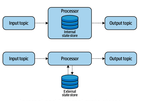
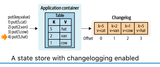
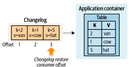
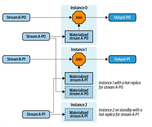
Adam Bellemare - Building Event-Driven Microservices

- 7. Stateful Streaming
  - State Stores and Materializing State from an Event Stream
    - Materialized state
      - A projection of events from the source event stream (immutable)
    - State store
      - Where your service's business state is stored (mutable)
      - 

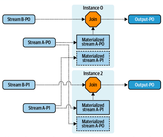
      - There are 2 types
        - Internal and External state stores
  - Recording State to a Changelog Event Stream
    - 

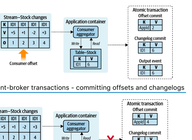
      - A state store with changelogging enabled
    - 

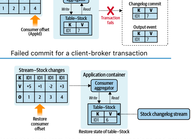
      - State store being restored from a changelog
  - Materializing State to an
    - Internal State Store
      - Global state store
        - Materializes the data of ALL partitions for a given event stream
        -

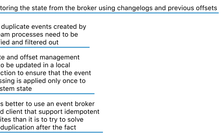
        - Global materialized state and nonglobal materialized state
      - Advantages
        - Scalability requirements are offloaded from the developer
        - High-performance disk-based options
        - Flexibility to use network-attached disk
      - Disadvantages
        - Limited to using runtime-defined disk
        - Wasted disk space
      - Scaling and Recovery
        - Using HOT REPLICAS
          - 

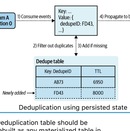
            - Stream-table join with three instances and two hot replicas per materialized input partition
            -

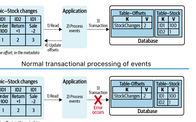
            - Rebalance due to instance 1 termination
            - 

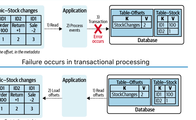
            - Normal processing with two instances and two hot replicas per materialized input partition
            - ⚠ Tradeoff: use of additional disk to maintain the replicas in exchange for the reduction in downtime due to an instance failure
          - Restoring and scaling FROM CHANGELOGS
            - 👎 Downtime
          - Restoring and scaling FROM INPUT EVENT STREAMS
            - Re-consume all of its input events from the very beginning of its assigned event stream partitions
            - 👎 Downtime
            - 👎 Can take much longer to rebuild state than restoring from a changelog
    - External State Store
      - I believe it is about ideal world. In real life we CAN share state (db) to save time or resources!
      - 👎 Do not share direct state access with other microservices.
      - 👍 Advantages
        - Full data locality
          - Can provide access to all materialized data for each microservice instance (any joins can be executed)
            - though each instance is still responsible for materializing its own assigned partitions
        - Technology
          - Can leverage technology that the organization is already familiar with
      - 👎 Drawbacks
        - Management of multiple technologies
          - ⚠ Each team must fully manage the external state stores for its microservices
        - Performance loss due to network latency
        - Financial cost of external state store services
        - Full data locality
          - For example, one instance may attempt to join an event on a foreign key that has not yet been populated by a separate instance
      - Scaling and Recovery
        - Using the source streams
        - Using changelogs
        - Using snapshots
          - 👍 Most widely used
          - ⚠ If the stored state is not idempotent and any duplicate events are not acceptable, then you should store your consumer's partition offsets alongside the data within the data store.
  - Rebuilding vs Migrating State Stores
    - Rebuilding
      - 👍 the state is built exactly as specified by the new business logic
      - 👎 it takes time
    - Migrating
      - 👍 good when no need to reprocess older data
        - e.g. new field added
      - 👎 complex migrations are riskier
  - Transactions and Effectively Once Processing
    - exactly once = effectively once
    - 👍 Idempotent writes (Idempotent production)
      - Commonly supported feature among event brokers
      - ⚠ It cannot mitigate duplicates introduced through faulty business logic
    - 👍 Effectively Once w/ Client-Broker Transactions
      - may also be supported by your event broker
        - 👍 currently, full transactional support is offered only by Apache Kafka
      - 

        - Client-broker transactions - committing offsets and changelogs
      - 

        - Failed commit for a client-broker transaction
        - 

        - Restoring the state from the broker using changelogs and previous offsets
    - Effectively Once w/o Client-Broker Transactions
      - 1. Any duplicate events created by upstream processes need to be identified and filtered out
      - 2. State and offset management need to be updated in a local transaction to ensure that the event processing is applied only once to the system state
      - ⚠ It is better to use an event broker and client that support idempotent writes than it is to try to solve deduplication after the fact
      - ⚠ What can generate duplicate events
        - Producer fails to receive acknowledgment from broker and retries
        - Producer crashes immediately after writing, before updating its own consumer offsets
      - Identifying duplicate events
        - Determine if the duplicates actually cause any problems
          - In many cases they don't
        - How to do this?
          - Generate unique ID for each event
            - 👍 such that any duplicates will generate the same unique hash
      - Guarding against duplicates
        - Perfect deduplication requires that each consumer indefinitely maintain a lookup of each dedupe ID already processed
          - but it is usually too expensive
        - Can use rolling time-window or offset-window as a best-effort attempt
          - 👍 Use time-to-live (TTL), a maximum cache size, and periodic deletions
        - 

        - Deduplication using persisted state
        - ⚠ Deduplication table should be rebuilt as any materialized table in case of failure
      - Maintaining consistent state
        - 

          - Normal transactional processing of events
          - 

            - Failure occurs in transactional processing
          -

            - Recovery of offsets during state restoration process