FLIP-168: Speculative Execution for Batch Job

Status

<table>
<thead>
<tr>
<th>Discussion thread</th>
<th><a href="https://lists.apache.org/thread/ot352tp8t7mcclzx9zf790gcmfwrq58">https://lists.apache.org/thread/ot352tp8t7mcclzx9zf790gcmfwrq58</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vote thread</td>
<td></td>
</tr>
<tr>
<td>JIRA</td>
<td><img src="https://issues.apache.org/jira/browse/FLINK-28131" alt="FLINK-28131 - FLIP-168: Speculative Execution for Batch Job" /></td>
</tr>
<tr>
<td>Release</td>
<td>1.16</td>
</tr>
</tbody>
</table>

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

- Status
- Motivation
- Public Interfaces
  - Configuration
  - Metrics
- Proposed Changes
  - SlowTaskDetector
    - ExecutionTimeBasedSlowTaskDetector
    - Definition of execution time
  - SpeculativeExecutionVertex
    - State Priority
  - SpeculativeScheduler
    - Scheduler directly deploys executions
    - Handling execution failures
    - Handling slow tasks
- Limitations
  - Batch jobs only
  - No support for DataSet jobs
  - No support for PIPELINED data exchanges
  - Speculative execution of sources and sinks are disabled by default
  - No integration with Flink's web UI
  - User defined functions must not be affected by its speculative instances
- Compatibility, Deprecation, and Migration Plan
- Test Plan

Motivation

In Flink batch processing, a job is usually divided into multiple parallel tasks that execute across many nodes in the cluster. It is common to encounter performance degradation on some nodes due to hardware problems, accident I/O busy, or high CPU load. This kind of degradation can slow the tasks running on it. And the slow tasks may become the bottleneck of the job.

To solve this kind of problem, we propose a speculative execution mechanism to Flink. The basic idea is to start mirror tasks on other nodes when a slow task is detected. The mirror task processes the same input data and produces the same data as the original task. The original task and mirror tasks are treated equally. Once any of them finishes, its output will be admitted and the remaining homogeneous tasks will be canceled.

Public Interfaces

Configuration

Speculative executions will work along with adaptive batch scheduling (FLIP-187). New configuration options will be added for the scheduler:

- `jobmanager.adaptive-batch-scheduler.speculative.enabled`, default to "false". It controls whether to enable speculative execution. Note that speculative execution only works with AdaptiveBatchScheduler, which requires "jobmanager.scheduler" to be set to "AdaptiveBatch".
- `jobmanager.adaptive-batch-scheduler.speculative.max-concurrent-executions`, default to "2". It controls how many executions (including the original one and speculative ones) of an ExecutionVertex can execute at the same time.
- `jobmanager.adaptive-batch-scheduler.speculative.block-slow-node-duration`, default to "1 min". It controls how long an identified slow node should be blocked for.

New configuration options will be added in SlowTaskDetectorOptions for slow task detection:
Metrics

We propose to add following metrics to expose job problems and show the effectiveness of speculative execution:

1. numSlowExecutionVertices: Number of slow execution vertices at the moment.
2. numEffectiveSpeculativeExecutions: Number of speculative executions which finish before their corresponding original executions finish.

Proposed Changes

When speculative execution is enabled, a SpeculativeScheduler (which extends AdaptiveBatchScheduler) will be used for task scheduling. SpeculativeScheduler will listen on slow tasks detected by SlowTaskDetector. It will create and deploy speculative executions for the slow tasks. Nodes that slow tasks located on will be treated as slow nodes and get **blocked**, so that speculative executions will not be deployed on them. Once any execution finishes, the remaining homogeneous tasks will be canceled, so that only one execution will be admitted as finished and only its output will be visible to downstream consumer tasks or in external sink services.

A SlowTaskDetector will periodically check all the current tasks/executions and notify the SlowTaskDetectorListener about the detected slow tasks. SpeculativeScheduler will register itself as the SlowTaskDetectorListener.
SlowTaskDetector

```java
public interface SlowTaskDetector {
    void start(
        ExecutionGraph executionGraph,
        SlowTaskDetectorListener listener,
        ComponentMainThreadExecutor mainThreadExecutor);

    void stop();
}

public interface SlowTaskDetectorListener {
    void notifySlowTasks(Map<ExecutionVertexID, Collection<ExecutionAttemptID>> slowTasks);
}
```

In the first version, an ExecutionTimeBasedSlowTaskDetector will be used. In the future, we can make SlowTaskDetector customizable to support different slow task detecting strategies, e.g. based on task throughput.

ExecutionTimeBasedSlowTaskDetector

For ExecutionTimeBasedSlowTaskDetector, if a task's execution time is much longer than that of most tasks of the same JobVertex, the task will be identified as slow. More specifically, it will compute an execution time baseline for each JobVertex. Tasks which execute longer than or equals to the baseline will be identified as slow tasks.

Here's the process to compute the baseline of a JobVertex:

- define $N =$ parallelism of JobVertex
- define $R =$ finished task ratio, set via config
- define $M =$ baseline multiplier, set via config
- define $B =$ lower-bound of baseline, set via config
- the baseline will be computed only if $N \times R$ ExecutionVertex of the JobVertex have finished
  - $T =$ median of the execution time of the earliest $N \times R$ finished tasks
  - baseline = max($T \times M, B$)

Related configurations are:

- finished task ratio: `slow-task-detector.execution-time.baseline-ratio`
- baseline multiplier: `slow-task-detector.execution-time.baseline-multiplier`
- lower-bound of baseline: `slow-task-detector.execution-time.baseline-lower-bound`

Note that the baseline will be used only after $N \times R$ ExecutionVertex of the JobVertex have finished. No task of the JobVertex will be identified as slow before that.

Definition of execution time

The execution time of a task/execution includes the deploying time, initializing time and running time. Because processes of task deploying, initializing and running all will be affected by a slow node. This means:

- for a FINISHED task, execution time = timestamp[FINISHED] - timestamp[DEPLOYING]
- for a DEPLOYING/INITIALIZING/RUNNING task, execution time = current timestamp - timestamp[DEPLOYING]
- for other tasks, execution time = 0

SpeculativeExecutionVertex

SpeculativeExecutionVertex will be used if speculative execution is enabled, as a replacement of ExecutionVertex to form an ExecutionGraph. The core difference is that a SpeculativeExecutionVertex can have multiple current executions running at the same time.

SpeculativeExecutionVertex extends ExecutionVertex. It needs to implement new methods as below:
**SpeculativeExecutionVertex**

```java
class SpeculativeExecutionVertex extends ExecutionVertex {
    public Execution getCurrentExecution(ExecutionAttemptID id);
    public Collection<Execution> getCurrentExecutions();

    /**
     * Create a new speculative execution and add it to current executions.
     */
    public Execution createNewSpeculativeExecution(long timestamp);

    /**
     * Returns whether the vertex contains source operators.
     */
    public boolean containsSources();

    /**
     * Returns whether the vertex contains sink operators.
     */
    public boolean containsSinks();

    ...
}
```

Besides that, it also needs to override some methods of ExecutionVertex:

- Operations like cancel/fail/markFailed/suspend must be applied to all the current executions.
- The current execution (returned by `getCurrentExecutionAttempt()`) should point to the execution which best represents the status of the execution vertex. We defined a **state priority** to achieve this goal.

**State Priority**

Even though SpeculativeExecutionVertex may have multiple current executions at the same time, `ExecutionVertex#getCurrentExecutionAttempt()` which returns one only current execution is still needed to:

- Show task execution status on the web UI. This is needed for compatibility purpose before the web UI is fully reworked to support display multiple concurrent executions.
- Help with the deployment of downstream tasks. In this case, the current execution must be FINISHED.

For the purposes above, the current execution should point to the FINISHED execution once any current execution has finished. If no execution has finished, the current execution should point to the execution with the fastest progress.

To achieve this goal, we define a state priority. A state has higher priority if it is more likely to transition to the FINISHED state. And the current execution should always point to the execution of the highest priority. If multiple current executions have the same highest priority, choose the earliest created one.
States from the highest priority to the lowest priority are:

- FINISHED
- RUNNING
- INITIALIZING
- DEPLOYING
- SCHEDULED
- CREATED
- CANCELING
- FAILED
- CANCELED

**SpeculativeScheduler**

A SpeculativeScheduler will be used if speculative execution is enabled. It extends AdaptiveBatchScheduler so that speculative execution can work along with the feature to adaptively tuning parallelisms for batch jobs.

The major differences of SpeculativeScheduler are:

- SpeculativeScheduler needs to be able to directly deploy an Execution, while AdaptiveBatchScheduler can only perform ExecutionVertex level deployment.
- SpeculativeScheduler does not restart the ExecutionVertex if an execution fails when any other current execution is still making progress.
- SpeculativeScheduler listens on slow tasks. Once there are slow tasks, it will block the slow nodes and deploy speculative executions of the slow tasks on other nodes.
- Once any execution finishes, SpeculativeScheduler will cancel all the remaining executions of the same execution vertex.

**Scheduler directly deploys executions**

Currently, the DefaultScheduler (base of AdaptiveBatchScheduler) can only perform ExecutionVertex level deployment. However, in this case, the scheduler is actually deploying the current execution of the ExecutionVertex.

Therefore, conceptually it is naturally to rework the DefaultScheduler to directly deploy an execution.

```java
public class DefaultScheduler {
    protected void allocateSlotsAndDeploy(List<ExecutionDeploymentOption> executionsToDeploy);
    ...
}

public class ExecutionDeploymentOption {
    ExecutionAttemptID executionId;
    DeploymentOption deploymentOption;
}
```

The ExecutionSlotAllocator also needs to be reworked to directly allocate slots for Executions.

```java
public interface ExecutionSlotAllocator {
    CompletableFuture<LogicalSlot> allocateSlotFor(ExecutionAttemptID executionAttemptId);
    /**
     * Cancel the ongoing slot request or release the allocated slot of an execution.
     */
    void cancel(ExecutionAttemptID executionAttemptId);
    ...
}
```

**Handling execution failures**
For SpeculativeScheduler, when a task fails,

1. if all the current executions of the execution vertex are no longer possible to finish, restart the execution vertex. An execution is possible to finish if and only if it is in CREATED/SCHEDULED/DEPLOYING/INITIALIZING/RUNNING/FINISHED state.
2. else if the failure is a PartitionException, it means that the consumed data is corrupted or lost. It will restart the producer execution vertex of the problematic partition.
3. otherwise, the execution vertex is still possible to finish. Do not trigger a restart in this case.

The failed execution will be removed from current executions to make room for possible speculative executions in the future.

Handling slow tasks

Once notified about slow tasks, the SpeculativeScheduler will handle them as below:

1. Block nodes that the slow tasks locate on. To achieve this, the scheduler will add the slow nodes to the blocklist. The block action will be MARK_B LOCKED so that future tasks will not be deployed to the slow node, while deployed tasks can keep running. (See FLIP-224 Blocklist Mechanism for more details)
2. Create speculative executions for slow tasks until the current executions of each execution vertex reach the concurrency limit (defined via config jobmanager.adaptive-batch-scheduler.speculative.max-concurrent-executions)
3. Deploy the newly created speculative executions

Limitations

Batch jobs only

Speculative execution only works with batch jobs. It is enabled if jobmanager.scheduler is AdaptiveBatch and jobmanager.adaptive-batch-scheduler.speculative.enabled is true in Flink configuration. Exceptions will be thrown if a streaming job is submitted in this case.

No support for DataSet jobs

DataSet API will be deprecated in the future and therefore is not supported. DataStream API is now the recommended low level API to develop Flink batch jobs.

No support for PIPELINED data exchanges

Currently, AdaptiveBatchScheduler does not support jobs with PIPELINED data exchanges. As a result, speculative execution does not support PIPELINED data exchanges either. Requiring all data exchanges to be BLOCKING also simplifies things, because each ExecutionVertex is an individual pipelined region in this case and can have individual speculations. Otherwise multiple ExecutionVertex from one pipelined region need to do speculative execution together.

This also means that

Speculative execution of sources and sinks are disabled by default

Speculative execution of sources and sinks are disabled by default, this is because not all sources and sinks can work with speculative execution.

There will be several follow up proposals to enable speculative execution of sources and sinks:

- FLIP-XXX(coming soon) is a proposal to enable speculative execution to work with most sources, without changing the source connectors
- FLIP-XXX(coming soon) is a proposal to improve Flink interfaces to support speculative execution of sinks. Sinks must implement these interfaces to work with speculative execution
- FLIP-XXX(coming soon) is a proposal to improve FileSink to work with speculative execution
- FLIP-XXX(coming soon) is a proposal to improve HiveTableSink to work with speculative execution

With these proposed improvements, the scheduler will be able to see whether the given source or sink support speculative execution. It will enable speculative execution for a JobVertex only if all the contained sources and sinks support speculative execution.

No integration with Flink’s web UI

The web UI does not show all the concurrent executions of each ExecutionVertex/subtask. It only shows the one with the fastest progress.

User defined functions must not be affected by its speculative instances

When a user defined function and its speculative instances run concurrently, they must not affect each other. For example,

- access to the same exclusive resources
- overriding the output to external services which happens as a side effect, i.e. not via Flink sinks
- competition for data ingestion. Note that it includes cases that
- user defined source function competition
- data ingestion happens as a side effect, i.e. not via Flink sources.

Once the concurrent instances can affect each other, it may result in task failures, or even worse, data inconsistency. So that speculative executions should not be enabled in this case.

Compatibility, Deprecation, and Migration Plan

Speculative execution is enabled only if the newly introduced configuration option `jobmanager.adaptive-batch-scheduler.speculative.enabled` is `true` (false by default). This entails that Flink's default behavior won't change.

Test Plan

The proposed changes must be covered with UT, IT and E2E tests. It should also be tested via TPC-DS benchmarks in a real cluster.