Data Source V2 API Improvement

Motivation

Data source V2 is out for a while, see the SPIP here. We have already migrated most of the built-in streaming data sources to the V2 API, and the file source migration is in progress. We also started some new features for data source v2, like catalog support, standard DDL logical plans, etc. In the meanwhile, we found several problems and want to address them before we stabilize the V2 API.

Scan execution order is not obvious

In the V2 read API we use mixin interfaces for everything: operator pushdown, column pruning, columnar scan, information reporting, etc. This is very flexible but makes data source developers confused about what is the completed scan process, and what's the execution order for these mixin interfaces. For example, from the API it's hard to tell that information reporting is conducted after the operator pushdown. Currently, the V2 API heavily depends on the documents to explain the scan process and the execution order of these mixin interfaces.

```
*
* There are mainly 3 kinds of query optimizations:
*   1. Operators push-down. E.g., filter push-down, required columns push-down(aka column
*      pruning), etc. Names of these interfaces start with `SupportsPushDown`.
*   2. Information Reporting. E.g., statistics reporting, ordering reporting, etc.
*      Names of these interfaces start with `SupportsReporting`.
*   3. Special scans. E.g, columnar scan, unsafe row scan, etc.
*      Names of these interfaces start with `SupportsScan`. Note that a reader should only
*      implement at most one of the special scans, if more than one special scans are implemented,
*      only one of them would be respected, according to the priority list from high to low:
*      {@link SupportsScanColumnarBatch}, {@link SupportsScanUnsafeRow}.
*
```

A good API design should be self-contained. The API itself should tell the basic assumptions to the developers/implementations.

Splitting and reading data partitions should be independent

Currently V2 implementation needs to define how to split and read the data partitions with one API: `DataSourceReader#planInputPartitions`. When migrating file sources to V2 API, we have to create a `FilePartition` class to represent a pure data partition and implement reading function elsewhere. The reason is: 1) file source has to support both row-based and columnar scan, and share the same partition splitting logic. 2) data partition splitting and reading are both non-trivial, it's more maintainable to put these 2 parts into different classes. It's also common in other systems that separate partition splitting and reading in the API, e.g. Hadoop's InputFormat , Presto's Connector .

Columnar Scan API should not be a mixin trait

Technically columnar scan API doesn't mixin new stuff, but replace the row-based scan. This brings several problems:
1. The interface definition is hacky. We need to first override the existing `planInputPartitions` to throw an exception, and then add a new method for different data format.
2. When a data source implements multiple special scan mixin traits, we have to document which trait is effective. Data source can't output different formats at the same time, but we don't prevent it at the API level.

Streaming API doesn't play well with the batch API

Streaming API doesn't support columnar scan and operator pushdown, so it's hard to migrate file source as file source supports both batch and streaming. We tried to resolve it before. The new proposal will cover that design.

Interface name is confusing

`DataSourceReader` is responsible for splitting input data, handle operator pushdown, report statistics, create actual readers, etc. So technically it's not just a "reader". And the API for read/write is inconsistent: In the read API we have `InputPartition` and `InputPartitionReader`, in the write API we have `DataWriterFactory` and `DataWriter`.

Non goals

1. Create a stable and efficient row representation. Currently we use InternalRow, which is efficient but not stable.
2. Define residual filter expressions per-partition. For the API, I think we can define a interface like:

```
interface PerPartitionResidualFiltersReporter {
   Filter[] getResidualFilters(InputPartition partition);
}
```

But we also need to update Spark side to support it.

Proposal

The following code is a rough prototype, the naming and details can be further revised.

The main idea is, separate an interface for operator pushdown which needs mutable states. Then all other interfaces can be stateless.

Compared to the previous API, this new proposal clearly defines the scan evaluation order: Spark would do the operator pushdown first, and then use the pushdown result to do data splitting, statistics reporting and the actual scan. The last 3 steps do not care about the order: they have individual methods and are totally orthogonal.

Data splitting and reading are separated now: `planInputPartitions(ScanConfig)` is responsible for data splitting and `PartitionReaderFactory` is responsible for data reading.

And now we put all the scan APIs in `PartitionReaderFactory`, there is no special scan mixin trait any more.

The `ReadSupport` is renamed to `BatchReadSupportProvider` and the `DataSourceReader` is renamed to `BatchReadSupport`. And now we have `InputPartition`, `PartitionReaderFactory`, `PartitionReader` vs `DataWriterFactory`, `DataWriter`. The new names are more consistent and can better reflect the actual differences between read and write.

The abstraction of data source V2:
A data source connector can be loaded at many places at the same time, e.g. one user is running `spark.read.option(...).format("xyz").load()` while there can be another user doing the same thing but with different options. When loading the data source connector, one or more scans are performed by Spark, e.g. when loading a streaming table, Spark will perform one scan for each micro-batch.

The API design follows this abstraction:

- `ReadSupport` usually represents a table/path/stream, an instance of `ReadSupport` will be used when Spark loading a table/path/stream. The instances can be different or can be same every time Spark asks the provider for it.
- When Spark performs a scan, Spark will ask the `ReadSupport` instance to create a

`ScanConfig` instance for this scan. If it's a stream query, the `ScanConfig` instance will be created with the offsets for this scan.

We also need an provider interface to provide `ReadSupport` instance to Spark. Currently it's `ReadSupportProvider`, which will be instantiated via reflection every time `DataFrameReader#load` is called. In the future when we finish the catalog integration, the catalog is responsible to do that.

The abstraction and design of write side is similar to read side.

Things that are specific to a single scan operation should take `ScanConfig` as input:

- `InputPartition[] planInputPartitions(ScanConfig config);`
- `PartitionReaderFactory createReaderFactory(ScanConfig config);`
- `Statistics estimateStatistics(ScanConfig config);`
- ...

Things that belong to the "Loading" should not take `ScanConfig` as input

- Streaming offset management. Offset itself is specific to a single scan, so in the streaming APIs, `ScanConfig` is created with offsets. Spark works with the streaming source to decide to offsets for each scan, so the offset management is cross-scan and belongs to the "Loading".

Read API:

```
public interface BatchReadSupportProvider {
  default BatchReadSupport create(
      StructType, schema,
      DataSourceOptions options) {
    throw new UnsupportedOperationException
  }

  BatchReadSupport create(DataSourceOptions options);
}

public interface MicroBatchReadSupportProvider {
  MicroBatchReadSupport create(
    Optional<StructType> schema,
    DataSourceOptions options,
    String checkpointLocation);
}

public interface ContinuousReadSupportProvider {
  ContinuousReadSupport create(
```

```
    Optional<StructType> schema,
    DataSourceOptions options,
    String checkpointLocation);
}

// package private
interface ReadSupport {
  StructType fullSchema();

  InputPartition[] planInputPartitions(ScanConfig config);

  PartitionReaderFactory createReaderFactory(ScanConfig config);
}

public interface BatchReadSupport extends ReadSupport {
  ScanConfigBuilder newScanConfigBuilder();
}

public interface StreamingReadSupport extends ReadSupport {
  // some common streaming methods.
  Offset deserializeOffset(String json);
  Offset initialOffset();
  void commit(Offset end);
  void stop();
}

public interface MicroBatchReadSupport extends StreamingReadSupport {
  // created for each epoch
  ScanConfigBuilder newScanConfigBuilder(Offset start, Offset end);

  Offset latestOffset();
}

public interface ContinuousReadSupport extends StreamingReadSupport {
  ScanConfigBuilder newScanConfigBuiler(Offset start);

  @Override
  ContinuousPartitionReaderFactory createReaderFactory(ScanConfig config);

  boolean needsReconfiguration();
}
```

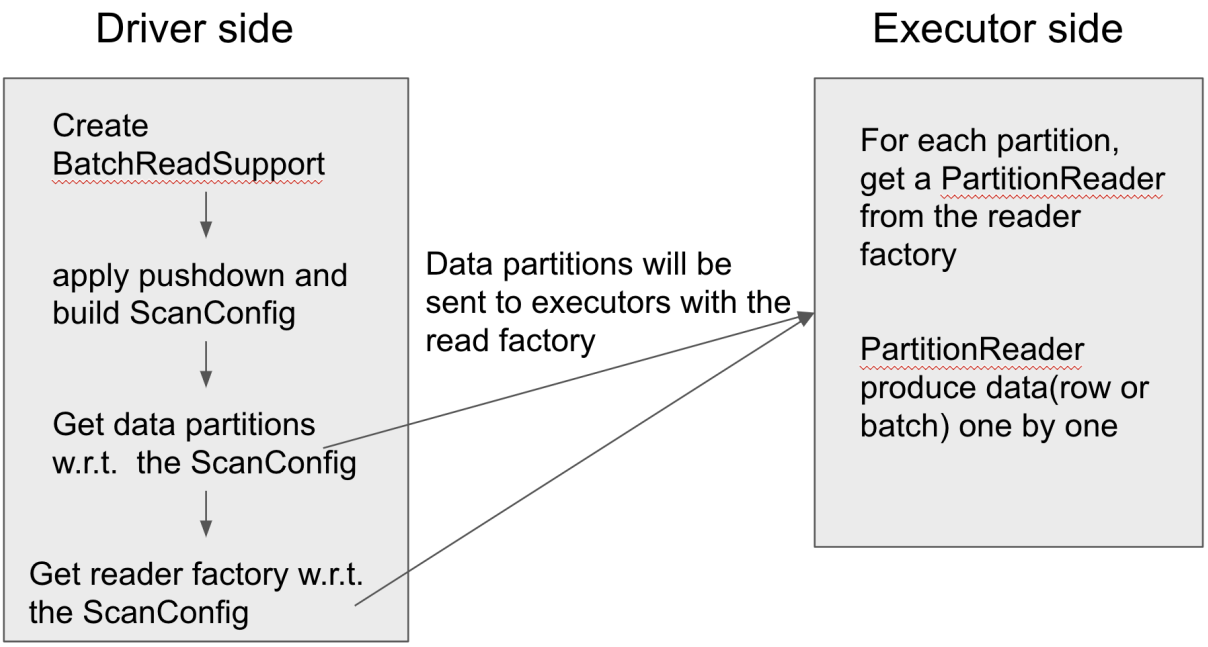
```
// works for both batch and streaming APIs
public interface SupportsReportStatistics extends ReadSupport {
  Statistics estimateStatistics(ScanConfig config);
}

// An interface that can mixin these SupportsPushDownXXX interfaces.
// should be mutable.
public interface ScanConfigBuilder {
  ScanConfig build();
}

// carries scan specific information like pushdown result and streaming
// offsets. Should be immutable.
public interface ScanConfig {}

public interface InputPartition extends Serializable {
   default String[] preferredLocations() { return new String[0]; }
}

public interface PartitionReaderFactory extends Serializable {
   PartitionReader<InternalRow> createReader(InputPartition partition);

   PartitionReader<ColumnarBatch> createColumnarReader(
     InputPartition partition);

   boolean supportColumnarReads();
}

public interface ContinuousPartitionReaderFactory
   extends PartitionReaderFactory {
   @Override
   ContinuousPartitionReader<InternalRow> createRowReader(
      InputPartition partition);

   @Override
   ContinuousPartitionReader<ColumnarBatch> createColumnarReader(
      InputPartition partition);
}

public interface ContinuousPartitionReader<T> extends PartitionReader<T> {
   Offset getOffset();
}
```

```
public interface PartitionReader<T> {
   boolean next();
   T get;
}
```

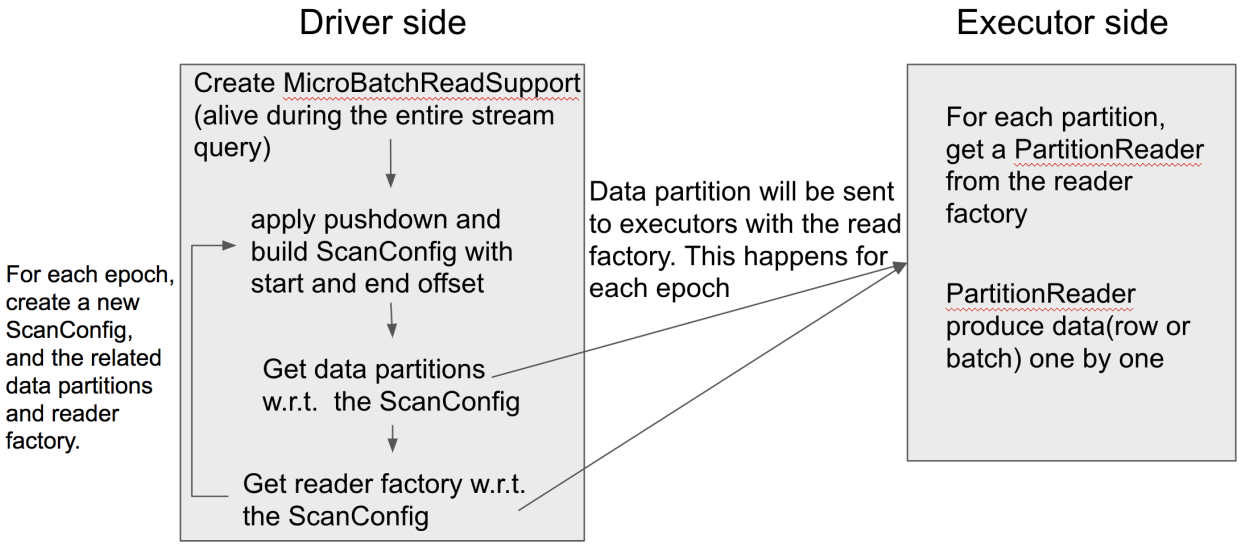
Write API:

```
public interface BatchWriteSupport {
  DataWriterFactory createWriterFactory();

  void commit(WriterCommitMessage[] messages);

  void abort …
  ...
}

public interface StreamWriteSupport {
  StreamDataWriterFactory createWriterFactory();

  void commit(int epochId, WriterCommitMessage[] messages);

  void abort …
  ...

}

public interface DataWriterFactory extends Serializable {
  DataWriter<InternalRow> createWriter(
    int partitionId, int attemptNumber);
}


public interface StreamDataWriterFactory extends Serializable {
  DataWriter<InternalRow> createWriter(
    int epochId, int partitionId, int attemptNumber);
}
```

The new workflow for the batch read path:



For micro-batch streaming:



For continuous streaming: