FLIP-281 Sink Supports Speculative Execution For Batch Job

Status

<table>
<thead>
<tr>
<th>Discussion thread</th>
<th><a href="https://lists.apache.org/thread/l05m6cf8fwkbpnjtzbg9l2lo40oxzd1">https://lists.apache.org/thread/l05m6cf8fwkbpnjtzbg9l2lo40oxzd1</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vote thread</td>
<td><a href="https://lists.apache.org/thread/21rznxp0og6c6nkib5zrmvkh8gptx7t">https://lists.apache.org/thread/21rznxp0og6c6nkib5zrmvkh8gptx7t</a></td>
</tr>
<tr>
<td>JIRA</td>
<td><img src="https://lists.apache.org/thread/l05m6cf8fwkbpnjtzbg9l2lo40oxzd1" alt="FLINK-30725" /> - FLIP-281: Sink Supports Speculative Execution For Batch Job [CLOSED]</td>
</tr>
<tr>
<td>Release</td>
<td>1.17</td>
</tr>
</tbody>
</table>

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

Motivation

This proposal is a follow up of FLIP-168: Speculative Execution For Batch Job. There are some limitations in FLIP-168, like Source and Sink do not support speculative execution. FLIP-245 has resolved the source issue, this proposal aims to make Sink support speculative execution.

Compared with normal operators, Sink is more special. Because Sink always interacts with external systems (often storage systems) which are not controlled by the Flink. Therefore when speculative sink execution is enabled, additional work is required to guarantee data consistency.

Public Interfaces

Introduce SupportsConcurrentExecutionAttempts interface

After supporting speculative Sink, it's still be disabled by default due to the compatibility considerations. As mentioned above, Sink always interacts with external systems. The current implementation of Sink probably does not support several subtask attempts running at the same time. We don't want to see the user's program have wrong results after upgrading the Flink version. So the speculative execution for Sink will be disabled unless the sink developer explicitly indicates that it can support concurrent execution attempts. We introduce a new interface SupportsConcurrentExecutionAttempts to do this.

```java
public interface SupportsConcurrentExecutionAttempts {
}
```

Flink enables speculative execution if the Sink implementation inherits this interface. This strategy works for all Sink APIs: SinkFunction, Sink V2 and OutputFormat.

Expose the attempt number for Sink V2 SinkWriter

Sink implementation may need the attempt number to isolate produced results. However, currently SinkWriter does not provide the attempt number.

So we add a new getAttemptNumber method in org.apache.flink.api.connector.sink2.InitContext.
SinkWriter

public interface Sink<InputT> extends Serializable {

    SinkWriter<InputT> createWriter(InitContext context) throws IOException;

    @PublicEvolving
    interface InitContext {
        ... 
        /**
         * Gets the attempt number of this parallel subtask. First attempt is numbered 0.
         * @return Attempt number of the subtask.
         */
        int getAttemptNumber();
    }
}

Expose the attempt number for OutputFormat

OutputFormat also does not provide the attempt number. We add a new open method in org.apache.flink.api.common.io.OutputFormat, and deprecate the old open method.
OutputFormat

`@Public
public interface OutputFormat<IT> extends Serializable {
    // ...

    @Deprecated
default void open(int taskNumber, int numTasks) throws IOException {}

    /**
     * Opens a parallel instance of the output format to store the result of its parallel instance.
     *<p>When this method is called, the output format is guaranteed to be configured.
     * @param context The context to get task parallel infos.
     * @throws IOException Thrown, if the output could not be opened due to an I/O problem.
     */
default void open(InitializationContext context) throws IOException {
        open(context.getTaskNumber(), context.getNumTasks());
    }
    // ...
}

interface InitializationContext {
    /**
     * Gets the parallelism with which the parallel task runs.
     * @return The parallelism with which the parallel task runs.
     */
    int getNumTasks();

    /**
     * Gets the number of this parallel subtask. The numbering starts from 0 and goes up to parallelism-1 (parallelism as returned by `getNumTasks()`).
     * @return The index of the parallel subtask.
     */
    int getTaskNumber();

    /**
     * Gets the attempt number of this parallel subtask. First attempt is numbered 0.
     * @return Attempt number of the subtask.
     */
    int getAttemptNumber();
}

Expose the finished subtask attempts for FinalizeOnMaster

We add a new `finalizeGlobal` method in `org.apache.flink.api.common.io.FinalizeOnMaster` to expose finished subtask attempts, and deprecate the old `finalizeGlobal` method. In this way, the `FinalizeOnMaster` of `OutputFormat` could commit the produced data exactly. It would be discussed detailedly in the proposal changes section.
Proposed Changes

The most important thing is that the intermediate or final data produced by Sink must be guaranteed that they would not interfere with each other when speculative execution is enabled. The concurrent tasks launched by speculative execution with the same subtask id have different attempt numbers which could be used to isolate produced results. So it’s necessary to expose the attempt number to Sink implementation.

Currently, there are several kinds of Sink, SinkFunction, Sink V1 (introduced by FLIP-143), Sink V2 (introduced by FLIP-191), OutputFormat based Sink. Since Sink V1 is deprecated and replaced by Sink V2, it is out of the scope of this discussion. For these different Sink APIs, we discuss them separately below.

SinkFunction

Currently, SinkFunction implementation (must inherit RichSinkFunction) could get the attempt number through RuntimeContext. There is not much abstraction for SinkFunction API, all behaviors are controlled by SinkFunction implementation. The implementation must fully consider the impact of enabling speculative execution.

Sink V2

Several components are introduced by Sink V2, we discuss them separately here.

SinkWriter

First of all SinkWriter needs to expose the attempt number which is discussed in the SinkFunction section above. There will be a new method named “getAttemptNumber” added in org.apache.flink.api.connector.sink2.Sink.InitContext (Please check the public interfaces section). Through the attempt number, SinkWriter may take care of the produced results even with several concurrent subtask attempts running at the same time just like SinkFunction mentioned above.
Committer

Speculative execution of Committer will be disabled. Because

1. The concurrent committing is hard or meaningless to support. As we know, commit always depends on the exclusive semantics of the external system. Parallel commits may cause conflicts with each other. Even if parallel commits do not cause problems, the exclusive semantics determine that it is difficult to execute the commit in parallel. Multiple committer attempts running in series is not what we want when speculative execution is enabled.

2. Commit is very unlikely to be the bottleneck of the batch job. Furthermore, the committer task is rarely bothered by hardware issues like high CPU load or busy I/O. Because Committer generally performs meta-related lightweight operations, rather than data-related heavy IO operations. For example, file renaming, meta or transaction committing, etc.

PreWriter, PreCommitter, PostCommitter

Technically, there is not much difference between these operators extended by Sink V2 with normal non-sink operators. The only thing different here is whether speculative execution of them should be enabled or not. This is more of a compatibility issue than a technical issue.

1. The speculative execution of pre/post-committers would be disabled as the same with Committer. Some Sink implementations use post-committer to achieve global committer semantics, like SinkV1Adapter.

2. The speculative execution of pre-writer would be enabled if SinkWriter enables speculative execution. "Imagine the scenario, pre-writer and SinkWriter are chained into one task, and they have different speculative execution strategies. As we all know, speculative execution works on task level. So there is a conflict between chain and speculative execution strategy. We can either break the chain or disable speculative execution for this task to solve the problem. If the chain is broken, it may cause a performance regression that we do not want to see. Therefore, we decided to respect chain strategy rather than speculative execution. By the way, there would be more options if fine-grained speculative execution is supported (operators could have different speculative execution strategy), we will discuss it in the rejected alternatives sections.

3. The chain and speculative execution semantics conflict also exists between SinkWriter and committer (or pre-committer). Currently, in the batch mode Flink breaks the chain between SinkWriter and committer (or pre-committer). So that would be fine SinkWriter enables speculative execution but committer does not.

OutputFormat

OutputFormat can be divided into two parts, writing data and committing data.

Writing Data

Like what we discussed in Sink V2 SinkWriter section, OutputFormat also needs to expose the attempt number which can be used to isolate data produced by speculative executions with the same subtask id. The old open method of OutputFormat would be deprecated. We introduce a new open method with a context provided that is easier to extend (Please check the public interfaces section). Users could get the attempt number through this context.

Committing Data

Currently, for the OutputFormat that implements the FinalizeOnMaster interface, the JobMaster will call the FinalizeOnMaster#finalizeGlobal method to perform a global finalization operation after all OutputFormat tasks are done. In this finalization, we can do some committing work, for example, FileSystemOutputFormat and HiveOutputFormat move temporary files to the final directory.

OutputFormat naturally has a global committer (FinalizeOnMaster), we can do data cleaning and committing in the global committer like Hadoop and Spark do. If speculative execution is enabled, there might be several subtask attempts finished (also several produced data) at the same time. However there is no data exchange between OutputFormat and FinalizeOnMaster. It’s hard for the FinalizeOnMaster to know which subtask attempt is the one marked as finished by JM. To solve this issue, we introduce a new finalizeGlobal method for FinalizeOnMaster. The new method would provide a context which the user could get the really finished subtask attempts from (Please check the public interfaces section). In this way, it can decide easily which part of the produced results can be discarded, and which part can be committed through the finished subtask attempt information.

Compatibility, Deprecation, and Migration Plan

As you can see above, all public interface changes are compatible.

Limitations

The speculative execution of Committer and pre/post-committer operators of Sink V2 are disabled. Please check more details in the rejected alternatives section.

Test Plan

1. The changes will be covered by unit and IT cases.
2. Test the functionality in a real Standalone/Yarn/Kubernetes cluster.
Rejected Alternatives

Compute speculative execution before chaining

The idea is about Flink respects speculative execution first rather than chaining strategy. If some operators with different speculative support can be chained, only the operators with the same speculative support would be chained. This proposal was rejected because we were afraid of performance regression after breaking the chain.

Introduce a way to clean leaked data of Sink V2

The proposal is introducing a way (a method or a new interface) to Sink V2 Committer to allow it to know which subtask attempts are “really” finished like what we do in FinalizeOnMaster changes. Hadoop and Spark have a similar mechanism in Committer. Currently, the Sink V2 Committer may solve the issue through a well-designed “Committable” protocol between SinkWriter and Committer. But if the Committer knows which subtask attempts are finished, it has an easier way to check if leaked data is produced by speculative execution or not (If several subtasks attempts finished at the same time, they may produce duplicated data in the external storage system). We rejected this proposal because

1. In a way, it’s an orthogonal issue. Even if speculative execution is disabled, there might be still leaked data produced by task failover. The Sink V2 implementation must take care of the intermediate result produced by failed SinkWriter. In the speculative execution scenario, a finished SinkWriter (in the perspective of TM not JM) may cause data leaking if another SinkWriter with the same subtask id finished faster. Somehow, speculative execution of Sink V2 just increases the possibility of an existing issue, not involving a new issue.
2. Unlike Hadoop or Spark there is no global committer mechanism of Sink V2. It’s a bit weird to do the cleaning in a non-global committer. There might be conflicts between parallel committers. We decide to not have this feature until a global committer is introduced.

Fine-grained control of enabling speculative execution for operators

The idea is about introducing a method of DataStream API to allow the user to decide the operator enabling speculative execution or not. In this way, the extended operators of Sink V2 could support more scenarios. For example, we could enable speculative execution for pre-committer to do some heavy IO operations, like compaction. We rejected this proposal because we thought it’s a bit out of this FLIP scope. It’s more like an orthogonal issue. Non-sink operators also encounter this problem. It’s better to have the discussion in a separate FLIP.