[Public] Flink Delta Source for Flink Steam API

History

Change Description	Author	Reviewer	Date
Draft 1 initial Version	Krzysztof Chmielewski	Tathagata Das	Dec 20th 2021
Delta Partition Handling	Krzysztof Chmielewski	Tathagata Das	Dec 31th 2021
Initial Table read in batches (Design Decision Matrix)	Krzysztof Chmielewski	Tathagata Das Scott Sandre	Jan 3rd 2022
Monitoring of Delta Table Changes	Krzysztof Chmielewski	Tathagata Das Scott Sandre	Jan 17th 2022
Added Appendix A: “Flink Exception Handling”	Krzysztof Chmielewski	Tathagata Das Scott Sandre	Jan 17th 2022
Added new section “Performance Optimizations”	Krzysztof Chmielewski	Tathagata Das Scott Sandre	Jan 17th 2022
Added Connector Options	Krzysztof Chmielewski	Tathagata Das Scott Sandre	Jan 31st 2022

Motivation

After successful development of Delta Standalone Reader the next step is to build upon it and implement independent connectors.

Apache Flink with its constantly growing popularity and community as a data processing framework becomes one of the most natural choices to start with. Additionally it will provide a solid underlining that Delta Lake is perfectly well suited for both batch and streaming cases.

Requirements

MUST:

- Provide independent module in Delta Lake Connectors repository containing Apache Flink Source with exposed Java API for Apache Flink Delta Source (FDS),
- Users can use FDS to create Flink's DataStreams API sources
  - Both batch and streaming cases should be supported
- Users can use FDS to create Flink's Table API sources
  - When having already implemented Flink's `DeltaSource` which will allow one to create sources for DataStreams, then additional classes need to be provided to be interoperable with Table API.
- Flink pipeline can provide end-to-end exactly-once guarantees
  - checkpoint all processed files to avoid reprocessing them in case of failover
  - checkpoint all created splits (unassigned and assigned to the readers)
- In Batch mode, the entire Delta Table snapshot is read and the connector ignores any further changes/updates for this table.
- In Batch mode, time travel is supported, meaning that the connector can read a single

historical Delta Table Version.
- In Streaming mode, the whole Delta Table snapshot is read and then incrementally added files.
- In the Streaming mode, the connector will not handle input that is not an append and by default will throw an exception if any modifications occur on the table being used as a source**.**
- In the Streaming mode, the connector will provide a functionality of reading historical changes starting from specified version or timestamp without processing the entire table.
- In the Streaming mode, the connector will not support time travel, only reading historical changes will be supported.
- Flink version - 1.13

SHOULD:
- N/A

COULD:
- N/A

Out-of-scope
- Flink TableAPI and Flink SQL support. Those will be captured in a dedicated design document.

Proposal sketch

OSS Repo and Maven Central
We will implement and add code to the current delta-io/connectors repository in the flink-connector module. This module will contain code for both, Flink Delta Source and Sink connectors and will produce one Jar artifact

Main Assumptions
The Delta Flink Source Connector will be builded based on Flink's new Unified Source Interface API Apache Flink starting from version 1.12 released a new API called “A Unified Source Interface” for building source connectors. In the future, this new API will fully replace the previous API based on the RichSourceFunction class and SourceFunction interface.

Currently the Flink community rewrites bundled Flink connectors to the new API. In Flink 1.14 connectors like Kafka, Hive, File or Pulsar are already based on the new API.

Details about The Flink Improvement Proposal document (FLIP-27) can be found here - https://cwiki.apache.org/confluence/display/FLINK/FLIP-27%3A+Refactor+Source+Interface

One of the benefits of the new API is that Sources implementing FLIP-27 can be very easily used to build a connector for TableApi (SQL). However those can be used only as Scan Sources and not LookupSources. Hence for Flink Lookup Joins, it is needed to implement a new Table Source based on LookupTableSource interface. This is not a problem in our case since Delta is not meant to fine grained look up workloads.

Using Parquet Files as a data source for Flink pipeline is already possible through Flink's File Source, hence Flink's FileSource would be a blueprint for building Flink DeltaSource.

Delta Source will participate with Flink checkpoint mechanism described in Flink documentation under:

1. https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/fault-tolerance/checkpointing/
2. https://nightlies.apache.org/flink/flink-docs-master/docs/learn-flink/fault_tolerance/

Every reference to "checkpoint" in this design dock means Flink' state checkpoint.

Unified Source Interface

A Data Source has four core components:

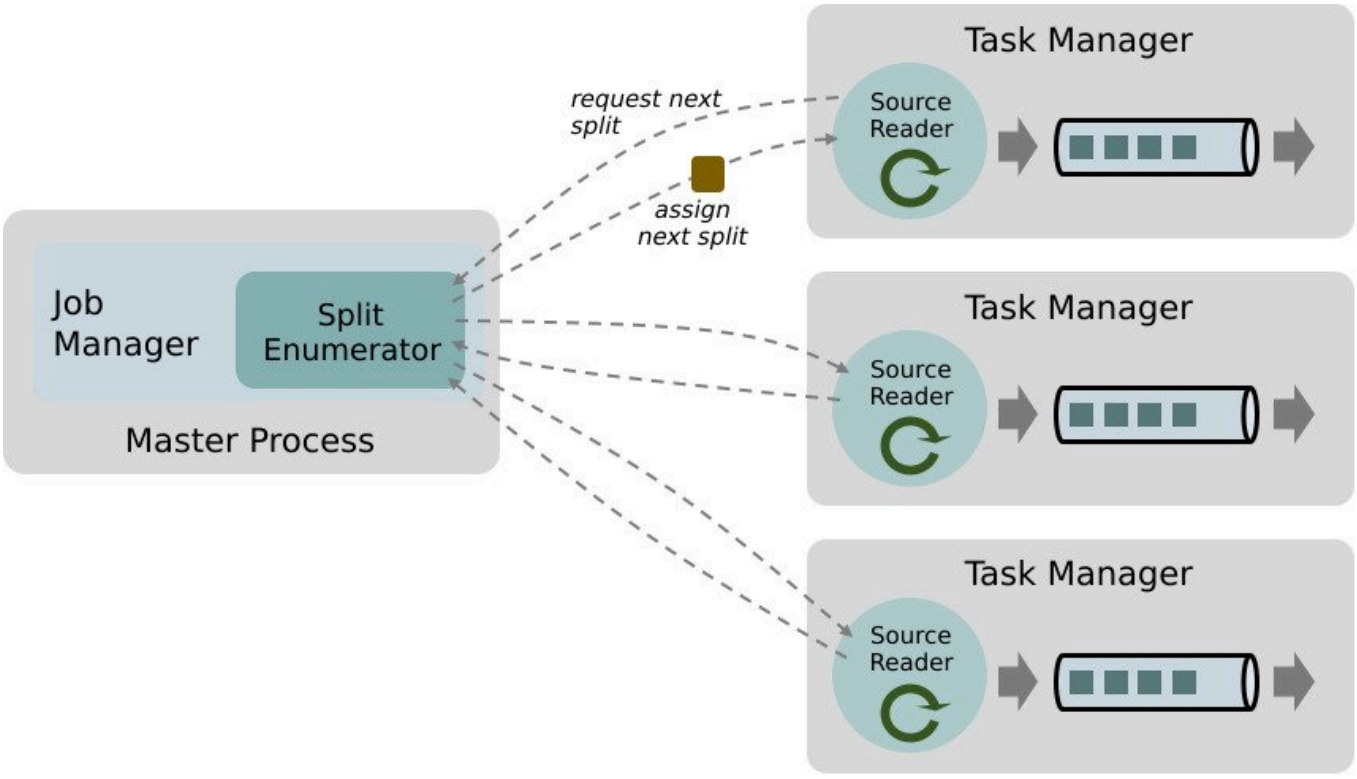
- Splits,
- SourceReader,
- SplitEnumerator,
- Source

A *Split* represents a unit of work that is consumed by a source connector. Split can represent a variety of things. It can be an entire file, file block or for example Kafka partition, Splits are granularity by which the source distributes the work and parallelizes the data reading.

The *SourceReader* requests *Splits* and processes them for example by reading the file or log partition represented by the *Split*. The *SourceReader* runs in parallel on the Task Manager in the SourceOperators and produces the parallel stream of events/records.

The *SplitEnumerator* generates the *Splits* and assigns them to the *SourceReaders*. It runs as a single instance on the Job Manager and is responsible for maintaining the backlog of pending *Splits* and assigning them to the readers in a balanced manner.

The *Source* class is an API entry point that ties the above three components together.



Existing File Source Implementation

This section describes existing Flink FileSource to provide the point of reference and better understanding about how Flink handles File sources. In this chapter there are no references to Delta Lake or Delta Table API. Those can be found in the next section - “Flink Delta Source”.

Flink already supports using Parquet files as a data source for pipeline. It is done through Flink’s File Source where Parquet is only one of the many file formats that this Source can read.

The FileSource is able to read files in blocks, where the block is defined by the underlying file system. For Example for local file system, meaning for local file system of the machine where Flink’s JVM runs, files will not be split into the blocks. However files sourced from HDFS may be split. There is also an additional condition that checks whether file format supports splitting, for example split will not be done for gz files. For DeltaSource we don't need this additional check since we will be reading only from Parquet files.

File source can work in one of two modes, **Bounded** and in **Continuous**. The bounded mode reads data from the file and finishes execution. It does not monitor for any data updates during that time. In the continuous mode on the other hand the Source’s monitor thread periodically checks whether there is any new file added to the monitored folder.

The SourceSplit implementation for FileSource defines the region of the file represented by the split. Splits can easily represent the whole file as well.

The split may furthermore have a "reader position", which is the checkpointed position from a reader previously reading this split. This position is typically *null* when the split is assigned from the enumerator to the readers, and it has a non-null value when the readers checkpoint their state in a file source split.

Source Checkpointing

Source state checkpointing is implemented based on Flink’s checkpoint mechanism.
Flink periodically takes persistent snapshots of all the state in every operator and copies it as state snapshots somewhere more durable, such as a distributed file system. In the event of the failure, Flink can restore the complete state of the application and resume processing as though nothing had gone wrong.

Details about Flink checkpoint mechanism can be found here - https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/fault-tolerance/checkpointing/

The checkpointing for FileSource is done in two places, on the Job Manager Node in SplitEnumerator and on the Task Manager Node inside SourceReader.

On Job Manager SplitEnumerator checkpoints all unassigned Source Splits and in case of Continuous mode, a list of already processed paths. In the Continuous mode, where FileSource periodically checks the monitored folder we need to know which files were already used for split creation.

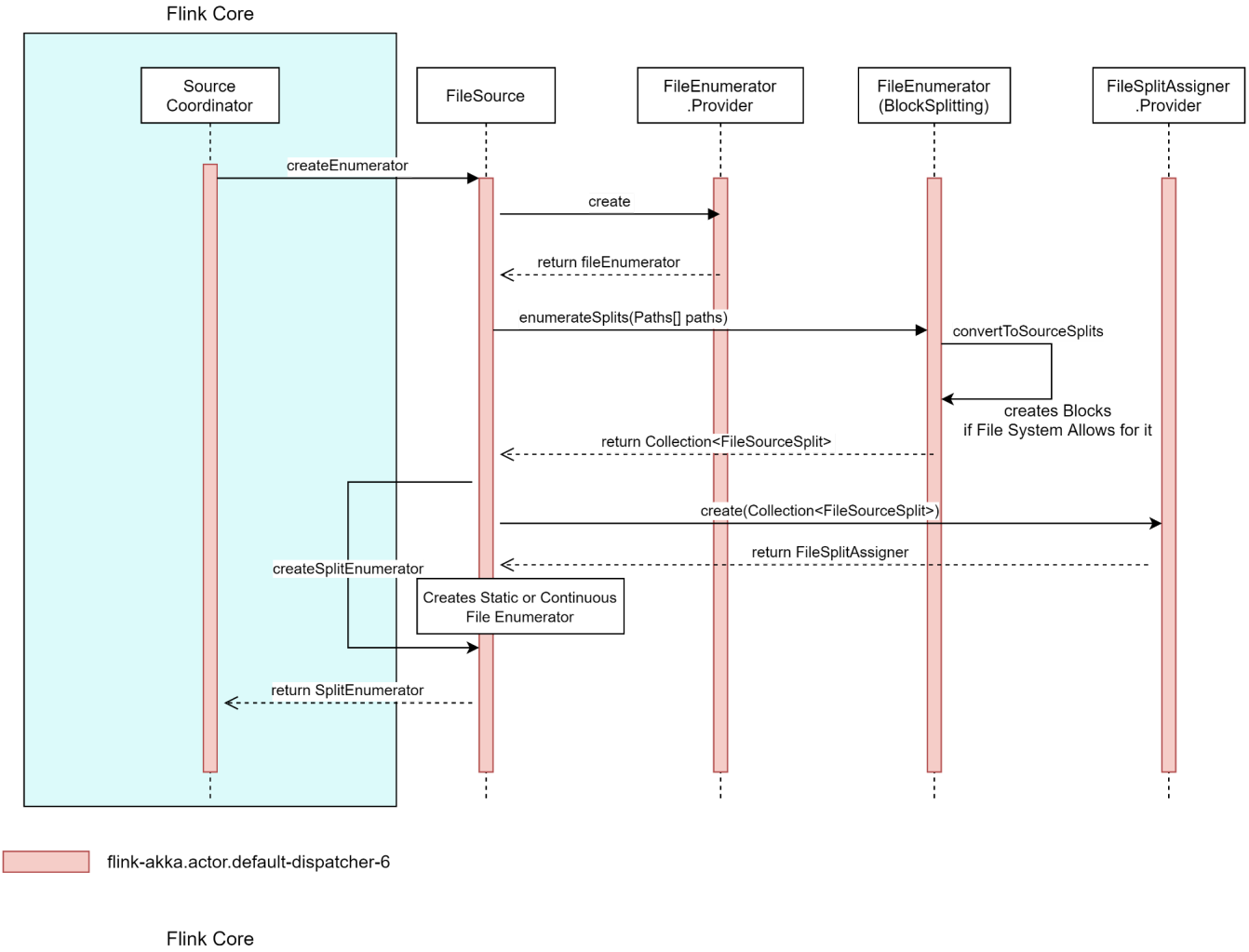
On each Task Manager Node, each SourceReader checkpoints only SourceSplits assigned to it.

Details about checkpointing for Delta Source Connector can be found in the Flink Delta Source > Checkpointing section.

Sequence Diagram

SplitEnumerator Creation

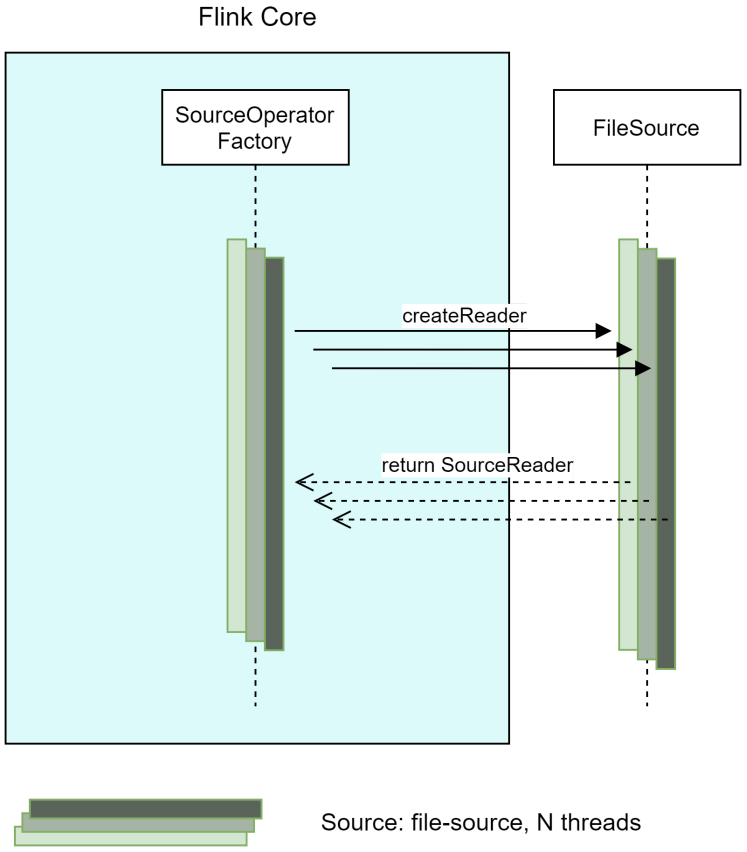
During this phase, SplitEnumerator creates all splits based on configuration paths.
In case of recovery from a checkpoint, it also recovers old, unassigned splits.
If Source is running in Continuous mode, the monitor thread is also started here.



SourceReader Creation

The number of source readers created by Flink is dictated by the parallelism level of source or entire job.

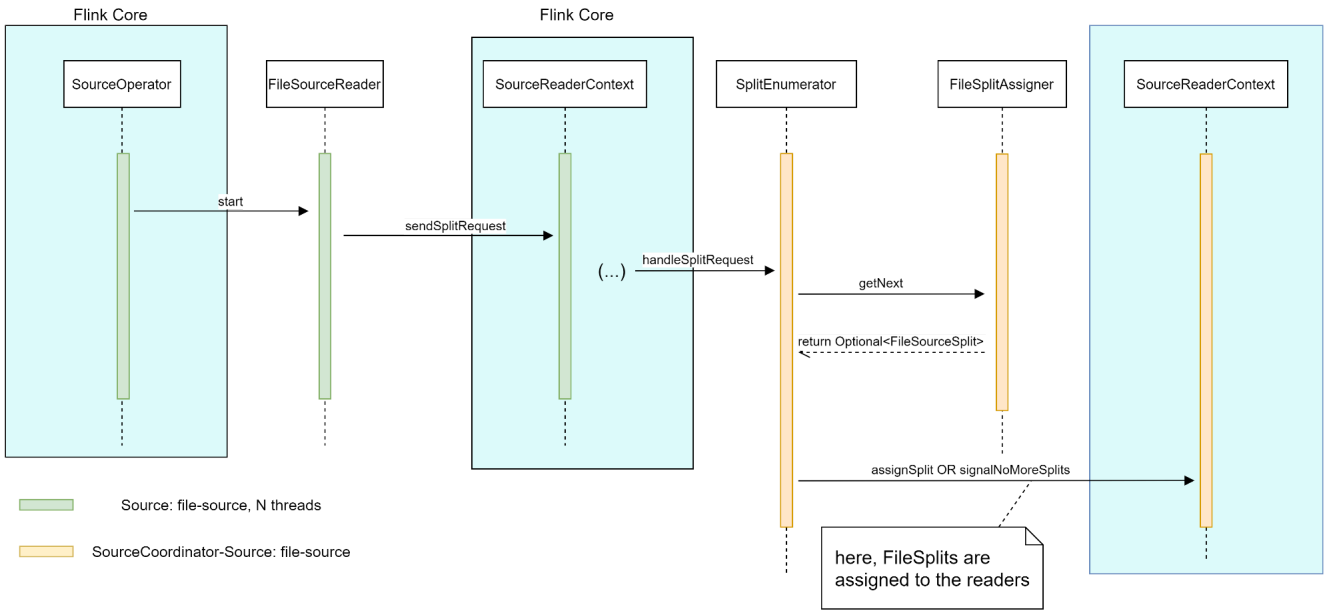
Each SourceReader is created by a separate thread.



Source Start

During this phase newly created SourceReaders (each on its own thread) are sending Split Requests through Flink Core to SplitEnumerator in order to manifest their readiness to process new data.
SplitEnumerator after receiving those requests, assigns new splits to the readers by calling *sourceReaderContext::assignSplit* method.

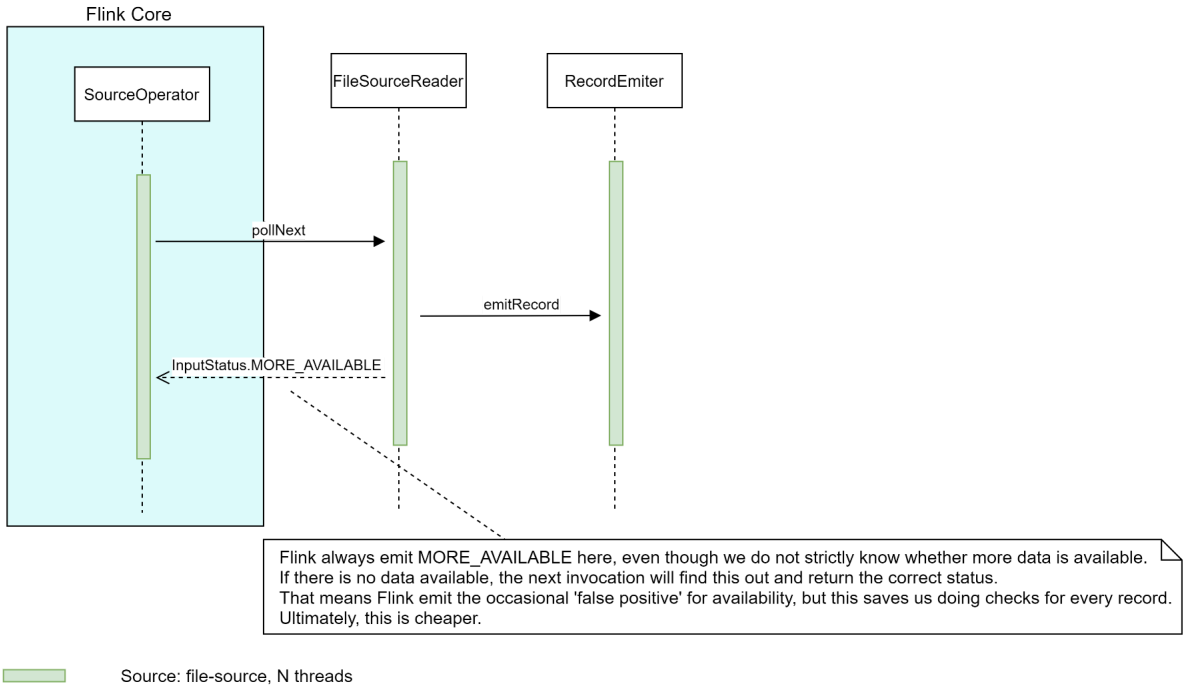
After that split is assigned to the SourceReader by calling *SourceReader::addSplits(List<Split>)* method by Flink Core. This last step was not presented on this diagram.



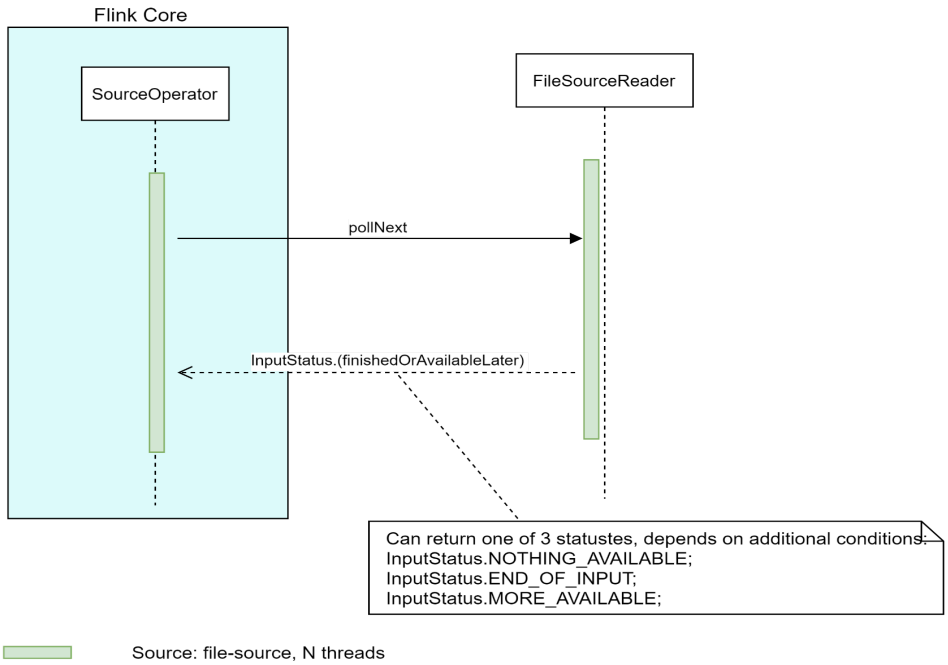
Source Reader

Reading from Source Reader in case there are non “empty” splits available. This operation is executed for every SourceReader on its own dedicated Source thread.

Reader has Splits



Reader has no Splits



A Flink task keeps calling pollNext(ReaderOutput) in a loop to poll records from the SourceReader. The return value of the pollNext(ReaderOutput) method, which is an instance of InputStatus type, indicates the status of the source reader.

MORE_AVAILABLE - The SourceReader has more records available immediately.
NOTHING_AVAILABLE - The SourceReader does not have more records available at this point, but may have more records in the future.
END_OF_INPUT - The SourceReader has exhausted all the records and reached the end of data. This means the SourceReader can be closed.

The exact implementation details where each Status is returned can be found in Flink's SourceReaderBase class which is a base class for many Source Readers for Source connectors like File Source, Kafka Source, Hive Source etc.

The actual code can be found under - https://github.com/apache/flink/blob/74a43511c714b3cc216f86500ab6bd3f5d1b49ad/flink-connectors/flink-connector-base/src/main/java/org/apache/flink/connector/base/source/reader/SourceReaderBase.java#L280

The InputStatus values are used for managing Flink's internal task lifecycle. End user or Source implementation do not have to handle those statuses. However, usually with returning NOTHING_AVAILABLE status, the reader will also notify SplitSourceEnumerator that it is available for the next splits. It does it by calling SourceReaderContext.sendSplitRequest() method which will be propagated to SplitEnumerator::handleSplitRequest method.

Flink Delta Source

Flink Delta Source will reuse existing File Source components whenever it is suitable without degrading code quality and benefit to cost ratio will be sufficient.

For example, low level classes like:

- FileSourceReader
- FileSplit
- FileSplitAssigner
- FileSourceSplitSerializer

Will be reused.

These classes handle processing of individual Parquet files along with providing a serialization facility needed for checkpoint mechanisms on the reader side. They are handling splitting files into individual data blocks that can be read in parallel.

On the other side, Delta Source Connector will have to implement its own:

- DeltaSource - Implementation of Source interface to "orchestrate" new connector.
- FileEumerators for Bounded and Streaming modes.
- Enumerator's state object wrapping File Split with Delta Log informations like, list of changes or list of already processed file paths.
- Serialization for Enumerator's state data.

These classes provide orchestration and what is called "a work discovery" for Source connector. They will leverage DeltaLog features.

Work Discovery is a term used in FLIP-27 to describe all necessary work needed to determine what data should be read both in bounded and continuous mode. An example could be a detection of a DeltaLog table data appended in a continuous source.

Reading the initial data

During initial Flink's job initialization, if there is no checkpoint to recover from, the logic to initialize splits from Delta Table will be the same for Bounded and Continuous modes.

The logic can be expressed with below code:

```
DeltaLog deltaLog = DeltaLog.forTable(new Configuration(),
"file:///path/to/table");

Snapshot snapshot = deltaLog.snapshot();
long startVersion = snapshot.getVersion();
List<AddFile> allFiles = snapshot.getAllFiles();
Collection<FileSourceSplit> createSplits(allFiles);
```

The Flink Delta Source Connector ideally should read data in chunks since we can have millions of files per Table. However, currently DeltaStandalone does not provide an API that will guarantee the Iteration order among returned AddFile iterators. Therefore, as for the first development iteration we will read the entire data at once.

Monitoring of Delta Table Changes

Overview

After processing the initial data, the connector will periodically check (configurable interval value) whether there are any new versions for the monitored Delta Table. If there are new versions, connector will then process all new AddFile actions and will ignore other actions such as Commit Metadata, ProtocolInfo etc.

The algorithm for Delta Table Work discovery is presented in code snippet below:

```
//recreating deltaLog for monitored table
DeltaLog deltaLog = DeltaLog.forTable(new Configuration(), "file:///path/to/table");

//list of files for which we already created a split.
List<String> alreadyProcessedFiles = SourceEnumerator::getAlreadyProcessedFiles();

long highestSeenVersion = currentVersion; //Current version will be checkpointed

Iterator<VersionLog> changes = deltaLog.getChanges(currentVersion, true);
while (changes.hasNext()) {
       VersionLog versionLog = changes.next();
       long version = versionLog.getVersion(); //get version for this Changes

      // track the highest version number for future use
       if (highestSeenVersion < version) {
              highestSeenVersion = version;
       }

      // create splits only for new versions
       if (version > currentVersion) {
              FileSourceSplit newSplits = buildSplitsForNewFiles(versionLog);
              alreadyProcessedFiles.addAll(extractPaths(newSplits));
       }
}

//update currentVersion so we will get changes starting from new value next time
currentVersion = highestSeenVersion;
```

Split creation is done per VersionLog element. It means that we will first check all actions for a given version before assigning any splits from this version to the Readers. This protects the connector from emitting any “outdated” data from Delta Table.

Handling Updates and Deletes

In case of RemoveFile action, the connector will do an additional check for the flag "dataChange". If this flag is set to true, it would mean that not only appending changes were applied on this table. In that case, the connector will stop further processing by throwing an Exception. For details about Flink exception handling please see Appendix A.

The RemoveFile actions can be ignored if the *"RemoveFile ::dataChange"* flag is set to false, and can be ignored. In that case no exception should be thrown.

The RemoveFile action with *"RemoveFile ::dataChange"* flag set to true will be ignored when user specifies one of two connector options:

- ignoreDeletes - if "true" and a particular version had only Deletes (no other actions) then do not throw an exception on RemoveFile regardless of the *"RemoveFile ::dataChange"* flag.

- ignoreChanges - if "true" and a particular version had a combination of deletes and other actions, then do not throw an exception on RemoveFile regardless of the *"RemoveFile ::dataChange"* flag.

Generally speaking, the Action can be processed in one of three ways:

- Normal processing, for example read data described by AddFile.
- Ignore Action - do nothing and continue reading.
- Throw an Exception.

The pseudocode covering this part would look as follows:

```
boolean ignoreChanges = sourceOptions.getValue(IGNORE_CHANGES);
boolean ignoreDeletes = ignoreChanges || sourceOptions.getValue(IGNORE_DELETES);

if (seenRemovedFile) {
       if (seenFileAdd && !ignoreChanges) {
              DeltaExceptions.deltaSourceIgnoreChangesError(version);
       } else if (!seenFileAdd && !ignoreDeletes) {
              DeltaExceptions.deltaSourceIgnoreDeleteError(version);
       }
}

boolean seenFileAdd is true when version had at least one AddFileAction with dataChange
flag set to true

boolean seenRemovedFile is true when version had at least one RemoveFileAction with
dataChange flag set to true;
```

Partitions

Delta's Protocol specifies that partition column's values have to be added to produced rows, and the actual partition values for a file must be read from the transaction log (https://github.com/delta-io/delta/blob/master/PROTOCOL.md#data-files).

Delta's AddFile Action will contain a map <String, String> where each entry represents a partition column name and its value. The value of that column must always be taken from AddFile partition information and not from the actual parquet row.

Flink API already supports such a scenario and can be reused for Delta Partition use cases. Flink's `ParquetColumnarRowInputFormat` class provides an API to define "user defined logic" which will compute the value for each partition column.
It is done via functional interface:

PartitionFieldExtractor<SplitT>
::extract(SplitT split, String fieldName, LogicalType fieldType)

Where "SplitT" is a generic type and must match the split type used in Source connector definition.

The partition column and its type has to be defined in a row format schema. This must be known upfront because Flinks has to create appropriate Serializers and Deserializes for every row column. This is done at a very early stage of Source creation.

The Partition column-value map will be passed to PartitionFileExtractor through DeltaSource Split implementation. The same approach for partitions was used for HiveSource in Flink's bundled connector.

DeltaSourceSplit

Due to Partition handling use cases, where information about partition value is kept in AddFile::partitionValues map we need to extend Flink's **FileSourceSplit** implementation by adding map with partitionValues. The class that implements this is named **DeltaSourceSplit**.

Checkpointing

Similar to File Source, DeltaSource will also handle checkpoints on Reader and Enumerator sides.

Reader

Reader checkpoints all its assigned Splits. Because we need to extend FileSourceSplit (partition use case), we need to provide a new (De)Serializer for the DeltaSourceSplit class - DeltaSourceSplitSerializer.

While checkpointing DeltaSourceSplit on a reader side we will checkpoint:

- map of partition values - field added in DeltaSourceSplit,
- splitId - comes from FileSourceSplit,
- file path for which this split is based on - comes from FileSourceSplit,
- position of first byte in the file to process - comes from FileSourceSplit,
- Length, which is the number of bytes in the file to process - comes from FileSourceSpit,
- number of bytes to process - comes from FileSourceSplit,
- readerPosition - comes from FileSourceSplit.

Detailed explanation for each field and entire FileSourceSplit class can be found in Flink Javadoc https://github.com/apache/flink/blob/master/flink-connectors/flink-connector-files/src/main/java/org/apache/flink/connector/file/src/FileSourceSplit.java

The DeltaSourceSplitSerializer will reuse Flink's FileSourceSplitSerializer for common fields.

Reader's checkpoint is done through Flink Coreby calling *SourceReader::snapshotState* method and serializing its result.

Split Enumerator

Split Enumerator checkpoints its state on a Task Manager node.

Information that is checkpointed by Split Enumerator contains all currently unassigned splits plus information from DeltaLog that is needed to resume work discovery in continuous mode and path for Delta Table since we need to recreate DeltaLog instance after recovery from checkpoint.

Recovering from checkpoint in the Continuous mode will always require refreshing the monitored Delta Table and creating new splits based on received changes.

Decision Matrix

Design Decision 1 - Using classes from Flink's codebase.

Delta Flink Source will be implemented using Unified Source Interface API.
There is a possibility to reuse already existing classes from File Source connector.
Problems like reading the Parquet format files, creating file blocks, Split management and checkpointing are already solved in that implementation.

Design assumes reusing FileSourceReader and FileSourceSplit classes along with FileSourceSplit serializer. Those classes seem to fit our use case without any issues.

However on the Enumerator side, things are a little bit more complex and they are boiled down to FileEnumerator implementation, whose job is to create SourceSplits based on given configuration.

Problems with existing code for FileEnumerator:

1. Inheritance is used heavily, where the good practice is always to favor composition over inheritance.

2. Switching between super and child class with contradictory names during file processing. An example for this would be *BlockSplittingRecursiveEnumerator* and its super type *NonSplittingRecursiveEnumerator*, where child class overrides only one method

3. Existing implementations contains checks and work pattern that is not needed in our case:
    a. Checks if the file is splittable - it will always be true in case of Parquet files.
    b. Expecting multiple source folders rather than one per source.
    c. Handling nested folders with Table directory.
    d. Filter unwanted files (since current logic works on a folder level not file level).

Option 1 - Reimplement logic using only needed code from FileSource (Preferred)

Implement Delta-specific implementation for creating SourceSplits. This will allow us to avoid all problems described in point 2 and 3. The "code duplication" with Flink's classes will remain on a low level since the existing implementation of *NonSplittingRecursiveEnumerator* has around 150 lines of code. Additionally the effort needed to implement Delta's version is small.

Pros:

- No dead code, unused conditions etc.
- Api and interfaces tailored to match our needs

- Easy to integrate smaller logical blocks rather than entire classes.

Cons:
- Had to duplicate some code from Flink base code

Option 2- Reuse existing code 1 to 1.
There will be pretty much none or very little development needed to achieve this option, however we will drag along all things described in above points to Delta Source Connector.

Pros:
- Very little work needed

Cons:
- Dead Code since not use cases from original code are relevant to Delta contex
- API not ideally matched to Delta Context
- Design chooses that makes code more coupled.

Design Decision 2 - Manage changes during Work Discovery
In Continuous mode, during normal execution or after recovering from checkpoint/savepoint most likely there will be a need to include changes that were appended to the monitored Delta Table. We are limiting the scope of supported changes to Appends, which means that only new data was added and there were no updates or deletes.

Delta Source has to keep in its state the last processed/discovered version number. This will be used to get new changes using Delta API.

Option 1 - using existing DeltaLog API (Preferred)
In this option we simply get all changes from previously observed highest version value and create new splits based on *AddAction* items for every VersionLog element from `Iterator<VersionLog>`

We need to track the version numbers along the process to find the highest one.

```
//recreating deltaLog for monitored table
DeltaLog deltaLog = DeltaLog.forTable(new Configuration(), "file:///path/to/table");

//list of files for which we already created a split.
List<String> alreadyProcessedFiles = SourceEnumerator::getAlreadyProcessedFiles();

long highestSeenVersion = currentVersion; //Current version will be checkpointed
```

```
Iterator<VersionLog> changes = deltaLog.getChanges(currentVersion, true);
while (changes.hasNext()) {
       VersionLog versionLog = changes.next();
       long version = versionLog.getVersion(); //get version for this Changes

      // track the highest version number for future use
       if (highestSeenVersion < version) {
              highestSeenVersion = version;
       }

      // create splits only for new versions
       if (version > currentVersion) {
              FileSourceSplit newSplits = buildSplitsForNewFiles(versionLog);
              alreadyProcessedFiles.addAll(extractPaths(newSplits));
       }
}

//update currentVersion so we will get changes starting from new value next time
currentVersion = highestSeenVersion;
```

Pros:

- Supported by Delta Standalone API

Cons:

- API users have to find out what is the highest version in given changes.
- There is no possibility to get changes from range of versions

Option 2a - enhance existing DeltaLog API

** This option doesn't improve correctness or performance. It suggests a potentially less error-prone API for the Delta Standalone :: DeltaLog class **

To detect table changes we need to keep track of the last processed table version.
In option 1 we check each AddFile for version number and check if the version is higher then the one we previously recorded.

We need to track the version number manually because in scenario:

```
oldversion = 5;
newestVersion = deltaLog.update().getVersion() → 10;
Iterator<VersionLog> changes = deltaLog.getChanges(oldversion , true);
```

The “changes” iterator can have changes from version higher than 10. Since there could have been changes applied to Delta Table in a short time between calling the deltaLog.update() and deltaLog.getChanges.

However, if we could know what is the newest table’s version without needing to check every VersionLog record and be able to get Delta Table’s changes bound to this upper version, the highest version check for every record would not be needed anymore.

The new proposed DeltaLog methods are highlighted in orange.

```
// recreating deltaLog for monitored table
DeltaLog deltaLog = DeltaLog.forTable(new Configuration(), "file:///path/to/table");

Snapshot snapshot = deltaLog.update();
long newCurrentVersion = snapshot.getVersion();

Iterator<VersionLog> changes = deltaLog.getChanges(lastSeenVersion, newCurrentVersion,
true);

while (changes.hasNext()) {
       VersionLog versionLog = changes.next();
       long version = versionLog.getVersion();

      // create splits only for new versions
       if (version > newCurrentVersion) {
              FileSourceSplit newSplits = buildSplitsForANewFiles(versionLog);
              alreadyProcessedFiles.addAll(extractPaths(newSplits));
       }
}

//update currentVersion so we will get changes starting from new value next time
lastSeenVersion = newCurrentVersion;
```

In this solution we are proposing to enhance DeltaLog API with a overloaded method

deltaLog.getChanges(startVersion, endVersion, failOnDataLoss).

This new method will provide changes between two provided versions. The *endVersion* and *startVersion* parameters must be inclusive.

Pros:

- API user can get changes in for range of versions
- API users don’t need to keep the highest version from changes since we have changes for a requested version range.

Cons:

- Work needed on Delta Standalone API
- API User has to create a new snapshot to get the current version although the snapshot object is not used later.

Option 2b - enhance existing DeltaLog API

** This option doesn't improve correctness or performance. It suggests a potentially less error-prone API for the Delta Standalone :: DeltaLog class **

The small variation, that might simplify this a little bit more could look like this:

```
//recreating deltaLog for monitored table
DeltaLog deltaLog = DeltaLog.forTable(new Configuration(), "file:///path/to/table");

//get the newest version.
long newCurrentVersion = deltaLog.getActuallVersion();

Iterator<VersionLog> changes = deltaLog.getChanges(lastSeenVersion, newCurrentVersion,
true);

while (changes.hasNext()) {
       VersionLog versionLog = changes.next();
       long version = versionLog.getVersion();

      // create splits only for new versions
       if (version > newCurrentVersion) {
              FileSourceSplit newSplits = buildSplitsForANewFiles(versionLog);
              alreadyProcessedFiles.addAll(extractPaths(newSplits));
       }
}

//update currentVersion so we will get changes starting from new value next time
lastSeenVersion = newCurrentVersion;
```

Pros:

- API user can get changes in for range of versions
- API users don't need to keep the highest version from changes since we have changes for a requested version range.
- API users can get the newest version of Delta Table without creating a new snapshot.

Cons:

- Work needed on Delta Standalone API

In this option, we do not create (at least not directly) a new snapshot. We simply ask DeltaLog for the actual version for the monitored table and we use it to get changes between previous and current versions.

Maybe getting the current version for Delta Table would not require creating a new snapshot even under the hood, which would be even more beneficial.

Design Decision 3 - How to handle RemoveFile in VersionLog's Actions List in case of Continuous mode

The VersionLog object returned inside of the iterator from *DeltaLog::getChanges* contains a list of actions that happened for this table starting from a particular version. This list can contain various Action types. Connector will process the AddFile actions and will ignore Commit and Metadata actions.

In case of RemoveFile action, the connector will do an additional check for the flag "dataChange". If this flag is set to true, it would mean that we do not have only appending changes on this table. In that case we need to stop further processing. RemoveFile actions with flag "dataChange" set to false, can be ignored.

The question is, how Source Connector should react in case there were other actions than just *AddFile*. In that case we would not be able to guarantee that we are not processing already processed data.

Option 1 - Throw an exception (Preferred)

In case the *RemoveFile* action was present, the connector can throw an exception because we cannot guarantee that we are not processing already processed data.

However in case of exception, with default settings the flink job will be restarted and will try to reprocess data. This will result in the same issue. "For details about Restart Strategy please go to "Flink Exception Handling" section of this document.

Pros:

- No Extra development other than throw an Exception is needed

Cons:

- Depending on the Restart Strategy, Job may loop up to the Restart Limit since after each restart we will get the same issue.

Option 2 - Log an error and switch into idle mode.

Source connector could log appropriate error messages to system logs and put itself in Idle state, meaning it will stop processing new records and shutting down work discovery as well.

This will prevent cluster from restarting the job many times, up to therrestart limit of used Restart Strategy. For Restart Strategy details please go to the "Flink Exception Handling" section.

The con of this solution would be the "visibility" meaning that the user might not be aware initially that something went wrong for a particular source. Since the state of the Source Task from the Flink point of view would be still *"Running"*. To mitigate this, we could implement a custom metric for Source Connector that would manifest the internal state of the Source allowing the end user to configure the alarm if Source is in Idle State.

Pros:

- System will stop producing errors in case of Restart Strategy with high Restart Limit

Cons:

- Need extra development Work
- Had to implement messaging schema between SplitEnumerator and Readers
- Would Have to extend existing FileReader to handle IDLE signal
- Would have to add a monitoring metric to the DeltaSource that will indicate Idle state.

Design Decision 4 - Reading initial data from DeltaTable

The DeltaTable can have millions of parquet files underneath hence the array returned from snapshot.getAllFiles(); can be too big to process it at once due to memory limits.

Preferable way would be to consume Delta's AddFile Iterator in batches.
This could be achieved by using `snapshot.scan().getFiles()` which returns an `Iterator<AddFile>`

This will run in a monitor thread in SplitSourceEnumerator providing new batches and skipping already processed data.

However, the current iterator that is returned from `snapshot.scan().getFiles()` does not have any guarantees for iteration order which is crucial for scenarios when we will be recovering from a checkpoint and we would need to skip already processed files.

Option 1 - Read all data at once (Chosen for now)
This can be already achieved using the existing Delta API without any extra work needed.
Flink Delta Source will use List<AddFile> allFiles = snapshot.getAllFiles(); to get fill table content

and will read it at once. All AddFile items will be converted to DeltaSourceSplit objects and passed to File Readers per reader request.

Pros:

1. Easy to implement on the Flink Connector side.
2. API exists on the DeltaLog side.

Cons:

1. Not suitable for large tables. This will create OOM very quickly for bigger tables.

Option 2 - Read data in chunks (Preferred but currently not supported by Delta API)

Use Delta API based on Delta API snapshot.scan.getFiles() which returns an Iterator<AddFile>. During checkpoint recovery we will skip N already processed files which would be faster than looking up N AddFile path in SET with already processed files. The number N will be checkpointed in Enumerator state.

To make this work, the Interator has to guarantee the same order across every iteration and every Iterator instance.

Prerequisite:
Delta API guarantees order of files for `snapshot.scan.getFiles()` or for snapshot.scan.getFiles()

Pros:

1. Allows to use more efficient index base skip instead path based lookup skip for already processed files.
2. We do not need to maintain a SET of already processed files in the memory.
3. Will not cause OOM for large tables.
4. Suitable both for large and small tables.

Cons:

1. Current Delta API does not have the order guarantee for Iterator.
2. Slightly more complex than reading everything at once

Options to solve order guarantee for Delta scan Interator:

Option 2a - Enhance Delta Standalone

Iterator order guarantee is provided by Delta Standalone.

Pros:

1. Can use Delta internal API/mechanism that are not exposed in the public API.
2. Can be used by other projects that will work with large tables through Delta Standalone.

Cons:

1. Requires changes on Delta Standalone and new version.

Option 2b - Implement sorting algorithm on the Connector side.

In this option, Delta Connector will sort entries while iterating through them.
We would need to assess if this is possible with currently available data provided by AddFile object fields, and Table MetaData information. It still may turn out that some changes on Delta would be required. The sorting algorithm would have to have a hard limit on used memory to prevent OOM. This means that we would not be able to keep all items in the memory, hence some sort of temporary filessolution and serialization/deserialization would have to be implemented.

Pros:

- No need to change Delta Standalone implementation

Cons:

- Created solution for sorting would not be available for other projects.
- Cannot use delta internal API, only exposed API which may not be sufficient to make this algorithm optimized.

Performance Optimizations

In this chapter we describe possible performance optimizations to the Flink Delta Source connector that could be implemented in future iterations. The Development effort is estimated using "T-shirt size" estimates: S, M, L, XL.

Reading Initial data in chunks

Motivation:
As described in "Reading the Initial Data" chapter, we ideally would like to read initial table data in chunks, since size of List<AddFile> returned from snapshot.getAllFilles() method can easily create OOM Error (Out Of Memory) due to fact that we Delta Table can have millions of files underneath.

Possible Solution

Currently non available due the fact that the current iterator that is returned from `snapshot.scan().getFiles()` does not have any guarantees for iteration order which is crucial for scenarios when we will be recovering from a checkpoint and we would need to skip already processed files. (Design Decision 5).

Development Effort on Connector side
Development Effort - **M**
Need to wrap already existing logic for reading initial data into "chunking" decorator.
Keep track of processed paths (index based instead file path based). Change SplitEnumerator state to track index instead of Set of processed paths.

Optimizing in PartitionFieldExtractor

As described in the "Partitions" chapter, the custom logic for extraction value for each partition column can be defined using the "PartitionFieldExtractor" functional interface.
This function is executed for every partition column and every row.
This is not the most efficient way, since values for each partition separately in scope of one Split (one AddFile) will remain the same, hence there is no need to calculate

The logic for extracting partition value for Delta partitions has two steps:

- Lookup the partition String value from the Partition Map that is added to DeltaSplit.
- Convert it to desired, output type

Possible Solution 1 - Cache

Add Cache for converted value instead of converting the value again every time.This would require to make PartitionFieldExtractor stateful, since its instance would have to maintain the converted value cache. As a result we would still need to do a HashMap lookup. The type conversion lookup would be done once for every split, since each split (AddFIle) can have different values for partitions.

Pros:

- Possible performance gain by limiting number of type conversions

Cons:

-
- Need to recalculate the value for every new AddFile since each AddFile may have different partition values..

Development Effort on Connector side

Development Effort - **S**

Need to implement cache in *"PartitionFieldExtractor"* and consider parametrizing its properties.

Possible Solution 2 - Contextualization (PREFERED)

We could use some sort of contextualization for *"PartitionFieldExtractor"*. We know that in the scope of one Split (one AddFile) that every partition column will have its own, constant individual value for every record that is described by this AddFile (Split) and we know how many partition columns there will be, since those are specified by end user during source definition. With this, instead of having one *"PartitionFieldExtractor"* for all partitions, we could have one *"PartitionFieldExtractor"* for every partition column. The "PartitionFieldExtractor" will keep the calculated partition value as "PartitionFieldExtractor" inner field and recalculate it whenever there is a new split processed.

This solution is **theoretically** possible but it would require implementing our own *ParquetColumnarRowInputFormat's* static *createPartitionedFormat* method in order to be able to create *"PartitionFieldExtractor* " for every partition column.

Development Effort on Connector side

Development Effort - **L**

Need to implement a custom factory method for creating a Partitioned Format that will provide a new Instance of *"PartitionFieldExtractor"* for each partition column. The *"PartitionFieldExtractor"* would have to "reset" its value for every new Split Id.

Pros:

- Possibly better performance gain than Solution 1.
- No HashMap Lookup but only one String comparison instead (comparing split ID)

Cons:

- Needs more effort to implement than Solution 1.
- Theoretically possible, but need more investigation and PoC.
- Recalculate the contextualized value for every new AddFile/Split

Appendix A - Flink Exception Handling

Any uncaught exception thrown from user code (custom operator, connector 3rd library) will cause a task failure in response to which Flink needs to restart the failed task and other affected tasks to recover the job to a normal state. Restart strategies and failover strategies are used to control the task restarting. Restart strategies decide whether and when the failed/affected tasks can be restarted. Failover strategies decide which tasks should be restarted to recover the job.

The cluster can be started with a default restart strategy which is always used when no job specific restart strategy has been defined. In case that the job is submitted with a restart strategy, its strategy overrides the cluster's default setting.

By default, if checkpointing is not enabled, the "no restart" strategy is used. If checkpointing is activated and the restart strategy has not been configured, the fixed-delay strategy is used with Integer.MAX_VALUE restart attempts.

With this strategy, Flink will restart job N times, reprocessing all data from the previous checkpoint. In case of code errors, such as bugs, unexpected input data or any other exceptions, Job will be restarted over and over again.

The common pattern in such situations is making sure that every exception is caught in the user code and handling it programatically, according to business requirements, for example sending a custom alert message to dedicated Sink.

Details about Flink's Restart Strategies can be found here - https://nightlies.apache.org/flink/flink-docs-release-1.13/docs/dev/execution/task_failure_recovery/

Appendix B - Creating DeltaSource instance

Bounded Mode

```
DeltaSource<RowData> deltaSource = DeltaSourceBuilder.builder()
            .tablePath(Path.fromLocalFile(new File(nonPartitionedTablePath)))
            .columnNames(new String[]{"name", "surname", "age"})
            .columnTypes(new LogicalType[]{new CharType(), new CharType(),
               new IntType()})
            .configuration(DeltaTestUtils.getHadoopConf())
            .build();
```

Bounded Mode with Partitions

```
DeltaSource<RowData> deltaSource = DeltaSourceBuilder.builder()
            .tablePath(Path.fromLocalFile(new File(partitionedTablePath)))
            .columnNames(new String[]{"name", "surname", "age", "col2"})
            .columnTypes(
                new LogicalType[]{new CharType(), new CharType(), new IntType(), new
CharType()})
            .configuration(DeltaTestUtils.getHadoopConf())
            .partitions(Arrays.asList("col1", "col2"))
            .build();
```

Continuous Mode

```
 DeltaSource<RowData> deltaSource = DeltaSourceBuilder.builder()
            .tablePath(Path.fromLocalFile(new File(nonPartitionedTablePath)))
            .columnNames(new String[]{"name", "surname", "age"})
            .columnTypes(new LogicalType[]{new CharType(), new CharType(),
               new IntType()})
            .configuration(DeltaTestUtils.getHadoopConf())
            .continuous()
            .build();
```

Defining Source Additional Options

```
       DeltaSource<RowData> deltaSource = DeltaSourceBuilder.builder()
            .tablePath(Path.fromLocalFile(new File(nonPartitionedTablePath)))
            .columnNames(new String[]{"name", "surname", "age"})
            .columnTypes(new LogicalType[]{new CharType(), new CharType(), new
IntType()})
            .hadoopConfiguration(DeltaTestUtils.getHadoopConf())
            .option("parquetBatchSize", 1024)
            .option("ignoreChanges", true)
            .build();
```

Appendix C - Configuration Options

Configuration options can be set through Delta Source Builder API for StreamApi use case.

Additionally, values for those parameters can be provided as Application/Job parameters - https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/application_parameters/#handling-application-parameters

Public Configuration Options

Below table presents a public configuration options that will be publicly documented and can be changed by the user during source creation.

Config option (tbd)	Description	Default Value	Mode
versionAsOf	Read snapshot for specified version. Will throw an exception if provided value is outside the range of available versions. This option is mutually exclusive with the timestampAsOf option.	No default value. If the option is not specified, then the connector will read the head version of Delta Table.	Bounded mode only- default value or specified by the user.
timestampAsOf	Travel back to the latest snapshot that was generated at or before the given timestamp. This option is mutually exclusive with the versionAsOf option.	No default value. If the option is not specified, the connector will read the head version of Delta Table.	Bounded mode only - specified by the user.

startingVersion	The Delta Lake version to start from. All table changes starting from this version (inclusive) will be read by the streaming source. To return only the latest changes, specify “latest” This option is mutually exclusive with the startingTimestamp option.	No default value. If the option is not specified, the connector will read the entire head version of Delta Table and will start monitoring for changes.	Continuous mode only.
startingTimestamp	The timestamp to start from. All table changes committed at or after the timestamp (inclusive) will be read by the streaming source. A timestamp string. For example, "2019-01-01T00:00:00.000Z". A date string. For example, "2019-01-01". This option is mutually exclusive with the startingVersion option.	No default value. If the option is not specified, the connector will read the entire head version of Delta Table and will start monitoring for changes.	Continuous mode only.
updateCheckIntervalMillis	A time interval value used for periodical	5000	Continuous mode only.

	table update checks. Value in milliseconds.		
ignoreDeletes	If “true” and a particular version had only Deletes (no other actions) then do not throw an exception on RemoveFile. regardless of the “*RemoveFile ::dataChange*” flag.	false	Continuous mode only.
ignoreChanges	if “true” and a particular version had a combination of deletes and other actions, then do not throw an exception on RemoveFile regardless of the “*RemoveFile ::dataChange*” flag.	false	Continuous mode only.

Non Public Configuration Options

Below table presents configuration options that are not hard coded and can be changed during Source creation. Those options however are not publicly documented.

Config Option Name	Description	Default Value	Mode
parquetBatchSize	Number of rows read per batch by Parquet Reader from Parquet file.	2048	Bounded and Continuous modes.
updateCheckDelayMillis	Delay time in milliseconds for first table change check.	1000	Continuous mode only.

The *parquetBatchSize* option has an impact on performance. This value describes how many rows are put into one vector. Vectors are used internally in Flink for performance reasons to enable faster execution on batches. Too big value however can cause OOM errors.

The *updateCheckDelayMillis* option might be useful whenever there is a need to delay the first Table check due to various reasons such as initialization of external systems or connections.

Hardcoded parameters

Parameter Name	Description	Hardcoded Value	Mode
parquetCaseSensitive	Denotes whether to use case sensitive Map the field/column names from parquet and match them to columns in Flink.	true	Bounded and Continuous modes.
parquetUtcTimestamp	Denotes whether timestamps should be represented as SQL UTC timestamps.	true	Bounded and Continuous modes.

The Impact of updateCheckIntervalMillis **Option**

The goal here is to show what would be the implications of using an extreme updateCheckIntervalMillis value.

The rule of thumb for this option is to favoure lower values for Tables with high rate of changes, where longer intervals will usually work well for Tables with lower rate of changes.

updateCheckIntervalMillis = 1s
Pros:

- Changes are visible/consumed very quickly.
- Flink's Source Coordinator thread has to convert a smaller number of versions to splits per cycle meaning it will not be blocked for a long time. Blocking this thread can lead to checkpoint delay.

Cons:

- Potential congestion on Delta Table or other resources caused by calling deltaLog.getChanges(...) very often. (Listing cost)

updateCheckIntervalMillis = 1h
Pros:

- Limited potential congestion on Delta Table from calling deltaLog.getChanges(...) not often.

Cons:

- Changes from Delta Table will be visible after a longer time, not suitable for tables with high change rate.
- Can cause OOM if there were a lot of changes during the last interval check.