Microservices

Persistence

Content

1. Overview and Patterns
2. Message-Broker
3. Caching
4. Event Sourcing / CQRS / Saga / Event Storming

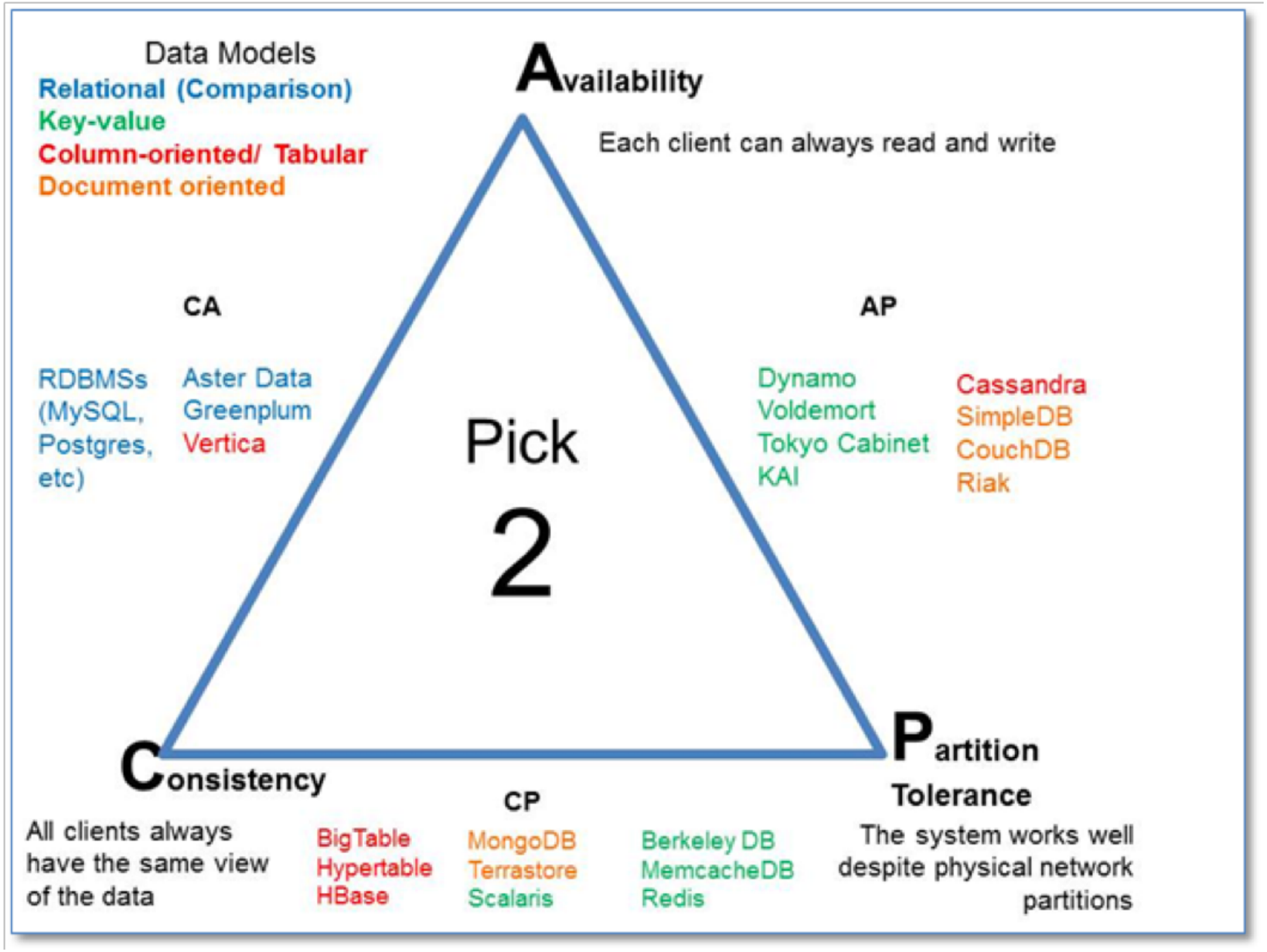
People try to copy Netflix, but they can only copy what they see. They copy the results, not the process.

— Adrian Cockcroft, former Chief Cloud Architect, Netflix

Data is the hardest part in microservices

- CRUD (Create, Read, Update, Delete) is often not enough for microservices
- You cannot do ACID (atomicity, consistency, isolation, durability) over multiple datasources transactions
- Better use BASE (Basically Available, Soft state, Eventual consistency)
- Choose the best for each service

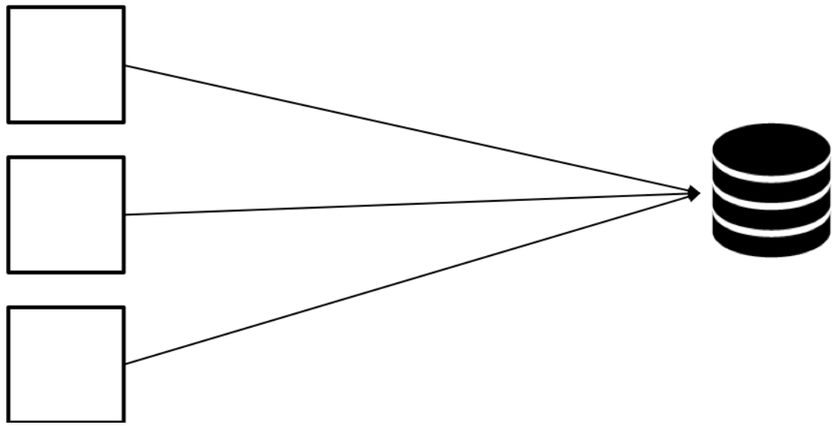
CAP


Data Models
Relational (Comparison)
Key-value
Column-oriented/ Tabular
Document oriented
Availability
Each client can always read and write
CA
RDBMSs (MySQL, Postgres, etc)
Aster Data
Greenplum
Vertica
Pick 2
AP
Dynamo
Voldemort
Tokyo Cabinet
KAI
Cassandra
SimpleDB
CouchDB
Riak
Consistency
All clients always have the same view of the data
Partition Tolerance
The system works well despite physical network partitions
CP
BigTable
Hypertable
HBase
MongoDB
Terrastore
Scalaris
Berkeley DB
MemcacheDB
Redis

Shared Datebase (Anti-Pattern)

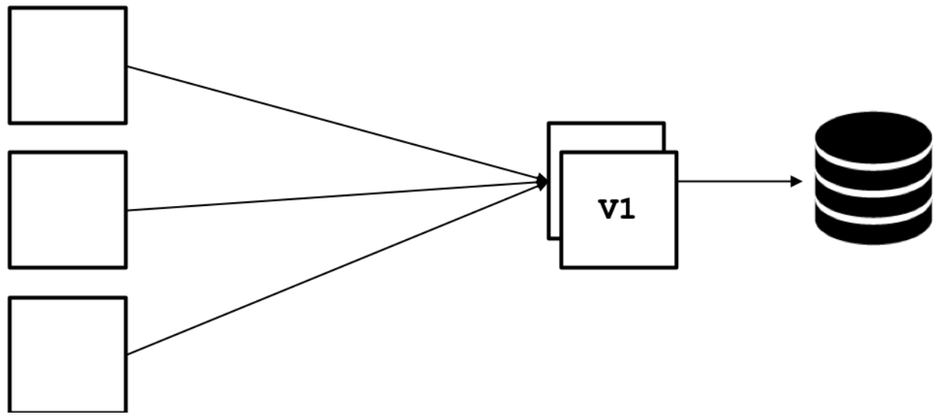
- While microservices appear independent, transitive dependencies in the data tier all but eliminate their autonomy.

Updates --> Locks --> Contention! --> BLOCK!!!



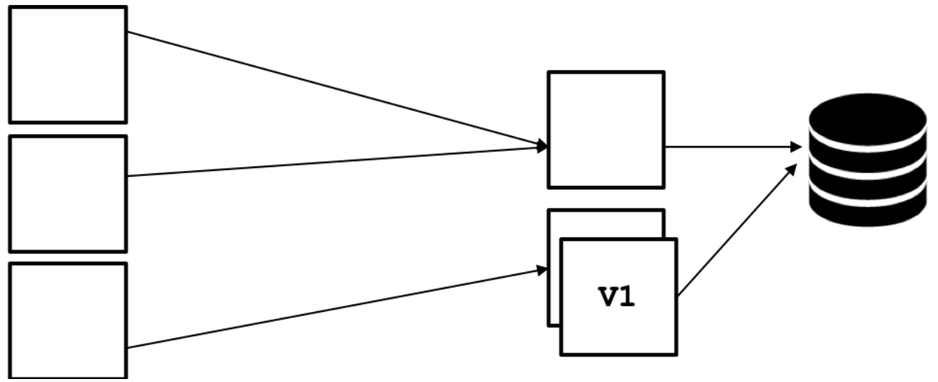
Data API

- Microservices do not acces data layer directly
- Expect for the microservice that implement the data API
- A Surface area to Implement access control, implementing throttling, perform logging and other policies
- Possibly coupled with Parallel deployment



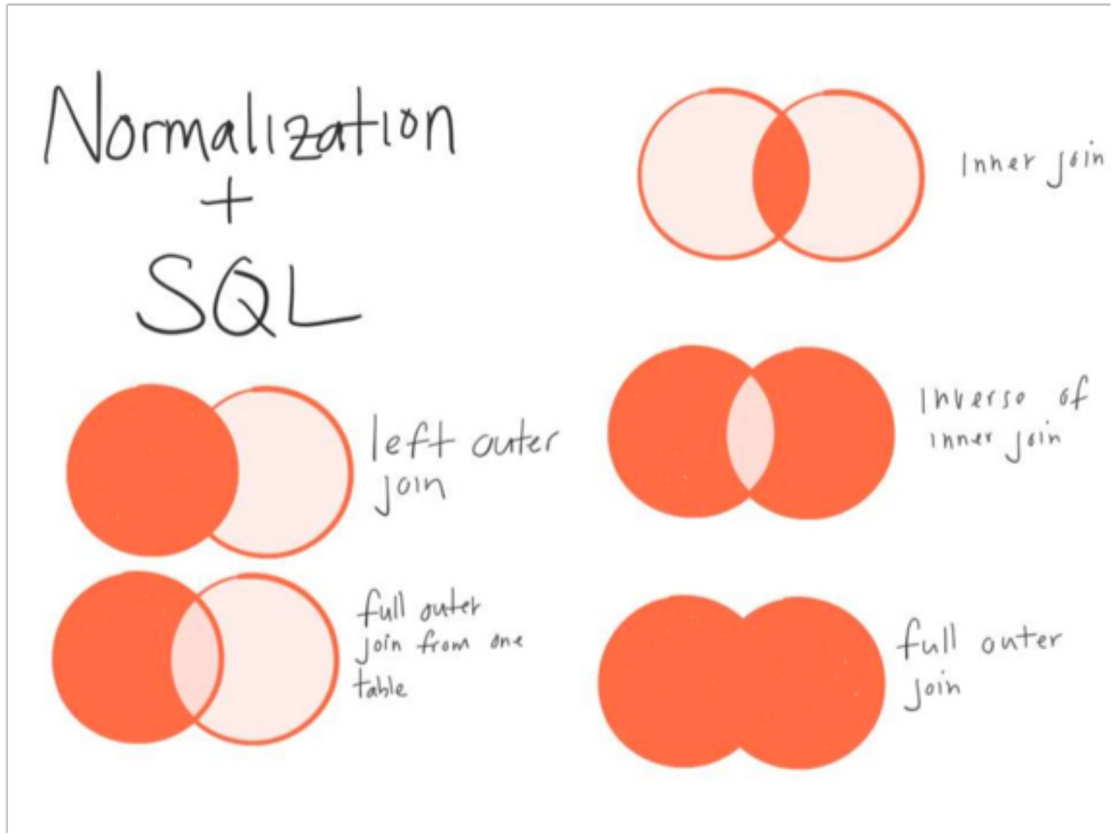
Bounded Context (Focus on Domain, not Data)

- Domain-Driven-Design
- Each bounded context has a single, unified model
- Relationships between models are explicitly defined
- A product team usually has a strong correlation to a bounded context
- Ideal pattern for Data APIs – do not fail into the trap of simply projecting current data models
- Model transactional boundaries as aggregates



Database per Service and Client Side Joins

- Support polyglot persistence
- Independent availability, backup/ restore, access patterns, etc.
- Services must be loosely coupled so that they can be developed, deployed and scaled independently
- Microservices often need a Cache and/or materialized Views



Message Broker

Asynchronous Messaging

- Service must handle requests from the applications clients. Furthermore, services must sometime collaborate to handle those requests. They must use an inter-process communication protocol.
- Use asynchronous messaging for inter-service communication. Services communcating by exchanging messages over messaging channels.
- There are numerous of asynchronous messaging technologies (e.g. Apache Kafka, RabbitMQ)

Benefits

- Loose coupling since it decouples client from services
- Improved availability since the message broker buffers messages until the consumer is able to process them
- Supports a variety of communication patterns including request/reply, notifications, request/async response, publish/subscribe, publish/async response etc

Example: Kafka

- Distributed, partitioned, replicated commit log service
- Pub/Sub messaging functionality
- Created by LinkedIn, now an Apache open-source project
- Fast
  - But helps with Back-Pressure (Fast Producer, Slow Consumer Problem)
- Resilient
  - Brokers persist data to disk
  - Broker Partitions are replicated to other nodes
  - Consumers start where they left off
  - Producers can retry at-least-once messaging
- Scalable
  - Capacity can be added at runtime wihout downtime
  - Topics can be larger than any single node could hold
  - Additional partitions can be added to add more parallelism

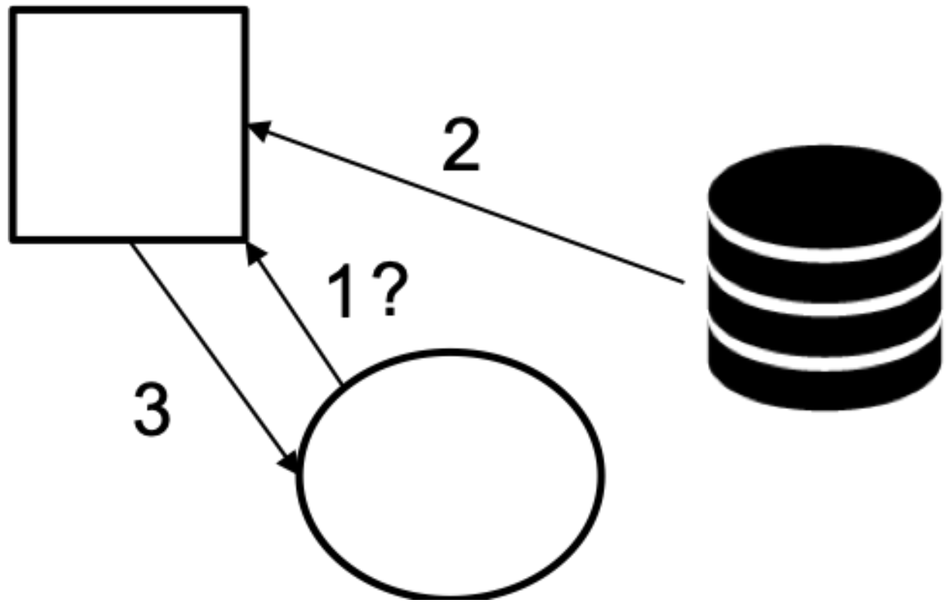
Caching

Caching Requirements

- Distributed
  - Over various failure boundaries AZ / Data Centers
- Data replication
  - Tunable consistency
- Available
  - Multi-node
  - Recovery process protects against any data loss
- Scalable
  - In-memory performance
  - Horizontally scalable
- Ease of Provisioning

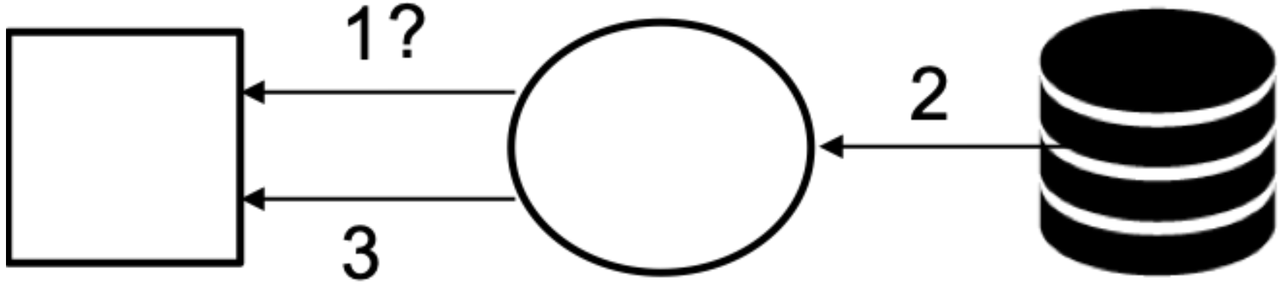
Caching (Look Aside)

- Attempt retrieval from cache
- Client retrieves from source
- Write into cache



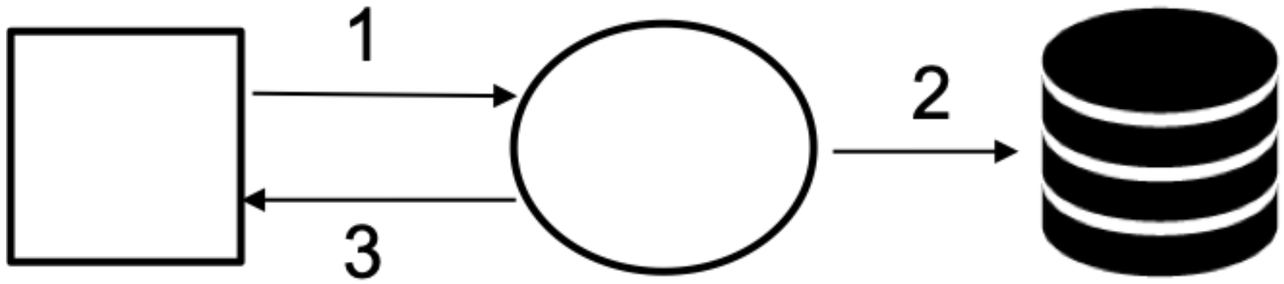
Caching (Read-through)

- Attempt retrieval from cache
- Cache retrieves from source and stores in cache
- Return value to client



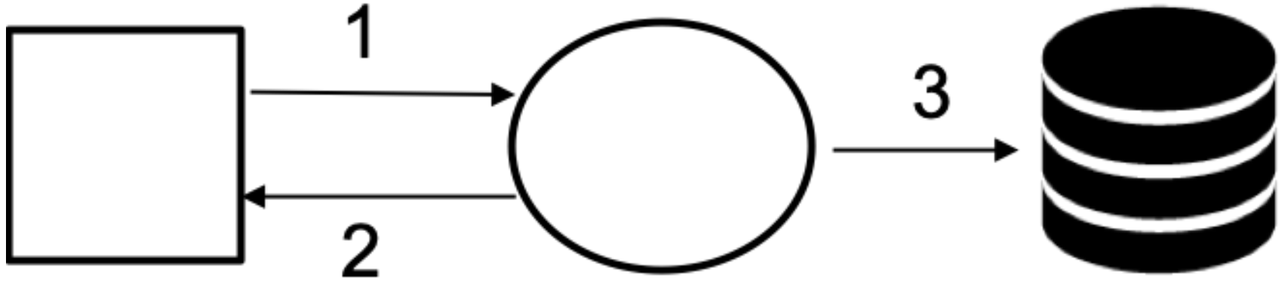
Caching (Write-through)

- Write to cache
- Cache writes to source
- Ack sent to client

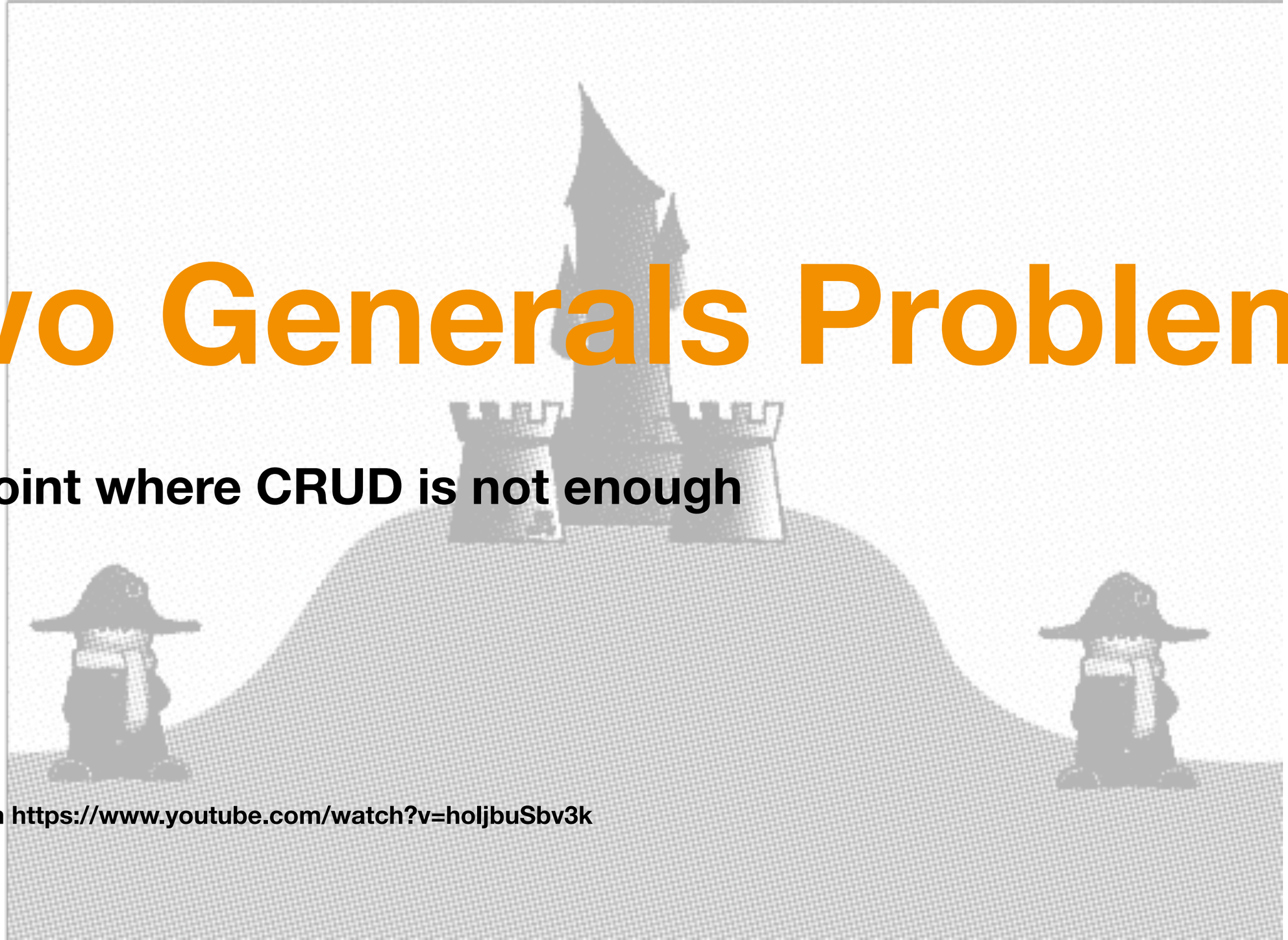


Caching (Write-behind)

- Write to cache
- Ack sent to client
- Cache writes to source asynchronously



Two Generals Problem [1]

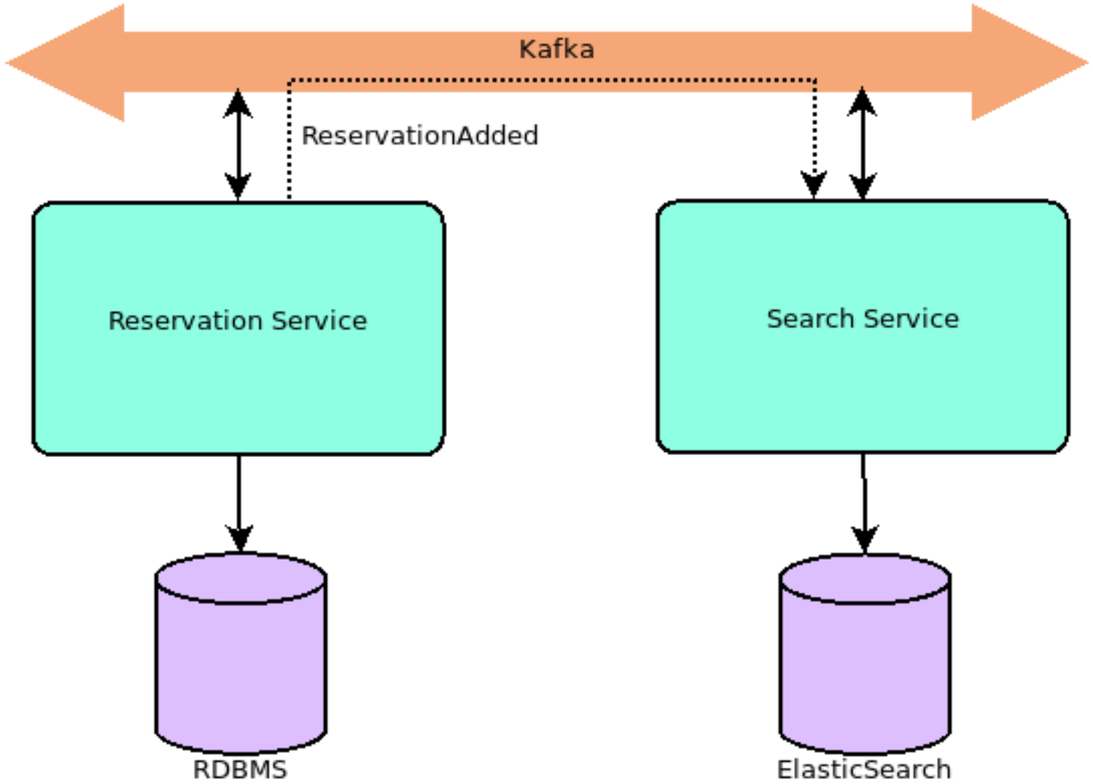


The point where CRUD is not enough

[1] Taken from https://www.youtube.com/watch?v=holjbuSbv3k

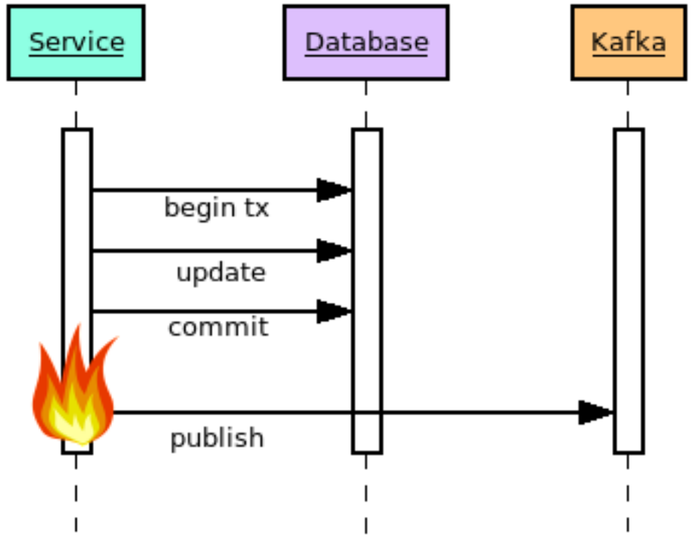
The Case: Holiday Rentrals

- AirBnb like application
- Asynchronous microservice architecture
- Kafka for messaging
- Reservation service uses CRUD persistence



Naive Approach

Update database and publish to Kafka



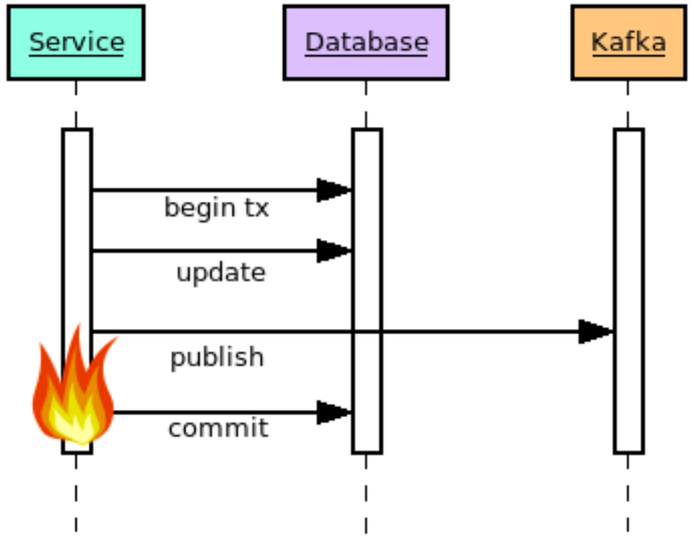
What if:

- **The service goes down?**
- **The database goes down?**
- **Kafka goes down?**
- **The network goes down?**


You want to search for a place to rent this summer?
Let me see...
Ah, here we are, a nice log cabin in the woods, not too far from the town, fantastic views, and it's free this summer!
No! it's not free this summer, I've taken a reservation for it!
Why didn't you tell me?
I tried, but Kafka was down!
This is what happens when you have two dependent write operations, but you don't do them atomically.

Another Solution?

We have just moved the problem. Now the search service thinks there is a reserveration, even when the reserveration wasn't complete.

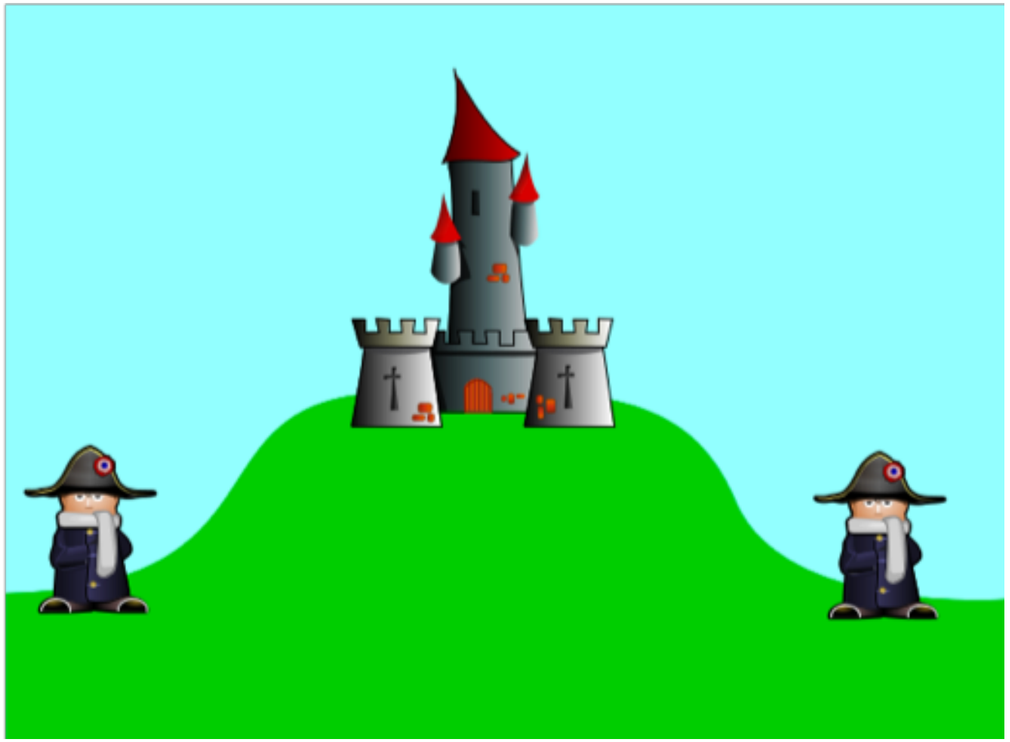


Two Generals Problem

The problem we have seen can be generalized as the 2 generals problem.

- **The 2 generals want to attack the city, but they can only win if they attack at the same time.**
- **They can just communicate by sending messages around the city, but there a city patrols around the city, which can intercept these messages.**
- **They need infinite acknowledgements**

This problem is prooven to be unsolveable!



We need a different plan to attack!

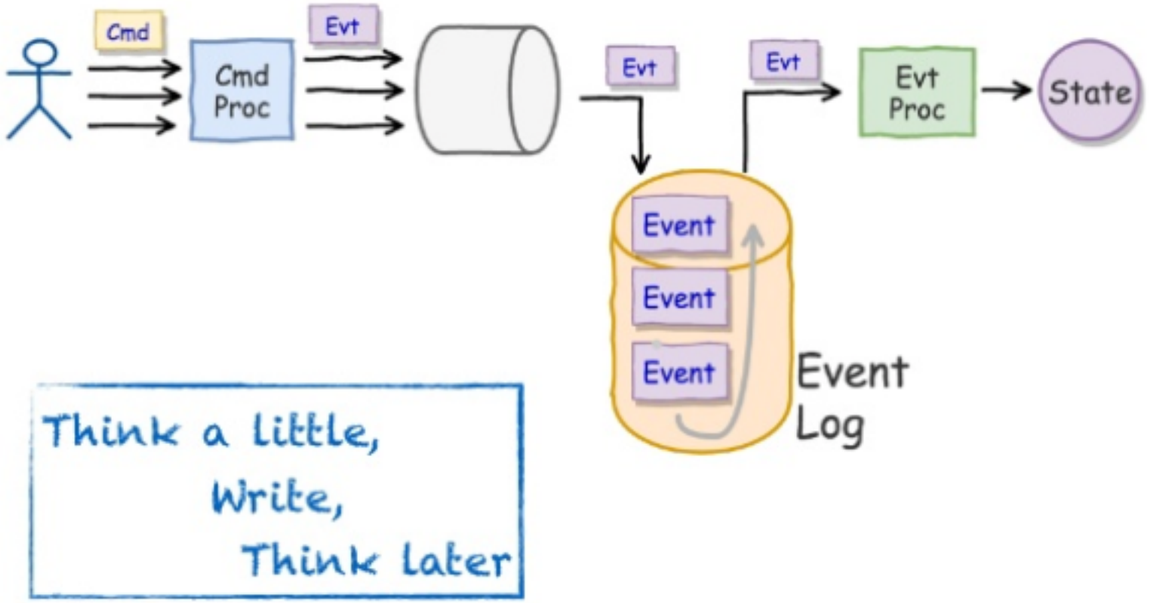
- We can't solve the 2 generals problem
- In different databases the application cannot simply use a local ACID transaction
- But we can come up with a different attack plan (BASE)

Event Sourcing

- Don't store the current state
- Store the events that occurred
- Compute the state from the events
- Avoid large numbers of events by saving snapshots

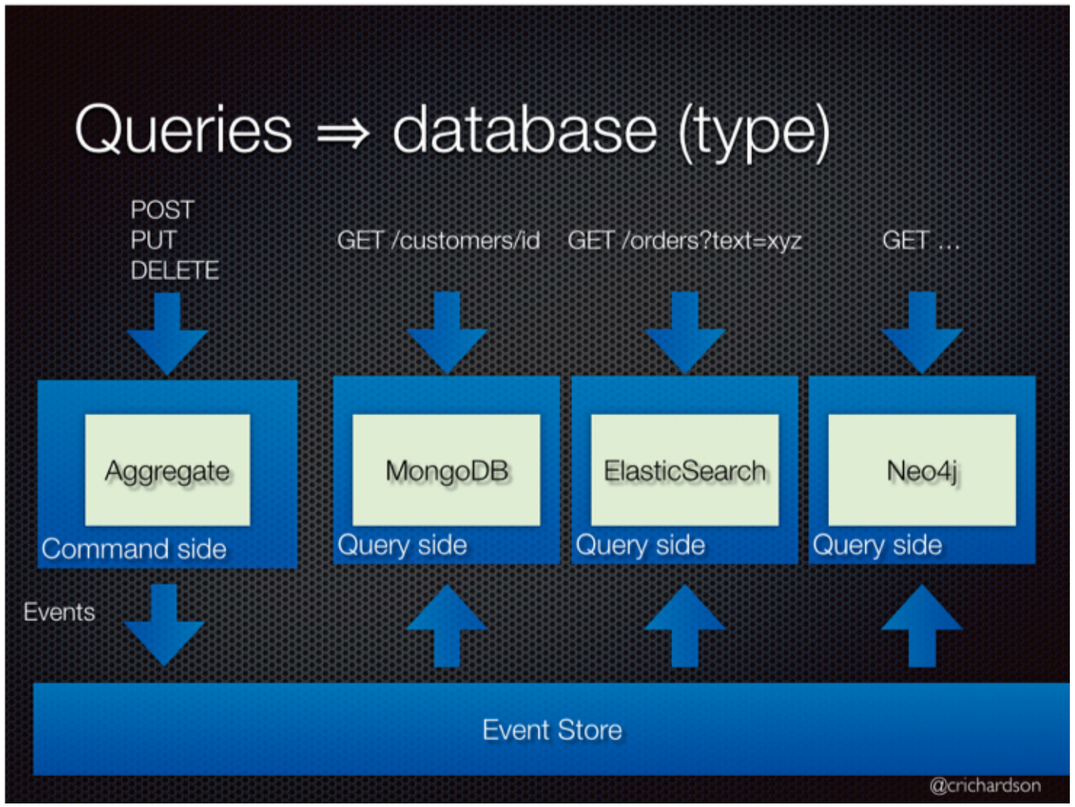
Main Benefits:

- Scalable, append only, fits distributed k/v stores, low-latency writes, allows asynchronous processing



Command Query Responsibility Segregation (CQRS)

Split the application into two parts: the command-side and the query-side. The command-side handles create, update, and delete requests and emits events when data changes. The query-side handles queries by executing them against one or more materialized views that are kept up to date by subscribing to the stream of events emitted when data changes.

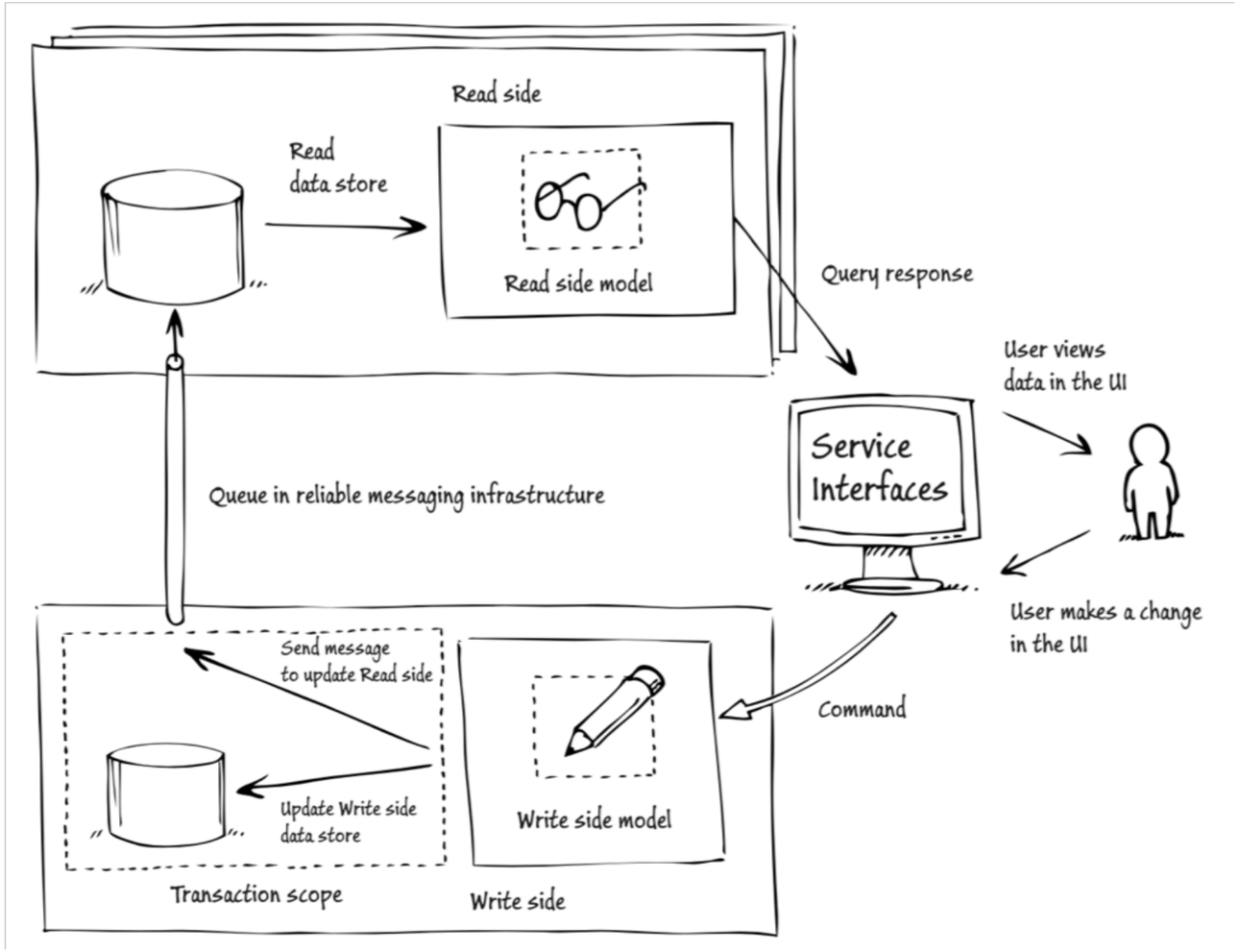


Event Sourcing and CQRS

- Your readside can be materialised Views in a RDBMS
- Or in a k/v store / cache
- Or in memory
- Or ...

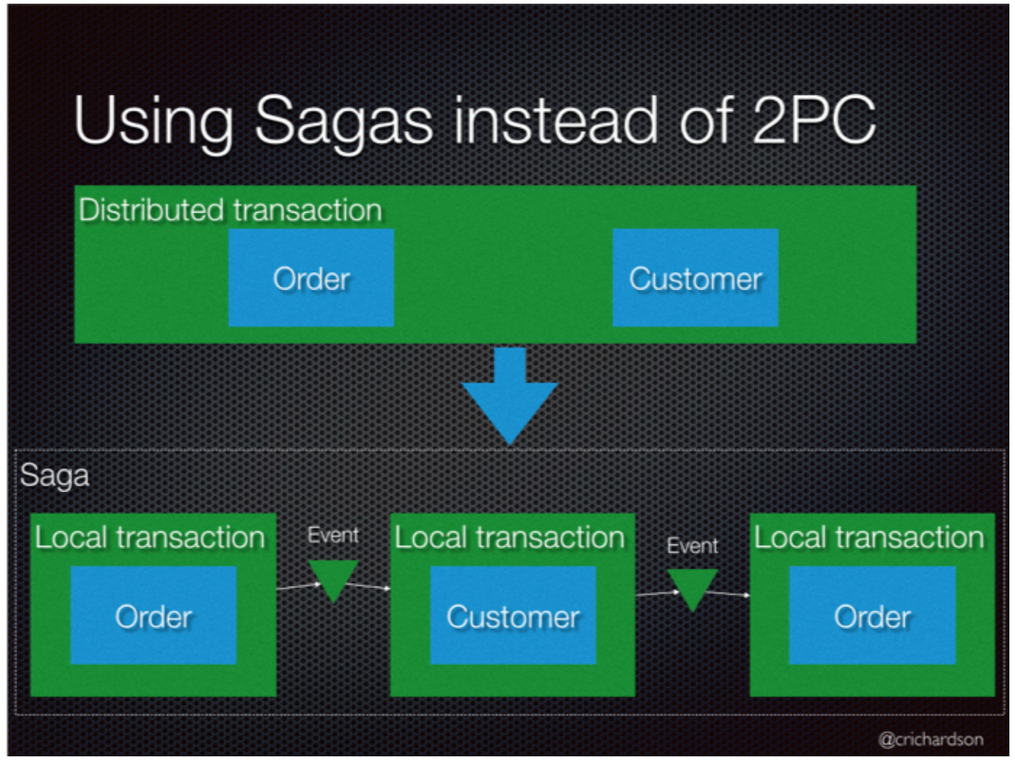
Views are optimised for specific query use cases. Multiple Views from same events. They are updated asynchronously and can be rebuilt from Events.

<u>Event Log is our Source of Truth</u>



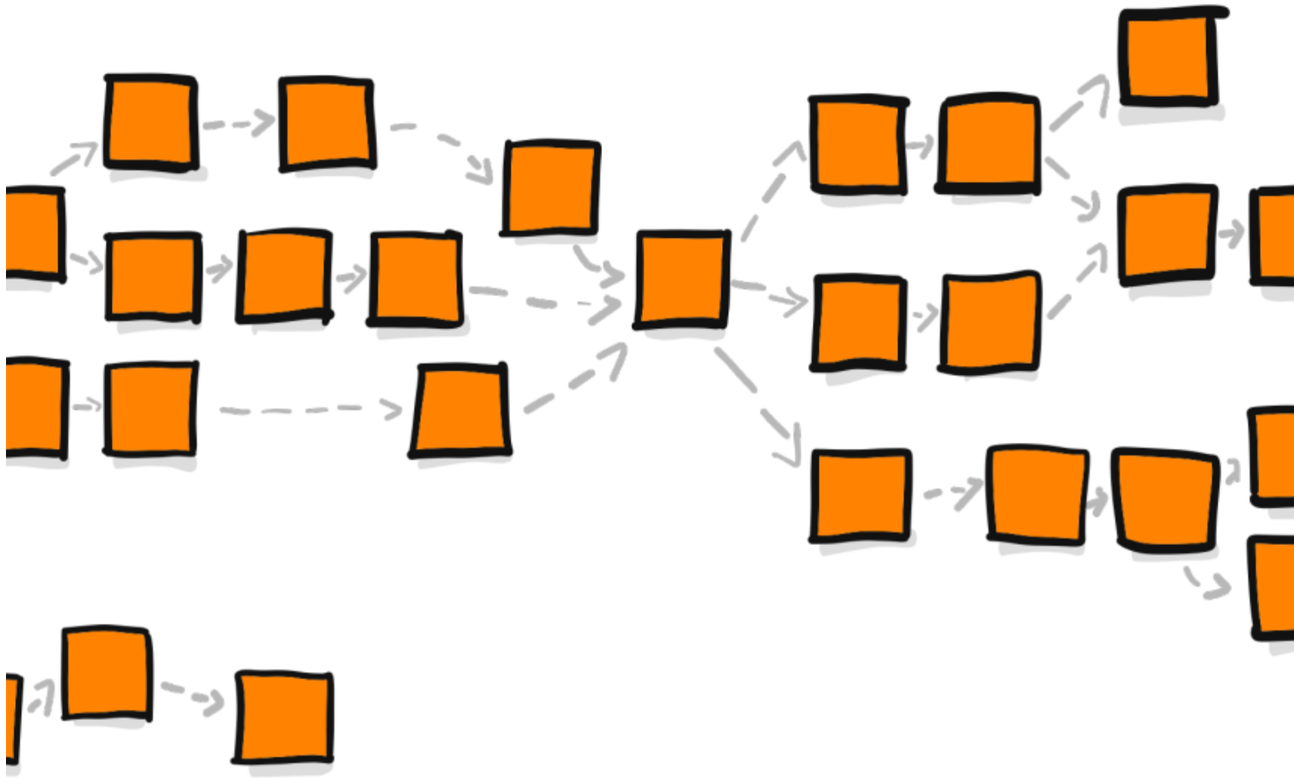
SAGA instead of 2 Phase Commit

Implement each business transaction that spans multiple services as a saga. A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.



How do we get our Domain Events? With Event Storming!

- Part of Domain Driven Design
- It is highly analog
- Steps:
  - Create domain events
  - add commands that caused the event
  - add an actor / user that executes the command
  - add corresponding aggregates



Interesting Videos and Sources

- Event Sourcing and Clustering: https://www.youtube.com/watch?v=2wSYcyWCtx4
- Two generals Problem: https://www.youtube.com/watch?v=holjbuSbv3k
- The hardest part of microservices is your data: https://www.youtube.com/watch?v=MrV0DqTqpFU