თბილისის თავისუფალი უნივერსიტეტი

კომპიუტერული მეცნიერების, მათემატიკისა და ინჟინერიის სკოლა (MACS[E])

კომპიუტერული მეცნიერების და მათემატიკის პროგრამა

თოფურია თამთა

ხოხიაშვილი ალექსანდრე

ნოზაძე ზვიად

საბაკალავრო პროექტი

Rust Resilient Distributed Datasets

ხელმძღვანელი: ლეკვეიშვილი გიორგი

თბილისი

2022

ანოტაცია

ჩვენი პროექტი არის Apache Spark-ის RDD API-ის Rust კომპილირებად ენაზე დაწერა და მიღებული სისტემის სიჩქარის შედარება ორიგინალ Spark-თან, რომელიც Scala-ზე არის იმპლემენტირებული. Spark არის სისტემა დიდი ზომის მონაცემების დისტრიბუციულად დამუშავებისთვის. ჩვენ გვაინტერესებს, თუ რა მოგების მიღება შეიძლება ასეთი სისტემის native ენაში დაწერით და მისი შედარება ინდუსტრიაში არსებულ აქტიურად გამოყენებად სისტემა Spark-თან. ასევე საინტერესოა რამდენად მორგებადია Spark-ის API Rust-ის უნიკალურ type სისტემასთან და საერთოდ შესაძლებელია თუ არა Spark-ის გადაწერა ასეთი განსხვავებული მოთხოვნებისა და შეზღუდვების მქონე ენაზე.

დღეს software-ის დიდ ნაწილს აქვს ფუფუნება, რომ იყოს არაეფექტური ჩქარა განვითარებადი hardware-ის გამო. პერსონალურ კომპიუტერზე გაშვებული პროგრამის performance დიდად მნიშვნელოვანი არ არის, რადგან ის, დიდი ალბათობით, მაინც საკმარისად ჩქარი იქნება. Distributed computing-ის ბიბლიოთეკებს ამის ფუფუნება არ აქვთ. ყველა ოპტიმიზაცია, რომლის მიღებაც შესაძლებელია, გამოსადეგია, რადგან ეს დიდი კომპანიებისთვის პირდაპირ ითარგმნება hardware-ზე დაზოგილ თანხაში. საუბარია იმდენად დიდ მონაცემებზე, რომ ერთი კომპიუტერის რესურსებით შეუძლებელია მათი დამუშავება. ასეთი სისტემების performance ძალიან მნიშვნელოვანია და ჩვენი მიზანია ვნახოთ შესაძლებელია თუ არა მისი გაუმჯობესება.

მიუხედავად უამრავი დაბრკოლებისა, რომლებსაც ქვემოთ უფრო ვრცლად განვიხილავთ, მივიღეთ საკმაოდ კარგი შედეგები.

Annotation

Our project is implementing Apache Spark's RDD API in Rust compiled language and comparing its performance to original Spark implemented in Scala. Spark is a system for distributed computation of Big Data. We want to know what kind of improvements will implementing Spark in a native language give us. It is also interesting how to fit Rust's unique type system to Spark's API and see if it is possible to implement Spark with such different limitations.

Today the majority of software has the privilege to be inefficient because of the rapidly improving hardware. That's why performance of a running program on a personal computer is not that important, as it will most likely be sufficiently fast. In distributed computing, we don't have this kind of privilege. Every optimization that we can get translates into saved costs on hardware for big companies. We are talking about such large data that is impossible to process with one computer's resources. Performance of such systems is vital and our goal is to see if it is possible to improve it.

We got good results despite many obstacles which we will discuss later in this report.

Thank you Page

We want to thank Giorgi Lekveishvili for his awesome tips and thoughtful advice

Figures used:

Introduction

Processing data on distributed servers has been an innovative way to work on Big Data. It takes parallel computing to another level, and can even process huge data using a big cluster of fairly cheap hardware. Although it might sound excellent on paper, distributed systems have many obstacles to slow the processing down and/or make implementation significantly harder and prone to bugs. These obstacles are: network communication, fault tolerance, consensus, event reordering, partitioning, and availability.

In a Distributed Systems course at our university, we learned the inner workings of a few such systems. One of these systems was Apache Spark. It is implemented in Scala, however, there is a much more widely used implementation of Spark on Python - PySpark. Today most of the users of PySpark use either Dataframe or DataSet APIs, but at its core, every operation is still done using RDDs in Scala. It is a very well-documented and measured fact that PySpark's RDD API is much slower than Scala's RDD API.

During the Data Engineering course, we worked on PySpark and listened to our lectures' concerns about it being slow on certain tasks. This gave us the idea that implementing Spark's RDDs in a native language might give us significantly better results than being implemented in an interpreted language like Scala. We were hoping we could see improvements similar to the jump from PySpark to Scala Spark between our Spark and "Native Spark".

Our next task was to choose an appropriate native language for this project. After careful consideration we chose Rust. Our reasoning was that it is type safe so it can express RDD API's functional paradigm and check all the closure types at compile time. It is also pretty close to Scala's type system. Also, Rust is famous for being fast and it's compiled down to LLVM, which itself does additional optimization passes, until it finally reaches x86-64. We also learned that one of the main reasons for picking Scala was because it was one of the few languages that had serializable lambda functions. Although it is thought that functions in compiled languages are not serializable we devised a unique way to make it happen.

This project is a research around the possibility of more efficient implementation of Spark RDDs without compromising on user friendly API. We researched if it's even possible to implement Spark on Rust and what kind of improvements it gives us. Our results not only provide fundamental ideas for further research, but we provide the solutions for implementing this distributed computing system in Rust. Main challenges being: function serialization, trait object serialization, api design, architecture design, networking, multicore processing, deploying.

Our user API is also in Rust. Rust has proven to be a harder language to write in, with a steep learning curve. Thus Rust software engineers are harder to find. However we believe that using our system would still be worth it, since intended users for our project are large corporations which care about shaving off every single percent of their computing costs. Those users can easily afford a few rust developers to implement long-running and cpu-bound tasks in our system.

Related work

Research about distributed data computing started a long time ago. One of the first widely used papers and implementations of such systems is Google's MapReduce[3]. This is the system we want to discuss because it is the only widely used system implemented in a native language - C++. Thus having no VM overhead in contrast to Spark. It is also different from our system as it can be run on nodes with heterogeneous (different operating systems/processor) hardware.

Google's Map-Reduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

MapReduce has many drawbacks compared ro Spark and our system.

First of all, MapReduce is limited to map and reduce operations only. So complex computations need to be expressed using multiple map-reduce tasks and makes it much harder to implement mapreduce jobs: e.g. K-means.

Secondly, in contrast to Spark, MapReduce is not RAM first, meaning computations always rely on persistent storage. This significantly slows down overall performance.

Last but not least, MapReduce offers a poor API: user defined functions are defined separately from MapReduce. Which complicates the build process. Since user defined functions can't be serialized, they must be built separately as shared libraries and those binaries need to be sent over the network to the compute nodes. There arises a much bigger problem because of this separate compilations: C compiler can't make global (project level) optimizations since project and user defined map/reduce functions are

compiled separately. So these optimizations are not possible: function inlining, cross function redundant computation elimination and more.

Apache Spark

Apache Spark's [1] innovation was to extend the idea of Map/Reduce by representing the user's computation as a graph of RDDs and also improving on its performance by doing most of the computation in RAM. At its core lies an abstraction called RDD, which represents an immutable distributed list of data. By applying transformations to the RDDs, user is able to build up a graph of RDDs where edges are certain transformations. When the user wants to compute the results they use "actions" which starts the computations on a distributed cluster. Each RDD is divided into partitions which are executed in parallel when possible.

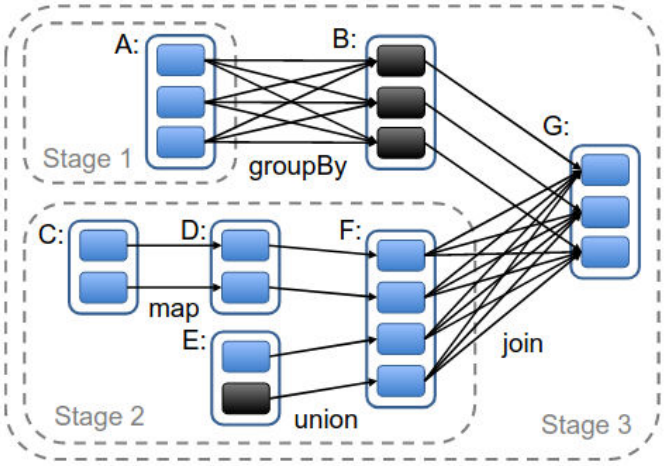

Fig. 1 Spark RDD lineage graph [1]

Since our project is all about improving the performance of apache spark let's review how the users' computation finally ends up running on the actual CPU:

1. User defines the computation graph by providing several closures which define the computation that the user wants to run
2. Since spark is written in Scala each closure ends up being a series of JVM instructions. This means that spark is able to serialize this closure at the master node and send it over to workers for execution on actual data.

    a. e.g. one such closure could be passed into `.map(...)` function
3. When the worker receives this closure it runs it using the iterator on the materialized data
4. All of this ends up being run by JVM and hot loops will actually go through the JIT process. This will try to JIT the JVM into x86 code with JVMs best efforts but unfortunately, scala iterators are too complex for JVM to optimize well enough, they still end up doing a lot of extra work besides pure user computation.
    a. For example, if we have an `RDD<int>` and we want to multiply each element by two, if we use `.map(x=>2*x)` scala will end up actually doing allocations and function calls besides just multiplying a single element by two.
    b. Some academic teams are trying to improve on these problems but the research is far from production. [3]

Spark RDD API is very comfortable to use for developers since all of the closure types are type-safe and inferred. Our goal was to retain the core design choices of this API to our best efforts.

Spark is probably the most used distributed compute engine in the industry. But most of the users don't end up writing their applications using low-level scala RDD API. Today this API is abstracted away using DataFrame and DataSet abstraction which offers SQL-like query language for writing distributed compute tasks. These new APIs are implemented for the Scala and Python versions of Spark. But an interesting fact is that all of these still use the core RDD APIs. The computation graph defined using SparkSQL is heavily optimized using Spark Catalyst into a final execution plan for RDDs. **All of this means that any optimizations for the underlying RDDs are optimizations for the entire spark ecosystem.**

Attempts at similar projects

Github user `rajasekarv` has also tried to implement spark RDD like api using the same language as us - Rust in their personal project `vega` [4]. We believe that some of the obstacles that we faced are solved in inefficient ways in their implementation. Analyzing their project is not in the scope of this report.

Our goal

Our project aims to explore the improvement opportunities of the API lying at the core of the Spark ecosystem. We implemented RDD API using a modern compiled language, Rust, and will discuss all the obstacles and findings in the next section.

Technical Details

Function Pointer Serialization

Most of the RDD API heavily relies on closures to specify users' actions. But there's one problem, these closures are intended to be executed on the worker nodes, while they're constructed on the master node. This problem asks for a way to serialize/deserialize closures. Spark is able to serialize its lambda functions and its captured values because each closure is just JVM instructions and some java objects. Meanwhile, in rust or other compiled languages functions are included in the binary and there's no way to actually figure out where the function resides, and it might call other functions so literally taking assembly on master and sending it over to worker was out of the question/impossible.

Our solution was to make sure that the identical binary was running on all the nodes. Assuming the above, each function is located at a fixed offset inside the binary on

each node. One more problem is that in modern systems, due to Address space layout randomization(ASLR), binaries are loaded at a random offset inside the memory space. So, in order to send over the function, we serialize a function pointer as a relative offset to some fixed BASE inside the binary, and send that over instead. See fig. 2

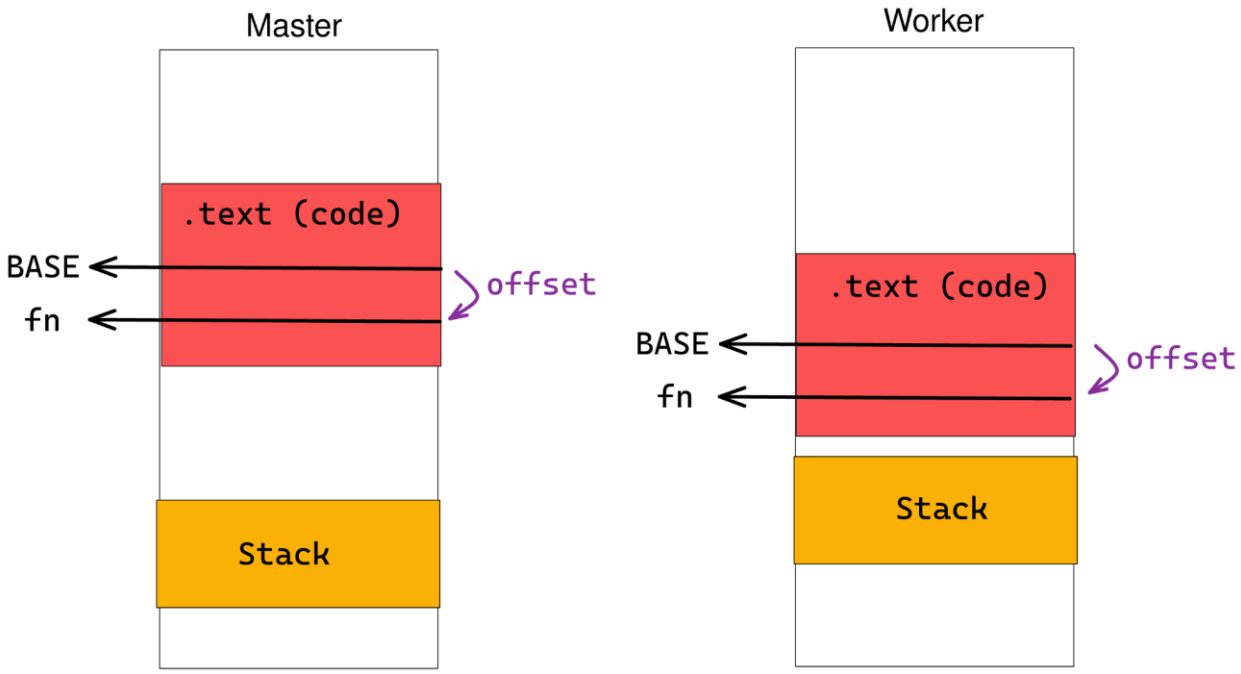

Fig. 2: visualization of function pointer serialization

In the figure above, the value of "offset" will be sent over the network. The worker can then add that offset to the BASE and derive the function pointer that the master had.

We've implemented function pointer serialization as a separate crate serde-fp which can be found inside our main repository.

RDD API design

Spark's RDD API design is one of its core innovations. We wanted to keep its user-friendliness while also adapting it to be idiomatic to rust programming language. We went through four different iterations of API design and settled on the one we believe has the best tradeoffs.

We had to change the API in the following manner:

```
Spark:          rdd.map(x => 2*x)
R2D2:           sc.map(rdd, |x|2*x)
```

When a user wants to call a transformation on an RDD they have to call it on the SparkContext object. This decision was the most sensible one due to Rusts lack of GC and its unique ownership model. We tried implementing the first version in Rust but it would result in cyclic references between RDD and the context objects. Even though we were able to make it work, we ended up with unnecessary complexity and non-idiomatic rust code. Additionally, Scala is able to serialize a graph of java Objects with no hassle using kryo library, which traverses over object graph in runtime. Due to rust having no reflections or type information in runtime there's no way we could achieve to serialize graph of objects connected by reference counted pointers. Due to all of the above problems, we made a compromise and settled on our final API.

Let's look at our sc.map function signature:

```
fn map<T: Data, U: Data>(&mut self, rdd: RddIndex<T>, f: fn(T) -> U) -> RddIndex<U>;
```

Each of our transformations returns RddIndex which is a type storing id of an RDD node inside the lineage graph. But one important detail is that our RddIndex is also generic over a T where T is the element stored inside the RDD partitions. Due to this, we are able to achieve type safety at compile time like Spark. As you can see in the above signature map function is generic over T and U and will only accept RddIndex over T and a closure which takes T and returns U. After this call we return RddIndex over U. User themselves won't be able to create a fake RddIndex since it's creating is limited only inside our library, this all means that every RddIndex is also guaranteed to represent a unique node inside the graph.

Our core API:

All of the functions except “actions”(e.g. collect) are lazy, meaning that they don’t start execution and just add a node to the lineage graph. Whenever an action is called all of the dependencies for that action are executed on workers.

```
actions:
fn collect(RDD[T]) -> Vec<T>
fn save(RDD[T], serializer, dir_path);
fn sort(RDD[T:Ord]) -> Rdd[T]

source:
fn new_from_list(Vec<Vec<T>>) -> RDD[T]
fn read_partitions_from(dir_path) -> RDD[file_path, content]

narrows:
fn map(RDD[T], fn: T -> U) -> RDD[U]
fn map_partitions(RDD[T], fn: Vec<T> -> Vec<U>) -> RDD[U]
fn flat_map(RDD[T], fn: T -> Vec<U>) -> RDD[U]
fn filter(RDD[T], fn: &T -> bool) -> RDD[T]
fn union(Vec<RDD[T]>) -> RDD[T]
fn sample(RDD[T], amount_per_partition: Int) -> RDD[T]
fn map_with_state(RDD[T], mapper)
fn flat_map_with_state(RDD[T], flatMapper)

wide:
fn shuffle(RDD[K, V], Partitioner[K], aggregator);
fn group_by(RDD[K, V], Partitioner[K]) -> RDD[K, Vec<V>]
fn partition_by(RDD[T], Partitioner[T]) -> RDD[T]
fn sum_by_key(RDD[K, V:Add], Partitioner[K]) -> RDD[K, V]
fn cogroup(RDD[K, V], RDD[K, W], Partitioner[K]) -> RDD[K, (Vec<V>,
Vec<W>)]
fn join(RDD[K, V], RDD[K, W], Partitioner[K]) -> RDD[K, (V, W)]
```

You can view exact signatures and bounds for the above functions here.

Runtime Type Erasure

In spark, the entire graph is accessible by traversing RDDs from a single one. This means that the lineage graph is represented as java objects inside the heap with references to each other. In scala, serializing such a graph is possible using the Kryo library which is

able to traverse object graph using runtime reflections. A similar thing is very hard to do in rust due to the fact that type information is erased at runtime. Additionally, when deserializing an object in rust you need to know the exact type that you're deserializing, but most of our RDDs are generic over types that are specified by the user code. All these user types will be known by the master when they are instantiated but workers will have no idea what kind of RDDs are inside the serialized graph. This means that the worker code will not able to specify the concrete type for deserialization. That's why we had to opt for trait object(similar to *holding an object as an interface* in other languages) serialization. This means that once the graph is delivered to workers all of the user-specified types are gone and all we have as nodes are trait objects for the RddBase trait. Our executors will interact with RDDs inside that graph through this RddBase interface and in turn, will be able to do actions that will actually involve the concrete types that the user specified.

Monomorphization performance benefits

Imagine we have an RDD holding integers and we want to multiply all the elements by two. Both our and spark API offers a map function for this. The code snippet that user will write will be the following:

```
Spark:          rdd.map(x => 2*x)
R2D2:           sc.map(rdd, |x|2*x)
```

Both of these closures will be executed for each element in the RDD inside the worker code. Both of these internally will also turn to a map function over the partition elements iterator:

```
Internal code:  partition.map(user_function_pointer).collect()
```

This function pointer comes from the serialized graph so neither rust nor scala compiler/JIT can possibly optimize this call over the code inside the closure. But we were able to find a way to leverage Rust's ability to monomorphize generic code to achieve

optimizations even over the user-specified code. To get these benefits we had to make user API a bit different. To multiple numbers by two user would have to write:

```
#[derive(Clone, Serialize, Deserialize)]
struct MulByTwo;

impl Mapper for MulByTwo {
      type In = usize;

      type Out = usize;

      fn map(&self, v: Self::In) -> Self::Out {
       v*2
      }
}
...
sc.map_with_state(rdd, MulByTwo)
```

Now, MapRdd will be generic over type M where M is a Mapper. Since rust will generate separate assembly code for every generic variant of a type(MapRdd), this means that for any specific Mapper rust will generate a separate partition mapping code. So, the code snippet below will know the exact function which is used to map the partition elements.

```
partition.map(|x|self.mapper.map(x)).collect()
```

We were able to confirm this behavior by disassembling and reviewing binaries generated by both techniques: passing if function pointer through API vs passing in type as a generic parameter and having map function inside that type's implementation.
We disassembled the generated code which ends up mapping the partition elements. Our disassembler gave us the code below:

```
n = param_2[2];
free(param_2);
if (data == 0) goto LAB_002c9bdb;
if (n != 0) {
  pcVar1 = (code *)self->fnptr;
  i = 0;
  do {
                    /* try { // try from 002c9b85 to 002c9b87 has its CatchHa
    res = (*pcVar1)(*(undefined8 *)(data + i * 8));
    *(undefined8 *)(data + i * 8) = res;
    i = i + 1;
  } while (n != i);
}
local_48 = (long *)data;
uStack64 = lVar2;
```

Fig. 3: work function with serialized function pointers

as you can see `self->fnptr` is fetched from MapRdd structure and when each element is mapped we have an overhead of a function call at the very least. This means that rustc was not able to inline the implementation inside the user-defined function.

For the second the method results were very promising:

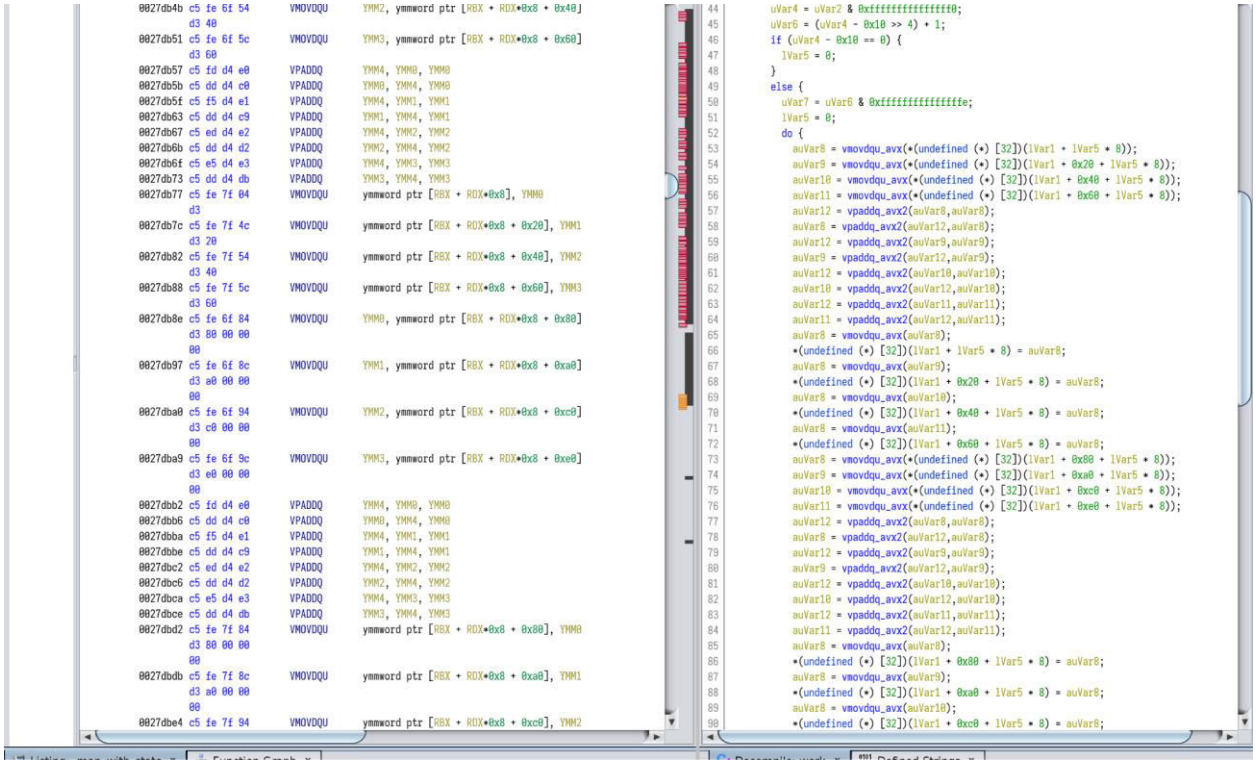

Fig. 4: Monomorphised disassembly

We found that each map call with different Mapper types resulted in a totally separate assembly for work functions(a function where the partitions are iterated over and mapped). Since rustc knew the specific type that the user used to create this specific MapRDD it was able to not just inline but also **do SIMD and unrolling optimizations as well for each of them!** These are the kind of optimizations that are impossible to achieve when user-defined closures are run as black boxes in scala.

Due to the tradeoff of user-friendliness and performance we decided to keep both of the APIs supported in our code and user can decide which one to use based on their needs. Second API also allows the user to attach state from master to worker inside the types.

General Project Architecture

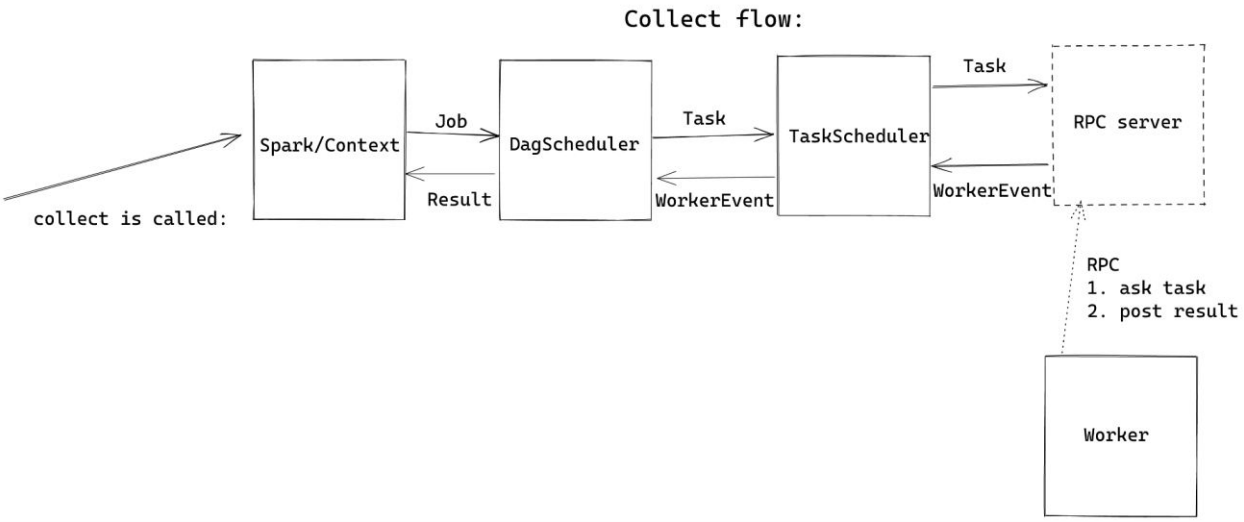

Fig. 5: Flow of a single Job in r2d2

Our general architecture is shown above. We have 4 “green” tokio(rust async runtime library) threads that are talking to each other over channels. When a user calls collect function it ends up submitting a job to the dag scheduler. The thead where the user code resides is now waiting for this jobs result. Dag scheduler divides up the work into tasks(discussed below) and submits them to TaskScheduler. TaskScheduler is simple

for now and is basically a proxy to RPC server. It is meant to future-proof our code for when we want to handle intra task failures.

The worker is always sitting in a loop asking our RPC server for new tasks. It is also responsible for sending events to the master about task completion, bucket receival, etc. These events are piped back to the dag scheduler and in case all the results needed for collect are received user thread receives the final results and returns. Our worker is able to receive multiple tasks and run CPU-bound work on separate OS threads but it also limits concurrent tasks to the thread count.

RDD Representation

We call all the functions of RDDs which are supposed to be called by the executor "work" functions. We'll use this naming convention below.

Narrow RDD representation

Narrow RDDs are fairly simple. They are always dependent on one or zero RDD. Their each partition is also dependent on previous RDD's one partition. That's why Narrow RDD's work function only consists of making certain computations on its input partition data and producing new partition data.

```
trait TypedNarrowRddWork {
      type InputItem: Data;
      type OutputItem: Data;

      fn work(
      &self,
      input_partition: Option<Vec<Self::InputItem>>,
      partition_id: usize,
      ) -> Vec<Self::OutputItem>;
}
```

Wide RDD representation

Wide RDD's Work functions are divided into three steps. First, two of the steps are computed on the previous stage, while the last step is computed on the next stage. We implemented ShuffleRdd with this API and wrote all our Wide RDD implementations using this ShuffleRdd.

Each shuffle RDDs implements these methods:

```
trait TypedRddWideWork: TypedRdd<Item = (Self::K, Self::C)> {
      type K: Data;
      type V: Data;
      type C: Data;

      fn partition_data(
      &self,
      input_partition: Vec<(Self::K, Self::V)>,
      ) -> Vec<Vec<(Self::K, Self::V)>>;
      fn aggregate_inside_bucket(
      &self,
      bucket_data: Vec<(Self::K, Self::V)>,
      ) -> Vec<(Self::K, Self::C)>;
      fn aggregate_buckets(
      &self,
      buckets_aggr_data: Vec<Vec<(Self::K, Self::C)>>,
      ) -> Vec<(Self::K, Self::C)>;
}
```

But it turns out that every single wide transformation can be represented using a single RDD which we call shuffle. It uses a partitioner and aggregator to know how to partition the data inside the input partition, and then uses an aggregator to represent any kind of reducing work(e.g. sum, groupby) the user wants to do.

Users can always supply us with their own implementation of aggregator functions. And most of our own wide operations(partition_by, group_by, sum_by_key, join…) boil down to pre-implemented Aggregators and narrow operations.

```
// () -> Acc
// (V, Acc) -> Acc
// (Acc, Acc) -> Acc
pub trait Aggregator: Data {
```

```
    type Value: Data;
    type Combiner: Data;

    fn create_combiner(&self) -> Self::Combiner;
    fn merge_value(&self, value: Self::Value, combiner:
Self::Combiner) -> Self::Combiner;
    fn merge_combiners(
    &self,
    combiner1: Self::Combiner,
    combiner2: Self::Combiner,
    ) -> Self::Combiner;
}
```

DAG Scheduler

We decided to mostly follow Spark's choices in terms of how we split up the work inside the computation graph.

Stages

Stages are sets of tasks that compute intermediate results in jobs, where each task computes the same function on partitions of the same RDD. Stages are separated at shuffle boundaries. In our project Stages are just an abstract concept. We don't have an explicit Stage type (struct or some data structure) representing a Stage.

Task's execution lifecycle

We define tasks as a sequence of narrow RDDs on a single partition until the wide RDD is encountered. Tasks are sent to the RPC server by Task Scheduler. First RDD of each task is either the very first RDD of a job(e.g. collect) or a wide RDD. Each Task can be uniquely identified using two numbers: `narrow_partition_id, wide_rdd_id.` `wide_rdd_id` corresponds to the next task's first wide RDD.

`narrow_partition_id` corresponds to the partition id in the narrow RDD right before the wide RDD.

The sequence of narrow RDD work function calls is done on the same worker. At the end of the task, wide RDD divides processed data to separate buckets. The Number of buckets is the same as the number of partitions of wide RDD. Wide RDD has a partition function often passed by the user. It is the function responsible for dividing the final data of the task into buckets i. e. repartitioning. Next, the wide RDD aggregates each bucket separately, parallelizing bucket aggregation and additionally compressing the size of the sending bucket. These steps involve the first two work functions defined above for Wide RDDs.

These buckets are then sent over the network to the target worker designated by Dag Scheduler. Dag Scheduler decides the target worker for the task before sending this task to Task Scheduler.

One key difference between our project and Spark is that our Dag Scheduler doesn't wait for the stage to finish to send the next Task to the worker. If the next Task has gotten all its buckets, it is free to start computing without waiting for other Tasks of the same stage to get their buckets of data.

Once a single wide partition gets all of its buckets, Task's first RDD starts executing the remaining function from our aggregation triad. This function aggregates received buckets' data together.

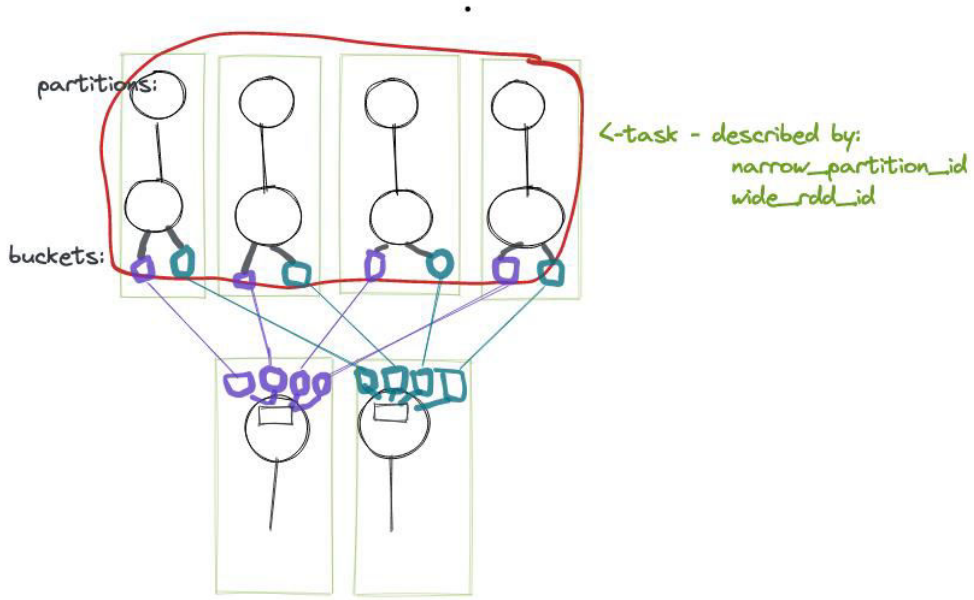

Fig. 6: Bucket visualization

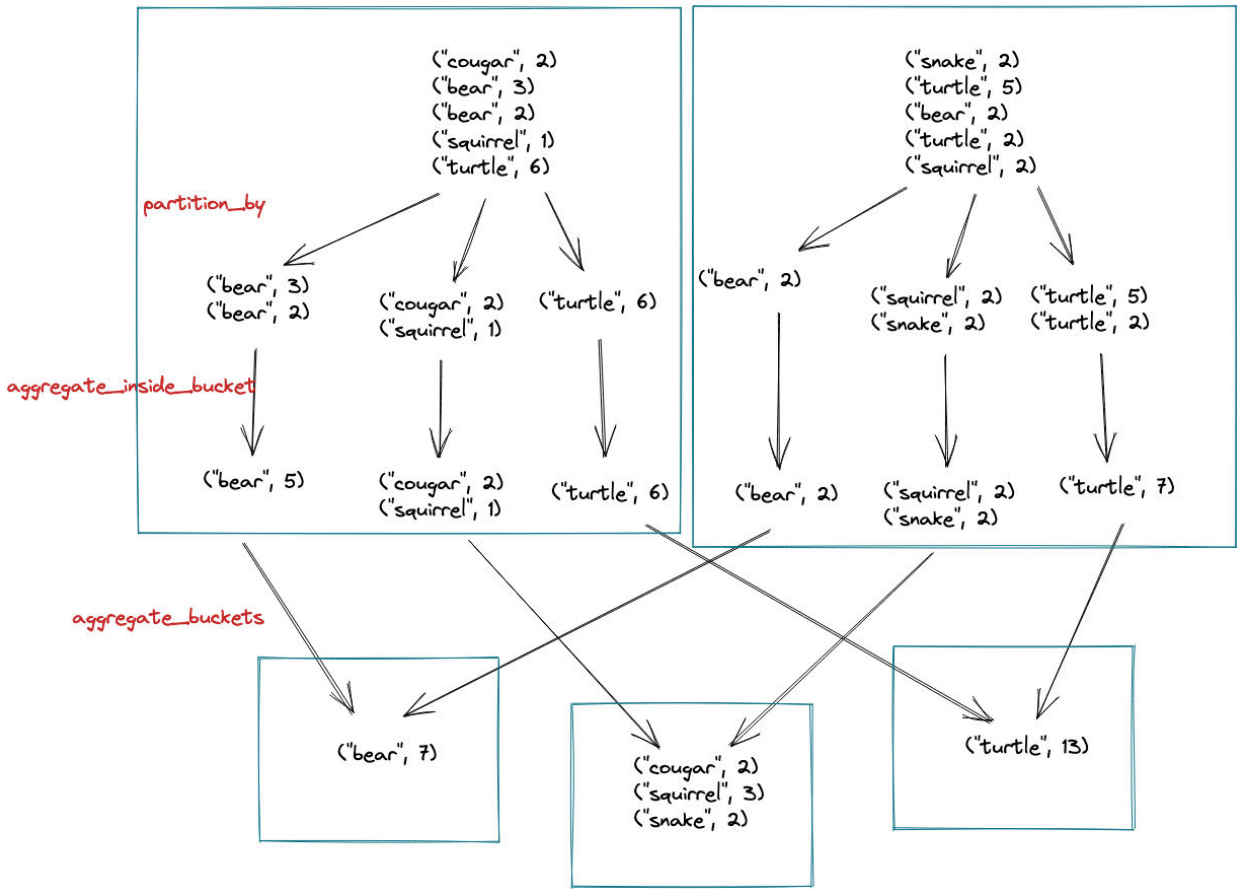

Fig. 7: Three steps of wide transformation

Assigning partitions to workers

When our Dag Scheduler receives a new job that contains a new graph first it figures out which partitions need to be assigned to the same workers. This is done using the DSU algorithm. Right away we try to put all the partitions which depend on each other using narrow dependency

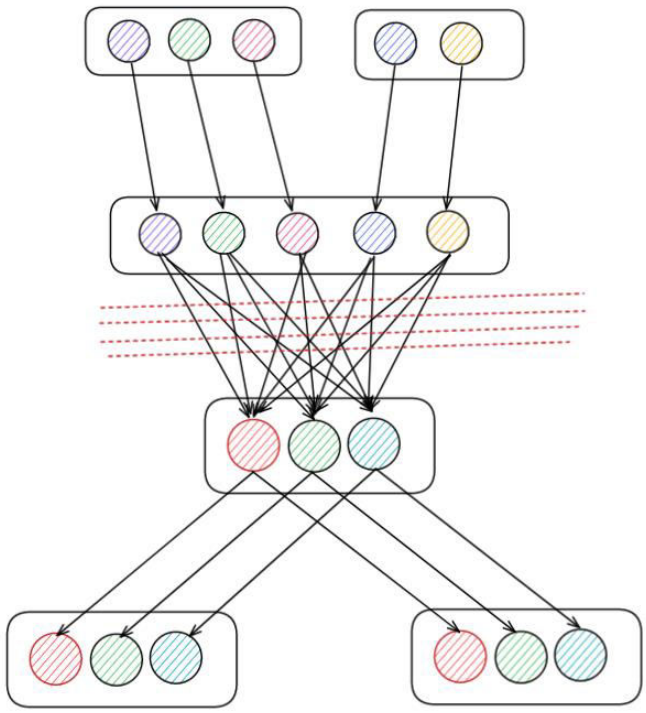

Fig. 8: DSU usage visualization

Communication between nodes

Communication between worker and master servers is done through RPC calls using Tonic library which provides gRPC over HTTP/2 implementation focused on high performance.

Worker - Master communication:

Worker-Master is a one-way communication: only workers call RPCs on master. For that, master runs gRPC server which provides two main procedures:

```
fn get_task(worker_id) -> serialized_task
fn post_task_result(serialized_result) -> ()
```

Worker:

On a high level, the worker program at first connects to the master. Then calls `master.get_task(worker_id)` until the shutdown action.

Based on task, it either receives:

- new graph, stores it and updates master that new graph is stored
- work task, deserializes it, computes result and calls `master.post_task_result(serialized_result)`
- wait message, and waits for other tasks
- shutdown message and program exits

In itself when a task is received, it can be either narrow or wide. When the former is the case, the worker spawns a new computing OS thread which actually runs RDD result computation in parallel to other tasks and when the task is finished notifies the master that task is finished. If the Wide task is received it waits only for the necessary RDD buckets and then resolves the RDD. To optimally run RDD tasks we limit the number of actively running tasks with the number of cpus on the worker machine, since, as mentioned above, all tasks are run in parallel by operating system level thread.

Master:

When the master node starts it runs a master gRPC server which responds to worker's procedure calls. Master service stores a state for each worker. Which is a thread safe `multi-producer-multi-consumer` channel's receiving end, used in `fn get_task(worker_id)` to receive work sent from task scheduler. Each worker has a separate receiving end so they don't try to acquire the same lock for getting the task. This design, in result, contributes to optimal concurrency. Master service also stores task schedulers mpmc's sending end, which is used in fn `post_task_result(serialized_result)` to pass the computed results from workers RPC to the task scheduler.

Worker - Worker communication

When a worker runs a wide task it then connects to appropriate worker's and sends the bucket via TCP. On the other hand, when a worker node starts, it spawns a new tokio asynchronous thread, which until the shutdown, listens to TCP port to receive partition buckets from other workers. These buckets, as described above, are from wide operation (shuffle). These are then stored and used to run later tasks. When the worker fully receives the bucket it updates the master about the event by calling `master.post_task_results(GraphReceived)`.

Evaluation

Task	Change tracking	Wordcount	Wordcount	Many transformations	Sort
Data per partition	77MB	62MB	6 MB	77 MB	200 MB
Data size	1450MB	620MB	600 MB	1540 MB	2000 MB
Number of partitions	20	10	100	20	10
Execution time (sec): r2d2	68.8	5.1	6.7	16	10.9
Execution time (sec): Apache spark version 3.3.0	74.6	8.3	13	16	133.2
Improvement factor	1.08x	**1.62x**	**1.94x**	1.00x	**11.2x**

Fig. 9: Performance comparisons of spark and r2d2 on different tasks

Evaluated Task Descriptions

Change tracking:

Similar process is frequently used in industry when there is a need to batch process user data and record only changes. Task consists of multiple steps, at first read partitioned data from two sources (from different timeline) where each partition file is csv format containing records of location data (e.g. id, name, surname, age, city). Then we deserialize input files to custom struct. Then join these two RDDs by row.id as key. Finally we save output to disk, if location between initial two sources was changed we save both rows (location data), otherwise we write just one.

WordCount:

Another widely used benchmarking technique is, which involves reading & deserializing large amounts of partitioned data of words. Than calling spark.map(rdd, |word| (word, 1)) then using word as a key we call spark.sum_by_key(rdd) to generate the number of occurrences for each word in the input.

Many Transformations:

This benchmark demonstrates spark's advantage to MapReduce by implementing a CPU heavy iterative algorithm.

Sort:

Ubiquitously used as an example for benchmarking. Which involves sorting a large list of numbers in a distributed manner. We implemented sampling sort, where from each worker's local data (sublist) we take a sample of some size. On master, we then sort sampled numbers and narrow them down to evenly distributed N samples. Then we repartition data based on sampled numbers and, on each worker, all the numbers end up in some range of the bucket. Finally, on each worker we sort the partition. Thus ending up with fully sorted data. Our sort is basically identical to sparks implementation of sort. We're happy to see such a big performance increase but will need to investigate deeper the root cause of this change in performance.

References:

[1] Matei Zaharia et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”, https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

[2] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

[3] Aleksandar Prokopec et al. “Making Collection Operations Optimal with Aggressive JIT Compilation”, https://www.davidleopoldseder.com/publications/graal_collections_authorversion.pdf

[4] “vega”, https://github.com/rajasekarv/vega

[5] “Performance Analysis of a CPU-Intensive Workload in Apache Spark”, https://db-blog.web.cern.ch/blog/luca-canali/2017-09-performance-analysis-cpu-intensive-workload-apache-spark

[6] “serde_traitobject”, https://github.com/alecmocatta/serde_traitobject

[7] Heather Miller, "Running Spark on a Cluster: The Basics", https://heather.miller.am/blog/launching-a-spark-cluster-part-1.html

Appendix

Main repository: https://github.com/freeuni-oxidizers/r2d2
Github “organization”: https://github.com/freeuni-oxidizers/