[StarRocks] New DLA FrameWork

Background and Motivation

In StarRocks we have implemented

1. a full vectorization engine
2. a new CBO optimizer
3. new pipeline based scheduling execution engine

We think these features will also greatly improve analytic queries on external storage systems. Now we have support hive/iceberg/hudi/jdbc, and we have users who want to query data stored on kudu.

However, when we want to support new storage system, we need to do something like below:

- add new scannode/scan operator
- add create external table semantic for metadata synchronization
- add new prune rules
- add new scan implementation rules
- other many changes on fe and be

It's really a tough job for most developers, and creating an external table for a new storage system is also an unfriendly experience for users.

We will introduce a new framework for these reasons. The new framework will named as Connector APIs

Target Personas

StarRocks DLA users, StarRocks developers

Goals

1. Users can make a small amount of modifications in FE/BE to achieve access to new data sources
2. Support read and write APIs
3. Users do not need to create external table in StarRocks to access new data sources
4. The new framework will not introduce performance issues

Non-Goals

1. Make optimizer rules more scalable

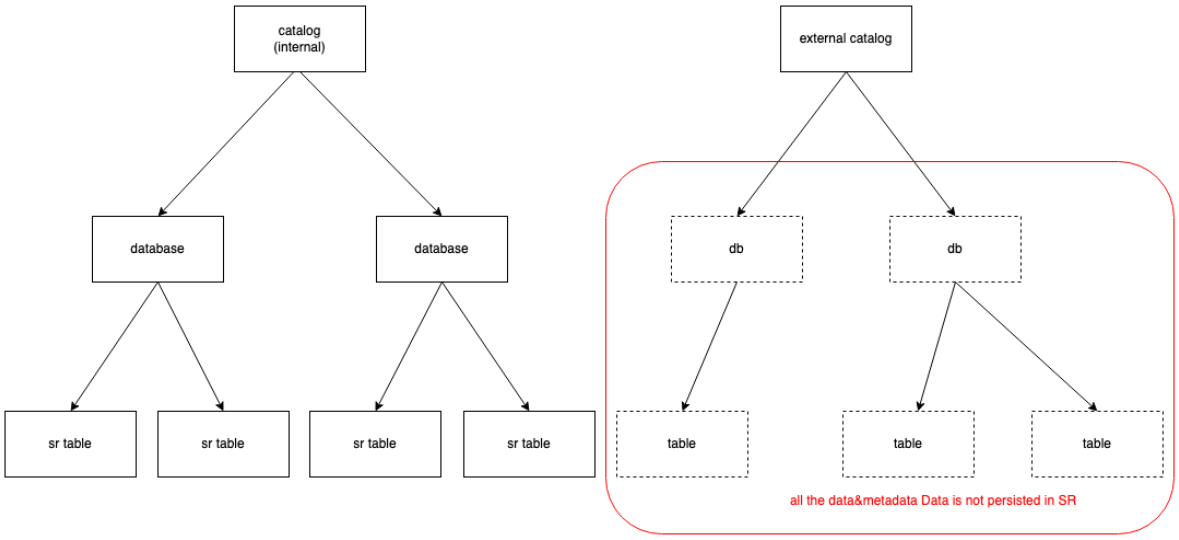
Design Sketch

APIs

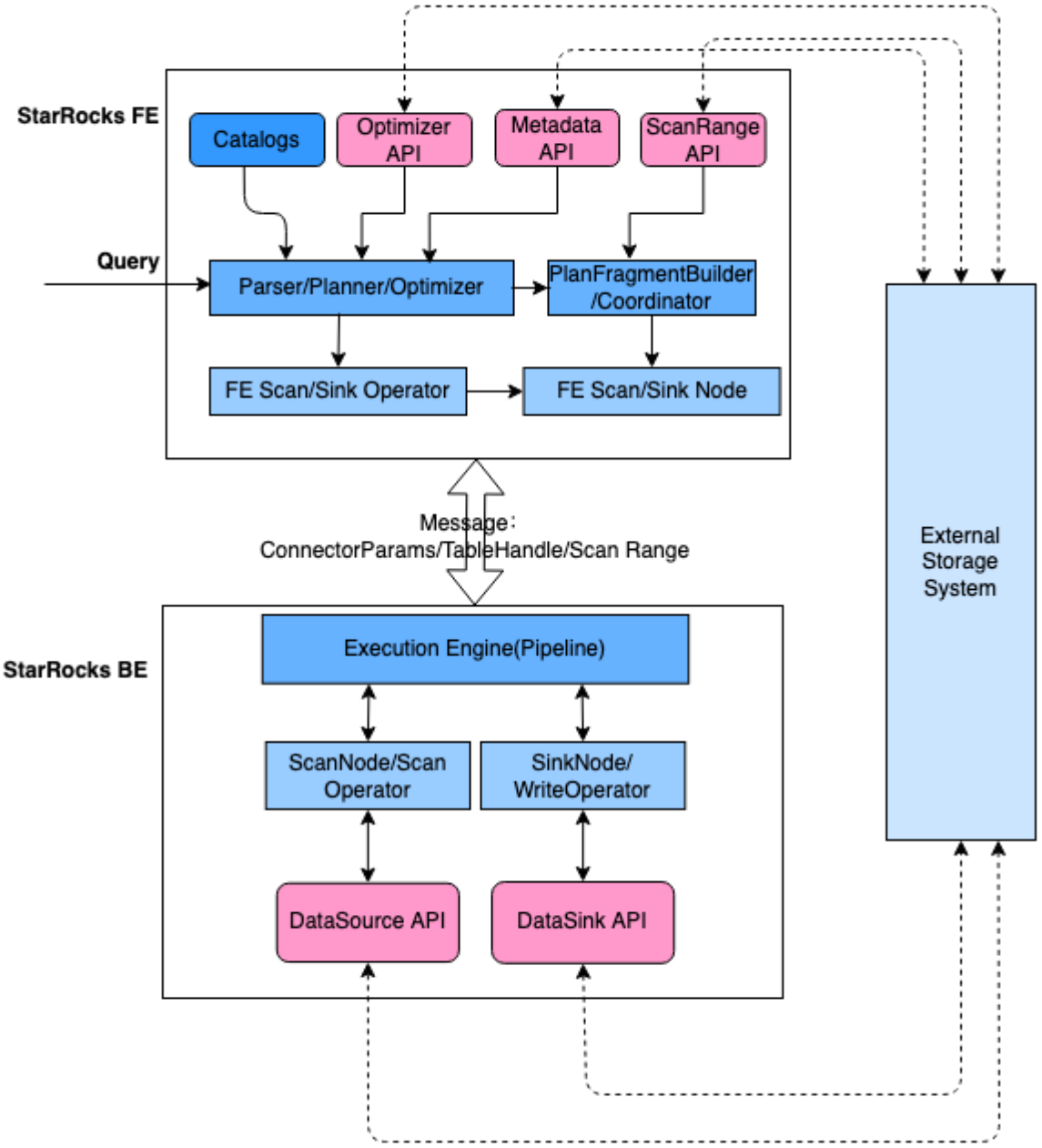
Concepts

Catalog

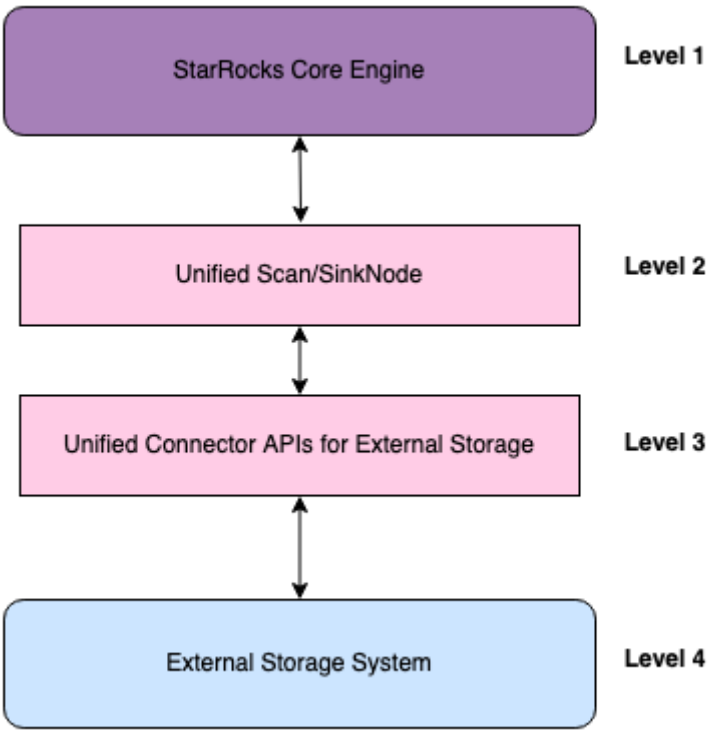
We will introduce a new concept which is named Catalog.The hierarchical relationship is as follows.



Architecture



- In order to support multiple data sources to provide federated query capabilities, the concept of Catalog is introduced
- The abstraction level of the new framework in FE&BE is generally divided into four layers:
    a. The first layer is Planner/Optimzier/Execution Engine
    b. The second layer is Scan/Sink Node
    c. The third layer is Connector APIs
    d. The fourth layer is the External Storage System



- The main purpose of doing this leveling and abstraction is to integrate the first and fourth layers faster and more conveniently. The following is a detailed description of the whole:
    a. In order to better expose the capability abstraction of the fourth layer to the first layer. The Connector mechanism is introduced in the new framework. Corresponding to the third layer mentioned above, Connector mainly contains an abstract collection of APIs. It mainly includes the following parts:
        - FE related Connector API:
            - MetaData API: Defines standard interfaces such as GetTable/GetDatabase/GetTableStatistics. Users provide specific MetaData API implementations when implementing new data sources, and organize specific meta information such as table/database/statistics according to the characteristics of the data source
            - ScanRange API: defines the standard interface of getScanRange, and the user provides a specific implementation to return the scan range of the data source
            - Optimizer API: Defines standard interfaces such as ApplyFilter/ApplyProjection. When implementing a new data source, users can provide a specific Optimizer API to achieve the goal of pushing down the data source-related optimization to the data source without paying attention to specific optimization. details of the rules
        - Connector related Handle:
            - TableHandle: The TableHandle implementation of the corresponding data source is provided by the Connector

implementer, mainly to save the relevant information that needs to be used in the query process
        - BE related Connector API:
            - DataSource API: defines standard interfaces such as getChunk, and the user provides the getChunk implementation of the specific data source
            - DataSink API: standard interfaces such as sendChunk are defined, and the user provides the sendChunk implementation of the specific data source
        - FE&BE communication protocol standard definition
            - Contains Connector Params, ScanRange, Handle information, which is provided by the user with the ability of serialization and deserialization
    b. After abstracting the third layer, for the Core Engine, we also need to abstract a layer similar to the Unified ScanNode, which further decouples the capabilities of the core engine:
        - Explain: The abstraction here is mainly for the existing HdfsScanNode/IcebergScanNode/JdbcScanNode and the future KuduScanNode

RoadMap

Please see https://github.com/StarRocks/starrocks/issues/5062

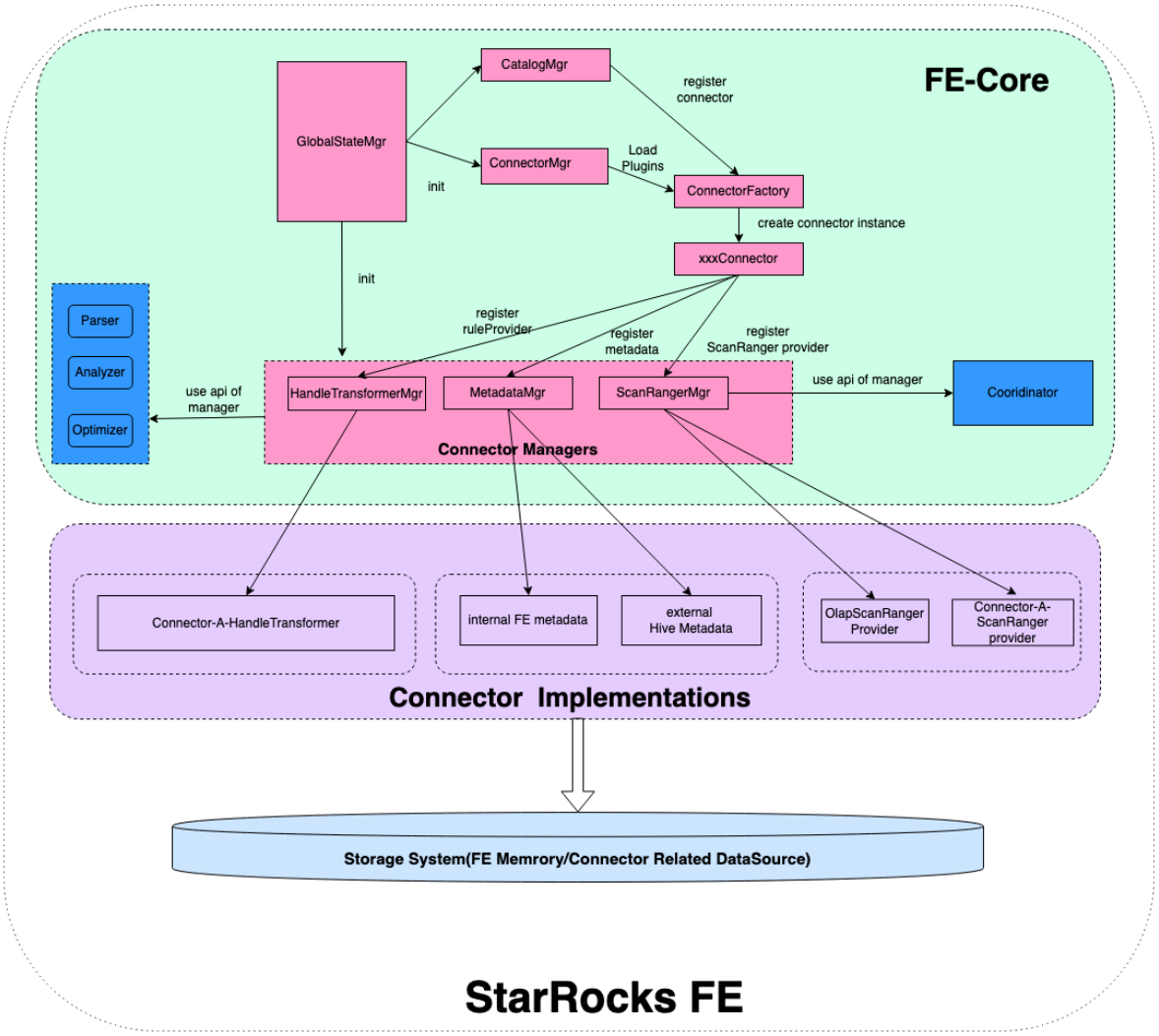
Tradeoffs

- Connector positioning:
    - Connector only provides a collection of APIs, and the specific implementation is provided by Connector developers
    - At the beginning of the design, only the connector-related abstraction on the FE side was considered, because the abstraction and development of BE are relatively high requirements for users, and it is difficult for different data sources to be provided by BE or other systems, but considering the product The unification of the concept is to provide the Api of DataSource/DataSink on the BE side.
- The internal table will eventually need to be merged into the Connector framework. In the long run, this can ensure that kernel-related modifications are unified for internal data sources and external data sources
- The configuration file does not use the file method, but is persisted to FE based on the catalog. Mainly to prepare for the follow-up Serverless
- Abstraction granularity
    - Option 1: Expose ScanNode/ScanOperator/Implementation Rule/Prune Rule and other interfaces to the user, and the user provides the specific implementation
        - The advantage is that there are relatively few invasive modifications at the system level

        - The disadvantage is that the cost of user understanding becomes higher, and at the same time, there will be more redundant code
    - Option 2: Hide ScanNode/Implementation Rule/Prune Rule and provide unified abstract implementation, use Connector API for abstract implementation
        - The advantage is that the user has a low cost of use, and does not need to understand the workflow of the optimizer, etc.
        - The disadvantage is that the adjustment to the existing code structure will be larger
- FE implementation mechanism
    - Using SPI method, add SPI module, encapsulate some interfaces of core into common, avoid circular reference between core and spi; consider that it will be simpler to implement later when connectors can be added.
- BE implementation mechanism
    - Option 1: By dynamically loading so, similar to udf implementation, users do not need to merge their own so packages into master
        - The advantage is that the BE system layer is simple to implement
        - The disadvantage is that the user access threshold is high, and the quality of the so package cannot be guaranteed
    - Option 2: The implementation of the BE side is based on native, in the master branch; provides the option of jni
        - The advantage is that the code quality is controllable
        - The disadvantage is that it takes a certain period of time to merge the near-main branch, and there will be performance loss in jni
    - Option 3: The Connector instance on the BE side does not correspond to one instance per Catalog, but one BE Connector for a class of Catalog.
- Avoid relying on Core implementations, such as OptExpression, when designing APIs. This part mainly considers the user learning cost and the modification iteration of the core part. So we need to have abstraction similar to Handle

Design Details

FE design and implementation

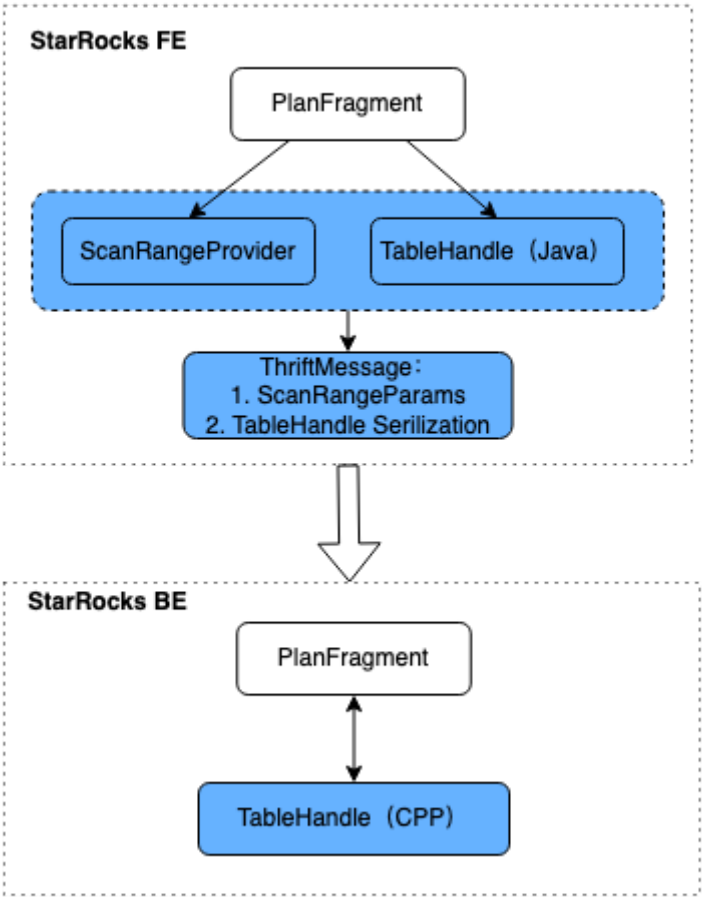


On the whole, it is aimed at the design and implementation level of the FE side. We are divided into FE Core modules and Connector implementations:

- FE Core module
    - GlobalStateManager
        - Manage the global state and split the functions related to the original FE Catalog
    - CatalogManager&ConnectorManager
        - Added for managing Catalog, Catalog will be loaded with Connector Factory when FE is initialized
        - Added the implementation for loading the corresponding Connector Plugin under the classpath in the form of a plugin during initialization
        - CatalogManager&ConnectorManage will initialize Connector Factory during initialization, then generate XXXConnector instances, and register Connector related implementations to the following Connector Managers

    - Connector Managers, mainly including:
        - HandleTransformerManager: global handle for Optimizer to perform push-down optimization
        - MetadataManager: global handle for Optimizer/Analyzer/Planner to obtain table-related information
        - ScanRangeManager: global handle for Coordinator to obtain ScanRange information
- Connector implementation, provided by Connector developers:
    - HandleTransformer: The optimizer pushes down related implementation classes
    - Metadata: The implementation class of the specific connector metadata api
    - ScanRangeProvider: Implementation class for scan range calculation
        - For example, OlapScanRangeProvider is calculated by tablet info; HiveScanRangeProvider is calculated by hive table metadata

Protocol of FE and BE



The communication protocol mainly needs to transmit three aspects of information: TableHandle, ScanRanges, ConnectorParams

- TableHandle&ScanRanges
    - TableHandle is the additional information necessary to query a certain Connector, such as additional filter conditions, etc.; ScanRanges is the fragmentation information of the connector corresponding to the data source

    - TableHandle and ScanRanges are opaque data for Framework, and use std::string to pass, but in actual use, users will definitely tend to use some existing serialization/deserialization facilities such as thrift/protobuf, especially for Static language friendly libraries such as C++
    - Another issue to consider is that because TableHandle may use some basic components in StarRocks such as predicate expressions Expr/TExpr, it is inevitable to use the thrift protocol to reuse the creation of expressions. So it is foreseeable that the user's development method will be:
        - Users include some basic definitions of SR in their own Connector implementation such as Expr.thrift
        - Then use your own serialization and deserialization library, assuming that you use protobuf, and use e.g. my_connector.proto on the FE/BE side
        - On the FE side, the definition of Expr is explicitly serialized to string and placed in my_connector.proto
        - The BE side needs to explicitly deserialize from string to take out the Expr content to construct the Expr
- ConnectorParams
    - Contains the access information of the ConnectorName and the corresponding data source of the Connector, mainly to inform the BE of the access and acquisition methods of the specific DataSource and DataSink

BE design and implementation

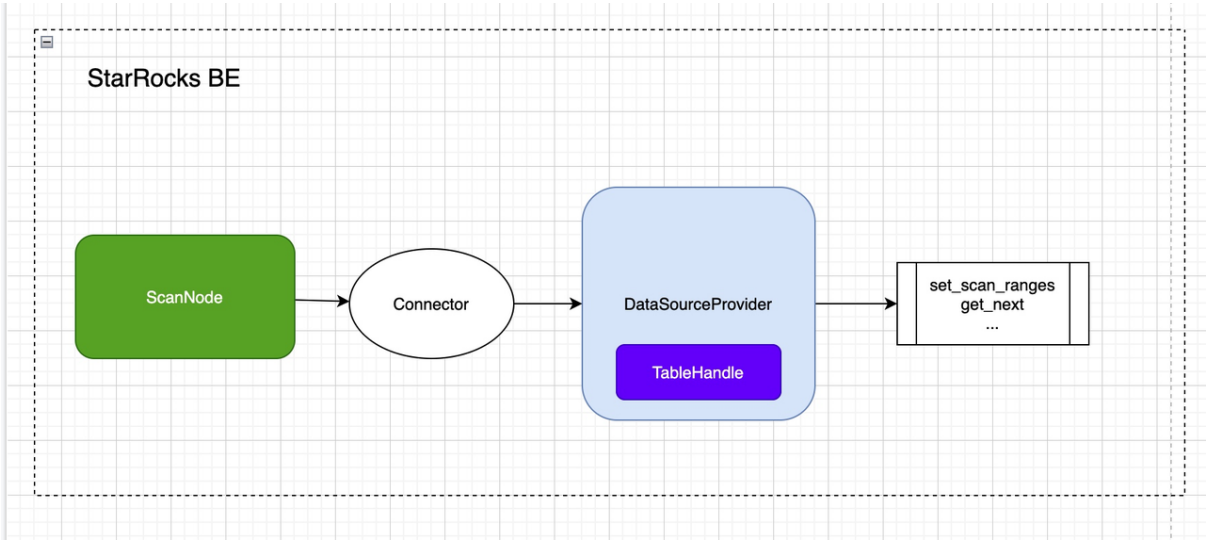
BE PlanNode abstraction

Currently BE distinguishes many types of PlanNode: OlapScanNode, HDFSScanNode, MySqlScanNode etc. These Node implementations are inherited from ScanNode. Mainly provide several methods:

- set_scan_ranges set scan range
- get_next keeps returning Chunk

So we can separate ScanNode and data source, the whole process is as follows:

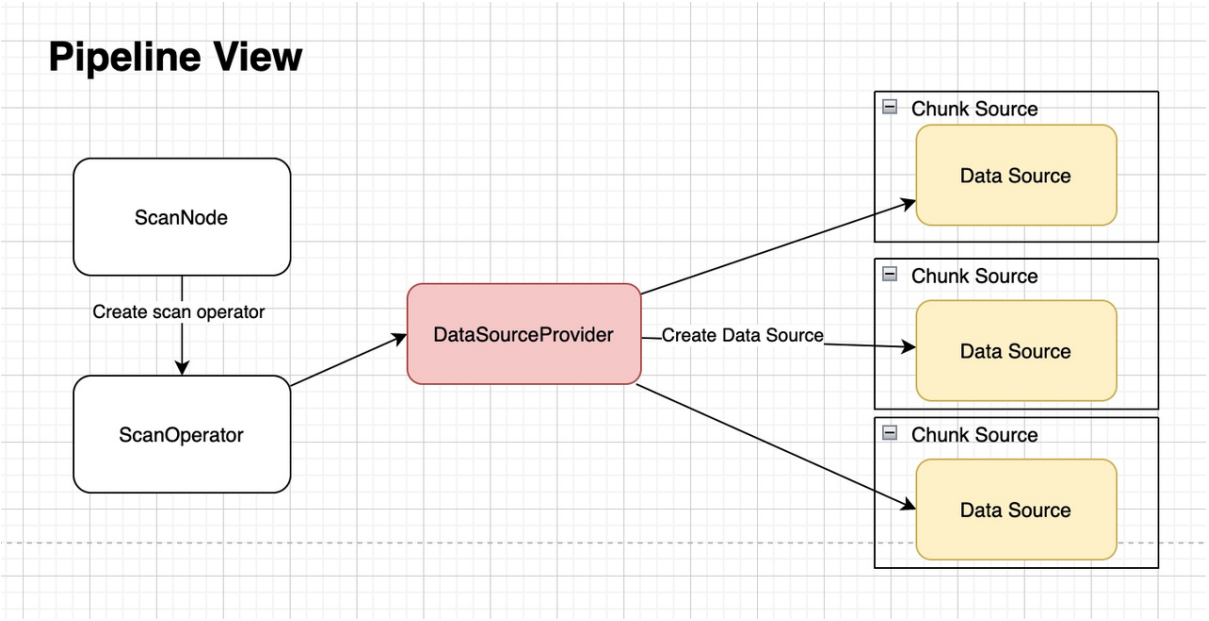
1. ScanNode gets the DataSourceProvider class through Connector
2. Use TableHandle to instantiate the DataSourceProvider
3. DataSourceProvider sets ScanRanges.



BE Connector abstraction

Because the current Connector only focuses on reading, we mainly discuss the DataSourceProvider/DataSource under the Connector.

- Generate PlanNode above, ScanNode already holds DataSourceProvider through Connector
- ScanNode is split into ScanOperator under the Pipeline framework, and ScanOperator can continue to hold this DataSourceProvider
- When ScanOperator needs to create chunk_source according to scan_range, the bottom layer creates DataSource object through DataSourceProvider, and then wraps it in ChunkSource
- DataSouce is oriented to data source developers, while ChunkSource is more oriented to the pipeline framework itself, and the bridging work of this piece is done by DLA Framework.



Reference

1. https://databricks.com/session/apache-spark-data-source-v2
2. https://prestodb.io/docs/current/connector.html

Appendix