# Interactive Programming with Flink Table API

Jiangjie (Becket) Qin, Ruidong Li, Xianda Ke, Shaoxuan Wang, Xiaowei Jiang

## Motivation

Generally speaking, applications may consist of one or more jobs, and they may want to share the data with others. In Flink, the jobs in the same application are independent and share nothing among themselves. If a Flink application involves several sequential steps, each step (as an independent job) will have to write its intermediate results to an external sink, so that its results can be used by the following step (job) as sources.

Although functionality wise this works, this programming paradigm has a few shortcomings:

1. In order to share a result, a sink must be provided.
2. Complicated applications become inefficient due to large amount of IO on intermediate result.
3. User experience is weakened for users using programing API (SQL users are not victims here because the temporary tables are created by the framework)

It turns out that interactive programming support is critical to the user experience on Flink in batch processing scenarios. The following code gives an example:

```
Table a = ….
Table b = a.select(...).filter(...)
int c = b.max()
int d = b.min() // recompute table b from table a
Table c = b.select(('f1 - min)/(max - min)).filter(...)
c.print() // recompute table b from table a
...
If (b.count() > 10) { // recompute table b from table a
      b.select(UDF1(...)).print()// recompute table b from table a
} else {
      b.select(UDF2(...)).print()
}
```

In the above code, because *b* is not cached, it will be computed from scratch multiple times whenever referred later in the program.

To address the above issues, we propose to add support for “interactive programming” in Flink Table API.

## Proposed Changes

We propose to do the following:

1. Add a pluggable temp table service to cache the temp(intermediate) tables. The temp tables are available to the jobs using the same TableEnvironment. We will implement a default file system based temp table service in the Flink, while providing the pluggable interface which allows to connect other external services (e.g. Hbase, Kafka) to cache/persistent the temp tables.
2. Enhance the table environment to maintain the entities (such as temp tables) shared among jobs in the same table environment.
3. Add new functionalities on table API which allows user to cache/persistent temp(intermediate) tables.

Interface wise, we propose the following changes:

- Add a new cache() / invalidateCache() method to the Table class.

```
/**
 * Cache the table for later reference. If the table has been
 * cached, there is no action taken.
 */
def cache(): Table

/**
 * Release the cache of this table.
 */
def invalidateCache(): Unit
```

- Add a new method to TableEnvironment to allow custom temp table service

```
/**
 * Specify a custom temp table service using the given
 * {@link TableFactory} and {@link TableProperties}. If no custom
 * table service is specified, the default table service will be used.
 */
void setTableService(TableFactory tf,
                     TableProperties properties,
                     TempTableCleanUpCallback cleanUpCallback);
```

- The TempTableCleanUpCallback is defined as following:

```
/**
 * A callback that will be invoked when the temp table service is
 * stopped on session exit. Users who has used external storage such
 * as Kafka, HDFS for their temp table service should implement the
 * clean up logic in this callback, so that the temporary tables will
 * be cleaned up when the Flink session exits.
 */
public interface TempTableCleanUpCallback extends Serializable {
     /**
```

```
     * Clean up the tables with the provided temp table ids.
     */
    void cleanUp(Set<String> tempTableIds);
}
```

With these capabilities, users can write the previous example in the following way:

```
Table a = ….
Table b = a.select(...).filter(...);
b.cache(); // cache the table.
int c = b.max()
int d = b.min() // reuse cached table b
Table c = b.select((‘f1 - min)/(max - min)).filter(...)
c.print() // reuse cached table b
...
If (b.count() > 10) { // reuse cached table b
     b.select(UDF1(...)).print() // reuse cached table b
} else {
     b.select(UDF2(...)).print()
}
```

When *b.cache()* is called, a UUID will be assigned on table *b as its temp table ID*, so that when *b.count()* is executed, the data in table *b* is cached internally by Flink. After that, when *b.select().print()* is invoked, *b* will not be re-computed.

We choose to reuse *TableFactory* as the IO interface for temp table service, so that all the existing connectors and their potential optimizations can be used in temp table services. In addition, TableFactory is also abstract enough to implement the default temp table service.

# Related Future Work

The following work are something that we would like to consider and support in the future.

## Cache the Stream Tables

The above caching mechanism would work as well if *b* is a stream table. After the job is submitted, the cached table b will be emitting results to the temp table service as a cached stream table. Thereafter this cached table can be referred by other streaming jobs.

## Persisted Tables Across Sessions

In this proposal, the cached tables are only available within a Yarn session. In the future, we may extend the temp table to be available across sessions. This involves a few additional questions to think about, such as what if the Table names conflict? Should there be something

like session group so there is an access scope between intra-session and public temp table? What is the lifecycle of the tables available across sessions? We would like to defer these discussions to a future thread.

## Detached Drivers

It is also helpful if users can submit their interactive program as a “driver” program and let Flink cluster harness the entire application. In that case, the client can be detached from the application completely.

## Flink Services

As a byproduct of our thoughts on making the temp table service pluggable, the idea of user defined service came up. More specifically, users can launch a service in Flink by providing a FlinkService implementation. The Flink runtime will configure and launch the user defined services before running the actual user job. And the user jobs can interact with the FlinkServices launched in the Flink environment.

In the context of cached tables, the FlinkService will implement the logic of a temp table service. This allows users to have their own temp table services based on different systems such as HDFS, Kafka, etc. The temp table service is actually more of a caching service and could be used by DataSet/DataStream as well.

User defined service is a very powerful primitive that enables a lot of possibilities in Flink. However, it’s is a pretty big topic and probably should be discussed in its own thread.

# Summary

As a summary, we propose to add interactive programming capability to Flink Tables. To achieve that, we would like to introduce a pluggable temp table service and enhance the Table API and table environment to be more context aware.

# Appendix: API Discussion

In this appendix section, we document all the discussions around API in the community.

# Option1

## API

```
// Do not return anything
void cache();

// Alternatively, return the original table for chaining purpose.
Table cache();

// Physically uncache the table to release resource
void uncache();

// To explicitly ignore cache, not part of this proposal but in the future
Table hint(“ingoreCache”)
```

## Semantic

```
TableEnvironment tEnv = ...
Table t = …
...
// cache the table
t.cache();
…
// Assume some execution physically created the cache.
tEnv.execute()
...
// The optimizer decide whether the cache will be used or not.
t.foo();
...
// Physically release the cache.
t.uncache();
```

## Pros

- Simple and intuitive, users only need to deal one variable of Table class

## Cons

- Side effect: a table may be cached / uncached in a method invocation, while the caller does not know about this.

```
{
```

```
  Table t  = …
   // cache the table
   t.cache();
   ...
   // The cache may be released in foo()
   foo(t);
   ...
   // cache is no longer available
   t.bar()
}


void foo(Table t) {
   // cache the table
   t.cache();
   ….
   // Physically drop the table
   t.uncache();
}
```

## Option 2

### API

```
// Return CachedTable
CachedTable cache();

public interface CachedTable extends Table {
  // Physically uncache the table to release resource
  void uncache();
}
```

### Semantic

```
TableEnvironment tEnv = ...
Table t = …
...
// cache the table
CachedTable cachedT = t.cache();
…
```

```
// Assume some execution physically created the cache.
tEnv.execute()
...
// Use the original DAG.
t.foo();
...
// Use cache
cachedT.bar()
...
// Physically release the cache.
t.uncache();
```

### Pros

- No side effect

### Cons

- Optimizer has no chance to kick in.
- Users have to distinguish between original table / cached table.
- Adding auto cache becomes a backward incompatible change.

## Option 3

### API

```
// Return CachedTable
CacheHandle cache();

public interface CacheHandle {
  // Physically release cache resource
  void release();
}
```

### Semantic

```
TableEnvironment tEnv = ...
Table t = …
...
// cache the table
CacheHandle handle1 = t.cache();
CacheHandle handle2 = t.cache;
```

```
…
// Assume some execution physically created the cache.
tEnv.execute()
...
// Optimizer decide whether cache will be used.
t.foo();
...
// Release the first handle, cache will not be released because handle 2 has not been released
handle1.release();
// Release the second handle, the cache will be released because all the cache handle have
been released.
handle2.release();
```

### Pros

- No side effect
- Users only deal with the variable.
- Easy to add auto caching.

### Cons

- The behavior of t.foo() changes after t.cache(), the concern is that this is considered as “modifies” table *t*, which is against the immutable principle.

## Option 4

### API

```
// Return the original table with a “cache” hint, the original table is not changed.
Table cache();

// Physically release the resource used by table cache.
void invalidateCache();
```

### Semantic

The semantic is different depending on whether auto caching (e.g. cache the table at shuffle boundaries) is enabled.

```
TableEnvironment tEnv = ...
Table t = …
...
// cache the table
Table cachedT = t.cache();
```

```
…
// Assume some execution physically created the cache.
tEnv.execute()
...
// Optimizer decide whether cache will be used.
cachedT.foo();
...
// If auto caching is disabled, the original DAG is used.
// If auto caching is enabled, optimizer decide whether cache will be used, i.e. same as
cachedT.foo()
t.foo();
...
// Physically release the resources used by the table cache.
t.invalidateCache()
// The cache is re-created for table t
t.bar()
cachedT.bar()
```

## Pros

- With a given configuration of AutoCachingEnabled, the behavior of t.foo() won't change after t.cache() is invoked.

## Cons

- The semantic becomes configuration based. (Assuming eventually auto caching will be enabled by default, the default behavior of option 4 is the same as option 1)