2026

Yaroslav Tkachenko

Streamling: Building A Lightweight Data Streaming Runtime On Apache Datafusion

DataFusion Community Showcase

Yaroslav Tkachenko

About me

- → Software Engineer, data/infra
- → Technical Lead at **Activision**, **Shopify**
- → Founding Engineer at **Goldsky**
- → Founder at **Irontools**

Current:

- → Data streaming **consulting**, **advising** and **training**
- → ❤️Arrow and DataFusion

OPEN SOURCE, BY GOLDSKY

Columnar streaming with native-speed plugins

A streaming runtime built in Rust on Arrow and DataFusion. Write plugins that run on the same zero-copy data plane as the built-ins. Define pipelines in YAML and ship in seconds.

```
$ curl -fsSL https://streamling.dev/install.sh | bash
```

Star on GitHub

pipeline.yaml

```yaml
sources:
  raw.transactions:
    type: kafka
    topic: raw.event.transaction

transforms:
  large_transactions:
    type: sql
    primary_key: id
    sql: |
      SELECT *
      FROM raw.transactions
      WHERE amount > 1000

sinks:
  pg.large_transactions:
    from: large_transactions
    type: postgres
    schema: public
    table: large_transactions
    primary_key: id
```

SOURCES

Kafka
JSON or Avro + Schema Registry. Consumer groups. Tracked offsets.

ClickHouse
Keyset pagination over sorting keys. Adaptive block ranges.

File
Local paths or S3/GCS. CSV, JSON, Parquet, Avro. Bounded or polling.

TRANSFORMS

SQL
Full DataFusion SQL. Projections, filters, UDFs.

TypeScript / WASM
Sandboxed JS/TS via Extism. One `process(input)` function.

HTTP handler
Enrich records via your API. Batched, retried.

Dynamic tables
Live, externally-updatable lookup tables.

SINKS

Postgres
Auto-created tables. `U256/I256` support. Upserts.

Postgres aggregate
Real-time aggregations via DB triggers.

ClickHouse
Auto-created tables. `ReplacingMergeTree` upserts. Gzip compression.

Kafka
Avro + Schema Registry. Operation headers.

Webhook
Push records to any HTTP endpoint as JSON.

Print
Stdout JSON. Perfect for local dev.

S3 `PLUGIN`
Parquet files to S3-compatible storage. Optional Hive partitioning.

MySQL `PLUGIN`
Upsert + delete CDC. Auto-created tables.

SQS `PLUGIN`
Each row as a JSON message. Batched, retried.

Easily add new connectors via plugins

Getting Slack Notifications From Kafka Events

WHY BUILD ANOTHER TOOL?

PREVIOUS: APACHE FLINK

- → Fully-managed Data Streaming platform on Apache Flink
- → 1000+ pipelines in production.
- → Paying the "Big Data" price: there is fixed overhead of launching a pipeline (e.g. having a JobManager).

NEXT: FOCUS ON EFFICIENCY

- → Find a way to perform the same workload, but much more efficient.
- → "Rewrite Big Data in Rust"
- → Started looking into Rust, Arrow and DataFusion.

Some numbers to keep your attention:

- **Over 30X less compute:** In a benchmarked Kafka-to-Clickhouse production workload (read from Kafka, enrich, write into Clickhouse), average CPU usage fell from **10+ cores to 0.3 cores** compared to Flink.
- **Over $1 million in savings per year:** Moving our pipelines to Streamling cut Goldsky's cloud costs by **six figures a month**.
- **Over 20x faster startup:** Cold starts, from a topology change to data written in a destination, dropped from over two minutes on Flink to **under 5 seconds**, including image download, pod creation, and execution.

SQL parser

Different dialects supported

Logical planner

With Optimizer

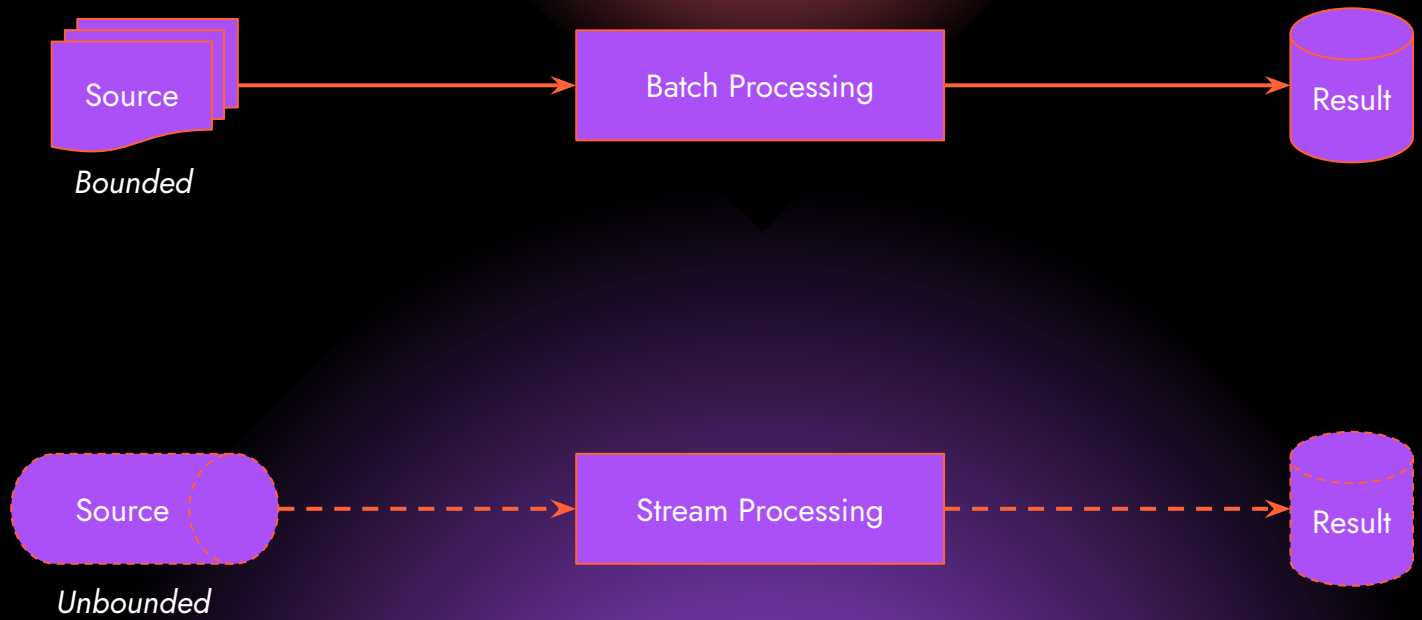
Physical planner

With Optimizer

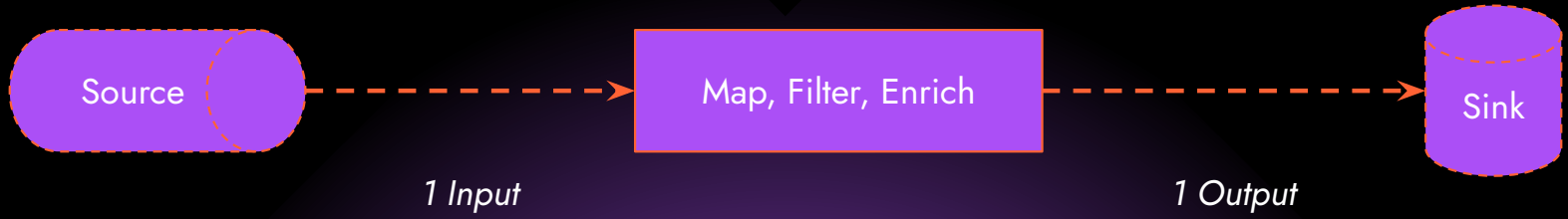
Vectorized executor

Arrow's RecordBatches with Tokio Streams

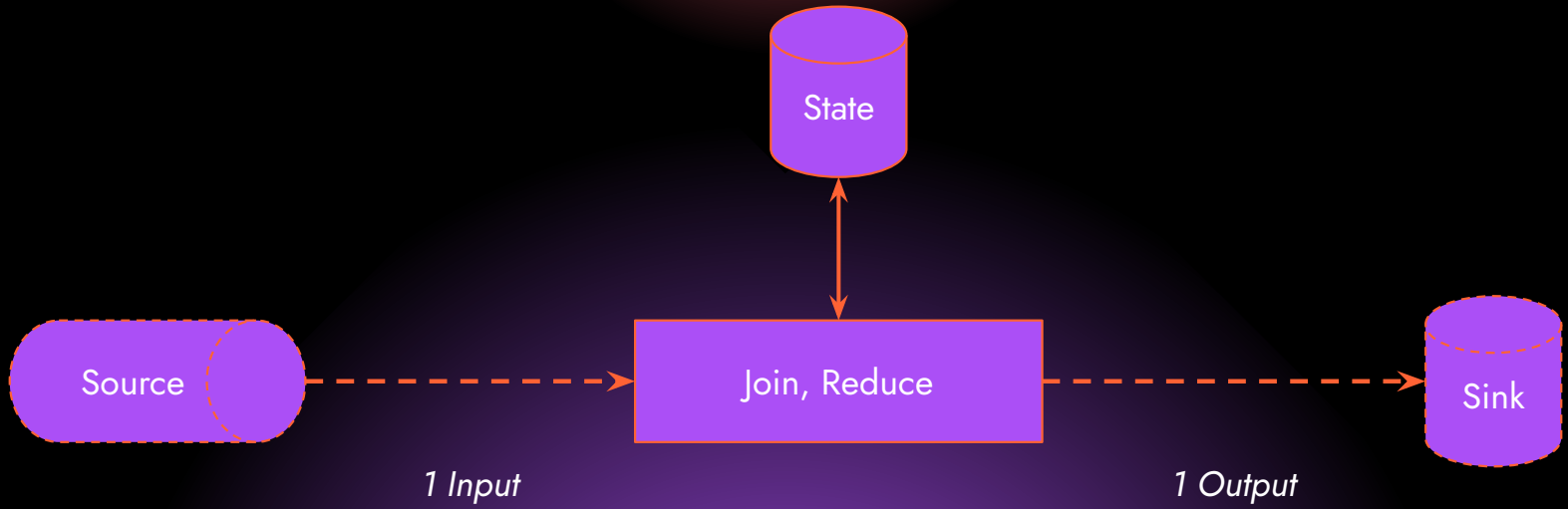
How do you turn a query engine into a streaming engine?



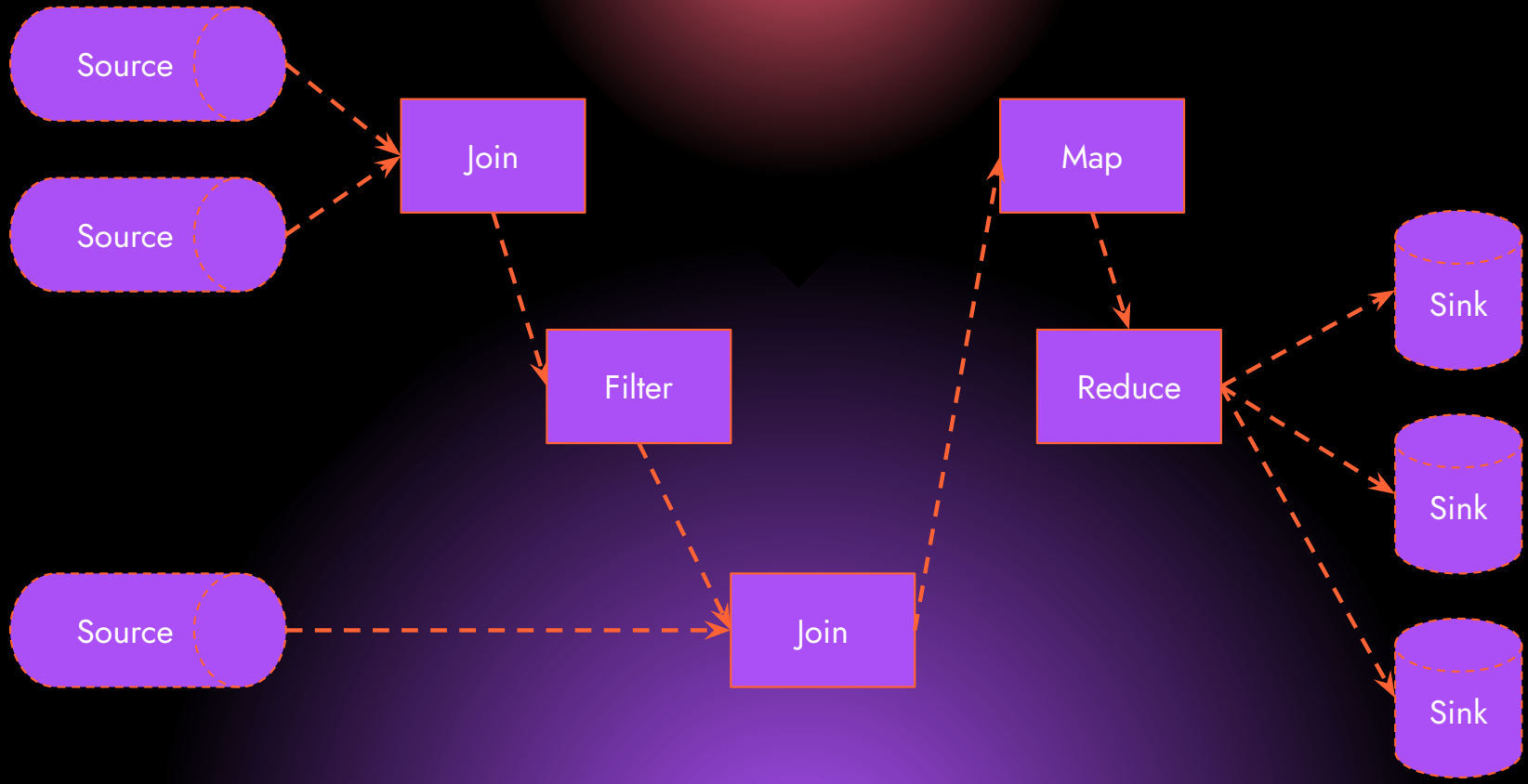
Batch vs Stream Processing



Stateless Stream Processing

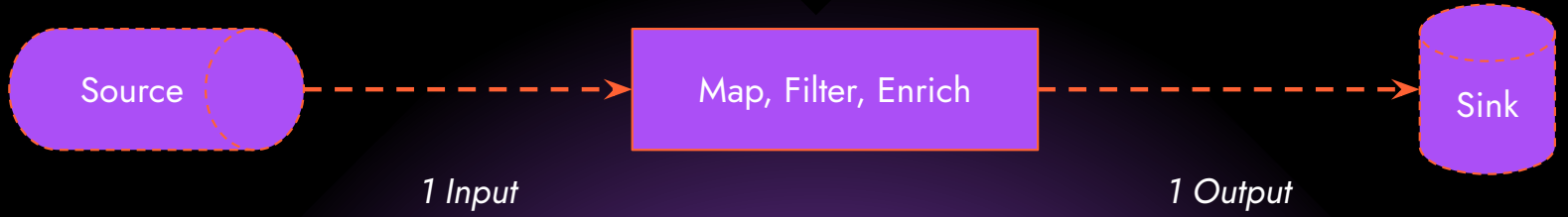


Stateful Stream Processing



Complex Topology

```rust
pub trait RecordBatchStream: Stream<Item = Result<RecordBatch>> {
    fn schema(&self) -> SchemaRef;
}

pub type SendableRecordBatchStream = Pin<Box<dyn RecordBatchStream + Send>>;
```


Source
Map, Filter, Enrich
Sink
1 Input
1 Output

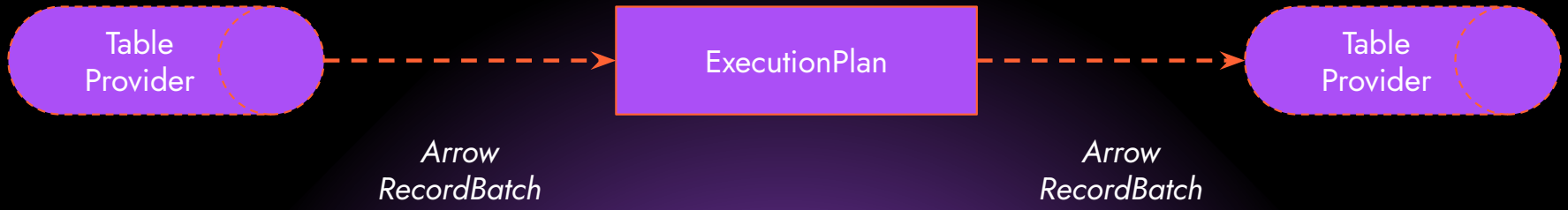
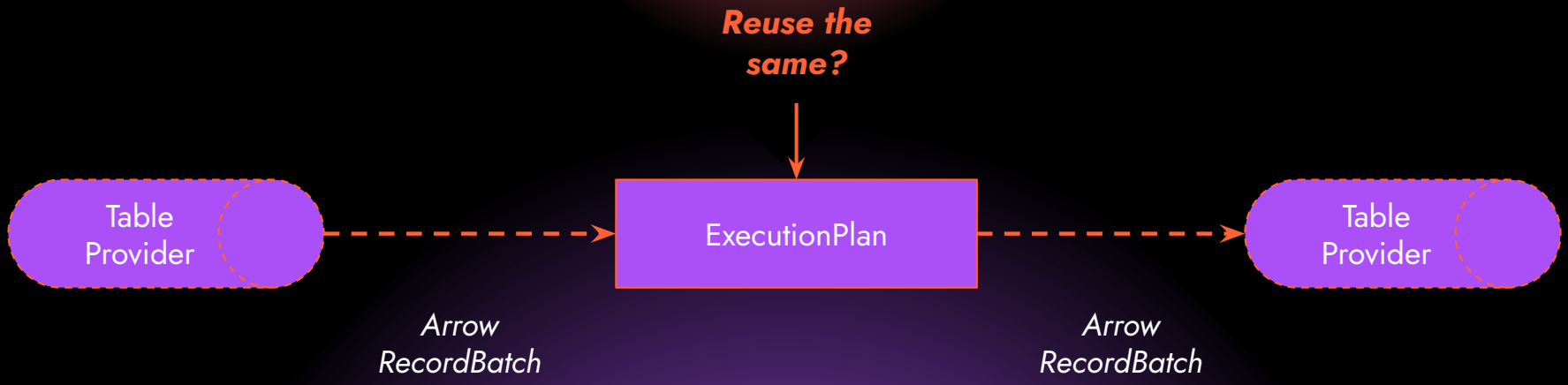


Table Provider
ExecutionPlan
Table Provider
Arrow RecordBatch
Arrow RecordBatch


Reuse the same?
Table Provider
ExecutionPlan
Table Provider
Arrow RecordBatch
Arrow RecordBatch

Using batches of columnar data for a streaming engine? Why?

ARROW AS A STREAMING FORMAT

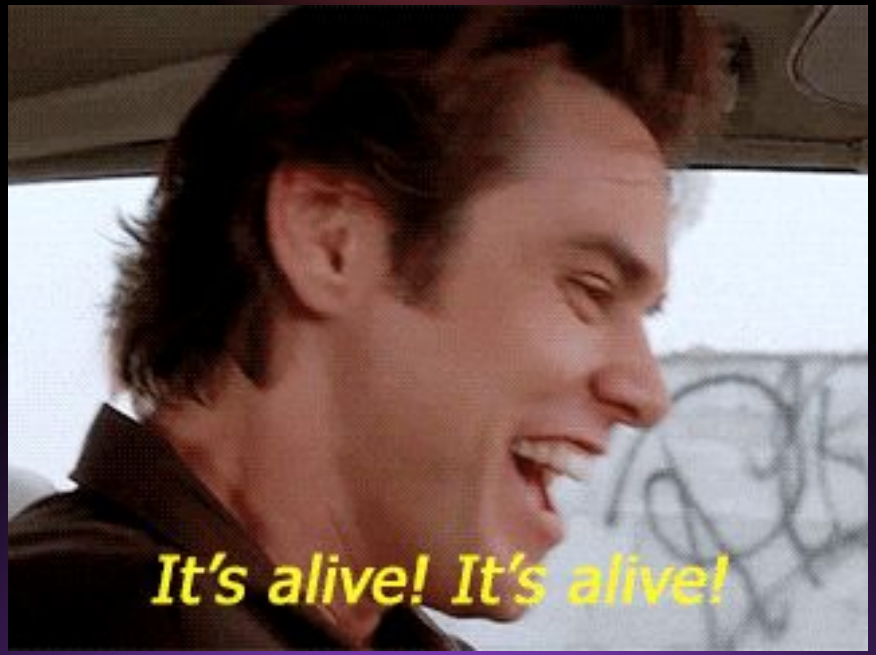
Columnar data is a great fit for a streaming system!

- Ultimate performance thanks to vectorization and good CPU cache locality
- Can control batch size to manage memory usage / throughput / latency
- Kafka has batching. Flink has batching. It's mostly hidden from the user

> For example, if we want at least 100 records in our batch to overcome fixed costs, the amount of time we need to wait to receive 100 records will depend on our throughput:
>
> - At 10 events/second, it takes 1 second
> - At 1,000 — 0.01 seconds (100ms)
> - At 1,000,000 — 0.0001 (0.1ms)

Micah Wylde

```rust
// (1) A source is a TableProvider.
impl TableProvider for KafkaSourceTableProvider {
  async fn scan(&self, /*..*/) -> Result<Arc<dyn ExecutionPlan>> {
    Ok(Arc::new(KafkaSourceExec::new(/*.. */)))
  }
}

// (2) The ExecutionPlan: stream, never ends, incremental.
impl ExecutionPlan for KafkaSourceExec {
  fn properties(&self) -> &PlanProperties {
    Boundedness::Unbounded { requires_infinite_memory:  false }
    EmissionType::Incremental
  }
  fn execute(&self, /*..*/) -> Result<SendableRecordBatchStream> {
    // returns Stream<RecordBatch> that yields forever.
  }
}
```


It's alive! It's alive!

Distribution via Kafka Consumer Groups

NON-INCREMENTAL OPERATORS

The following operators can have EmissionType::Final:

- HashJoinExec
- NestedLoopJoinExec
- AggregateExec
- SortExec
- ...

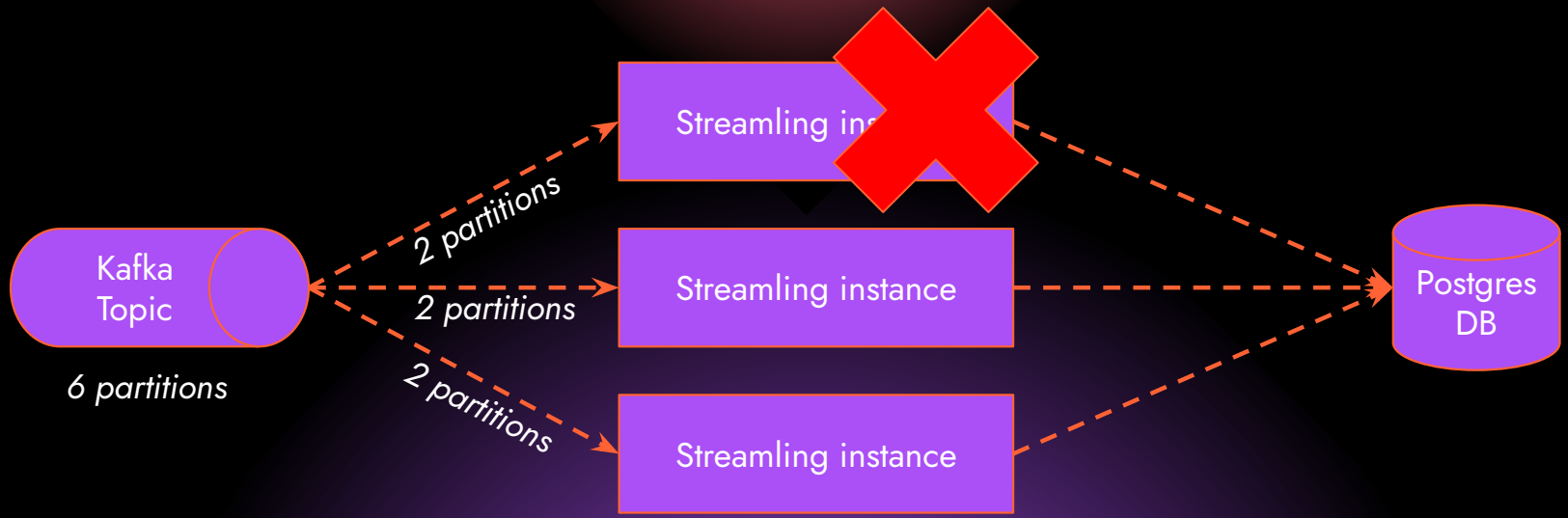
```
pub enum EmissionType {
    /// Records are emitted incrementally as they arrive and are processed
    Incremental,
    /// Records are only emitted once all input has been processed
    Final,
    /// Records can be emitted both incrementally and as a final result
    Both,
}
```

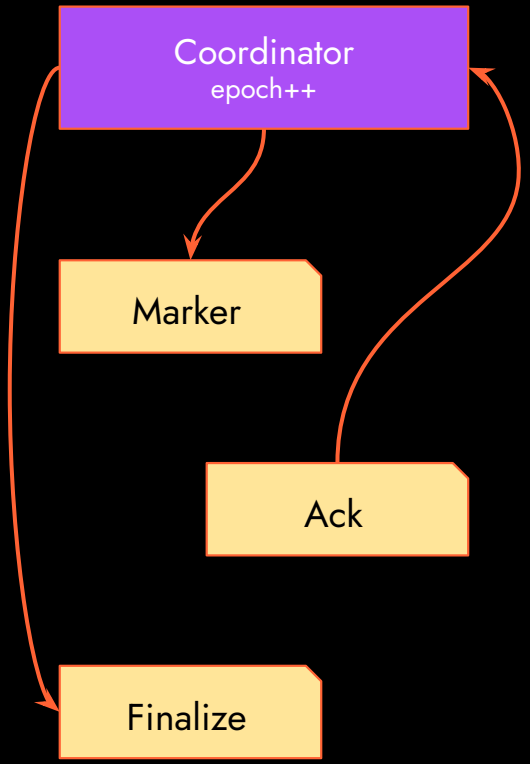
STATEFUL DISTRIBUTED OPERATORS

- → Need multi-node support for shuffles
  - DataFusion Ballista, DataFusion Ray, datafusion-distributed
- → Need multi-node aware operators
  - Some built-in operators can be reused
- → Need state backend support
  - To store intermediate values

STATEFUL DISTRIBUTED OPERATORS

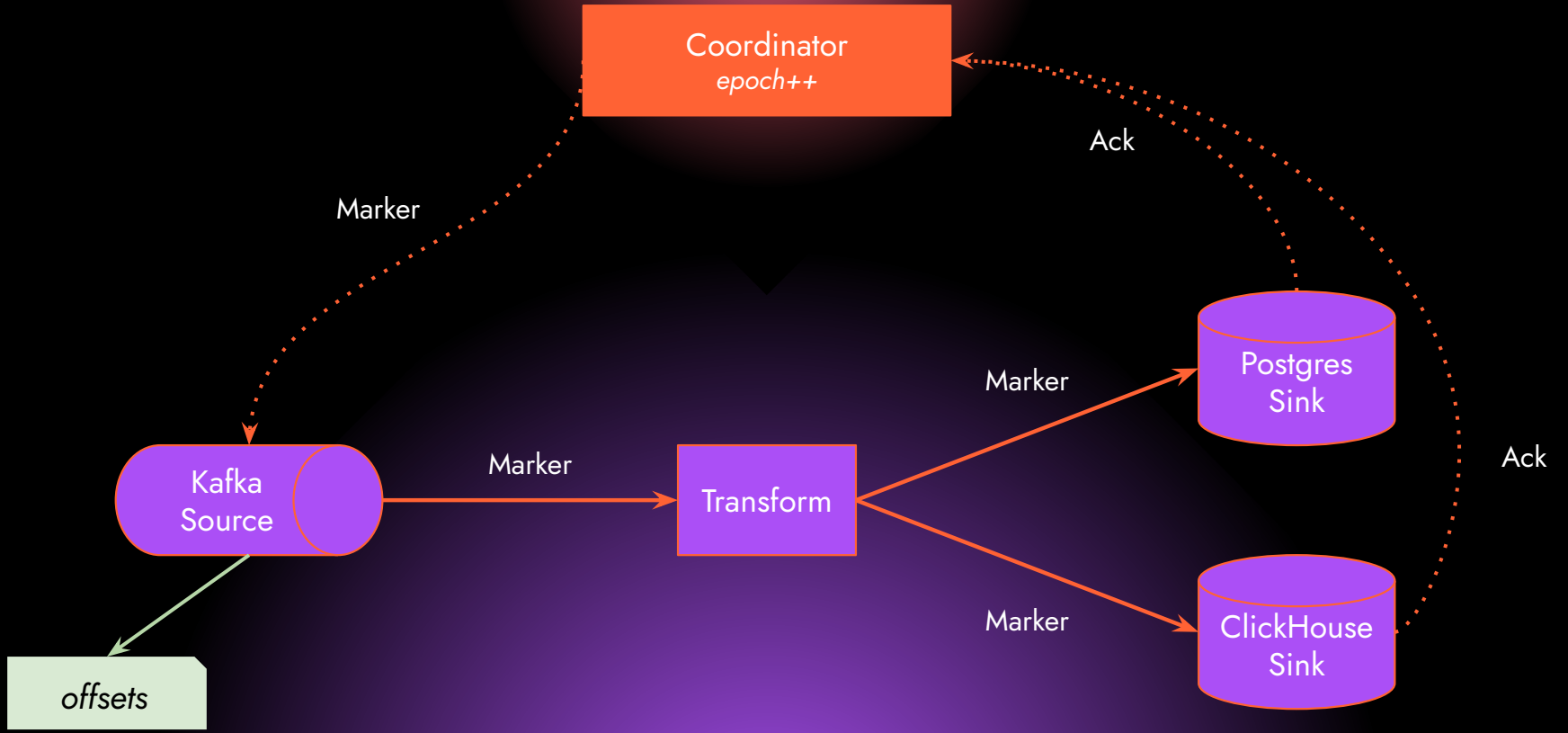
- → Need multi-node support for shuffles
  - DataFusion Ballista, DataFusion Ray, datafusion-distributed
- → Need multi-node aware operators
  - Some built-in operators can be reused
- → Need state backend support
  - To store intermediate values


Kafka
Topic
6 partitions
2 partitions
2 partitions
2 partitions
Streamling ins
Streamling instance
Streamling instance
Postgres
DB

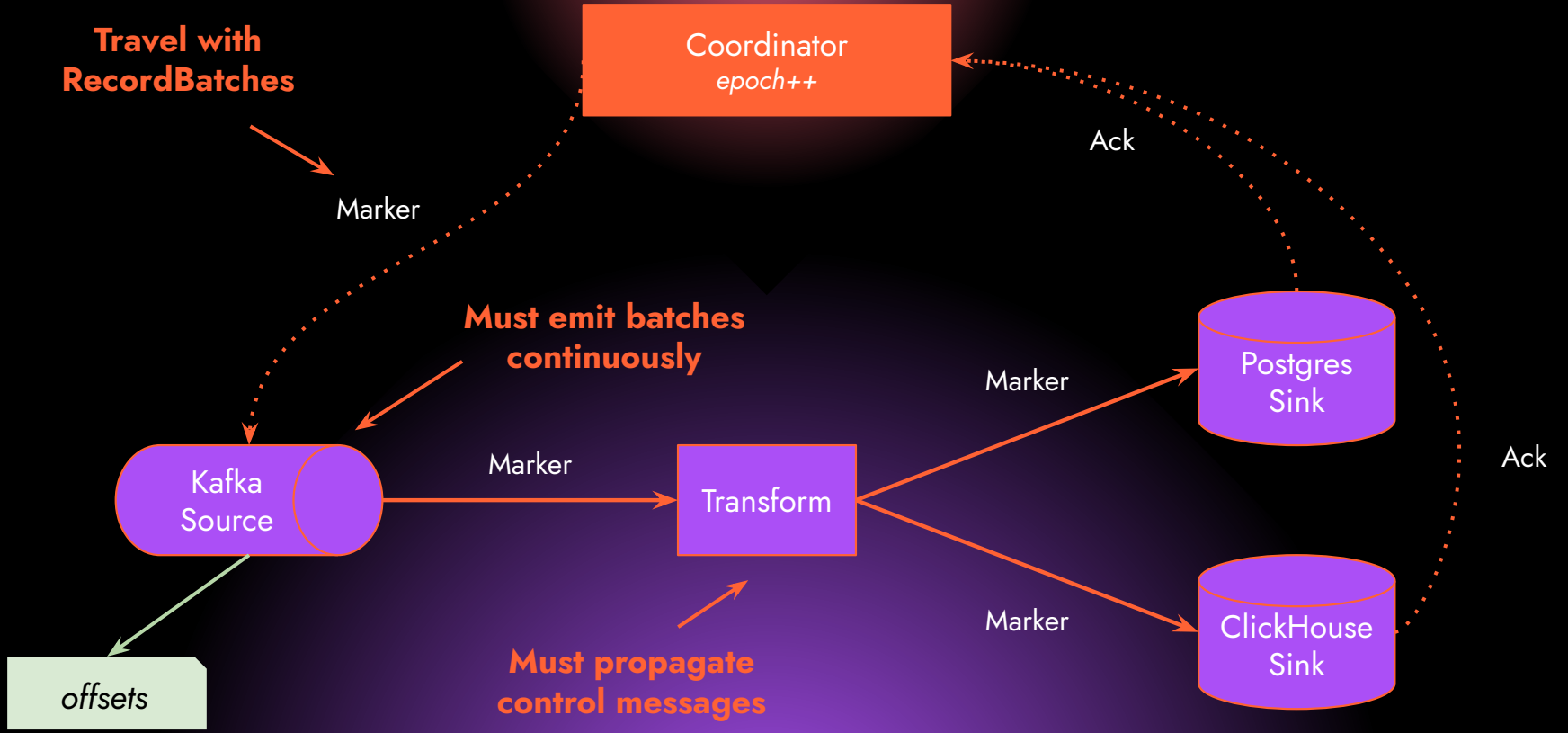


Chandy–Lamport Style Checkpointing

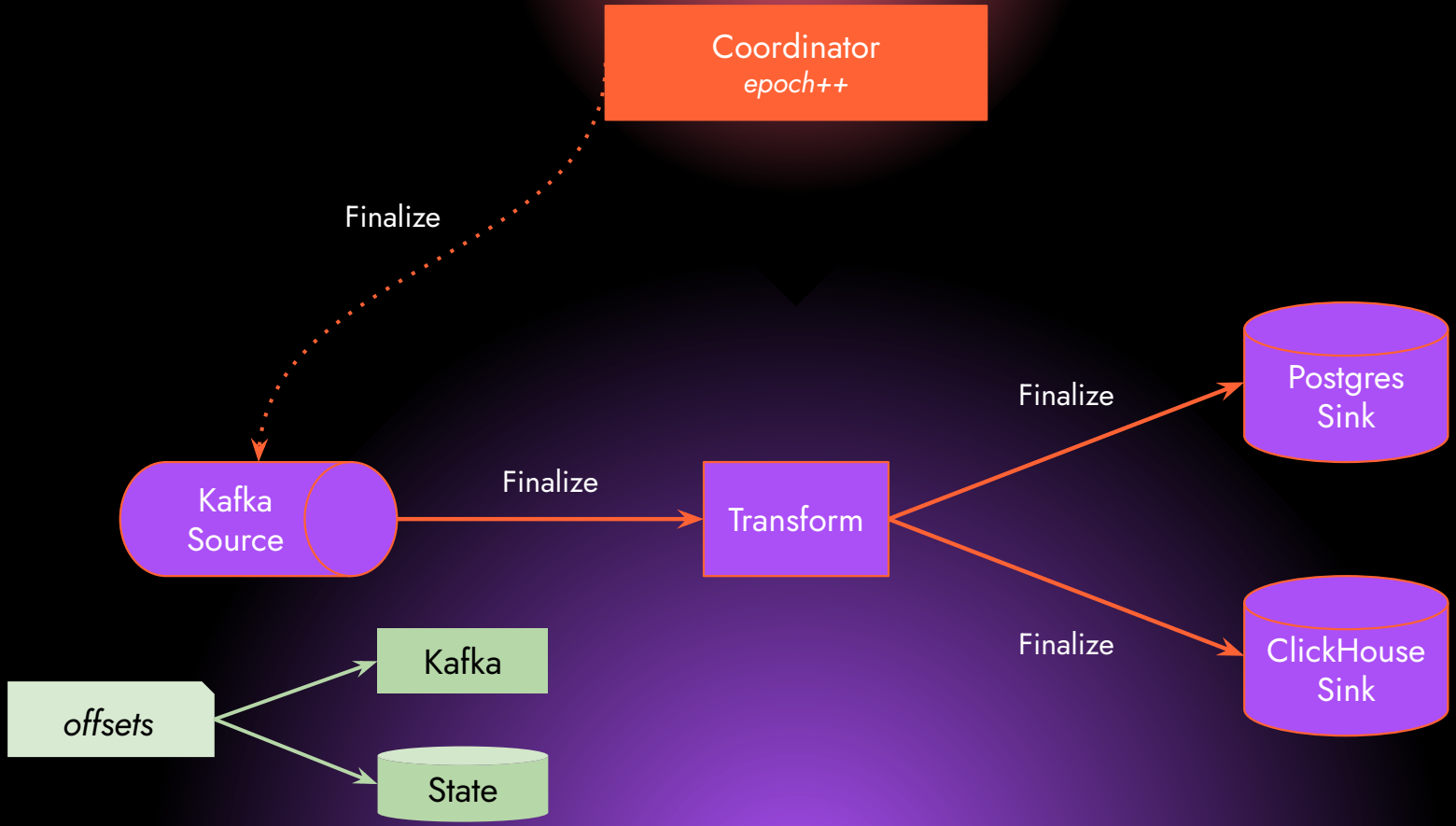
- → Each node runs the algorithm independently
- → Messages travel with RecordBatches
- → The algorithm needs RecordBatches to flow continuously, even if empty
- → Every source, transformation and sink must support checkpointing
- → Can be combined with State Backend and external system for at-least-once delivery



Chandy–Lamport Style Checkpointing Process: Phase 1



Chandy–Lamport Style Checkpointing Process: Phase 1



Chandy–Lamport Style Checkpointing Process: Phase 2

```rust
pub fn enrich_batch_with_metadata(
    batch: RecordBatch,
    metadata: HashMap<String, String>,
) -> Result<RecordBatch> {
    let schema = batch.schema();
    let enriched_schema = Arc::new(Schema::new_with_metadata(schema.fields().clone(), metadata));
    RecordBatch::try_new(enriched_schema, batch.columns().to_vec())
        .streamling_context("failed to create RecordBatch with enriched metadata")
}
```

Enriching RecordBatches with control messages

STATE BACKEND

Operators and connectors can use state backend for storing metadata.

- Built-in support for Sqlite and Postgres
- Used by the Kafka source to store offsets
- Used by various plugins to store metadata

```rust
#[async_trait]
pub trait StateOperatorBackend<V>{
    async fn get(&self, key: StateKey)
-> Result<Option<V>, Error>;
    async fn put(&self, key: StateKey,
value: V) -> Result<(), Error>;
    async fn remove(&self, key:
StateKey) -> Result<(), Error>;
    async fn clear(&self) ->
Result<(), Error>;
}
```

Simple stateless workloads should work now... Right?

DATAFUSION GOTCHAS

Stable Functions

ScalarUDFs can be **Immutable**, **Stable** or **Volatile.**

→ *"A stable function may return different values given the same input across different queries but must return the same value for a given input within a query."*

→ now(), current_time() and current_date()

→ Had to reimplement them as Volatile and override built-in functions.

Batches Must Flow

Surely projections and filters should work with streaming sources… Right?

→ Some built-in operators don't emit empty batches, it doesn't make sense for a query engine.

→ But batches need to carry checkpoint messages!

→ Therefore, we need to modify their behavior.

→ Using PhysicalOptimizerRules makes it really easy.

Scan Sharing

Query engines LOVE projection and predicate pushdown.

→ Many optimizations don't make sense in case of a streaming source like Kafka.

→ Implemented a way to perform scan sharing (reuse the same Kafka source) in case of multiple consumers.

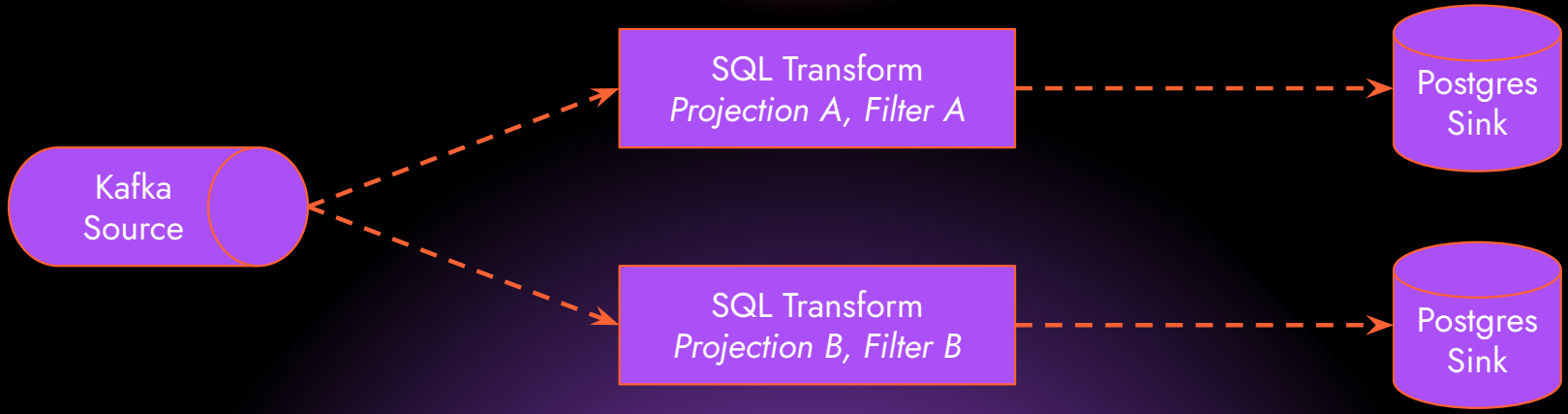
→ The source is consumed once and the data is broadcasted to multiple downstream nodes.

```rust
impl PhysicalOptimizerRule for StreamingFilterRewritePhysicalOptimizerRule {
    fn optimize(
        &self,
        plan: Arc<dyn ExecutionPlan>,
        config: &ConfigOptions,
    ) -> Result<Arc<dyn ExecutionPlan>> {
        plan.transform_down(|input_plan| {
            if let Some(original_filter) = input_plan.as_any().downcast_ref::<FilterExec>() {
                let streaming_filter =
                    StreamingFilterExec::from_original(original_filter.clone()).unwrap();
                Ok(Transformed::yes(Arc::new(streaming_filter)))
            } else {
                Ok(Transformed::no(input_plan))
            }
        })
        .data()
    }
    fn name(&self) -> &str { "StreamingFilterRewrite" }
}
```

StreamingFilterRewritePhysicalOptimizerRule

```
SELECT id, value, now() as ingested_at
```

	id	value	ingested_at
Row 1	1	100.00	2026-01-01 10:10:12
Row 2	2	200.00	2026-01-01 10:10:12
Row 3	3	300.00	2026-01-01 10:10:12



Logical Plan: Single Kafka Source

Actual Execution: Kafka Source per Branch

ARROW AS A STREAMING FORMAT

- → Zero-copy abstractions!
  - Modifying a single column in a large dataset? No other columns affected
- → Language-independent
  - Same memory layout regarding of the language
- → Supported by more and more databases and data processing tools
  - **The goal is to pass Arrow end-to-end**

CLICKHOUSE HAS NATIVE ARROW SUPPORT

SCENARIO	SOURCE	THROUGHPUT	VS. KAFKA
Kafka baseline	WarpStream (Avro)	~50K rows/s	1x
Bloom-filter scan	ClickHouse (Arrow)	~350K rows/s	~7x
Partition-based scan	**ClickHouse (Arrow)**	**~600K rows/s**	**~12x**

PLUGIN SYSTEM

- Support for **source**, **transform**, **sink**, UDF and topology preprocessor plugins
- Based on FFI and abi_stable
- Friendly, high-level interface
- Zero-copy RecordBatch passing
- Dynamically loaded binaries, configured with env vars
- Runtime handles validation, backpressure and checkpointing

```rust
#[async_trait]
pub trait SinkPlugin: SupportsGracefulShutdown + Send
+ Sync {
    async fn initialize(&self) -> Result<(),
PluginError>;
    fn labels(&self) -> Vec<PluginLabel>
    async fn process_batch(&self, data: RecordBatch)
-> Result<(), PluginError>;
    async fn process_checkpoint_marker(&self, epoch:
CheckpointEpoch) -> Result<(), PluginError>;
    async fn process_checkpoint_finalizer(&self,
epoch: CheckpointEpoch)
    -> Result<(), PluginError>;
}
```

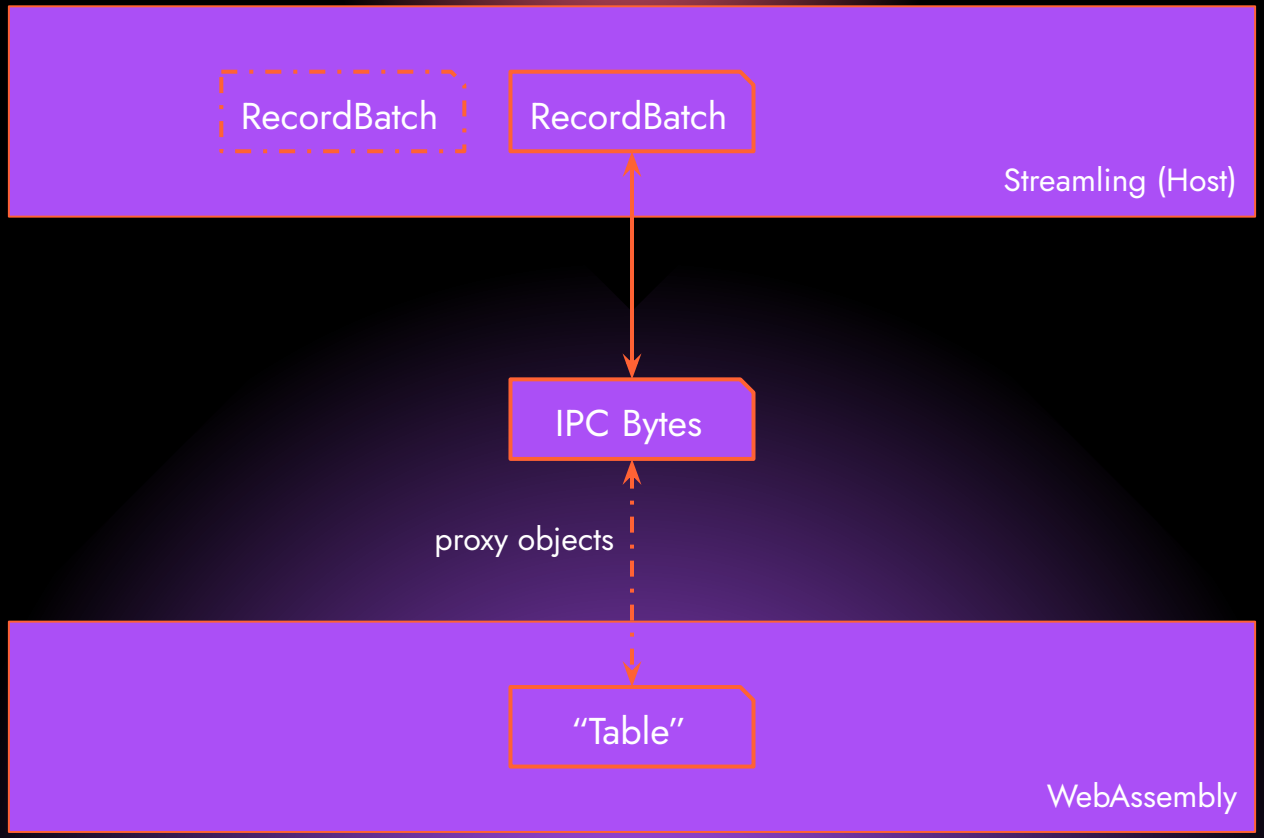
Many developers don’t want to write SQL!

```
transforms:
  reshape_data:
    type: script
    from: transfers
    language: typescript
    primary_key: transfer_id
    schema:
      transfer_id: string
      sender: string
      receiver: string
      value_eth: float64
      timestamp: string
    script: |
      function invoke(data) {
        return {
          transfer_id: data.id,
          sender: data.from_address,
          receiver: data.to_address,
          value_eth: Number(data.value) / 1e18,
          timestamp: new Date(data.block_timestamp * 1000).toISOString()
        };
      }
```

SCRIPTING SUPPORT VIA WEBASSEMBLY

- → **Extism**: Wasm framework, mostly used for inputs/outputs
- → **Flechette**: JS library for reading and writing Arrow data
- → TypeScript is transpiled to JavaScript on the host side (Rust) via SWC
- → User code is executed with `eval` 😬

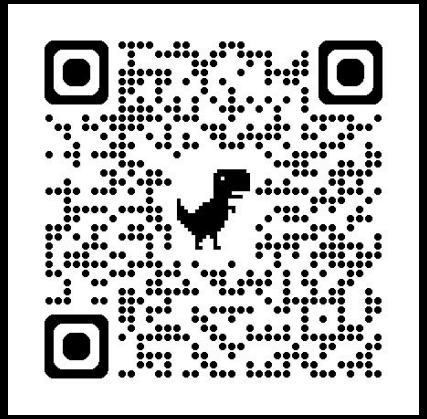
```
const fn = eval("(" + code + ")");

for (let i = 0; i < numRows; i++) {
  const result = fn(inputTable.get(i));
  // ...
}
```


RecordBatch
RecordBatch
Streamling (Host)
IPC Bytes
proxy objects
"Table"
WebAssembly

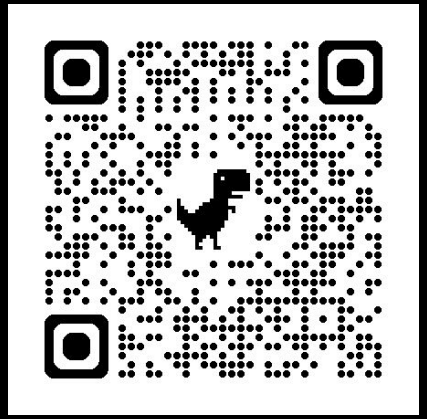
SUMMARY

- → Most of the data-intensive applications should default to Arrow (or other columnar formats), unless it's OLTP
- → Using DataFusion is a cheat code for building a new data-intensive application: get there in 3 months instead of 3 years
- → Streaming semantics offer unique challenges when relying on a query engine as a foundation. But they can be solved!

THANK YOU!



https://sap1ens.com



https://streamling.dev