Damions Awesome Rescaling Algorithm

The issue with Scaling

If we have a dataflow processing messages, it will often have a routing function, which routes messages to different workers, based on their key. These workers may maintain arbitrary internal state, which is necessary to process the messages correctly. The problem: If we scale, meaning we add or remove workers, our routing function must change. This however leads to future messages being routed to workers, which do not have the right state to accurately process them. Therefore, we need to redistribute the state.

A simple non-solution

Our initial intution may be, to simply inform all of the workers of the rescaling and kindly ask them, to send their state for a specifc key to the “right” worker, if that “right worker” is a different one under the new routing function.

Unfortunately, doing so opens the door to all kinds of un-fun race conditions. Say we have a dataflow

X -> Y

X -> Z

and we would like to remove `Z`, i.e. scale down. When should `X` stop sending messages to `Z`? If we start routing messages to `Y` too early, it may not yet have obtained the necessary state from `Z`. If we keep sending data to `Z` for too long, it will already have transmitted the state for that data to `Y`, but now the new data would change this state again! We could inform `X` about the keys of transmitted states, and ask it to not route messages of those keys to `Z` anymore. This sounds like a reasonable approach, until you consider that communication is asynchronous: In between the state transmission and informing `X`, there may have been more messages. CHAOS!

Being lazy

We could solve these race conditions by being lazy. Let’s say, we stop routing messages to `Z` entirely and instead route everything to `X`. Now whenever `X` encounters a key, for which it does not have the state, it asks `Z` to send it that state and only proceeds once it has received it, i.e. it **lazy-loads** the state. This does solve our correctness issue, but

1. it adds latency, since we need to communicate for every key between `Y` and `Z`
2. if we get a new key, which neither `Y` nor `Z` have seen before, we still need to communicate, since `Y` has no way of knowing, whether the key is “new” or just “new to me”
3. if we don’t get messages for a specific key for a long time, this state will not be transmitted until we do, which would delay our rescaling

All of the above is solvable, but when looking deeper, we did not get rid of race conditions entirey:

In data parallel executions, there will not be a single distributor `X`, but many. Some of them will start rerouting messsages slightly earlier than others. Think about this scenario.

- There are two `X`: `X1` and `X2`
- `X1` stops sending `Z` messages
- `X1` routes a message to `Y`, `Y` gets the state from `Z`
- `X2` routes a message to `Z`
- `X2` only now stops sending messages to `Z`

Again, we have a race condition, since `Z`s state could have changed after being transferred.

How Others Do It

Flink being a nervous wreck simply panics and shuts down, when asked to rescale. To be fair though, this is rather effective, as it can then restart at the new scale and use its recovery mechanisms to load state in the right places.

This kind of “stop-everything” approach is not acceptable though if you have stringent latency requirements.

Megaphone (Timely Dataflow) has the routing operator (in our case `X`) coordinate the transfer of state. This has the advantage, that state can be transferred key by key, which is advantageous for latency and throughput, but requires that `X` actually knows, for which keys `Z` maintains state. Therefore `Z` must inform `X` whenever it drops (or creates) state for a key. If your keyspace is very large, storing and maintaining the information on which worker is responsible for which key, may proof difficult.

A Better Way To Scale

Below we will describe an algorithm, which attempts to solve all the issues elaborated above.

Scaling Down

Consider the dataflow graph

X -> Y

X -> Z

which processes messages of form `(key, value)`. `X` has some function `F` which will distribute messages to `Y` or `Z` based on their `key` and some function `F'` which will distribute messages but never to `Z`.

`Y` and `Z` each have the state for some keys, which is needed for the computation. Lets say the keys are integers, of an unkown domain, and currently the state is distributed like so:

```
Y: 1, 2, 3
Z: 4, 5, 6
```

We now want to remove `Z` while keeping the interruption to the computation minimal. We do this by applying these steps:

1. `X` sends a Message `SHUTDOWN` to `Z`
2. Upon receiving `SHUTDOWN`, `Z` choses one of the keys, it has state for, let’s say `4` and creates two disjunct sets, `whitelist` and `hold` like so:

   ```
   whitelist = {5, 6}
   hold = {4}
   ```

   It then sends those two sets to `X`
3. Upon receiving the sets, `X` changes its behaviour:
   - If for a key `K` the result of `F(K) == Z`, the messages is
     - either forwared to `Z` if $K \in \text{whitelist}$
     - stored if $K \in \text{hold}$
     - distributed using `F'` if $K \notin (\text{whitelist} \cup \text{hold})$
4. `X` now sends `Z` a message `FINAL(4)` containing the key which was added to the `hold` set.
5. Upon receiving `FINAL(4)`, `Z` knows, that it will never see a message of key `4` again, thus it is now save, to package the state `S4`. `Z` packages the state `S4`, choses another key, lets say `5` and sends `X` a message `REDISTRIBUTE(4, S4, 5)`.
6. Upon receiving `REDISTRIBUTE(4, S4, 5)`, `X`
   - calls `F'(4)` to determine a distribution target, in this case `Y`
   - sends `S4` to the distribution target i.e. `Y`
   - removes `4` from `hold`
   - removes `5` from `whitelist`
   - adds `5` to `hold`
   - sends all messages it had stored for `4` to `Y`

- sends `FINAL(5)` to `Z` (this is step 4!)

Steps 4, 5 and 6, repeat until `Z` runs out of keys, which will result in the message `REDISTRIBUTE(6, S6, None)` being sent to `X`. After processing this message on `X`,
- `whitelist` and `hold` will be empty
- `Y` will have received all state from `Z`
- `Z` will never again receive any more data (because `whitelist`) is empty

At this point, `Z` can be safely shut down, `X` can use `F'` for all future messages and delete `whitelist` and `hold`.

What if `Z` is comprised of multiple operators?
Good question, Damion! Let say Z is made up of two operators, which store states for differing sets of keys:

```
Z1: 4, 5
Z2: 5, 6
```

`Z1` **must forward** the `FINAL(N)` message if it has state for `N`, but create the `REDISTRIBUTE(N, SN, M)` message, for example `REDISTRIBUTE(4, S4, 5)`.

If `Z2` sees a message REDISTRIBUTE(N, SN, M) it either:
- has state for `N` (here this would be `5`), it then adds its state to the message, so that we get `REDISTRIBUTE(N, {Z1: SN, Z2: SN}, M)`
- has no state for `N`, in which case it will forward the message unchanged

Thus `Z1` will drain its state until it emits the message `REDISTRIBUTE(N, SN, None)`, which would here be `REDISTRIBUTE(5, S5, None)`

If `Z2` sees a message `REDISTRIBUTE(5, S5, None)` it:
- adds any of its own state `SN`
- replaces the `None` with a key for which it still has state, if any.

If `Z1` gets a message `FINAL(N)`, but has no state for `N` (here this would be key `6`), it must simply forward this `FINAL` message unchanged.

What if there are multiple `X`?
To account multiple distributors, `Z` must delay reacting to `SHUTDOWN` and `FINAL(N)` until it has received thos from all `X`s. It must in turn also send the `REDISTRIBUTE` messages and sets to all `X`. From the perspective of `X` nothing changes.

Scaling Up

Consider we would want to scale up, and add the node `U`. In this case `F(K)` will be a function that distributes to `Y` or `Z` and `F'(K)` a function which distributes to `U`, `Y` or `Z`. To scale up,

1. `X` sends the message `SCALEUP(*F')`, containing a pointer to `F'` to `Y` and `Z`. **(in the follwing we will look at the process on `Y`, but it is the same on `Z`)**
2. For a key, for which they have state, `Y` executes `F'(K)`.
   - If `F'(K) == Y`,repeat this step with a different key, until all keys have been checked
   - If `F'(K) != Y`, create two disjunct sets, `whitelist` and `hold`, where `hold` contains `K` and `whitelist` contains all other keys, for which `Y` has state and where `F'(K) != Y`. Lets say `F'(4) == U` and `F'(5) == U`:

     ```
     whitelist = {5}
     hold = {4}
     ```
   - If there is no key for which `F'(K) != Y`, create `hold` and `whitelist` as empty sets.

   `Y` then sends those two sets to `X`
3. Upon receiving the sets, `X` changes its behaviour:

- If for a key `K` the result of `F'(K) == U`, the messages is
  - either forwared to `Y` if $K \in \text{whitelist}$
  - stored if $K \in \text{hold}$
  - forwared to `U` if $K \notin (\text{whitelist} \cup \text{hold})$

4. `X` now sends `Y` a message `FINAL(4)` containing the key which was added to the `hold` set.
5. Upon receiving `FINAL(4)`, `Y` knows, that it will never see a message of key `4` again, thus it is now save, to package the state `S4`. `Y` packages the state `S4`, choses another key for which `F'(K) != Y`, lets say `5` and sends `X` a message `REDISTRIBUTE(4, S4, 5)`.
6. Upon receiving `REDISTRIBUTE(4, S4, 5)`, `X`
   - calls `F'(4)` to determine a distribution target, in this case `U`
   - sends `S4` to the distribution target i.e. `U`
   - removes `4` from `hold`
   - removes `5` from whitelist
   - adds `5` to `hold`
   - sends all messages it had stored for `4` to `U`
   - sends `FINAL(5)`to `Y` (this is step 4!)

Steps 4, 5 and 6, repeat until `Y` runs out of keys requiring redistribution, which will result in the message `REDISTRIBUTE(N, NS, None)` being sent to `X`. After processing this message on `X`,
- `whitelist` and `hold` will be empty
- `U` will have received all state it needs from `Y`

At this point, `X` can use `F'` for all future messages and delete `whitelist` and `hold`.

This step can be run for both `Y` and `Z` in parallel. In that case, there are multiple `whitelist` and `hold` sets. `X` can only switch to using `F'` permanently, once it has as many empty `hold` sets, as outgoing edges.

Speeding up Rescaling

For some applications, the process of redistributing the state of each key individually, might take too long. In this case, instead of processing each key one by one, with each message multiple keys could be redistributed using the same process. Packaging multiple states will possibly result in larger messages and more time spent on serialization. Therefore if we have a number `N` of keys being redistributed at once, the relationship that follows is
- small N, small impact on throughput/latency, longer rescale time
- large N, shorter rescale time, higher impact on throughput/latency

Latency and throughput

If scaling key by key, latency and throughput will only be negatively affected for the key currently being rescaled. Assuming the overhead of set look-ups is neglible (which it is most of the time), latency and throughput for other keys is unaffected during rescaling.