

Wie man qualitativ hochwertige Kafka Streams-Applikationen entwickelt

Entwicklertag Karlsruhe 2025

andrena
OBJECTS

Paul Bauknecht



Vorstellung

Paul Bauknecht

- Seit 2019 bei andrena objects in Frankfurt a.M.
- Full-Stack-Entwickler
- Fokus auf Backend, Kafka, DevOps und Cloud



Inhalt

1. Kurzeinführung
2. Beispiel-App
3. Software Design
4. Parallelisierung und Race Conditions
5. Testing
6. Internal Topic Naming
7. Weitere Fallstricke
8. State Stores
9. Monitoring & Observability
10. Integration mit angrenzenden Systemen
11. Fazit

Ziel:
Bewusstsein für typische Herausforderungen
mit Kafka Streams schaffen und Lösungswege
aufzeigen



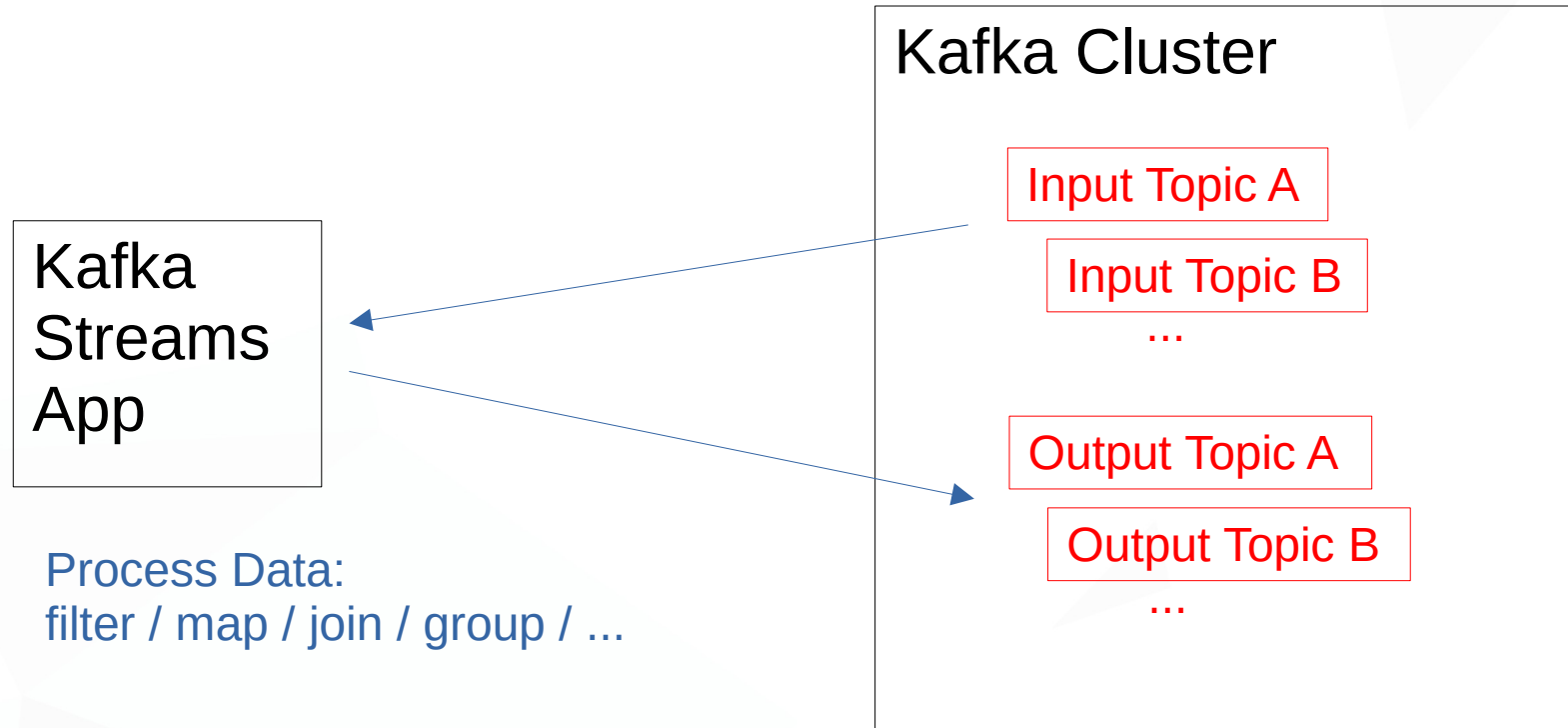
Kurzeinführung

Kafka Streams ist

- ein Daten-Streaming-Framework

Daten-Streaming = Kontinuierliche Verarbeitung von Datenströmen (Kafka Messages) in Echtzeit

- Skalierbar
- Fehlertolerant (redundantes verteiltes System)
- Exactly-Once-Fähig





Beispiel- Applikation

Sämtlicher gezeigter Code stammt aus der Beispielapplikation
<https://github.com/selbstereg/kafka-streams-example-etka25.git>

Das Repo enthält außerdem:

- README.md
 - gesamter Inhalt des Vortrags
 - nützliche weiterführende Links
- presentation.**pdf** - die Präsentation



Input Topic user-lists:

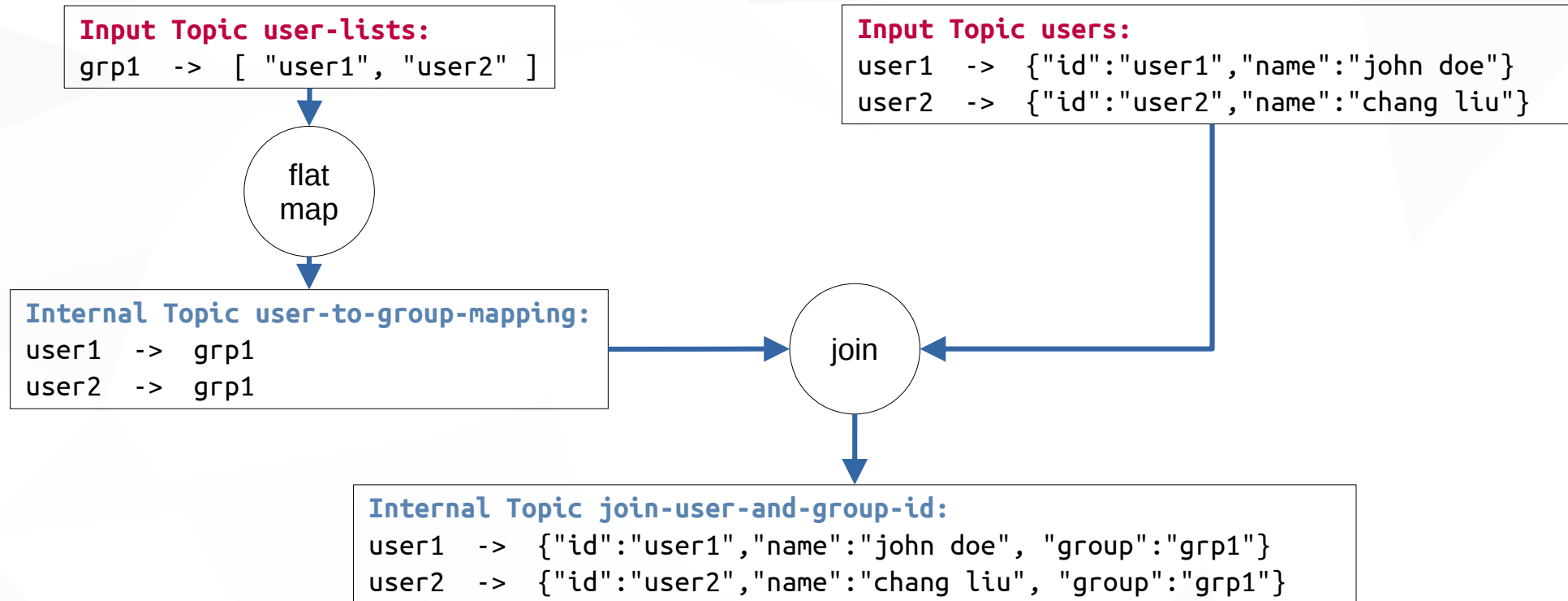
grp1 -> ["user1", "user2"]

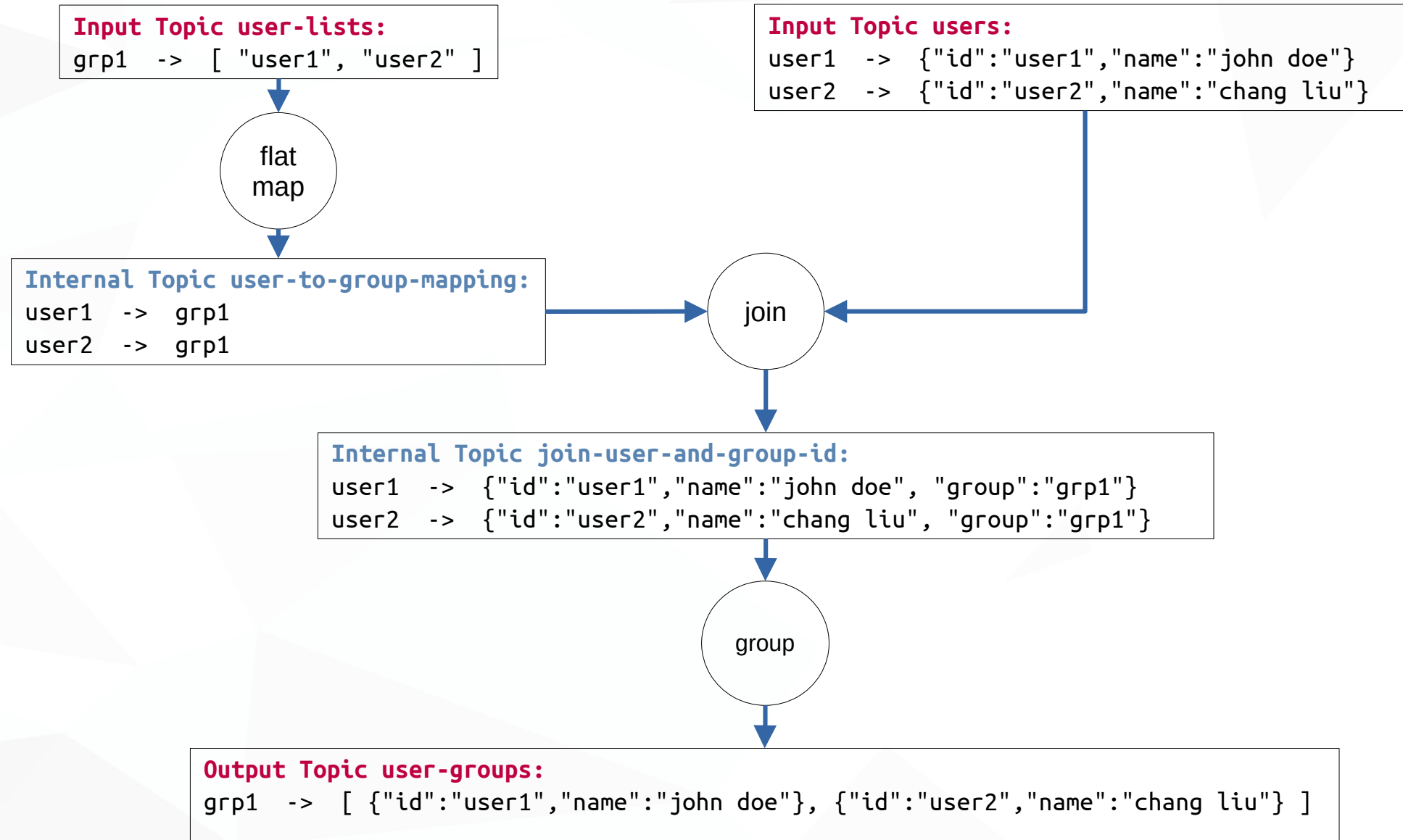
flat
map

Internal Topic user-to-group-mapping:

user1 -> grp1

user2 -> grp1







Software Design

```
@Configuration
@RequiredArgsConstructor
public class UserGroupRootTopology {
    public static final String USER_LISTS_INPUT_TOPIC = "user-lists";
    public static final String USERS_INPUT_TOPIC = "users";
    public static final String USER_GROUPS_OUTPUT_TOPIC = "user-groups";

    // ... code left out

    @PostConstruct
    public void createUserGroupRootTopology() {
        // key: groupId, value: list of userIds
        KStream<String, String> userLists = topicConnector.readInputTopic(USER_LISTS_INPUT_TOPIC);
        KStream<String, String> users = topicConnector.readInputTopic(USERS_INPUT_TOPIC);

        // 1. flat map user group - key: userId, value: groupId
        KStream<String, String> userToGroupMapping = userListFlatMappingTopology.build(userLists);

        // 2. join users with userToGroupMapping to resolve their groups -
        // key: userId, value: user with group
        KTable<String, String> usersWithGroupIds = userGroupIdJoinTopology.build(users, userToGroupMapping);

        // 3. aggregate users
        // key: groupId, value: list of user objects
        KTable<String, String> userGroups = groupAggregationTopology.build(usersWithGroupIds);

        topicConnector.writeToTopic(userGroups, USER_GROUPS_OUTPUT_TOPIC);
    }
}
```

input →

Verarbeitungsschritte

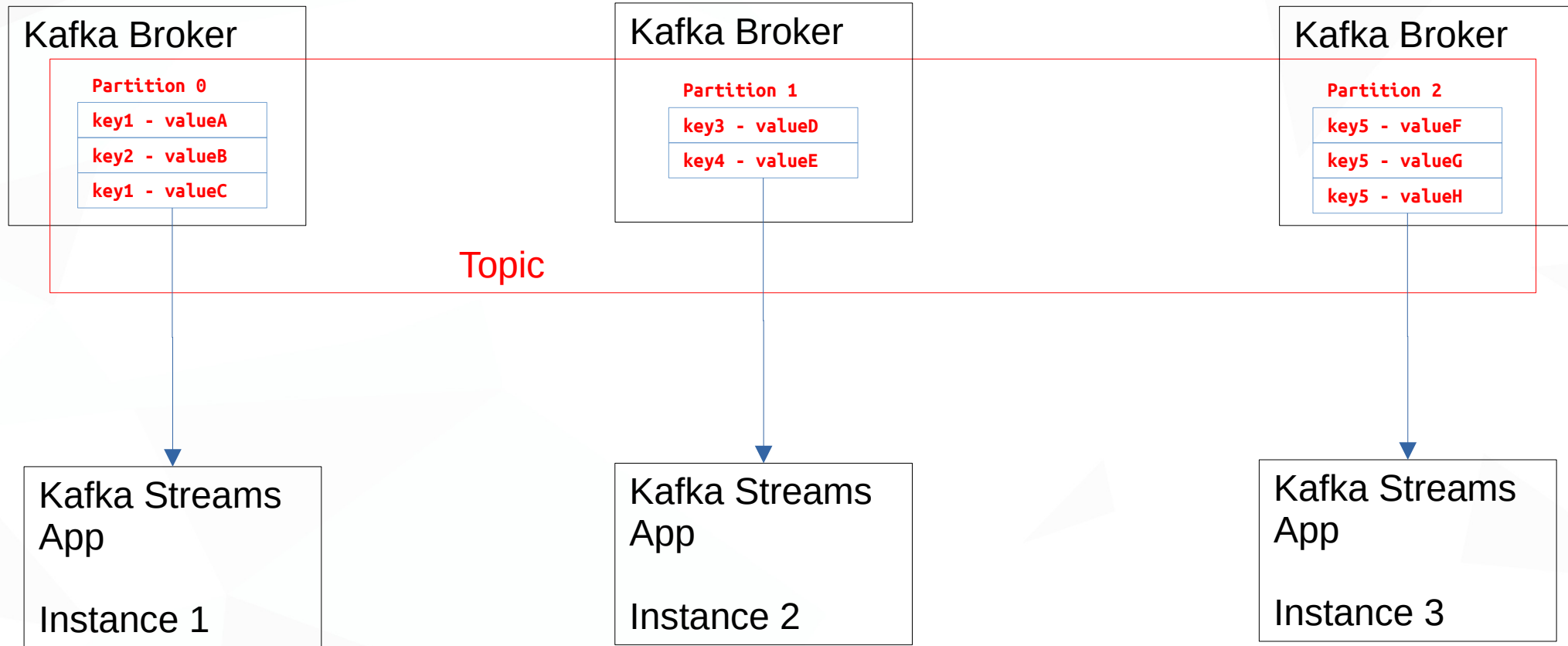
output →

```
@Component
@RequiredArgsConstructor
public class GroupAggregationTopology {
    private final UserMapper userMapper;

    public KTable<String, String> build(KTable<String, String> usersWithGroupIds) {
        return usersWithGroupIds
            .groupBy(
                (key, userJson) -> {
                    User user = userMapper.deserialize(userJson);
                    return KeyValue.pair(user.group(), userJson);
                }
            )
            .aggregate(
                () -> "[]",
                (key, userJson, aggregate) -> {
                    List<User> userGroup = userMapper.deserializeGroup(aggregate);
                    User user = userMapper.deserialize(userJson);
                    userGroup.add(user);
                    return userMapper.serialize(userGroup);
                },
                (key, userJson, aggregate) -> {
                    List<User> userGroup = userMapper.deserializeGroup(aggregate);
                    User user = userMapper.deserialize(userJson);
                    userGroup.remove(user);
                    if (userGroup.isEmpty())
                        return null; // treat empty group as deleted => emit tombstone
                    else
                        return userMapper.serialize(userGroup);
                },
                // ... code left out
            )
    }
}
```

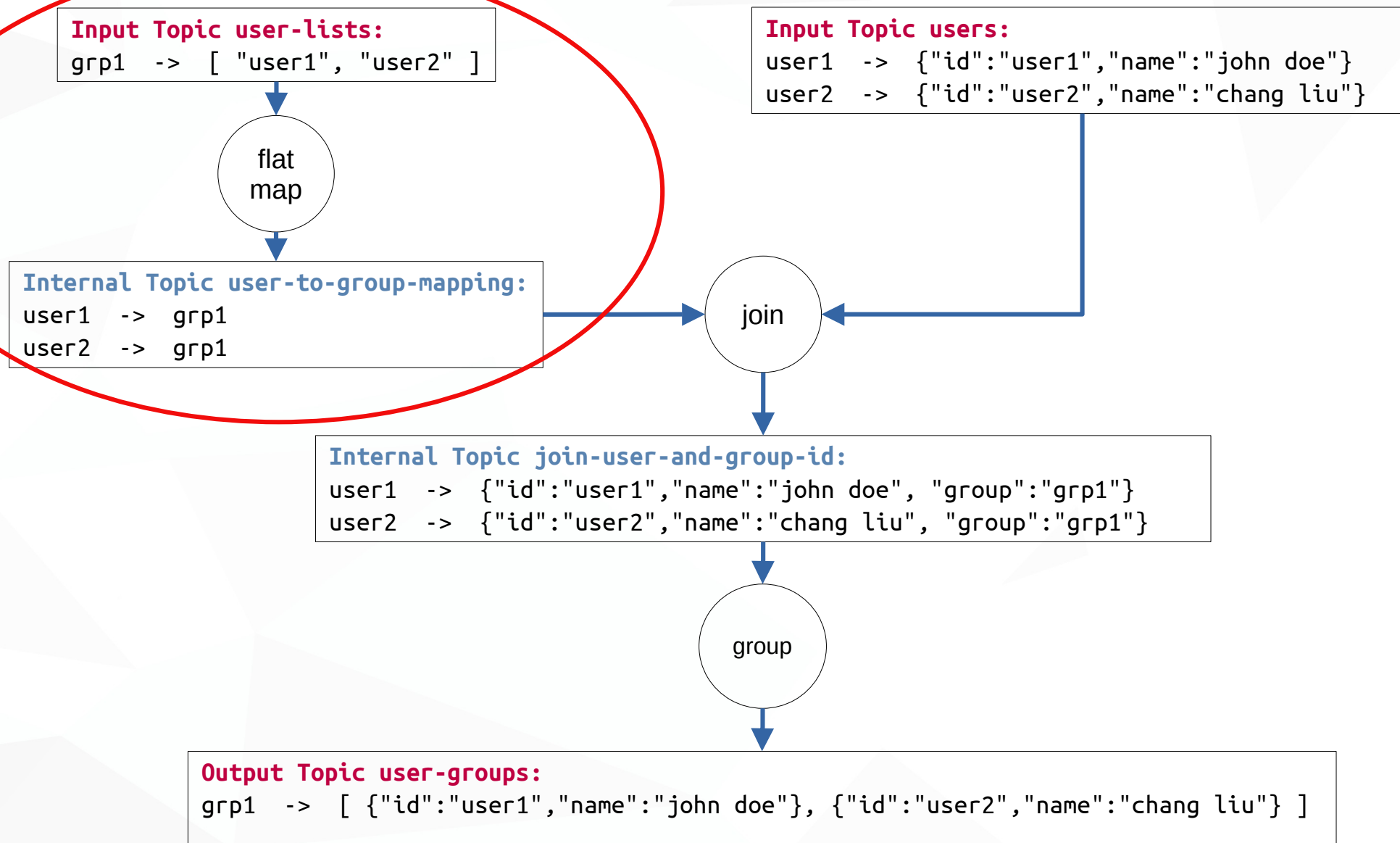


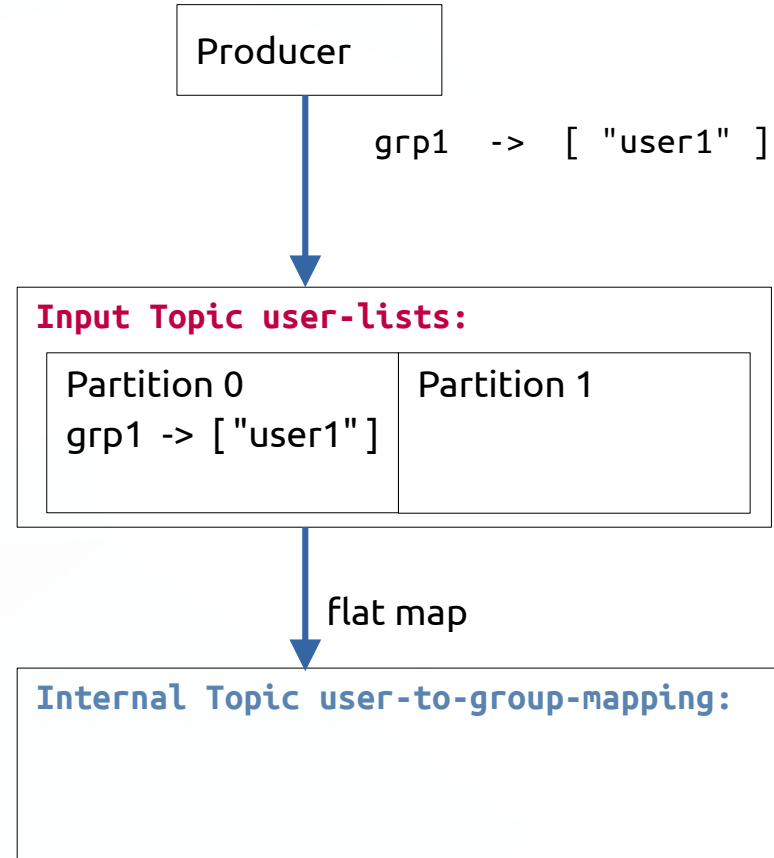
Parallelisierung und Race Conditions

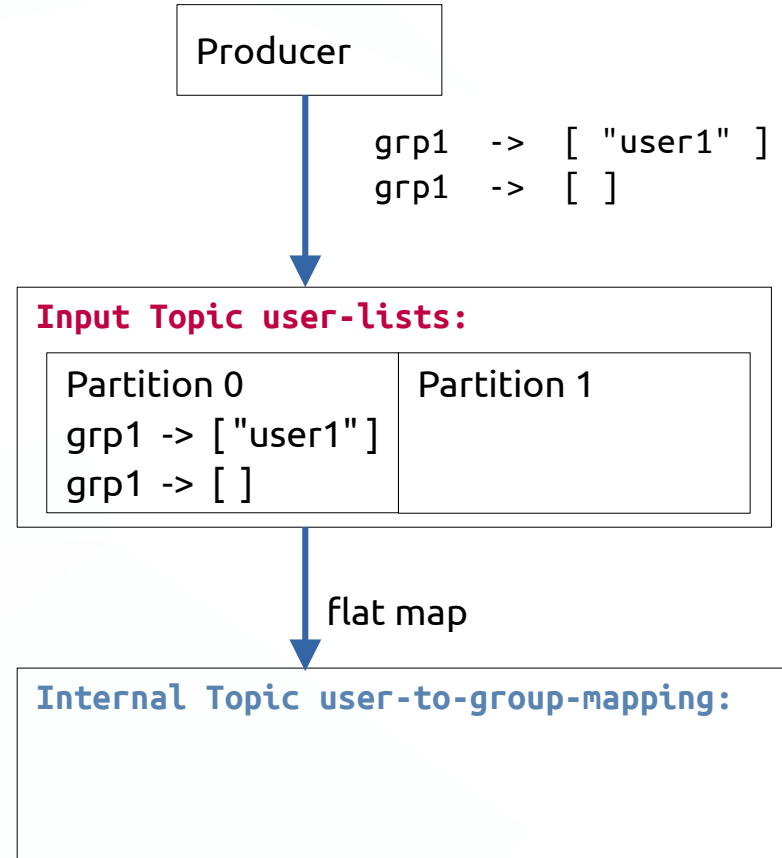


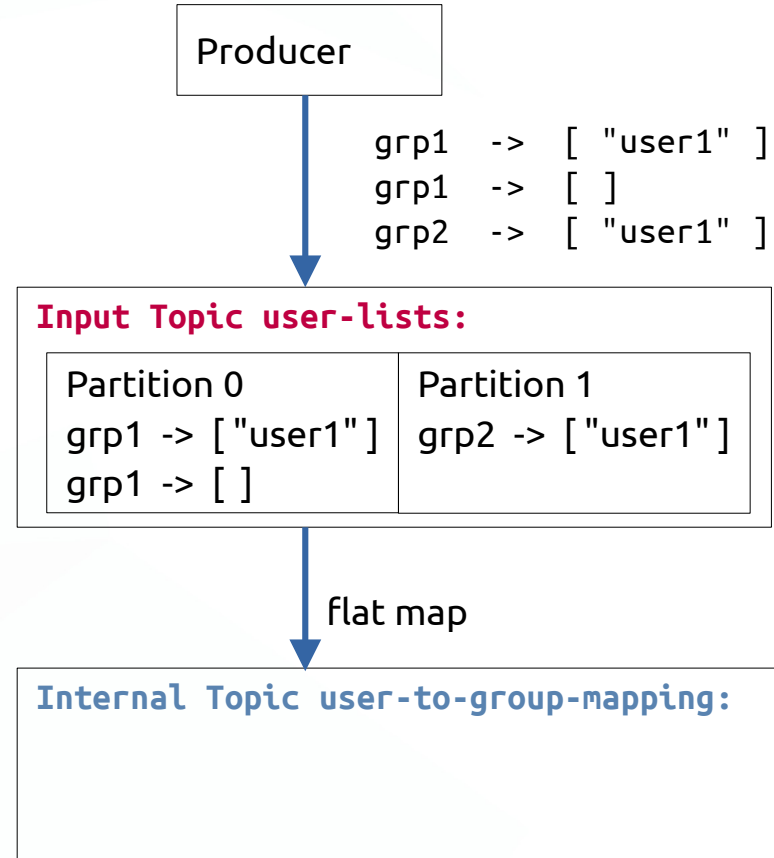
Probleme:

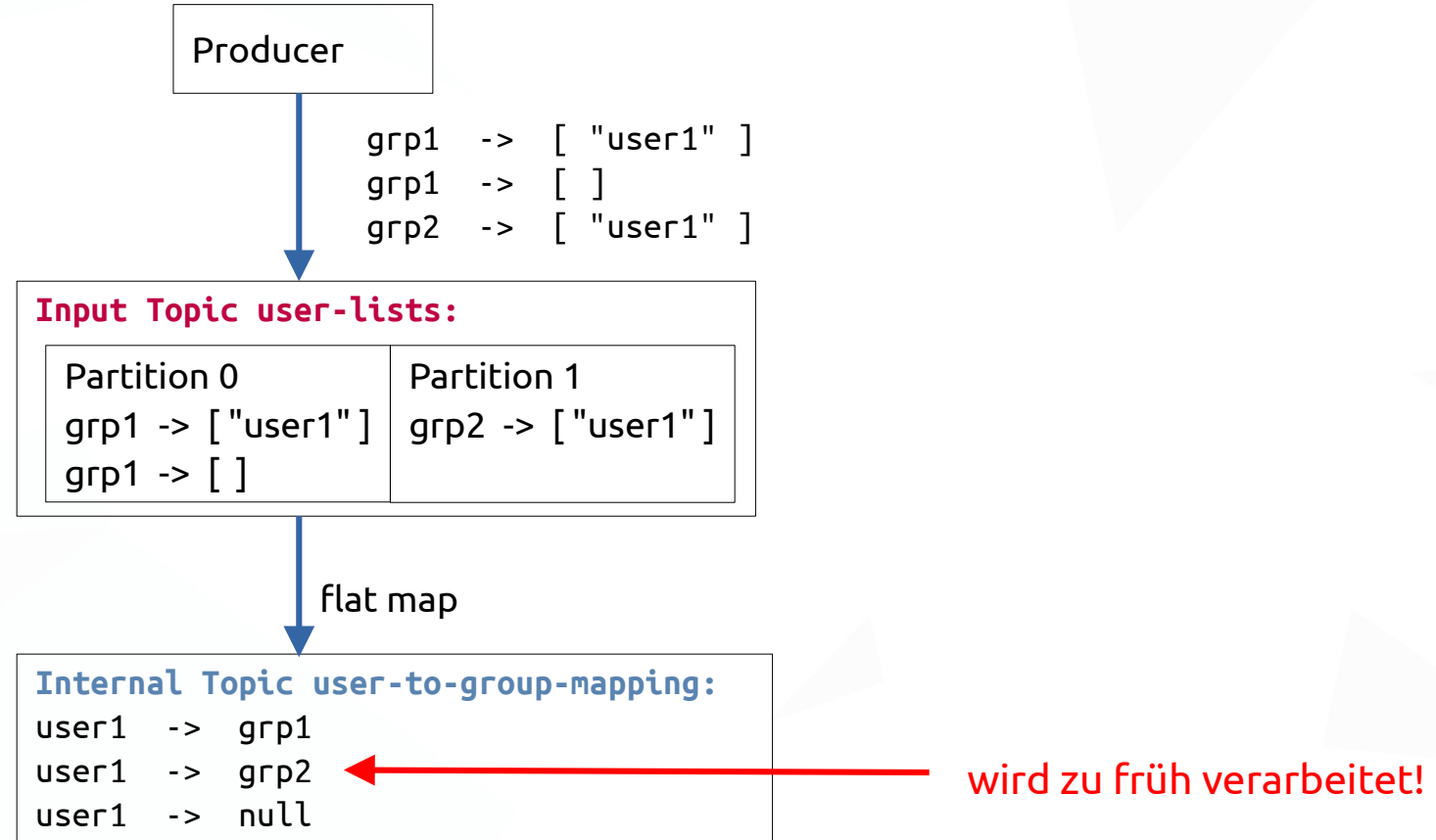
- Reihenfolge der Messages nur innerhalb einer Partition definiert
=> Race Conditions möglich
- Ein Stream-Task kennt nur einen Teil des Datensatzes
=> Für manche Berechnungen Repartitionierung oder Nutzung von GlobalKTable nötig











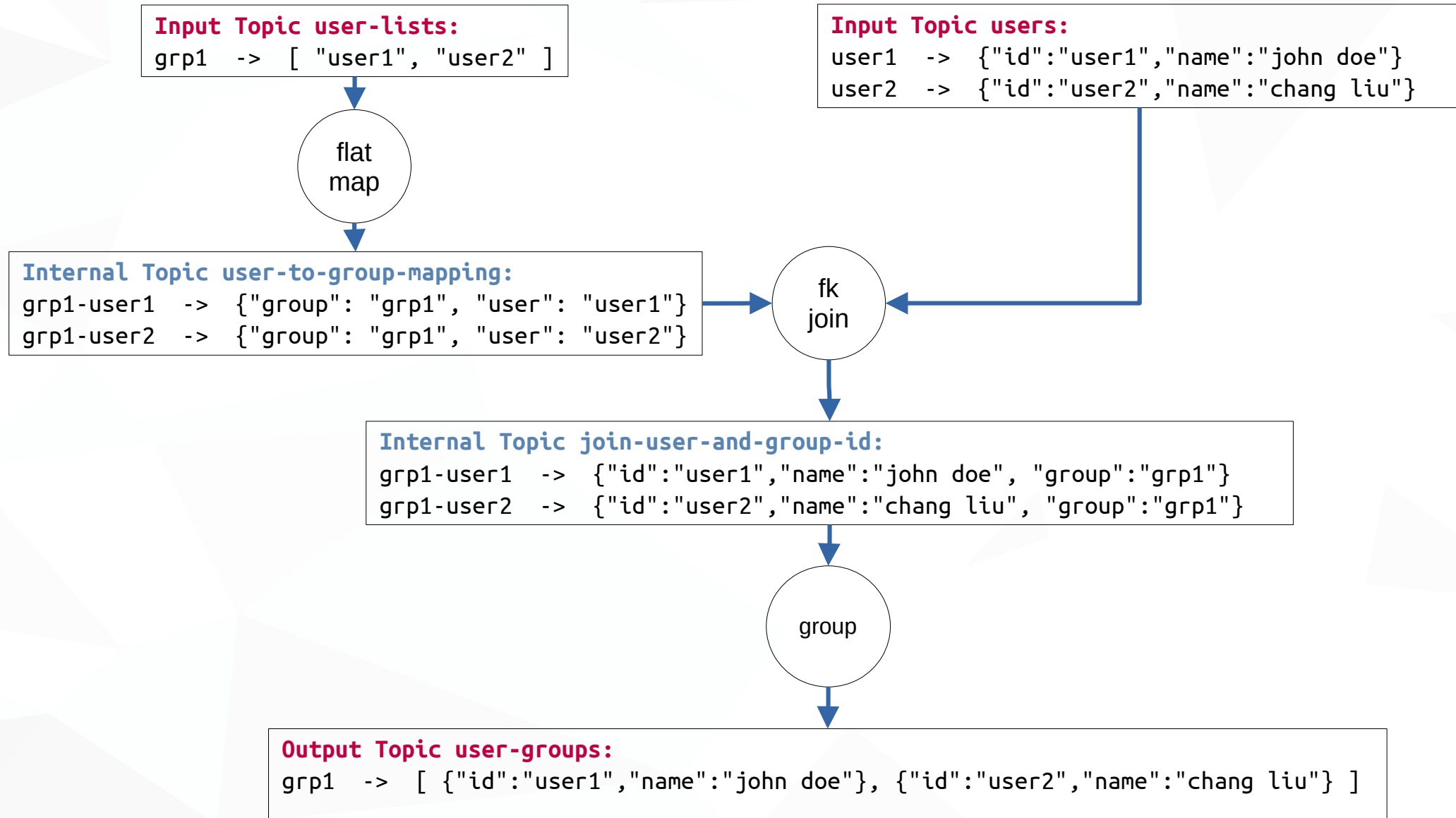
=> user1 gehört zu keiner Gruppe mehr
=> Gruppe 2 ist leer und wird auch gelöscht

Besondere Vorsicht bei:

- Key-Änderung
 - `KStream#map()`
 - `KStream#flatMap()`

Lösungen:

- geeignete Integrationstests
- E2E-Tests: Race Conditions können auch durch Interaktion mehrerer Streams-Apps entstehen
- Child-Entities nicht an Parent-Entities hängen
- nur solche Berechnungen in Kafka Streams implementieren, die sich leicht mit der Kafka Streams DSL abbilden lassen
- Nur eine Partition verwenden (verhindert horz. Skalierung)
- Geschicktes Key-Design: Beim Flat-Map Composite Key `grp1-user1` verwenden statt nur `user1`





Testing

```
public class UserListFlatMappingTopologyTest extends AbstractTopologyUnitTest {  
  
    // ... code left out  
  
    @Test  
    public void createsRecordForEachUserWithUserIdAsKeyAndGroupAsValue() {  
        inputTopic.pipeInput("grp1", "["user1", "user2", "user3"]");  
  
        var records = outputTopic.readKeyValuesToList();  
        assertThat(records).containsExactlyInAnyOrder(  
            KeyValue.pair("user1", "grp1"),  
            KeyValue.pair("user2", "grp1"),  
            KeyValue.pair("user3", "grp1")  
        );  
    }  
  
    // ... code left out  
}
```

```
public abstract class AbstractTopologyUnitTest {  
    private StreamsBuilder streamsBuilder;  
    private TopologyTestDriver testDriver;  
  
    @BeforeEach  
    public void setUp() {  
        streamsBuilder = new StreamsBuilder();  
  
        Topology topology = createTopology(streamsBuilder);  
  
        var streamsConfig = createStreamsConfig();  
        testDriver = new TopologyTestDriver(topology, streamsConfig);  
  
        createTestTopics();  
    }  
  
    protected abstract Topology createTopology(StreamsBuilder streamsBuilder);  
  
    protected abstract void createTestTopics();  
  
    // ... code left out  
}
```

Test basiert auf **TopologyTestDriver**

=> keine Partitionierung

=> keine Parallelisierung

```
@SpringBootTest
@Testcontainers
class RaceConditionIntegrationTest {

    @Container
    static final ConfluentKafkaContainer kafka =
        new ConfluentKafkaContainer(DockerImageName.parse("confluentinc/cp-kafka:7.6.1"));

    private static TestConsumer<List<User>> consumer;

    // ... code left out

    @BeforeAll
    public static void setUp() {
        KafkaAdmin adminClient = new KafkaAdmin(new HashMap<>() {{
            put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, kafka.getBootstrapServers());
        }});

        adminClient.createOrModifyTopics(TopicBuilder.name(USER_LISTS_INPUT_TOPIC)
            .partitions(5)
            .build());
        adminClient.createOrModifyTopics(TopicBuilder.name(USERS_INPUT_TOPIC)
            .partitions(5)
            .build());

        // ... code left out
    }
}
```

Teste Effekte der Parallelisierung

RaceConditionIntegrationTest.java

@Test

```
public void testRaceCondition() {
    int expectedGroupCount = 1000;

    // Create users and groups.
    // Each group contains one user
    // grp1 - [ "user1" ]
    // grp2 - [ "user2" ]
    for (int i = 1; i <= expectedGroupCount; i++) {
        String groupId = "grp" + i;
        User user = new User("user" + i, "john doe", null);
        List<String> userList = List.of(user.id());
        kafkaTemplate.send(USER_LISTS_INPUT_TOPIC, groupId, userMapper.serialize(userList));
        kafkaTemplate.send USERS_INPUT_TOPIC, user.id(), userMapper.serialize(user));
    }

    // Switch groups: each user goes to the next group
    // grp1 - [ "user1" ] ==> grp1 - [ ]
    // grp2 - [ "user2" ] ==> grp2 - [ "user1" ]
    // grp3 - [ "user3" ] ==> grp3 - [ "user2" ]
    for (int i = 1; i <= expectedGroupCount; i++) {
        String groupId = "grp" + i;
        kafkaTemplate.send(USER_LISTS_INPUT_TOPIC, groupId, "[ ]"); // first remove from old group
    }
    for (int i = 1; i <= expectedGroupCount; i++) {
        String newGroupId = "grp" + (i + 1);
        List<String> userList = List.of("user" + i); // then add to new group
        kafkaTemplate.send(USER_LISTS_INPUT_TOPIC, newGroupId, userMapper.serialize(userList));
    }

    // ... code left out
}
```

```
public void testRaceCondition() {
```

```
// ... code left out
```

```
Map<String, List<User>> userGroupMap = consumer.start(USER_GROUPS_OUTPUT_TOPIC);
```

```
await().atMost(10, SECONDS)
    .pollInterval(1, SECONDS)
    .until(() -> expectedGroupCount == userGroupMap.size());
```

```
await().atMost(10, SECONDS)
    .pollInterval(1, SECONDS)
    .until(() -> IntStream.range(1, expectedGroupCount+1)
        .allMatch(i -> purpleGroupMap.containsKey("grp" + (i+1))));
```

```
await().atMost(10, SECONDS)
    .pollInterval(1, SECONDS)
    .until(() -> IntStream.range(1, expectedGroupCount+1).allMatch(i ->
        userGroupMap.get("grp" + (i+1)).get(0).id().equals("user" + i)));
```

}




Internal Topic Naming

Interne Topics

Kafka Streams erzeugt interne Topics u.A. für:

- Repartitioning
- Changelog (State Store Daten)
- ...

```
KTable<String, String> usersTable = users.toTable(  
    Materialized.as("users-table") // for changelog topic  
);
```



**Interne Topics müssen explizit benannt werden,
sonst werden sie einfach durchnummeriert**

Gründe:

- Topologie debuggen (z.B. eine erwartete Message ist nicht im Output-Topic. Wo "geht sie verloren"?)
- Bei Veränderung der Topologie ändert sich die Nummerierung der Topics => State kann nicht geladen werden

Maßnahmen

- Topologie nach unbenannten Topics scannen (TopicNamingTest.java)

TopicNamingTest.java

Topologies:

Sub-topology: 0

Source: KSTREAM-SOURCE-0000000000 (topics: [user-lists])

--> stateful-flat-map

Processor: stateful-flat-map (stores: [flat-map-user-id-list])

--> KSTREAM-FILTER-0000000006

<-- KSTREAM-SOURCE-0000000000

Processor: KSTREAM-FILTER-0000000006 (stores: [])

--> KSTREAM-SINK-0000000005

<-- stateful-flat-map

Sink: KSTREAM-SINK-0000000005 (topic: KSTREAM-TOTABLE-0000000003-repartition)

<-- KSTREAM-FILTER-0000000006

Sub-topology: 1

Source: KSTREAM-SOURCE-0000000001 (topics: [users])

--> users-table

Source: KSTREAM-SOURCE-0000000007 (topics: [KSTREAM-TOTABLE-0000000003-repartition])

--> KSTREAM-TOTABLE-0000000003

...

```
@Test
void assertThatTopicNamesWereSetManually() {
    Topology topology = streamsBuilder.build();
    String topologyString = topology.describe().toString();

    System.out.println(topologyString);

    // ... code left out
}
```



Automatisch
benannte
interne Topics



Weitere Fallstricke

Data Explosion: explosionsartige Vermehrung der Datenmenge durch die Verarbeitung

Ursachen:

- implementierte Logik, z.B.
 - kartesische Produkte
 - Flattening von Datenstrukturen
- Probleme mit State Stores/Record Cache
- ...

```
@SpringBootTest
@Testcontainers
@TestPropertySource(properties = "spring.kafka.streams.properties.statestore.cache.max.bytes=0")
public class DataExplosionIllustration {

    @Test
    public void illustrateDataExplosion() throws InterruptedException {
        List<String> group1 = List.of("user1", "user2", "user3", "user4",
                                   "user9", "user10");
        List<String> group2 = List.of("user11", "user12", "user13", "user14",
                                   "user19", "user20");

        // ... code left out

        // Update the user lists
        kafkaTemplate.send(USER_LISTS_INPUT_TOPIC, "group1", userMapper.serialize(group1));
        kafkaTemplate.send(USER_LISTS_INPUT_TOPIC, "group2", userMapper.serialize(group2));

        // ... code left out

        AtomicInteger outputMessageCount = consumer.countMessages(USER_GROUPS_OUTPUT_TOPIC);

        Thread.sleep(10 * 1000); // Wait for processing

        System.out.println("Number of consumed messages: " + outputMessageCount);
    }
}
```

Simuliert Problem mit Record Cache
(keine Compaction)

Number of consumed messages: ≥ 20

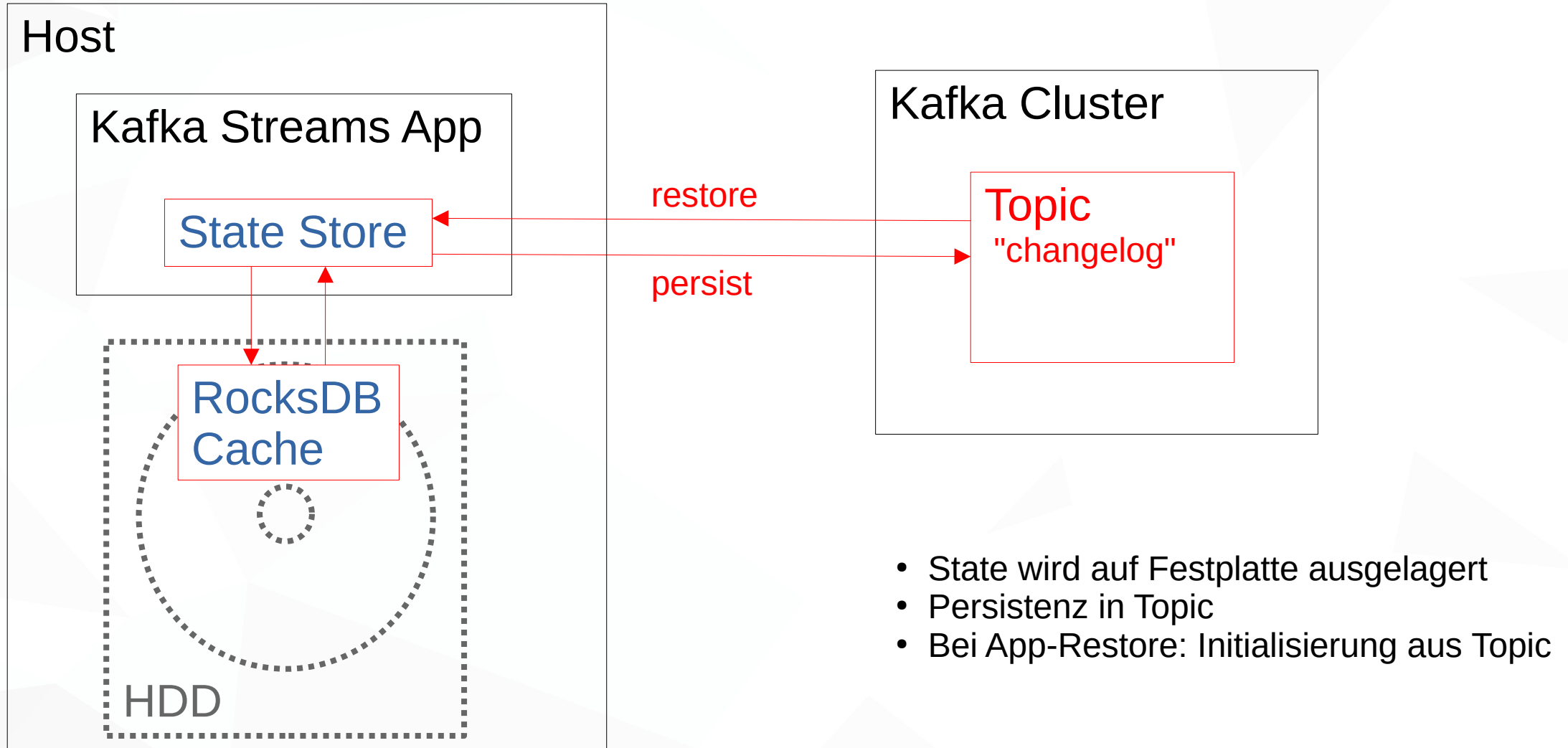
Maßnahmen:

- Monitoring
 - Producer-Metrik records-sent beobachten
 - Record Cache monitoren
- Vorsicht bei kartesischen Produkten (z.B. foreign key join) und Flattening von Datenstrukturen
- Deduplikation implementieren

- AVRO-Schemas nicht für Keys nutzen!
 - AVRO-Messages enthalten Metadaten!
 - Metadaten wirken sich auf Partitionierung aus
 - Konkretes Szenario: doc-Feld im Schema ändern
 - => Schema ID ändert sich
 - => Partitionierung ändert sich, obwohl Key Payload gleich bleibt
- RecordTooLargeException
 - Message Size Limits überschritten
 - Bei groupBy+aggregate ausprobieren, ob Record zu groß werden kann
 - ggf. Limits erhöhen (Broker: `message.max.bytes`, App: `max.request.size`)



State Stores



Vorteile

- State kann über das Memory-Limit hinaus gehen

Nachteile

- Hohe Komplexität der State Stores, incl. zahlreicher Settings
- Mögliche Probleme sind z.B.
 - Hoher Festplattenverbrauch
 - Hohe Disc I/O
 - Zu viele File Handles

Maßnahmen

- "keep the number of state stores on a single node under 30" (m5.xlarge + EBS in AWS)¹
 - unsere Beispieltopologie hat bereits 5!!!
 - verarbeitet eine App mehrere Partitionen, startet sie auch mehrere Instanzen der Topologie mit eigenen State Stores!!!
 - ggf. horizontale Skalierung nötig
- Monitoring: Siehe Metriken und weiterführende Links in README.md
- Fine-Tuning über zahlreiche Konfigurationsmöglichkeiten
- Ggf. in-memory-state-stores verwenden

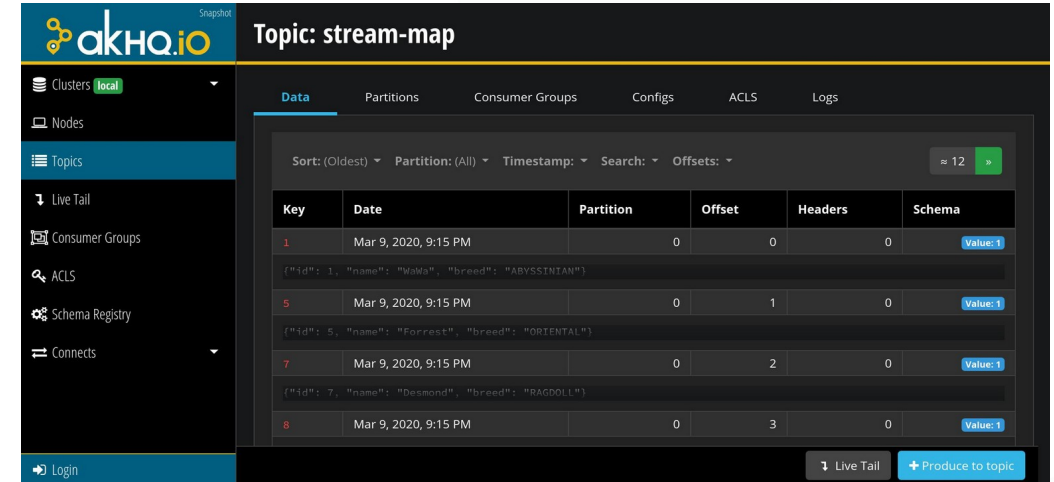
1) Almog Gavra "Don't Panic: The Definitive Guide to Kafka Streams State"

<https://www.responsive.dev/blog/guide-to-kafka-streams-state>, abgerufen am 05.05.2025, 14:30



Monitoring & Observability

- Gute OSS Tools:
 - AKHQ <https://akhq.io/docs/>
 - Prometheus
 - Grafana
- Metriken aus verschiedenen Quellen verfügbar
 - kafka-streams
 - kafka-clients
 - Kafka Cluster
- Liste wichtiger Metriken, Link zu fertigen Grafana Dashboards aus der Community, sowie weiterführende Links in <https://github.com/selbstereg/kafka-streams-example-etka25/blob/main/README.md>



| Key | Date | Partition | Offset | Headers | Schema |
|-----|----------------------|-----------|--------|---------|----------|
| 1 | Mar 9, 2020, 9:15 PM | 0 | 0 | 0 | Value: 1 |
| 5 | Mar 9, 2020, 9:15 PM | 0 | 1 | 0 | Value: 1 |
| 7 | Mar 9, 2020, 9:15 PM | 0 | 2 | 0 | Value: 1 |
| 8 | Mar 9, 2020, 9:15 PM | 0 | 3 | 0 | Value: 1 |



Prometheus



Grafana



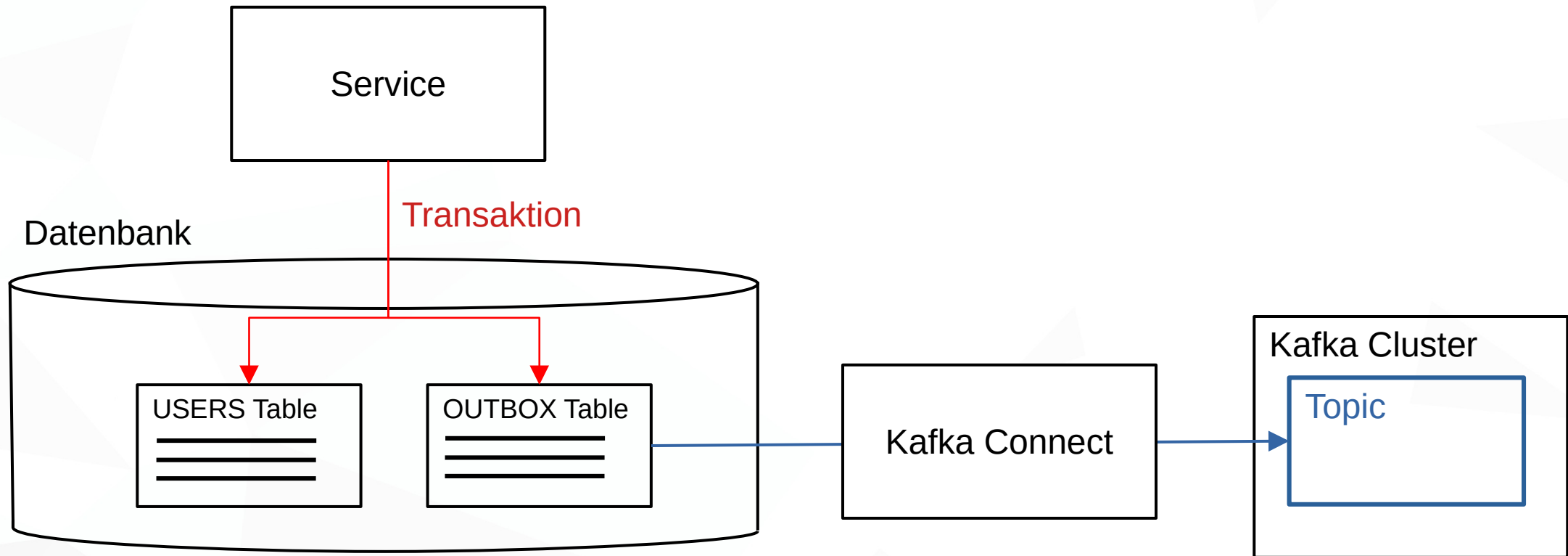
Integration mit angrenzenden Systemen

Kafka Connect

- mächtiges Tool zur Integration von Kafka mit externen Systemen
- Source- und Sink-Connectors ermöglichen Import/Export
- großes Ökosystem von fertigen Connectors z.B. für
 - Relationale Datenbanken
 - S3
 - Elasticsearch
 - ...

Outbox Pattern

In einer atomaren Operation sowohl Datenbanktabelle updaten, als auch Message schicken





Fazit

Kafka Streams ist komplex, deshalb:

- Anforderungen klären:
 - Welche Features nutzen wir, um welche Anforderung zu erfüllen?
- Nur den Teil des Systems in Kafka Streams implementieren, der diese Features benötigt
- Realistische Integrationstests und ggf. E2E-Tests schreiben
- Von Anfang an Monitoring & Observability implementieren

Vielen Dank für die Aufmerksamkeit :)



<https://github.com/selbstereg/kafka-streams-example-etka25.git>