Topics

Pay

Description:

It will hold all messages needed to checkout or rollback a transaction pay

Producers:

- order/checkout: produces initial checkout payment statement
- order consumer: produces a rollback-increment statement (when needed)

Consumers:

- pay consumer: deals with committing changes in durable storage

Messages:

- pay (key: "orderid_tr#", value: ('pay', amnt))
- cancelPayment (key: "orderid_tr#", value: ('cancel_pay', amnt))

Stock

Description:

It will hold all messages needed decrement or increment all quantities needed for a transaction. Per transaction there are at most two messages: - one to decrement quantities for all items in an order - the compensation message to increment quantities for all items in an order

Producers:

- order/checkout: produces initial checkout decrement statement
- order consumer: produces a rollback-increment statement (when needed)

Consumers:

- stock consumer

Messages:

- decrement (key: "orderid_tr#", value: ('decrement', [(item, amnt), ...]))
- increment (key: "orderid_tr#", value: ('increment', [(item, amnt), ...]))

Outcome P&S (ops)

Description:

It holds success or failure messages produced by pay consumer and stock consumer relative to their changes in db

Producers:

- pay consumer
- stock consumer

Consumers:

- order consumer

Messages:

- paySuccess (key: "orderid_tr#", value: ('psuccess', amnt))
- payFail (key: "orderid_tr#", value: ('pfail', amnt))
- stockSuccess (key: "orderid_tr#", value: ('ssuccess', [(item, amnt), ...]))
- stockFail (key: "orderid_tr#", value: ('sfail', [(item, amnt), ...]))

Outcome checkout (oc)

Description:

It holds messages relative to the success or failure of transactions as a whole

Producers:

- order consumer

Consumers:

- order/checkout

Messages:

- success (key: "orderid_tr#", value: 'success')
- fail (key: "orderid_tr#", value: 'fail')

Offsets Outcome P&S (oops)

Description:

Per partition it holds the offset of the start of the oldest transaction not fully processed. Order consumers have to:

- read messages from **ops**
- process the message and commit to kafka (increment internal offset)
- Keep tract of all the transaction ids and the offset of their first message (an outcome from pay or stock)
- Once the oldest transaction is done (we received both messages from stock and pay)

    - no need to keep track of that transaction any more
    - push to Offset **ops** the next oldest (smallest) offset
- If a transaction that is not the oldest is done: just lose track of it

When first initializing and after rebalancing, consumers have to fetch the latest offset posted in oops, and read messages until they reach the committed offset in **ops**, **without** processing them, just in order to rebuild the lost state. When latest committed offset is reached, then they have to start processing messages, sending response messages like normal

Producers:

- order consumer

Consumers:

- order consumer (when initializing, needed for fault recovery)

Messages:

- offset (key: partition#, value: offset)

Participants

order/checkout

Topics:

- **stock**: producer
- **pay**: producer
- **oc**: consumer

order consumer

Topics:

- **ops**: consumer
- **oc**: producer
- **oops**: producer and consumer

pay consumer

- **pay**: consumer
- **ops**: producer

stock consumer

- **stock**: consumer
- **ops**: producer

Important notes

- **tr#** is the number of times order/checkout is called, we need this to deal with same messages per same order (namely: checkout -> fail -> cheange something in order -> checkout again). This means that the order table has to be changed like this:

OrderId	user_id	paid	items	amount	tr#

- For better throughput we can set autocommit to true for kafka consumers. This ensures that messages are read at least once in all partitions, but not that they are read exactly once. To deal with this we can insert the orderid_tr# along with the meaning of the message (pay/decrement vs rollback). This means that we need to create the following table in stock and payment dbs:

Orderid	tr#	Was Rollback
1	2	False
1	2	True

  When reading a message, when doing the transaction we can do a conditional insert to the database, and go through on if the key is not already in db. All three attributes have to compose the primary key