Developing (experimental) Rust SDK and a Beam engine for IoT devices

Sho Nakatani
TOYOTA Motor Corporation
@laysakura (GitHub)
Goal

- About Beam Rust SDK: Make it the 5th Beam SDK
  - Sharing the motivation behind its development
  - Presenting the current status of the project
  - Encouraging collaboration and gathering contributors

- About SpringQL:
  - Providing a brief overview of SpringQL, a stream processor specifically designed for IoT devices
About Me

- Research and Development in stream processing for cloud and IoT devices
  - Implementing SpringQL in Rust [GitHub repo](https://github.com/laysakura/SpringQL
- Recognizing Beam as standard stream processing model for the next 10 years
  - Desire to support the Beam model for SpringQL
- Active involvement in the development of Beam Rust SDK since February 2023

Sho Nakatani
laysakura

A low-level system developer / backend engineer in Tokyo.
Agenda

● Rust SDK Development (17 minutes)
  ○ Motivation
  ○ Design
  ○ Rust-specific challenges
  ○ History and future prospects

● Introduction to SpringQL & Integration with Beam (3 minutes)
Rust SDK: Motivation
Motivation for Rust SDK

- **For Pipeline Construction (or Programming)**
  - Leveraging Rust's *statically-typed* nature and generics
  - Meeting the demand from Rustaceans for a dedicated Beam Rust SDK

- **For Worker**
  - Memory safety
  - Performant
    - Comparing to Go: More lightweight runtimes (e.g. no garbage collection)
    - (My interest) High performance single-node SPEs with Beam model?
      - Relevant Research: Scabbard, SABER/LightSaber, StreamBox
      - "Do We Need Distributed Stream Processing?" (blog post)
      - "a single multicore server can provide better throughput than a multi-node cluster for many streaming applications"

<table>
<thead>
<tr>
<th>Throughput (million tuples/sec)</th>
<th>Spark</th>
<th>Flink</th>
<th>SABER</th>
<th>Handwritten C++</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>4.8</td>
<td>11.8</td>
<td>23</td>
</tr>
</tbody>
</table>

*Table 1: Single CPU core throughput for Yahoo Streaming Benchmark*
Rust SDK: Design
Where Rust SDK Works

- Rust SDK works in:
  - **Client** to construct pipelines
  - **Workers** to execute Rust-specific functions

- An application is built as a binary statically linked with the Rust SDK
  - Binaries are deployed to both Client and Workers
  - Different binaries are built from the same app (source)
Where Rust SDK Works

- Rust SDK works in:
  - **Client** to construct pipelines
  - **Workers** to execute Rust-specific functions

- An application is built as a **binary** statically linked with the Rust SDK
  - Binaries are deployed to both **Client** and **Workers**
  - Different binaries are built from the same app (source)
Where Rust SDK Works

- Rust SDK works in:
  - **Client** to construct pipelines
  - **Workers** to execute Rust-specific functions

- An application is built into binary statically linked with Rust SDK
  - Binaries are deployed to both Client and Workers
  - Different binaries are built from the same app (source)
Design Concepts

- Mainly influenced by TypeScript (features) and Go (compilation & deployment)
- Statically-typed pipeline construction
- Removal of Pipeline APIs (explained later)
- Asynchronous execution of workers

Note: The design concepts may require further synchronization with other contributors.
Design Concepts

- Mainly influenced by TypeScript (features) and Go (compilation & deployment)
- Statically-typed pipeline construction
- Removal of Pipeline APIs
- Asynchronous execution of workers

Note: The design concepts may require further synchronization with other contributors.

Show the concepts via a word-count pipeline
fn word_count(lines: PCollection<String>) -> PCollection<KV<String, i32>> {
    lines
        .apply(ParDo::from_map{
            // convert lines to lowercase
            |line| line.to_lowercase(),
        })
        .apply(ParDo::from_flat_map{|line|{
            // split a line into words
            line.split_whitespace()
        }})
        .apply(ParDo::from_map{|word|{
            // count each word
            KV::new(word, 1)
        }})
        .apply(ParDo::default())
        .apply(Combine::per_key(|values| values.count()))
}

#[tokio::test]
async fn main() {
    DirectRunner::new()
        .run(root|{
            let lines = root.apply(Create::new(vec![
                "And God said, Let there be light: and there was light",
            ]));
            let result = word_count(lines);

            result.apply(AssertEqualUnordered::new(&[
                KV::new("and".to_string(), 2),
                KV::new("god".to_string(), 1),
                KV::new("said".to_string(), 1),
                KV::new("let".to_string(), 1),
                KV::new("there".to_string(), 2),
                KV::new("be".to_string(), 1),
                KV::new("light".to_string(), 2),
                KV::new("was".to_string(), 1),
            ]))
        })
        .await;
    
}
Word-count pipeline...

```rust
fn word_count(lines: PCollection<String>) -> PCollection<KV<String, i32>> {
    lines
        .apply(ParDo::from_map(|| {
            line.to_lowercase()
        )))
        .apply(ParDo::from_flat_map(|| {
            line.split_whitespace()
        }))
        .apply(ParDo::from_map(|| {
            word
        }))
        .apply(GroupByKey::default())
        .apply(Combine::per_key(|| {
            values.count()
        }))
}
```

and its usage from DirectRunner

- **Statically-typed (w/ automatic type-inference)**
  - `line`: String
  - `line.split_whitespace()`: Vec<String>
    → flat-mapped into String
  - `word`: String
  - (output PCollection): KV<String, i32>
  - (output PCollection): KV<String, Vec<i32>>
  - (output PCollection): KV<String, i32>
Word-count pipeline...

```java
fn word_count(lines: PCollection<String>) -> PCollection<KV<String, i32>> {

  lines
    .apply(ParDo::from_map(
      fn from_map<F, In, Out>(func: F) -> ParDo
        where
          F: Fn(&In) -> Out,
          In: ElemType, Out: ElemType
    ))
    .apply(ParDo::from_flat_map(
      fn from_flat_map<F, In, Out>(func: F) -> ParDo
        where
          F: Fn(&In) -> Vec<Out>,
          In: ElemType, Out: ElemType
    ))
    .apply(ParDo::from_map(
      fn from_map(F) -> ParDo
        where
          F: Fn(&In) -> Vec<Out>,
          In: ElemType, Out: ElemType
    ))
    .apply(Combine::per_key(F, In, Out)
      where
        F: Fn(&In) -> Vec<Out>,
        In: ElemType, Out: ElemType
    )
    .apply(GroupByKey::default())
    .apply(Combine::per_key(F, In, Out)
      where
        F: Fn(&In) -> Vec<Out>,
        In: ElemType, Out: ElemType
    )
}
```

and its usage from DirectRunner

Statically-typed (w/ generics)

```java
fn from_map<F, In, Out>(func: F) -> ParDo
  where
    F: Fn(&In) -> Out,
    In: ElemType, Out: ElemType

fn from_flat_map<F, In, Out>(func: F) -> ParDo
  where
    F: Fn(&In) -> Vec<Out>,
    In: ElemType, Out: ElemType

fn per_key<F, In, Out>(func: F) -> Combine
  where
    F: Fn(&In) -> Vec<Out>,
    In: ElemType, Out: ElemType
```

BEAM SUMMIT NYC 2023
Runner.run() instead of Pipeline.run()
- Same API as TypeScript SDK.
Runner.run() introduce pipeline root (PValue)
- Proposed in a design doc.

Simplifying Apache Beam
or Pipelines Considered Harmful

[Image with code snippet]

https://s.apache.org/no-beam-pipeline

Robert Bradshaw (robertwb@google.com)
Rust SDK: Rust-specific Challenges
How to share functions? (between client and worker)

- **Functions (and closures)**
  - User-defined ParDo, CombineFn, Coder, ...
- Both binaries contain the same functions, but how does a **worker** determine which functions to execute?
How to share functions?
(between client and worker)

- Functions (and closures)
  - User-defined ParDo, CombineFn, Coder, ...
- Both binaries contain the same functions, but how does a worker determine which functions to execute?
  - From Fn API, worker receives:

```protobuf
message FunctionSpec {
    // (Required) A URN that describes the accompanying payload.
    // For any URN that is not recognized (by whomever is inspecting it) the parameter payload should be treated as opaque and passed as-is.
    string urn = 1;

    // (Optional) The data specifying any parameters to the URN. If the URN does not require any arguments, this may be omitted.
    bytes payload = 3;
}
```

Client

```
message JobSpec {
    // A set of tasks, each of which may be represented as a FnSpec
    repeated FnSpec fn_specs = 1;
}
```

Runner

```
message FuncSpec {
    // A list of parameters to accompany the FnSpec
    repeated FuncParam fn_params = 1;
}
```

Runner API

```
message FnResult {
    // The results of running the FnSpec
    repeated FuncResult fn_results = 1;
}
```

Binary

```
Rust SDK
```

```
Other SDK
```

Harness

```
Portable Runner
```

```
Other Runner
```

Runner API

```
Input:
- Pipeline structure
- Functions (DoFn, CombineFn, Coder, ...)
- Input records

Output:
- Output records
```

Input

```
- Pipeline structure
- Functions (DoFn, CombineFn, Coder, ...)
- Input records
```

Output

```
- Output records
```

message FuncParam {
    // The name of the parameter
    string name = 1;

    // The value of the parameter
    string value = 2;
}

message FuncResult {
    // The result of running the FuncSpec
    repeated FuncParam fn_results = 1;
}
```
How to share functions? (between client and worker)

- How does a worker decide which function to execute?
- Deserialize function body from payload?
  - Cannot serialize functions in Rust (especially for generic ones).
    - See discussion in a design doc for detail

```c
message FunctionSpec {
    // (Required) A URN that describes the accompanying payload.
    // For any URN that is not recognized (by whomever is inspecting
    // it) the parameter payload should be treated as opaque and
    // passed as-is.
    string urn = 1;

    // (Optional) The data specifying any parameters to the URN. If
    // the URN does not require any arguments, this may be omitted.
    bytes payload = 3;
}
```
How to share functions? (between client and worker)

- How does a worker decide which function to execute?
- Function symbols in URN?
  - No reflection in Rust (cannot call function from its symbol)
  - Closures are unnamed
  - Different from Go SDK

```message FunctionSpec {
  // (Required) A URN that describes the accompanying payload.
  // For any URN that is not recognized (by whomever is inspecting it) the parameter payload should be treated as opaque and passed as-is.
  string urn = 1;

  // (Optional) The data specifying any parameters to the URN. If the URN does not require any arguments, this may be omitted.
  bytes payload = 3;
}
```

- Rust SDK
  - Binary aarch64 / macOS
  - Rust SDK Harness

- Other SDK
  - Binary x86-64 / Linux
  - Other SDK Harness

- Runner API
  - Input:
    - Pipeline structure
    - Functions (DoFn, CombineFn, Coder, ...)
    - Input records
  - Output:
    - Output records

- Other Runner
  - Runner
  - Other SDK
  - Harness

- Portable Runner
  - Runner API

- Fn API
  - Input:
    - Pipeline structure
  - Output:
    - Output records
How to share functions? (between client and worker)

- How does a worker decide which function to execute?
- Registering such map? “URN → function pointer”
  - Init function might register the map
    - Note that function pointers differ in Client and Worker (different binary)
  - Requires macro and further implementation efforts, but seems not a bad idea

Rust SDK

```
Binary
aarch64 / macOS
```

Runner API

```
Input:
- Pipeline structure

Runner

Portable
Runner

Other
Runner
```

```
Input
- Functions (DoFn, CombineFn, Coder, ...)
- Input records

Output
- Output records
```

```
Fn API
```

```
Binary
x86-64 / Linux
```

```
Rust SDK Harness
```

```
Other SDK Harness
```

```
Other SDK
```

```
Client
```

```
Other SDK
```

BEAM SUMMIT NYC 2023
How to share functions? (between client and worker)

- How does a worker decide which function to execute?
- Registering such map? "URN → function pointer"
  - Init function might register the map
  - Note that function pointers differ in Client and Worker (different binary)
  - Requires macro and further implementation efforts, but seems not a bad idea

We are currently working on the development of safe serialization for functions.
Rust SDK: Development history and future
Why History?

- While I currently serve as the repository owner of the experimental Beam Rust SDK, I am not the project's original contributor.
- It is important for me to acknowledge and honor the contributions of past and current individuals involved in the project.

I apologize if I have unintentionally omitted mentioning any specific contributor names.
Started from a JIRA Ticket

- The Rust SDK issue was created in July 2021 on [JIRA](#).
- There was a recommendation to learn from the TypeScript SDK.
- An initial concept of pipeline construction was shared in a [Gist](#).
- Contributor: jayendra13
- Advisers: kennknowles, robertwb, lostluck
● The issue on GitHub is still active to this day
● Experimental implementation repos:
  ○ kennknowles/beam [old]
  ○ ↓ (merged into)
  ○ nivaldoh/beam [old]
  ○ ↓ (forked to)
  ○ laysakura/beam [current]
● Organizer: brucearctor
Project initiation: January 2023

The Google Cloud Dataflow team started a Rust SDK development

Later merged into nivaldoh/beam repository

Contributors
  - antonbobkov
  - robertwb
  - JayDosunmu
  - y1chi
[Old repo] nivaldoh/beam

- Project initiation: November 2022
- Added:
  - Codes for pipeline construction (partial)
  - Worker codes (partial)
- Development activities ceased since February 2023
- Contributors
  - nivaldoh
  - sjvanrossum
  - laysakura (me)
  - Miuler
Project initiation: February 2023
- Forked from nivaldoh/beam
- Added:
  - Coder serialization (partial)
  - More worker codes (partial)
  - General function serialization (doing)
  - The Beam Programming Guide for Rust (doing)
- Contributors
  - dahlbaek
  - sjvanrossum
  - Kelvinyu1117
  - laysakura (me)

Unfortunately, it seems that nivaldoh's repository is inactive as of February 1st, 2023. There are 5 pull requests that have not been reviewed or merged.

To address this issue, I have created a fork of the repository. In my fork, I have:

- hand-merged a topic branch from @robertwb
- (wip) stopped using .txt, and instead used generics for PTransform in-out parameters
- made many other refactorings to make the code more Rust-like

Apache Beam Programming Guide

The Beam Programming Guide is intended for Beam users who want to use the Beam SDK to build and test your pipeline. The process is a language-agnostic, high-level guide to programmatically building your Beam pipeline. It includes code samples in multiple languages to help illustrate how to implement Beam.

If you want a brief introduction to Beam's basic concepts before reading the progr
Future work

- Technically challenging implementations
  - Serialization/deserialization of functions (including closures), led by sjvanrossum
- Align design considerations for non-trivial features
  - Registration of user-defined objects (possibly through init function w/ macros)
  - Coders (custom coders, row coders, etc.)
  - Artifact staging service
- Completion of the Programming Guide and working examples
- Call for more contributors!
  - Will create good-first issues in laysakura/beam
SpringQL: Introduction and integration with Beam
SpringQL’s Target

- Stream Processing Engine for IoT devices
  - Targeting middle-to-high end devices
    - Raspberry Pi
    - Connected vehicles

- Support semi-realtime stream processing
  - Input:
    - Sensor data
    - UI
  - Output:
    - Device actuation
    - Aggregated data (sent to edge/cloud)
    - UI (display, sound, …)
SpringQL’s Current Status

- Implemented in Rust ([repo](https://github.com/springql))
- Distributed as libraries:
  - Rust (static)
  - C (static / dynamic)
- User interface
  - Client: Rust / C
  - Pipeline construction: SQL-like
  - Operation: Streaming SQL
- Problems
  - Difficulty in constructing DAGs using SQL-like language
  - Limited operations available through streaming SQL

Desire to utilize Beam for U/I
Initial Idea: Integration with Beam

- App, Beam SDK, and SpringQL library are all within the same process and binary
- SpringQL library serves as:
  - Client interface
  - Dedicated runner
- SpringQL Runner receives pipeline graph via Runner API in protobuf format
- SpringQL runner calls SDK Harness to execute UDFs
  - May use “LOOPBACK“ SDK Harness (config doc)

Diagram:

- Beam Rust SDK
- SpringQL rlib (Client I/F)
- Runner API
- Runner
- Pipeline proto
- Fn / bundle proto
- Fn API
- Worker
- Beam Rust SDK Harness

Diagram notes:

- 1-process / 1-binary
Initial Idea: Integration with Beam

- App, Beam SDK, and SpringQL library are all within the same process and binary
- SpringQL library serves as:
  - Client interface
  - Dedicated runner
- SpringQL Runner receives pipeline graph via Runner API in protobuf format
- SpringQL runner calls SDK Harness to execute UDFs
  - May use "LOOPBACK" SDK Harness (config doc)
Initial Idea: Integration with Beam

- App, Beam SDK, and SpringQL library are all within the same process and binary
- SpringQL library serves as:
  - Client interface
  - Dedicated runner
- SpringQL Runner receives pipeline graph via Runner API in protobuf format
- SpringQL runner calls SDK Harness to execute UDFs
  - May use "LOOPBACK" SDK Harness (config doc)
Initial Idea: Integration with Beam

- App, Beam SDK, and SpringQL library are all within the same process and binary
- SpringQL library serves as:
  - Client interface
  - Dedicated runner
- SpringQL Runner receives pipeline graph via Runner API in protobuf format
- SpringQL runner calls **SDK Harness** to execute UDFs
  - May use “LOOPBACK” SDK Harness ([config doc](#))
Summary

- About Beam Rust SDK
  - Motivation behind its development
  - Current status of the project
  - Call for contributions
- About SpringQL
  - SpringQL’s target systems and architecture
  - Integration idea with Beam
QUESTIONS?

Twitter: @laysakura
GitHub: laysakura