Expose State Backend Interface for UDAGG

[1. Background](#_u5sl59b5ofsf)

[2. Design](#_xuet7sze70wz)

[Public Interfaces](#_w3ob5xgbqrnj)

[Usage](#_kwrrfkmpajkc)

[Example](#_39s1pvppv8ex)

[Implementation](#_mqci7mi4s2rj)

[Prototype](#_x93wcce4jnp)

[Current Limit](#_wrsw9t8opcky)

[Implementation Plan](#_qecjxim5e8gu)

## 1. Background

Currently UDAGG functions can not register custom state, the intermediate result can only be stored in memory inside user defined accumulator. This will lead to two problems:

* Every time when processing an element, the accumulator will be deserialized from state backend as a whole before calculation, and serialized to state backend as a whole after calculation. In some cases (such as distinct count & retract min/max), most parts of the accumulator is not necessary to be deserialized. So the state I/O can be reduced.
* The intermediate result are easy to grow exceed allocated memory limit and triggers the OOM.

So it's necessary to provide users with a convenient and efficient way to register custom state within the UDAGG. Meanwhile, in some scenes, such as batch table and windowed group, state backend access is not supported. We do not expect users to implement two sets of UDAGG, one for state backend, another for heap. This design document proposes a new interface to use efficient data structures in UDAGGs which may backed by state backend or heap objects. As a result, users do not need to learn the concepts of states.

## 2. Design

### Public Interfaces

In the first version, we plan to introduce two class: MapView and ListView.

|  |
| --- |
| public class MapView<UK, UV> extends DataView {  UV get(UK key) throws Exception {  throws new UnsupportedOperationException(“...”);  }  void put(UK key, UV value) throws Exception {...}  void putAll(Map<UK, UV> map) throws Exception {...}  void remove(UK key) throws Exception {...}  boolean contains(UK key) throws Exception {...}  Iterable<Map.Entry<UK, UV>> entries() throws Exception {..}  Iterable<UK> keys() throws Exception {...}  Iterable<UV> values() throws Exception {...}  Iterator<Map.Entry<UK, UV>> iterator() throws Exception {...}  void clear() throws Exception {...} }  public final class ListView<T> extends DataView {  Iterable<T> get() throws Exception {  throws new UnsupportedOperationException(“...”);  }  void add(T value) throws Exception {...}  void clear() throws Exception {...}  } |

We declare the MapView and ListView as a class not an interface is because we hope users can use it the same as HashMap/ArrayList, including the creation.

The implementation of MapView includes HeapMapView and StateMapView. The HeapMapView is an implementation that delegate all calls to a Java HashMap, and the StateMapView will delegate all calls to Flink MapState. ListView is similar to MapView. Choosing which implementation to use is depend on the query, and is chosen by Flink automatically.

In case of non-windowed group and OVER window, we will use StateMapView, because we implement them using ProcessFunction and can get the RuntimeContext to access the States. **In case of windowed group, WindowedStream.aggregate(...) doesn’t support RichAggregateFunction which means we can’t get the RuntimeContext to access the States**, so we have to use HeapMapView in this case. In batch envrionement, no state backend provided, so we have to use HeapMapView as well.

### Usage

MapView and ListView can only used in POJO classes currently (can support MapView/ListView in Java and Scala Tuples and case classes when we support MapViewTypeInfo&ListViewTypeInfo). To use the MapView, the only thing need to do is to declare one or more fields whose type is MapView or ListView in user’s ACC class (must be a POJO). And can use it in UDAGG just like a HashMap or a ArrayList, but the MapView and ListView interface differs from the HashMap and ArrayList interfaces.

### Example

The following is an example

|  |
| --- |
| **public** **class** **MyAcc** {  **public** MapView<String, Long> directory = new MapView<>();  **public** **int** count = 0; }  **public** **class** **DistinctCount** **extends** **AggregateFunction**<**Long**, **MyAcc**> {  **public** MyACC **createAccumulator**() {  **return** **new** MyAcc();  }   **public** **void** **accumulate**(MyAcc acc, String id) **throws** Exception {  **if** (acc.directory.contains(id)) {  acc.count++;  } **else** {  acc.directory.put(id, 1);  }  }   **public** Long **getValue**(MyAcc acc) {  **return** acc.count;  } }  *// usage* val distinct = **new** DistinctCount  *// non-windowed group will use state backend MapView* val reuslt = table.groupBy('a).select('a, distinct('c))  *// windowed group will use heap HashMap MapView* val result = table  .window(... as 'w)  .groupBy('a, 'w)  .select('w.start, distinct('c)) |

### Implementation

**Where to inject the actual MapView?**

The answer is code generation of GeneratedAggregations. If it is a state backend, we will inject the StateMapView into user’s ACC before every accumulate, retract, merge, getValue, reset is called. If it is a heap backend, we will inject the HeapMapView into user’s ACC only after the createAccumulator is called, because HeapMapView is de/serialized with the ACC.

**How to create StateDescriptor?**

Before generate the GeneratedAggregations, we need the information about MapView/ListView in order to create the StateDescriptor. We analyze ACC type (maybe in AggregateUtils#transformToAggregateFunctions) to extract which fields are MapView/ListView and what the key & value TypeInformation of MapView and what what the element TypeInformation of ListView. Also generate the name of state descriptor, we using the schema of “agg\_index$field\_name”. In the above example, we will create the name “1$directory” as the state name. This name is unique in an operator.

**De/serialization of ACC?**

In case of state backend, the ACC is only de/serialize the non-MapView/ListView fields. In this case, we will create a new PojoType which removes the MapView/ListView fields as the new ACC TypeInformation. In case of heap backend, the MapView/ListView is de/serialized with the ACC, just like a Java HashMap. So we do not need to change the TypeInfromation of ACC. **An improvement here is to create a custom MapViewTypeSerializer and ListViewTypeSerializer just like MapTypeSerializer and ListTypeSerializer to improve the performance of de/serialization of MapView & ListView.**

### Prototype

Yes, we have created a prototype for this design. You can checkout this branch: <https://github.com/kaibozhou/flink/pull/7/files>

### Current Limit

* ACC is a container for MapView/ListView. MapView/ListView can only be declared in ACC, can’t take the place of ACC type. Which means the following code is illegal. We can support it in the future.

|  |
| --- |
| public class DistinctCount  extends AggregateFunction<Long, MapView<String, Long>> |

* MapView/ListView can only be declared in a POJO class, not in Scala case class or tuple class or Flink Tuple class.

### Implementation Plan

1. **Implementation of MapView and ListView to support state access for UDAGG**
2. Support getAccumulatorType for MapView and ListView
3. Refactor MaxWithRetractAccumulator and MinWithRetractAccumulator using the DataView
4. Support MapView and ListView in Java and Scala Tuples and CaseClasses or as the accumulator of AggregateFunction itself. ~~Support MapView and ListView can be as a ACC type.~~
5. Support RichAggregateFunction for window aggregate (affect DataStream level)
6. Use StateMapView/StateListView for windowed group aggregates