

微服务架构之服务拆分

Wen Zhenglin

2016-3-28



概要

- 案例
- 拆分问题和相关考虑
- 一些解决办法

Breaking Apart MusicCorp

- 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 Reasons to Split the Monolith

- Pace of Change
 - If we split out the warehouse team as a service now, we could change that
 - service faster, as it is a separate autonomous unit
- Team Structure
 - take full ownership
- Security
 - 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
- Technology
 - 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

Other reasons

the mother of all tangled dependencies: the database

databases as a method of integrating multiple services

Databases, however, are tricky beasts

Splitting up the repository layer into several parts

1. look at the code itself and see which parts of it read from and write to the database
2. grouped our code into packages representing our bounded contexts, do the same for our database access code
3. 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

MusicCorp

Catalog

Warehouse

Finance

Recommendation

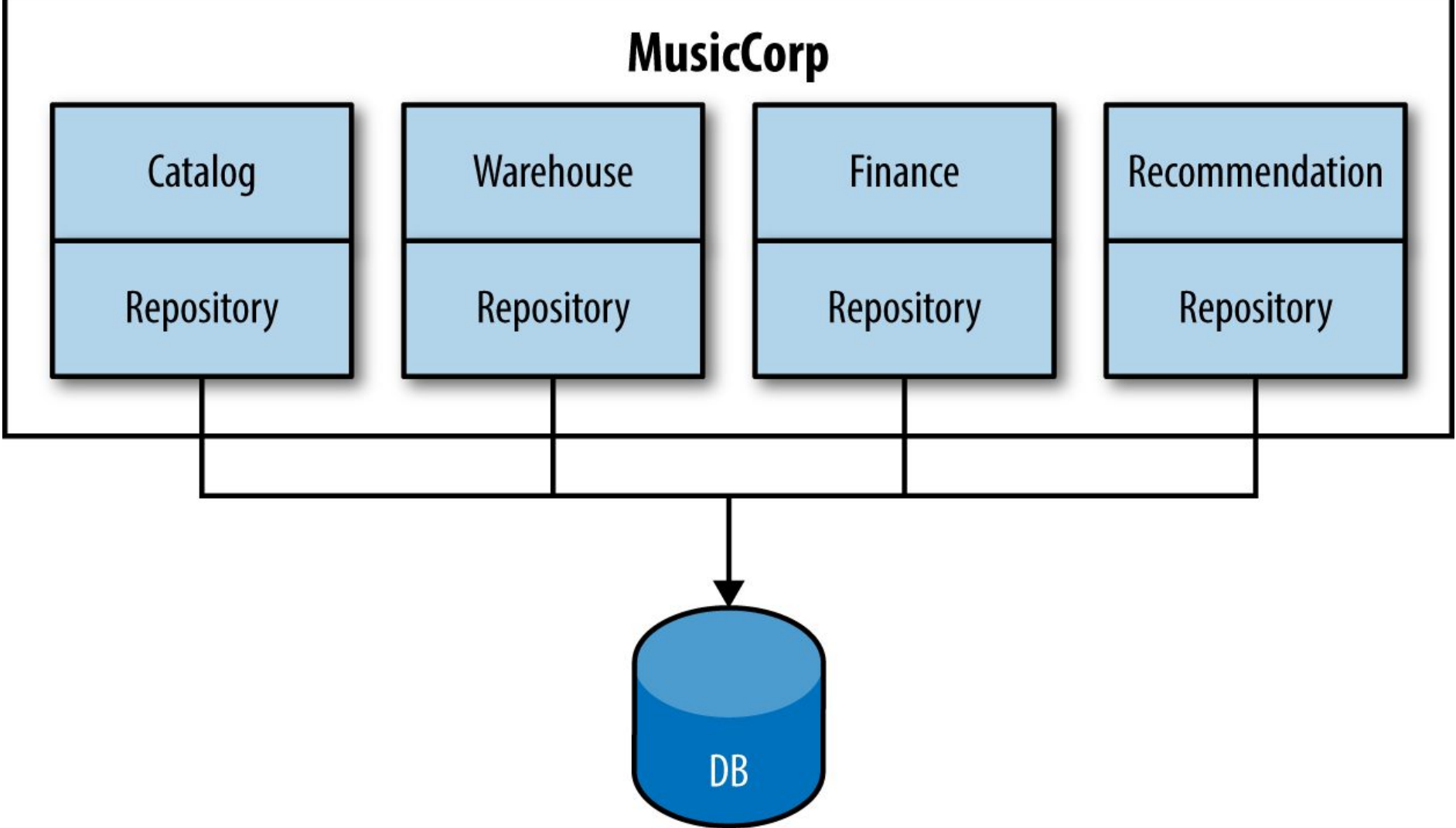
Repository

Repository

Repository

Repository

DB



Concerns

it might not be clear that the database enforces a foreign key relationship

To see these database-level constraints, which may be a stumbling block, we need to use another tool to visualize the data (SchemaSpy ? 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.

Example: Breaking Foreign Key Relationships

catalog code uses a generic line item table to store information about an album

finance code uses a ledger(总账) table to track financial transactions

reporting code in the finance package will reach into the line item table to pull out the title for the SKU(库存单位)

MusicCorpMono

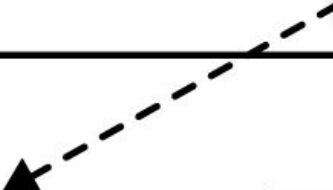
Catalog

Finance



Line items

Ledger

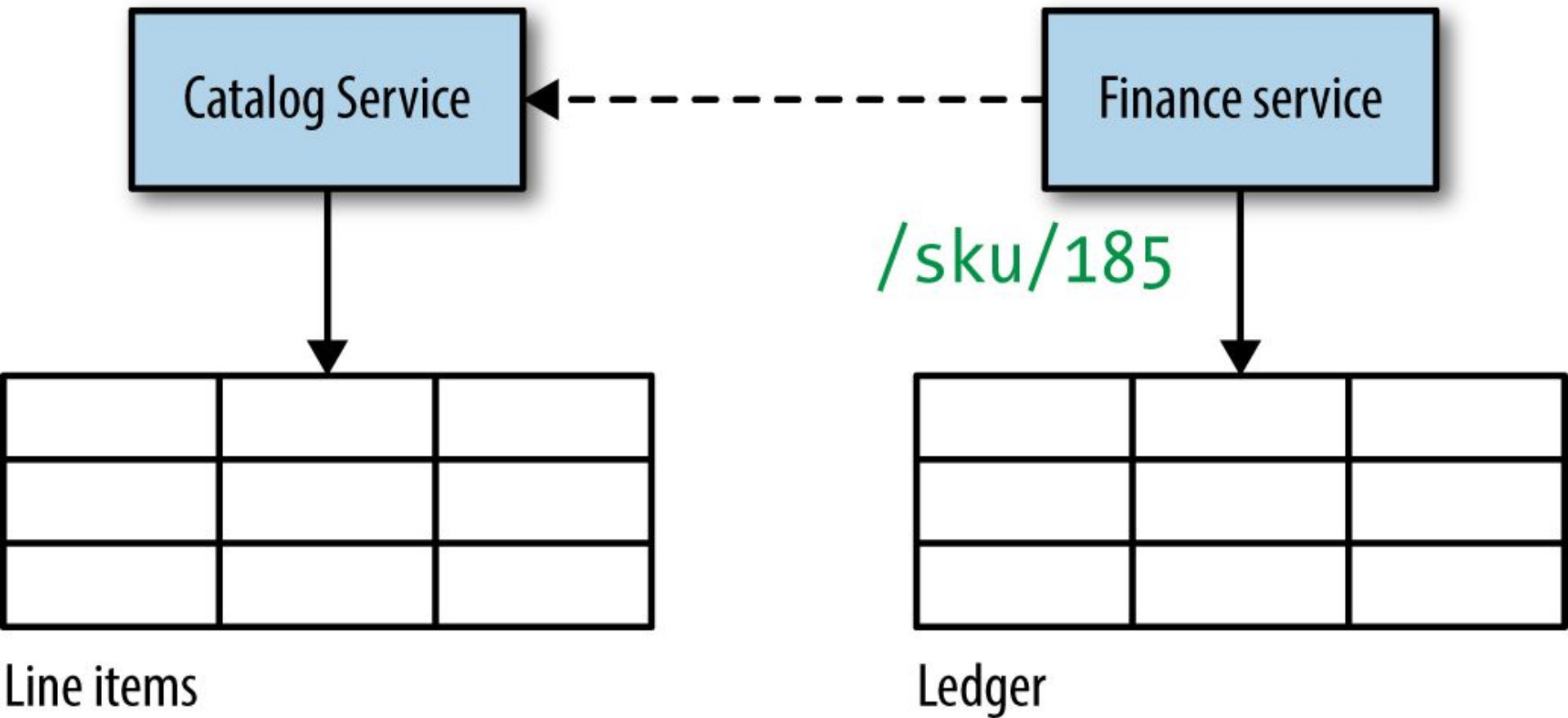


So how do we fix things here

First, we need to stop the finance code from reaching into the line item table, as this table really belongs to the catalog code

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.

MusicCorp system



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

Often not a technologist's choice to make

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, how is this represented in the order when it is displayed?

If we don't, 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

many country codes stored in databases, have written StringUtils classes for in-house Java projects

Imply that plan to change the countries our system supports way more frequently than we'll deploy new code

whatever the real reason, shared static data being stored in databases come up a lot

what do we do in our music shop if all our potential services read from the same table like this?

MusicCorpMono

Catalog

Warehouse

Finance

The diagram illustrates a data flow from three business units to a shared data structure. Three light blue boxes labeled 'Catalog', 'Warehouse', and 'Finance' are positioned at the top. Arrows from each box point to a 3x3 grid of white cells. The grid is labeled 'Country codes' at the bottom.

Country codes

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

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

A third option, which may well be extreme, is to push this static data into a service of its own right, it's probably overkill if we are just talking about country codes

Example: Shared Data

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

To keep things simple, we have stored all this information in a fairly generic customer record table

MusicCorp

Warehouse

Finance

The diagram illustrates a data flow from two departments, Warehouse and Finance, to a shared Customer record table. Warehouse and Finance are represented as light blue boxes with black outlines. Arrows point from the bottom center of each box to the first and third columns of the table, respectively. The table is a 3x3 grid with black outlines.

Customer record

How can we tease this apart?

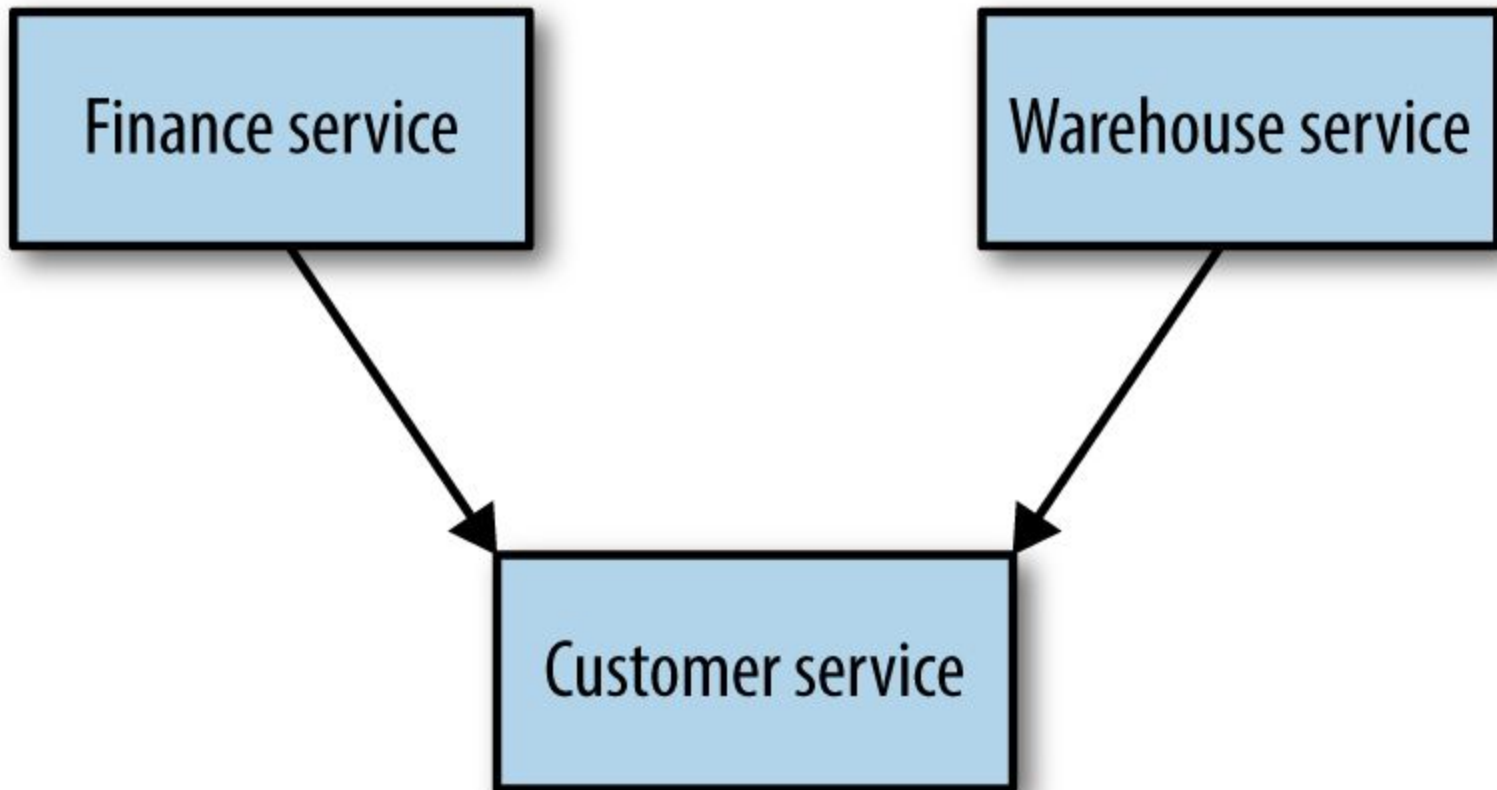
finance and the warehouse code are writing to, and probably occasionally reading from, the same table

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 create a new package called Customer. We can then use an API to expose Customer code to other packages, such as finance or warehouse

we may now end up with a distinct customer service

MusicCorp system



Example: Shared Tables

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

MusicCorp

Catalog

Warehouse

The diagram illustrates the relationship between MusicCorp's Catalog and Warehouse components and a shared Item table. Two light blue boxes labeled 'Catalog' and 'Warehouse' are positioned at the top. Arrows from each box point down to a table with three columns and three rows. The table is labeled 'Item' at the bottom.

Item

Pulling apart the shared table

The answer here is to split the table in two


perhaps creating a stock list table for the warehouse

and a catalog entry table for the catalog details


MusicCorp

Catalog

Warehouse



Catalog item



Stock levels

Refactoring Databases

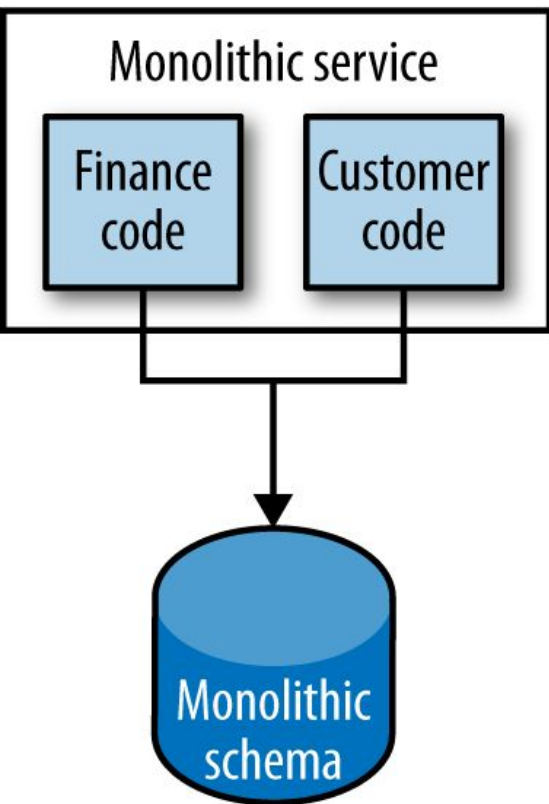
preceding examples are a few database refactorings that can help you separate your schemas

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)

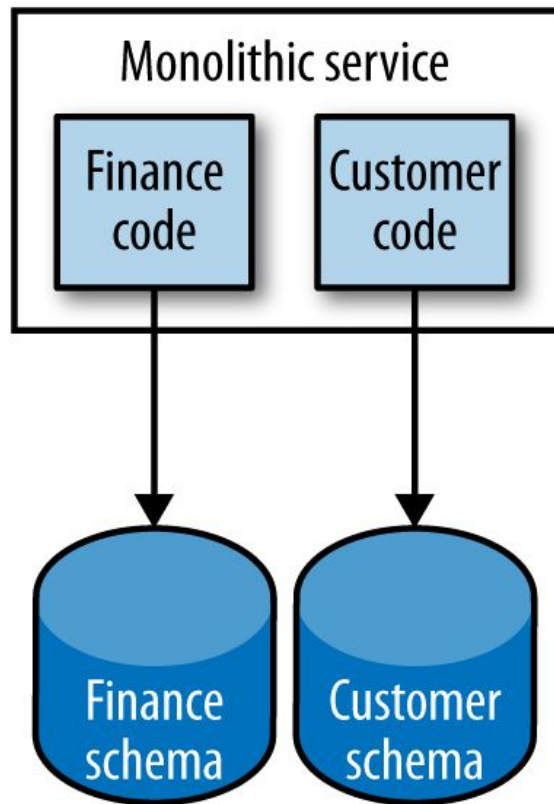
What next?

from one monolithic service with a single schema to two services, each with its own schema?

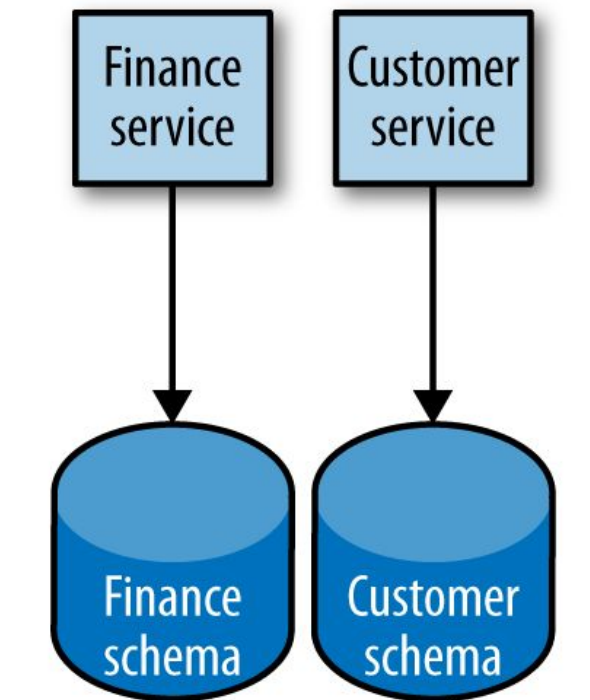
recommend that you split out the schema but keep the service together before splitting the application code out into separate microservices



1. Single schema



2. Split schemas



3. Split application into services

Significant impact on our applications

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, we end up breaking transactional integrity

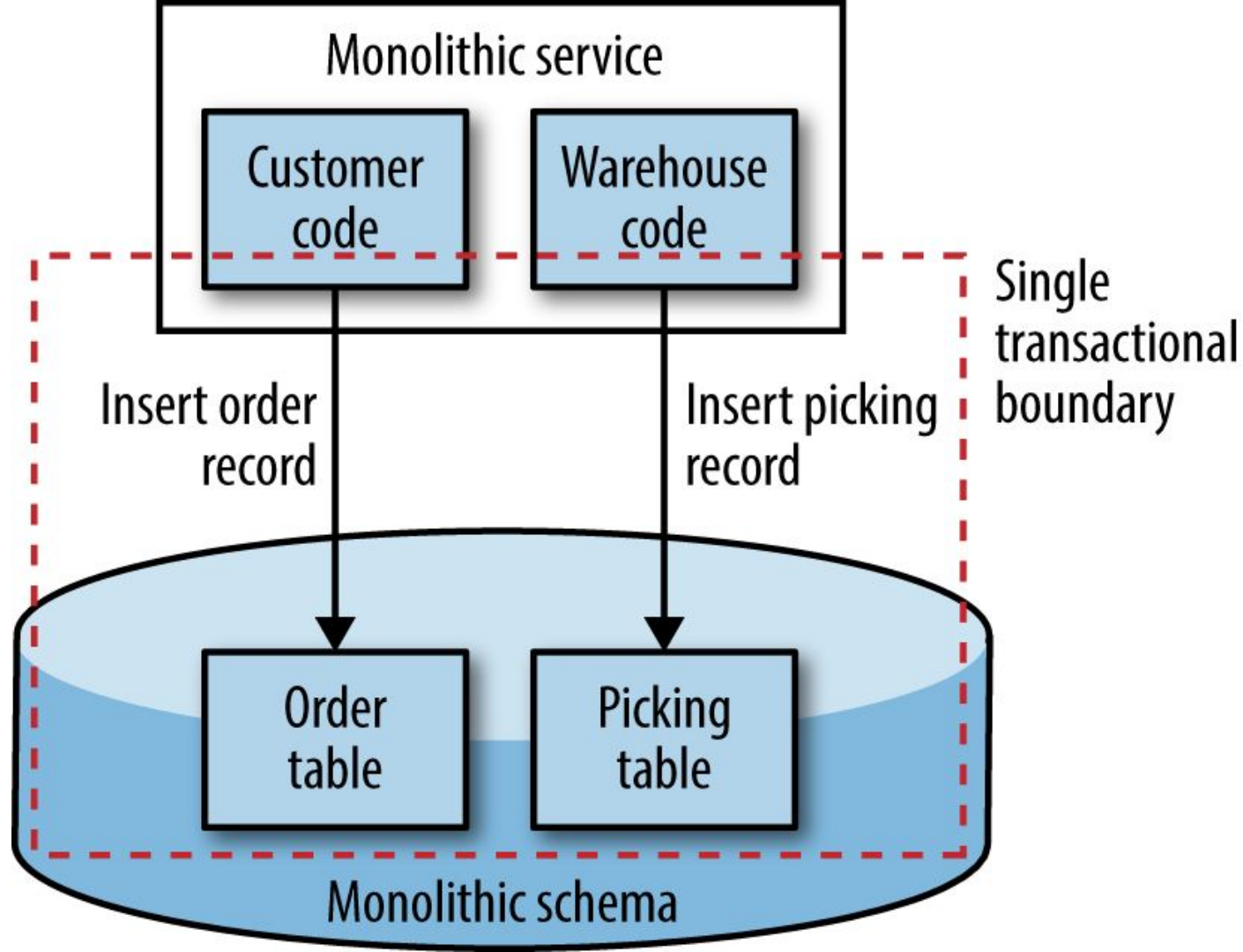
By splitting the schemas out but keeping the application code together, we give ourselves the ability to revert our changes or continue to tweak things without impacting any consumers of our service. Once we are satisfied that the DB separation makes sense, we can then think about splitting out the application 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

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

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

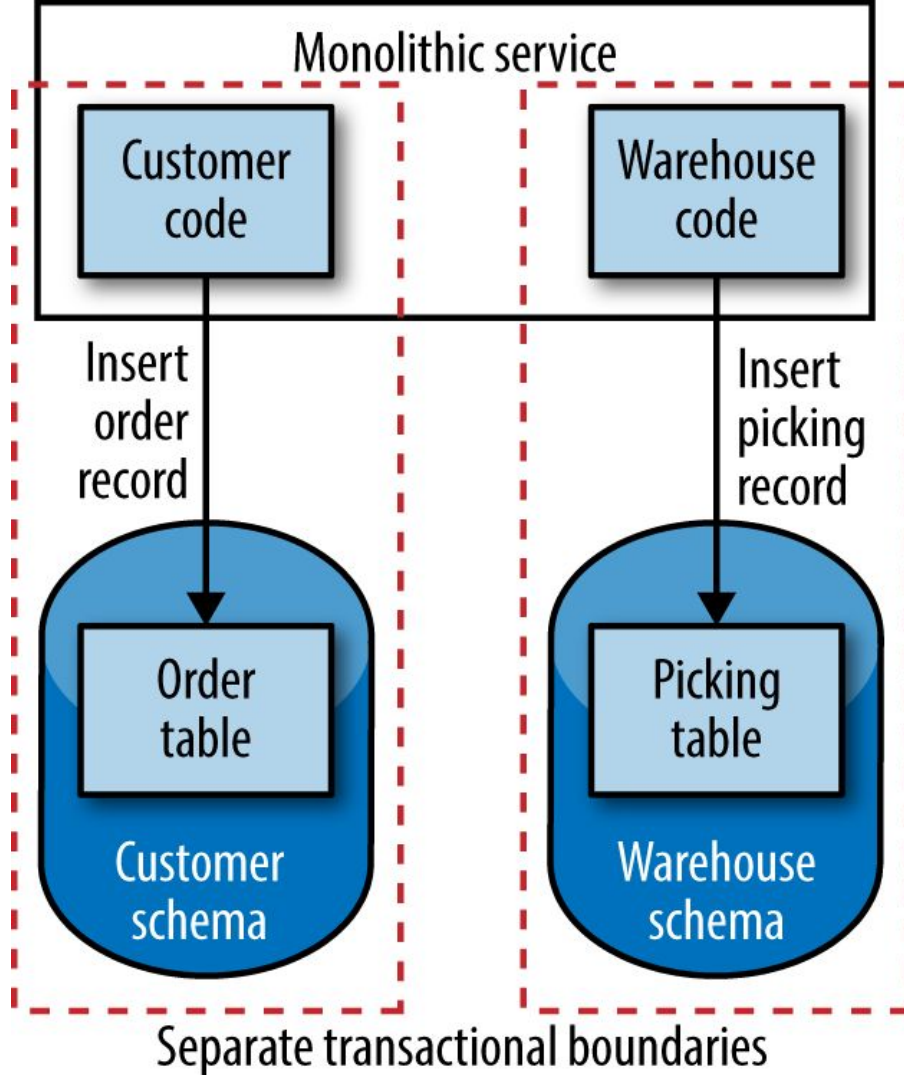


We have lost this transactional safety

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

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?



Try Again Later

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 Or some sorts of operations this makes sense, but we have to assume that a retry would fix it

This is another form of what is called eventual consistency

Abort the Entire Operation

Another option is to reject the entire operation, 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

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

At what logic(service)

Does the logic to handle the compensating transaction live in the customer service, the order service, or somewhere else?

what happens if our compensating transaction fails? It's certainly possible, either need to retry the compensating transaction, or allow some backend process to clean up the inconsistency later on

could be something as simple as a maintenance screen that admin staff had access to, or an automated process

More same issue?

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

most common algorithm for handling distributed transactions—especially short-lived transactions is to use a two-phase commit

Two-Phase Commit(2PC)

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

2PC algorithm isn't foolproof, It just tries to catch most failure cases

If the transaction manager goes down, the pending transactions never complete. If a cohort fails to respond during voting, everything blocks

there is also the case of what happens if a commit fails after voting, if a cohort says yes during the voting period, then we have to assume it will commit

Making scaling systems much more difficult

This coordination process also mean locks; that is, pending transactions can hold locks on resources. Locks on resources can lead to contention

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 algorithms are hard to get right, suggest you avoid trying to create your own, see if you can use an existing implementation

So What to Do?

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?

Try really hard

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 (eg. idea of an “in-process-order” gives you a natural place to focus all logic around processing the order end to end (and dealing with exceptions)

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

Reporting

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 Reporting Database

Reporting typically needs to group together data from across multiple parts of our organization in order to generate useful output

We might want to enrich the data from our general ledger with descriptions of what was sold, which we get from a catalog

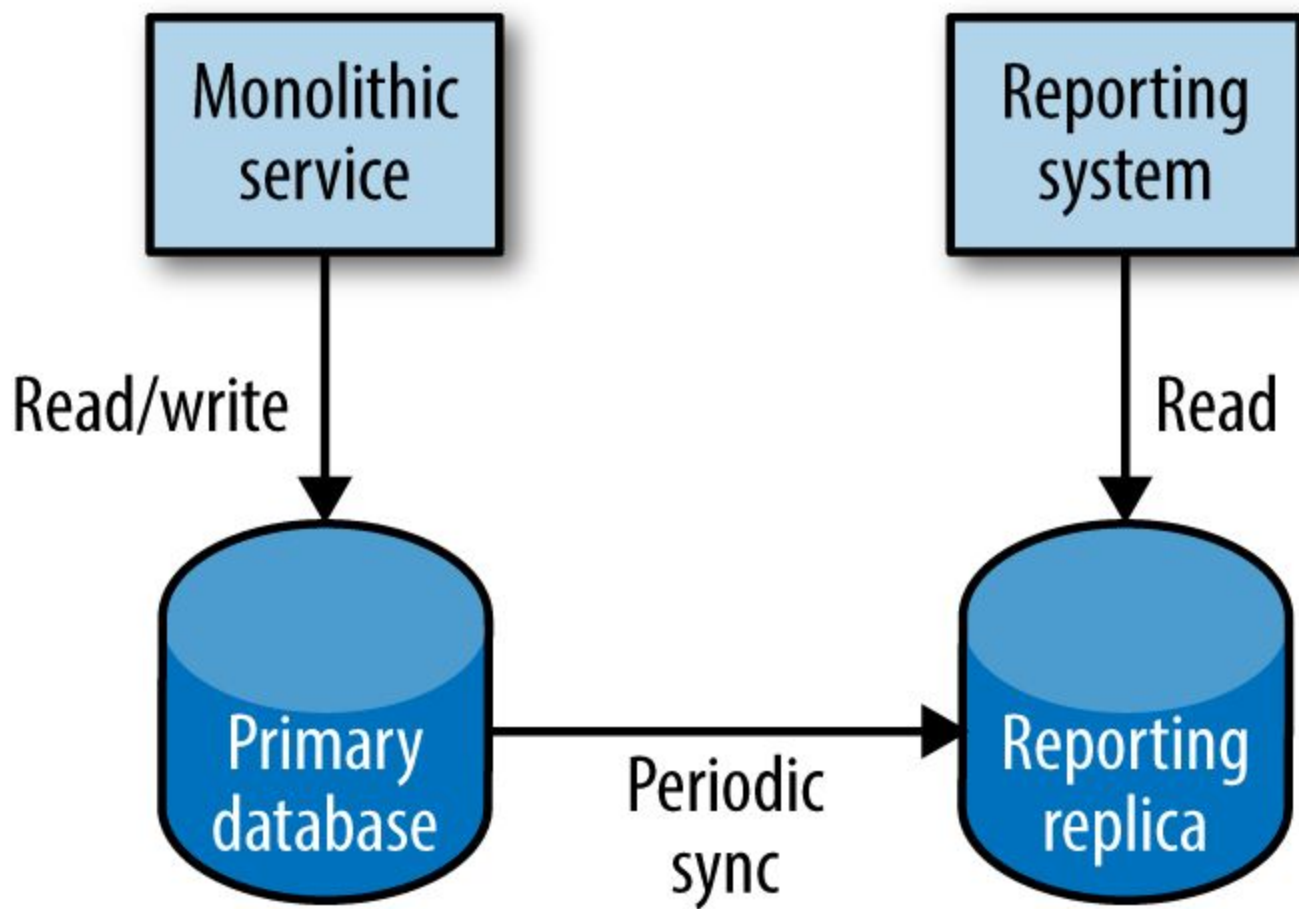
Or we might want to look at the shopping behavior of specific, high-value customers, which could require information from their purchase history and their customer profile

Standard read replication

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



Some downsides

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

Reduces the chances of anyone wanting to take on the task of making and coordinating such a change

Limited options as to how the database can be optimized for either use case—backing the live system or the reporting system

We cannot structure the data differently to make reporting faster if that change in data structure has a bad impact on the running system

Some downsides (cont.)

The database options available to us have exploded recently

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?

What if we'd rather use a document store like MongoDB?

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

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)

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?

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

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

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, Keeping a local copy of this data in the reporting system is dangerous, as we may not know if it has changed

This quickly will become a very slow operation

Pull data periodically into a SQL database

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

But this still presents us with some challenges, 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

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

Adding cache headers

We could speed up some of the data retrieval by adding cache headers to the resources exposed by our service

Data cached in something like a reverse proxy, the nature of reporting is often that we access the long tail of data, means that we may well request resources that no one else has requested before

Resulting in a potentially expensive cache miss

Exposing batch APIs to make reporting easier

Our customer service could allow you to pass a list of customer IDs to it to retrieve them in batches

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 Return an HTTP 202 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

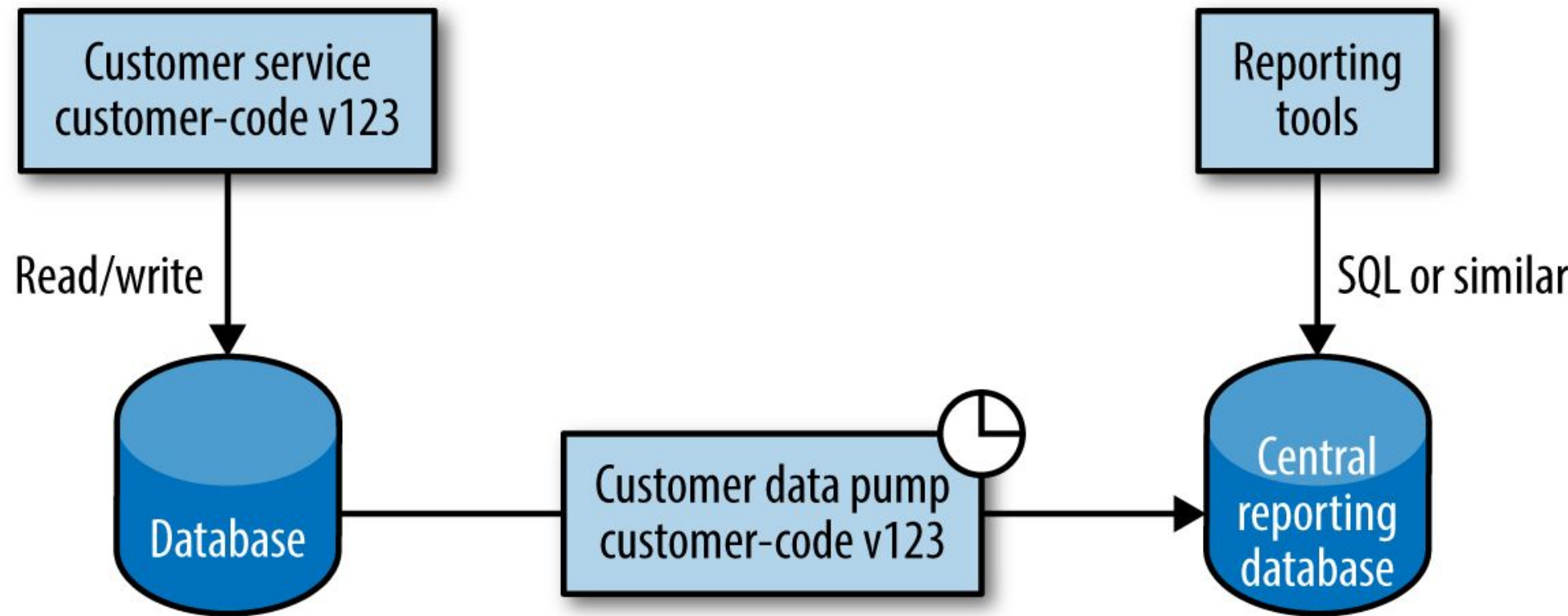
This allow potentially large data files to be exported without the overhead of being sent over HTTP, the system could simply save a CSV file to a shared location

Data Pumps (potentially simpler solutions)

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

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



But

You said having lots of programs integrating on the same database is a bad idea!

This approach, if implemented properly, is a notable exception

Making the reporting easier mitigated the downsides of the coupling

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

Try to reduce the problems with coupling to the service's schema

Having the same team that manages the service also manage the pump, version-control these together

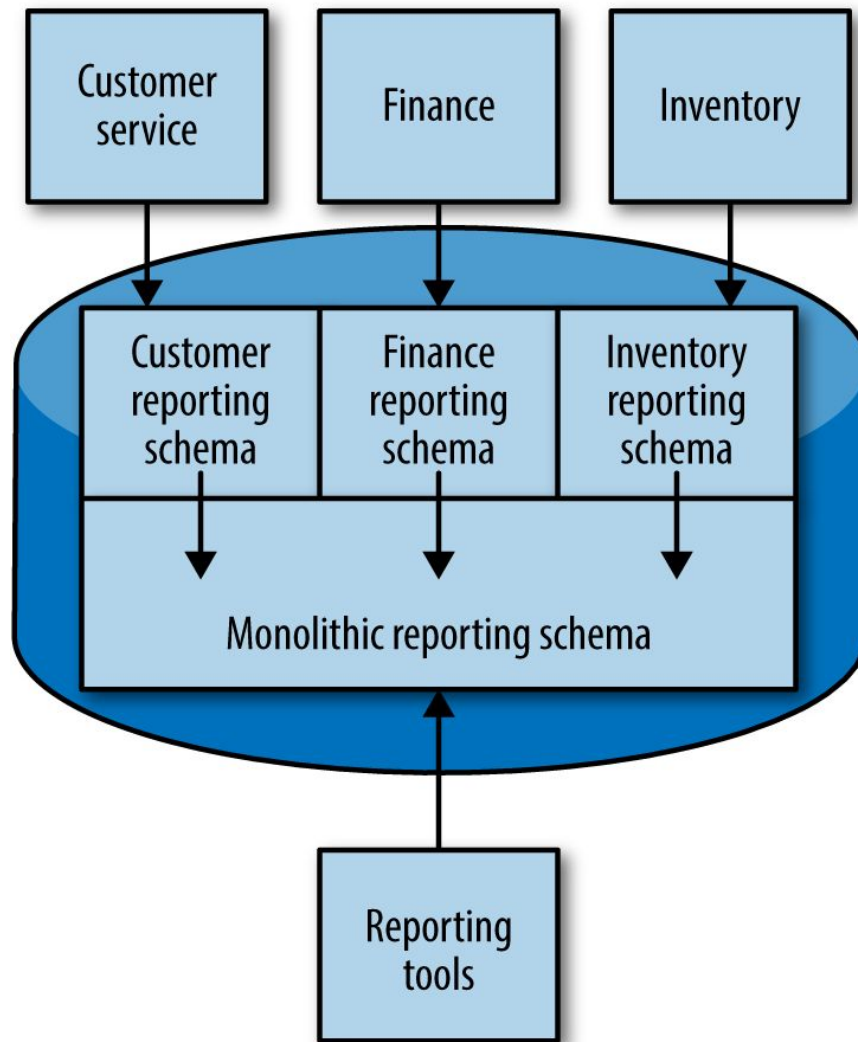
Have builds of the data pump created as an additional artifact as part of the build of the service itself

Whenever you deploy one of them, 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

Example of this for relational databases

Have one schema in the reporting database for each service, using things like materialized views to create the aggregated view

Expose only the reporting schema for the customer data to the customer data pump



Complexity of a segmented schema is not worthwhile

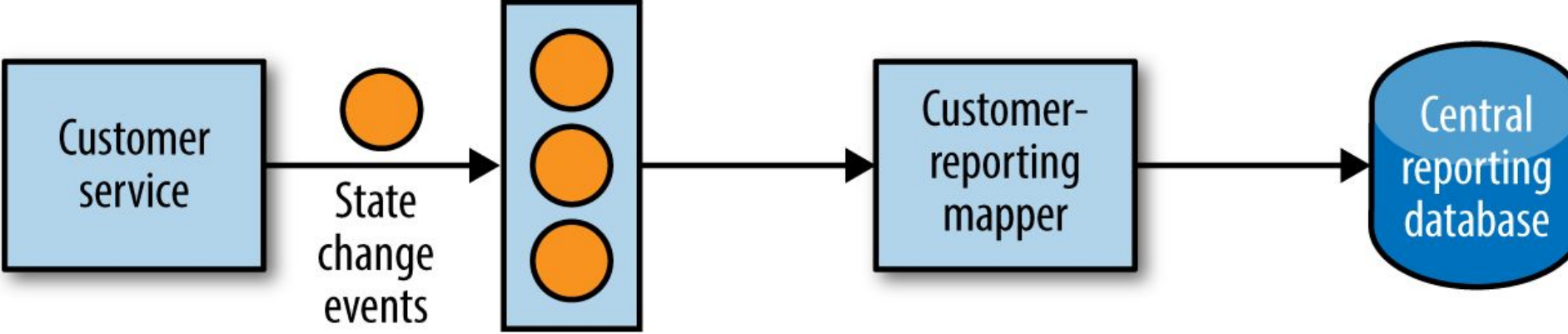
the complexity of integration is pushed deeper into the schema, and will rely on capabilities in the database to make such a setup performant

the complexity of a segmented schema is not worthwhile, especially given the challenges in managing change in the database

Event Data Pump

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



Flow faster

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

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

The main downsides

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

Netflix needs to report across all this data, but given the scale involved this is a nontrivial 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.

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

Netflix are moving more and more toward generic eventing systems capable of routing our data to multiple different places

Cost of Change

Why promote the need to make small, incremental changes, 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

We should also understand how best to mitigate the costs of those mistakes

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

The whiteboard

Sketch out your proposed design, See what happens when you run use cases across what you think your service boundaries will be

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?

When working through a proposed design, for each service, 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?

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

Knowing where to start

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 all of this has helped

Another challenge

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 non-trivial tasks

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

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

Reference

Sam Newman (2015) “Building Microservices--DESIGNING FINE-GRAINED SYSTEMS”

Thank you !

