NSTREAM

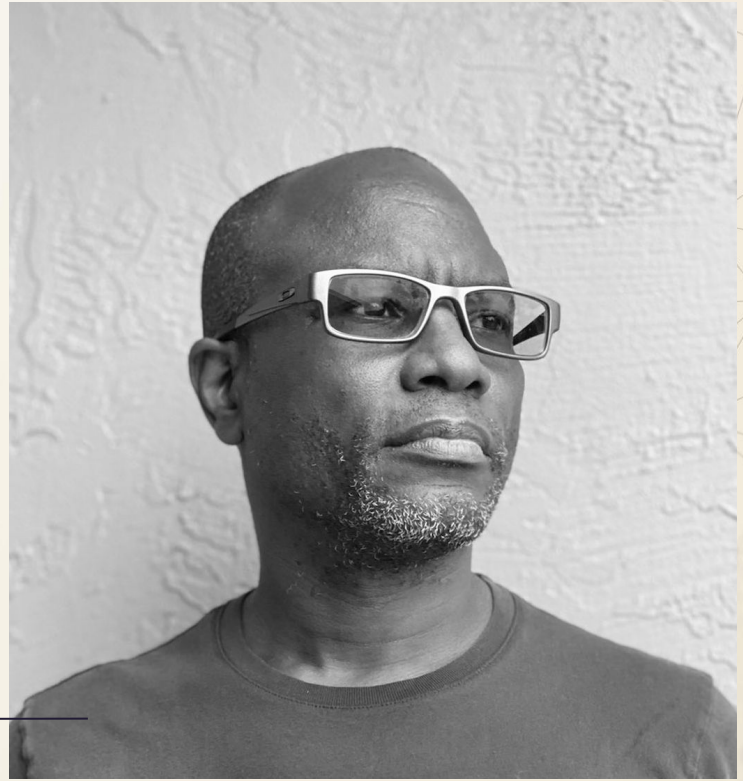
Going Customer 360 with Kafka, Flink, & SwimOS

Real-time data streaming for entities & events

03 / 17 / 2023

Fred Patton

- Nstream Developer Evangelist
- Long-time backend engineer, generalist, and polyglot
- Serverless and fullstack by night
- Avid technologist
- Lover of the arts and various spoken languages

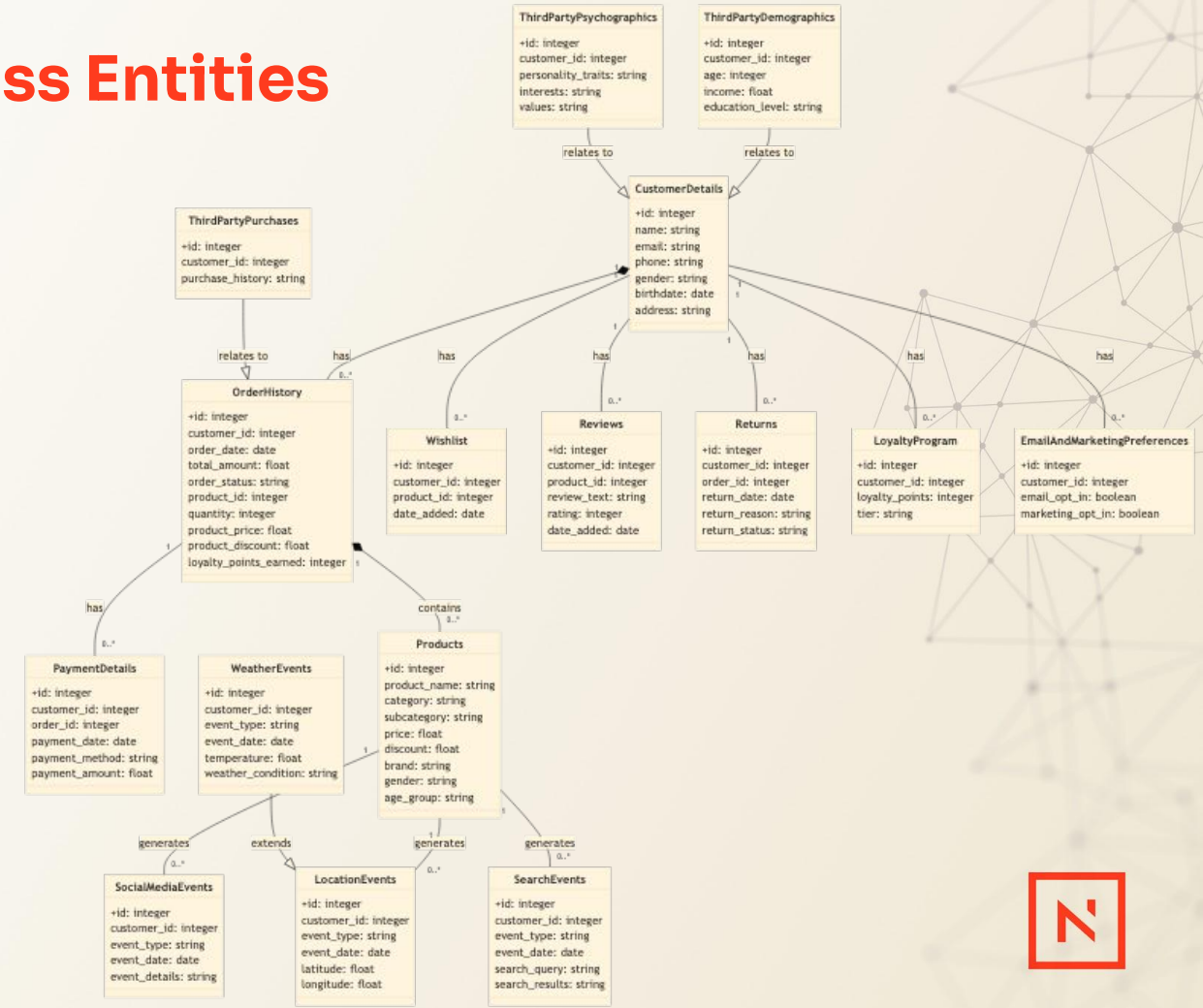


Customer 360 Use Case for

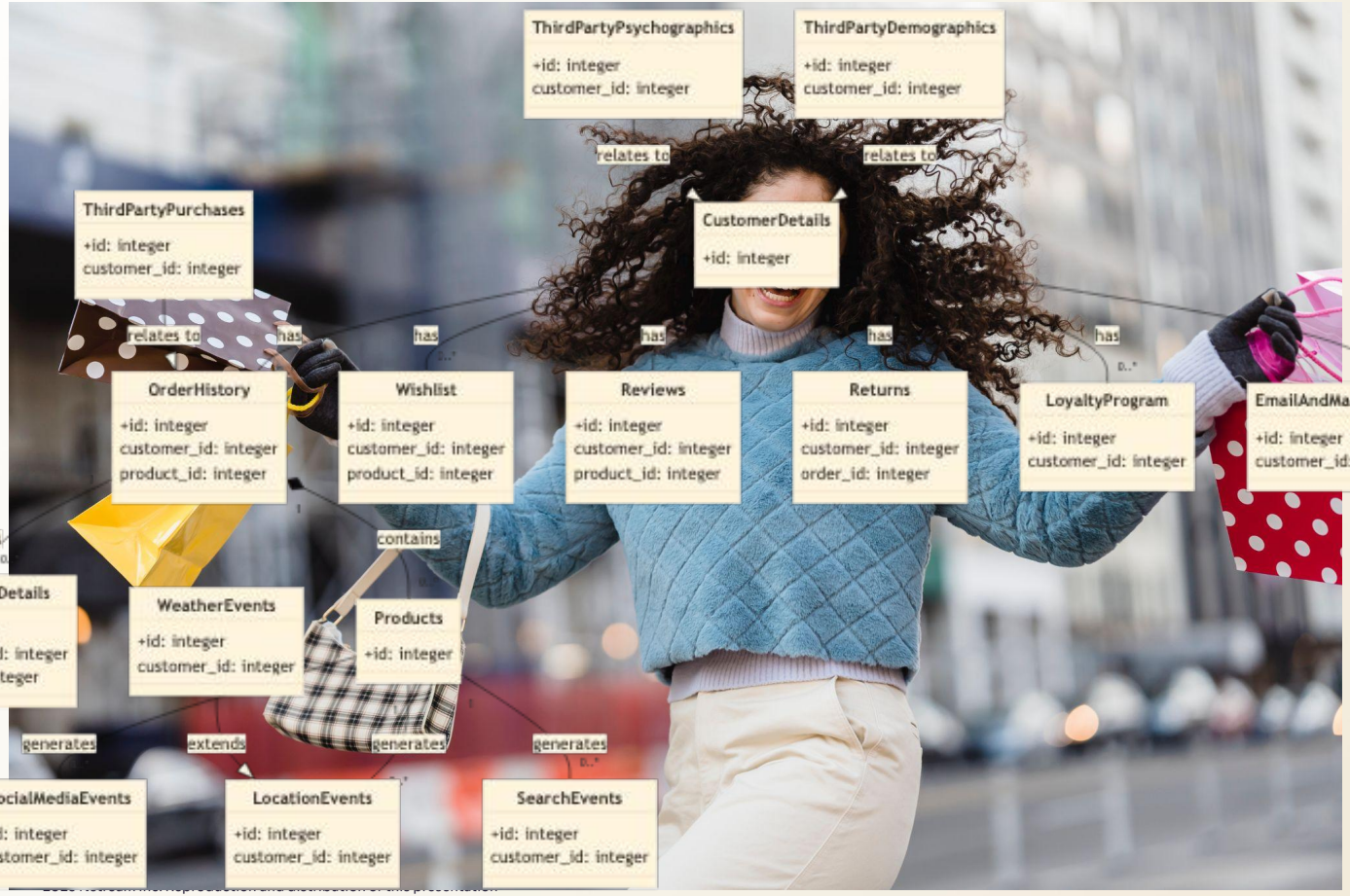
large online fashion retailer

- Purchase history aware
- Browse/Search history aware
- Social Media action aware
- Geo-location aware
- Inventory aware
- Weather aware
- Physical event aware
- Third-party data aware

Relevant Business Entities



Relevant Business Entities



Focal streams

- Customer details
- Order history
- Wishlist
- Loyalty program stats
- Products
- Weather events
- Social media events
- Location events
- Search (and browse) events
- Third-party purchases
- Third-party demographics
- Third-party psychographics

Taking Customer 360 to Its Logical Conclusion

CRMs with up to the moment information, and decision automation in lockstep with analytics

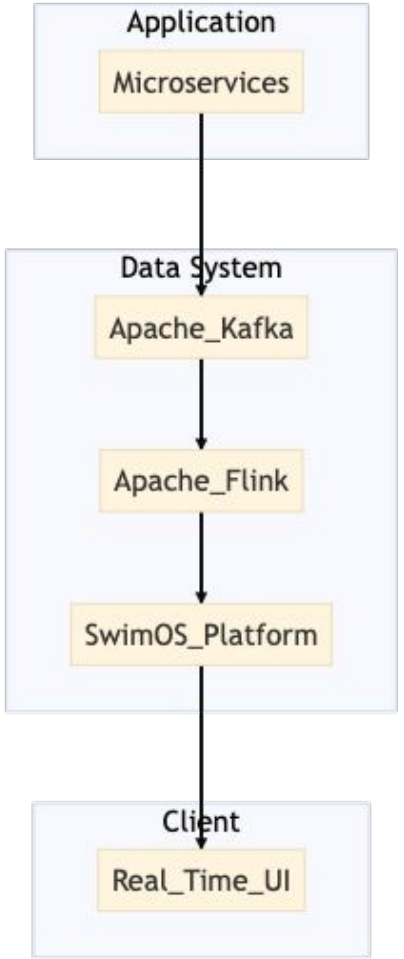
“Righteous Analytics: Right Data. Right Time. Right Way — David W. Johnson”

The right time is…

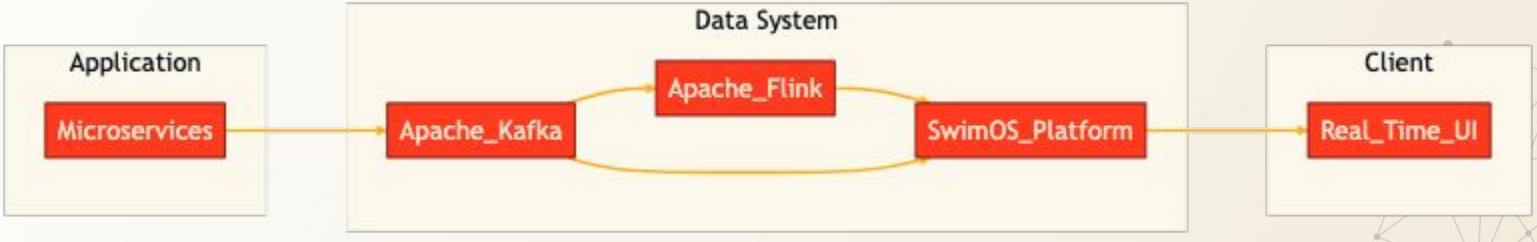
- **Within days ✅**
- **Within hours ✅**
- **Within minutes ✅**
- **Within seconds 😲**
- **Within milliseconds 😱**

Kafka + Flink + SwimOS

ETL + Analytics + Business Logic



High-Level Architecture



We begin with an event-driven application where microservices store events in Kafka. Flink will be used to provide enrichments for analytics, such as windowing, summations, and aggregations. SwimOS will ingest application events from Kafka along with analytics results from Flink and apply business logic in the same real-time. Where a moment later, a real-time UI will consume streaming entity state directly from SwimOS, despite being outside the network.

Entities

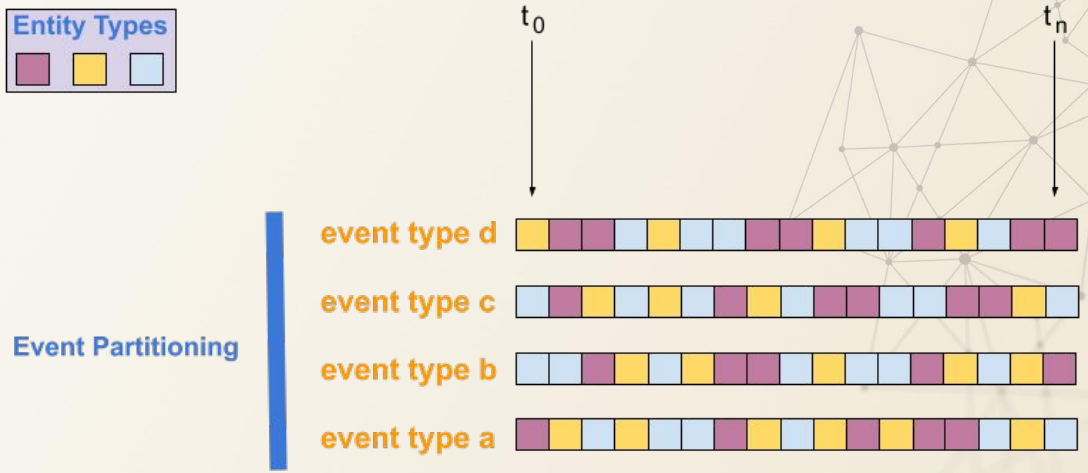
- Simple object state transmitted as a binary diff to its consumers
- Multiplexed with other entity state needed by respective consumers
- Consumer is notified via an update handler at which point the binary diff has already been applied
- Consumer simply executes its specific logic in the same way as it would without streaming

Events

- Event is replicated to its consumers
- The event captures the minimal fields needed to make record of an occurrence with fidelity
- Each consumer processes the given event and takes appropriate action
- Any information not contained in the event has to be retrieved somehow adding latency

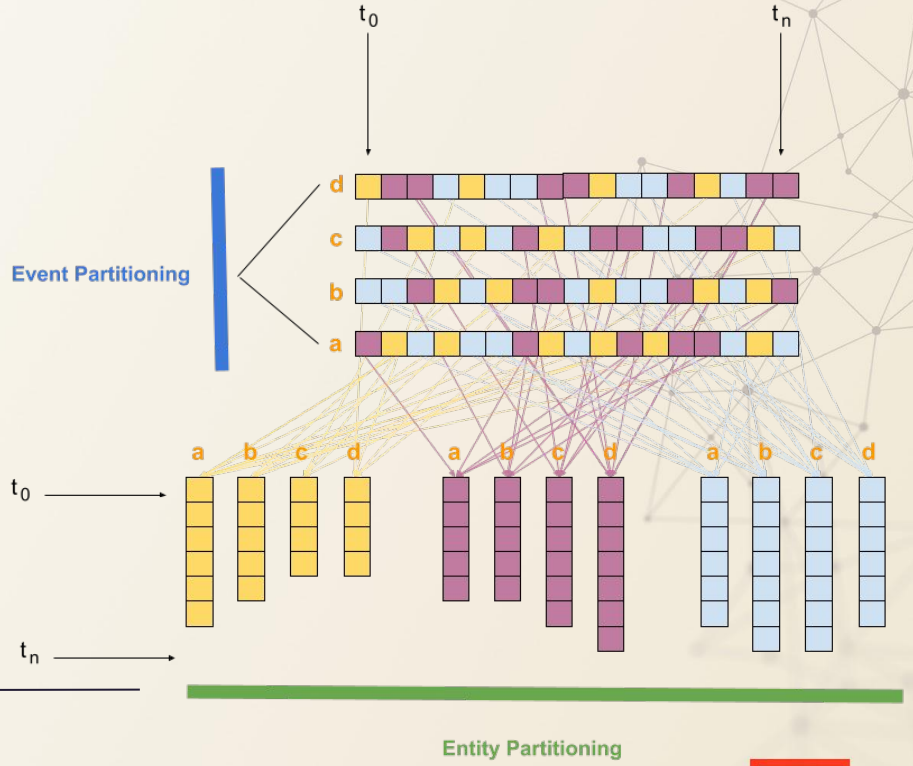
Real-Time Event Data

- The application and analytics data streams are made up of events partitioned by event type
- This is standard operating procedure to maximize throughput for ETL and analytics
- It is also critical to optimize for data-loss prevention



Real-Time Entity Data

- Streams coming out of SwimOS contain entity state partitioned by entity ID so there is no need for stream joins
- Authorized consumers outside the network may sync entity data in real-time even at the field level
- Consumers may also subscribe to lifecycle methods to get context beyond state changes
- Consumers may also invoke commands that stream back the other way



Real-Time Data Observability

- Data does not accumulate unbounded, rather it moves through a window of relevance for business logic processing
- Entity-specific business logic dictates the point at which information decays, either immediately or scaled within the step function of roll-ups
- Data that will never be stored can still be processed, in which case the extracted value can be conditionally stored

Exhibit A: An Intractable Many Stream Join

To add insult to injury, this 501-line massive stream join in Flink can't even lift itself off the tarmac let alone satisfy any real-time SLAs. It is not Flink's fault. It is the wrong algorithm for the problem.

```
// Join all datasets using customer_id as the join key
DataSet<Tuple20<Integer, String, String, String, String, String, String, Integer, String, Float, String, Integer,
                String, Float, Float, Integer, Integer, String, String, String>> joinResult =
    customerDetails
    .join(orderHistory)
    .where(new KeySelector<CustomerDetails, Integer>() {
            @Override
            public Integer getKey(CustomerDetails customer) throws Exception {
                return customer.id;
            }
        })
    .equalTo(new KeySelector<OrderHistory, Integer>() {
            @Override
            public Integer getKey(OrderHistory order) throws Exception {
                return order.customer_id;
            }
        })
    .join(paymentDetails)
    .where(new KeySelector<Tuple2<CustomerDetails, OrderHistory>, Integer>() {
            @Override
            public Integer getKey(Tuple2<CustomerDetails, OrderHistory> tuple) throws Exception {
                return tuple.f1.id;
            }
        })
    .equalTo(new KeySelector<PaymentDetails, Integer>() {
            @Override
            public Integer getKey(PaymentDetails payment) throws Exception {
                return payment.order_id;
            }
        })
    // ... omitting the bulk of this 501-line massive query
```

Exhibit B: Entity Partitioning w/ Differential State Sync

Here, we find simple application-style logic, where a function called runBusinessLogic is an application defined method that re-runs its business logic when associated state changes. In this case, the logic does not have to be expressed in terms of mathematical operators, but is a simple user defined function. Outside of the user defined method, the code is less than 200 lines.

```
@Override
public void didStart() {
    super.didStart();
    orderHistoryJoin = this.<Integer, OrderHistory>joinValueLane()
            .didUpdate((key, newValue, oldValue) -> {
                runBusinessLogic(ORDER_HISTORY_CHANGED);
            });

    paymentDetailsJoin = this.<Integer, PaymentDetails>joinValueLane()
            .didUpdate((key, newValue, oldValue) -> {
                runBusinessLogic(PAYMENT_DETAILS_CHANGED);
            });

    wishlistJoin = this.<Integer, Wishlist>joinValueLane()
            .didUpdate((key, newValue, oldValue) -> {
                runBusinessLogic(WISHLIST_CHANGED);
            });

    reviewsJoin = this.<Integer, Reviews>joinValueLane()
            .didUpdate((key, newValue, oldValue) -> {
                runBusinessLogic(REVIEWS_CHANGED);
            });
```

About Nstream...

Nstream has been pursuing a vision of a real-time streaming Internet powered by Web Agents — web-addressable as the name suggests — and delivering on much of the promise of Digital Twins and Distributed Actors along the way.

https://www.linkedin.com/pulse/state-streaming-data-christopher-sachs

Get started!

- READMEs: https://github.com/swimos/swim
- Developer site: https://www.swimos.org/
- Sample Applications: https://github.com/swimos
- Cookbook: https://github.com/swimos/cookbook
- Tutorials: https://github.com/swimos/tutorial
- Yours truly: https://www.linkedin.com/in/thoughtpoet/

Thank You!