From Stream Processing to Continuous and Deep Analytics

PARIS CARBONE  Senior Researcher-RISE, Committer @ Apache
SEIF HARIDI  Professor-KTH, Principal Sc. Advisor-RISE
The Core Team (RISE SICS + KTH)

Lars Kroll

Paris Carbone

Klas Segeljakt

Max Meldrum

Theodore Vasiloudis

Frej Drejhammar

Christian Schulte

Seif Haridi

Daniel Gillblad
Our Mission

Minimise Time Needed for Decision Making
A..Retrospective Notion of Data

- Databases, scalable processing frameworks etc. have been operating on input data …where ‘data’ is a **bounded** resource

**Dictionary**

**data**

/ˈdeɪta/ (ɪ)

**noun**

facts and statistics collected together for reference or analysis. There is very little data available.

**synonyms:** facts, figures, **statistics**, details, particulars, specifics, features; *More*

**Logic**

queries

tasks

functions

**COLLECTED DATA**

answers

reports

analytics

**RETROSPECTIVE PROCESSING**
Data-Programming - Before

Past Data Sets

Past Models
The Batch-Centric Approach
The Batch-Centric Approach
Origins of Data Streaming

- Data is born and evolves continuously as a stream (e.g., user interactions, sensor events, server logs)
The Paradigm Shift

**Logic**
- queries
- tasks
- functions

**COLLECTED DATA**
- answers
- reports
- analytics

**RETROSPECTIVE PROCESSING**
- Data
  - ...01110011
  - ...11100111
  - ...01110100

**CONTINUOUS PROCESSING**
- Logic
- paradigm shift
  - answers
  - reports
  - analytics

*real time...*
Stream Processing - Today

- No regrets to fast and reliable processing
Stream Processing - Highlights

Long Running Task Execution
• schedule-once / run forever
• dedicated compute resources
• no scheduler

Persistent, Flexible State
• embedded sharded state
• no external storage/databases
• async epoch commits (snapshots)
Reasoning about Reliability

- **Fault Tolerance**: The system needs to hide failures from the user transparently.
- **Consistency**: Output and states should offer read-committed isolation.
Epoch-Based Stream Processing

The Intuition

- **deterministic input streams**
- **system configuration (states) after completing an epoch**
- **divide computation into epochs**

**states**

- **success:** commit system configuration
- **failure:** abort and start from previous epoch

**reliable processing = epoch-based processing**
How we Guarantee Reliability

Logged Input

Committed Output

Committed System States

HDFS

S3
Synchronous Epoch Commits

- **prepare ep1**
  - ep1 prepared
  - commit ep1
  - ep1 committed
  - prepare ep2
  - ep2 prepared

Coordinato

Tasks

- **ep1**
- **ep2**

Π_{ep1} Stable Storage

- **compute**
- **idle**
Logged Input

Committed Output

Asynchronous Global State Snapshotting

async copies zero idle times

Committed System States

HDFS S3

\( \prod_{ep_i} \)

\( \prod_{ep_3} \)

\( \prod_{ep_2} \)

\( \prod_{ep_1} \)
Asynchronous Epoch Commits

Coordinato

Tasks

Stable Storage

prepare ep1

prepare ep2

ep1 prepared

commit ep1

ep2 prepared

commit ep1

How? Using Snapshots
Data Pipelines today need to **combine** diverse workloads!
(ML Training & Serving, Relational Algebra, Streams, Tensors, Graphs)
The Problem

critical decision making

End-to-End Data Pipeline

Why?

time until decision

Why?
Data Pipelines today need to **combine** diverse workloads and Frontends (ML Training & Serving, SQL, Streams, Tensors, Graphs)
Data Programming - The Big Picture

- Mis-match of processing guarantees and state management
- Excessive IO/ Data Movement of intermediate results
- Isolated HW Execution - No cross-framework optimisation
The Problem

critical decision making

time until decision

Data Pipeline

Data Pipeline (+Optimised)

Data Pipeline (+Hardware Accelerated)

Data Pipeline +Optimised +Hardware Accelerated
A Fundamental Challenge

Existing Optimisers target one...

- Model/Frontend (e.g., Relational Algebra, Graphs, Tensors)
- Language (e.g., Scala, Python)
- System/Runtime (e.g., Flink, Tensorflow)

Thus, we need a common underlying model, language and system
Arcon - A General Data Compute Architecture

- Dynamic Graphs
- Stream ML
- Online Reasoning
- SQL

Arc IR / Compiler

Arcon Runtime
Purpose of an IR

Data Pipeline

- Dynamic Graphs
- Tensors
- Relational Streams

Universal Translation
- Data Dependencies
- Data Types
- Operations on Collections

Logical Optimisation
- Type Inference
- Avoid Materialisation
- Min. Data Passes
- Match Data Layouts

Physical Optimisation
- IR Fragments -> Accelerator Code
- Instruction Selection (tree tilling)
- Cost Model Analysis
- Constraint Solving?
ARC

- **Arc** is an extension of Stanford’s **Weld IR** with streaming primitives

  **Weld:**
  - A language for describing data transformations
  - Side-effect free and hardware agnostic
  - Has an online compiler that produces **LLVM IR**
  - Simple: Builder, Value and Struct (Tuple) Types & Expressions

- **Arc Additions:**
  - Pipelined dependencies (sources, sinks)
  - Stream window discretisation and aggregation primitives
Weld Intro

- **Value types**: *Read-only* data types
  - Scalar + SIMD types: bool, i8...i64, u8...u64, f32, f64
  - Collections: vec, dict
- **Builder types**: *Write-only* data types
  - appender, merger, groupmerger, ...
  - Additive monads which construct value types
- **Structs**: {...}, similar to tuples
  - Structs of values are also values
  - Structs of builders are also builders
Weld Expressions

- **for** is a parallel loop over a collection (or iterator) and into a builder
- **merge** takes a builder and a value and produces a new builder with the value merged in (according to the builder’s semantics)
- **result** turns a builder into the corresponding type, i.e.
  - `append[i32]` into `vec[i32]` (map)
  - `merger[i32,+]` into `i32` (reduce)
- Builders are consumed when used, e.g., you may only call **result** on a builder once (*linear type*)
- **if**, **lookup**, math functions, binary ops, casts, C-UDFs, …
Weld Example

**Scala**

```scala
input.map(value: Int => value + 5)
```

**Weld**

```weld
|input: vec[i32] |
result(
  for(input, appender[i32],
    |app: appender[i32], index:i64, value:i32|
      merge(app, value + 5)))
```
Arc (so far)

- **Arc** generalises **Weld** with operator pipelining
- **Source** is a **collection** stream[T]
- **Sink** is a **builder** streamappender[T]
- Calling result on a **Sink** returns a **Source** and creates a **Channel** between them
- Special Primitives Added for **Stream Windows**
Arc Example

**Scala**  
```scala
input.map(i: Int => i + 5)
```

**Arc**  
```
source: stream[i32], sink: streammapper[i32] |  
for(source,  
sink,  
| out: streammapper[i32], i:i32 |  
merge(out, i + 5))
```
A Real Example

Beam and Pandas (Python)

```python
import apache_beam as beam
import apache_beam.transforms.window as window
import pandas as pd

def matmul(numbers):
    df = pd.DataFrame(numbers, numbers)
    s = pd.Series(numbers)
    return df.dot(s)

p = beam.Pipeline(...)
source = (p | beam.io.ReadFromPubSub(Kafka(...)) |
    'filter' >> beam.Filter(lambda x: x > 0) |
    beam.WindowInto(window.FixedWindows(60)) |
    'map' >> beam.Map(matmul) |
    'sink' >> beam.io.WriteToSink(Kafka)
)
p.run()
```

Arc Translation

type ts = u64, val = i32, elem = { ts, val }
|source: stream[elem], sink: streamappender[elem]|
let filtered = filter(source, |e: elem| e.$1 > 0);
let windowed = for(filtered,
    windower[unit, appender[val]](
        e:elem,_,_! { [v.$0/60L], ()},
        wm:ts,open:vec[ts],_! {filter(open,|t|t<wm),()},
        |t:ts,agg:appender[val]| {t.matmul(result(agg))},
        e:elem,w:windower| merge(w,e));
drain(windowed_stream, sink)
Compilation Pipeline

- **Frontends**
- **ARC**
- **Logically Optimised ARC**
- **Physically Optimised ARC**
- **Deployment Artefacts**

Available Resources
Stream Metadata
Code Generation

- Dataflow nodes are still (mostly) IR
- Translate nodes to Rust executables, generating glue code for other backends (e.g., Weld, CUDA, SDAccel, …)
- Specialise code for target architectures (known at this point!)
- Network IO, State Access etc. are provided by the runtime.
Distributed JIT Compilation

- Discovered better Plan
- Change in Resources
- Change in Load Distribution

constraint solver

Physically Optimised IR

Monitoring

Deployed Code

...
Arcon - A General Data Compute Architecture

- Dynamic Graphs
- Stream ML
- Online Reasoning
- SQL

Arc IR / Compiler

Arcon Runtime
Stream Processing

Runtime Baseline

complex control flows

dynamic scheduling

Hardware Acceleration

TPUs/
FPGAs

GPUs
Arcon - A General Data Compute Architecture

- Dynamic Graphs
- Stream ML
- Online Reasoning
- SQL

Arc IR / Compiler

Operational Plane

Arcon Runtime

Execution Plane
Arcon Runtime Components

• The two **sides of the coin:**

  • **1. Operational Plane:** ~ The Control Backbone
    • Dataflow Placement, Monitoring, Deployment, Snapshots
  
  • **2. Execution Plane:** ~ The Runnables
    • High Perf. IO, Channels, Flow Control, Dynamic Task Scheduling/Preemption, Special Hardware Acceleration
The General-Purpose Runtime of Arcon

Operational Plane

Appmaster

Statemaster

Execution Plane

Static

Dynamic

Tasks

workers

IO - Channels / State

Dynamic Scheduler

control data / snapshot metadata

dataflow deployment
Dataflow Deployment Example

```
type ts = u64, val = i32, elem = { ts, val };  
|source: stream[elem], sink: streamappender[elem]|  
let filtered = filter(source, |e: elem| e.$1 > 0);  
let windowed = for(filtered,  
    windower[unit, appender[val]](  
        |e:elem,_,_| {[v.$0/60L], ()},  
        |wm:ts,open:vec[ts],_| {filter(open, |t|t<wm),()},  
        |t:ts,agg:appender[val]| {t,matmul(result(agg))}),  
        |e:elem,w:windower| merge(w,e))};  
drain(windowed_stream, sink)
```

Appmaster starting up at 127.0.0.1:3000...  
Generating execution plan...100%  
Deploying Application

Source
name = Kafka

Operator
name = filter  
resources = 2G mem, 2 Cores  
status = deploying

Operator
name = map  
resources = 2G mem, 2 Cores  
status = running

Sink
name = Kafka

[success] Total time: 1 s, completed Mar 18, 2019 10:45:55 AM
Kompact Middleware

**KOMP**ics (Component Model) + **ACTors**

A Rust-based Middleware for building reliable, high performance distributed data processing systems on top.

- **Component Model**: Module Interfaces, Type Checking, Validation, Safety, Decoupling Spec + Implementation
  - e.g., channel properties (fifo, spilling, flow control) vs channel implementation (TCP, UDT)
- **Actor Model**: Virtualisation, Location Transparency, Simplicity

- Used solely to reliably build our **Execution Plane**

[https://github.com/kompics/kompact](https://github.com/kompics/kompact)
Next Steps

• Arc Extensions, Logical Optimisations
• Full Support of specific frontends (e.g., Calcite SQL)
• Semantics of dynamic computation
• Dynamic Task Scheduler + lineage tracking
• Design and Native Arcon Support of new DSLs