FLIP-91: Support SQL Client Gateway

Motivation

The whole conception and architecture of SQL Client are proposed in FLIP-24 which mainly focuses on embedded mode. The goal of this FLIP is to extend FLIP-24 to support gateway mode and REST/JDBC interfaces that could make it easier for users to use Flink.

As we know, the Java Database Connectivity (JDBC) API is the industry standard for database-independent connectivity between the Java programming language and a wide range of databases. JDBC technology allows you to use the Java programming language to exploit "Write Once, Run Anywhere" capabilities for applications that require access to enterprise data.
JDBC interface is designed based on traditional databases, which defines many operations on a bounded dataset. As Flink batch is also bounded data processing, we could use JDBC interface to manipulate flink batch jobs (e.g. submit queries, cancel jobs, retrieve results, etc.). However, Flink streaming is unbounded data processing and some operations of JDBC can not be supported, for example:

1. Statement#executeQuery() will return the affected row count for DML statements, but we can't get the result because the streaming job won't finish.
2. Stream-specific features are not defined in JDBC, like pausing a streaming job.

This FLIP will only involve JDBC on Flink batch, and whether we can add limited support for JDBC on Flink streaming will be discussed in further FLIPs and design documents if needed.

Representational state transfer (REST) is a software architectural style that defines a set of design principles and constraints to be used for creating Web services based on HTTP. It's not an industry standard like JDBC, so we are free to design the APIs to meet our requirements (such as pausing a streaming job). This FLIP will only introduce APIs which meet the requirement of current embedded mode's functionality and basic JDBC implementation.

There are three typical user scenarios which will be supported in this FLIP:

1. Users can connect to the gateway using CLI client(s) and execute batch or streaming jobs.
2. User program(s) can connect to the gateway through REST API and execute batch or streaming jobs. REST API makes it easy and lightweight for user programs to manipulate a Flink cluster. e.g. we could use Shell program (even Postman application) to execute Flink jobs through REST API.
3. Any program which supports JDBC can connect to the gateway and execute batch jobs. The programs can be BI tools (e.g. Tableau), JDBC clients (e.g. Beeline) or user Java programs. These programs will depend on the JDBC implementation on Flink (or named Flink JDBC driver).

Like FLIP-24, this is also an initial minimum viable product for SQL client gateway which could allow users to use Flink with JDBC and REST. More features will be added based on further discussion and feedback from users and contributors.

Public Interfaces

- JDBC interface
- REST APIs

Proposed Changes

General Architecture

FLIP-24 had proposed an architecture of SQL client, we make some additions and adjustments:

1. REST API will be **the unified interface** for connecting different clients to the server. Not only CLI client can be connected to server through REST API, the communication protocol of java.sql.Connection in JDBC can also be REST API.
2. The architecture of embedded and gateway mode will be unified. We treat the embedded mode as a special case of the gateway mode. So that all components and code of gateway mode can be reused. After unification, in gateway mode we will setup a server JVM process and multiple CLI client JVM processes; while in embedded mode, both CLI client and gateway will run in the same single JVM process. CLI client will become a very lightweight component. It only needs to send user queries to the server and display the results from the server.
3. Currently, *Executor* supports basic *Session* functionality in embedded mode. But the current implementation of *Executor* can't meet the requirement of gateway. In

gateway mode, we should consider that if there are too many sessions (clients), the *Executor* may be OOM. To solve this problem, a separate component should be introduced named *SessionManager*, which can store all sessions, limit the max number of living sessions, remove the expired sessions, etc.

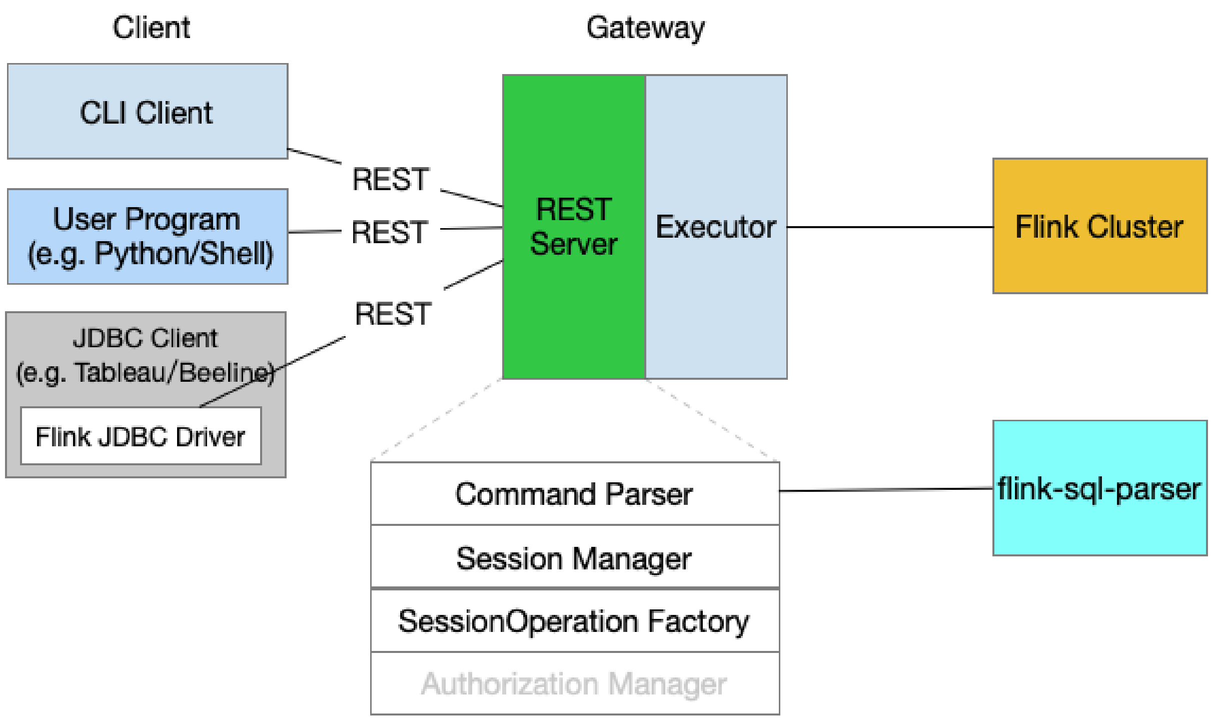
4. A flink job can be canceled through REST API before it ends, so the statement API shall be asynchronous and support retrieving the results or canceling a running job later on (actually, JDBC also requires support for canceling jobs). Currently, *Executor* only supports canceling a SELECT query, not canceling other queries. To solve this, we're introducing Session*Operation*. Session*Operation* is a specific representation of a command. For example *ShowCatalogOperation*, *ExecuteQueryOperation*, etc. It can execute the command asynchronously and provide the execution result. Each Session*Operation* has an identifier which could be used to cancel the operation or get the result. Currently, we only provide asynchronous REST API, and will provide synchronous REST API based on users' feedback in future.
   A Session*Operation* belongs to a specific *Session*, so the operations will be removed when the session is closed or expired. To do all these management works, we will also introduce an Session*OperationManager*. In current design, the executor will be kept and the execution of a Session*Operation* will be delegated to *Executor*.
   In a long-term, most functionality of *Executor* will be replaced with Session*Operation*. If a new feature is needed, a new Session*Operation*, rather than a method in the *Executor* interface, will be added. (This design also matches the Open-Closed design principle as adding methods to the interface requires changes in the old sub-classes of that interface.)

Component Description

- **CLI Client**: client that receives user commands and displays results returned from gateway. ~~The specific planner (old or blink) and the specific execution type (batch or stream) must be given when CLI client is created.~~ (corresponding to scenario 1)
- **User Program**: user program that could manipulate the Flink cluster based on REST API. (corresponding to scenario 2)
- **JDBC Client**: client that could manipulate the Flink cluster based on JDBC interface. (corresponding to scenario 3, like beeline, tableau)
- **Flink JDBC Driver**: the implementation of JDBC interface based on REST API. (it means the communication protocol is REST API) **Only batch is supported now** because JDBC interface is defined for traditional database.
- **REST Server**: a web server could receive commands from clients and execute sql jobs on flink cluster
  - **CommandParser**: parse user commands itself (few commands), or calls flink-sql-parser. One command corresponds to one SessionOperation.
  - **Session**: similar to HTTP Session, which could maintain user identity and store user-specific data during multiple request/response interactions between a client and REST Server. A session preserves:
    - Information about the session itself (session identifier, creation time, time last accessed, etc.)
    - Actions that the session could have (executeStatement, etc)
  - **SessionManager**: manage sessions, including
    - store all sessions
    - create a session when a client connects the server
    - remove a session when a client closes its session
    - periodically remove expired sessions
    - limit the max number of living sessions to avoid OOM in Gateway

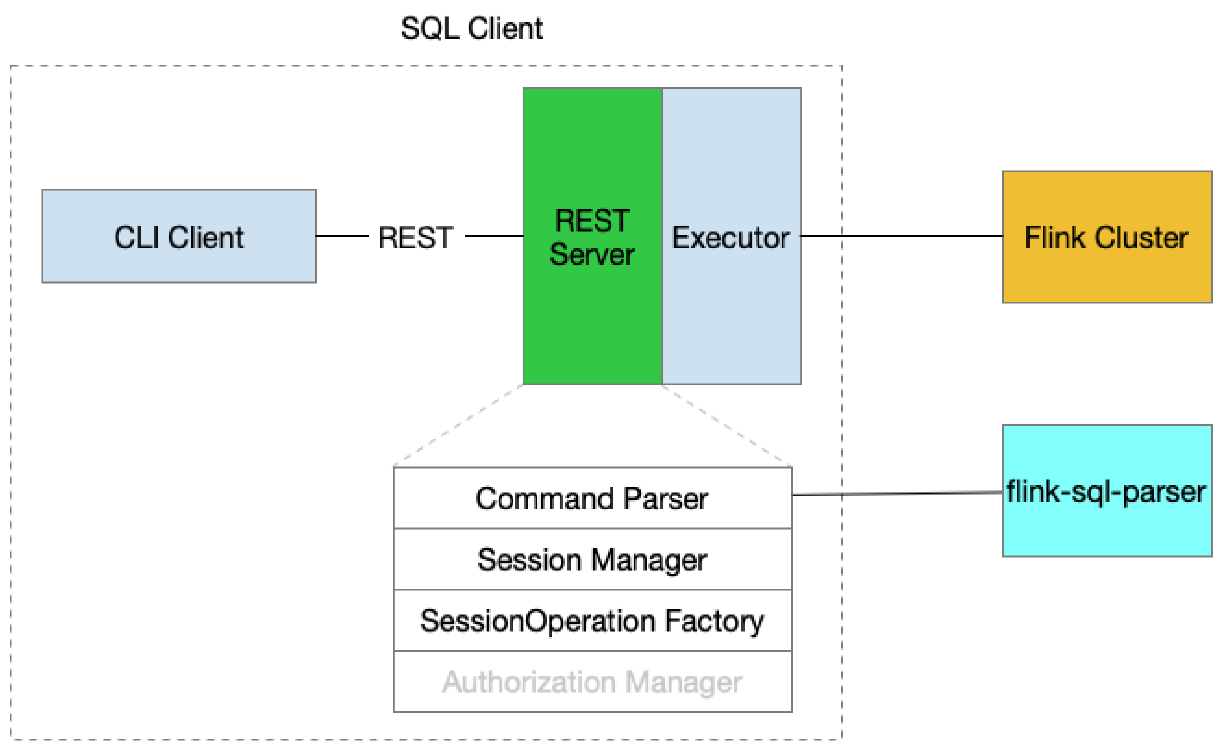
    - **SessionOperation**: a specific representation of a command, which could execute (sync or async) the command and return the executing result. A SessionOperation is created in a specific session, and when the session is expired, the living operations belonging to the session will also be destroyed.
    - **SessionOperationFactory**: create a new session operation based on command
    - AuthorizationManager: manage authorization which is not involved in this design
- **Executor**: a gateway for communicating with Flink and other external systems

Gateway Mode



Embedded Mode

The architecture of embedded mode is similar to gateway mode, and all components could be reused. The only difference is that the processes of client and gateway are different in gateway mode, while in embedded mode they are in the same process.



Gateway

REST Server

Flink has introduced REST server in flink-runtime module to support web monitor and dispatcher. A lot of classes are very common, for example: *RestServerEndpoint*, *RequestBody*, *ResponseBody*, *MessageParameters* and *Router*. However some classes can only be used for flink-runtime, for example: *RestfulGateway, AbstractRestHandler*.
There are two approaches to support REST server on SQL client gateway:

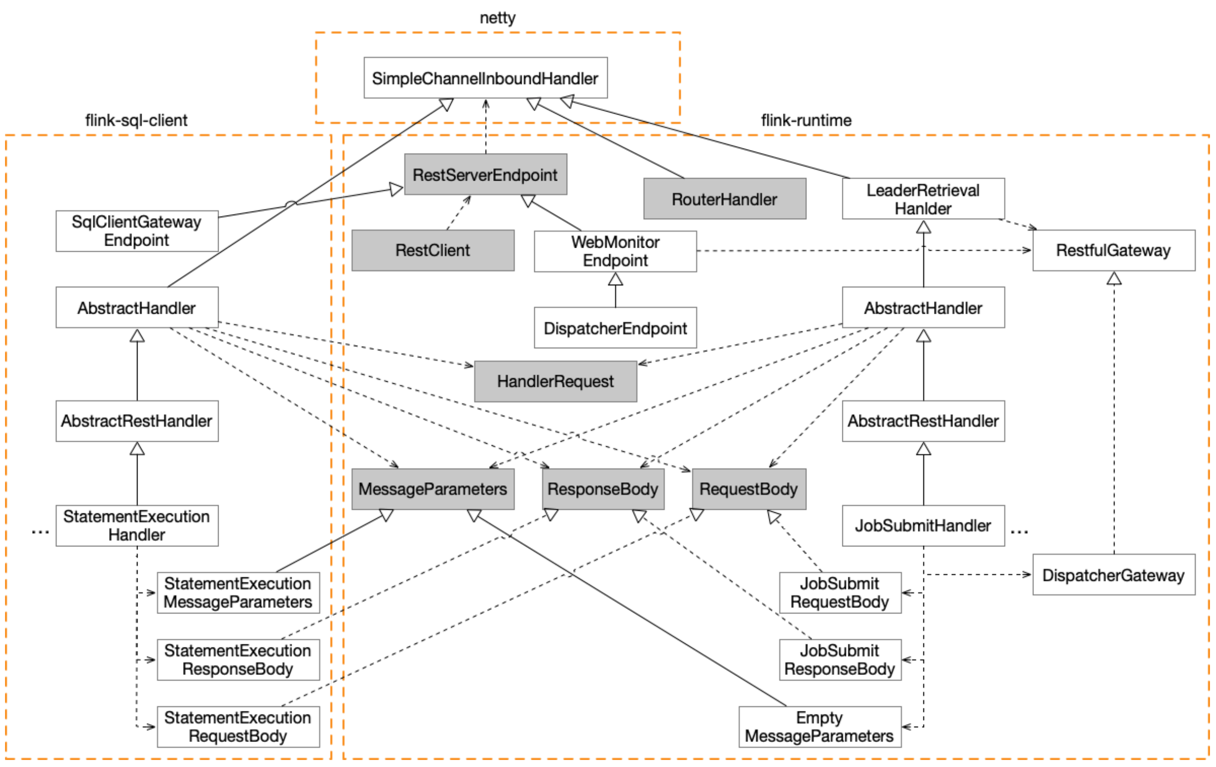
1. reuses part of the REST server code in flink-runtime
    - advantages
        i. unified REST server framework and a lot of code can be reused
    - disadvantages
        i. may need to change REST server code in flink-runtime module to support new features in SQL client
        ii. some classes can't be reused, for example: *AbstractRestHandler*
        iii. although a lot of classes can be reused, those classes are designed for runtime (their package name is org.apache.flink.**runtime**.rest.xx).

It's better to move REST server common classes in flink-runtime module to a separate module (named flink-rest-server) to solve the above questions in future?

2. implements a totally new SQL client REST server
    - advantages
        i. no need to consider the problems caused by code reuse

- disadvantages
    i. there are two REST server implementations in Flink
3. have any other better ideas ?

We tend to choose the first approach! The class architecture of REST server is



(the components with gray color can be reused)

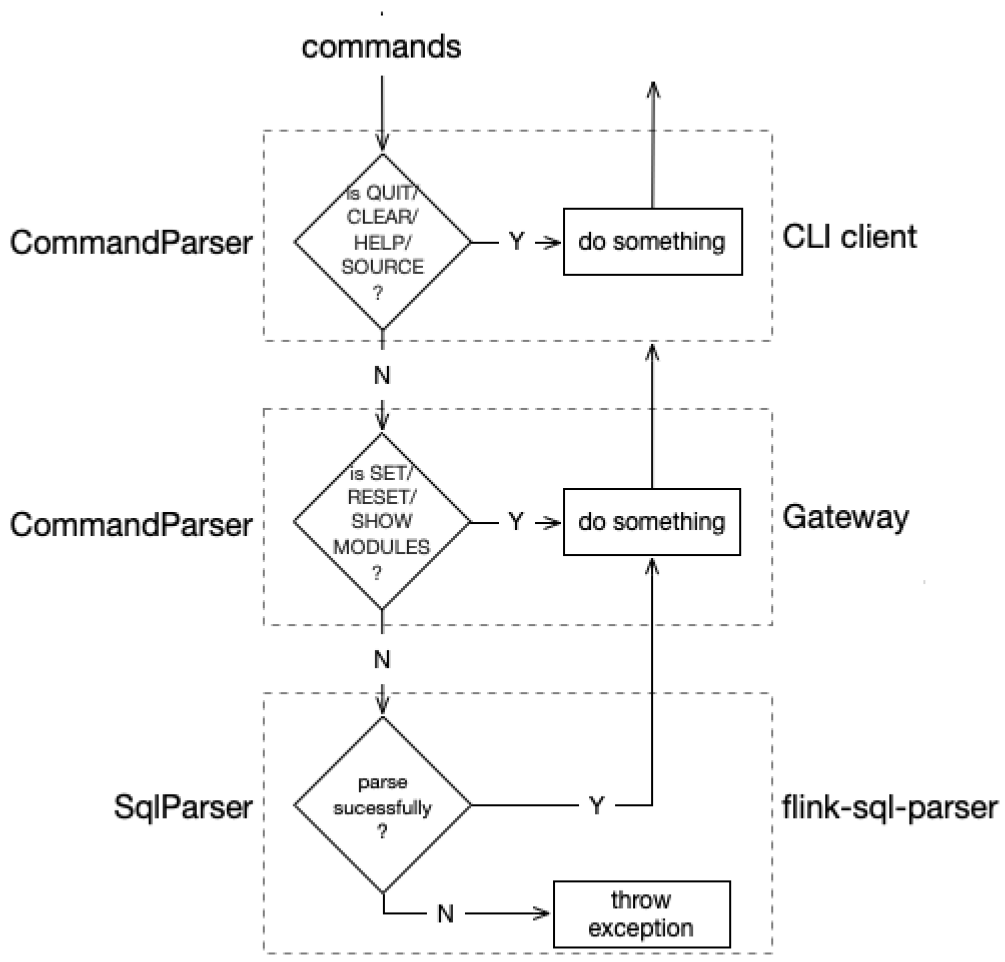
Executor

LocalExecutor is the implementation of the *Executor* interface for embedded mode now, but according to the design above, a *RemoteExecutor* is no longer needed for gateway mode. Only one executor is needed to communicate with Flink and other external systems in gateway.

CommandParser

After the architecture of embedded and gateway mode unified, the *CommandParser* in gateway can be refactored. Currently, command parser is implemented through regular expression matching which fails to support some SQL queries. (e.g. FLINK-15175)

- the workflow of command parsing after refactor



Deployment

We will deploy the SQL client gateway as an independent service like HiveServer2.

REST API

The REST API is versioned, with specific versions being queryable by prefixing the url with the version prefix. Prefixes are always of the form v[version_number]. For example, to access version 1 of /foo/bar one would query /v1/foo/bar.

Querying unsupported/non-existing versions will return a 404 error.

The response is always a JSON type data.

It's also better to provide an unified REST API for clients to execute all statements (e.g. SELECT xx, SHOW xx, SET xx, etc. see link) rather than different REST APIs to execute different kinds of statements. Otherwise, users (especially users who manipulate a Flink cluster through REST API) have to classify the statements by himself (maybe need to parse the statement) and call the specific API for the specific statement.

Currently, the REST APIs will be marked as Internal.

Error handling

When the REST API encounters an error, it will return a JSON object containing only one field “errors”, a string array in which the strings are the details of the errors. This behavior reuses the error handling in the original Flink REST server code. We will not apply additional “error” fields in the following APIs.
Currently, the HTTP response code when encountering an error will be either 400 BAD REQUEST (if the parameters provided by the user is invalid) or 500 INTERNAL SERVER ERROR (otherwise).

Create a session

/v1/sessions	
Verb: POST	Response code: 200 OK
Create a new session with a specific *planner* and *execution_type*. A specific properties could be given for current session which will override the gateway's default properties.	

Request body	{
	    "session_name": "", # optional
	    "planner": "old"/"blink", # required, case insensitive
	    "execution_type": "batch"/"streaming", # required, case insensitive
	    "properties": { # optional, properties for current session
	        "key": "value"
	    }
	}

Response body	{
	    "session_id": "", # if session is created successfully
	}

Trigger session heartbeat

/v1/sessions/:session_id/heartbeat	
Verb: POST	Response code: 200 OK
Trigger heartbeat to tell the server that the client is active, and to keep the session alive as long as configured timeout value. If a session does not receive a heartbeat or any other operations, the session will be destroyed when the timeout is reached.	
Request body	{}
Response body	{}

Execute a statement

/v1/sessions/:session_id/statements	
Verb: POST	Response code: 200 OK

Execute a statement which could be DQL, *DDL(only CREATE/ALTER/DROP DATABASE/TABLE/VIEW xx are supported), DML(only INSERT xx are supported), SET/RESET, SHOW/EXPLAIN/DESCRIBE, USE*.

The *SET xx=yy* statement will override/update the TableConfig held by current session, and the RESET statement will reset all properties set by *SET xx=yy* statement.

The USE MODULE/CATALOG/DATABASE xx statement will update the default module/catalog/database in TableEnvironment held by current session.

The statement must be a **single** command, otherwise the server will throw an exception.

The execution will return a response which is a JSON type data. If the statement execution fails, the errors field in response JSON data will be filled with messages. Otherwise, the result will contain *columns* and *data. T*he *columns* field is an array type and each value is a map which contains column name and column type. The *data* field is also an array type, and each value is a row which is an array type.
please refer to the appendix for more detail about column name and column type.
For SELECT/INSERT statements, only a Flink job is submitted, and the response contains the **job_id** which you can use to get the result (refer to Fetch result).
For other statements, the response contains the complete result data.

We use the same API to run all statements because otherwise users have to determine what statements they are submitting by themselves. Under our design, users can tell immediately what they have submitted from the JSON object and perform specific actions.

Note:
It's a useful feature that a statement contains multiple commands. (scenario: compiling multiple queries into a single job, or executing a job through sql file) However, there are

many unclear things to define for streaming because the streaming job may be long-running. such as:

- could the statement contain an INSERT query and a SELECT query both ?
- could an INSERT query be in the middle of a statement ?
- the execution order for multiple command may be different for batch and stream. Batch job could be executed sequentially. however, if there are multiple INSERT queries for stream job, sequential execution order is not allowed.
- how to define which queries could be compiled into a single job ?
- etc.

Request body

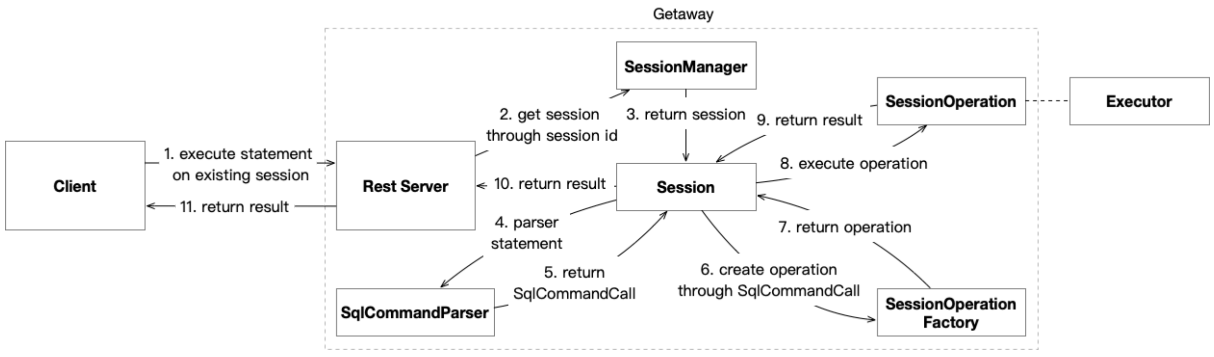
```
{
    "statement": "", # required
    "execution_timeout": "" # execution time limit in milliseconds, optional, but required for stream SELECT ?
}
```

Response body

```
{
    "statement_types": ["", ], # defined in the appendix. This helps the clients do different things on each type. (e.g. for SELECT, a separate view will be displayed). Currently, there is only one result because only one statement is supported now
    "results": [ # if the execution is successful. currently, there is only one result.
        {
            "columns":  [
                {
                    "name": "",
                    "type":  # string value of LogicalType
                },
            ],
            "data": [
                ["value", ], # a row data
```

	] },] }

the workflow of executing a statement:



Fetch result

/v1/sessions/:session_id/jobs/:job_id/result/:token	
Verb: GET	Response code: 200 OK
Fetch a part of result for a flink job execution. If the result data is too large or the result is streaming, we can use this API to get a part of the result at a time. The initialized value of token is 0. The token in the next request must be the same as the token in the current request or must be equal to token (in the current request) + 1, otherwise the client will get an exception from server. If multiple requests are executed with the same token, the result is the same. This design makes sure the client could get the result even if some errors occur in client. The client can get the next part of result using */v1/sessions/:session_id/jobs/:job_id/result/:{token+1}* (which is the value of next_result_uri in the response data). If next_result_uri is empty, no more data is remaining in the server. The server could drop the old data before current token. (The client successfully obtains those data) We will introduce fetch_size or max_wait_time (to reach the fetch_size) for optimization in future.	

The execution will return a response which is a JSON type data. If the statement execution fails, the errors field in response JSON data will be filled with messages. Otherwise, the result will contain *columns*, *data and change_flags. T*he *columns* field is an array type and each value is a map which contains column name and column type. The *data* field is also an array type, and each value is a row which is an array type. The *change_flags* field is an array type and each value indicates whether the row (corresponding to the position) is append data (*true*) or delete data (*false*) for the streaming SELECT. The array length of *change_flags* and *data* must be equal.
please refer to the appendix for more detail about column name and column type.

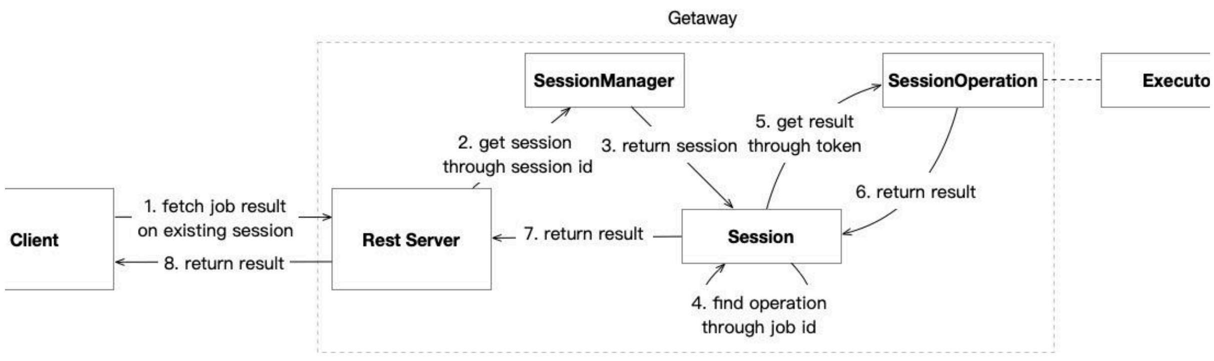
Request body	{}

Response body

```
{
    results: [ # currently, there is only one result now. If multiple queries
is executed in a single job, there are many results.
        {
            "columns":  [ # if the execution is successful
                    {
                        "name": "",
                        "type":  # string value of LogicalType
                    },
                ],
              "data": [
                  ["value", ], # a row data
              ],
            "change_flags": [
                true/false, # append(true)/delete(false) row for streaming
SELECT
            ],
        },
    ],
```

	"next_result_uri": /v1/sessions/:session_id/jobs/:job_id/result/:{token+1} # if not empty, uses this uri to fetch next part of result, else there is no more result. }

the workflow of fetching result:



Get job status

/v1/sessions/:session_id/jobs/:job_id/status	
Verb: GET	Response code: 200 OK
Get the status of a running job. If the session is expired, the server will throw "session not found" exception. If the job is finished, the server will throw "job not found" exception.	
Request body	{}
Response body	{ "status": "" # refer to JobStatus }

Cancel a job

/v1/sessions/:session_id/jobs/:job_id	
Verb: DELETE	Response code: 200 OK

Cancel the running job. If the session is expired, the server will throw "session not found" exception. If the job is finished, the server will throw "job not found" exception.	
Request body	{}

Response body	{
	
	    "status": "CANCELED" # if cancel successfully
	
	}

Close a session

/v1/sessions/:session_id	
Verb: DELETE	Response code: 200 OK
close a session, release related resources including operations and properties	
Request body	{}

Response body	{
	
	    "status": "CLOSED" # if cancel successfully
	
	}

Get info

/v1/info	
Verb: GET	Response code: 200 OK
Get meta data for this cluster	
Request body	{}

Response body	{
	    "product_name": "Apache Flink",
	    "version": "1.11" # Flink version
	}

The typical process for executing a query

```
// 1. open session
POST /v1/sessions

// 2. parse response, get session_id

// 3. submit a non-SELECT/INSERT statement
POST /v1/sessions/:session_id/statements

// 4. parser response, get result data

// 5. submit a SELECT/INSERT statement
POST /v1/sessions/:session_id/statements

// 6. parser response, get job_id and next_result_uri

// 7 fetch first part of result
GET /v1/sessions/:session_id/jobs/:job_id/result/:token

// 8. parse response and get result.
// trigger the next request to fetch the next part of result if next_result_uri is not empty,
otherwise the client has got all the data.

// 9. close the ssion
DELETE /v1/sessions/:session_id
```

Execute a statement without session

(To be discussed. This is mainly designed for testing)

/v1/statements/	
Verb: POST	Response code: 200 OK

Submit a statement without session, and the response contains result data (execute synchronously and return the results). This API could be used for testing. The statement must be *DDL/SELECT/INSERT/SHOW/EXPLAIN/DESCRIBE* and the *INSERT/SELECT* is not supported on stream now.	

Request body	{
	    "statement": "", # required
	    "planner": "old"/"blink", # required, case insensitive
	    "execution_type": "batch"/"streaming", # required, case insensitive
	    "properties": { # optional
	        "key": "value"
	    },
	    "execution_timeout": "" # execution time limit in milliseconds, optional, but required for stream SELECT ?
	}

Response body	as same as Fetch result except that there is no next_result_uri part

JDBC on Batch

As JDBC interface is designed for databases, its semantics on streaming is very limited, so currently we will not support streaming JDBC interfaces.

We will create a new sub-module named flink-jdbc-driver in flink-table module (flink-jdbc already exists in flink-connectors), and build a standalone jdbc driver jar.

For detailed JDBC design, see this document.

Connection

The JDBC connection URL is something like

```
jdbc:flink://localhost:8083?planner=blink
```

Currently supported methods: `createStatement, close, isClosed, setCatalog, getCatalog, getMetaData, setSchema, getSchema`

Statement

Note that one `Statement` object can only open up one `ResultSet` at the same time, so if another execution method is called from another thread when one execution method is running, the running execution method will be implicitly cancelled.
Currently supported methods: `executeQuery, executeUpdate, execute, cancel, close, getMaxRows, setMaxRows, getQueryTimeout, setQueryTimeout, getResultSet, getUpdateCount, getMoreResults, getResultSetType, getConnection, isClosed`

ResultSet

One concern about fetching the results of the queries is how shall we fetch them.

- Shall we use accumulators so that all the results are sent to the job manager after the query is finished? If so, it may put a heavy burden on the job manager if the result set is large or if there are many queries running in the cluster. Also, the memory of job manager may not be enough to hold all the results.
- Shall we store the result in an external storage system like Kafka? If so, it's very likely that users don't have Kafka on their clusters.
- Shall we create a new "result server" to forward the results to the clients in an iterative style? If so, what if the gateway and the task managers can't talk directly to each other, and we cannot determine if the current batch of result is the last batch.

Currently supported methods: `next, close, wasNull, getString, getBoolean, get…, getMetaData, findColumn, isBeforeFirst, isFirst, getRow, setFetchDirection, getFetchDirection, getType, rowUpdated, rowInserted, rowDeleted, getStatement`

ResultSetMetaData

These results are derived from the response json of GET

/v1/sessions/:session_id/jobs/:job_id/result/:token

Currently supported methods: `getColumnCount, isCaseSensitive, isSearchable, isNullable, getColumnLabel, getColumnName, getPrecision, getScale, getColumnType, getColumnTypeName, getColumnClassName`

DatabaseMetaData

We lazily call GET /v1/info to fetch some of the meta data

Currently supported methods (Whether to support these methods is still under discussions. We may remove supports for some methods during the discussion):

`allTablesAreSelectable`, `getURL`, `nullsAreSortedHigh`, `nullsAreSortedLow`, `nullsAreSortedAtStart`, `nullsAreSortedAtEnd`, `getDatabaseProductName`, `getDriverName`, `getDriverVersion`, `getDriverMajorVersion`, `getDriverMinorVersion`, `getIdentifierQuoteString`, `getSQLKeywords`, `supportsAlterTableWithAddColumn`, `supportsAlterTableWithDropColumn`, `supportsColumnAliasing`, `supportsTableCorrelationNames`, `supportsDifferentTableCorrelationNames`, `supportsExpressionsInOrderBy`, `supportsOrderByUnrelated`, `supportsGroupBy`, `supportsGroupByUnrelated`, `supportsGroupByBeyondSelect`, `supportsLikeEscapeClause`, `supportsNonNullableColumns`, `supportsOuterJoins`, `supportsFullOuterJoins`, `supportsLimitedOuterJoins`, `getSchemaTerm`, `getCatalogTerm`, `isCatalogAtStart`, `getCatalogSeparator`, `supportsCatalogsInDataManipulation`, `supportsCatalogsInTableDefinitions`, `supportsSchemasInDataManipulation`, `supportsSchemasInTableDefinitions`, `supportsSubqueriesInComparisons`, `supportsSubqueriesInExists`, `supportsSubqueriesInIns`, `supportsSubqueriesInQuantifieds`, `supportsCorrelatedSubqueries`, `supportsUnion`, `supportsUnionAll`, `getMaxColumnNameLength`, `getTableTypes`, `getPrimaryKeys`, `getTypeInfo`, `supportsResultSetType`, `getConnection`, `getDatabaseMajorVersion`, `getDatabaseMinorVersion`, `getJDBCMajorVersion`, `getJDBCMinorVersion`, `getFunctions`

Compatibility, Deprecation, and Migration Plan

This FILP will guarantee alignment with the functionality of old embedded mode, and does not involve any old public user interfaces. So there is no Compatibility, Deprecation, and Migration.

Implementation Plan

1. Basic features

- add basic web server
- supports basic REST API
- supports Session & Session Management
- supports Basic SessionOperations
- supports CLI Client connecting to REST server
  - supports basic commands
  - supports submitting batch job & display results
  - supports submitting stream job & display results
- supports loading jar files
- refactor embedded mode

2. Supports JDBC on batch

- supports basic methods mentioned above

3. Refactor Command Parser

- implements new command parser
- remove old command parser

Rejected Alternatives

- port HiveServer2 to Flink, like Spark thrift server. The advantage of this approach is that we can reuse the most code of HiveServer2 except operation part which should be changed to connect to Flink cluster instead of Hive cluster. However the

HiveServer2 does not support REST API, and is hard to extend support for streaming. If we change the Thrift protocol, the new protocol may be incompatible with old ones.

To be discussed in future

1. supports thrift server ? (FLINK-15017)
2. supports JDBC on streaming ?
3. supports more features on REST API, e.g. pause/resume streaming jobs
4. supports authorization management
5. supports gateway persistence and high availability

Appendix

In sync with JDBC interface

- green: Can be submitted by Statement#executeQuery
- yellow: Can be submitted by Statement#executeUpdate

Command	**Column Name**	**Column Type**	**value**
submit SELECT query	job_id	VARCHAR	the flink job id corresponding the SELECT query
fetch SELECT result	(selected field names)	(selected field types)	(selected rows)
DDL, SET, RESET, USE	affected_row_count	BIGINT	0 (according to jdbc Statement#executeUpdate)
submit DML query	job_id	VARCHAR	the flink job id corresponding the DML query

fetch DML result	affected_row_count	BIGINT	the number of affected rows (only one row) **for stream job, the value is always java.sql.Statement.SUCCESS_NO_INFO = -2**
SHOW MODULES	modules	VARCHAR	module names in current session
SHOW CATALOGS	catalogs	VARCHAR	catalog names in current session
SHOW DATABASE	databases	VARCHAR	database names in current catalog
SHOW TABLES	tables	VARCHAR	table names in current catalog and database
	type	VARCHAR	type of table (TABLE or VIEW)
SHOW FUNCTIONS	functions	VARCHAR	function names in current session
DESCRIBE xx	table_schema	VARCHAR	table schema **JSON** value (It's hard to convert TableSchema to table format, because watermark spec and regular columns are never in the same row)
EXPLAIN	explanation	VARCHAR	explain result (only one row)

Statement types:

Command	**Statement Type**
SELECT	SELECT
INSERT	INSERT
CREATE TABLE	CREATE_TABLE

CREATE VIEW	CREATE_VIEW
DROP TABLE	DROP_TABLE
DROP VIEW	DROP_VIEW
SHOW MODULES	SHOW_MODULES
SHOW CATALOGS	SHOW_CATALOGS
SHOW DATABASE	SHOW_DATABASE
SHOW TABLES	SHOW_TABLES
SHOW FUNCTIONS	SHOW_FUNCTIONS
DESCRIBE	DESCRIBE
EXPLAIN	EXPLAIN
SET	SET
RESET	RESET