DAIMPL Praktikum - Local First: Daten-Synchronisierung mit Postgres-CDC

TECHNISCHE UNIVERSITÄT DARMSTADT

Fabian Oberthür, Erik Gelbing
25. September 2023

Inhaltsverzeichnis

1 Einleitung

Im Rahmen eines Praktikum an der TU Darmstadt haben wir uns mit Local-First Software auseinander gesetzt. Neben initialer Recherche und Diskussion über verschiedene Anwendungsbereiche und Problematiken haben wir dabei eine praktische Implementierung[1] einer Todo-Anwendung unternommen welche Local First Prinzipien einhalten soll.

Während es bereits fortgeschrittene Methoden zur Konfliktresolution gibt wie AutoMerge oder PouchDB, sollten wir uns die Frage stellen wie die Umsetzung von Local-First Paradigmen mit klassischen SQL Datenbanken funktionieren kann.

Eine vielversprechende Technologie zur Umsetzung einer solchen Anwendung erschien dabei die Verwendung sogenannter Change Data Capture (CDC) welche von vielen SQL Datenbanken wie MySQL oder PostgreSQL unterstützt wird.

Change Data Capture

Ein CDC-Event, kurz für Change Data Capture Event, dient als Mitteilung oder Informationseinheit, die Modifikationen in einer Datenquelle, meist einer Datenbank, festhält. Diese Events haben den Zweck, Veränderungen in der Datenbank nahezu in Echtzeit an andere Systeme und Applikationen zu kommunizieren.

Zur Implementierung von CDC bieten viele Datenbanksysteme spezielle Funktionen. In der Regel werden bei Aktivierung dieser Funktionen die Events in einer gesonderten Struktur, die einer Tabelle ähneln kann, abgelegt. Von hier aus können sie dann von anderen Systemen abgerufen werden.

Jedes CDC-Event enthält in der Regel sowohl Metadaten als auch die tatsächlich geänderten Daten. Dies gibt den Konsumenten dieser Events die Möglichkeit, den Kontext der Änderung zu erfassen und angemessen darauf zu reagieren.

[1] `https://github.com/gelbing/daimpl`

Für unser Praktikum haben wir uns vorgenommen eine Todo-Anwendung zu imlementieren. Diese soll lokal[2] unter Verwendung von SQL Datenbanken funktionieren, und dennoch kollaboratives Arbeiten mit anderen Teilnehmern ermöglichen. Dafür haben wir nach geeigneten Technologien gesucht (Abschnitt 3), deren Implementierung in einem Local First Szenario untersucht (Abschnitt **??** und uns mit einfachen Möglichkeiten einer Konfliktresolution beschäftigt (5).

2 Projektrecherche

Im folgenden Abschnitt werden wir uns mit der Projektrecherche befassen, welche kontinuierlich während des gesamten Projekts durchlaufen wurde. Die Recherche umfasst allgemeine Grundlagen und weiterführende Themen. Die Recherche ermöglicht einen intialen Überblick über die Grundprinzipen der Local-First-Idee und bildet die Grundlage für weiterführende Ideen.

Allgemeine Grundlagen

Die allgemeinen Grundlagen zum Thema haben wir auf der folgenden Website erhalten. `https://localfirstweb.dev/` Hier finden sich aktuelle Entwicklungen und Einführungen zu dem Thema. Als Einstieg in das Thema der Local First Software und deren Motivation bietet sich der Artikel **Local First Software - You own your data, in spite of the cloud** von der Website `https://www.inkandswitch.com/local-first/` an. Insbesondere die sieben genannten Local First Prinzipien haben uns einen guten Einblick in das Thema und die dahinter stehenden Ideen vermitteln können.

Desweiteren haben wir Informationen über konfliktfrei reproduzierten Daten gesammelt. Diese stellen die Grundlegenden Datentypen für Lokal First Software dar. Diese **CRDTS**[3] werden verwendet, um Daten in verteilten Systemen zu speichern und zu synchronisieren, der Fokus liegt dabei vorallem auf der Konfliktvermeidung und der Konsistenzgewährleistung in dezentralen Umgebungen. Wir haben wertvolle Informationen für unser Verständnis von Konfliktresolution gewinnen können, besonders durch Betrachtung von AutoMerge[4] und anderen Baumstrukturen wie LSEQ[5] Sie ermöglichen die gleichzeitige Bearbeitung von Daten durch mehrere Knoten ohne zentrale Koordination. Dabei haben wir auch fesstellen können das primitivere Methoden der Konfliktresolution existieren wie Observe-Remove Sets oder Last-Write-Wins Strategien. Letztere haben wir für unsere Anwendung umgesetzt.

Weiterführende Themen

Während des Projektes haben wir immer wieder Anregungen erhalten zu gewissen Technologien oder Artikeln. Allerdings haben diese für unser Praktikum meist entweder eine zu hohe Komplexität, thematisch nur wenig Bezug, oder einen hohen Anteil benötigter Modifikation zum aktuellen Projektstand. Beispiele für diese Anregungen sind

FinScan Web Services API

Während des Projekts haben wir uns **FinScan Web Services API** `https://api.innovativesystems.com/` angeschaut. Dabei handelt es sich um eine Schnittstelle, welche eine nahtlose Integration von Compliance-Workflow in die internen Geschäftsprozess ermöglicht. Dies ist vorallem für die Einhaltung von Vorschriften relevant. Zudem ermöglicht es ein Echtzeitscreening. Die API verarbeitet gegebene Input Parameter wie Name, Adresse, Geburtsdatum ...) und aggregiert aus mehreren Datenquellen Compliance Informationen zu gegebener Person.

Northwind Database

Microsoft Northwind[6] ist eine fiktive Unternehmensdatenbank. Die Datenbank enthält verschiedene Tabellen, Abfragen, Formulare und Berichte, die typische Geschäftsprozesse simulieren. Beispielsweise enthält sie Tabellen für Kunden, Lieferanten, Produkte und Bestellungen. Sie bietet eine breite Palette an Daten und Beziehungen zwischen den Daten, sodass Benutzer eine Vielzahl von Abfragen und Analysen durchführen können.

TPC-C

TPC-C ist ein anerkannter Industriestandard-Benchmark, der die Leistung von relationalen Datenbanksystemen in Bezug auf Transaktionsverarbeitung misst. Wie die Northwind-Datenbank bringt auch TPC-C ein eigenes Datenmodell mit, das eine Reihe von Geschäftsprozessen simuliert, um realistische Tests zu ermöglichen. Die Leistung wird üblicherweise in "Transaktionen pro Minuteängegeben und bietet einen nützlichen Vergleichsmaßstab für verschiedene Datenbanksysteme.

[2] lokal: Kein Internetzugriff bzw. keine online REST-API notwendig um anwendung zu verwenden und mit der Datenbank zu interagieren
[3] `https://vlcn.io/blog/intro-to-crdts`
[4] `https://automerge.org/`
[5] https://www.youtube.com/watch?v=M8-WFTjZoA0
[6] Beispiel Postgres Implementierung: `https://github.com/pthom/northwind_psql`

3 Verwendete Software

3.1 Debezium

Debezium ist ein Open-Source-Change-Data-Capture (CDC) Tool, das Datenbankänderungen in Echtzeit erfasst und an Apache Kafka weiterleitet. Es ermöglicht Anwendungen, auf Datenänderungen in Datenbanken zu reagieren, ohne eine zusätzliche Belastung für diese Systeme zu verursachen.

Die für uns wichtigsten Elemente eines CDC-Event von Debezium in Kafka sind:

- **before:** Zeigt den Zustand des Datensatzes vor der Änderung.
- **after:** Zeigt den Zustand des Datensatzes nach der Änderung.
- **source:** Informationen über die Herkunft der Änderung, einschließlich Datenbank, Tabellenname, Zeitstempel der Änderung usw.
- **op:** Der Typ der durchgeführten Operation – hinzufügen (create), ändern (update) oder löschen (delete).

3.2 Kafka

Apache Kafka ist eine verteilte Streaming-Plattform, die es ermöglicht, Daten in Echtzeit zu verarbeiten, zu speichern und zu analysieren. Kafka kann als Message Queue oder als Event-Streaming-Plattform verwendet werden.

Ein Kafka-Record wird typischerweise von Produzenten erstellt und von Konsumenten verarbeitet.

Debezium ist dafür entwickelt wurden in Kombination mit Kafka verwendet zu werden. Debezium kann hierbei sowohl also Produzent agieren (in unserer Anwendung), aber bietet auch spezielle Adapter um als Konsument, Daten aus Kafka zu verarbeiten. Davon machen wir in diesem Projekt aber keinen Gebrauch.

Ein Kafka besteht aus folgenden Bestandteilen. Wie diese beschrieben werden kann frei durch den Producer (hier Debezium) gewählt werden

- **Key:** Der Schlüssel eines Records. Debezium gibt hier Datentyp und Spaltenname des / der modifizierten Primärschlüssel (PK) an.
- **Value:** Der Hauptinhalt der Nachricht. Es handelt sich um die eigentlichen Daten, die an die Konsumenten gesendet werden sollen. In unserem Fall, das CDC Event (siehe Abschnitt 3.1)
- **Offset:** Der Offset wird verwendet, um den genauen Ort einer Nachricht innerhalb der Partition zu identifizieren. Konsumenten verwenden den Offset, um den Fortschritt des Nachrichtenverbrauchs zu verfolgen und bei Bedarf die Verarbeitung an einem bestimmten Punkt fortzusetzen.
- **Timestamp:** Das Datum und die Uhrzeit, zu denen die Nachricht produziert wurde. Wir verwenden diesen für unsere Anwendung aber nicht sondern nutzen ein automatisch generierten Zeitstempel innerhalb der Relation.

Auch wenn wir durch Debezium als gewählte Technologie ohnehin gezwungen sind auch Kafka zu verwenden, bieten besonders die Offsets in einer Event Streaming Plattform wie Kafka für unsere Anwendung Vorteile.

Durch diese können unsere Konsumenten ihre Aufgabe auch bei Verbindungsverlust dort fortsetzen wo sie zuletzt steh geblieben sind.

Außerdem sorgt Kafka für eine inhärente Ordnung der Einträge. Auch wenn aus dieser Reihenfolge nicht die semantisch korrekte Ordnung der CDC-Events abgeleitet werden kann hilft dies in unserem Beispiel dabei eine Deterministische Merge-Strategie zu entwickeln.

Auch zum Buffering von Nachrichten bietet Kafka entscheidende Vorteile. Dazu finden sich noch Anmerkungen in Abschnitt 6.

4 Architektur

Unsere Todo-Anwendung besteht wie in Abschnitt 3 bereits angedeutet, aus vielen einzelnen Komponenten welche gemeinsam die gewünschte Funktionalität realisieren.

Neben den in Abschnitt 3 angesprochenen, zentralen Bausteinen, finden sich noch weitere Services in unserer Architektur.

Eine Darstellung dieser Architektur ist in Abbildung 1 zu sehen. Für den weiteren Verlauf dieser Dokumentation möchten wir eine Gliederung der Bestandteile unserer Anwendung in zwei Hauptkomponenten vornehmen. Den auf der rechten Hälfte der Grafik sichtbaren Block, bestehend aus unserer Kafka Instanz und unserem Sink[7] beschreiben wir im weiteren Verlauf als *Server*. Auf der

[7] Kafka Consumer der Elemente aus einem oder mehreren Topics ließt und diese weiterverarbeitet

Linken Bildhälfte, sind mehrere sogenannte Client's zu erkennen und ein Frontend welches außerhalb liegt. Für den weiteren Verlauf dieses Dokumentes sind aussagen welche einen *Client* betreffen, universell auf alle Clienten übertragbar da diese sich die identische Konfiguration teilen. Fälle in denen wir unterscheiden welche Rolle ein Client gerade einnimmt werden wir explizit benennen. Wenn wir von einem Client sprechen ist ebenfalls auch immer das Frontend inklusive in dieser Rolle, auch wenn dieses auf der Grafik separat dargestellt ist. Mehr dazu in Abschnitt 4.3

Im restlichen Teil dieses Abschnittes spezifizieren wir die einzelnen Technologien die wir verwenden um die abgebildete Architektur zu realisieren.

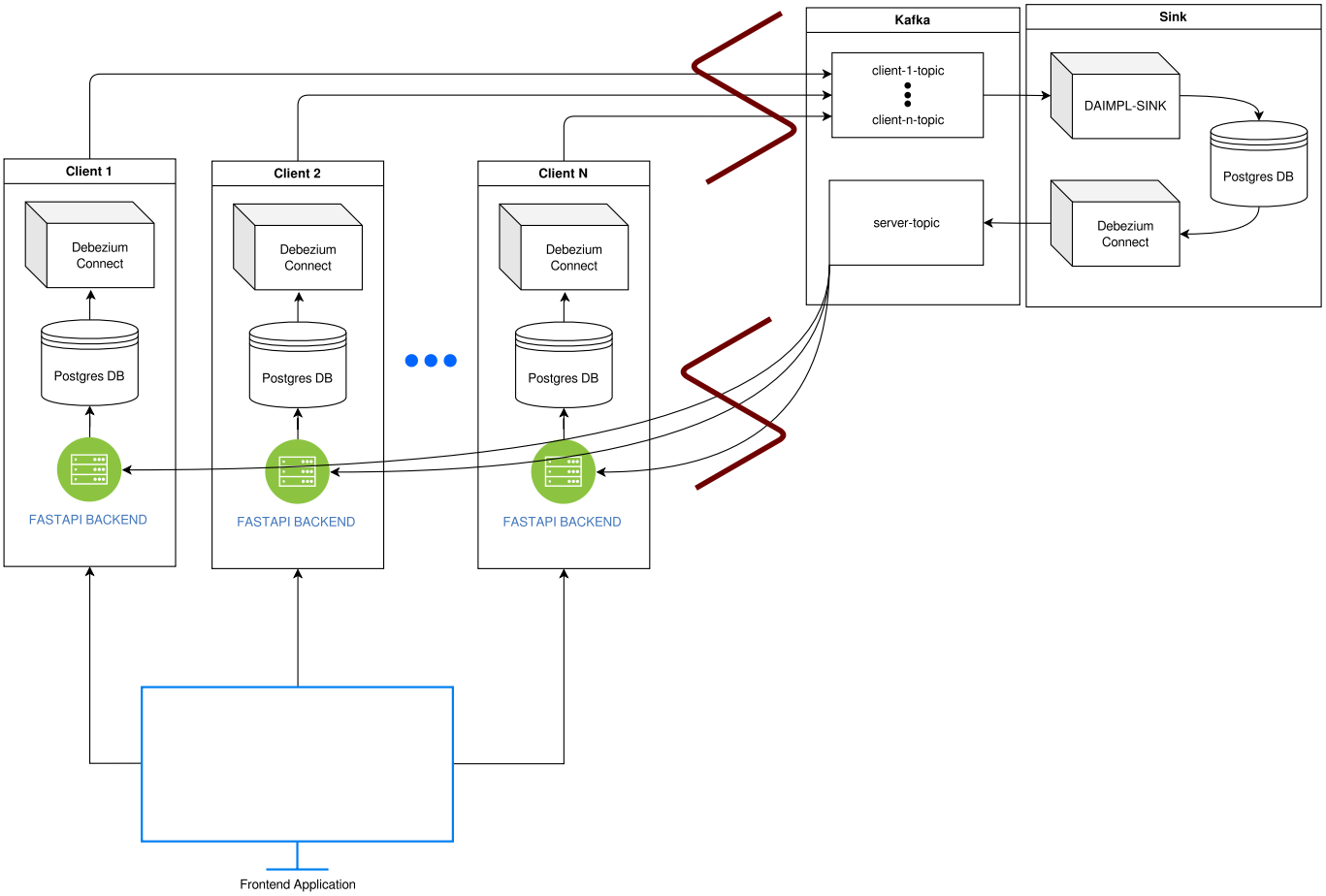

Abbildung 1: Architekturübersicht der Anwendung, Links sind mehrere Clients zu sehen und rechts die Kafka Instanz getrennt von Sink Postgres Datenbank und Debezium Connector. Diese bilden zusammen den Server

4.1 Server

Der Server stellt das zentrale Element der Anwendung dar und besteht aus den folgenden Einheiten:

- **Zookeeper:** Verantwortlich für die Verwaltung von Kafka-Clustern.
- **Kafka:** Übernimmt die Verwaltung der CDC-Events, welche von den Clients gesendet werden.
- **Redpanda-Console:** Ermöglicht über den Port 8080 die visuelle Darstellung und Verwaltung der Kafka-Topics im Webbrowser.
- **Postgres Datenbank:** Dient als zentrale Datenbank, in welche alle aktualisierten Daten geschrieben werden.
- **Daimpl-Sink:** Kafka Konsument. Liest fortlaufend die Updates aus den Kafka-Topics und schreibt diese in die zentrale Postgres-Datenbank.
- **Debezium Postgres Connector:** Generiert für jede Änderung in der zentralen Datenbank ein CDC-Event und leitet dieses an ein spezifisches Kafka-Topic weiter.

4.2 Client

Der Client repräsentiert den Endbenutzer und setzt sich aus den folgenden Einheiten zusammen:

- **Postgres Datenbank:** Hier werden alle CRUD-Operationen des jeweiligen Clients gespeichert.
- **Debezium Postgres Connector:** Für jede Änderung in der Datenbank wird ein CDC-Event generiert und an ein Client-Spezisisches Kafka-Topic weitergeleitet.
- **FastAPI Backend:** Dieses Backend nimmt CRUD-Anfragen vom Frontend entgegen und schreibt die Daten in die zugehörige Datenbank.
- Kafka Konsument. Liest wann immer eine Verbindung zu Kafka besteht, Änderungen aus dem Server-Topic und überträgt diese in die eigene Datenbank.

4.3 Streamlit Frontend

Unser Frontend stellt eine einfache einfache To-Do-Anwendung dar, über welche dem Nutzer ermöglicht wird, kleine Einträge bestehend aus einem Titel und einer Beschreibung zu erstellen, zu aktualisieren und zu löschen. Wir haben das Frontend in eine separate Komponente ausgelagert um die bereits recht komplexe Architektur ein wenig zu vereinfachen. In unserer Implementierung existiert lediglich ein Frontend, innerhalb dessen man per Drop-Down auswählen kann mit welchem Backend es interagieren soll (sieht Abbildung 1 unten links). Sämtliche durchgeführtewn CRUD-Interaktionen (Create, Read, Update, Delete) werden dann durch das spezifizierte Backend durchgeführt.

Auch wenn es erscheinen mag als wenn das Frontend eine separate Komponente sei, möchten wir darauf hinweisen dass wir dieses, auch im weiteren Verlauf dieser Dokumentation, als einen Bestandteil des Clients betrachten. Die Auslagerung in eine separate Komponente ist wie bereits beschrieben optional und bietet für Funktion der Anwendung keine Vorteile (mehr dazu in Abschnitt 6).

5 Implementationsdetails

Client

Unser Datenbank Schema verwendet einen zusammengesetzten Primärschlüssel (PK), der aus einer Sequenz-ID (vom Typ `integer`) und einer `client_id` (z. B. `client-1`) besteht. zusätzlich haben wir Felder für den Zeitpunkt der erstellung eines Eintrages sowie der letzten Bearbeitung dem Schema hinzugefügt.

Daimpl-Sink

Der `daimpl-sink` liest kontinuierlich alle Client-Topics, identifiziert die durchzuführende Operation und aktualisiert anschließend die zentrale Postgres-Instanz. Hierbei wird eine einfache Last-Write-Wins (LWW) Strategie angewendet. Dabei wird der Client-seitig gesetzte Zeitstempel verwendet.

Synchronisierung (Integriert in Client Backend)

Im Client Backend Container läuft parallel zum Fastapi-Backend ein einfacher Synchronisations Job welcher das selbe Interface verwendet wie der daimpl-sink. Der Sync Job reagiert auf Änderungen im Server-Topic und schreibt diese nach einer festgelegten Strategie in die Datenbank.

Code-Struktur und Funktionsweise

Unsere Implementierung betont die Erweiterbarkeit durch abstrakte Klassen und Python-Protokolle. Die Datei `consumer.py` steuert das Lesen aller Client-Topics und ruft den `MessageHandler` aus `handler.py` auf. Der `MessageHandler` identifiziert die CDC-Event-Operation und ruft die zugehörige Methode der gewählten Strategie (z.B. LWW) auf. Schließlich definiert `strategy.py` die LWW-Strategie, eine Implementierung des `AbstractStrategy` Protokolls, das Methoden für Create-, Update- und Delete-Operationen bereitstellt.

5.1 Bemerkungen zur Implementierung

Die vorgestellte Implementierung adressiert speziell den Use-Case einer Todo-Anwendung. Clienten können Todos erstellen, aktualisieren, löschen und natürlich auch einfach nur lesen. Dabei nutzen sie immer ausschließlich ihre eigene Datenbank und niemals den direkten Zugriff auf die Server Instanz (siehe Abbildung: 1).

Wie im vorherigen Abschnitt bereits angedeutet findet dabei ein Spiel statt zwischen dem Client und dem Server. Wenn ein Client eine modifizierende Aktion (Create, Update, Delete) ausführt, wird immer ein CDC-Event ausgelöst welches an das entsprechende Topic gesendet wird. Selbiges gilt für Aktionen des Servers auf seiner eigenen Postgres Instanz. Da der Client wieder auf Events im Server Topic reagiert besteht hier die Gefahr in einer endlosen Schleife zu landen.

Um das zu verhindern nutzen wir die Zeitstempel der Events und gleichen ebenfalls die Objekte welche im CDC-Event mitgesendet werden mit der eigenen Datenbank ab. Wenn ein Event keine Änderung mehr propagiert so wird dieses als Synchronisation erkannt und verworfen. Ein detaillierter Flowchart zur Behandlung von Interaktionen ist in Abbildung: 2 zu sehen.

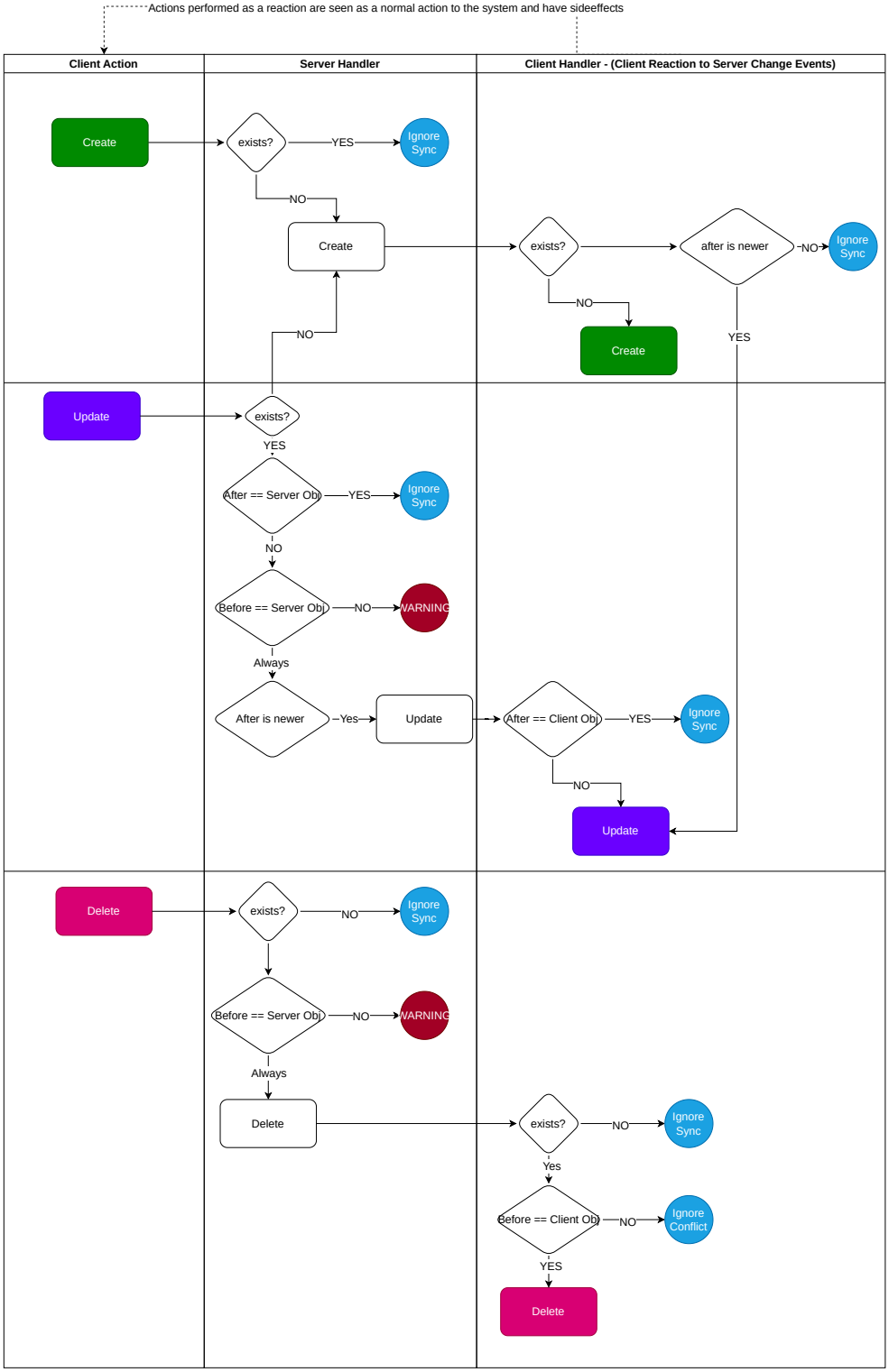

Abbildung 2: Flowchart für Create, Update und Delete Events durch den Client. Jede Modifikation einer der Datenbanken führt zur erneuten generierung eines CDC Events. Um eine endlose Schleife zu verhindern sind die Strategien auf Server und Client Seite so beschrieben dass sie diese Synchronisations-CDCs erkennen und verwerfen.

5.2 Beispiele anhand unserer Anwendung

Im Folgenden möchten wir genauer auf die Modifizierenden Datenbankaktionen (CREATE, UPDATE und DELETE) eingehen. Das zuvor bereitgestellte Flowchart Diagram (Abbildung 2) soll helfen den hier beschriebenen Abläufen besser folgen zu können.

Ausführliche Beschreibung anhand einer CREATE Aktion

1. Ein Client erstellt neuen Datenbank Eintrag mit einem Zeitstempel und einzigartigem Schlüssel
2. Server reagiert auf CDC-Event und erstellt diesen Eintrag ebenfalls in der eigenen Datenbank. Dabei wird das Objekt nicht modifiziert (Schlüssel und Zeitstempel werden übernommen)
3. Alle Clients registrieren die Änderung auf Server Seite.
   - Der Client welcher der ursprüngliche Autor war ignoriert die Änderung da ein identischer Eintrag bereits existiert
   - Die anderen Clients speichern das Objekt ebenfalls in ihrer Datenbank wobei sie den Eintrag ebenfalls nicht modifizieren
4. Der Server reagiert auf neu generierte Events durch nicht-Autoren und ignoriert diese da sie keine Änderung zum zuvor gespeicherten Eintrag darstellen

UPDATE und DELETE

UPDATE und *DELETE* laufen ähnlich ab. Den genauen Ablauf kann man dem Diagram in Abbildung: 2 entnehmen.

Besonderheiten bei Update sind dass der Server, falls das aktualisierte Objekt sich nicht mehr in der eigenen Datenbank befindet, stattdessen ein *CREATE* Befehl durchführt. Das kann passieren wenn ein *DELETE* zuvor durch einen anderen Client durchgeführt wurde und der Autor des Updates zum Zeitpunkt der Erstellung nicht synchronisiert ist. Clients übernehmen diesen *CREATE* wenn sie den Eintrag selbst zuvor gelöscht haben oder wenn sie in der Zwischenzeit selbst kein *UPDATE* vorgenommen haben. Durch die Last-Writer-Wins Implementierung können dabei Änderungen verloren gehen wenn mehrere Clients offline eine Modifikation durchgeführt haben.

Besonderheiten bei Delete sind dass wir durch unsere Implementierung keinen Zeitstempel des *DELETE* Events haben, da im CDC Event nur der Zeitstempel des gelöschten Objektes zu sehen ist. Wir haben uns entschieden dabei möglichst semantisch korrektes System zu bauen. Eine Löschung wird immer umgesetzt auf Server Seite (außer es handelt sich um eine Synchronisation). Clients ignorieren diese aber wenn sie beim herstellen einer Verbindung merken dass sie den zu löschenden Eintrag bereits editiert haben. In diesem Fall wird beim Herstellen der Verbindung bereits das UPDATE CDC Event gesendet welches den Eintrag erneut an alle Clients propagiert.

Auf diese Art haben nach Herstellung einer Verbindung alle Clients immer die selben Daten und das System befindet sich in einem geregelten Zustand. Wir haben versucht mit dieser Implementierung möglichst die User Intention und die Anforderungen an eine Todo-App abzubilden. Im Code werden sämtliche Konflikte als Warning geloggt um eine genauere Untersuchung zu ermöglichen.

6 Diskussion

6.1 Diskussion zur Implementierung

Bei der Integration unseres Ansatzes mit Legacy-Datenbanken wäre eine Umstrukturierung erforderlich, um einen zusammengesetzten Primärschlüssel zu implementieren. Die Wahl eines zusammengesetzten Primärschlüssels dient der Kollisionsvermeidung: Bei isolierter Verwendung einer einzigen ID könnten mehrere Clients potenziell identische IDs generieren.

Obwohl wir uns die Implementierung von Funktionen in einer externen Bibliothek vorstellen können, die Anpassungen an Legacy-Datenbanken überflüssig macht (oder zumindest reduziert), haben wir uns für das Projekt dafür entschieden, die Komplexität gering zu halten, indem wir Unterschiede in den Datenbankschemas zwischen Client und Server vermeiden. Diese Unterschiede müssten während der Synchronisation berücksichtigt werden. Insbesondere das Management unterschiedlicher IDs auf Client- und Serverseite könnte eine Herausforderung darstellen, die wir jedoch für lösbar halten, indem auf der Serverseite entsprechende Informationen über Herkunft und Identität eines Records gespeichert werden (ähnlich einer Lookup-Tabelle).

Die gesammelten Erkenntnisse aus unserem Prototypen können bei der Entwicklung einer generalisierbaren Bibliothek von Nutzen sein. Beispielsweise könnten Clients den Zeitstempel des Kafka-Records nutzen, statt ein separates `updated_on`-Feld in der Datenbank zu pflegen. Zudem könnten `source`-Informationen aus den Debezium CDC-Events für die Zuordnung zu spezifischen Clients herangezogen werden. Dies würde allerdings bedeuten, dass Unterschiede im Datenbankschema zwischen Client und Server während der Synchronisation berücksichtigt werden müssten.

Allgemein ist die Verwendung von Zeitstempeln eventuell suboptimal. Wenn es Unterschiede in den Systemuhren der verschiedenen Clients gibt, könnten diese Zeitstempel inkonsistent sein. Eine Implementierung, welche die CDC-Events in einem CRDT synchronisiert, könnte hier robuster sein, führt aber zu neuen Herausforderungen.

Ein weiterer Optimierungsvorschlag für zukünftige Implementierungen wäre, alle Clients in dasselbe Kafka-Topic schreiben zu lassen. Der Grund hierfür liegt in der Skalierbarkeit von Kafka: Bei einer Vielzahl individueller Topics für verschiedene Clients könnte die Performance leiden, bedingt durch die notwendigen Ressourcen des wachsenden Kafka-Clusters.

6.2 Allgemeine Diskussion

Verwendung von CDC-Events für Local First Apps

Die Nutzung von CDC-Events stellt einen innovativen Ansatz in der Entwicklung von Local First Apps dar. Change Data Capture (CDC) Technologien, wie sie von Debezium bereitgestellt werden, sind leistungsstark und bieten eine Fülle von Informationen, die nützlich sind, insbesondere wenn es darum geht, komplexere Systeme oder CRDTs zu implementieren. Ein Vorteil dieses Ansatzes ist, dass Entwickler keine eigene komplexe Middleware erstellen müssen. Stattdessen können sie sich auf bereits bestehende und bewährte Technologien verlassen. In unserer Implementierung konnten wir beispielsweise die Handhabung von Zeitstempeln und das unterschiedliche Verhalten beim Speichern von Einträgen, durch den Einsatz von generischen Klassen bewältigen. Durch die Kombination von Schema-Spezifikationen mit Pydantic und einer generischen CRUD-Klasse konnten wir ohne zusätzliche Logik das gewünschte Verhalten erreichen. Das ermöglicht es, den Fokus auf die eigentliche Konfliktresolution zu legen.

Generalisierbarkeit

Wir sehen durchaus Potential in der Verwendung von CDC-Events zum erstellen von Local-First Applikationen. Eine der Stärken dieses Ansatzes ist die Möglichkeit, komplexere Merge-Strategien zu implementieren, die auf einzelnen Spalten basieren, anstatt nur auf dem gesamten Objekt. Dazu können die Metadaten welche ein CDC-Event mit sich bringt hilfreich sein. Dennoch muss jeder Anwendungsfall individuell betrachtet werden.

Struktur unseres Ansatzes

Obwohl unser Ansatz Vorteile bietet, folgt er nicht strikt dem Local First Prinzip. Unsere Architektur behält eine Client-Server-Struktur bei. Eine spannende Betrachtung wäre eine Implementierung in einem Peer-to-Peer-Netzwerk, in dem es keine zentrale Autorität gibt. Dennoch ermöglicht unser Ansatz Arbeiten ohne ständige Verbindung, insbesondere wenn Konflikte minimal sind. Ein Nachteil unserer Implementierung ist der zusätzliche Overhead, der durch Nachrichten entsteht, die einen Round-Trip durchlaufen müssen. Außerdem werden die CDC-Events immer sequentiell verarbeitet. Ein Client der ein Dokument n mal editiert und zum Schluss die editierungen rückgängig macht, so dass wieder das ursprüngliche Dokument entsteht, löst trotzdem n CDC-Events aus. Optimierung durch ein buffering auf Serverseite könnte hier helfen um weniger Events durch Client Reaktionen auszulösen. Das Nutzen vieler bestehender Technologien reduziert den Implementierungsaufwand erheblich. Dies könnte durch die Bereitstellung einer Bibliothek, die den Entwicklungsprozess von Local First Apps vereinfacht, weiter optimiert werden. Der vorgestellte Ansatz erhöht jedoch auch die Abhängigkeiten der Anwendung. In unserem Fall sind dies Kafka, Postgres, Debezium sowie die Sink- und Sync-Implementierungen. Das Backend welches wir verwenden könnte man vermeiden zu nutzen, da ohnehin alles direkt auf dem Client ausgeführt wird und die Datenbank nicht online ist. Dennoch könnte ein solches Interface wertvoll sein, um Implementierungsfehler zu vermeiden.