Stream Processing with Faust

What is Faust?

- Faust was created at Robinhood
- Design goal was to replicate Kafka Streams in Python
- Shares conceptual design patterns with Kafka Streams
- Stream Processing natively written in Python
- Concepts applicable in other frameworks as well

Introduction to Faust

Writing Faust apps is simple and requires no external dependencies

```
import faust

app = faust.App("demo-app", broker="PLAINTEXT://localhost:9092")
topic = app.topic(“a-kafka-topic”)

@app.agent(topic)
async def purchase(purchases):
    async for purchase in purchases:
      # do something!

if __name__ == "__main__":
    app.main()
```

Hello Faust

Writing Faust apps is simple and requires no external dependencies

```
import faust

app = faust.App("demo-app", broker="PLAINTEXT://localhost:9092")

topic = app.topic("a-kafka-topic")

@app.agent(topic)
async def purchase(purchases):
    async for purchase in purchases:
      # do something!

if __name__ == "__main__":
    app.main()
```

Demo: Your First Faust Application Demonstration

Serialization and Deserialization

Python Dataclasses

Python 3.7 Dataclasses are used only for containing data with type hints

```
from dataclasses import dataclass


@dataclass(frozen=True)
class Purchase:
    username: str = “”
    currency: str = “”
    amount: int = 0
```

Faust Deserialization

Deserialization is handled by specifying a type to the Faust topic

```
import faust

class Purchase(faust.Record, validation=True, serializer="json"):
    username: str
    currency: str
    amount: int

app = faust.App("demo-app", broker="PLAINTEXT://localhost:9092")
topic = app.topic("purchases", key_type=str, value_type=Purchase)

@app.agent(topic)
async def purchase(purchases):
    async for purchase in purchases:
        # do something!
```

Demo: Deserialization

Faust Serialization

Faust Serialization shares the same semantics as Deserialization

```
import faust

class Purchase(faust.Record, validation=True, serializer="binary|json"):
    username: str
    currency: str
    amount: int

app = faust.App("demo-app", broker="PLAINTEXT://localhost:9092")
topic = app.topic("purchases", key_type=str, value_type=Purchase)

@app.agent(topic)
async def purchase(purchases):
    async for purchase in purchases:
        # do something!
```

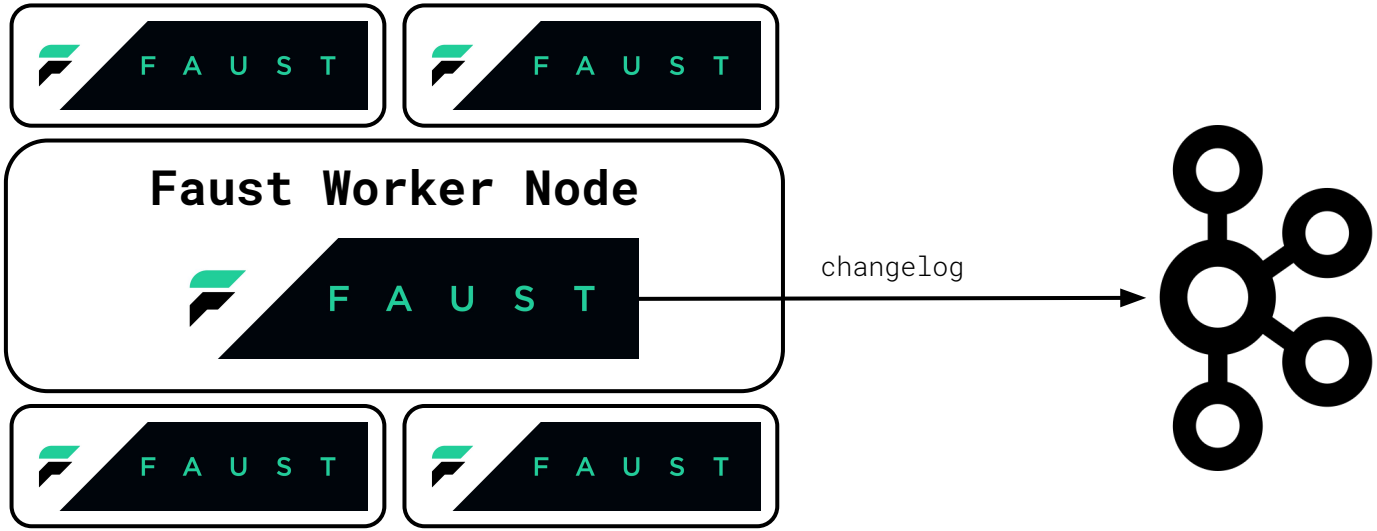
Demo: Serialization

Serialization and Deserialization Summary

Storage Overview

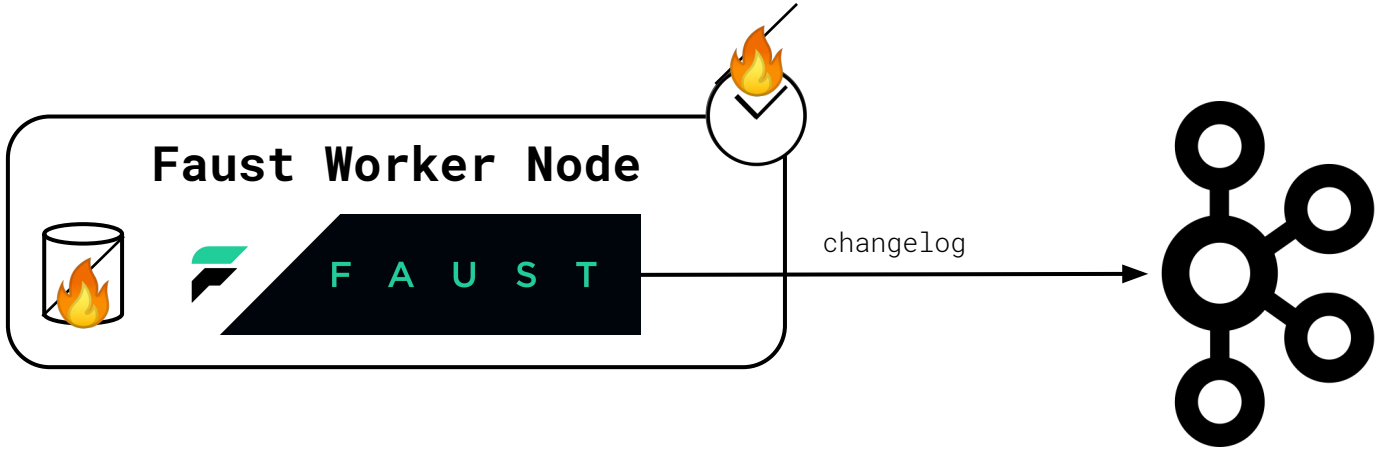
Kafka State

Faust stores its changelog in Kafka, and uses RocksDB for local state



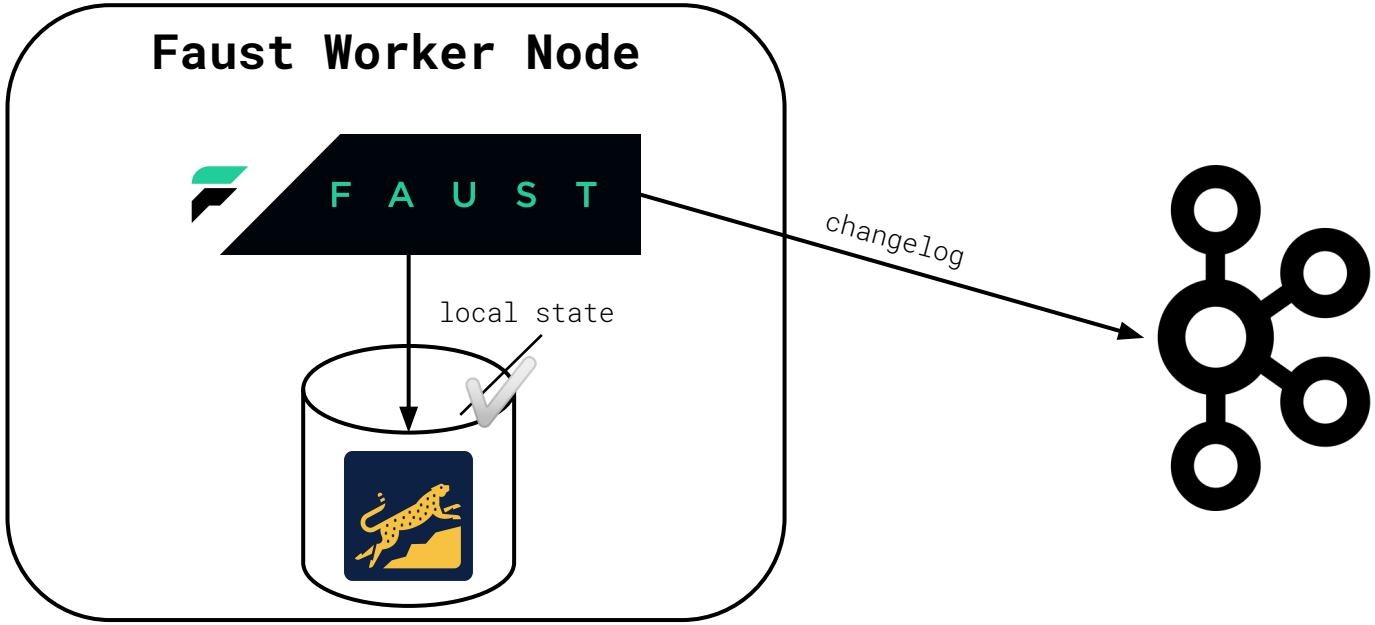
In-Memory

In-Memory storage should only be used for test and local development



RocksDB

Faust uses RocksDB to store local state in Production



Streams

Streams Basics

Streams are automatically created for agent functions

```
source_topic = app.topic('source-topic')

@app.agent(source_topic)

async def process(stream):

    async for value in stream:

        # do something
```

Streams Basics

Results of stream operations are placed in a defined output topic

```
source_topic = app.topic('source-topic')

dest_topic = app.topic(‘dest-topic’)

@app.agent(source_topic)

async def process(stream):

    async for num_msg, demo_value in stream.enumerate():

        await dest_topic.send(key=num_msg, value=demo_value)
```

Faust Streams

Message Life Cycle and Acknowledgment

- Faust manages consumer groups and offsets
- Faust uses the `aiokafka` to interact with Kafka
- `aiokafka` manages the consumer and offsets
- Uses one underlying subscription to topics for all agents

Demo: Streams in Action

Processors

Processor functions can be chained into streams to transform data

```
def transform(input_model):
    return transform(input_model)

async def delete_field(input_model):
    return delete_pii(input_model)

demo_stream = app.stream(
    demo_topic,
    processors=[transform, delete_field]
)
```

Operations

Faust Streams

- Operations are helpers for processing streams
- `group_by` re-partitions the input stream on a new key
- `filter` removes unwanted data

Demo: Processors and Operations

Streams Summary

Tables

Tables

Tables are the result of aggregations and have `dict`-like syntax

```
purchases_topic = app.topic('purchases', value_type=Purchase)

currency_table = app.Table('currency-table', default=int)

@app.agent(purchases_topic)
async def process(purchases):
    async for purchase in purchases.group_by(Purchase.currency):
        currency_table[purchase.currency] += purchase.amount
```

Demo: Tables in Action

Windowing in Faust

Faust supports Tumbling and Hopping Windows

```
table = app.Table("currency_total", int)
# Tumbling Window
tumbling_table = table.Tumbling(size=timedelta(minutes=5))
# Hopping Window
hopping_table = table.Hopping(
    size=timedelta(minutes=5),
    step=timedelta(minutes=1),
    expires=timedelta(minutes=60),
)
```

Demo: Tumbling Windows

Demo: Hopping Windows

Faust Summary