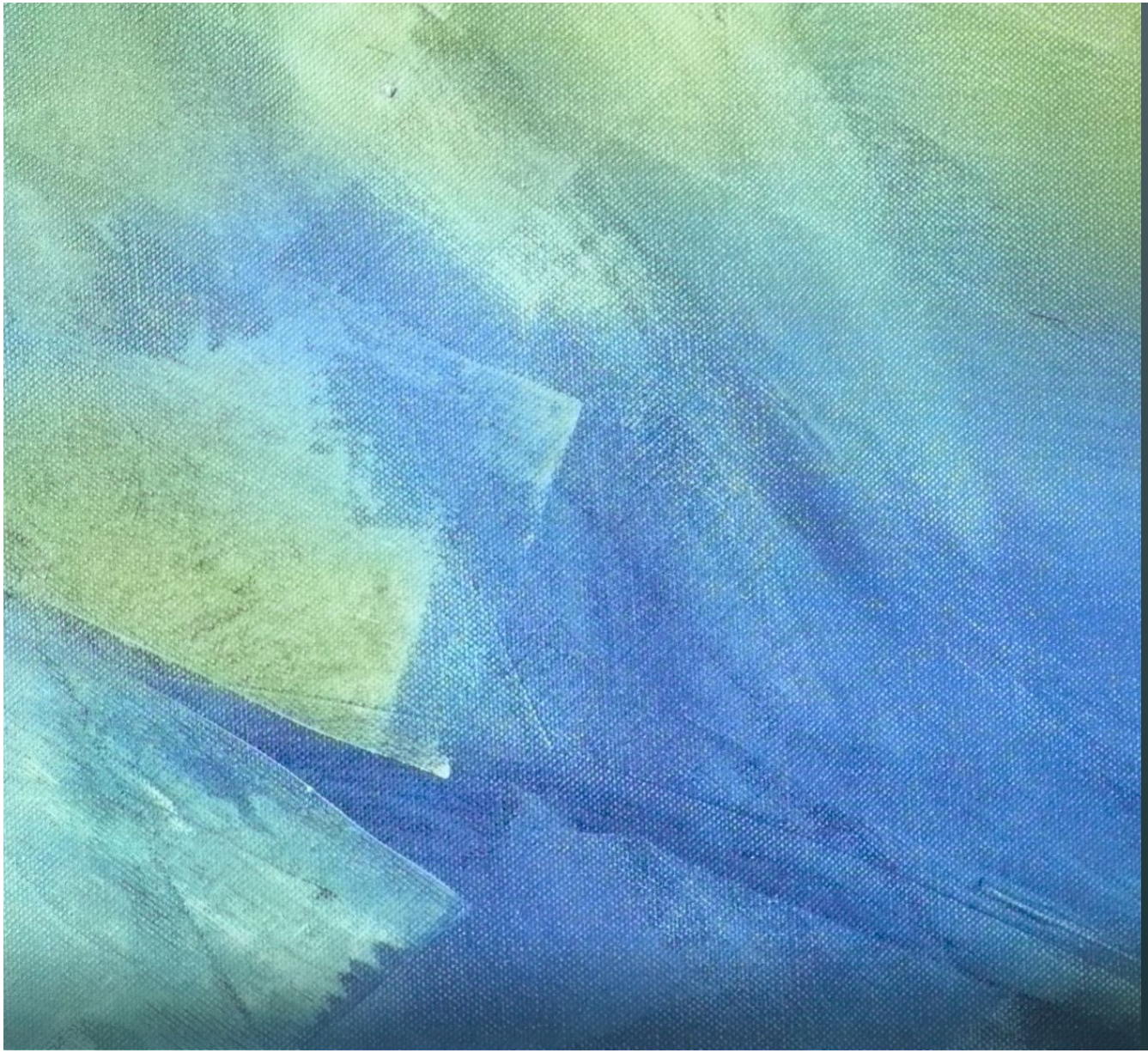


Implementing queries in a microservice architecture

Wiwat Vatanawood

Duangdao Wichadakul

Nuengwong Tuaycharoen

Pittipol Kantavat

This chapter covers

- The challenges of querying data in a microservice architecture
- When and how to implement queries using the API composition pattern
- When and how to implement queries using the Command query responsibility segregation (CQRS) pattern

Agenda

- **Overview of querying data in a microservice architecture**
- Querying using the API composition pattern
- Querying Using the CQRS pattern
- Designing CQRS views

Overview of querying data (1)

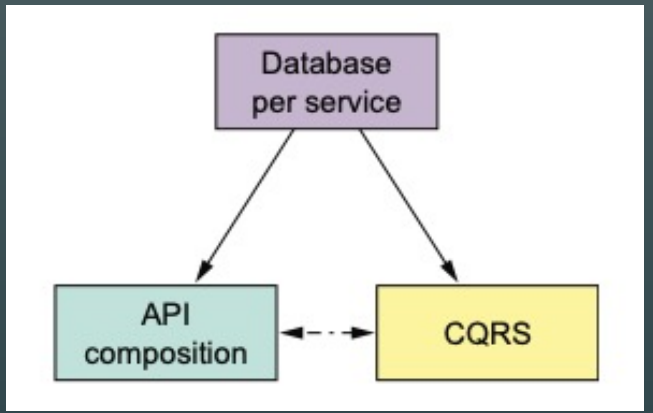
- An application implements a variety of query operations.
- Querying in the monolithic application is straightforward
  → because it has a single database.
- But in the microservice, queries often need to retrieve data scattered among the databases owned by multiple services

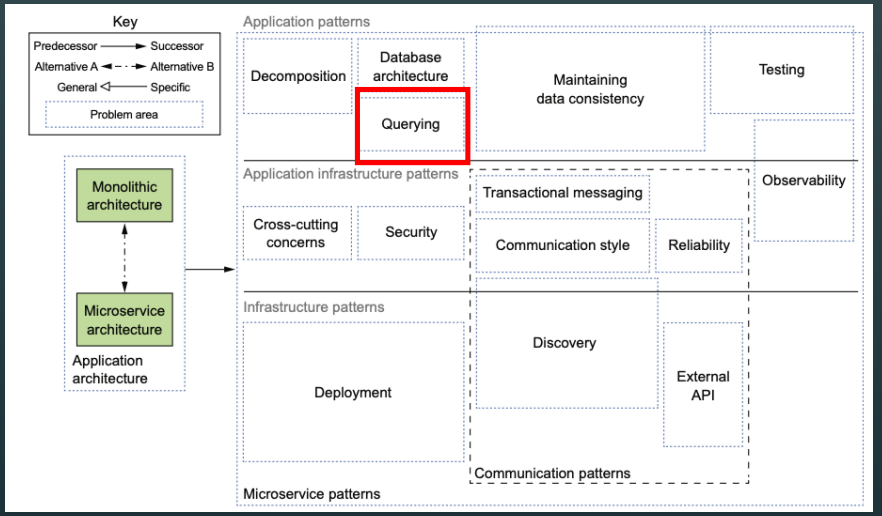
Overview of querying data (2)

- There are two different patterns for implementing query operations:
  1. **The API composition pattern**
     - The simplest approach and should be used whenever possible.
  2. **The Command query responsibility segregation (CQRS) pattern**
     - More powerful than the API composition pattern, but also more complex.

Issue: How to perform queries? (Recalling)

- Context :-
  - Each service has its own database
- Force :-
  - Queries must join data from multiple services
  - Data is private to a service





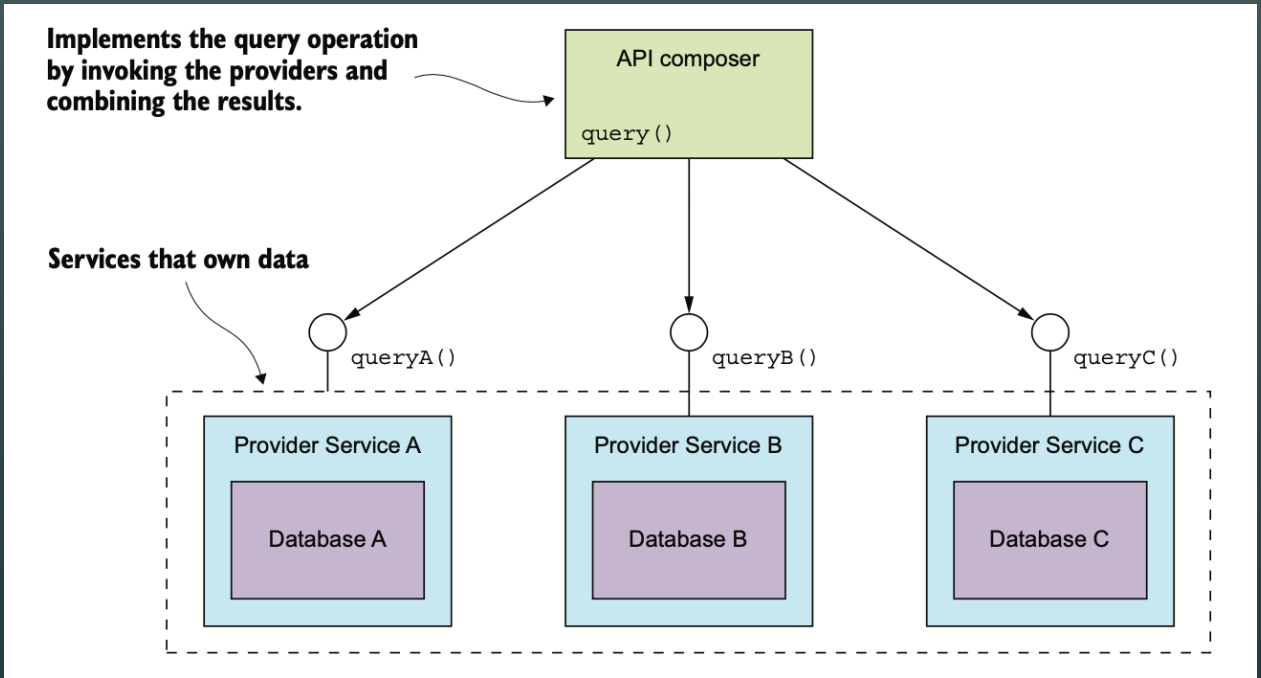
Agenda

- Overview of querying data in a microservice architecture
- **Querying using the API composition pattern**
- Querying Using the CQRS pattern
- Designing CQRS views

Overview of the API composition pattern (1)

- This pattern implements a query operation by
  - Invoking the services that own the data
  - Combining the results
- The structure of this pattern has two types of participants:
  1. An API composer - This implements the query operation by querying the provider services.
  2. A provider service - A service that owns some of the data that the query returns.

Overview of the API composition pattern (2)



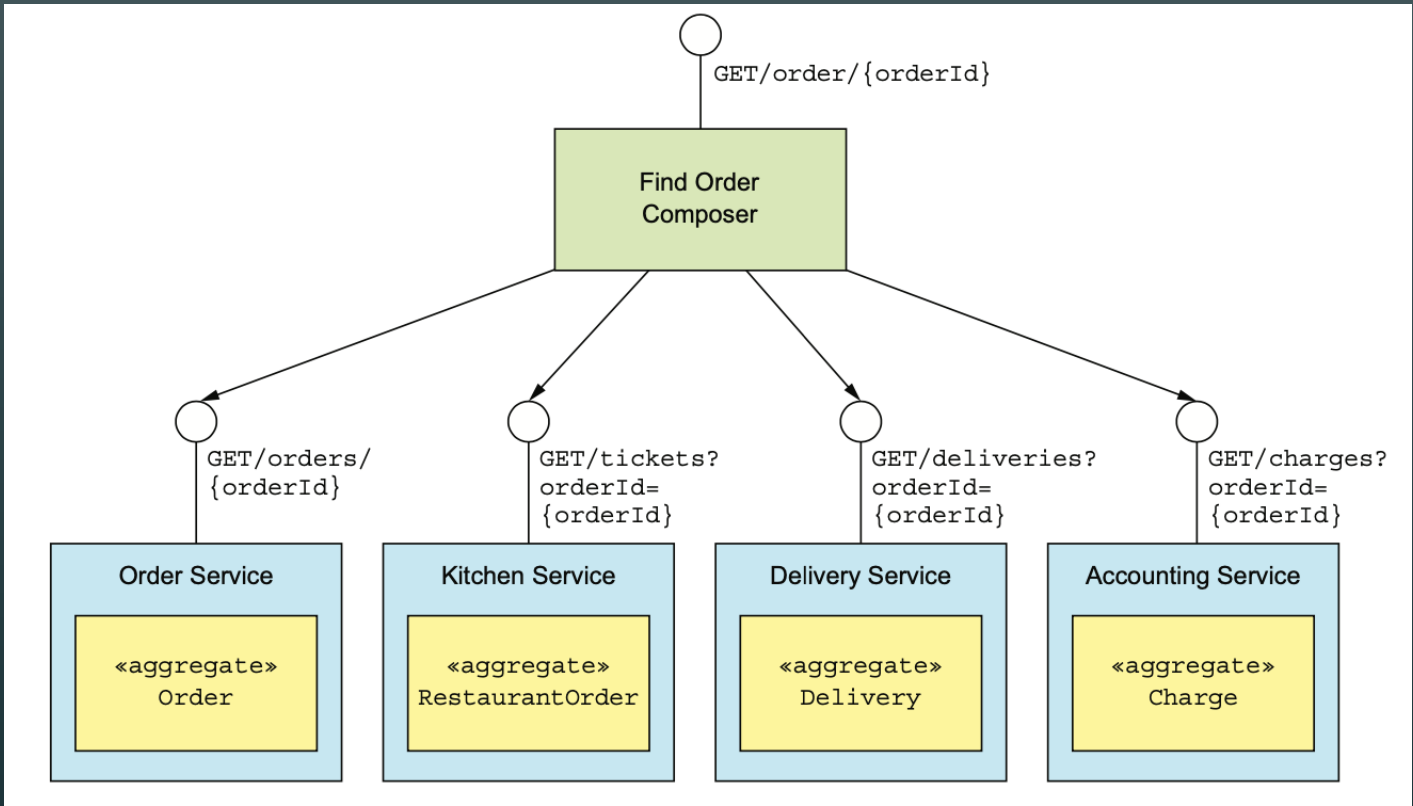
- The API composition pattern consists of an API composer and two or more provider services.

Overview of the API composition pattern (3)

- An API composer might be :-
  - A client, such as a web application
  - A service, such as an API gateway
- To implement a particular query operation depends on :-
  - How the data is partitioned
  - The capabilities of the APIs exposed by the services that own the data
  - The capabilities of the databases used by the services
- The aggregator might need to perform in-memory join of large datasets

Example from FTGO

- Implementing the *findOrder()* using the API composition pattern

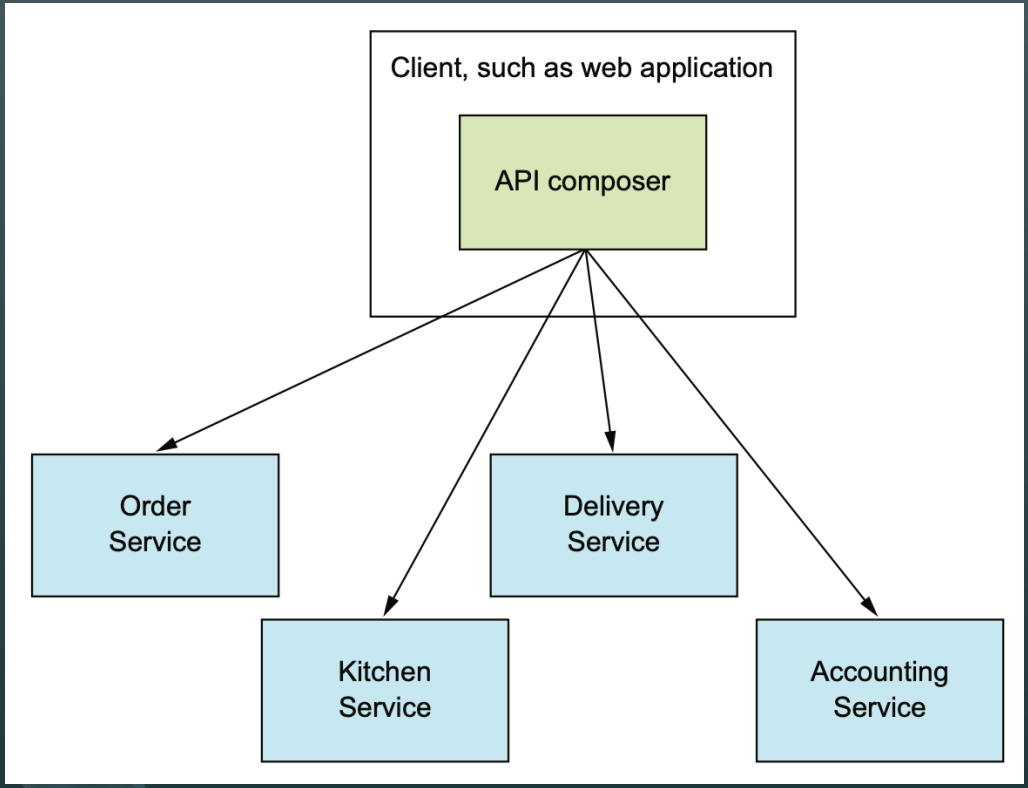


API composition design issues

A couple of design issues:

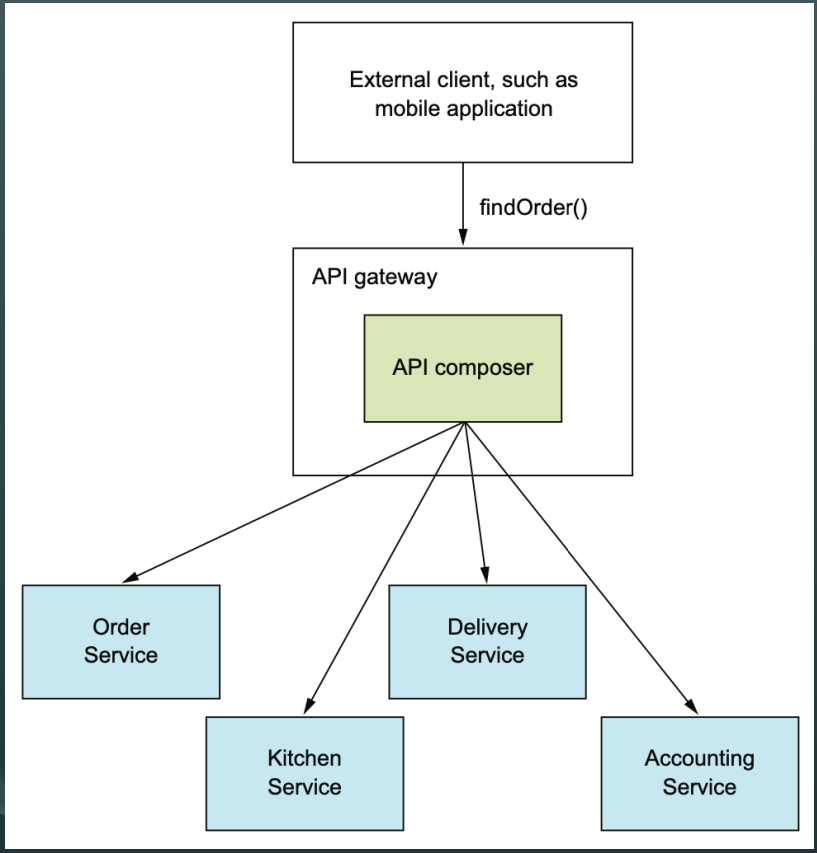
1. Deciding which component in your architecture is the query operation's API composer
   - Use a client of the services to be the API composer
   - Use an API gateway to play the role of an API composer
   - Implement an API composer as a stand-alone service
2. How to write efficient aggregation logic

Option 1 - Using a client of the services to be the API composer



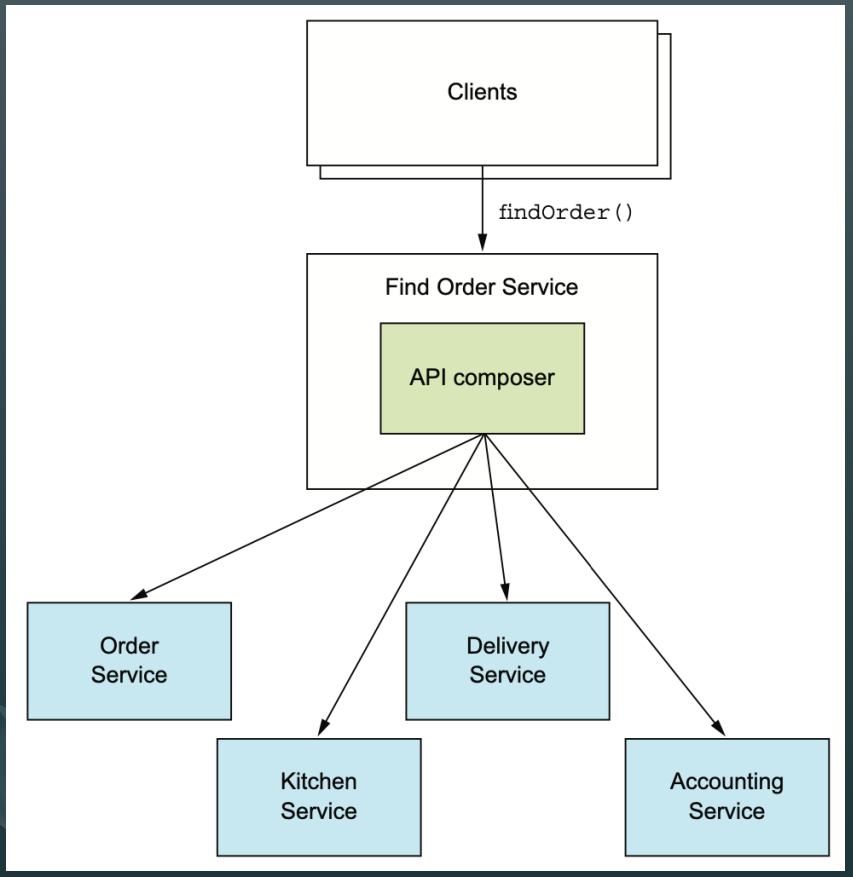
- Implementing API composition in a client.
- The client queries the provider services to retrieve the data
- Drawback :-
  - Probably not practical for clients that are outside of the firewall and access services via a slower network

Option 2 - Using an API gateway to play the role of an API composer



- Implementing API composition in the API gateway.
- This option makes sense if the query operation is part of the application's external API
- Enables a client, such as a mobile device, that's running outside of the firewall to efficiently retrieve data from numerous services with a single API call.

Option 3 - Using an API composer as a stand-alone service



- Implement a query operation used by multiple clients and services as a stand-alone service.
- Use this option for a query operation that's used internally by multiple services.
- Can also be used for externally accessible query operations whose aggregation logic is too complex to be part of an API gateway.

How to write efficient aggregation logic?

- API composer should use a reactive programming model
  - To program using and relying on events.
  - Instead of the order of lines in the code.
- An API composer should call provider services in parallel in order to minimize the response time for a query operation.
- The logic to efficiently execute a mixture of sequential and parallel service invocations can be complex.

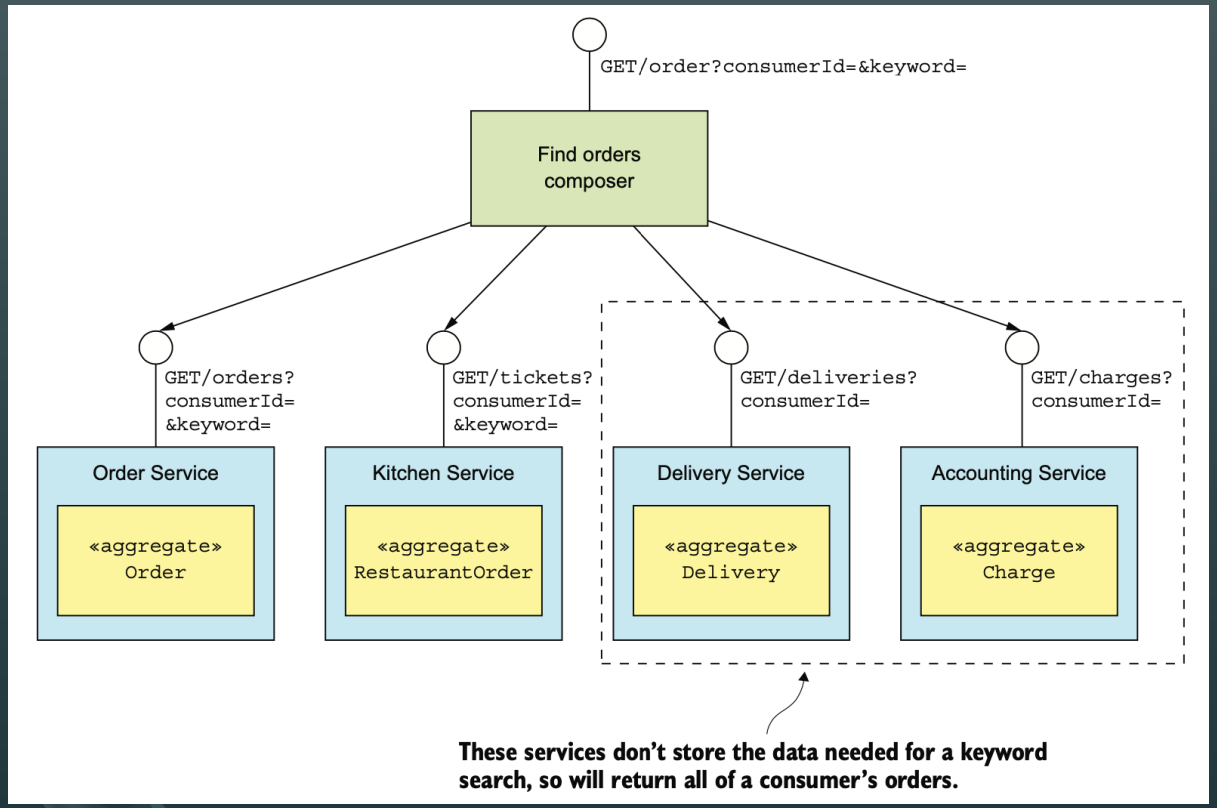
Drawbacks of the API composition pattern

- Increased overhead
  - Computing and network resources are required.
  - Increasing the cost of running the application.
- Risk of reduced availability
  - The availability of an operation declines with the number of services that are involved.
- Lack of transactional data consistency
  - An API composer might not always be able to detect inconsistent data.

Agenda

- Overview of querying data in a microservice architecture
- Querying using the API composition pattern
- **Querying Using the CQRS pattern**
- Designing CQRS views

Motivations for using CQRS (1)



- The *findOrderHistory()* operation retrieves a consumer's order history.
- Not all services store the attributes that are used for filtering or sorting.
  - Neither *Delivery Service* nor *Accounting Service* can sort by **menu** items
  - Neither *Kitchen Service* nor *Delivery Service* can sort by the **orderCreationDate** attribute.

Motivations for using CQRS (2)

- Solution 1 - Do an in-memory join
  1. Retrieves all orders for the consumer from Delivery Service and Accounting Service.
  2. Then, performs a join with the orders retrieved from Order Service and Kitchen Service.
- Drawback :-
  - It potentially requires the API composer to retrieve and join large datasets, which is inefficient.

Motivations for using CQRS (3)

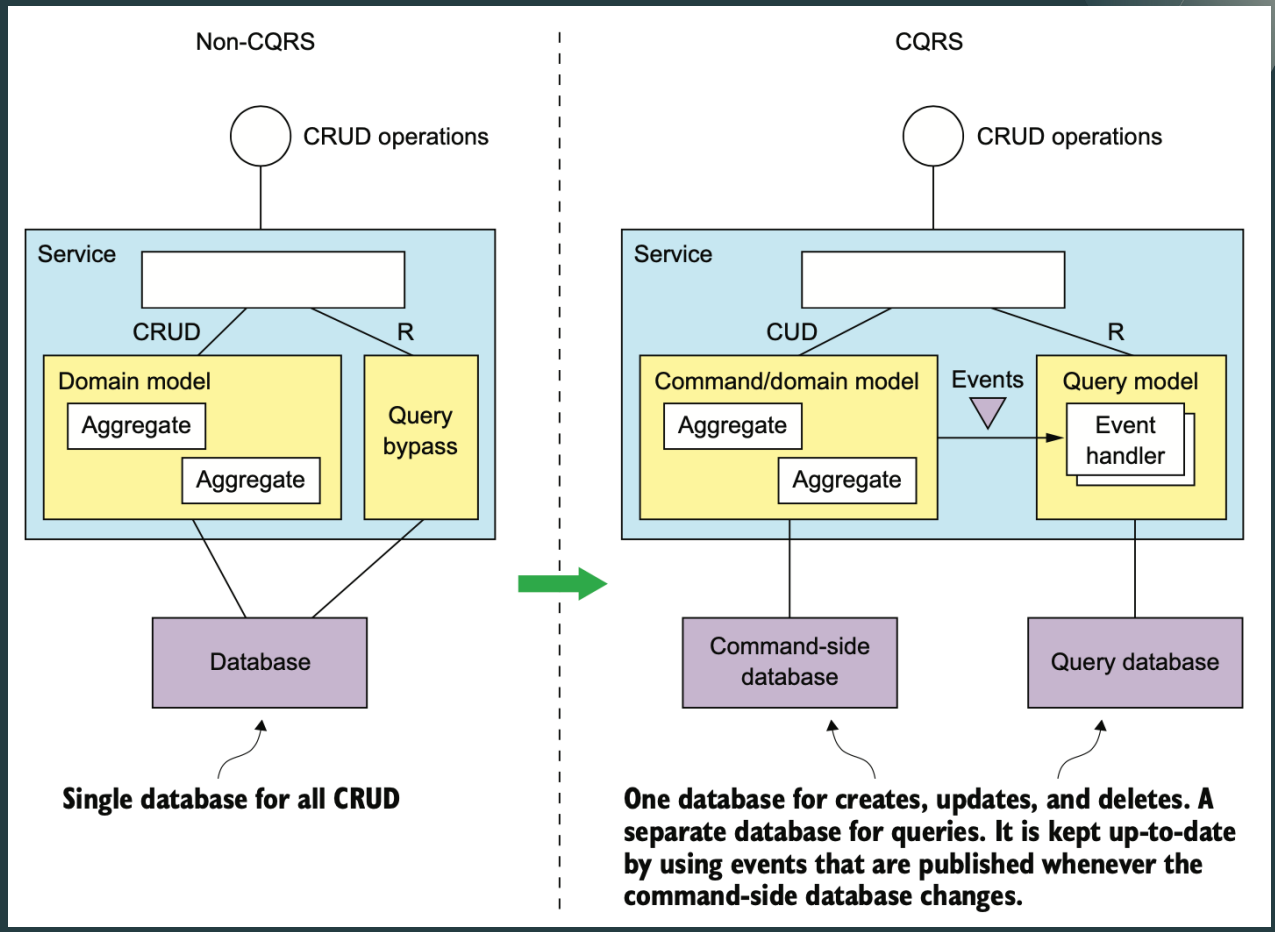
- Solution 2 - Matching then request by fetching ID
  1. Matching orders from Order Service and Kitchen Service and
  2. Then, request orders from the other services by ID
- Drawback :-
  - Only practical if those services have a bulk fetch API.
  - Inefficient because of excessive network traffic.

Motivations for using CQRS (4)

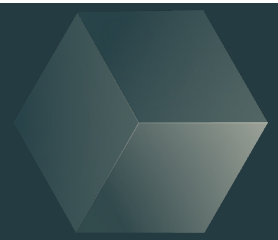
- Even queries that are local to a single service can be difficult to implement.
- There are a couple of reasons :-
  1. It's not appropriate for the service that owns the data to implement the query. (There is a need to separate concerns)
  2. Sometimes a service's database (or data model) doesn't efficiently support the query.
- For example, consider the *findAvailableRestaurants()* query operation.
  - The key challenge is performing an efficient geospatial query.

Command Query Responsibility Segregation (CQRS)

- CQRS separates command from queries
- Restructures a service into command-side and query-side modules, which have separate databases.



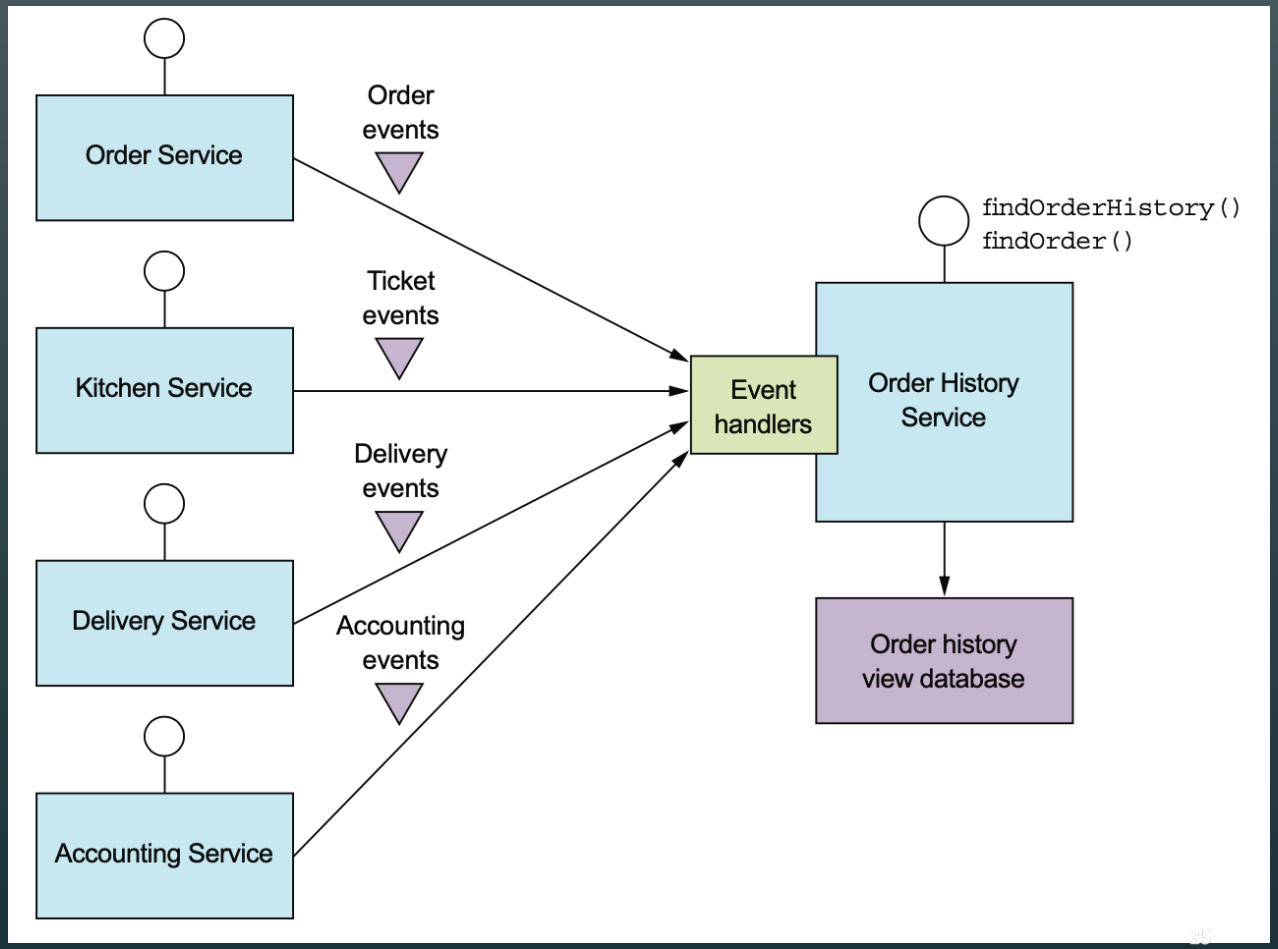
CQRS and query-only service



- We can use CQRS pattern to define query services.
  - Consisting of only query operations—no command operations.
  - Keeping up-to-date by subscribing to events published by one or more other services.
- This kind of view doesn't belong to any particular service.
- So, it makes sense to implement it as a standalone service.

(Example) The design of Order History Service

- It implements the *findOrderHistory()* query operation by querying a database, which it maintains by subscribing to events published by multiple other services.
- The popular approach :-
  - Use RDBMS as the system of record
  - Use a text search engine, such as Elasticsearch, to handle text queries



Benefits of CQRS

- Enables the efficient implementation of queries in a microservice architecture.
- Enables the efficient implementation of diverse queries.
- Makes querying possible in an event sourcing-based application.
  - Capture all changes to an application state as a sequence of events.
- Improves separation of concerns.

Drawbacks of CQRS

- More complex architecture
  - Developers must write the query-side services that update and query the views.
  - An application might use different types of databases
- Dealing with the replication lag
  - There's delay between the command side publishes an event and the query side processes the view updated.

Solution to the replication lag

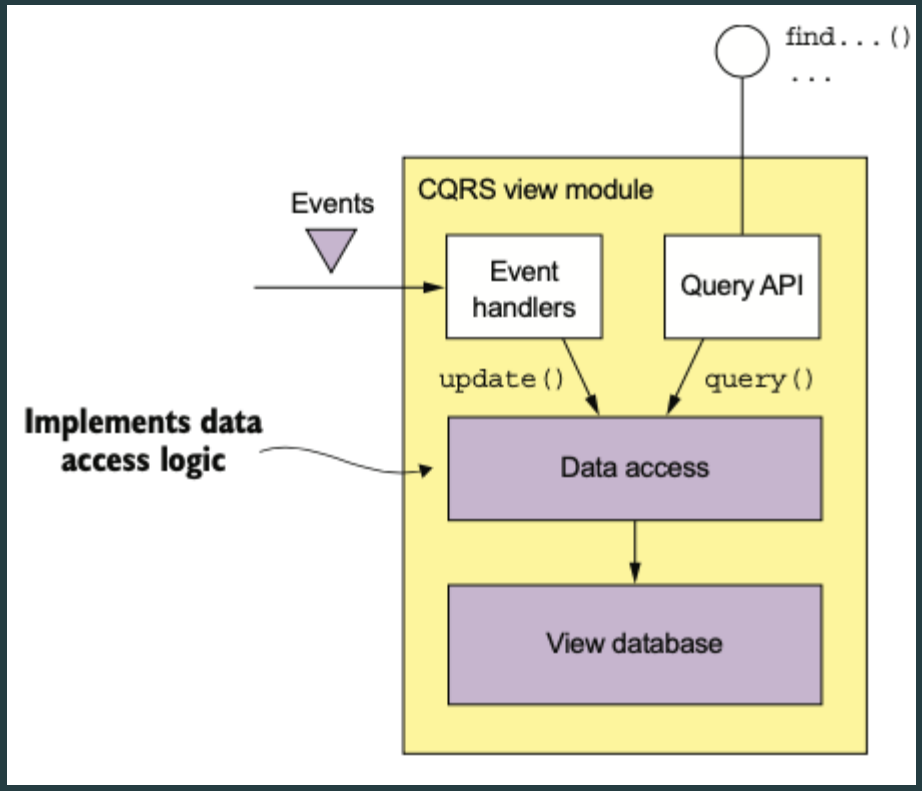
- Solution 1 - Supply the client with version information.
  - Enables the client to tell that the query side is out-of-date.
  - A client can poll the query-side view until it's up-to-date.
- Solution 2 - Updating its local model once the command is successful without issuing a query.
  - Update its model using data returned by the command.
  - Drawback - The UI code may need to duplicate server-side code in order to update its model.

Agenda

- Overview of querying data in a microservice architecture
- Querying using the API composition pattern
- Querying Using the CQRS pattern
- **Designing CQRS views**

Designing CQRS views (1)

- A CQRS view module consists of a view database and three submodules.
  - The data access module implements the database access logic.
  - The event handlers module subscribes to events and updates the database.
  - The query API module implements the query API.



Designing CQRS views (2)

We must make some important decisions when developing a view module :-

- Choose a database and design the schema.
- When designing the data access module
  - Ensuring that updates are idempotent.
  - Handling concurrent updates.
- When implementing a new view or changing the schema, you must
  - Implement a mechanism to efficiently build or rebuild the view.
- Decide how to enable a client of the view to cope with the replication lag.

Choosing a view datastore (1)

Key issue 1 - SQL vs. NoSQL databases

- Not that long ago, the SQL-based RDBMS rule them all.
- As the Web grew in popularity, the RDBMS couldn't satisfy scalability.
- For certain use cases, NoSQL have certain advantages over SQL databases.
  - More flexible data model
  - Better performance and scalability
- NoSQL database is often a good choice for a CQRS view.
  - leverage its strengths and ignore its weaknesses

Query-side view stores

If you need	Use	Example
PK-based lookup of JSON objects	A document store such as MongoDB or DynamoDB, or a key value store such as Redis	Implement order history by maintaining a MongoDB document containing the per-customer.
Query-based lookup of JSON objects	A document store such as MongoDB or DynamoDB	Implement customer view using MongoDB or DynamoDB.
Text queries	A text search engine such as Elasticsearch	Implement text search for orders by maintaining a per-order Elasticsearch document.
Graph queries	A graph database such as Neo4j	Implement fraud detection by maintaining a graph of customers, orders, and other data.
Traditional SQL reporting/BI	An RDBMS	Standard business reports and analytics.

Choosing a view datastore (2)

Key issue 2 – Supporting update operations

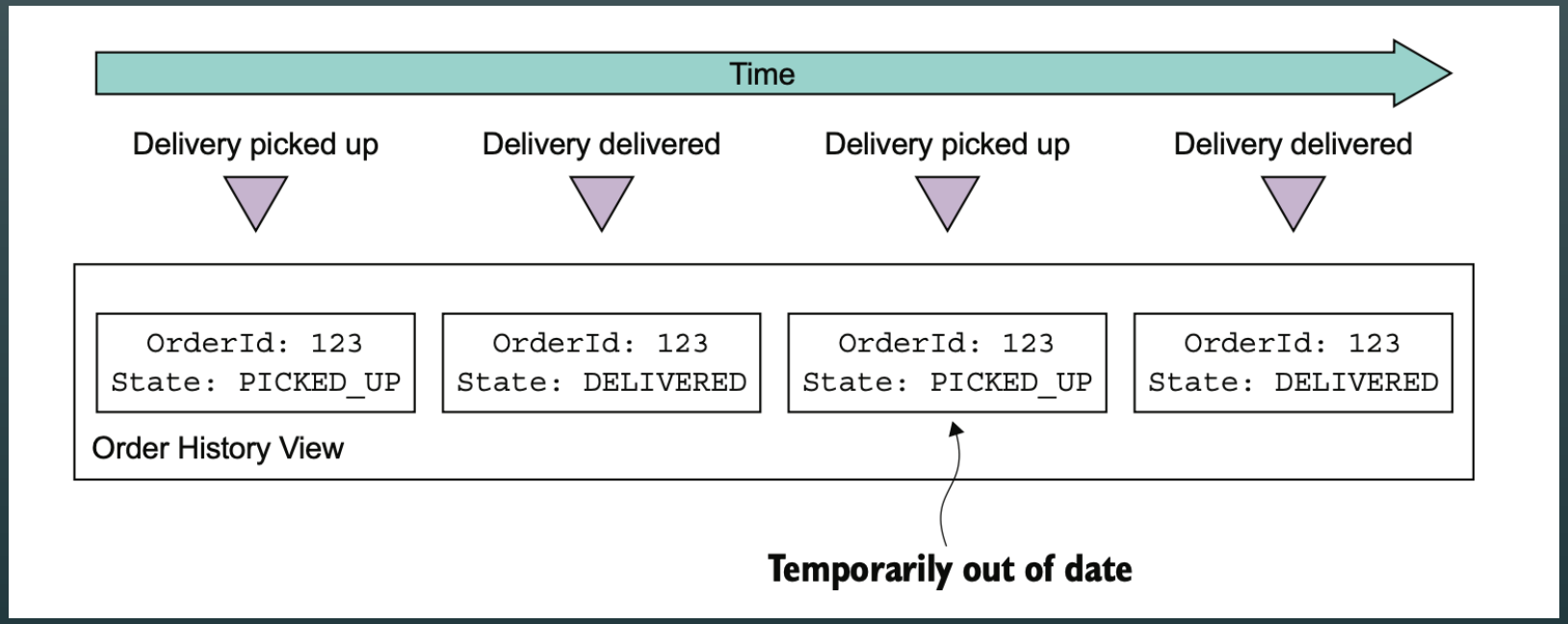
- Implement the efficient update operations executed by the event handlers.
  - Using the primary key.
  - Sometimes, using foreign key.
- Some types of database efficiently support foreign-key-based update operations.
  - RDBMS or MongoDB can create an index on the necessary columns

Data access module design (1)

Key issue 1 – Handling concurrency

- The event handlers use the data access module, consisting of a data access object (DAO).
- DAO must handle concurrent updates to the same database record.
  - If a view subscribes to events published by multiple aggregate types,
  - It must not allow one update to overwrite another.

Data access module design (2)



Data access module design (3)

Key issue 1 (cont.) – Idempotent event handlers

- An event handler may be invoked with the same event more than once.
  - Not a problem if a query-side event handler is idempotent
- A non-idempotent event handler must detect and discard duplicate events
  - Recording the IDs of processed events in the view datastore.
  - Recording the event ID and update the datastore atomically.

Data access module design (3)

Key issue 2 – Enable a client to use an eventually consistent view

- A client that updates the command side and then immediately executes a query might not see its own update.
- The command and query module APIs can enable the client to detect an inconsistency
  - A command-side operation returns a token containing the ID of the published event to the client.
  - The client then passes the token to a query operation, which returns an error if the view hasn't been updated by that event.

Adding and updating CQRS views

- Sometimes you need to add a new view to support a new query.
- At other times you might need to re-create a view
  - Because the schema has changed.
  - Or you need to fix a bug in code that updates the view.
- To create a new view,
  - Develop the query-side module → set up the datastore,→ deploy the service
- To update an existing view
  - Change the event handlers → rebuild the view from scratch or update incrementally

Summary (1)

- Implementing queries that retrieve data from multiple services is challenging because each service's data is private.
- There are two ways to implement these kinds of query:
  - The API composition pattern
  - Command query responsibility segregation (CQRS) pattern
- The API composition pattern is the simplest way to implement queries and should be used whenever possible.
- A limitation of the API composition pattern is that some complex queries require inefficient in-memory joins of large datasets.

Summary (2)

- The CQRS pattern, which implements queries using view databases, is more powerful but more complex to implement.
- A CQRS view module must handle concurrent updates as well as detect and discard duplicate events.
- CQRS improves separation of concerns by enabling a service to implement a query that returns data owned by a different service.
- Clients must handle the eventual consistency of CQRS views.