FLIP-424: Asynchronous State APIs

Discussion thread	https://lists.apache.org/thread/ct8smn6g9y0b8730z7rp9zfpnwmj8vf0 https://lists.apache.org/thread/nmd9qd0k8l94ygcfgllxms49wmtz1864
Vote thread	https://lists.apache.org/thread/tplyf17n3409l605bo9promf4o8tvl2j
JIRA	FLINK-34974 - FLIP-424: Asynchronous State APIs IN PROGRESS
Release	2.0

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

This is a sub-FLIP for the disaggregated state management and its related work, please read the FLIP-423 first to know the whole story.

Motivation

Currently, in Flink, each task processes elements sequentially within a single thread, which includes accessing the state. When employing a local disk-based state backend, accessing the state often entails I/O operations, which are markedly slower than CPU computations, making the stateful operator easily become a bottleneck of the whole job. For several years, users have frequently sought advice on performance issues related to stateful operators, particularly in jobs that manage a large state. I/O latency is the primary contributor to these issues, and it also prevents the task thread from fully exploiting CPU resources. Scaling out by increasing parallelism can mitigate the issue and enhance overall throughput, yet this approach demands additional resources, notably CPU and memory —— resources that are not the actual bottleneck.

To maximize I/O capacity utilization and enhance the use of pre-allocated computational resources, this FLIP proposes the introduction of asynchronous state APIs. These APIs permit state access to be executed in threads separate from the task thread, returning the result when available. Consequently, the task thread can process another element while awaiting multiple pending state results. This enables concurrent processing of multiple records, ensuring that the latency of individual I/O operations no longer has a direct impact on job performance. This approach is particularly advantageous in scenarios where I/O bandwidth is underutilized and I/O latency is the limiting factor. The Disaggregated Storage Architecture, as discussed in FLIP-423, is a prime example of a scenario characterized by abundant and easily scalable I/O bandwidth coupled with higher I/O latency. The asynchronous state APIs hold great promise for significantly enhancing Flink's performance when dealing with disaggregated state.

Public Interfaces

All asynchronous state APIs employ the newly introduced Future class as their return type. We choose a new Future class instead of CompletableFuture for the following reasons:

- CompletableFuture offers robust features, including the ability for users to designate the executor for a callback. In contrast, we advocate for executing callbacks within the task thread (as outlined in FLIP-425) and aim to abstract these intricacies away from the user.
- Prior to executing a callback, a context switch is required, a process that will be seamlessly integrated within the implementation of the new Future class.

The challenges for asynchronous execution will be discussed in FLIP-425. While in this FLIP, we focus on the definition of the APIs.

A new set of APIs and interface classes are proposed, **offering both synchronous API and asynchronous counterpart** for state access. For async interfaces that yield a collection of elements, an iterator is provided to facilitate asynchronous loading of the elements. The new API set is named "State API V2" residing under the package path 'org.apache.flink.api.common.state.v2' and the `flink-core-api` module. They work closely with the Datastream API V2 and are annotated with @Experimental in first few versions, and will be promoted to @PublicEvolving alongside the DataStream API V2. **The synchronous APIs are basically identical with the original ones**, thus we only discuss the new asynchronous part only.

StateFuture & FutureUtils

`StateFuture` provides selected subset of functions in `CompletableFuture` . The implementation will encapsulate a `CompletableFuture` to do the actual work.

StateFuture

```
@PublicEvolving
public interface StateFuture<T> {
    /**
     * Returns a new StateFuture that, when this future completes
     * normally, is executed with this future's result as the argument
     * to the supplied function.
     *
     * @param fn the function to use to compute the value of
     * the returned StateFuture
     * @param <U> the function's return type
     * @return the new StateFuture
     */
    <U> StateFuture<U> thenApply(Function<? super T,? extends U> fn);

    /**
     * Returns a new StateFuture that, when this future completes
     * normally, is executed with this future's result as the argument
     * to the supplied action.
     *
     * @param action the action to perform before completing the
     * returned StateFuture
     * @return the new StateFuture
     */
    StateFuture<Void> thenAccept(Consumer<? super T> action);

    /**
     * Returns a new future that, when this future completes normally,
     * is executed with this future as the argument to the supplied function.
     *
     * @param action the action to perform
     * @return the new StateFuture
     */
    <U> StateFuture<U> thenCompose(
        Function<? super T, ? extends StateFuture<U>> action);


    /**
     * Returns a new StateFuture that, when this and the other
     * given future both complete normally, is executed with the two
     * results as arguments to the supplied function.
     *
     * @param other the other StateFuture
     * @param fn the function to use to compute the value of
     * the returned StateFuture
     * @param <U> the type of the other StateFuture's result
     * @param <V> the function's return type
     * @return the new StateFuture
     */
    <U,V> StateFuture<V> thenCombine
    (StateFuture<? extends U> other,
     BiFunction<? super T,? super U,? extends V> fn);
}
```

And some utilities as follows:

FutureUtils

```
@PublicEvolving
public class StateFutureUtils {
    /**
     * Returns a completed future that does nothing and return null.
     */
    public static <V> StateFuture<V> completedVoidFuture() {
        // xxx
    }

    /** Returns a completed future that does nothing and return provided result. */
    public static <V> StateFuture<V> completedFuture(V result) {
        // xxx
    }

     /**
     * Creates a future that is complete once multiple other futures completed. Upon successful
     * completion, the future returns the collection of the futures' results.
     *
     * @param futures The futures that make up the conjunction. No null entries are allowed,
     *        otherwise a IllegalArgumentException will be thrown.
     * @return The StateFuture that completes once all given futures are complete.
     */
    public static <T> StateFuture<Collection<T>> combineAll(
            Collection<? extends StateFuture<? extends T>> futures) {
        // xxx
    }
}
```

StateIterator

Asynchronous iterators allow to iterate over data that comes asynchronously, on-demand.

StateIterator

```
@PublicEvolving
public interface StateIterator<T> {
    /**
     * Async iterate the data and call the callback when data is ready.
     * @param iterating the data action when it is ready. The return is the state future for
     * chaining.
     * @param <U> the type of the inner returned StateFuture's result
     * @return the Future that will trigger when this iterator and all returned state future get its
     * results.
     */
    <U> StateFuture<Collection<U>> onNext(Function<T, StateFuture<? extends U>> iterating);


    /**
     * Async iterate the data and call the callback when data is ready.
     * @param iterating the data action when it is ready
     * @return the Future that will trigger when this iterator ends.
     */
    StateFuture<Void> onNext(Consumer<T> iterating);

    /** Return if this iterator is empty synchronously. */
    boolean isEmpty();
}
```

State

Base class for all state definitions:

State

```
@PublicEvolving
public interface State {

    /** Removes the value mapped under the current key. */
    StateFuture<Void> asyncClear();
}
```

ValueState

ValueState

```
@PublicEvolving
public interface ValueState<T> extends State {
    /**
     * Returns the current value for the state asynchronously. When the state is not partitioned the
     * returned value is the same for all inputs in a given operator instance. If state partitioning
     * is applied, the value returned depends on the current operator input, as the operator maintains
     * an independent state for each partition. When no value was previously set using {@link #asyncUpdate
(Object)},
     * this will return {@code null} asynchronously.
     *
     * @return The {@link StateFuture} that will return the value corresponding to the current input.
     */
     StateFuture<T> asyncValue();

    /**
     * Updates the operator state accessible by {@link #asyncValue()} to the given value. The next time
     * {@link #asyncValue()} is called (for the same state partition) the returned state will represent
     * the updated value. When a partitioned state is updated with {@code null}, the state for the
     * current key will be removed.
     *
     * @param value The new value for the state.
     * @return The {@link StateFuture} that will trigger the callback when update finishes.
     */
     StateFuture<Void> asyncUpdate(T value);
}
```

ListState

ListState

```
@PublicEvolving
public interface ListState<T> extends State {

    /**
     * Returns the current iterator for the state asynchronously. When the state is not partitioned the
     * returned value is the same for all inputs in a given operator instance. If state partitioning
     * is applied, the value returned depends on the current operator input, as the operator maintains
     * an independent state for each partition. This method will never return (@code null).
     *
     * @return The {@link StateFuture} that will return list iterator corresponding to the current input.
    */
    StateFuture<StateIterator<T>> asyncGet();

    /**
     * Updates the operator state accessible by {@link #asyncGet()} by adding the given value to the list
     * of values. The next time {@link #asyncGet()} is called (for the same state partition) the returned
     * state will represent the updated list.
     *
     * <p> Null is not allowed to be passed in
     *
     * @param value The new value for the state.
     * @return The {@link StateFuture} that will trigger the callback when update finishes.
     */
    StateFuture<Void> asyncAdd(T value);

    /**
     * Updates the operator state accessible by {@link #asyncGet()} by updating existing values to the
     * given list of values. The next time {@link #asyncGet()} is called (for the same state partition)
     * the returned state will represent the updated list.
     *
     * <p> Null value passed in or any null value in list is not allowed.
     *
     * @param values The new value list for the state.
     * @return The {@link StateFuture} that will trigger the callback when update finishes.
     */
    StateFuture<Void> asyncUpdate(List<T> values);

    /**
     * Updates the operator state accessible by {@link #asyncGet()} by adding the given values to the list
     * of values. The next time {@link #asyncGet()} is called (for the same state partition) the returned
     * state will represent the updated list.
     *
     * <p> Null value passed in or any null value in list is not allowed.
     *
     * @param values The new values to add for the state.
     * @return The {@link StateFuture} that will trigger the callback when update finishes.
     */
    StateFuture<Void> asyncAddAll(List<T> values);
}
```

MapState

MapState

```
@PublicEvolving
public interface MapState<UK, UV> extends State {

    /**
     * Returns the current value associated with the given key asynchronously. When the state is
     * not partitioned the returned value is the same for all inputs in a given operator instance.
     * If state partitioning is applied, the value returned depends on the current operator input,
     * as the operator maintains an independent state for each partition.
     *
     * @return The {@link StateFuture} that will return value corresponding to the current input.
```

```
 */
StateFuture<UV> asyncGet(UK key);

/**
 * Update the current value associated with the given key asynchronously. When the state is
 * not partitioned the value is updated for all inputs in a given operator instance.
 * If state partitioning is applied, the updated value depends on the current operator input,
 * as the operator maintains an independent state for each partition.
 *
 * @param key The key that will be updated.
 * @param value The new value for the key.
 * @return The {@link StateFuture} that will trigger the callback when update finishes.
 */
StateFuture<Void> asyncPut(UK key, UV value);

/**
 * Update all of the mappings from the given map into the state asynchronously. When the state is
 * not partitioned the value is updated for all inputs in a given operator instance.
 * If state partitioning is applied, the updated mapping depends on the current operator input,
 * as the operator maintains an independent state for each partition.
 *
 * @param map The mappings to be stored in this state.
 * @return The {@link StateFuture} that will trigger the callback when update finishes.
 */
StateFuture<Void> asyncPutAll(Map<UK, UV> map);

/**
 * Delete the mapping of the given key from the state asynchronously. When the state is
 * not partitioned the deleted value is the same for all inputs in a given operator instance.
 * If state partitioning is applied, the value deleted depends on the current operator input,
 * as the operator maintains an independent state for each partition.
 *
 * @param key The key of the mapping.
 * @return The {@link StateFuture} that will trigger the callback when update finishes.
 */
StateFuture<Void> asyncRemove(UK key);

/**
 * Returns whether there exists the given mapping asynchronously. When the state is
 * not partitioned the returned value is the same for all inputs in a given operator instance.
 * If state partitioning is applied, the value returned depends on the current operator input,
 * as the operator maintains an independent state for each partition.
 *
 * @param key The key of the mapping.
 * @return The {@link StateFuture} that will return true if there exists a mapping whose key
 * equals to the given key
 */
StateFuture<Boolean> asyncContains(UK key);

/**
 * Returns the current iterator for all the mappings of this state asynchronously. When the state
 * is not partitioned the returned iterator is the same for all inputs in a given operator instance.
 * If state partitioning is applied, the iterator returned depends on the current operator input,
 * as the operator maintains an independent state for each partition.
 *
 * @return The {@link StateFuture} that will return mapping iterator corresponding to the current input.
*/
StateFuture<StateIterator<Map.Entry<UK, UV>>> asyncEntries();

/**
 * Returns the current iterator for all the keys of this state asynchronously. When the state
 * is not partitioned the returned iterator is the same for all inputs in a given operator instance.
 * If state partitioning is applied, the iterator returned depends on the current operator input,
 * as the operator maintains an independent state for each partition.
 *
 * @return The {@link StateFuture} that will return key iterator corresponding to the current input.
*/
StateFuture<StateIterator<UK>> asyncKeys();

/**
 * Returns the current iterator for all the values of this state asynchronously. When the state
```

```
     * is not partitioned the returned iterator is the same for all inputs in a given operator instance.
     * If state partitioning is applied, the iterator returned depends on the current operator input,
     * as the operator maintains an independent state for each partition.
     *
     * @return The {@link StateFuture} that will return value iterator corresponding to the current input.
     */
    StateFuture<StateIterator<UV>> asyncValues();

    /**
     * Returns whether this state contains no key-value mappings asynchronously. When the state is
     * not partitioned the returned value is the same for all inputs in a given operator instance.
     * If state partitioning is applied, the value returned depends on the current operator input,
     * as the operator maintains an independent state for each partition.
     *
     * @return The {@link StateFuture} that will return true if there is no key-value mapping, otherwise false.
     */
    StateFuture<Boolean> asyncIsEmpty();
}
```

Code Example

Such usage in the following example code in `processElement` is an intermediate step. And as such it will be used for development and PoC purposes only. The long term solution will be discussed later.

To leverage asynchronous state APIs, users should change their code accordingly:

- **For Table API & SQL users**: No change.
- **For DataStream API users** or **Table API & SQL developers**:

Given a case that two streams join. Stream A is inner while Stream B is outer. Both streams have unique key. The join key contains Stream A's unique key but not B's unique key. When there is a record from Stream A coming, the code using old State APIs will be:

Synchronous state API (current)

```
private final ValueState<RowData> aState;

private final MapState<RowData, Tuple2<RowData, Integer>> bState;

private final KeySelector<RowData, RowData> uniqueKeySelector;

public void processElement1(StreamRecord<RowData> element) throws Exception {
    RowData input = element.getValue();
    boolean isAccumulateMsg = RowDataUtil.isAccumulateMsg(input);
    if (isAccumulateMsg) {
        for (Tuple2<RowData, Integer> value : bState.values()) {
            if (joinCondition.apply(input, value.f0)) {
                output(input, value.f0);
                RowData key = uniqueKeySelector.getKey(value.f0);
                bState.put(key, Tuple2.of(value.f0, value.f1 + 1));
            }
        }
        aState.update(input);
    } else {
        // omit retraction
    }
}
```

And the code using new State APIs will be:

New APIs usage

```
private final ValueState<RowData> aState;

private final MapState<RowData, Tuple2<RowData, Integer>> bState;

private final KeySelector<RowData, RowData> uniqueKeySelector;

public void processElement1(StreamRecord<RowData> element) throws Exception {
    RowData input = element.getValue();
    boolean isAccumulateMsg = RowDataUtil.isAccumulateMsg(input);
    if (isAccumulateMsg) {
        bState.asyncValues().thenCompose((iterator) -> {
            return iterator.onHasNext((value) -> {
                // value is Tuple2<RowData, Integer>
                if (joinCondition.apply(input, value.f0)) {
                    output(input, value.f0);
                    RowData key = uniqueKeySelector.getKey(value.f0);
                    return bState.asyncPut(key, Tuple2.of(value.f0, value.f1 + 1));
                } else {
                    return FutureUtils.emptyFuture();
                }
            });
        }).thenAccept((e) -> {
            return aState.asyncUpdate(input);
        });
        // or just:
        // aState.asyncUpdate(input);
    } else {
        // omit retraction
    }
}
```

The state APIs described above are flexible and allowing user to define the execution order as wished. By providing a callback via `thenApply` , `thenCompose` or `thenCombine` of a StateFuture, user could easily assemble their tasks using multiple code segments. The interface and functionality of `StateFuture` mirror those of `CompletableFuture` , which is extensively adopted and proven to be complete and powerful enough.

Internal APIs and implementations

The original state-related internal APIs and implementation are designed based on single-thread execution model, so they may be not suitable for new execution model where multiple state requests are on going. For example, the KeyedStateBackend holds the key context and the internal states hold the namespaces. It is better not to provide these context-related methods in global instances. Thus we will add new interface classes and implementations as needed, and left the old ones untouched. This FLIP will not list them in detail as we cannot predict all situations until we finished our implementations.

Compatibility, Deprecation, and Migration Plan

The newly introduced APIs are designed to provide synchronous and asynchronous APIs. The code path for the new APIs is completely independent from the original one. We strongly recommend that users stick to either the synchronous APIs or asynchronous APIs exclusively, rather than mixing their usage. Although using both the asynchronous and synchronous APIs simultaneously won't cause compatibility issues, it may lead to suboptimal performance. The synchronous API can block the task thread until its execution is complete, potentially resulting in a performance regression when compared to using the asynchronous state calls exclusively. But still, the mixed usage could out-perform the sync APIs usage only, a code example is:

```
bState.asyncValues().thenApply((iterator) -> {
    iterator.onHasNext((value) -> {
         if (joinCondition.apply(input, value.f0)) {
             output(input, value.f0);
             RowData key = uniqueKeySelector.getKey(value.f0);
             bState.put(key, Tuple2.of(value.f0, value.f1 + 1));
         }
    });
});
aState.update(input);
```

While the `update` or `put` is sync state access, the state iteration is executed asynchronously. The update and put are relatively lightweight methods compared with the iteration, so they won't block too much the asynchronous pipeline and the job could perform better than pure synchronous execution. Compared with the example of stream join using pure asynchronous APIs, the mixed usage like this cannot totally compete with the asynchronous one in performance, but it is much more easier to use, especially for those users who are not familiar with writing asynchronous programs and want to migrate their job from Flink 1.x to 2.0. We plan to allow the mixed usage considering user-friendliness, but warn them about the performance concern in runtime.

The original APIs, will be deprecated alongside the DataStream V1 APIs after Flink 2.0, we will discuss this in future.

Test Plan

New UT/ITs will be introduced for asynchronous APIs. New E2E tests of jobs using asynchronous state APIs will also be delivered.

Rejected Alternatives

- **Batch State APIs:** The core idea is to batch the elements first, and user should write code processing a batch of elements and accessing the state with a batch of keys.
    - **Pros**:
        - The state execution is more simpler, no mixed updates and gets should be categorized.
        - Less intermediated memory consumption. There is only one callback and context for a batch of state access.
    - **Cons**:
        - Complicated user code with batch processing.
        - Expose Key Context to user.
        - The state iterator for a batch of keys is hard to understand and difficult to use.