact implicitly modeled in the database. Here, the domain concept that is missing is that of `Customer`.

We need to make the current abstract concept of the customer concrete. As a transient step, we create a new package called `Customer`. We can then use an API to expose `Customer` code to other packages, such as finance or warehouse. Rolling this all the way forward, we may now end up with a distinct customer service (Figure 5-6).

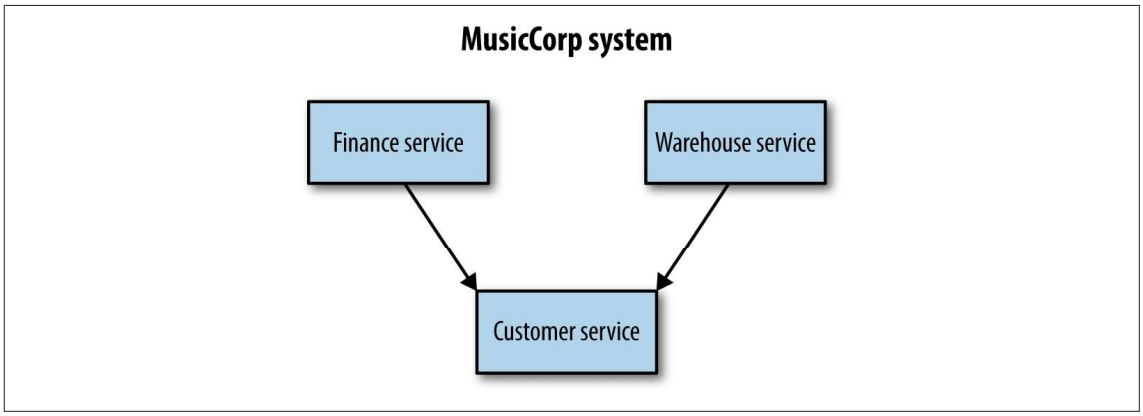

Figure 5-6. Recognizing the bounded context of the customer

Example: Shared Tables

Figure 5-7 shows our last example. Our catalog needs to store the name and price of the records we sell, and the warehouse needs to keep an electronic record of inventory. We decide to keep these two things in the same place in a generic line item table. Before, with all the code merged in together, it wasn't clear that we are actually

conflating concerns, but now we can see that in fact we have two separate concepts that could be stored differently.

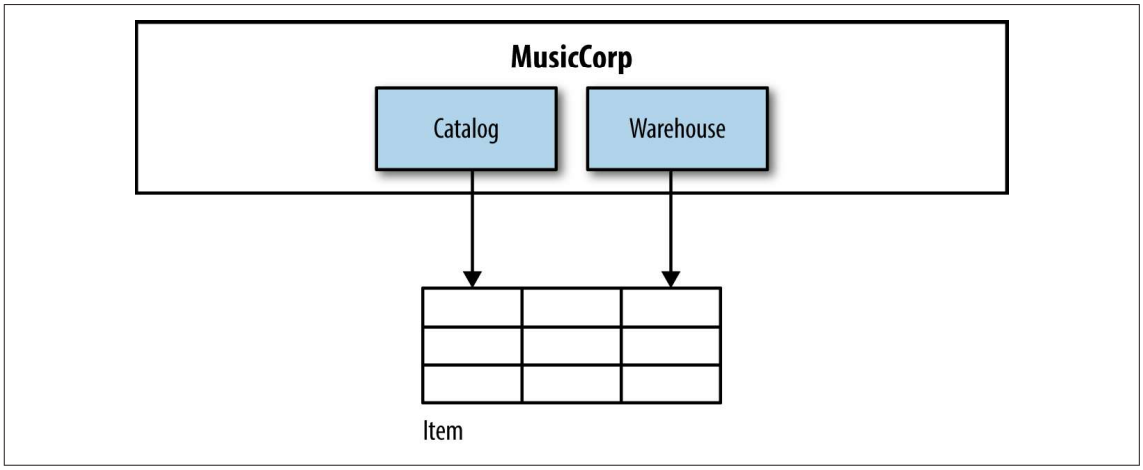

Figure 5-7. Tables being shared between different contexts

The answer here is to split the table in two as we have in Figure 5-8, perhaps creating a stock list table for the warehouse, and a catalog entry table for the catalog details.

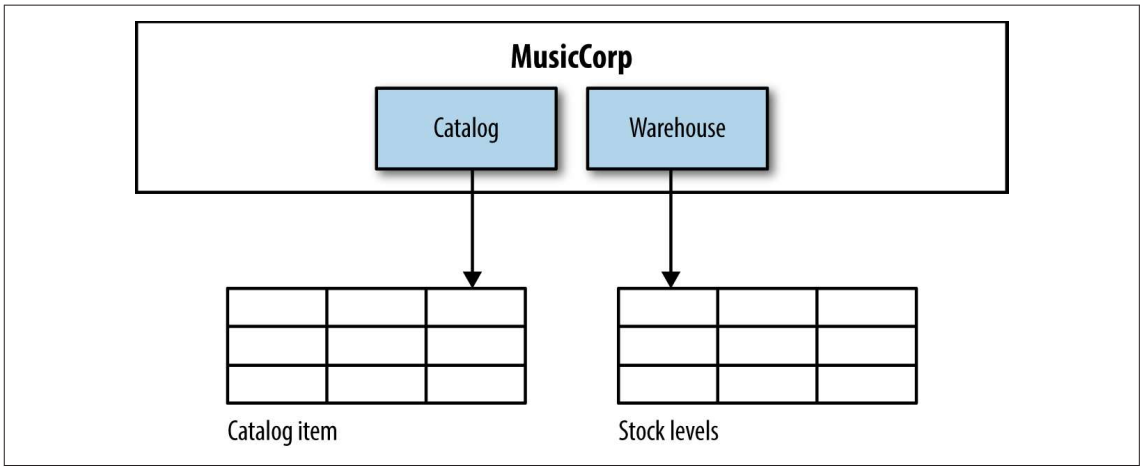

Figure 5-8. Pulling apart the shared table

Refactoring Databases

What we have covered in the preceding examples are a few database refactorings that can help you separate your schemas. For a more detailed discussion of the subject, you may want to take a look at *Refactoring Databases* by Scott J. Ambler and Pramod J. Sadalage (Addison-Wesley).

Staging the Break

So we've found seams in our application code, grouping it around bounded contexts. We've used this to identify seams in the database, and we've done our best to split those out. What next? Do you do a big-bang release, going from one monolithic service with a single schema to two services, each with its own schema? I would actually recommend that you split out the schema but keep the service together before splitting the application code out into separate microservices, as shown in Figure 5-9.

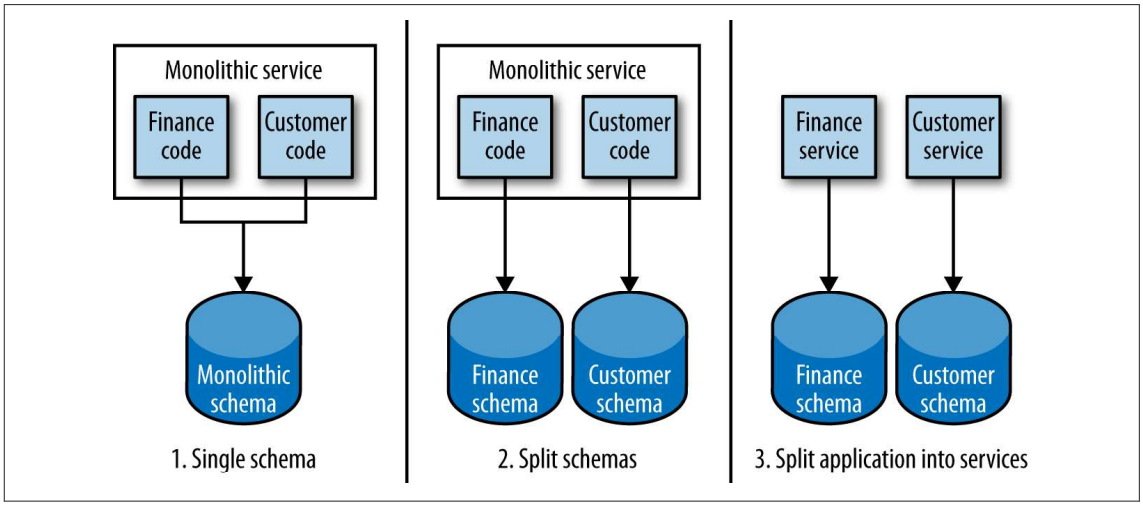

Figure 5-9. Staging a service separation

With a separate schema, we'll be potentially increasing the number of database calls to perform a single action. Where before we might have been able to have all the data we wanted in a single `SELECT` statement, now we may need to pull the data back from two locations and join in memory. Also, we end up breaking transactional integrity when we move to two schemas, which could have significant impact on our applications; we'll be discussing this next. By splitting the schemas out but keeping the application code together, we give ourselves the ability to revert our changes or continue to tweak things without impacting any consumers of our service. Once we are satisfied that the DB separation makes sense, we can then think about splitting out the application code into two services.

Transactional Boundaries

Transactions are useful things. They allow us to say *these events either all happen together, or none of them happen*. They are very useful when we're inserting data into a database; they let us update multiple tables at once, knowing that if anything fails, everything gets rolled back, ensuring our data doesn't get into an inconsistent state. Simply put, a transaction allows us to group together multiple different activities that

take our system from one consistent state to another—everything works, or nothing changes.

Transactions don't just apply to databases, although we most often use them in that context. Message brokers, for example, have long allowed you to post and receive messages within transactions too.

With a monolithic schema, all our create or updates will probably be done within a single transactional boundary. When we split apart our databases, we lose the safety afforded to us by having a single transaction. Consider a simple example in the MusicCorp context. When creating an order, I want to update the order table stating that a customer order has been created, and also put an entry into a table for the warehouse team so it knows there is an order that needs to be picked for dispatch. We've gotten as far as grouping our application code into separate packages, and have also separated the customer and warehouse parts of the schema well enough that we are ready to put them into their own schemas prior to separating the application code.

Within a single transaction in our existing monolithic schema, creating the order and inserting the record for the warehouse team takes place within a single transaction, as shown in Figure 5-10.

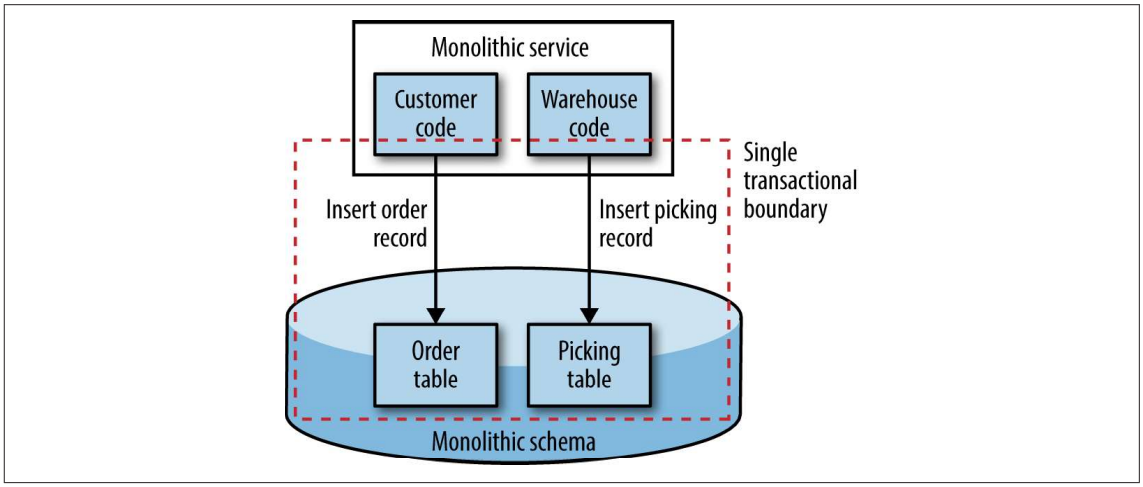

Figure 5-10. Updating two tables in a single transaction

But if we have pulled apart the schema into two separate schemas, one for customer-related data including our order table, and another for the warehouse, we have lost this transactional safety. The order placing process now spans two separate transactional boundaries, as we see in Figure 5-11. If our insert into the order table fails, we can clearly stop everything, leaving us in a consistent state. But what happens when the insert into the order table works, but the insert into the picking table fails?

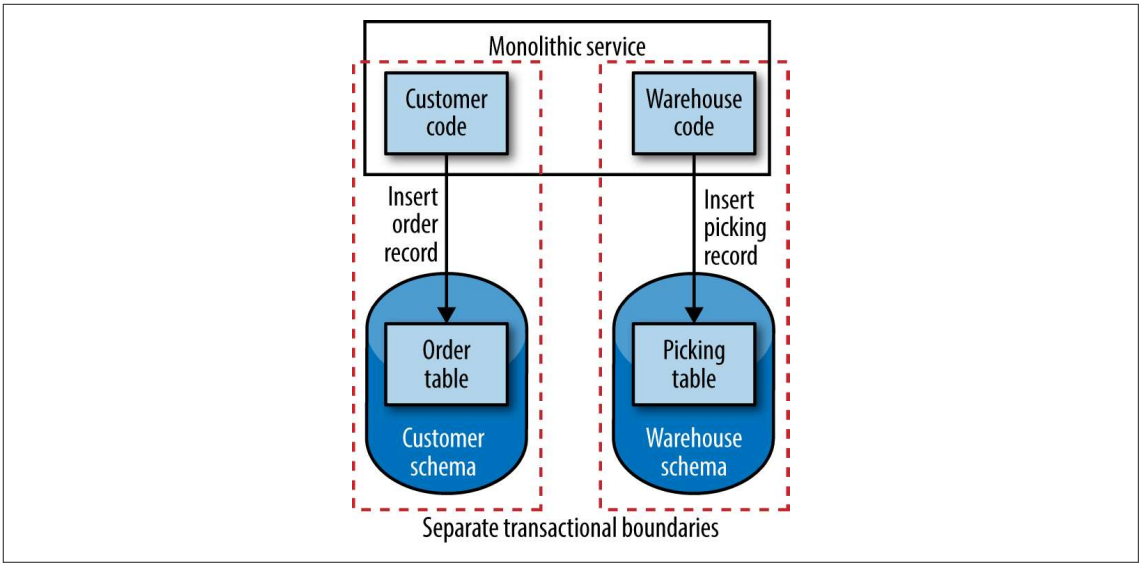

Figure 5-11. Spanning transactional boundaries for a single operation

Try Again Later

The fact that the order was captured and placed might be enough for us, and we may decide to retry the insertion into the warehouse's picking table at a later date. We could queue up this part of the operation in a queue or logfile, and try again later. For some sorts of operations this makes sense, but we have to assume that a retry would fix it.

In many ways, this is another form of what is called *eventual consistency*. Rather than using a transactional boundary to ensure that the system is in a consistent state when the transaction completes, instead we accept that the system will get itself into a consistent state at some point in the future. This approach is especially useful with business operations that might be long-lived. We'll discuss this idea in more depth in Chapter 11 when we cover scaling patterns.

Abort the Entire Operation

Another option is to reject the entire operation. In this case, we have to put the system back into a consistent state. The picking table is easy, as that insert failed, but we have a committed transaction in the order table. We need to unwind this. What we have to do is issue a *compensating transaction*, kicking off a new transaction to wind back what just happened. For us, that could be something as simple as issuing a `DELETE` statement to remove the order from the database. Then we'd also need to report back via the UI that the operation failed. Our application could handle both aspects within a monolithic system, but we'd have to consider what we could do when we split up the application code. Does the logic to handle the compensating transaction live in the customer service, the order service, or somewhere else?

But what happens if our compensating transaction fails? It's certainly possible. Then we'd have an order in the order table with no matching pick instruction. In this situation, you'd either need to retry the compensating transaction, or allow some backend process to clean up the inconsistency later on. This could be something as simple as a maintenance screen that admin staff had access to, or an automated process.

Now think about what happens if we have not one or two operations we want to be consistent, but three, four, or five. Handling compensating transactions for each failure mode becomes quite challenging to comprehend, let alone implement.

Distributed Transactions

An alternative to manually orchestrating compensating transactions is to use a *distributed transaction*. Distributed transactions try to span multiple transactions within them, using some overall governing process called a *transaction manager* to orchestrate the various transactions being done by underlying systems. Just as with a normal transaction, a distributed transaction tries to ensure that everything remains in a consistent state, only in this case it tries to do so across multiple different systems running in different processes, often communicating across network boundaries.

The most common algorithm for handling distributed transactions—especially short-lived transactions, as in the case of handling our customer order—is to use a *two-phase commit*. With a two-phase commit, first comes the voting phase. This is where each participant (also called a *cohort* in this context) in the distributed transaction tells the transaction manager whether it thinks its local transaction can go ahead. If the transaction manager gets a *yes* vote from all participants, then it tells them all to go ahead and perform their commits. A single *no* vote is enough for the transaction manager to send out a rollback to all parties.

This approach relies on all parties halting until the central coordinating process tells them to proceed. This means we are vulnerable to outages. If the transaction manager goes down, the pending transactions never complete. If a cohort fails to respond during voting, everything blocks. And there is also the case of what happens if a commit fails after voting. There is an assumption implicit in this algorithm that this cannot happen: if a cohort says *yes* during the voting period, then we have to assume it *will* commit. Cohorts need a way of making this commit work at some point. This means this algorithm isn't foolproof—rather, it just tries to catch most failure cases.

This coordination process also mean locks; that is, pending transactions can hold locks on resources. Locks on resources can lead to contention, making scaling systems much more difficult, especially in the context of distributed systems.

Distributed transactions have been implemented for specific technology stacks, such as Java's Transaction API, allowing for disparate resources like a database and a message queue to all participate in the same, overarching transaction. The various algo-

rithms are hard to get right, so I'd suggest you avoid trying to create your own. Instead, do lots of research on this topic if this seems like the route you want to take, and see if you can use an existing implementation.

So What to Do?

All of these solutions add complexity. As you can see, distributed transactions are hard to get right and can actually inhibit scaling. Systems that eventually converge through compensating retry logic can be harder to reason about, and may need other compensating behavior to fix up inconsistencies in data.

When you encounter business operations that currently occur within a single transaction, ask yourself if they really need to. Can they happen in different, local transactions, and rely on the concept of eventual consistency? These systems are much easier to build and scale (we'll discuss this more in Chapter 11).

If you do encounter state that really, really wants to be kept consistent, do everything you can to avoid splitting it up in the first place. Try *really* hard. If you really need to go ahead with the split, think about moving from a purely technical view of the process (e.g., a database transaction) and actually create a concrete concept to represent the transaction itself. This gives you a handle, or a hook, on which to run other operations like compensating transactions, and a way to monitor and manage these more complex concepts in your system. For example, you might create the idea of an "in-process-order" that gives you a natural place to focus all logic around processing the order end to end (and dealing with exceptions).

Reporting

As we've already seen, in splitting a service into smaller parts, we need to also potentially split up how and where data is stored. This creates a problem, however, when it comes to one vital and common use case: reporting.

A change in architecture as fundamental as moving to a microservices architecture will cause a lot of disruption, but it doesn't mean we have to abandon everything we do. The audience of our reporting systems are users like any other, and we need to consider their needs. It would be arrogant to fundamentally change our architecture and just ask them to adapt. While I'm not suggesting that the space of reporting isn't ripe for disruption—it certainly is—there is value in determining how to work with existing processes first. Sometimes we have to pick our battles.

The Reporting Database

Reporting typically needs to group together data from across multiple parts of our organization in order to generate useful output. For example, we might want to

enrich the data from our general ledger with descriptions of what was sold, which we get from a catalog. Or we might want to look at the shopping behavior of specific, high-value customers, which could require information from their purchase history and their customer profile.

In a standard, monolithic service architecture, all our data is stored in one big database. This means all the data is in one place, so reporting across all the information is actually pretty easy, as we can simply join across the data via SQL queries or the like. Typically we won't run these reports on the main database for fear of the load generated by our queries impacting the performance of the main system, so often these reporting systems hang on a read replica as shown in Figure 5-12.

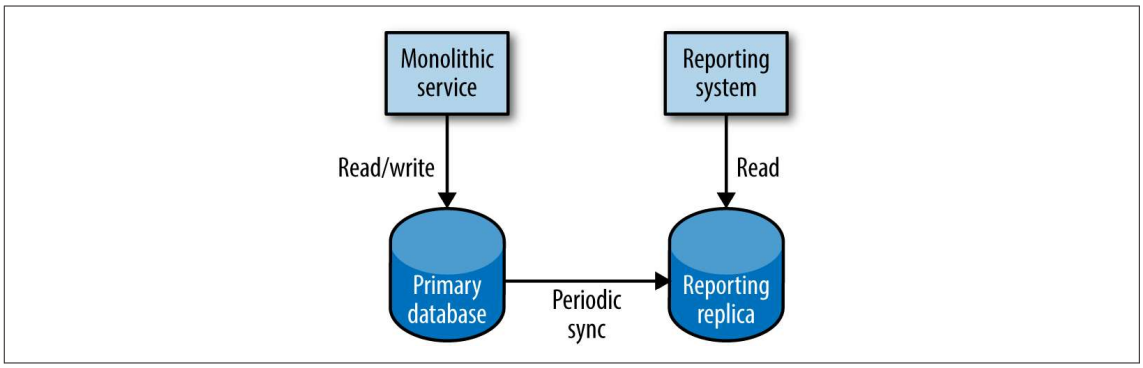

Figure 5-12. Standard read replication

With this approach we have one sizeable upside—that all the data is already in one place, so we can use fairly straightforward tools to query it. But there are also a couple of downsides with this approach. First, the schema of the database is now effectively a shared API between the running monolithic services and any reporting system. So a change in schema has to be carefully managed. In reality, this is another impediment that reduces the chances of anyone wanting to take on the task of making and coordinating such a change.

Second, we have limited options as to how the database can be optimized for either use case—backing the live system or the reporting system. Some databases let us make optimizations on read replicas to enable faster, more efficient reporting; for example, MySQL would allow us to run a different backend that doesn't have the overhead of managing transactions. However, we cannot structure the data differently to make reporting faster if that change in data structure has a bad impact on the running system. What often happens is that the schema either ends up being great for one use case and lousy for the other, or else becomes the lowest common denominator, great for neither purpose.

Finally, the database options available to us have exploded recently. While standard relational databases expose SQL query interfaces that work with many reporting tools, they aren't always the best option for storing data for our running services.

What if our application data is better modeled as a graph, as in Neo4j? Or what if we'd rather use a document store like MongoDB? Likewise, what if we wanted to explore using a column-oriented database like Cassandra for our reporting system, which makes it much easier to scale for larger volumes? Being constrained in having to have one database for both purposes results in us often not being able to make these choices and explore new options.

So it's not perfect, but it works (mostly). Now if our information is stored in multiple different systems, what do we do? Is there a way for us to bring all the data together to run our reports? And could we also potentially find a way to eliminate some of the downsides associated with the standard reporting database model?

It turns out we have a number of viable alternatives to this approach. Which solution makes the most sense to you will depend on a number of factors, but we'll explore a few different options that I have seen in practice.

Data Retrieval via Service Calls

There are many variants of this model, but they all rely on pulling the required data from the source systems via API calls. For a very simple reporting system, like a dashboard that might just want to show the number of orders placed in the last 15 minutes, this might be fine. To report across data from two or more systems, you need to make multiple calls to assemble this data.

This approach breaks down rapidly with use cases that require larger volumes of data, however. Imagine a use case where we want to report on customer purchasing behavior for our music shop over the last 24 months, looking at various trends in customer behavior and how this has impacted on revenue. We need to pull large volumes of data from at least the customer and finance systems. Keeping a local copy of this data in the reporting system is dangerous, as we may not know if it has changed (even historic data may be changed after the fact), so to generate an accurate report we need all of the finance and customer records for the last two years. With even modest numbers of customers, you can see that this quickly will become a very slow operation.

Reporting systems also often rely on third-party tools that expect to retrieve data in a certain way, and here providing a SQL interface is the fastest way to ensure your reporting tool chain is as easy to integrate with as possible. We could still use this approach to pull data periodically into a SQL database, of course, but this still presents us with some challenges.

One of the key challenges is that the APIs exposed by the various microservices may well not be designed for reporting use cases. For example, a customer service may allow us to find a customer by an ID, or search for a customer by various fields, but wouldn't necessarily expose an API to retrieve all customers. This could lead to many calls being made to retrieve all the data—for example, having to iterate through a list

of all the customers, making a separate call for each one. Not only could this be inefficient for the reporting system, it could generate load for the service in question too.

While we could speed up some of the data retrieval by adding cache headers to the resources exposed by our service, and have this data cached in something like a reverse proxy, the nature of reporting is often that we access the *long tail* of data. This means that we may well request resources that no one else has requested before (or at least not for a sufficiently long time), resulting in a potentially expensive *cache miss*.

You could resolve this by exposing batch APIs to make reporting easier. For example, our customer service could allow you to pass a list of customer IDs to it to retrieve them in batches, or may even expose an interface that lets you page through all the customers. A more extreme version of this is to model the batch request as a resource in its own right. For example, the customer service might expose something like a `BatchCustomerExport` resource endpoint. The calling system would POST a `BatchRequest`, perhaps passing in a location where a file can be placed with all the data. The customer service would return an HTTP 202 response code, indicating that the request was accepted but has not yet been processed. The calling system could then poll the resource waiting until it retrieves a 201 Created status, indicating that the request has been fulfilled, and then the calling system could go and fetch the data. This would allow potentially large data files to be exported without the overhead of being sent over HTTP; instead, the system could simply save a CSV file to a shared location.

I have seen the preceding approach used for batch insertion of data, where it worked well. I am less in favor of it for reporting systems, however, as I feel that there are other, potentially simpler solutions that can scale more effectively when you're dealing with traditional reporting needs.

Data Pumps

Rather than have the reporting system pull the data, we could instead have the data pushed to the reporting system. One of the downsides of retrieving the data by standard HTTP calls is the overhead of HTTP when we're making a large number of calls, together with the overhead of having to create APIs that may exist only for reporting purposes. An alternative option is to have a standalone program that directly accesses the database of the service that is the source of data, and pumps it into a reporting database, as shown in Figure 5-13.

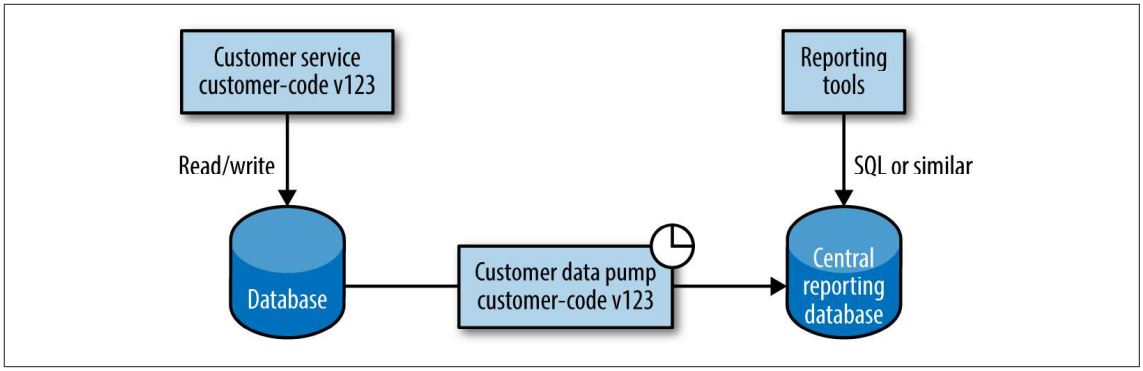

Figure 5-13. Using a data pump to periodically push data to a central reporting database

At this point you'll be saying, "But Sam, you said having lots of programs integrating on the same database is a bad idea!" At least I *hope* you'll be saying that, given how firmly I made the point earlier! This approach, if implemented properly, is a notable exception, where the downsides of the coupling are more than mitigated by making the reporting easier.

To start with, the data pump should be built and managed by the same team that manages the service. This can be something as simple as a command-line program triggered via `Cron`. This program needs to have intimate knowledge of both the internal database for the service, and also the reporting schema. The pump's job is to map one from the other. We try to reduce the problems with coupling to the service's schema by having the same team that manages the service also manage the pump. I would suggest, in fact, that you version-control these together, and have builds of the data pump created as an additional artifact as part of the build of the service itself, with the assumption that whenever you deploy one of them, you deploy them both. As we explicitly state that we deploy these together, and don't open up access to the schema to anyone outside of the service team, many of the traditional DB integration challenges are largely mitigated.

The coupling on the reporting schema itself remains, but we have to treat it as a published API that is hard to change. Some databases give us techniques where we could further mitigate this cost. Figure 5-14 shows an example of this for relational databases, where we could have one schema in the reporting database for each service, using things like materialized views to create the aggregated view. That way, we expose only the reporting schema for the customer data to the customer data pump. Whether this is something that you can do in a performant manner, however, will depend on the capabilities of the database you picked for reporting.

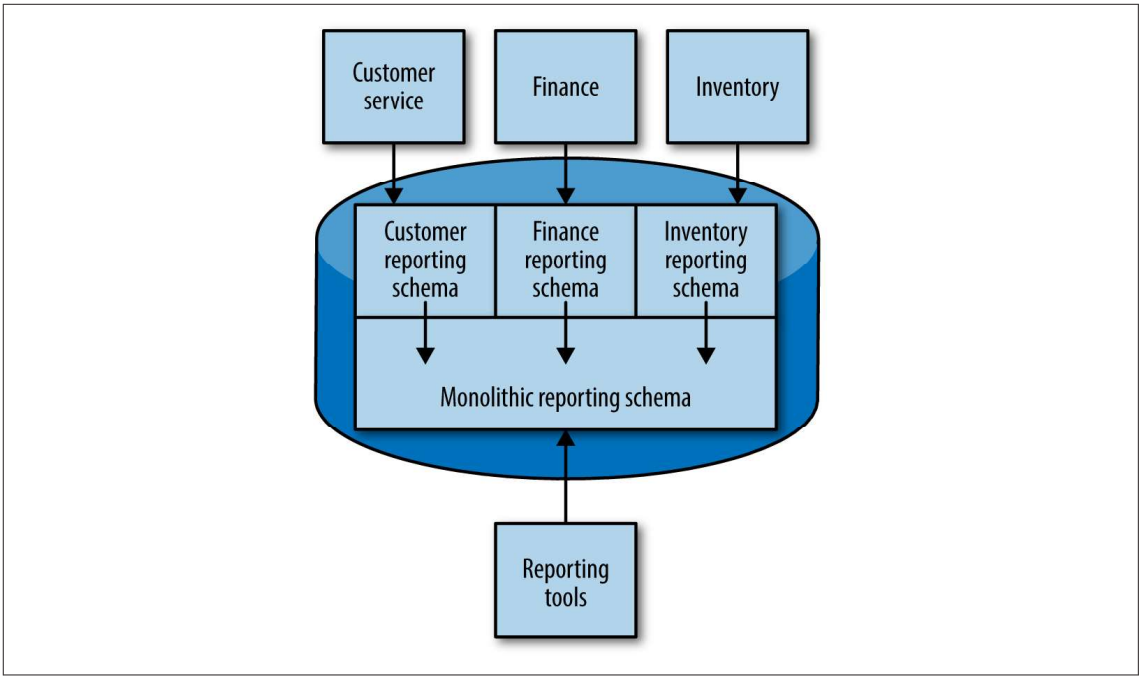

Figure 5-14. Utilizing materialized views to form a single monolithic reporting schema

Here, of course, the complexity of integration is pushed deeper into the schema, and will rely on capabilities in the database to make such a setup performant. While I think data pumps in general are a sensible and workable suggestion, I am less convinced that the complexity of a segmented schema is worthwhile, especially given the challenges in managing change in the database.

Alternative Destinations

On one project I was involved with, we used a series of data pumps to populate JSON files in AWS S3, effectively using S3 to masquerade as a giant data mart! This approach worked very well until we needed to scale our solution, and at the time of writing we are looking to change these pumps to instead populate a cube that can be integrated with standard reporting tools like Excel and Tableau.

Event Data Pump

In Chapter 4, we touched on the idea of microservices emitting events based on the state change of entities that they manage. For example, our customer service may emit an event when a given customer is created, or updated, or deleted. For those microservices that expose such event feeds, we have the option of writing our own event subscriber that pumps data into the reporting database, as shown in Figure 5-15.

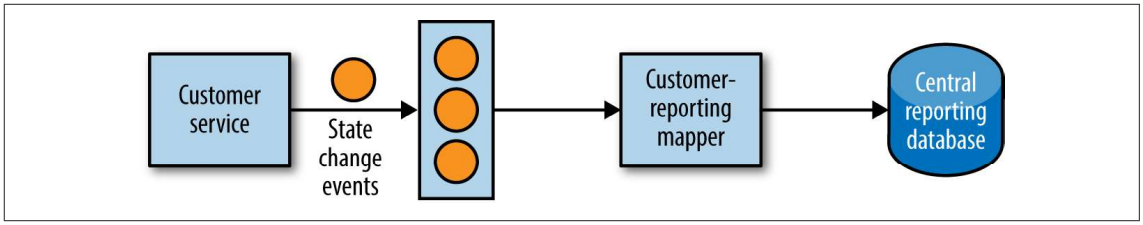

Figure 5-15. An event data pump using state change events to populate a reporting database

The coupling on the underlying database of the source microservice is now avoided. Instead, we are just binding to the events emitted by the service, which are designed to be exposed to external consumers. Given that events are temporal in nature, it also makes it easier for us to be smarter in what data we sent to our central reporting store; we can send data to the reporting system as we see an event, allowing data to flow faster to our reporting system, rather than relying on a regular schedule as with the data pump.

Also, if we store which events have already been processed, we can just process the new events as they arrive, assuming the old events have already been mapped into the reporting system. This means our insertion will be more efficient, as we only need to send deltas. We can do similar things with a data pump, but we have to manage this ourselves, whereas the fundamentally temporal nature of the stream of events (*x* happens at timestamp *y*) helps us greatly.

As our event data pump is less coupled to the internals of the service, it is also easier to consider this being managed by a separate group from the team looking after the microservice itself. As long as the nature of our event stream doesn't overly couple subscribers to changes in the service, this event mapper can evolve independently of the service it subscribes to.

The main downsides to this approach are that all the required information must be broadcast as events, and it may not scale as well as a data pump for larger volumes of data that has the benefit of operating directly at the database level. Nonetheless, the looser coupling and fresher data available via such an approach makes it strongly worth considering if you are already exposing the appropriate events.

Backup Data Pump

This option is based on an approach used at Netflix, which takes advantage of existing backup solutions and also resolves some scale issues that Netflix has to deal with. In some ways, you can consider this a special case of a data pump, but it seemed like such an interesting solution that it deserves inclusion.

Netflix has decided to standardize on Cassandra as the backing store for its services, of which there are many. Netflix has invested significant time in building tools to

make Cassandra easy to work with, much of which the company has shared with the rest of the world via numerous open source projects. Obviously it is very important that the data Netflix stores is properly backed up. To back up Cassandra data, the standard approach is to make a copy of the data files that back it and store them somewhere safe. Netflix stores these files, known as SSTables, in Amazon's S3 object store, which provides significant data durability guarantees.

Netflix needs to report across all this data, but given the scale involved this is a non-trivial challenge. Its approach is to use Hadoop that uses SSTable backup as the source of its jobs. In the end, Netflix ended up implementing a pipeline capable of processing large amounts of data using this approach, which it then open sourced as the Aegisthus project. Like data pumps, though, with this pattern we still have a coupling to the destination reporting schema (or target system).

It is conceivable that using a similar approach—that is, using mappers that work off backups—would work in other contexts as well. And if you're already using Cassandra, Netflix has already done much of the work for you!

Toward Real Time

Many of the patterns previously outlined are different ways of getting a lot of data from many different places to one place. But does the idea that all our reporting will be done from one location really stack up anymore? We have dashboards, alerting, financial reports, user analytics—all of these use cases have different tolerances for accuracy and timeliness, which may result in different technical options coming to bear. As I will detail in Chapter 8, we are moving more and more toward generic eventing systems capable of routing our data to multiple different places depending on need.

Cost of Change

There are many reasons why, throughout the book, I promote the need to make small, incremental changes, but one of the key drivers is to understand the impact of each alteration we make and change course if required. This allows us to better mitigate the cost of mistakes, but doesn't remove the chance of mistakes entirely. We can—and will—make mistakes, and we should embrace that. What we should also do, though, is understand how best to mitigate the costs of those mistakes.

As we have seen, the cost involved in moving code around within a codebase is pretty small. We have lots of tools that support us, and if we cause a problem, the fix is generally quick. Splitting apart a database, however, is much more work, and rolling back a database change is just as complex. Likewise, untangling an overly coupled integration between services, or having to completely rewrite an API that is used by multiple consumers, can be a sizeable undertaking. The large cost of change means that these

operations are increasingly risky. How can we manage this risk? My approach is to try to make mistakes where the impact will be lowest.

I tend to do much of my thinking in the place where the cost of change and the cost of mistakes is as low as it can be: the whiteboard. Sketch out your proposed design. See what happens when you run use cases across what you think your service boundaries will be. For our music shop, for example, imagine what happens when a customer searches for a record, registers with the website, or purchases an album. What calls get made? Do you start seeing odd circular references? Do you see two services that are overly chatty, which might indicate they should be one thing?

A great technique here is to adapt an approach more typically taught for the design of object-oriented systems: class-responsibility-collaboration (CRC) cards. With CRC cards, you write on one index card the name of the class, what its responsibilities are, and who it collaborates with. When working through a proposed design, for each service I list its responsibilities in terms of the capabilities it provides, with the collaborators specified in the diagram. As you work through more use cases, you start to get a sense as to whether all of this hangs together properly.

Understanding Root Causes

We have discussed how to split apart larger services into smaller ones, but why did these services grow so large in the first place? The first thing to understand is that growing a service to the point that it needs to be split is completely OK. We *want* the architecture of our system to change over time in an incremental fashion. The key is knowing it needs to be split before the split becomes too expensive.

But in practice many of us will have seen services grow well beyond the point of sanity. Despite knowing that a smaller set of services would be easier to deal with than the huge monstrosity we currently have, we still plow on with growing the beast. Why?

Part of the problem is knowing where to start, and I'm hoping this chapter has helped. But another challenge is the cost associated with splitting out services. Finding somewhere to run the service, spinning up a new service stack, and so on, are nontrivial tasks. So how do we address this? Well, if doing something is right but difficult, we should strive to make things easier. Investment in libraries and lightweight service frameworks can reduce the cost associated with creating the new service. Giving people access to self-service provision virtual machines or even making a platform as a service (PaaS) available will make it easier to provision systems and test them. Throughout the rest of the book, we'll be discussing a number of ways to help you keep this cost down.

Summary

We decompose our system by finding seams along which service boundaries can emerge, and this can be an incremental approach. By getting good at finding these seams and working to reduce the cost of splitting out services in the first place, we can continue to grow and evolve our systems to meet whatever requirements come down the road. As you can see, some of this work can be painstaking. But the very fact that it can be done incrementally means there is no need to fear this work.

So now we can split our services out, but we've introduced some new problems too. We have many more moving parts to get into production now! So next up we'll dive into the world of deployment.