Developing Transactional Microservices Using Aggregates, Event Sourcing and CQRS

- The Microservice architecture functionally decomposes an application into services, each of which corresponds to a business capability.

- A key challenge when developing microservice-based business applications is that transactions, domain models, and queries resist decomposition.

- A domain model can be decomposed into Domain Driven Design aggregates.

- Each service’s business logic is a domain model consisting of one or more Domain Driven Design aggregates.

- Within a service, each transaction creates or updates a single aggregate.

- Events are used to maintain consistency between aggregates (and services).

Decomposing an application into services is not as easy as it sounds. Several different aspects of applications - domain models, transactions and queries - are difficult to decompose.

Problem #1 - Decomposing a Domain Model

The Domain Model pattern is a good way to implement complex business logic. The domain model for an online store would include classes such as Order, OrderLineItem, Customer and Product.

In a microservices architecture, the Order and OrderLineItem classes are part of the Order Service, the Customer class is part of the Customer Service, and the Product class belongs to the Catalog Service.

The challenge with decomposing the domain model, however, is that classes often reference one another. For example, an Order references its Customer and an OrderLineItem references a Product. What do we do about references that want to span service boundaries?

## Aggregate from Domain-Driven Design (DDD) solves this problem.

Microservices and Databases

A distinctive feature of the microservice architecture is that the data owned by a service is only accessible via that service’s API. In the online store, for example, the OrderService has a database that includes the ORDERS table and the CustomerService has its database, which includes the CUSTOMERS table.

Because of this encapsulation, the services are loosely coupled. At development time, a developer can change their service’s schema without having to coordinate with developers working on other service. At runtime, the services are isolated from each other.

Problem #2 - Implementing Transactions That Span Services

A traditional monolithic application can rely on ACID transactions to enforce business rules.

 Imagine, for example, that customers of the online store have a credit limit that must be checked before creating a new order. The application must ensure that potentially multiple concurrent attempts to place an order do not exceed a customer’s credit limit. If Orders and Customers reside in the same database it is trivial to use an ACID transaction.

## we cannot use such a straightforward approach to maintain data consistency in a microservices-based application. The ORDERS and CUSTOMERS tables are owned by different services and can only be accessed via APIs

## solution is to use an event-driven architecture based on a technique known as event sourcing.

Problem #3 - Querying and Reporting

In a traditional monolithic application it is extremely common to write queries that use joins.

We cannot use this kind of query in a microservices-based online store.

Event Sourcing, which makes querying even more challenging.

## The solution is to maintain materialized views using an approach known as Command Query Responsibility Segregation (CQRS).

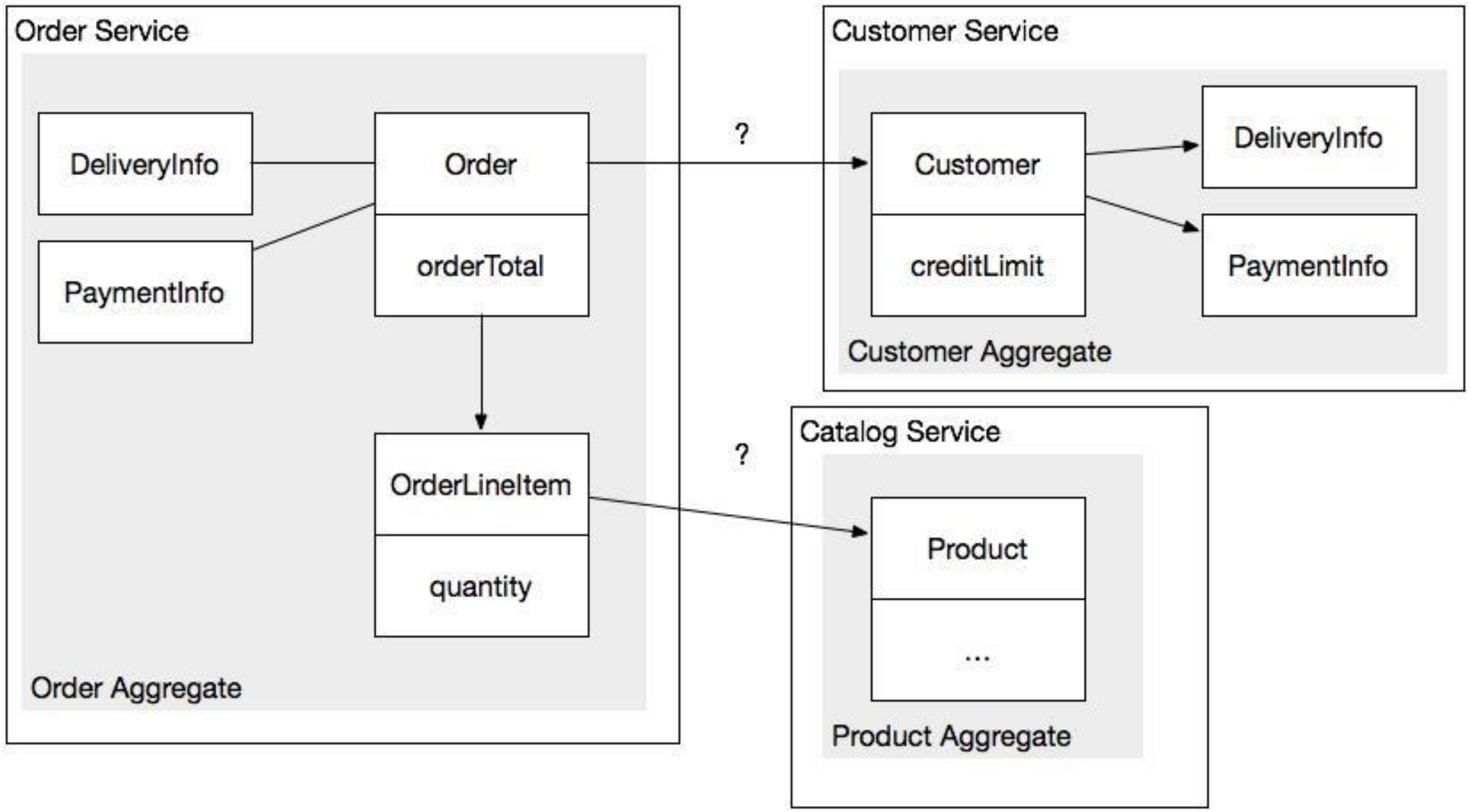
Domain-Driven Design(DDD) Aggregates (Building Blocks of Microservices)

What is an Aggregate?

An aggregate is a cluster of domain objects that can be treated as a unit. It consists of a root entity and possibly one or more other associated entities and value objects.

For example, the domain model for the online store contains aggregates such as Order and Customer. An Order aggregate consists of an Order entity (the root), one or more OrderLineItem value objects along with other value objects such as a delivery Address and PaymentInformation.

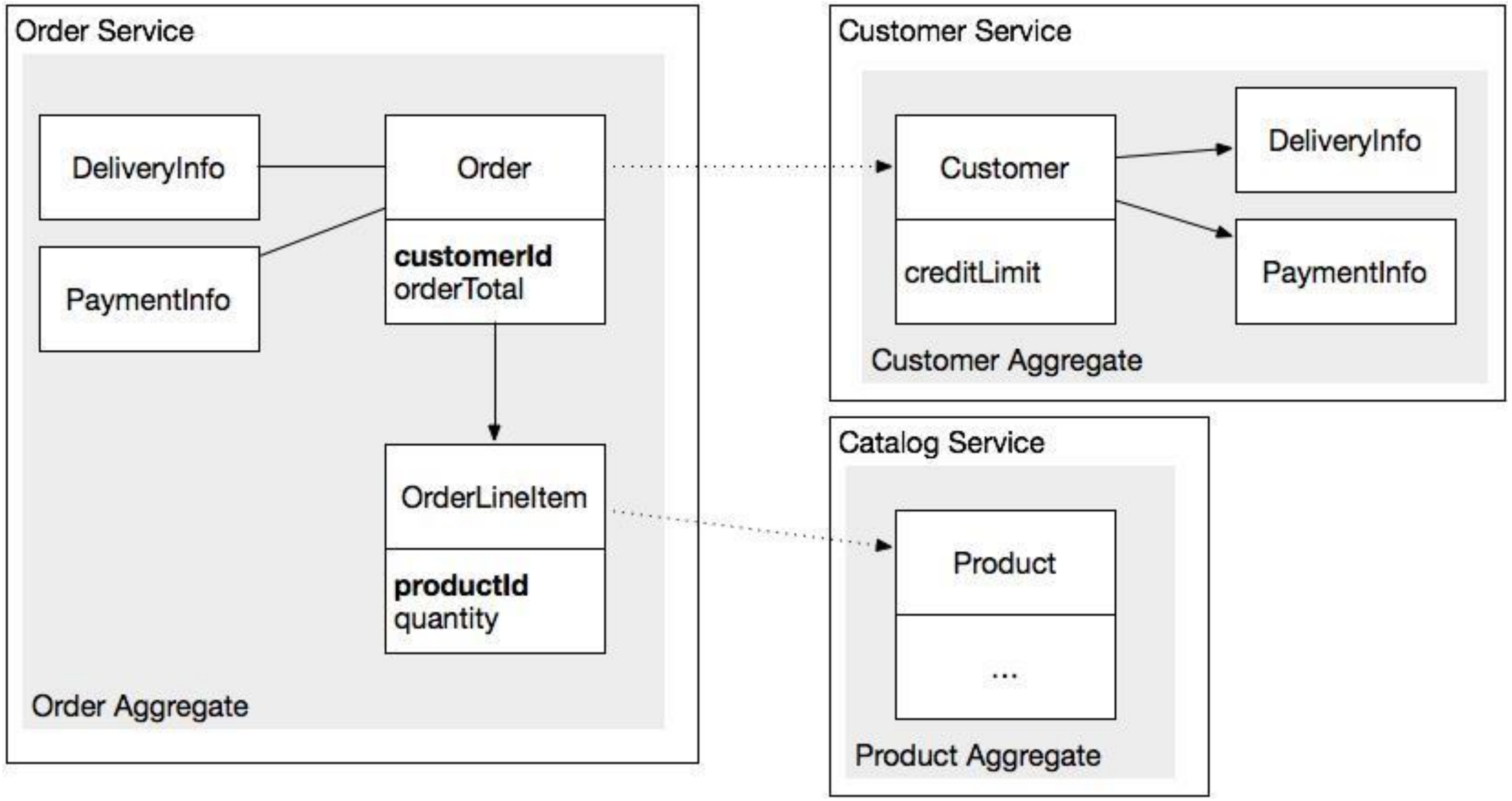
A Customer aggregate consists of the Customer root entity along with other value objects such a DeliveryInfo and PaymentInformation.


Order Service
DeliveryInfo
Order
orderTotal
PaymentInfo
OrderLineItem
quantity
Order Aggregate
?
?
Customer Service
Customer
creditLimit
DeliveryInfo
PaymentInfo
Customer Aggregate
Catalog Service
Product
...
Product Aggregate

Inter-Aggregate References Must Use Primary Keys

The first rule is that aggregates reference each other by identity (e.g. primary key) instead of object references.

For example, an Order references its Customer using a customerId rather than a reference to the Customer object. Similarly, an OrderLineItem references a Product using a productId.


Order Service
DeliveryInfo
Order
PaymentInfo
customerId
orderTotal
OrderLineItem
productId
quantity
Order Aggregate
Customer Service
Customer
creditLimit
DeliveryInfo
PaymentInfo
Customer Aggregate
Catalog Service
Product
...
Product Aggregate

This approach is quite different than traditional object modelling, which considers foreign keys in the domain model.

The use of identity rather than object references means that the aggregates are loosely coupled. You can easily put different aggregates in different services.

In fact, a service’s business logic consists of a domain model that is a collection of aggregates.

For example, the OrderService contains the Order aggregate and the CustomerService contains the Customer aggregat

One Transaction Creates or Updates One Aggregate

The second rule that aggregates must obey is that a transaction can only create or update a single aggregate This constraint also matches the limited transaction model of most NoSQL databases.

Even though a transaction can only create or update a single aggregate, applications must still maintain consistency between aggregates.

The Order Service must, for example, verify that a new Order aggregate will not exceed the Customer aggregate’s credit limit. There are a couple of different ways to maintain consistency.

Using Events to Maintain Data Consistency

We define a domain event as something that has happened to an aggregate. An event usually represents a state change.

Consider, for example, an Order aggregate in the online store. Its state changing events include Order Created, Order Cancelled, Order Shipped.

Events can represent attempts to violate a business rule such as a Customer’s credit limit.

Using an Event-Driven Architecture

Services use events to maintain consistency between aggregates as follows: an aggregate publishes an event whenever something notable happens, such as its state changing or there is an attempted violation of a business rule.

Other aggregates subscribe to events and respond by updating their own state.

The online store verifies the customer’s credit limit when creating an order using a sequence of steps:

1. An Order aggregate, which is created with a NEW status, publishes an OrderCreated event

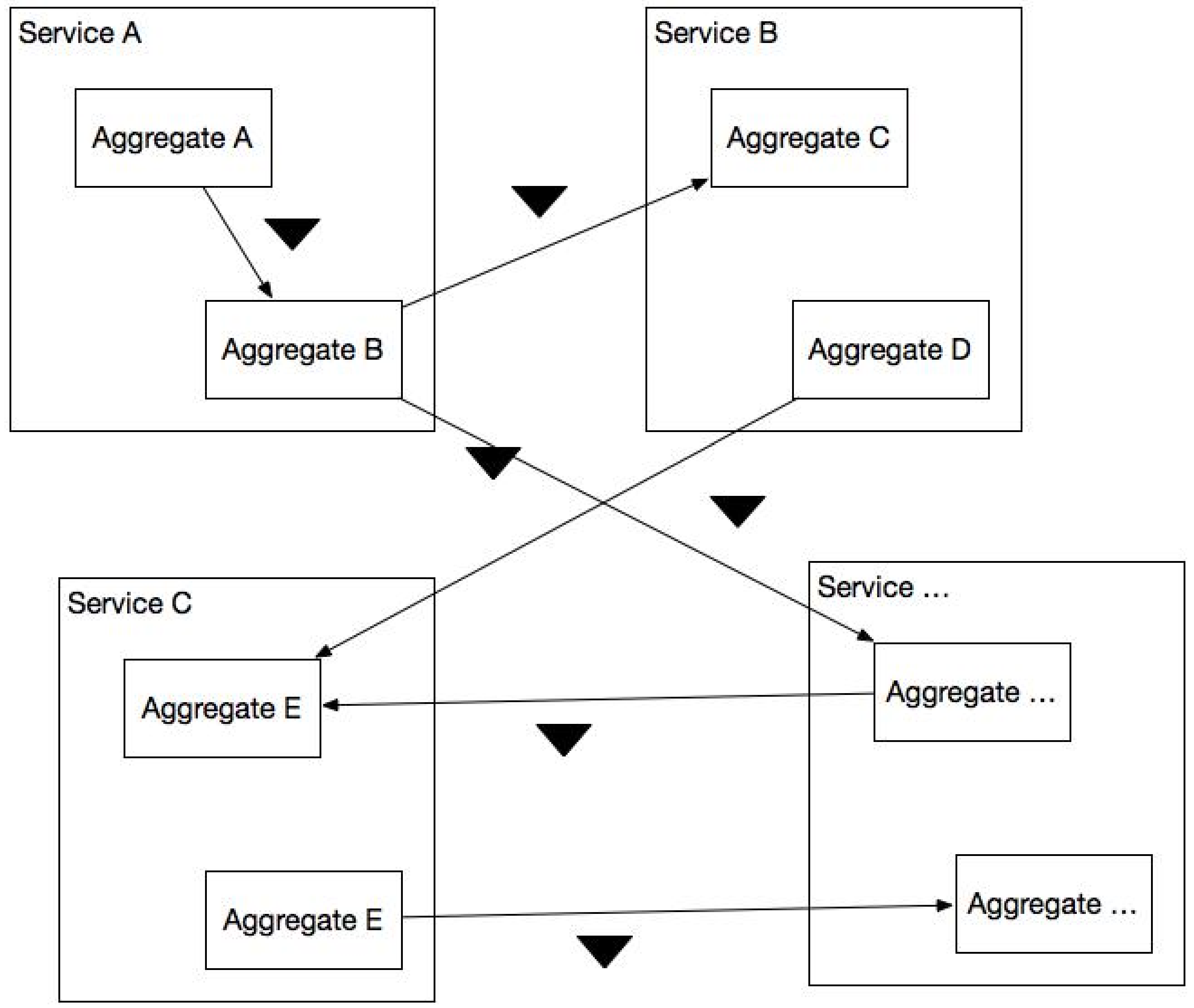
2. The Customer aggregate consumes the OrderCreated event, reserves credit for the order and publishes an CreditReserved event

3. The Order aggregate consumes the CreditReserved event, and changes its status to APPROVED

Microservice Architecture as a Web of Event-Driven Aggregates

Each service's business logic consists of one or more aggregates. Each transaction performed by a service updates or creates a single aggregate. The services maintain data consistency between aggregates by using events.

Microservice Architecture



Event Sourcing

•Event Sourcing is a technique for reliably updating state and publishing events that overcomes limitations of other solutions.

•The design concepts for an event-driven architecture, using event sourcing, align well with microservices architecture patterns.

•Snapshots can improve performance when querying aggregates by combining all events up to a certain point in time.

•Event sourcing can create challenges for queries, but these are overcome by following CQRS guidelines and materialized views.

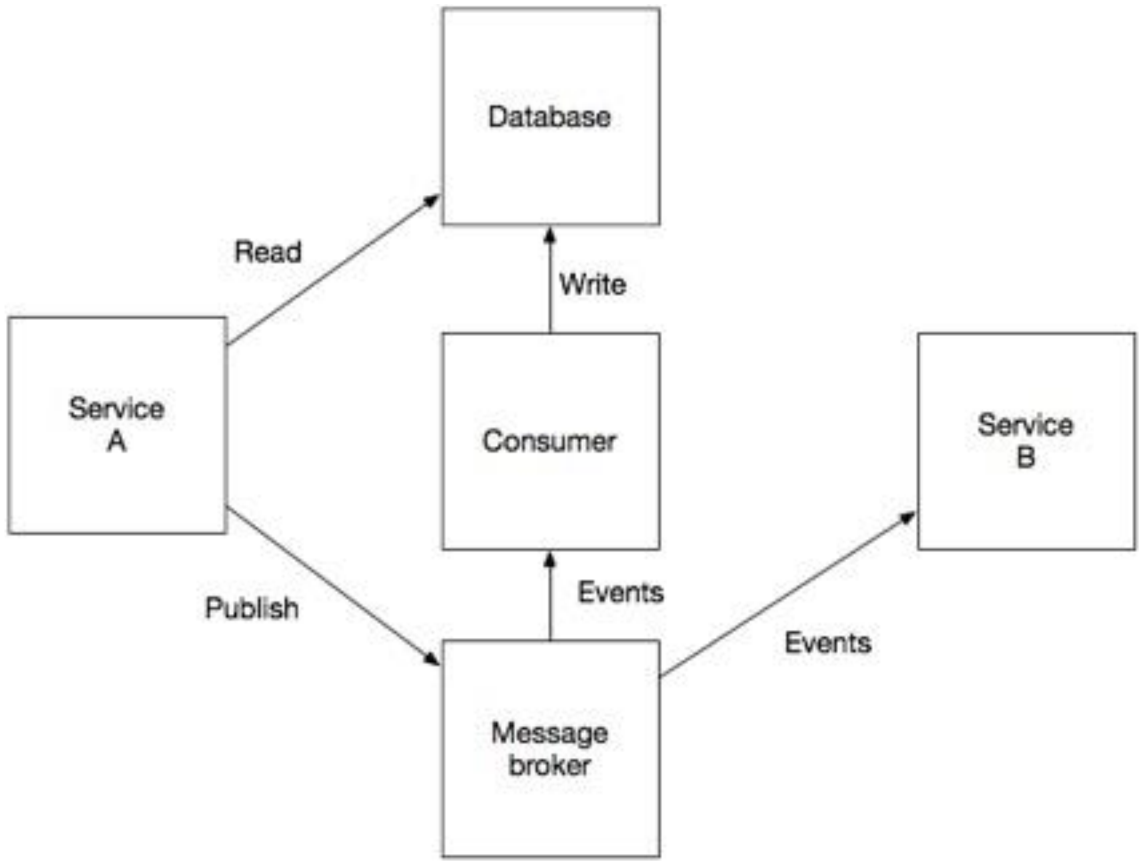
•Event sourcing and CQRS do not require any specific tools or software, and many frameworks exist which can fill in some of the low-level functionality.

Reliably Updating State and Publishing Events

When a service creates or updates an aggregate in the database it simply publishes an event. But there is a problem: updating the database and publishing an event must be done atomically.

Otherwise, if, for example, a service crashed after updating the database but before publishing an event then the system would remain in an inconsistent state.

## One solution, is for the application to perform the update by publishing an event to a message broker such as Apache Kafka.


Database
Read
Write
Service A
Consumer
Service B
Publish
Events
Events
Message broker

Benefits and Drawbacks of Event Sourcing

A major benefit of event sourcing is that it reliably publishes events whenever the state of an aggregate changes. It is a good foundation for an event-driven microservices architecture.

Also, because each event can record the identity of the user that made the change, event sourcing provides an audit log that is guaranteed to be accurate.

Another benefit of event sourcing is that it stores the entire history of each aggregate

The main drawback of event sourcing is that querying the event store can be challenging.

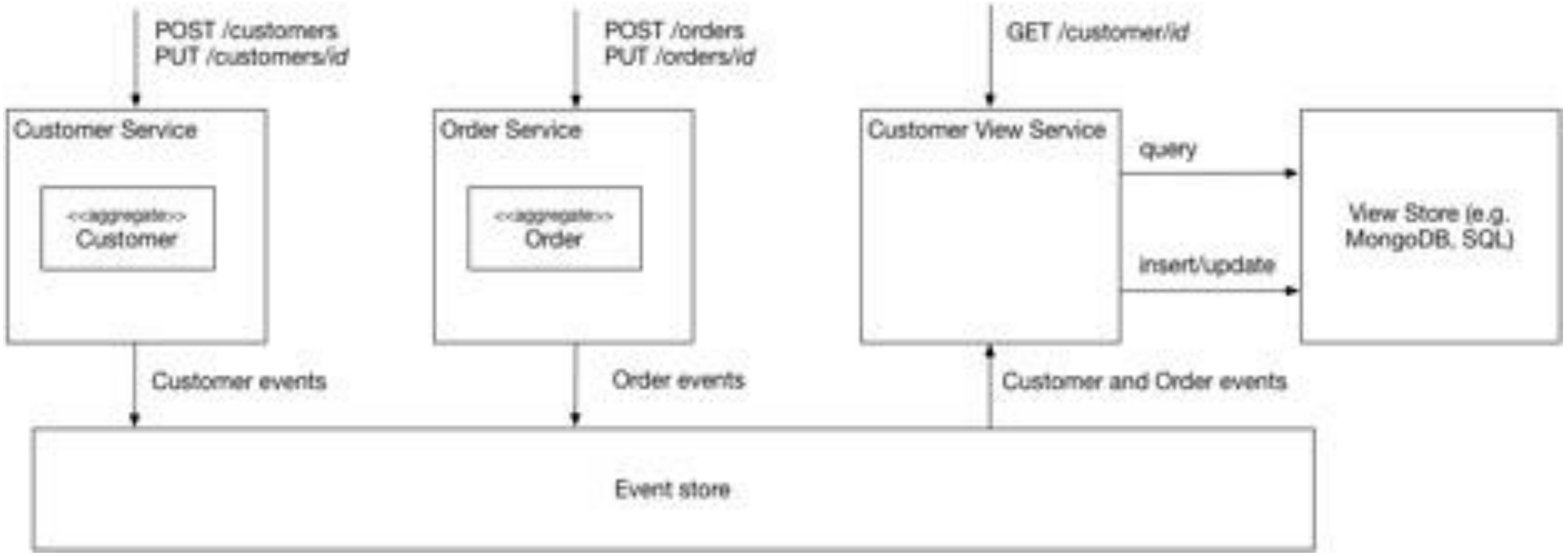
SELECT * FROM CUSTOMER WHERE CREDIT_LIMIT < ? AND c.CREATION_DATE > ?. There isn’t a column containing the credit limit. Instead, we must use a more complex and potentially inefficient query that has a nested SELECT to compute the credit limit by folding events that set the initial credit and adjust it.

Implementing Queries Using CQRS

A good way to implement queries is to use an architectural pattern known as Command Query Responsibility Segregation (CQRS). CQRS, as the name suggests, splits the application into two parts.

The first part is the command-side, which handles commands (e.g. HTTP POSTs, PUTs, and DELETEs) to create, update and delete aggregates. These aggregates are, of course, implemented using *event sourcing*.

The second part of the application is the query side, which handles queries (e.g. HTTP GETs) by querying one or more materialized views of the aggregates. The query side keeps the views synchronized with the aggregates by subscribing to events published by the command side.


POST /customers
PUT /customers/*id*
POST /orders
PUT /orders/*id*
GET /customer/*id*
Customer Service
<<aggregate>>
Customer
Order Service
<<aggregate>>
Order
Customer View Service
query
insert/update
View Store (e.g. MongoDB, SQL)
Customer events
Order events
Customer and Order events
Event store

The Customer View Service subscribes to the Customer and Order events published by the command-side services.

It updates a view store that is implemented using MongoDB. The service maintains a MongoDB collection of documents, one per customer.

Each document has attributes for the customer details. It also has an attribute that stores the customer's recent orders. This collection supports a variety of queries

Summary

A major challenge when using events to maintain data consistency between services is atomically updating the database and publishing events.

The traditional solution is to use a distributed transaction spanning the database and the message broker.

2PC, however, is not a viable technology for modern applications.

A better approach is to use event sourcing, which is an event-centric approach to business logic design and persistence.

Another challenge in the microservice architecture is implementing queries. Queries often need to join data that is owned by multiple services.

However, joins are no longer straightforward since data is private to each service.

Using event sourcing also makes it even more difficult to efficiently implement queries since the current state is not explicitly stored.

The solution is to use Command Query Responsibility Segregation (CQRS) and maintain one or more materialized views of the aggregates that can be easily queried.