FLIP-131: Consolidate the user-facing Dataflow SDKs/APIs (and deprecate the DataSet API)

Discussion thread	https://lists.apache.org/thread/l95wtg4rtp7zoy6x7hy595521jzqcb7s
Vote thread	https://lists.apache.org/thread/j62brmocsdb63nwlr981z3m1dxdl0rk3
JIRA	**FLINK-19153** - FLIP-131: Consolidate the user-facing Dataflow SDKs/APIs (and deprecate the DataSet API) OPEN
Release	

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

Motivation

Flink provides three main SDKs/APIs for writing Dataflow Programs: Table API/SQL, the DataStream API, and the DataSet API. We believe that this is one API too many and propose to deprecate the DataSet API in favor of the Table API/SQL and the DataStream API. Of course, this is easier said than done, so in the following, we will outline why we think that having too many APIs is detrimental to the project and community. We will then describe how we can enhance the Table API/SQL and the DataStream API to subsume the DataSet API's functionality.

In this FLIP, we will not describe all the technical details of how the Table API/SQL and DataStream will be enhanced. The goal is to achieve consensus on the idea of deprecating the DataSet API. There will have to be follow-up FLIPs that describe the necessary changes for the APIs that we maintain.

Why does Flink have three APIs?

The three APIs have developed organically over the lifetime of the project and were designed initially for specific use cases. The DataSet API is Flink's oldest API and supports batch-style execution on bounded data. Some may not remember, but Flink was originally a batch processor. Early on, the community realized its pipeline-based architecture was well suited for stream processing which gave rise to the DataStream API. The latter was developed for unbounded, streaming use cases and has special facilities for dealing with state, event-time, and windowing. As Flink became popular in the analytics space, Table API/SQL was introduced to provide a higher-level, relational API that supports both batch and stream processing.

For the discussion below, it will be helpful to understand the distinguishing features of each API.

DataSet API:

- Only supports bounded sources
- Provides no special support for event-time/windowing
- All-or-nothing output: a job either produces data or it doesn't
- Fine-grained, region-based failover, that you need for large scale batch jobs. Failures in one part of the execution don't necessarily require restarts of the whole topology
- Efficient database-style operators: hash-join, merge-join, sorting/grouping for aggregations that make use of the knowledge that input data is bounded

DataStream API:

- Sources can be both bounded and unbounded
- Special support for working with event-time and windows
- "Incremental" output, based on watermarks or checkpoints
- Checkpointing for failure recovery, which means restarting the whole topology in case one operator fails
- You can execute bounded programs but that's not efficient:
  - pessimistic assumptions, no complete knowledge of input, "you don't know what will come next"
  - for aggregations, need to keep all keys in a "hash map"
  - for event-time handling, we need to keep multiple "windows" open
  - no fine-grained recovery based on blocking, persistent shuffles

Table API/SQL:

- Bounded and unbounded sources
- A declarative API, along with SQL

- Data has a schema which is known a priori, thus allowing for additional optimizations, e.g. deserialising only the needed parts of a record when grouping, working completely on binary data, and whole-query optimization
- The same query/program works on bounded and unbounded sources
- Efficient execution for streaming and batch, meaning that for bounded execution we can use the execution model that the DataSet API uses and for streaming use cases the execution model of the DataStrem API is used. This happens transparently to the user.
- No low-level operator-style API, i.e. no timers, state
- No control over the generated execution DAG query optimizer prevents savepoint compatibility

The question naturally arises "Why did the community not originally extend the DataSet API to handle unbounded/streaming workloads instead of adding the DataStream API"? The simple answer is that we didn't invest the time back then to think about how a single API could serve both use cases.

Why is it bad to have too many APIs?

We see two main problems with the current situation:

It is not practical to reuse Flink application's for unbounded and bounded processing when requiring a physical API:

We think it is common enough for users to write a pipeline for analyzing streaming/unbounded data and then, later on, want to reuse that same code for processing bounded/batch data. For example, a real-time pipeline would read from Kafka while you might want to process historical data from S3. In theory, you can use the DataStream API with bounded sources, but you will not get efficient execution or fault-tolerance behavior. On failures, the entire pipeline has to restart. This is different from the execution model of the DataSet API where just a single operation or connected subgraph needs to be restarted because we can keep intermediate results of operations.

Working with event-time semantics is a lot easier when you know all your input beforehand. The watermark can always be "perfect" because there is no early or late data, and the algorithms and data structure we use for batch-style execution can take that into account.

The DataSet and DataStream API have different sets of available connectors because they use different APIs for defining sources and sinks. You cannot, for example, read a bounded interval from Kafka with a batch-style job.

Finally, we think that the event-time/windowing features of the DataStream API can be useful also for batch processing. For example, when you want to work with time-series data. Currently, you can use the DataStream API and live with subpar execution behavior or use the DataSet API and manually implement the windowing using sorting/grouping.

Users have to decide upfront between APIs:

This increases cognitive load and makes Flink less approachable to new users. If they make the wrong choice initially, they will not be able to switch later without significant time investments.

Another facet of this is that bigger organisations that want to adopt Flink might be discouraged by having to educate its engineers on two different APIs and their potential semantic differences, e.g. what is lateness, what is event time and if it is relevant to batch, etc.

Proposed Changes

We propose to deprecate the DataSet API in favor of the Table API/SQL and the DataStream API. For this to be feasible, we need to enhance the Table API/SQL and DataStream API to be useable replacements for cases where you would previously use the DataSet API. We will sketch the required changes here but defer more concrete plans to follow-up FLIPs. With this proposal, we only want to get community consensus on the general idea of deprecating the DataSet API with an overview of the required changes in the other APIs.

Required Changes in the "survivor" APIs

Table API/SQL:

- It must be easily possible to define sources/sink inline in code. This is covered by FLIP-129: Refactor Descriptor API to register connectors in Table API.
- We should have "imperative" operations on `Table` that are easy to use, that is there should be "ergonomic" `map/filter/flatMap`. These operations should be Row/Record oriented, as opposed to the column-oriented nature of the regular Table API operations. Users then don't have to learn the expression DSL syntax to write operations.
- Transitively, we also want to deprecate/remove the Legacy Table API batch planner along with the batch execution environments, because that interoperates with the DataSet API

DataStream API:

- We need a source API that works for bounded and unbounded sources. This is covered by FLIP-27: Refactor Source Interface.
- We need a sink API that works for unbounded and bounded sources. This is covered by FLIP-143: Unified Sink API
- We need to define a common set of execution semantics that work for both batch and streaming execution, this includes re-thinking some of the decisions in the DataStream API to make them work in a unified world. This is covered by FLIP-134: Batch execution for the DataStream API
- We need to use efficient, batch-style execution for DataStream programs when the topology is bounded. This is covered by FLIP-140: Introduce batch-style execution for bounded keyed streams
- Especially for machine learning use cases, we need robust support for iterative computation. The current support for iterations in the DataStream API should be considered experimental and it doesn't support dynamic termination criterions as does the DataSet API. However, I think we will need to tackle this in a follow-up FLIP.

What API/SDK should be used for which use case

We currently don't have clear guidance for users about which API they should use. We need to agree on recommendations and then proactively promote them in the documentation and general marketing. This could potentially take the form of a decisions tree or other graphical decision-making tool/app.

A summary of our current thinking is:

- If you have a schema and do not have "low level" operations SQL/Table
- If you need explicit control over the execution graph, the operations, the state in the operations use the DataStream API

This being said, it should be possible to freely convert between the Table API and DataStream API. FLIP-136: Improve interoperability between DataStream and Table API is making these efforts more concrete.

Compatibility, Deprecation, and Migration Plan

The DataSet API should be marked as deprecated in the documentation and code with a description of what the future direction of the project will be. We should remove the DataSet API in an upcoming version once 1) users had enough time to migrate existing use cases to other APIs, and 2) we are sure enough that the remaining APIs sufficiently cover the use cases of the DataSet API. This is contingent on the follow-up FLIPs mentioned above so this FLIP can be considered *complete* when the conditions sketched above are met.

It is important to keep in mind that we cannot simply remove the DataSet API from one release to the next. This will be a longer process where we need to ensure that existing users of the DataSet API can migrate and do migrate. There are companies that heavily invested in the DataSet API and cannot leave them behind.

Rejected Alternatives

We think there is no alternative to reducing the number of available APIs if their overlap is as obvious as for the DataStream and DataSet APIs. We could theoretically deprecate the DataStream API in favour of the DataSet API but we think that the DataStream API is the more widely used API and that it is also currently more fully-featured (see event-time handling and windows). This also resonates well with the thought that a batch is a part of a stream and that batch processing is a strict subset of stream processing.