The ICADD algorithm

Assumptions

Before explaining the rescaling algorithm, we must first lay out some assumptions. You will see, that these are very reasonable, and probably already hold true in any streaming system you are using today.

- All **distributable** state an operator may hold, is partitioned by some set of keys `{K}`.
- All state for a given key `K` is located at one and only one worker.
- The worker where state for a key `K` is localized is determined by some **deterministic** function `F(K, W) = Worker` where `W` is the set of available workers in the current cluster configuration.

It therefore follows, that if the amount of available workers is reconfigurable (a “resize”), there must be a set `W'` which contains all available workers for the new configuration.

It also follows, there must be a distribution function `F(K, W') == WorkerId_X` which determinse where the state for `K` is located.

For brevity, we will shorten `F(K, W)` to `F` and `F'(K, W')` to `F'`.

Keying vs Distributing

We asserted before, that there is a function, which distributes messages by key. For there to be a key, there must also be an operator which turns unkeyed messages into keyed message.

Since all distributable state, must be partitioned by key, and the partitioning happens in the distributor operator, it follows that there **can not be** any stateful operator in between the keying and distributing operator.

For now we will actually assume, that there is no other operator at all between key and distribute. Therefore **keying IS distributing**.

Redistributing state

When the cluster size changes, we must possibly redistribute state. We will see how we can achieve this via the follwing example. Consider this extremely simple dataflow:

```
[distributor] --> [stateful operator]
```

The stateful operator may hold state, which may for the new configuration need to be located at another worker. In the old configuration, the distributor determined the keys of this state by applying `F`. In the new configuration, this will be done by applying `F'`.

With this knowledge we can say there are two sets of keys:

- `S1` which contains all keys for which `F(K) == Local`
- `S2` containing all keys for which `F'(K) != Local`

The intersection of these two sets, `S1 ∩ S2`, contains all keys we will need to redistribute. This presents us with an issue though: The keyspace i.e. the domain of the key type is potentially very large or possibly infinite. We can not feasibly materialize these sets in their entirety. However, we will see how the ICADD algorithm works around this problem.

What we want to achieve

We need to understand now, that when receiving a message, a distributor has two possible choices. It can either, send the message to a remote worker, or pass it downstream locally.

Under the old cluster configuration (i.e. size) all distributors distribute a message with key `K` via these rules

1. `If (F(K) == Local)`
    - send the message downstream
2. `If (F(K) != Local)`
    - send the message to the remote worker determined by `F`

Under the new cluster configuration, we will want to instead apply these rules:

1. `If (F'(K) == Local)`
    - send the message downstream
2. `If (F'(K) != Local)`
    - send the message to the remote worker determined by `F'`

We will see, how we can make this transition, while maintaining state integrity and causal message ordering.

Applying ICADD

ICADD is an abbreviation for “Interrogate - Collect - Aquire - Drop - Done”. These are the four message types of our distribution algorithm.

Let’s first establish a basic structure for our distributor and then go through the algorithm.

Distributor Structure

As we already said, our distributor has a distribution function `F`. We will introduce another component; the Version. The `Version` is essentially just a number, telling us how often the distribution function has changed. We will get more into this later. In Rust, the structure of the distributor could look like this

```
struct Distributor<K> {
  func: Box<dyn Fn(&K, &HashSet<WorkerId>) -> &WorkerId>,
  version: u64
}
```

IMPORTANT: There is a special rule, which applies **always**, that may not make much sense now, but will become important later:

Every time a Distributor sends a message to another worker, it attaches its own version to the message. When a Distributor receives a message, where the attached version number is higher than its own, it passes the message downstream, no matter what.

You will see why this matters shortly.

Interrogation Phase

The “Interrogation” phase begins with the distributor creating three new empty sets called

- `whitelist`
- `hold`
- `finished`

Don’t worry, we will explain their purpose shortly.

Let’s set up some new rules in our distributor to deal with an incoming message with a key `K`.

- Rule 1.1: `If (F(K) == Local) && (F'(K) != Local)`
    - add the key `K` to the set `whitelist`
    - pass the message downstream
- Rule 1.2: `If (F(K) == Local) && (F'(K) == Local)`
    - pass the message downstream
- Rule 2: `If (F(K) != Local)`
    - send the message to the worker determined by `F`

IMPORTANT: It does not matter if the incoming message comes from the local upstream or a remote worker, we will apply the same rules to both.

At the moment the sets of `hold` and `finished`, remain empty while `whitelist` gets filled with incoming keys (as per rule 1.1). We will see how the others fill in a moment.

The distributor now sends a message `Interrogate` downstream. When an operator receives an instance of `Interrogate` it calls the method `add_keys([K])` on that instance with all keys for which it holds state. This call adds those keys to the distributor's `whitelist`. Once all operators have processed the `Interrogate` message, the distributor has full knowledge of all keys for which their exists local downstream state because

- It knows of all state for keys received in the past, as those are the keys which the operators have added via `add_keys`.
- It knows of all other keys which went downstream as it has recorded those into `whitelist` itself as per rule 1.1.

As soon as all downstream operators have processed the `Interrogate` message, the distributor can move to the next phase:

Collection Phase

To start this phase, we first increment our Distributor's version number by `1`.

Next, we will remove all keys from `whitelist` where `F'(K) == Local`, since we will not need to redistribute the state for these keys. Now `whitelist` only contains keys, for which we have local state, which we can not keep under the new cluster configuration.

To maintain this property, we must stop accepting new keys into `whitelist`. Therefore, for any incoming message we will apply the following **new** distribution rules, completely discarding the rules from the Interrogation phase:

- Rule 1: `If (F'(K) != Local) && K ∈ whitelist`
  - *We will not have the state under the new configuration, but currently it is still located here*
  - -> pass downstream
- Rule 2: `If (F'(K) != Local) && K ∉ whitelist`
  - *We do not have state for this key and we will not have it under the new configuration*
  - -> distribute via `F'`

These cases are easy, there is however one more possible combination which is more challenging:

- Rule 3: `If (F'(K) == Local) && K ∉ whitelist`

Because this case can arise from two possibilities:

A: The state for `K` is still at another worker

B: `K` is an entirely new key and there exists no state globally for it.

Unfortunately we can not distinguish between these cases, as we do not have knowledge on the global state keyspace. We do however know the following: If case A is true, the state must be at the Worker `F(K) == Worker`. In order to resolve this, we will send the message according to `F(K)`. At the remote worker, there are now three things which may happen:

- A.1 The remote worker has not yet started the ICADD process, it will process the message we sent as normal.
- A.2 The remote worker is in the `Interrogate` phase, it will add `K` to its own whitelist and process the message
- A.3 The remote worker is in the `Collect` phase: One of the first two rules must apply:
  - A.3.1 Rule 1 applies: The remote worker processes the message
  - A.3.2 Rule 2 applies: The remote worker will send us back the message, since its call `F'(K)` will result in our

  worker as a distribution target.

You may already see the trick: Case B (the key is globally new) and case A.3.2 **are the**

same: The remote worker must have had the state, if it existed, but it returned the message, indicating it has no state for this key.

A suboptimal game of ping-pong

There is still a catch to what we just discovered. Remember: For our distribution rules, it does not matter whether a message comes from upstream locally or remotely. This means, the case we just discovered would result in a game of ping-pong: We send the message to the remote (Rule 3), it sends us the message (Rule 2), we send it to the remote (Rule 3), it sends us the message (Rule 2)….

This situation is suboptimal. We must break the cycle.

So here is the trick: When the sender of the message, is the same as the recipient given by `F(K)`, we know, that we are in the ping-pong situation. We also know, that this recipient, has no state for `K`, as otherwise it would not have sent us the message.

Now lets modify rule 3:

Rule 3: `(F'(K) == Local) && K ∉ whitelist`
- if `F(K) == Sender`: -> pass downstream
- else: distribute the message via `F`

This breaks the cycle and saves us from eternal pingpong .

Yet another game of ping pong

If you are particularly eagle-eyed, you may have noticed how rule 2 can also result in a ping-pong situation. Say we apply rule 2: We determine a recipient for the message by calling `F'(K)`. We sent the message to that recipient. The recipient however still uses the old distribution function `F`. It happens to be the case, that the result of `F(K)` is our own WorkerId. So the recipient will just yeet the message back to us… or will it? Now is finally the time to apply that special rule we set up earlier: If you get a message with a version number, higher than your own, **you must take it**. If the recipient, still uses `F`, either because it has not started the migration, or because it is still in the “Interrogate” phase, it will have a lower version number than we do. It **will** accept the message. Nice!

Moving state

We have extensively covered how messages are distributed during the `Collect` phase, but let us now see how we advance to our real goal: moving state.

A tiny recap: The set `whitelist` contains all keys, for which we have local state, that we must not keep.

We need to start draining this set. We start doing so, by applying the following procedure:

1. Choose a key from `whitelist`, remove it and add it to `hold`
2. Buffer all incoming messages for which `K ∈ hold`

This ensures, that our downstream operators, will stop seeing messages with key `K`. Since all state is partitioned by key, they will in turn stop modifying the state for key `K`. Since the state will not be modified anymore, we collect it.

We send the message `Collect(K)` downstream. Upon receiving this message, each operator calls the method `add_state(operator, state)` on this message. This adds their state to a collection.

Once all operators have processed this message, we have all state for `K` in the collection. We will now create a message `Acquire(K)` which contains this collection. We send this message to the worker determined by `F'(K)` and remove `K` from `hold`.

The receiving worker send the message `Acquire(K)` downstream. All operators receiving an `Acquire` message call the method `take(operatorId)` on this message, which will return any state the message may contain for this operator.

Since the recipient now has the state, we drain the queue of all messages we held and send them to the recipient.

There is no way our local downstream operators will see any more messages of key `K`, therefore we now create the message `Drop(K)` and send it downstream. This message tells an operator, that it is now safe to drop any state it may hold for `K`.

We have now achieved the the following situation: `K ∉ whitelist && K ∉ hold`. Therefore, whenever we see `K` in the future, we simply apply Rule 3 and the message will get routed correctly.

Recapping

Given the steps above, we now know, that our full set of distribution rules actually looks like this:

Rule 1.1: `(F'(K) != Local) && K ∈ whitelist`
- We will not have the state under the new configuration, but currently it is still located here
- -> pass downstream

Rule 1.2: `(F'(K) != Local) && K ∈ hold`
- We will not have the state under the new configuration, but currently it is being collected here
- -> buffer the message

Rule 2: `(F'(K) != Local) && K ∉ whitelist && K ∉ hold`
- We do not have state for this key and we will not have it under the new configuration
- -> distribute via `F'`

Rule 3: `(F'(K) == Local)`
- if `F(K) == Sender`: -> pass downstream
- else: distribute the message via `F`

Finishing the migration

We repeat the steps laid out in “Moving state” for all keys from `whitelist`. Eventually we will end up with the sets `whitelist` and `hold` being completely empty.

Given this knowledge, we can see that distribution rules 1.1 and 1.2 can not possibly apply anymore. In fact, the condition `K ∉ whitelist && K ∉ hold` is now true for every `K`. Therefore we can simplify rule 2 to be

Rule 2: `If (F'(K) != Local)` -> distribute via `F'`

and Rule 3 to be

Rule 3: `If (F'(K) == Local)`
- if `F(K) == Sender`: -> pass downstream
- else: distribute the message via `F`

We are almost done, but Rule 3 is still awfully complex. Let’s remember the reason this rule exists: The state for a key, which we will hold under the new configuration, may still be located at another worker. We have given all the state we needed to give, but we do not know if this is true for the other workers. At this point, there is no choice, but to coordinate.

One intermission: **Did you notice, how so far this algorithm required absolutely no coordination between workers? How crazy is that?!?**

We broadcast a message `Done(WorkerId)` with our own `WorkerId` to all other workers, indicating, that we have no more state to give. When a worker receives this message, it adds the contained `WorkerId` to the set `finished`. Recall from the assumptions, that the set `W` is the set of all workers of the old configuration. Therefore, when we observe that the set `W` is equal to the set `finished`, we can conclude, that the entire migration is done.

Now, finally, we can simplify Rule 3. Our full set of rules is now just

- `(F'(K) != Local)` -> distribute via `F'`
- `(F'(K) == Local)` -> pass downstream

At this point we are done! We can drop all sets we created. We can also forget about the function `F`, since it does not show up in any of our distribution rules. This means, we can also forget about the set `W`, since it has been superseded by the set `W'` everywhere.

Appendix

The following sections describe how to deal with some special cases as well as the implementation of ICADD in JetStream.

Keyed Regions

We said, that all state after a distributor is logically partitioned by some function. But what if we have multiple keying steps, like in this execution graph?

```
[distributor] --> [stateful_map] --> [distributor] --> [stateful_map]
```

We will call this area in between the distirbutors a **region**. Given the graph above, we have two regions. Let's call them "A" and "B"

```
[distributor] --> [stateful_map] --> [distributor] --> [stateful_map]
             |_______Region A_______|             |_____Region B_____|
```

This situation works out just fine using the ICAD algorithm, if we adhere to one simple rule: The ICAD messages (Interrogate, Collect, Acquire, Done) do not cross region boundaries. This means, when any of those messages reach the second distributor, it simply discards them.

Dealing with Sources

In a real processing graph, we may have operators other than the distributor generating keys. These are called "sources". This presents us with an issue in the Interrogate phase: When an operator downstream from a distributor creates keys, we will not have them in `whitelist` if they were created after the "Interrogate" message has passed this operator. This could break the assumption we made earlier, that all keys, for which the worker has state are contained in `whitelist`.

Let's take a look at an example: Say we have a datasource, which produces data from a directory. It watches the directory, and whenever a new file is added, it opens that file, reads it row by row, and produces the rows as records.

This source is stateful, because it has to retain the current cursor offset per file. Now remember our very first assumption:

All ***distributable*** *state an operator may hold, is partitioned by some set of keys* `{K}`

If we want to make the state our source is holding distributable, it must be partitioned and in fact, it already is. Ther cursor position is *per file*. We can make this state distributable, by keying it by the file name. For this, we split our source into two: The first source watches the directory and produces the names of files as data. The second source reads the files and produces the content as data. In between these to, we add a distributor, which keys by filename. Our graph now looks like this:

```
[directory watcher] - filenames -> [distribute(by filename)] - (key: filename, value:
()) -> [file reader]
```

Now the state held after the distributor is keyed and all possible keys will pass through the distributor. This means all keys will also be added to `whitelist` in the collect phase.

For fun, lets apply this pattern to a different kind of source. We will use an operator reading Kafka topics as a source. This operator holds state, namely the read offset per partition and topic. So it would be most natural to key this state by a tuple of (partition, topic). We can get the list of topics and partitions per topic from the Kafka broker. So in turn, our stream could look like this

```
KafkaLister: Queries Kafka, outputs a list of (topic, partition) tuples
KafkaReader: Takes (topic, partition) tuples as input and outputs messages from Kafka
```

```
[KafkaLister] - (topic, partition) -> [distribute(by (topic, partition))]  - (key:
(topic, partition), value: ()) -> [KafkaReader]
```

You see, that in both examples we achieved distributable state by

1. Splitting the source into a stateless and a stateful component
2. Keying messages by their value

Dealing with multi-pathing

In an execution graph we may have splits and unions within a keyed region. Since our `Interrogate` and `Collect` messages must reach every operator at a split, we have no choice, but to copy them and send them into every outgoing edge of the split. However when the graph is unioned again, the operators after the union will see multiple `Interrogate` and `Collect` messages, we call this issue "multi-pathing".

This is not an issue though. They can simply call the `add_keys` method on each message. Since `add_keys` adds to the `whitelist` set, invocations of the method following the first invocation will either have no effect, or add all keys which the operator obtained in between processing the multiple `Interrogate` messages, which is exactly what we want anyway! For `Collect` we must simply overwrite the collected state for an operator, if this operator calls `add_state` again, as the subsequent invocation will add a more up-to-date state than the first one.

What about Acquire?

Multi-pathing can also occur for the `Acquire` message. Here the rule to follow is: The first invocation of `take` returns the state `Some(S)`, all following invocations return `None`. Operators must only use the `Some` value to update the state they hold.

Why the first and not subsequent? Because in between receiving the first `Acquire` and another one, there may have been data messages of the relevant key, which could have mutated the state. To maintain causal ordering, we must use the first `Acquire`

Unions of Keyed Regions

TODO

Implementing ICADD

This section is a TODO. Essentially the best way is probably to use reference counting for the ICAD messages. That way the distributor can hold onto one reference and when the count drops to `1` we know all operators have processed all ICAD messages.