2026

Yaroslav Tkachenko

Streamling: Building A Lightweight Data Streaming ~~Framework~~ Runtime On Apache Datafusion

Yaroslav Tkachenko

About me

- → Software Engineer, background in startups
- → Technical Lead at **Activision**, **Shopify**
- → Founding Engineer at **Goldsky**
- → Founder at **Irontools**

- → Data streaming **advising**, **consulting** and **training**

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
```

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
```

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.

ASF Links

- Apache Software Foundation
- License
- Donate
- Thanks
- Security

Links

- GitHub and Issue Tracker
- crates.io
- API Docs
- Blog
- Code of conduct
- Download

User Guide

- Introduction
- Example Usage
- Features
- Concepts, Readings, Events
- Crate Configuration
- DataFusion CLI
- DataFrame API
- Gentle Arrow Introduction
- Expression API
- SQL Reference
- Configuration Settings
- Reading Explain Plans

Apache DataFusion

Star Fork

DataFusion is an extensible query engine written in Rust that uses Apache Arrow as its in-memory format.

The documentation on this site is for the core DataFusion project, which contains libraries and binaries for developers building fast and feature rich database and analytic systems, customized to particular workloads. See use cases for examples.

The following related subprojects target end users and have separate documentation.

- DataFusion Python offers a Python interface for SQL and DataFrame queries.
- DataFusion Java offers a Java interface for SQL and DataFrame queries.
- DataFusion Comet is an accelerator for Apache Spark based on DataFusion.
- DataFusion Ballista is distributed processing extension for DataFusion.

"Out of the box," DataFusion offers SQL and Dataframe APIs, excellent performance, built-in support for CSV, Parquet, JSON, and Avro, extensive customization, and a great community. Python Bindings are also available. Ballista is Apache DataFusion extension enabling the parallelized execution of workloads across multiple nodes in a distributed environment.

DataFusion features a full query planner, a columnar, streaming, multi-threaded, vectorized execution engine, and partitioned data sources. You can customize DataFusion at almost all points including additional data sources, query languages, functions, custom operators and more. See the Architecture section for more details.

```rust
use datafusion::prelude::*;

#[tokio::main]
async fn main() -> datafusion::error::Result<()> {
  // register the table
  let ctx = SessionContext::new();
  ctx.register_csv("example", "tests/data/example.csv",
CsvReadOptions::new()).await?;

  // create a plan to run a SQL query
  let df = ctx.sql("SELECT a, MIN(b) FROM example WHERE
a <= b GROUP BY a LIMIT 100").await?;

  // execute and print results
  df.show().await?;
  Ok(())
}
```

SQL parser

Different dialects supported

Logical planner

With Optimizer

Physical planner

With Optimizer

Vectorized executor

Arrow's RecordBatches with Tokio Streams

```rust
pub enum LogicalPlan {
    Projection(Projection),
    Filter(Filter),
    Window(Window),
    Aggregate(Aggregate),
    Sort(Sort),
    Join(Join),
    Repartition(Repartition),
    Union(Union),
    TableScan(TableScan),
    EmptyRelation(EmptyRelation),
    Subquery(Subquery),
    SubqueryAlias(SubqueryAlias),
    Limit(Limit),
    Statement(Statement),
    Values(Values),
    Explain(Explain),
    Analyze(Analyze),
    Extension(Extension),
    Distinct(Distinct),
    Dml(DmlStatement),
    Ddl(DdlStatement),
    Copy(CopyTo),
    DescribeTable(DescribeTable),
    Unnest(Unnest),
    RecursiveQuery(RecursiveQuery),
}
```

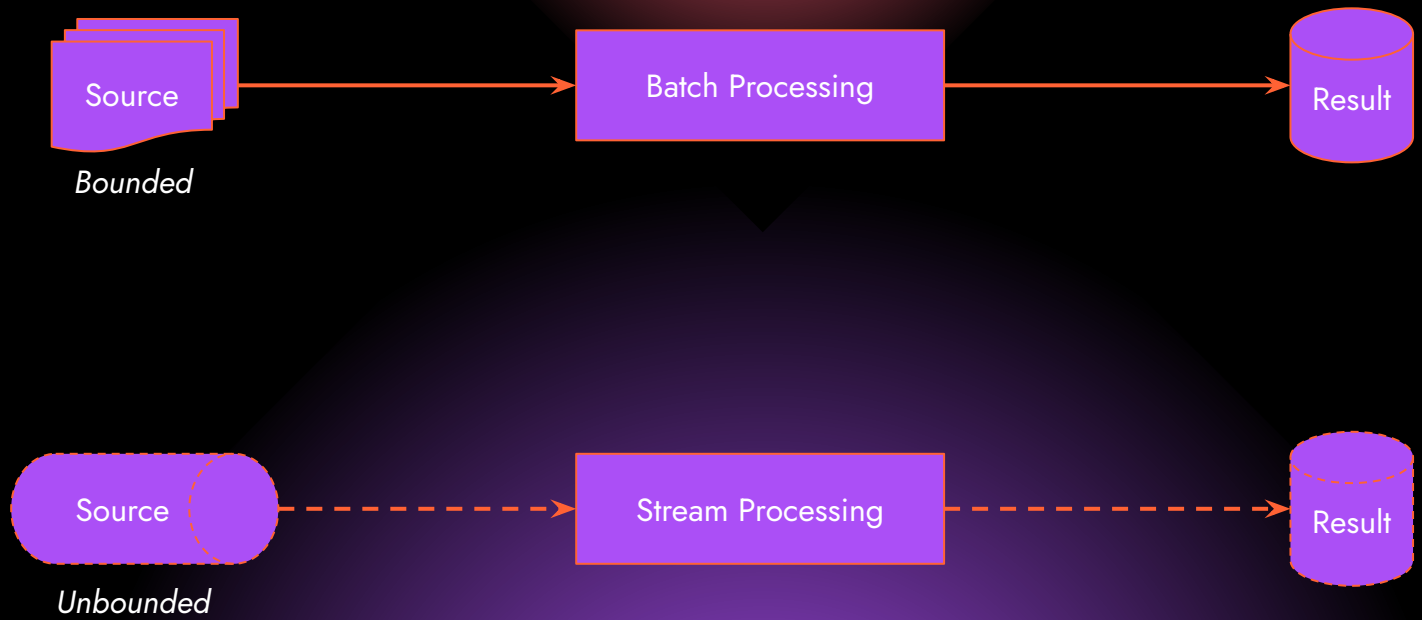
```rust
pub trait ExecutionPlan {
    // ...

    fn execute(
        &self,
        partition: usize,
        context: Arc<TaskContext>,
    ) -> Result<SendableRecordBatchStream>;
}
```

```sql
SELECT "WatchID" AS wid, "hits.parquet"."ClientIP" AS ip
FROM 'hits.parquet'
WHERE starts_with("URL", 'http://example.com/')
ORDER BY wid ASC, ip DESC
LIMIT 5;
```

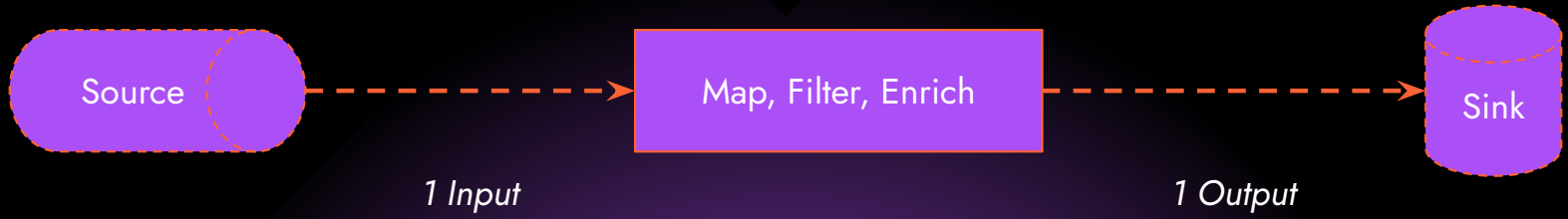
```
Sort: wid ASC NULLS LAST, ip DESC NULLS FIRST, fetch=5
  Projection: hits.parquet.WatchID AS wid, hits.parquet.ClientIP AS ip
    Filter: starts_with(hits.parquet.URL, Utf8("http://example.com/"))
      TableScan: hits.parquet projection=[WatchID, ClientIP, URL], partial_filters=...
```

```
SortPreservingMergeExec: [wid@0 ASC NULLS LAST,ip@1 DESC], fetch=5
  SortExec: TopK(fetch=5), expr=[wid@0 ASC NULLS LAST,ip@1 DESC], preserve_partitioning=[true]
    ProjectionExec: expr=[WatchID@0 as wid, ClientIP@1 as ip]
      CoalesceBatchesExec: target_batch_size=8192
        FilterExec: starts_with(URL@2, http://example.com/)
          DataSourceExec: file_groups={16 groups: [[hits.parquet:0..923748528], ...]}, ...
```

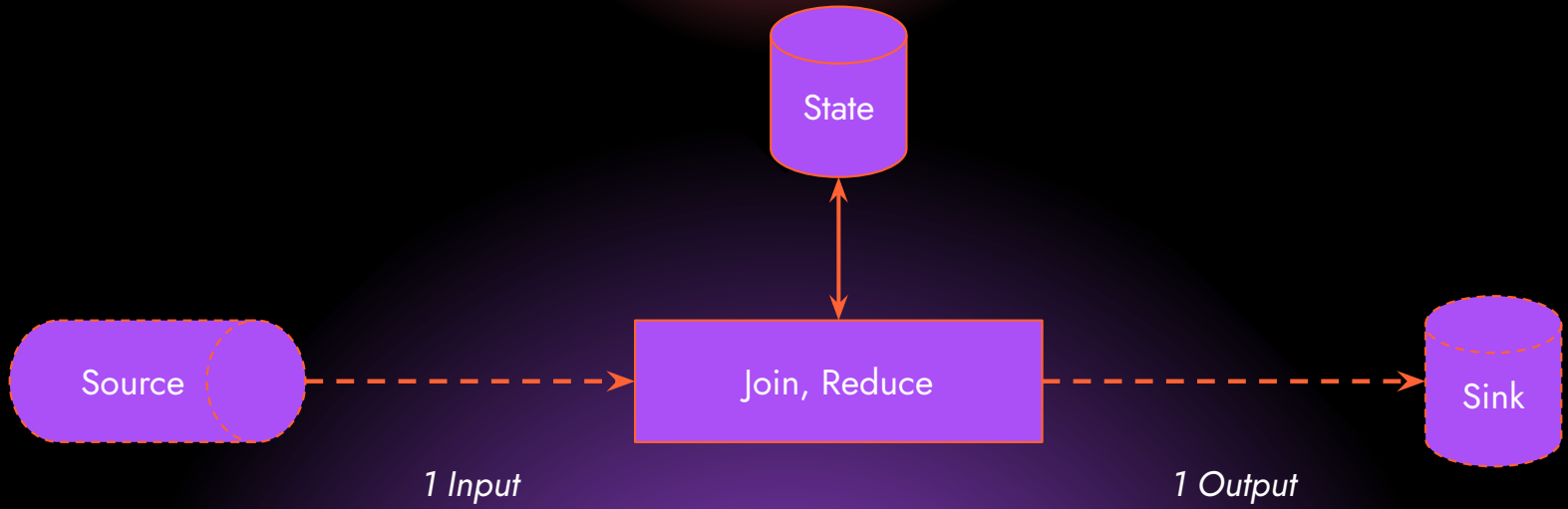
How do you turn a query engine into a streaming engine?



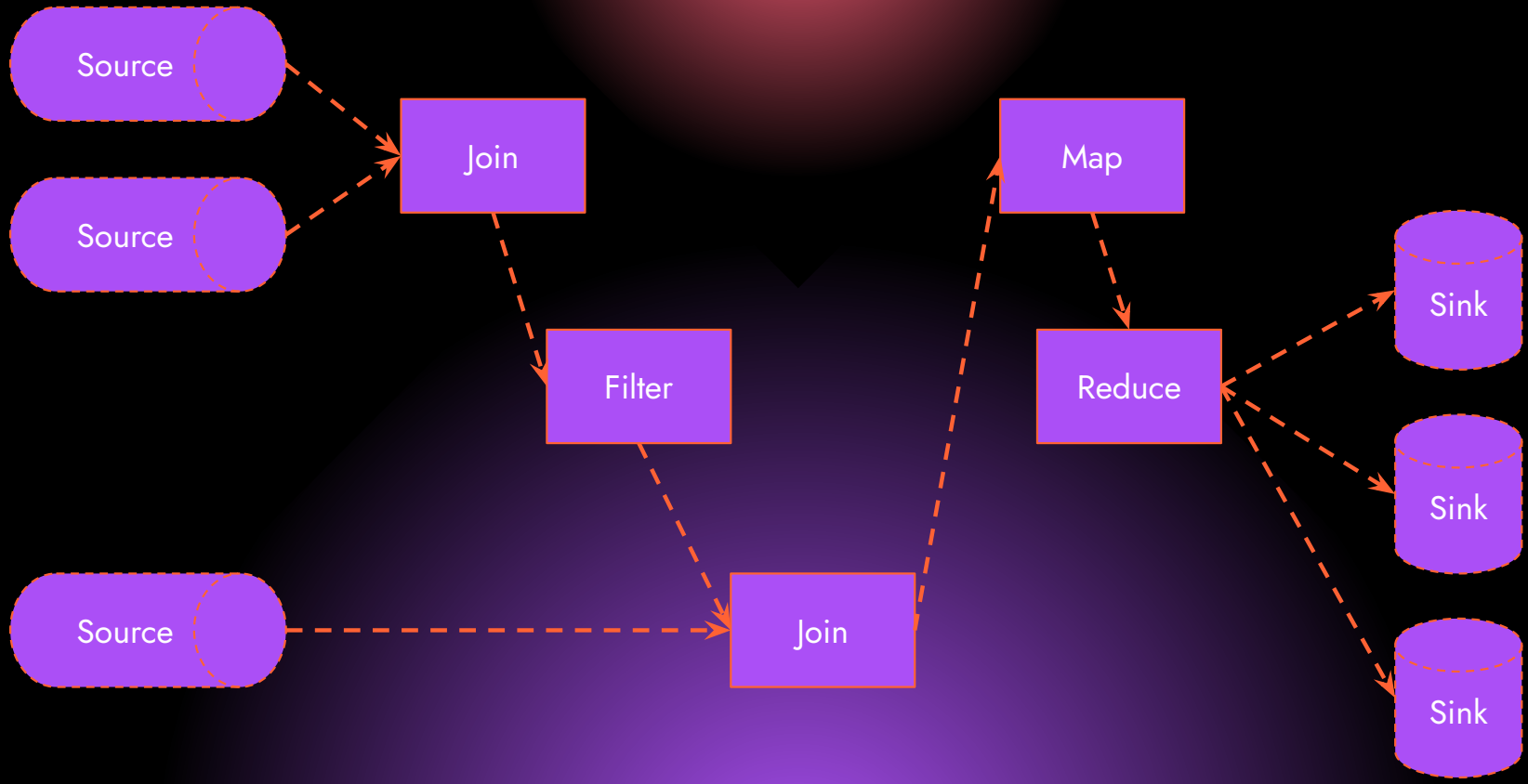
Batch vs Stream Processing



Stateless Stream Processing

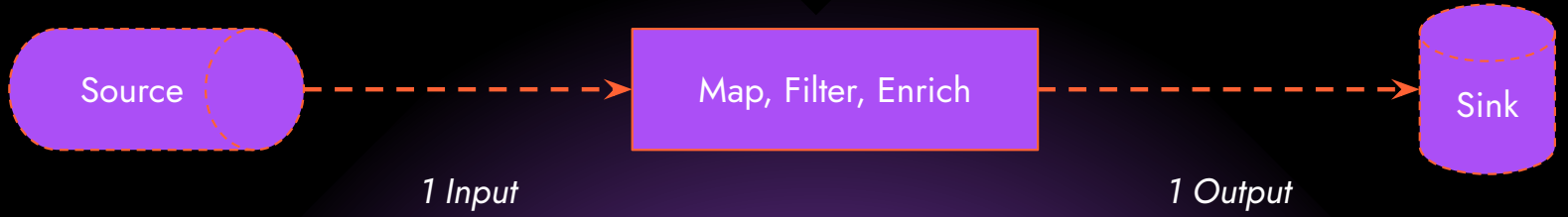


Stateful Stream Processing



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

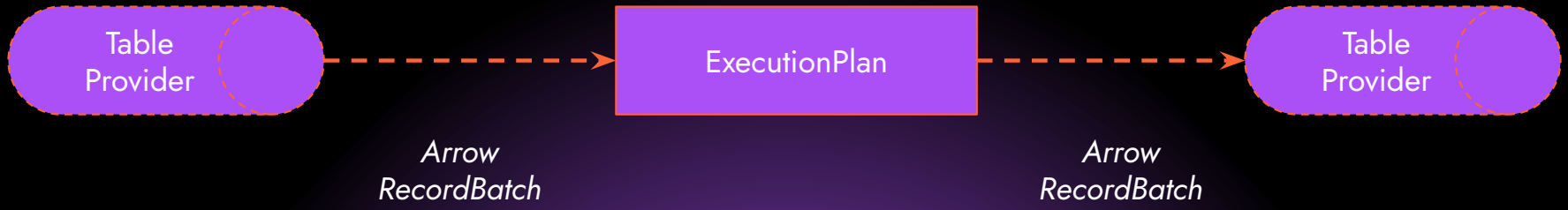
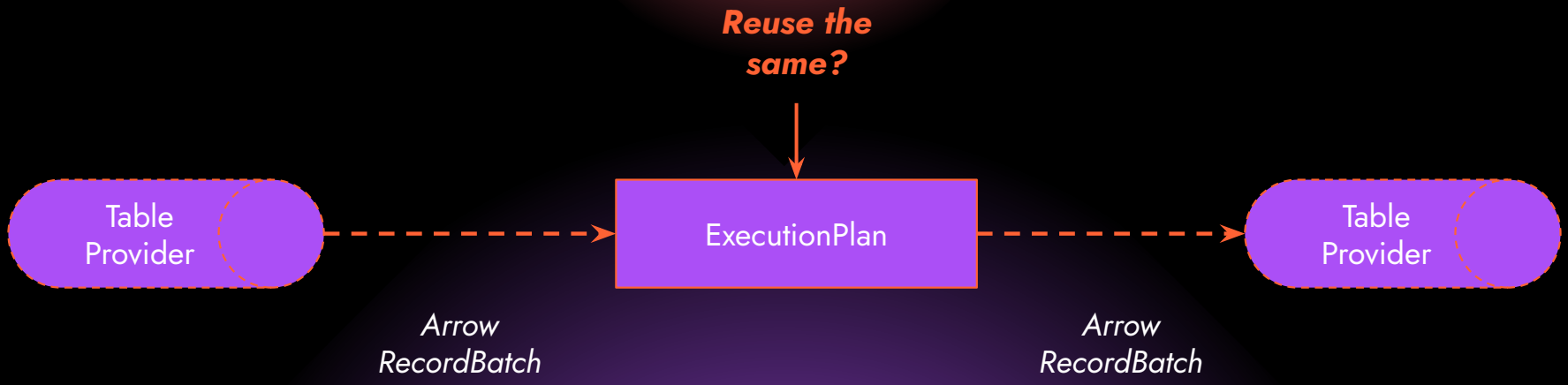


Table Provider
ExecutionPlan
Table Provider
Arrow RecordBatch
Arrow RecordBatch


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

```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.
  }
}
```

Distribution via Kafka Consumer Groups

NON-INCREMENTAL OPERATORS

The following operators can have EmissionType::Final:

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

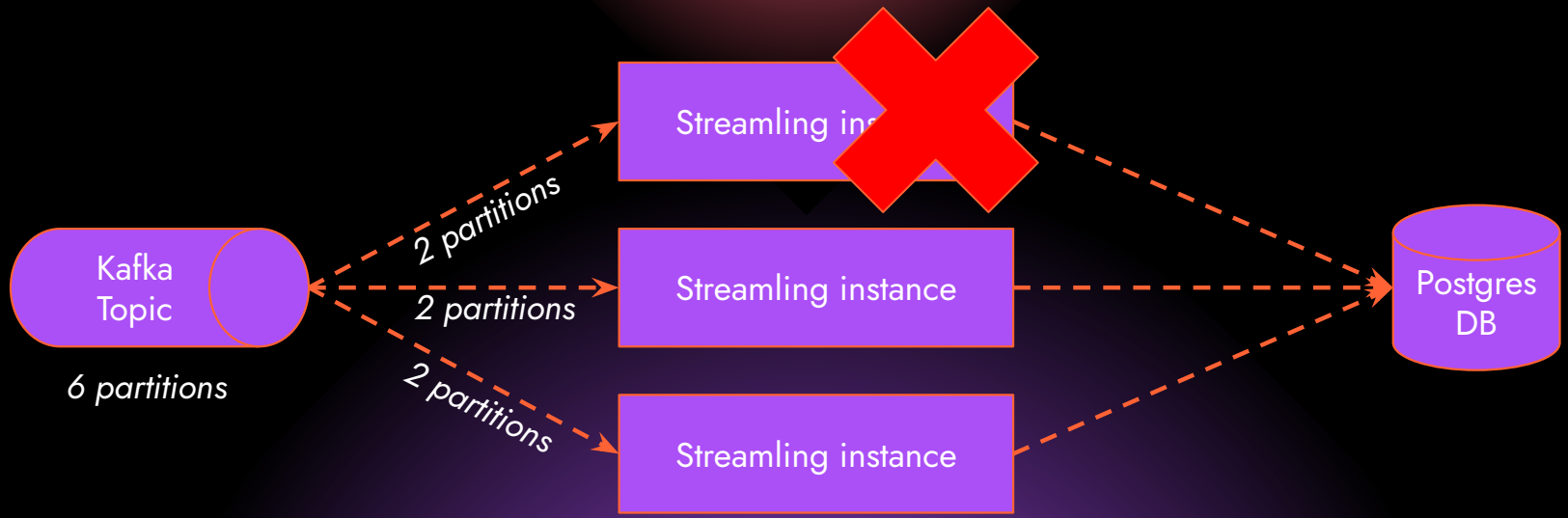
```
pub enum EmissionType {
    /// Records are emitted incrementally as they arrive and are processed
    Incremental,
    /// Records are only emitted once all input has been processed
    Final,
    /// Records can be emitted both incrementally and as a final result
    Both,
}
```

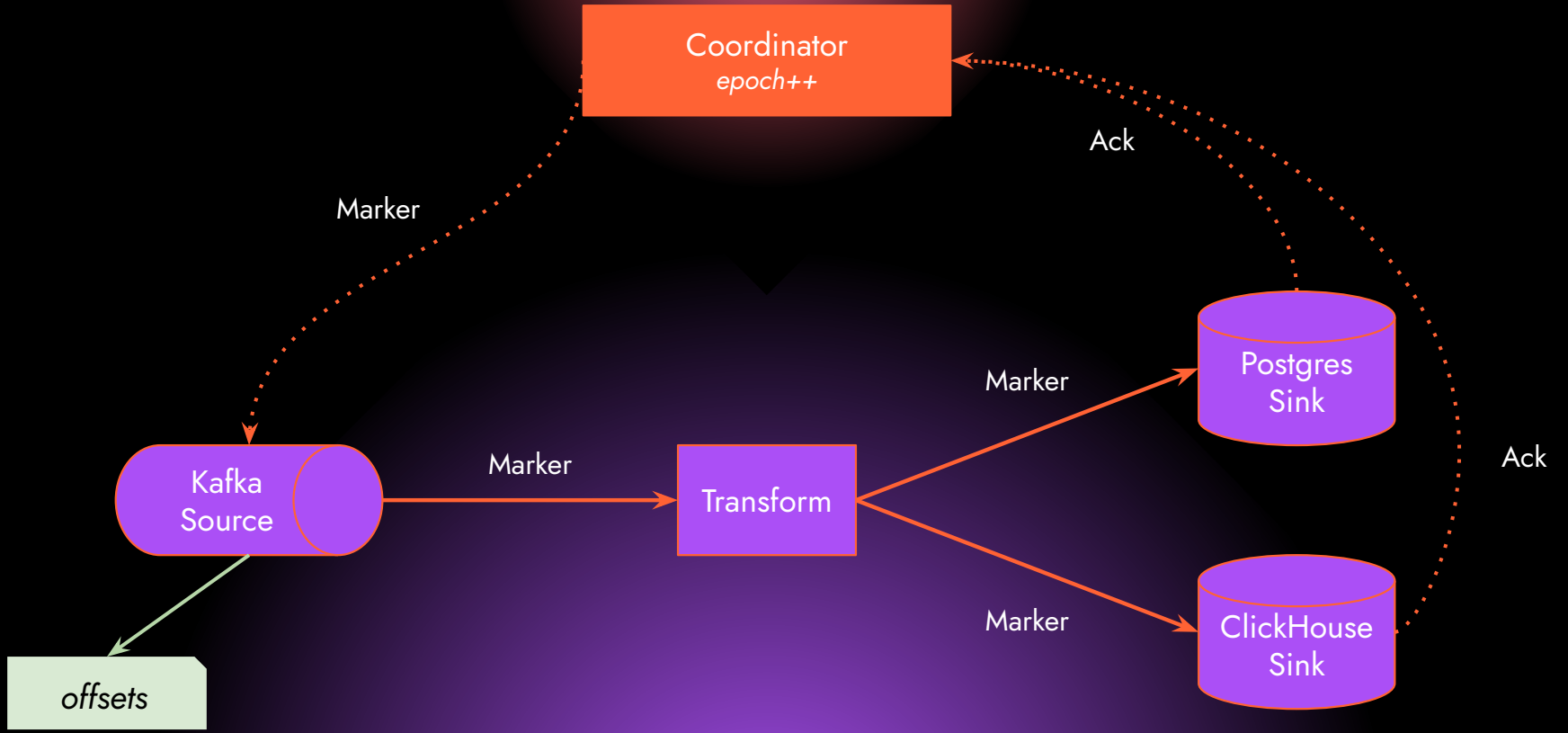
STATEFUL DISTRIBUTED OPERATORS

- → Need multi-node support (DataFusion is a single-node system)
  - 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

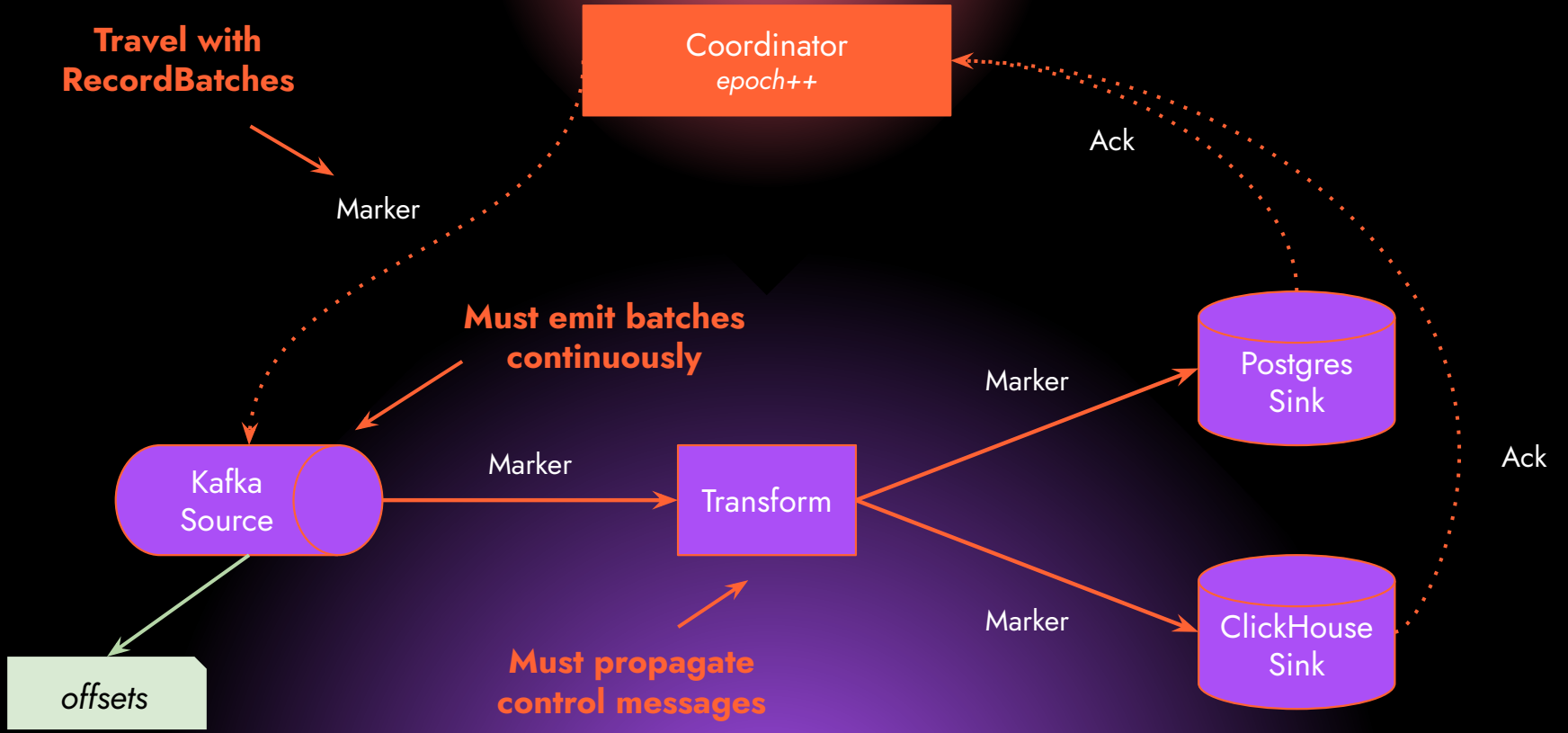
STATEFUL DISTRIBUTED OPERATORS

- → Need multi-node support (DataFusion is a single-node system)
  - 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

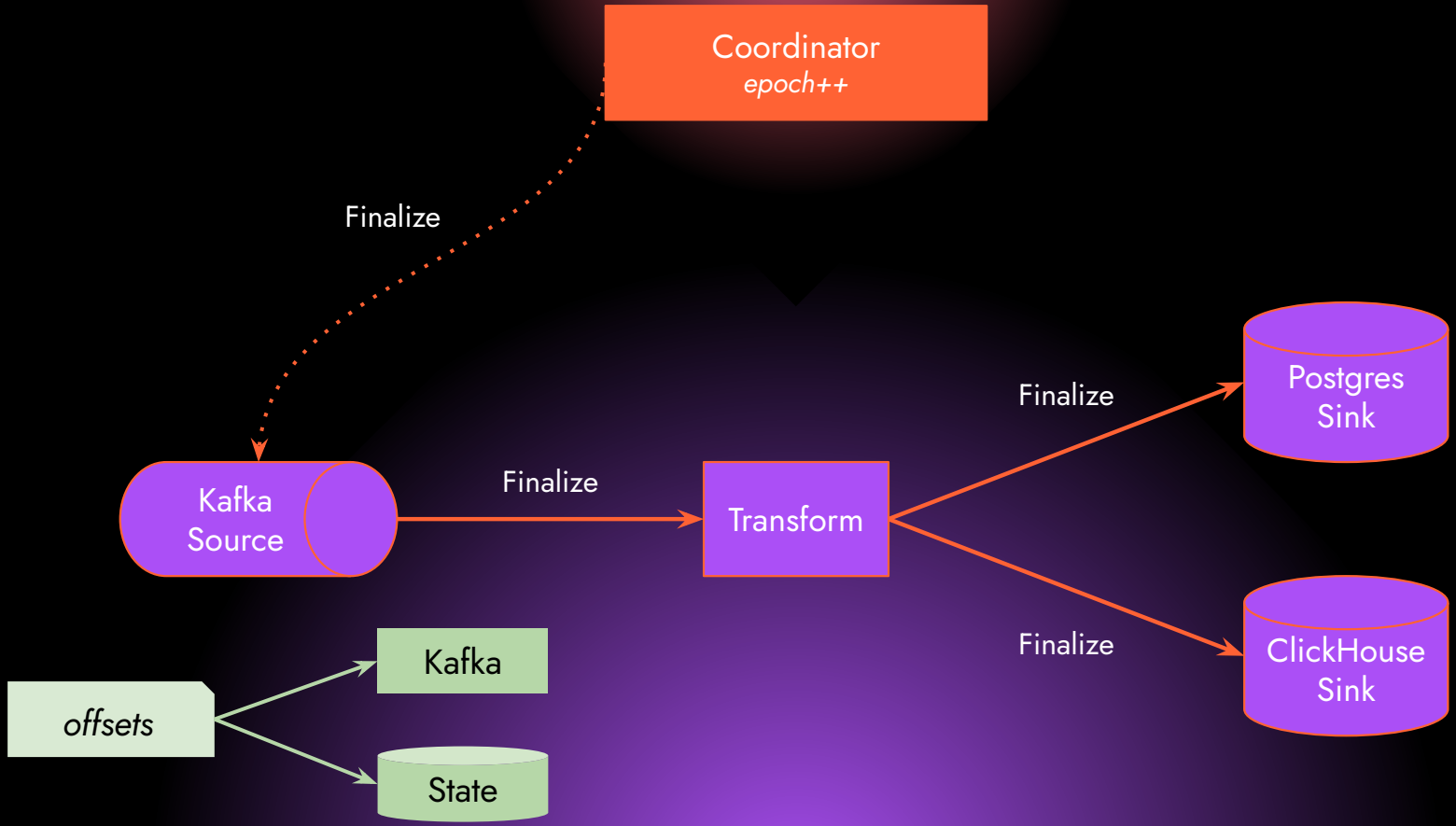

Kafka
Topic
6 partitions
2 partitions
2 partitions
2 partitions
Streamling ins
Streamling instance
Streamling instance
Postgres
DB



Chandy–Lamport Style Checkpointing Process: Phase 1



Chandy–Lamport Style Checkpointing Process: Phase 1



Chandy–Lamport Style Checkpointing Process: Phase 2

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() are stable functions.

→ 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.

```
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

```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

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!

- Usually no need for point lookups
- 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

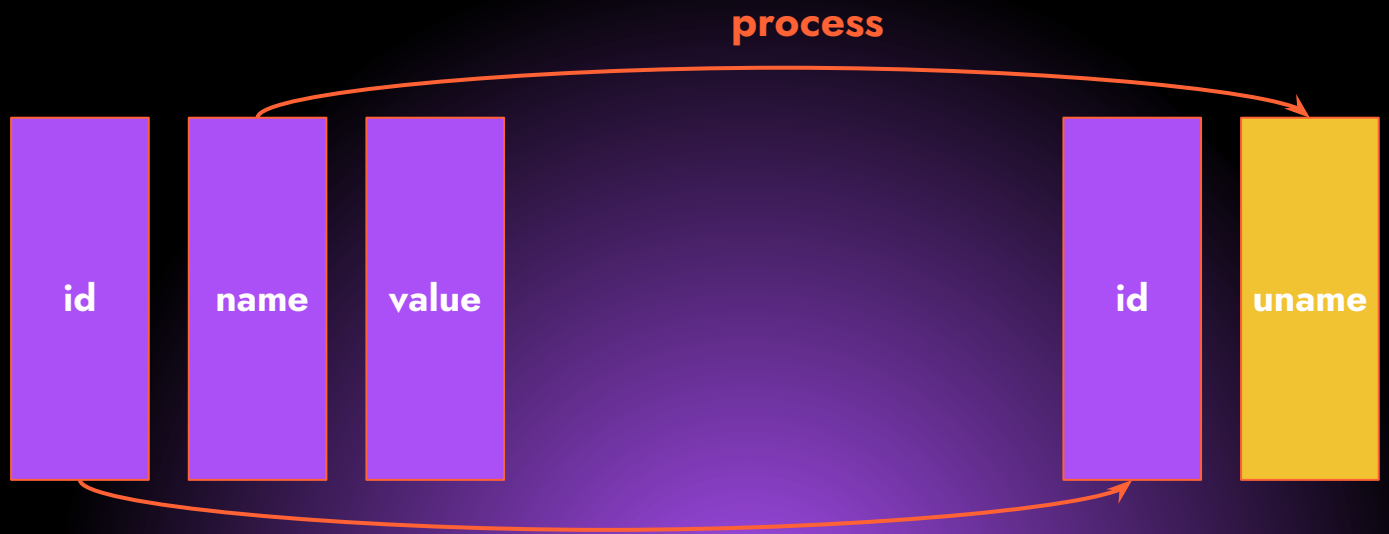
> For example, if we want at least 100 records in our batch to overcome fixed costs, the amount of time we need to wait to receive 100 records will depend on our throughput:
>
> - At 10 events/second, it takes 1 second
> - At 1,000 — 0.01 seconds (100ms)
> - At 1,000,000 — 0.0001 (0.1ms)

Micah Wylde

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**

```
SELECT id, upper(name) as uname
```



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**

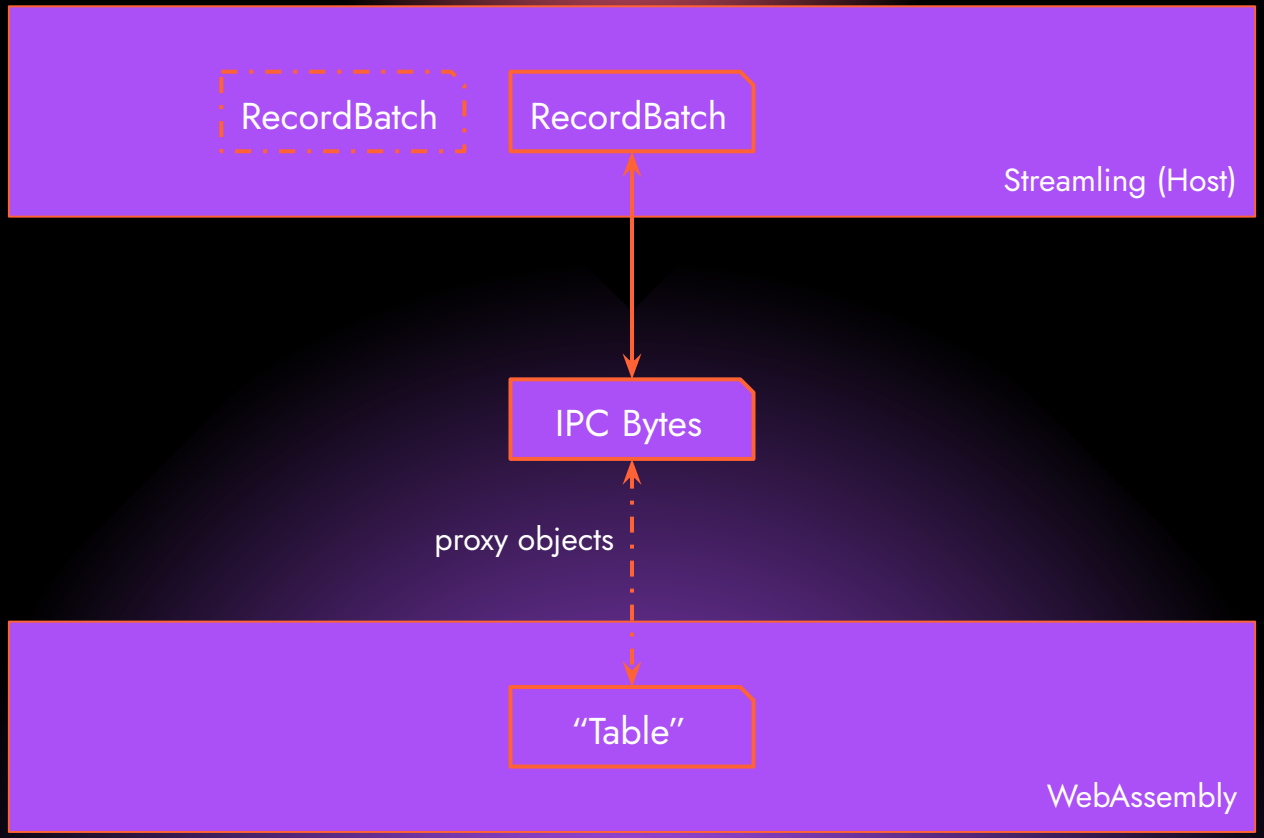
Many developers don’t want to write SQL!

```
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

PLUGIN SYSTEM

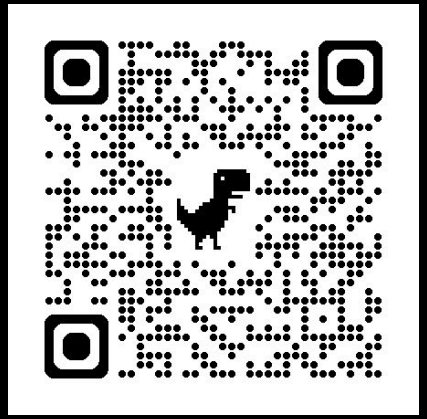
- Dynamically loaded binaries, configured with env vars
- Support for **source**, **transform**, **sink**, UDF and topology preprocessor plugins
- Friendly, high-level interface
- Zero-copy RecordBatch passing
- Runtime handles validation, backpressure and checkpointing

```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>;
}
```

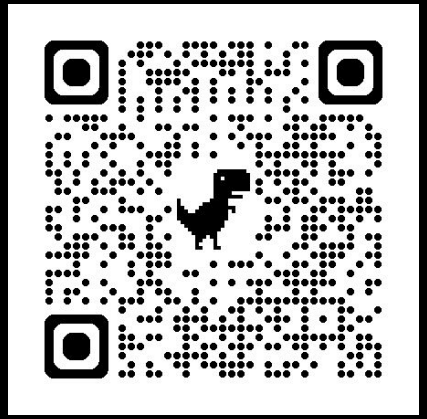
SUMMARY

- → 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!
- → If you haven't tried WebAssembly (or tried a while ago), try it now!

THANK YOU!



https://sap1ens.com



https://streamling.dev