KAPITEL 6: QUERY LANGUAGES

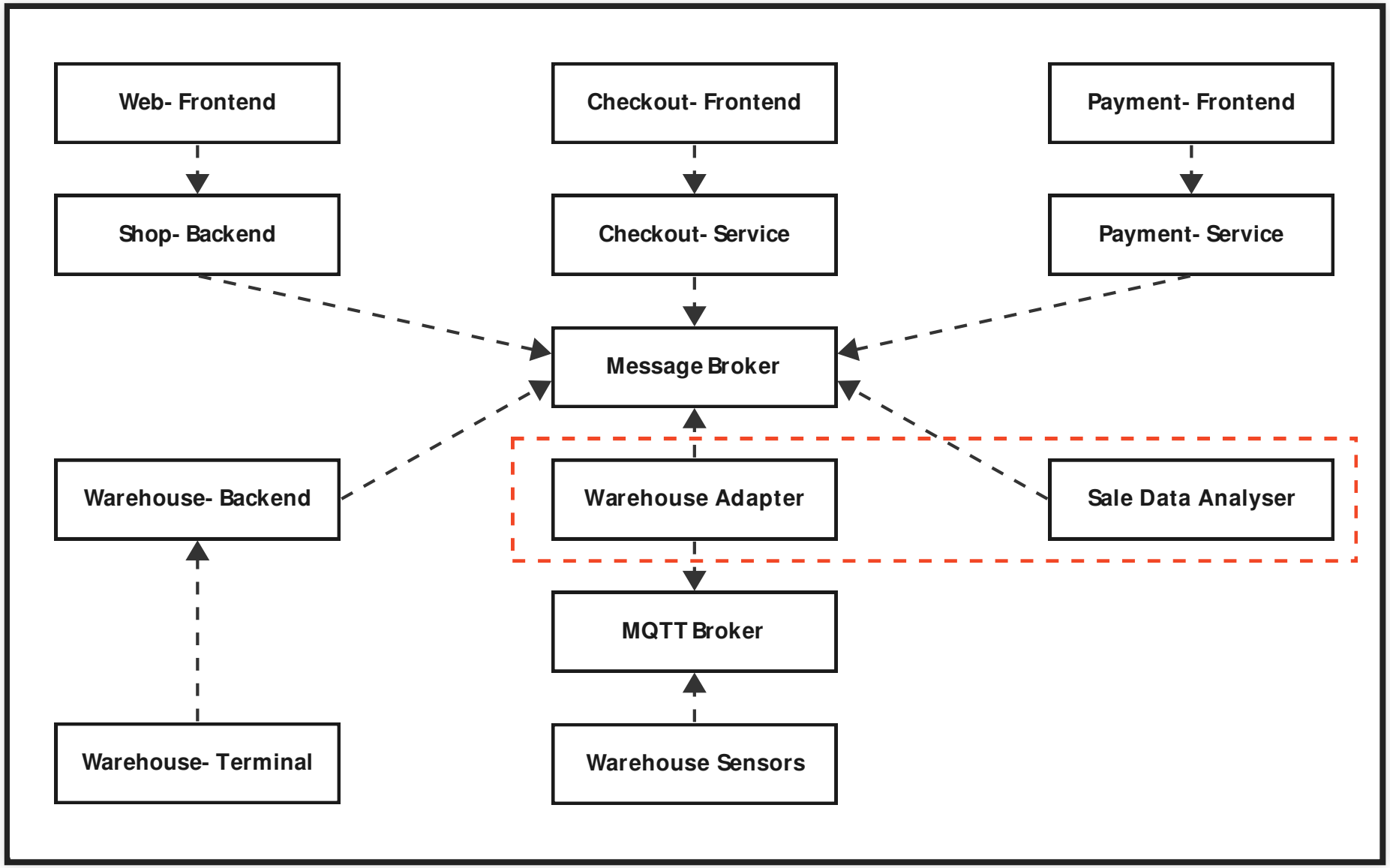
LERNZIELE

- Unterschiede zwischen SQL und ksqlDB erklären
- Grundlegenden Aufbau von DSL und DML Queries in ksqlDB erklären
- Vorgehen zur Verarbeitung von Datenströmen mit ksqlDB darstellen
- ksqlDB Queries und deren Bestandteile erklären sowie in Kafka Stream übersetzen können

6.1 MOTIVATION

Dienste zur Integration von Endpunkten oder die Verarbeitung von Datenströmen benötigte bisher eine eigenständige Implementierung, welche die Anbindung und die Verarbeitung der Daten sowie anschließende Bereitstellung der Ergebnisse in der Infrastruktur ermöglicht.

Beispiel Shop Scenario



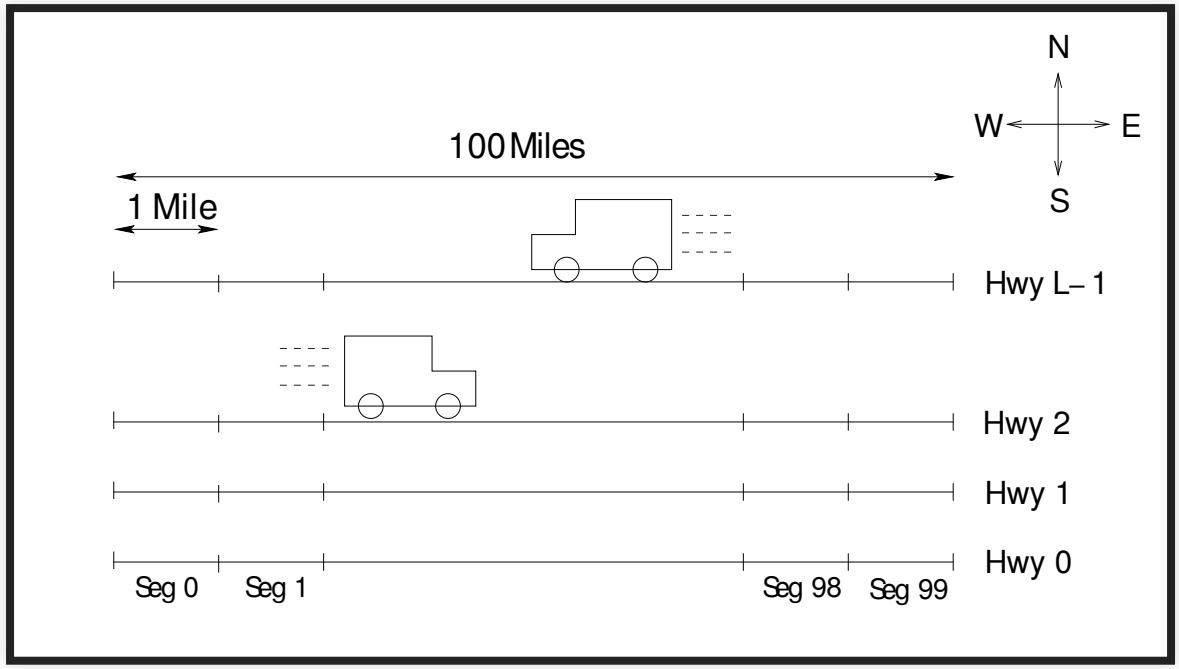
Mögliche Ansätze zur Umsetzung

- ... unter Verwendung von Client-Bibliotheken zur Anbindung der Endpunkte (HTTP, MQTT, Kafka), ggf. Verarbeitung der Ereignisse mit funktionionaler Programmierung
- ... mittels Apache Camel, Fokus i.R. auf Dienste zur Integration wie der Warehouse Adapter
- ... mittels Kafka Streams, Fokus i.R. auf Verarbeitung von Datenströmen wie bei einem Analysierer für Verkaufsdaten (Sale Data Analyser)
- ... mittels einer Datastreaming Platform wie Apache Flink, Fokus i.R. auf Verarbeitung von Datenströmen

In allen Ansätzen steht die Frage im Raum, wie die Datenquelle und -senke angebunden werden kann und wie Datentransformationen skalierbar und wartbar beschrieben werden können.

Continuous Query Language

Für die Integration von Diensten und die Verarbeitung von Datenströmen existieren Abfragesprachen, welche die Natur der endlosen Ereignisketten in einen SQL-ähnlichen Syntax überführen.




```
PosSpeedStr(21,130,0,east,A9)
PosSpeedStr(42,130,99,west,A9)
PosSpeedStr(21,130,1,east,A9)
PosSpeedStr(42,130,98,west,A9)
PosSpeedStr(21,130,2,east,A9)
PosSpeedStr(42,130,97,west,A9)
...
```

Quelle: Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. In The VLDB Journal, 2004. DOI: 10.1007/s00778-004-0147-z.

Beispiel zur Abfrage von Informationen mittels einer CQL

```
SELECT DISTINCT vehicleId FROM PosSpeedStr [Range 30 Seconds]
```

Diese Abfrage setzt sich zusammen aus einem Operator für ein sliding Window und einem Relation-to-Relation Operator für das entfernen von Duplikaten. Das Ergebnis dieser Abfrage enthält die Menge der "aktiven Fahrzeuge", d. h. jener Fahrzeuge, die innerhalb der letzten 30 Sekunden eine Positionsgeschwindigkeitsmessung übermittelt haben.

Quelle: Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. In The VLDB Journal, 2004. DOI: 10.1007/s00778-004-0147-z.

Abfragesprachen im Kontext von Datenströmen

- ESPER
- Siddhi (im Kontext von WSO2)
- KSQL, bzw. ksqlDB

6.2 KSQLDB

ksqlDB (ursprünglich KSQL) ist eine Open-Source-Event-Streaming-Datenbank, die 2017 von Confluent veröffentlicht wurde

Es wird die Art und Weise vereinfacht, wie Stream-Processing-Anwendungen erstellt, bereitgestellt und gewartet werden und eine High-Level-SQL-Schnittstelle für die Interaktion mit Kafka angeboten.

Unter der Haube werden Push-Abfragen in Kafka-Streams-Anwendungen kompiliert und sind ideal für ereignisgesteuerte Microservices, die Ereignisse schnell beobachten und darauf reagieren müssen.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Unteranderen umfasst dies:

- Modellierung von Daten als Streams oder Tabellen mit einer Art SQL
- SQL-Konstrukten, z. B. zum Verbinden, Aggregieren, Transformieren, Filtern und Windowing von Daten, um neue abgeleitete Daten zu erhalten
- Abfrage von Streams und Tabellen, die kontinuierlich ausgeführt werden und Ergebnisse an Clients senden, sobald neue Daten verfügbar sind
- Externe Datenquellen können über Kafka Connect eingebunden werden

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Zusammenhang zu SQL

Die SQL-Grammatik lehnt sich an ANSI SQL an, jedoch wurde ein spezieller SQL-Dialekt abgeleitet, um Daten sowohl in Streams als auch in Tabellen zu modellieren.

Herkömmliche SQL-Datenbanken befassen mit Daten in Tabellen, da es keine unendlichen Datenströme (Streams) gibt.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Dies äußert sich allgemein in zwei Arten von Anweisungen in klassischem SQL:

Klassische DDL (Data Definition Language)-Anweisungen konzentrieren sich auf das Erstellen und Löschen von Datenbankobjekten (in der Regel Tabellen, manchmal aber auch Datenbanken, Ansichten usw.):

```
CREATE TABLE users ...;
DROP TABLE users;
```

Die klassischen DML-Anweisungen (Data Manipulation Language) konzentrieren sich auf das Lesen und Bearbeiten von Daten in Tabellen:

```
SELECT username from USERS;
INSERT INTO users (id, username) VALUES(2, "Izzy");
```

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Gemeinsamkeiten zwischen ksqlDB und klassischem SQL

- *SQL-Schnittstelle:*
  enthält ksqlDB eine SQL-Grammatik, einen Parser und eine Ausführungsmaschine. Der SQL-Dialekt von ksqlDB enthält die Sprachkonstrukte, die aus SQL erwartet werden würden, einschließlich SELECT für die Projektion, FROM für die Definition von Quellen, WHERE für die Filterung, JOIN für die Verbindung usw.
- *DDL- und DML-Anweisungen:*
  DDL und DML sind die beiden großen Kategorien von Anweisungen, die sowohl in herkömmlichen SQL-Datenbanken als auch in ksqlDB unterstützt werden.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

- *Netzwerkdienst und Clients zur Übermittlung von Abfragen:*
  Es kann über ein Netzwerk mit der Datenbank eine Verbindung aufgebaut werden, und es gibt Standard-Client-Implementierungen (z. B. eine CLI) für die Übermittlung von Abfragen.
- *Schemas:*
  Die Sammlungen, mit denen interagiert wird, enthalten Schemadefinitionen, die Feldnamen und Typen enthalten, benutzerdefinierte Typen sind ebenfalls möglich.
- *Materialisierte Ansichten:*
  Zur Steigerung der Leseleistung sind materialisierte Ansichten verfügbar, ksqlDB ermöglicht dies für Streams und Tabellen, wenn neue Daten verfügbar sind

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

- *Eingebaute Funktionen und Operatoren für die Datentransformation:*
  Enthält eine Vielzahl von Funktionen und Operatoren für die Arbeit mit Daten, z.B. String-Funktionen, mathematischen Funktionen, Zeitfunktionen, Tabellenfunktionen, Geodatenfunktionen usw.
- *Datenreplikation:*
  Leader-basierte Replikation, bei der Daten, die auf dem Leader-Knoten geschrieben werden, an einen Follower-Knoten (oder Replikat) weitergegeben werden. ksqlDB übernimmt diese Replikationsstrategie sowohl von Kafka (für die zugrunde liegenden Topic-Daten) als auch von Kafka Streams (für zustandsbehaftete Tabellendaten).

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Unterschiede zwischen ksqlDB und klassischem SQL

- *Verbesserte DDL- und DML-Anweisungen:*
  DDL- und DML-Anweisungen beziehen sich durch die dualität in Kafka auf Tabellen und Datenströme
- *Push-Abfragen:*
  ksqlDB unterstütz Abfragen im Lookup-Stil, jedoch ebenfalls kontinuierliche Abfragen um der unendlichkeit der Datenströme gerecht zu werden
- *Einfache Abfragefunktionen:*
  ksqlDB ist eine hochspezialisierte Datenbank, zugeschnitten auf bestimmte Reihe von Anwendungsfällen zugeschnitten, darunter Streaming ETL, materialisierte Caches und ereignisgesteuerte Microservices

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

- *ANSI-inspiriertes SQL, aber nicht vollständig konform*:
  Der SQL-Standard wird nicht vollständig unterstützt
- *Lokale und entfernte Speicherung:*
  Daten befinden sich in Kafka, lokale Statusspeicher können verwendet werden.
- *Konsistenzmodell:*
  ksqlDB hält sich an ein schließlich konsistentes und asynchrones Konsistenzmodell, während traditionelle Systeme sich eher an das Modell Atomicity, Consistency, Isolation, Durability (ACID) halten.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Architecture

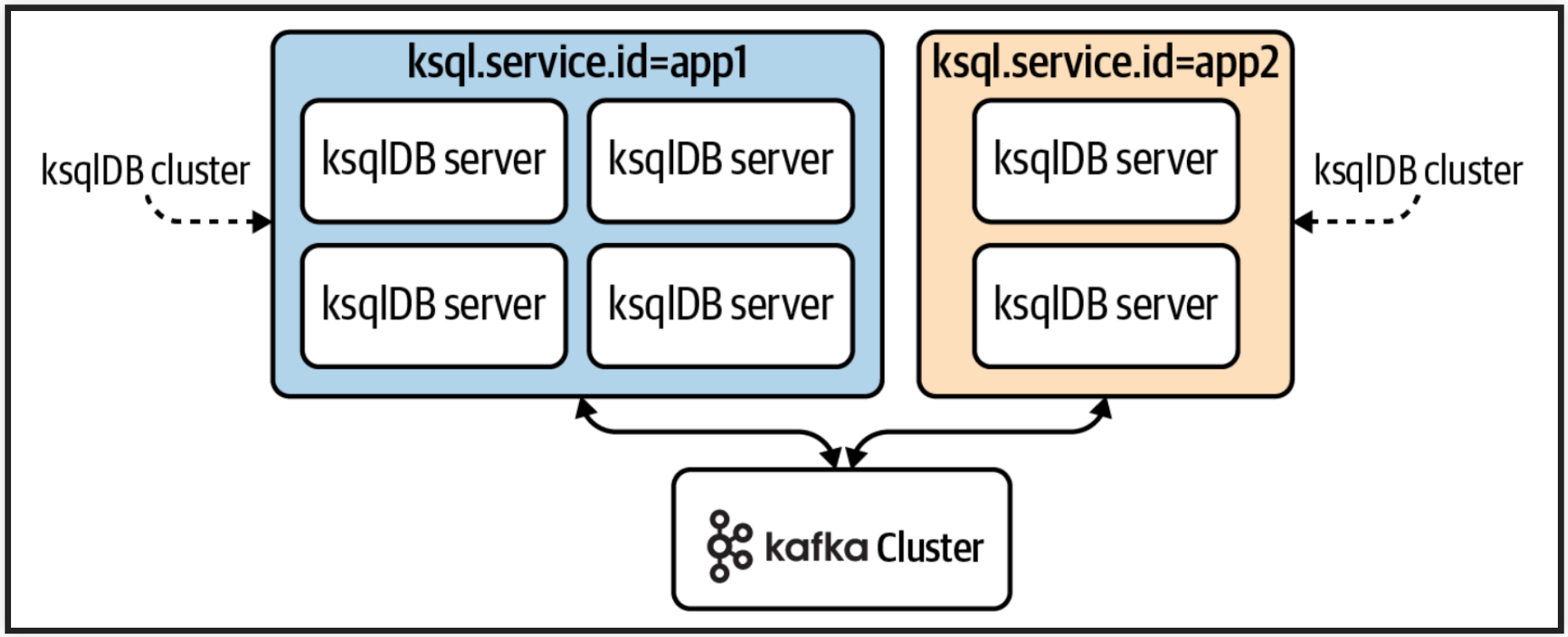
ksqlDB Server

ksqlDB-Server sind für die Ausführung Ihrer Stream-Processing-Anwendungen verantwortlich. Jeder Server ist konzeptionell vergleichbar mit einer einzelnen Instanz einer Kafka-Streams-Anwendung, und die Arbeitslasten (die durch den Abfragesatz erzeugt werden) können über mehrere ksqlDB-Server mit derselben ksql.service.id-Konfiguration verteilt werden. Wie Kafka-Streams-Anwendungen werden ksqlDB-Server getrennt vom Kafka-Cluster bereitgestellt (normalerweise auf separaten Maschinen/Containern von den Brokern selbst).

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

ksqlDB-Cluster

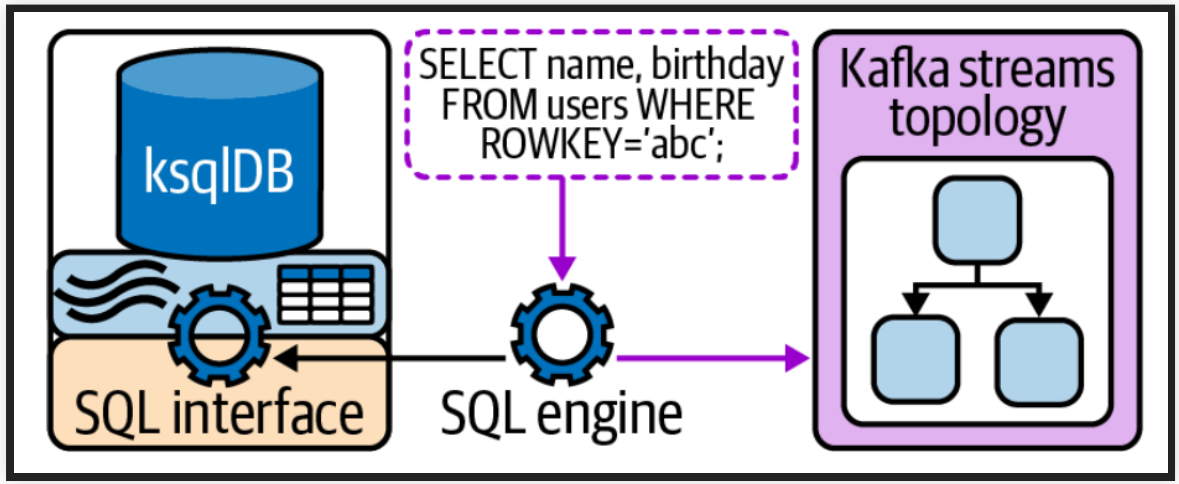
Eine Gruppe von kooperierenden ksqlDB-Servern wird als ksqlDB-Cluster bezeichnet, und es wird im Allgemeinen empfohlen, die Arbeitslasten für eine einzelne Anwendung auf Clusterebene zu isolieren.



Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

SQL Engine

Die SQL-Engine ist für das Parsen einer SQL-Anweisung, die Konvertierung in eine oder mehrere Kafka-Streams-Topologien und schließlich die Ausführung der Kafka-Streams-Anwendungen verantwortlich.



Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

REST service

ksqlDB enthält eine REST-Schnittstelle, über die Clients mit der SQL-Engine interagieren können. Diese wird verwendet, um Abfragen an die Engine zu übermitteln (d. h. DML-Anweisungen, die mit SELECT beginnen), andere Anweisungstypen (z. B. DDL-Anweisungen) auszuführen, den Cluster-Status/Gesundheitszustand zu überprüfen und vieles mehr. Standardmäßig wird auf Port 8088 gelauscht und über HTTP kommuniziert.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

ksqlDB Clients

- **ksqlDB CLI**: Die ksqlDB CLI ist eine Befehlszeilenanwendung, mit der Sie mit einem laufenden ksqlDB-Server interagieren können, bietet sich zum Experimentieren mit ksqlDB an:

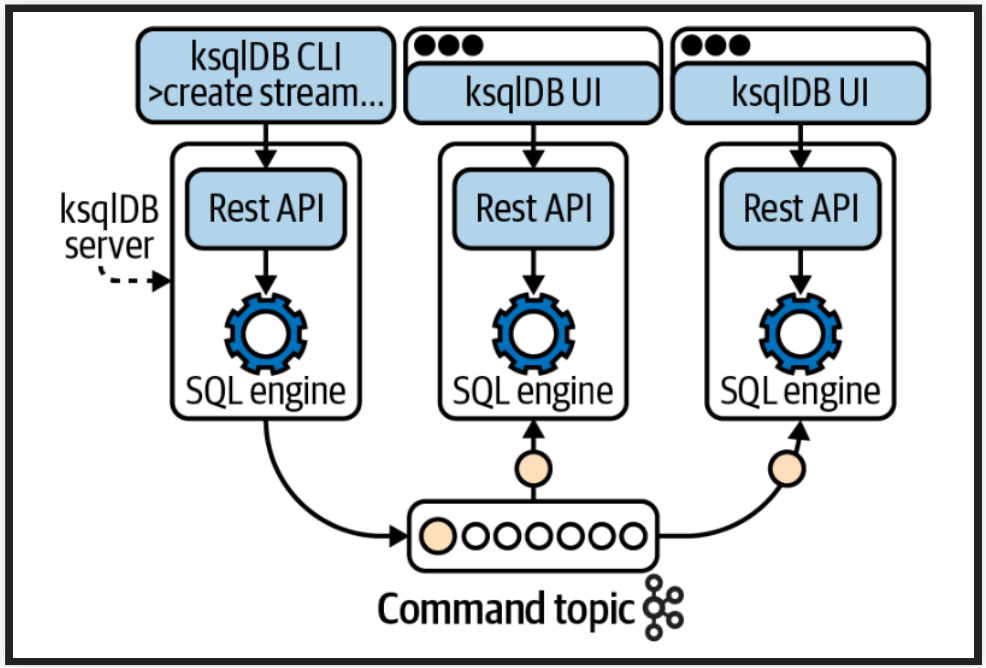
```
ksql http://localhost:8088
```

- **ksqlDB UI**: kommerzielles Feature von Confluent für die Interaktion mit ksqlDB

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Deployment Mode: Interactive Mode

Wenn ksqlDB im interaktiven Modus ausgeführt wird, können Clients jederzeit neue Abfragen über die REST-API übermitteln. Ermöglicht Streams, Tabellen, Abfragen und Konnektoren nach Belieben zu erstellen und abzubauen.



Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

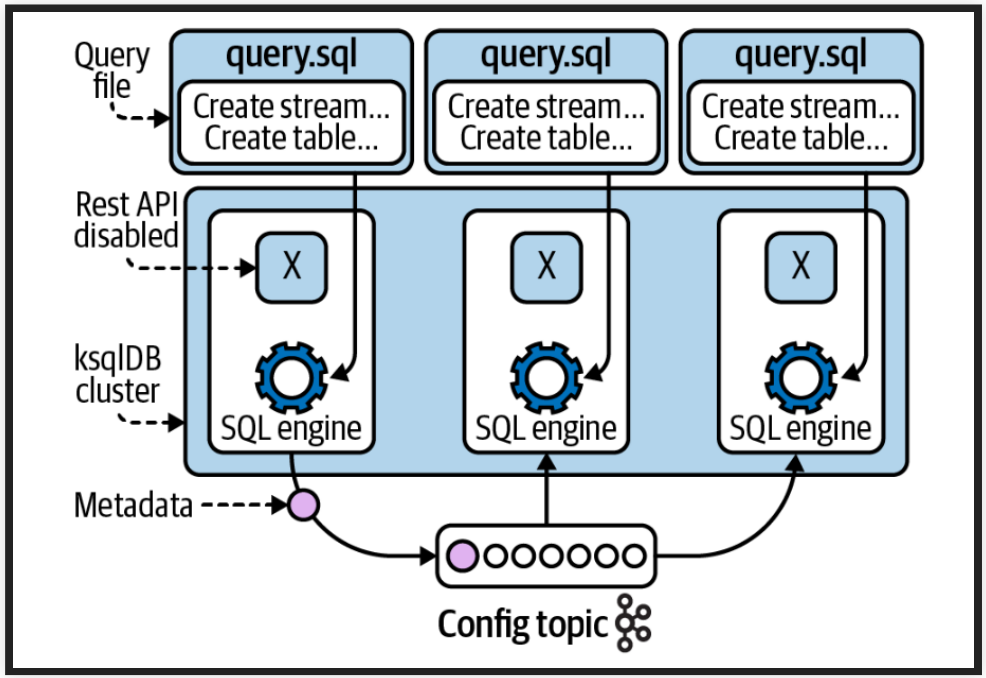
Im interaktiven Modus werden alle Abfragen, die an die SQL-Engine (über die REST-API) übermittelt werden, in ein internes Thema, das sogenannte Command-Topic, geschrieben werden. Dieses Thema wird von ksqlDB automatisch erstellt und verwaltet und dient der anweisungsbasierten Replikation, sodass alle ksqlDB-Server im Cluster die Abfrage ausführen können.

Der interaktive Modus ist der Standard-Einsatzmodus für ksqlDB.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Deployment Mode: Headless Mode

In Fällen, in denen eine Erweiterung des Clusters nicht vorgesehen ist (z.B. für Produktive Systeme), kann ksqlDB in einem Headless Mode betrieben werden. Damit ist die REST-API deaktiviert.



Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Beim Starten der ksqlDB Server sind die Queries anzugeben, dies wird über eine Konfiguration realisiert: `queries.file=/path/to/query.sql`

Im Headless Mode gibt es kein Command-Topic. Es werden einige interne Metadaten in ein Configuration-Topic geschrieben.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

FIRST KSQLDB SERVER

Folgendes Paket bietet eine Konfiguration zum Starten eines Zookeepers, Kafka und ksqlDB Servers.

```
docker compose up -d
```

Nutzen Sie das bereitgestellte Projekt-Archiv ksqldb.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

Create Topics

Für die folgenden Beispiele ist eine einfache Topic in Kafka notwendig, die wie bisher über die Admin CLI erzeugt werden kann.

```
docker compose exec kafka kafka-topics.sh \
      --bootstrap-server localhost:9092 \
      --topic users \
      --replication-factor 1 \
      --partitions 4 \
      --create
```

Das Beispiel startet eine CLI welche die Eingabe von KSQL-Befehlen erlaubt

```
~/ksqldb $ docker compose exec ksqldb ksql http://localhost:8088
                  ===========================================
                  =       _              _ _____  _____      =
                  =      | | _____  __ _| |  _ \| __ )      =
                  =      | |/ / __|/ _` | | | | |  _ \      =
                  =      |   <\__ \ (_| | | |_| | |_) |     =
                  =      |_|\_\___/\__, |_|____/|____/      =
                  =                   |_|                   =
                  =        The Database purpose-built       =
                  =        for stream processing apps       =
                  ===========================================

Copyright 2017-2022 Confluent Inc.

CLI v0.28.2, Server v0.28.2 located at http://localhost:8088
Server Status: RUNNING

Having trouble? Type 'help' (case-insensitive) for a rundown of how things work!

ksql>
```

Show Topics

```
ksql> SHOW Topics;

 Kafka Topic | Partitions | Partition Replicas
-------------------------------------------------
 users       | 4          | 1
-------------------------------------------------
```

Create Stream

```
ksql> CREATE STREAM users (
    ROWKEY INT KEY,
    USERNAME VARCHAR
) WITH (
    KAFKA_TOPIC='users',
    VALUE_FORMAT='JSON'
);

 Message
----------------
 Stream created
----------------
```

- Im Beispiel wird davon ausgegangen, dass ein Integer Key zum Einsatz kommt
- Als Attribut für den Stream kommt ein USERNAME als Zeichenkette hinzu
- Der Stream wird auf das Kafka Topic users abgebildet und dort als JSON-Daten seralisiert / deserialisiert

Insert Data

```
ksql> INSERT INTO users (username) VALUES ('izzy');
ksql> INSERT INTO users (username) VALUES ('elyse');
ksql> INSERT INTO users (username) VALUES ('mitch');
```

Mittels `INSERT INTO` lassen sich Einträge ergänzen. Welche ebenfalls in der Topic `users` beobachtet werden können:

```
docker compose exec kafka kafka-console-consumer.sh --topic users \
   --group foo --from-beginning --bootstrap-server localhost:9092

{"USERNAME":"izzy"}
{"USERNAME":"elyse"}
{"USERNAME":"mitch"}
```

Push Query

```
ksql> SET 'auto.offset.reset' = 'earliest';
ksql> SELECT 'Hello, ' + USERNAME AS GREETING
FROM users
EMIT CHANGES;
```

Mittels `SELECT` kann anschließend eine kontinuierliche Verarbeitung erfolgen, z.B. jeder Eintrag im Stream users als Greeting-Mitteilung.

ÜBERWACHEN VON ÄNDERUNGEN IN EINEM VIDEO STREAMING SERVICE

Ziel dieser Anwendung ist es einen Strom von Produktionsänderungen abzurufen, die Daten für die Verarbeitung filtern und umwandeln, die Daten für Berichtszwecke anreichern und aggregieren und schließlich die verarbeiteten Daten den nachgelagerten Systemen zur Verfügung stellen.

- Das Topic `title` enthält Metadaten (Titelname, Veröffentlichungsdatum usw.) für Filme und Fernsehsendungen
- In das Topic `production_changes` wird immer dann geschrieben, wenn sich die Besetzung, das Budget, das Veröffentlichungsdatum oder die Länge der Staffel eines in Produktion befindlichen Titels ändert

Example record: titles

```
{
    "id": 1,
    "title": "Stranger Things",
    "on_schedule": false
}
```

Example record: production_change

```
{
    "uuid": 1,
    "title_id": 1,
    "change_type": "season_length",
    "before": {
        "season_id": 1,
        "episode_count": 12
    },
    "after": {
        "season_id": 1,
        "episode_count": 8
    },
    "created_at": "2021-02-08 11:30:00"
}
```

Erstellen der Title-Table

```
CREATE TABLE titles (
    id INT PRIMARY KEY,
    title VARCHAR
) WITH (
    KAFKA_TOPIC='titles',
    VALUE_FORMAT='JSON',
    PARTITIONS=4
);
```

- Mittels `CREATE {TABLE | STREAM}` können neue Tabellen bzw. Streams beschrieben werden.
- Dabei ist der Key anzugeben (falls vorhanden) sowie die Daten die im Payload existieren
- Über den `WITH-Block` wird der Stream / die Tabelle konfiguriert, z.B. wird die zugrundeliegende Topic angegeben und in welchem Format Daten in der Topic transportiert werden
- Wird eine Partitionsanzahl angegeben, erstellt ksqlDB das Topic falls es noch nicht existiert

Details zur Erstellten Tabelle anzeigen

Mittels SHOW TABLES bzw. DESCRIBE-Befehl lassen sich Details zur erstellten Tabelle anzeigen.

```
SHOW TABLES [ EXTENDED ];
DESCRIBE [ EXTENDED ] titles;
```

Beispieldaten für die Tabelle können wie folgt erzeugt werden:

```
INSERT INTO titles VALUES (1, 'Stranger Things');
INSERT INTO titles VALUES (2, 'Black Mirror');
INSERT INTO titles VALUES (3, 'Bojack Horseman');
```

Substrukturen

In dem Topic für Produktionsänderungen gibt es eine Substruktur (z.B. `after` und `before`):

```
/* ...*/
    "after": {
        "season_id": 1,
        "episode_count": 8
    },
/* ...*/
```

Diese können über Typen in ksqlDB beschrieben werden. Ein Beispiel für die Season-Length-Informationen wäre:

```
CREATE TYPE season_length AS STRUCT<season_id INT, episode_count INT> ;
```

```
SHOW TYPES;
```

Erstellen des Production-Changes-Streams

```
CREATE STREAM production_changes (
    rowkey VARCHAR KEY,
    uuid INT,
    title_id INT,
    change_type VARCHAR,
    before season_length,
    after season_length,
    created_at VARCHAR
) WITH (
    KAFKA_TOPIC='production_changes',
    PARTITIONS='4',
    VALUE_FORMAT='JSON',
    TIMESTAMP='created_at',
    TIMESTAMP_FORMAT='yyyy-MM-dd HH:mm:ss'
);
```

- Der Stream besteht aus den beschriebenen Daten
- Konfiguriert wird der Stream gegen das Kafka Topic `production_changes` mit 4 Partitionen und im JSON-Format
- Zeitbezug für das Ereignis über das Attribut `created_at`, Format kann über die Möglichkeiten der Java Klasse `java.time.format.DateTimeFormatter` beschrieben werden

Streams und Tables löschen

Erzeuge Streams und Tabellen lassen sich mit DROP-Befehlen wieder entfernen.

```
DROP { STREAM | TABLE } [ IF EXISTS ] <identifier> [DELETE TOPIC];
```

Für die erstellten Tabellen / Streams `titles` und `production_changes` ist dies:

```
DROP TABLE IF EXISTS titles DELETE TOPIC ;
DROP STREAM IF EXISTS production_changes DELETE TOPIC ;
```

Beispieldaten für den Stream könnten wie folgt aussehen:

```
INSERT INTO production_changes (uuid, title_id, change_type, before,
    after, created_at)
VALUES (1, 1, 'season_length',
    STRUCT(season_id := 1, episode_count := 12),
    STRUCT(season_id := 1, episode_count := 8),
    '2021-02-08 10:00:00'
);
```

```
INSERT INTO production_changes (ROWKEY, ROWTIME, uuid, title_id,
    change_type, before, after, created_at)
VALUES ('2', 1581161400000, 2, 2, 'release_date',
    STRUCT(season_id := 1, release_date := '2021-05-27'),
    STRUCT(season_id := 1, release_date := '2021-08-18'),
    '2021-02-08 10:00:00'
);
```

SELECT

SELECT-Statements erlauben die Abfrage von Daten aus Streams oder Tabellen. Der vollständige Syntax ist folgend aufgeführt und besteht aus typischen SQL-Elementen wie SELECT, FROM, WHERE, GROUP BY sowie Stream-spezifischen Elementen wie WINDOW und PARTITION BY.

```
SELECT select_expr [, ...]
FROM from_item
[ LEFT JOIN join_collection ON join_criteria ]
[ WINDOW window_expression ]
[ WHERE condition ]
[ GROUP BY grouping_expression ]
[ PARTITION BY partitioning_expression ]
[ HAVING having_expression ]
EMIT CHANGES
[ LIMIT count ];
```

Für die folgenden Beispiele sollte erneut darauf geachtet werden, dass der offset bei dem ältesten Element beginnt.

```
SET 'auto.offset.reset' = 'earliest';
```

Simple Selects (Transient Push Queries)

Die zwei folgenden SELECT-Anweisungen fragen die Inhalte der Tabellen und Streams vollständig ab.

```
SELECT * FROM production_changes EMIT CHANGES;
SELECT * FROM titles EMIT CHANGES ;
```

Diese sind transient, da sie nicht weiter gespeichert werden. `EMIT CHANGES` liefert Aktualisierungen, wenn sich etwas ändert. Eine Alternative wäre `EMIT FINAL`.

Select Fields

Für die Abfrage bestimmter Felder kann wie bei SQL die Angabe anstatt dem * erfolgen. Hier werden Titel, Before, After und CreatedAt in das Ergebnis einbezogen.

```
SELECT title_id, before, after, created_at
FROM production_changes
EMIT CHANGES;
```

Filtering

Mittels `WHERE` kann ein Filtern der Abfrage vorgenommen werden. Als Beisßiel könnte nach dem `change_type` gefiltert werden.

```
SELECT title_id, before, after, created_at
FROM production_changes
WHERE change_type = 'season_length'
EMIT CHANGES ;
```

Wildcards

Als weiteres Beispiel könnte man nur nach change_types suchen, die mit "season" beginnen:

```
SELECT title_id, before, after, created_at
FROM production_changes
WHERE change_type LIKE 'season%'
EMIT CHANGES ;
```

Logical operators

... oder nach Ereignissen suchen, die eine bestimmte Anzahl an Episoden nach der Änderungsmitteilung haben:

```
SELECT title_id, before, after, created_at
FROM production_changes
WHERE NOT change_type = 'release_date'
AND ( after->episode_count >= 8 OR after->episode_count  <=20 )
EMIT CHANGES ;
```

Between (range filter)

Die Suche nach einem `8 <= episode_count <= 20` könnte z.B. auch als Range angegeben werden:

```
SELECT title_id, before, after, created_at
FROM production_changes
WHERE change_type = 'season_length'
AND after->episode_count BETWEEN 8 AND 20
EMIT CHANGES ;
```

Flattening/Unnesting Complex Structures

Sub-Strukturen können mit der Abfrage ebenfalls aufgebrochen werden, z.B. wenn nur die Season-ID und der Episoden-Count nach der Änderung interessant ist.

```
SELECT
    title_id,
    after->season_id,
    after->episode_count,
    created_at
FROM production_changes
WHERE change_type = 'season_length'
EMIT CHANGES ;
```

QUERIES

Bisher wurden Daten nur direkt abgefragt. Select-Statements können mit einem neuen Stream in Verbindung gebracht werden und damit kontinuierlich verarbeitet und an Kafka zurückgespielt werden. Der vollständige Syntax ist folgend angegeben:

```
CREATE { STREAM | TABLE } [ IF NOT EXISTS ] <identifier>
WITH (
    property=value [, ... ]
)
AS SELECT select_expr [, ...]
FROM from_item
[ LEFT JOIN join_collection ON join_criteria ]
[ WINDOW window_expression ]
[ WHERE condition ]
[ GROUP BY grouping_expression ]
[ PARTITION BY partitioning_expression ]
[ HAVING having_expression ]
EMIT CHANGES
[ LIMIT count ];
```

Ein einfaches Beispiel wäre das folgende, welches Informationen aus den `production_changes` nimmt und in einen neuen Stream season_length_changes veröffentlicht.

```
CREATE STREAM season_length_changes
WITH (
    KAFKA_TOPIC = 'season_length_changes',
    VALUE_FORMAT = 'JSON',
    PARTITIONS = 4,
    REPLICAS = 1
) AS SELECT
    ROWKEY,
    title_id,
    IFNULL(after->season_id, before->season_id) AS season_id,
    before->episode_count AS old_episode_count,
    after->episode_count AS new_episode_count,
    created_at
FROM production_changes
WHERE change_type = 'season_length'
EMIT CHANGES;
```

Anzeigen von Details zum Query, bzw. das Entfernen des Queries ist wie folgt möglich:

```
{ LIST | SHOW } QUERIES [EXTENDED];
EXPLAIN season_length_changes
DROP STREAM season_length_changes ;
```

Ein Blick über die Kafka CLI in die entsprechende Topic zeigt, dass die Informationen im Kafka aufgelaufen sind.

```
docker compose exec kafka kafka-console-consumer.sh \
    --topic season_length_changes --group foo --from-beginning \
    --bootstrap-server localhost:9092
```

RUN SQL-FILES

Neben dem direkten Eingeben der SQL-Befehle in die ksqlDB-CLI lassen sich diese auch über SQL-Dateien direkt ausführen. Grundlage hierfür ist der folgende RUN-Befehl:

```
RUN SCRIPT <sql_file>;
```

Ein Beispiel wäre z.B.:

```
RUN SCRIPT '/etc/sql/streaming.sql' ;
```

Das Beispiel muss nur ausgeführt werden, wenn Sie die vorangegangenen Kapitel übersprungen haben. Es enthält die Tabellen, Streams und Typen sowie Beispieldaten.

JOIN

JOIN in ksqlDB unterliegen den selben Regeln wie bei Kafka Streaming, entsprechend ist Rücksicht auf die Partitionierung und Verteilung von Keys über die Partitionen zu nehmen. Über SQL-übliche Befehle wie INNER JOIN lassen sich Datenströme miteinander kombinieren:

```
SELECT
    s.title_id,
    t.title,
    s.season_id,
    s.old_episode_count,
    s.new_episode_count,
    s.created_at
FROM season_length_changes s
INNER JOIN titles t
ON s.title_id = t.id
EMIT CHANGES ;
```

Repartitioning Data

Eine Möglichkeit ist zum Beispiel zuvor eine Repartitionierung wie in Kafka Streams vorzunehmen, welche die Verteilung korrigiert.

```
CREATE TABLE titles_repartition
WITH (PARTITIONS=4) AS
SELECT * FROM titles
EMIT CHANGES;
```

Persistence Joins

Das Ergebnis eines Joins kann erneut als neuer Stream in einem eigenen Kafka Topic bereitgestellt werden:

```
CREATE STREAM season_length_changes_enriched
WITH (
    KAFKA_TOPIC = 'season_length_changes_enriched',
    VALUE_FORMAT = 'JSON',
    PARTITIONS = 4,
    TIMESTAMP='created_at',
    TIMESTAMP_FORMAT='yyyy-MM-dd HH:mm:ss'
) AS SELECT
    s.title_id,
    t.title,
    s.season_id,
    s.old_episode_count,
    s.new_episode_count,
    s.created_at
FROM season_length_changes s
INNER JOIN titles t
ON s.title_id = t.id
EMIT CHANGES;
```

WINDOWED JOINS

Joins mit direkten Bezug zu Zeitfenstern ermöglichen auftretende Ereignisse in einem Zeitfenster miteinander zu kombinieren. Für die folgenden Beispiele werden zwei weitere Streams erstellt, diese können direkt mit dem Script ergänzt werden.

```
RUN SCRIPT /etc/sql/watching.sql
```

Start Watching Events

In dem Stream `start_watching_events` werden Ereignisse platziert, welche dann entstehen, wenn ein Nutzer eine Serie anfängt zu schauen.

```
CREATE STREAM start_watching_events (
    session_id STRING,
    title_id INT,
    created_at STRING
) WITH (
    KAFKA_TOPIC='start_watching_events',
    VALUE_FORMAT='JSON',
    PARTITIONS=4,
    TIMESTAMP='created_at',
    TIMESTAMP_FORMAT='yyyy-MM-dd HH:mm:ss'
);
```

Stop Watching Events

In dem Stream `stop_watching_events` werden Ereignisse platziert, welche dann entstehen, wenn ein Nutzer eine Serie aufhört zu schauen.

```
CREATE STREAM stop_watching_events (
    session_id STRING,
    title_id INT,
    created_at STRING
) WITH (
    KAFKA_TOPIC='stop_watching_events',
    VALUE_FORMAT='JSON',
    PARTITIONS=4,
    TIMESTAMP='created_at',
    TIMESTAMP_FORMAT='yyyy-MM-dd HH:mm:ss'
);
```

Beispieleinträge für Start/Stop Watching Events

```
INSERT INTO start_watching_events
VALUES ('session_123', 1, '2021-02-08 02:00:00');
INSERT INTO stop_watching_events
VALUES ('session_123', 1, '2021-02-08 02:01:30');
INSERT INTO start_watching_events
VALUES ('session_456', 1, '2021-02-08 02:00:00');
INSERT INTO stop_watching_events
VALUES ('session_456', 1, '2021-02-08 02:25:00');
```

Die Einträge sind im Script watching.sql bereits enthalten.

Join in zwei Minuten Zeitfenster

Das folgende Beispiel erzeugt einen Inner Join wie vorher, begrenzt den Join jedoch auf ein zwei Minuten Zeitfenster.

```
SELECT
    A.title_id as title_id,
    A.session_id as session_id
FROM start_watching_events A
INNER JOIN stop_watching_events B
WITHIN 2 MINUTES
ON A.session_id = B.session_id
EMIT CHANGES ;
```

Hier werden Start- und Stop-Ereignisse kombiniert, welche in einem nahen Zeitfenster auftreten. Dies könnte zum Beispiel genutzt werden um das Abbrechen direkt nach dem Start abzufangen.

AGGREGATION BASICS

ksqlDB bietet verschiedene Aggregationsfunktionen, welche in den Statements verwendet werden können. Ein erstes Beispiel wäre `COUNT`, welches die Anzahl von Ereignissen zählt und `LATEST_BY_OFFSET` welches den letzten Wert einer Ereigniskette liefert.

```
SELECT
    title_id,
    COUNT(*) AS change_count,
    LATEST_BY_OFFSET(new_episode_count) AS latest_episode_count
FROM season_length_changes_enriched
GROUP BY title_id
EMIT CHANGES ;
```

Die Aggregation erfolgt über die Gruppierung, hier nach `title_id`. Weitere Beispiele für Aggregationen wären `AVG`, `COUNT_DISTINCT`, `MAX`, `MIN`, `SUM`.

Selects mit Aggregationen können auch weitere Informationen enthalten

```
SELECT
    title_id,
    season_id,
    COUNT(*) AS change_count,
    LATEST_BY_OFFSET(new_episode_count) AS latest_episode_count
FROM season_length_changes_enriched
GROUP BY title_id, season_id
EMIT CHANGES ;
```

Dies erfordert jedoch, dass `season_id` als GROUP BY mit aufgeführt wird. Alternativ kann auf `EARLIEST_BY_OFFSET` bzw. `EARLIEST_BY_OFFSET` zurückgegriffen werden

Aggregationen in Windows

Soll die Aggregation in einem Zeitfenster realisiert werden, können die Daten erneut gruppiert werden und über z.B. ein `WINDOW TUMBLING` in einem entsprechenden Fenster aggregiert werden.

```
SELECT
    title_id,
    season_id,
    COUNT(*) AS change_count,
    LATEST_BY_OFFSET(new_episode_count) AS latest_episode_count
FROM season_length_changes_enriched
WINDOW TUMBLING (SIZE 1 HOUR)
GROUP BY title_id, season_id
EMIT CHANGES ;
```

Verspätungen über Grace-Konfiguration

Wie bei Kafka Streams ist bei Zeitfenstern zu berücksichtigen, dass Ereignisse verspätet eintreffen können. Die Grace-Period kann für Zeitfenster entsprechend angegeben werden.

```
SELECT
    title_id,
    season_id,
    COUNT(*) AS change_count,
    LATEST_BY_OFFSET(new_episode_count) AS episode_count
FROM season_length_changes_enriched
WINDOW TUMBLING (SIZE 1 HOUR, GRACE PERIOD 10 MINUTES)
GROUP BY title_id, season_id
EMIT CHANGES ;
```

6.3 SIDDHI

ToDo