Cloudera Impala Source Code Explanation and Analysis

Yue Chen

http://linkedin.com/in/yuechen2

http://dataera.wordpress.com

Impala Architecture

I. **SQL Interface**

- This part is borrowed from **Hive**, including **ODBC/Beeswax**. Client' s SQL query is sent to any **impalad** in the cluster through the **Thrift API** of **ODBC/Beeswax**. And then this **impalad** becomes the coordinator of this query.

II. **Unified metastore**

- Impala' s metadata storage approach is borrowed from Hive. Impala provides **statestored** process to collect the resource information from every **impalad** for query scheduling. **Statestored** provides **Thrift** service, and distributes its tables' metadata to each **impalad**.

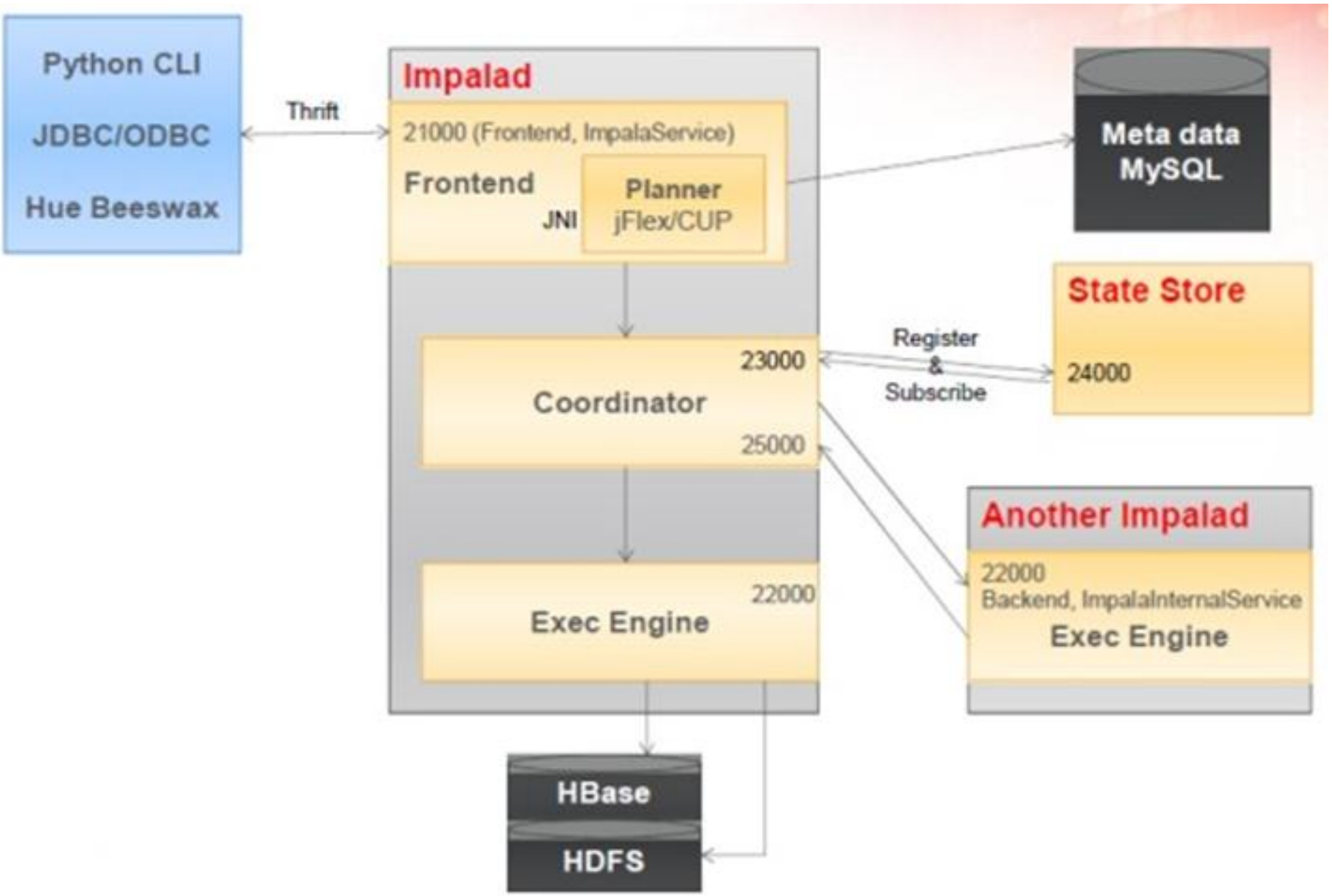
Impala Architecture

III. **Impala daemon**

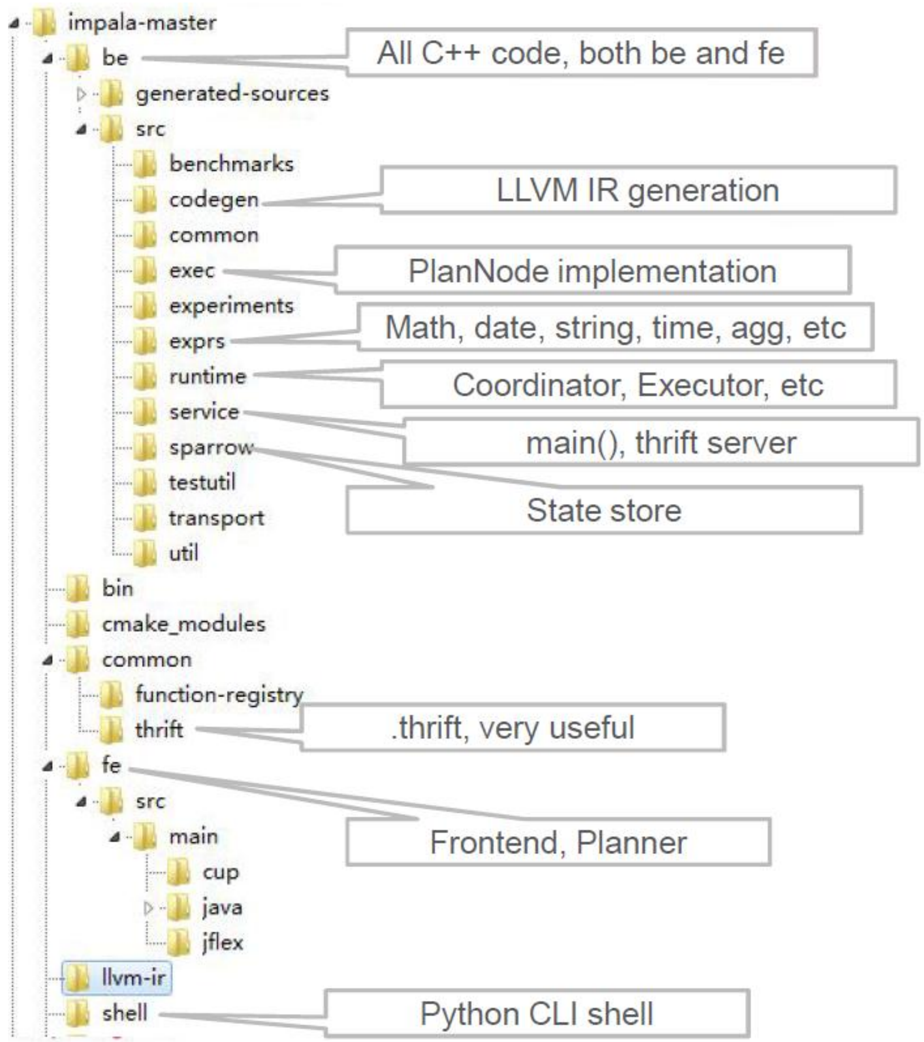
- Named **impalad**, it has two functions: 1. coordinates the query execution, assigns tasks to other **impalad**, and collects the results from other **impalad**. 2. This **impalad** would execute the tasks from other **impalad**. And it mainly operates data in the local HDFS and HBase. They are local IO operations. What is more, HDFS supports **dfs.client.read.shortcircuit**, reading data directly from the local disk if the client is co-located with the data, instead of reading data and sending over TCP.

IV. **Impala' s supporting format includes Parquet, Trevni, RCFile, etc.**

Impala Architecture



Impala Source Code Structure



Impala RPC

Component	Service	Port	Access Requirement	Comment
ImpalaDaemon	Impala Daemon Backend Port	22000	Internal	ImpalaBackendService export
	Impala Daemon Frontend Port	21000	External	ImpalaService export
	Impala Daemon HTTP Server Port	25000	External	Impala debug web server
	StateStoreSubscriber Service Port	23000	Internal	StateStoreSubscriberService
ImpalaStateStore Daemon	StateStore HTTP Server Port	25010	External	StateStore debug web server
	StateStore Service Port	24000	Internal	StateStoreService export
Show Impala Catalog	Impala Catalog	25020	External	To See Impala Catelog

Impala Thrift RPC Interface

1) **Client <-> impalad (frontend) (RPC client <-> RPC server)**

- BeeswaxService(beeswax.thrift): client uses query() to submit SQL query, and asynchronously calls get_state() to listen to the status of this query. Once finished, it calls fetch() to get the result.
- TCLIService(cli_service.thrift): client submits SQL query, similar as above. It supports DDL operation, e.g., GetTables() returns the metadata of the given table.
- ImpalaService and ImpalaHiveServer2Service(ImpalaService.thrift) are subclasses of the above class, having more functionalities, while their core functionalities are similar.

Impala Thrift RPC Interface

2) **impalad (backend) <-> statestored**

- StateStoreService(StateStoreService.thrift): statestored keeps the status database of all backend service, this is a data exchange center. The state kept here is soft state or volatile. Once the machine is down, all the information is gone. Every Impala backend would call StateStoreService.RegisterService() to register themselves to statestored when started. This is done through StateStoreSubscriber bound to this service. And then it calls StateStoreService.RegisterSubscription() to show its StateStoreSubscriber will receive the update from statestored.

Impala Thrift RPC Interface

3) **statestored <-> impalad (backend)**

- StateStoreSubscriberService(StateStoreSubscriberService.thrift): statestored periodically sends status update information to backend bound StateStoreSubscriber. Then impalad backend calls StateStoreSubscriberService.UpdateState() to update the state. Meanwhile UpdateState() reutrns its update to statestored.

Impala Thrift RPC Interface

4) **impalad (backend) <-> other impalad (backend)**

- These **impalad** are mutual client/server.
- **ImpalaInternalService(ImpalaInternalService.thrift)**: one backend coordinator would send an execution plan fragment to others (submit **ExecPlanFragment** and request to return **ReportExecStatus**).

Impala Thrift RPC Interface

5) **impalad backend <-> other frontend**

- ImpalaPlanService(ImpalaPlanService.thrift): other frontend could generate TExecRequest and it is fetched by backend for executing.

Impala Thrift RPC Interface

- Impala frontend is written using Java, while backend is in C++. Frontend parses the SQL and generates the execution plan, and pass it to the backend using Thrift serialization/ deserialization. TExecRequest is the intermediate transfer data structure, representing a Query/DML/DDL query request. It is also the interface of frontend and backend. So we could replace the frontend, using other stuff to build this TExecRequest to send to backend for executing. This is what ImpalaPlanService does, in the previous slide.

Impala-shell

- To begin with, when opening the Impala shell, it calls impala_shell.py to execute the commands and queries.
- In this Python script, the OptionParser() parses the command line.
- If there exist -query or -query_file or -f in the command, the script would call execute_queries_non_interactive_mode(options) for non-interactive queries. I.e., SQL queries in a file.
- If not, it gets into the ImpalaShell.cmdloop(intro) loop.

Impala-shell

- Once get into the command loop, it connects to one impalad. The command is like "connect localhost:21000", and it calls function do_connect(self, args).
- This function generates the corresponding socket connecting to impalad, according to the host and port set by the user.
- Let's see the following code:
- self.imp_service = ImpalaService.Client(protocol) in function __connect(self)
- imp_service is the request submitter of the client.

Impala-shell

- Here we give an example. If client types in command like "select col1, col2 from table1", it goes into function do_select(self, args). In this function it generates BeeswaxService.Query object, and fill this with query statement and configuration. Then it goes into __query_with_result(), using imp_service.query(query) to submit query. Notice that ImpalaService is asynchronous. After submission, it returns a QueryHandle. Then it is in a while loop to get the state using __get_query_state(). If finding that this query is FINISHED, it calls fetch() RPC to get the result.

statestored

- statestored provides StateStoreService RPC service, StateStoreSubscriberService RPC service is provided in the impalad process. StateStoreService RPC is implemented in StateStore class.
- When statestored receives backend’ s RegisterService RPC request, calls StateStore::RegisterService() to process.

statestored

- What StateStore::RegisterService() does:
- Adds service to StateStore.service_instances_ according to the service_id that TRegisterServiceRequest provides.
- Sends subscriber_address when impalad backend RegisterService to statestored. At statestored, it would generates corresponding Subscriber object according to this subscriber_address. Then it adds this Subscriber to a map. Each Subscriber has a unique ID. So the impala backend in the cluster has a global unique ID.

statestored

- If backend/StateStoreSubscriber/SQL task fails, statestored will know. And it will notify other relevant backend.

- How each backend update information?
- StateStore::UpdateLoop() periodically pushes all service member update to eash backend.

impalad

- impalad is wrapped in the class ImpalaServer. ImpalaServer includes the features of frontend (fe) and backend (be), implements ImpalaService(Beeswax), ImpalaHiveServer2Service(HiveServer2) and ImpalaInternelService API.
- Global function CreateImpalaServer() creates a ImpalaServer including multiple ThriftServer, shown as follows.

impalad

1) Creates a ThriftServer named beeswax_server, providing ImpalaService(Beeswax) service, mainly supporting querying. It is the core frontend service. Port 21000.
2) Creates a ThriftServer named hs2_server, providing ImpalaHiveServer2Service. It provides Query, DML, DDL operations. Port 21050.
3) Creates a ThriftServer named be_server providing ImpalaInternalService to other impalad inside the system. Port 22000.

impalad

4) Creates ImpalaServer object. The previous 3 ThriftServer’s Tprocessor are assigned by this ImpalaServer object. A typical example: we submit BeeswaxService.query() request through Beeswax interface. It is finished by void ImpalaServer::query(QueryHandle& query_handle, const Query& query) in impala-beeswax-server.cc.

SQL Parsing and Executing

- SQL query goes this way:
- BeeswaxService.Query->TClientRequest->TExecRequest impala-coordinator gives TExecRequest to multiple backend to handle.

SQL Parsing and Executing

- Here we give an example about how to process the SQL query as follows:

```
select jobinfo.dt,user,
max(taskinfo.finish_time-taskinfo.start_time),
max(jobinfo.finish_time-jobinfo.submit_time)
from taskinfo join jobinfo on jobinfo.jobid=taskinfo.jobid
where jobinfo.job_status='SUCCESS' and
  taskinfo.task_status='SUCCESS'
group by jobinfo.dt,user
```

SQL Parsing and Executing

- First we call AnalysisContext.analyze(String stmt) to analyze this SQL query.
- This calls SelectStmt.analyze() to analyze the query the register information to Analyzer.

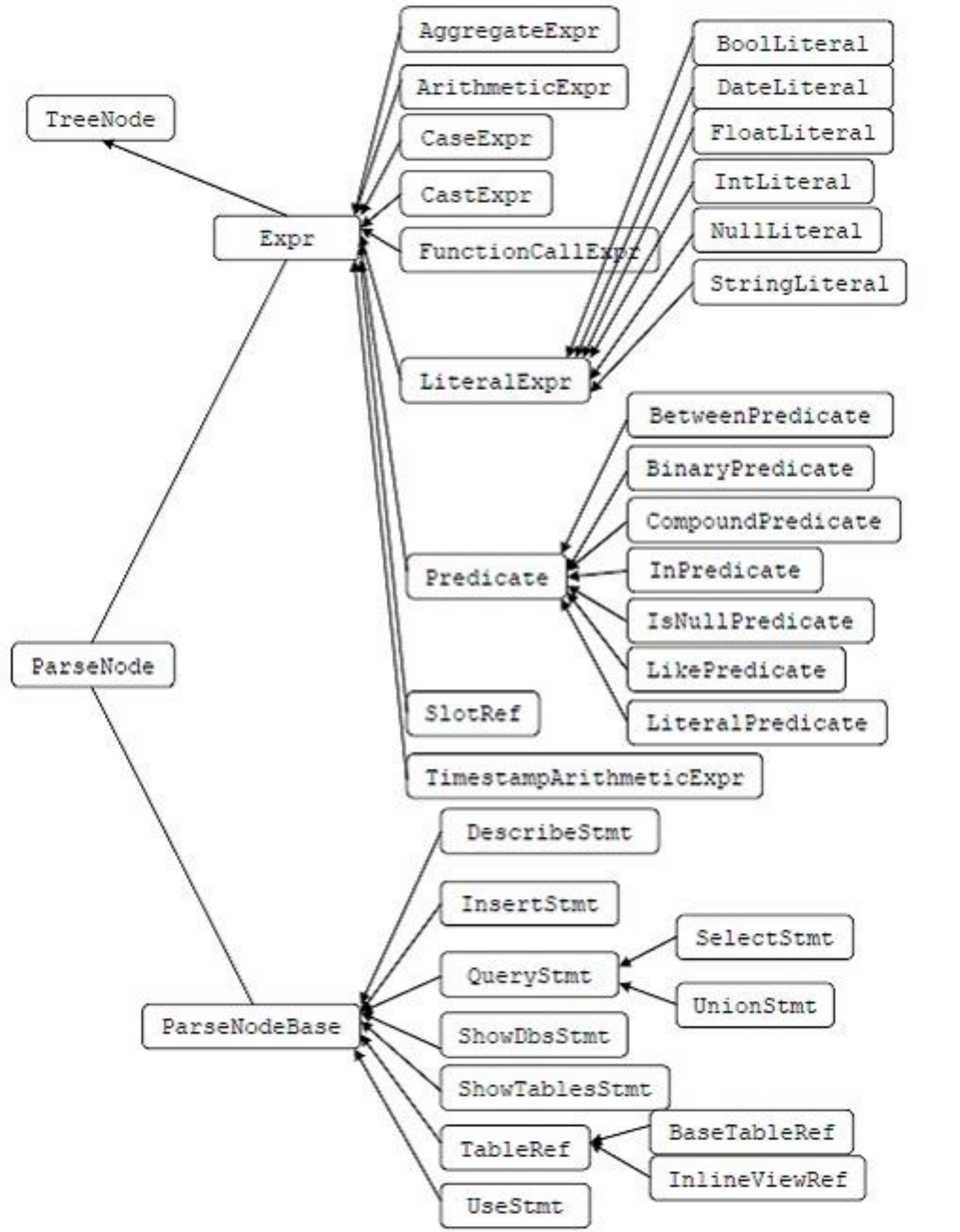
SQL Parsing and Executing

1) First it processes the Table of this query, namely TableRefs. These Table are extracted from the from clause, including from, join, on/using. It fills in TupleDescriptor, and register to registerBaseTableRef.
2) Handles select (select, MAX(), AVG()) clause. It analyzes which items does the SQL select, and fill in resultExprs and colLabels. Then it recursively analyzes the Expr in resultExprs till the bottom layer of the tree, and register SlotRef to Analyzer.

SQL Parsing and Executing

3) Analyzes where clause, recursively analyze the Expr tree, and fills in the member variables of Analyzer using registerConjunct(), and fill in where Clause Conjunct.
4) Handles sort (order by). First parses aliases and ordinals, then extracts Expr and puts them into orderingExprs. Finally creates SortInfo object.
5) Handles aggregation (group by, having, AVG, MAX), register information to Analyzer using registerConjunct().
6) Handles InlineView.

Data Structure View



TExecRequest

1) If DDL (use, show tables, show databases, describe), calls createDdlExecRequest();
2) If Query or DML, creates and fills TQueryExecRequest.

PlanNode and PlanFragment

- Planner converts the parsed expression tree to Plan fragments, to be executed in backend.
- Planner planner = new Planner();
- ArrayListfragments = planner.createPlanFragments(analysisResult, request.queryOptions);
- Execution Plan is represented as PlanFragment array. It will be serialized to TQueryExecRequest.fragments, and be given to backend coordinator for scheduling and executing.

PlanNode and PlanFragment

- In Planner.createPlanFragments(), PlanNode is the logical functionality node; PlanFragment is the real execution plan node.

Create PlanNode

- PlanNode singleNodePlan = createQueryPlan(queryStmt, analyzer, queryOptions.getDefault_order_by_limit());

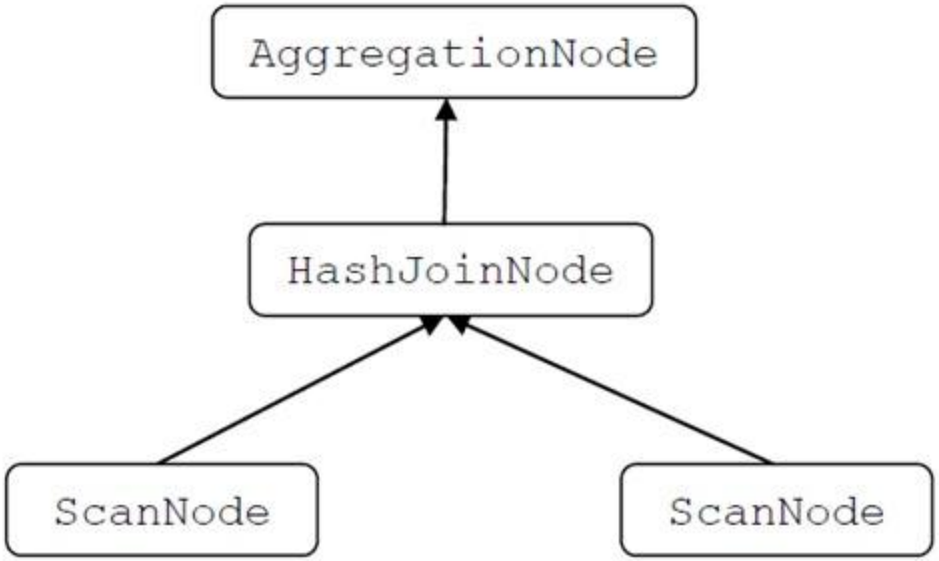
1) According to the TableRef in the from clause, creates a PlanNode, normally ScanNode (HdfsScanNode or HBaseScanNode). This one is associated to a ValueRange array because of the range needed to be read. Also it is associated with a conjunct (where clause).
2) If there is join in the query, we need to create HashJoinNode. It is Planner.createHashJoinNode(). HashJoinNode is tree-structured and has two child nodes (ScanNode).

Create PlanNode

3) If group by clause, it creates AggregationNode and set the previous HashJoinNode as its children.
4) If order by… limit clause, it creates SortNode.

Create PlanNode

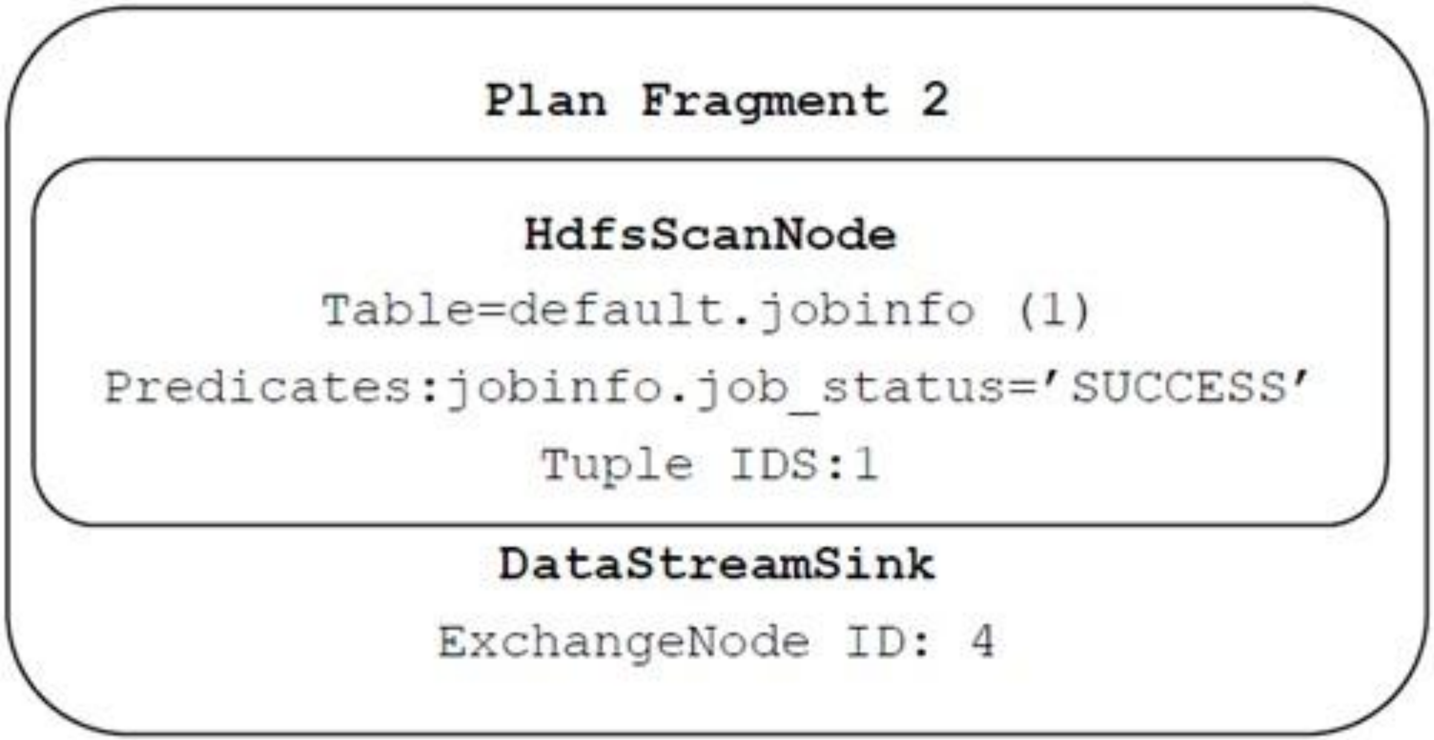
- Now createQueryPlan() is finished, the PlanNode’s execution tree is shown below:



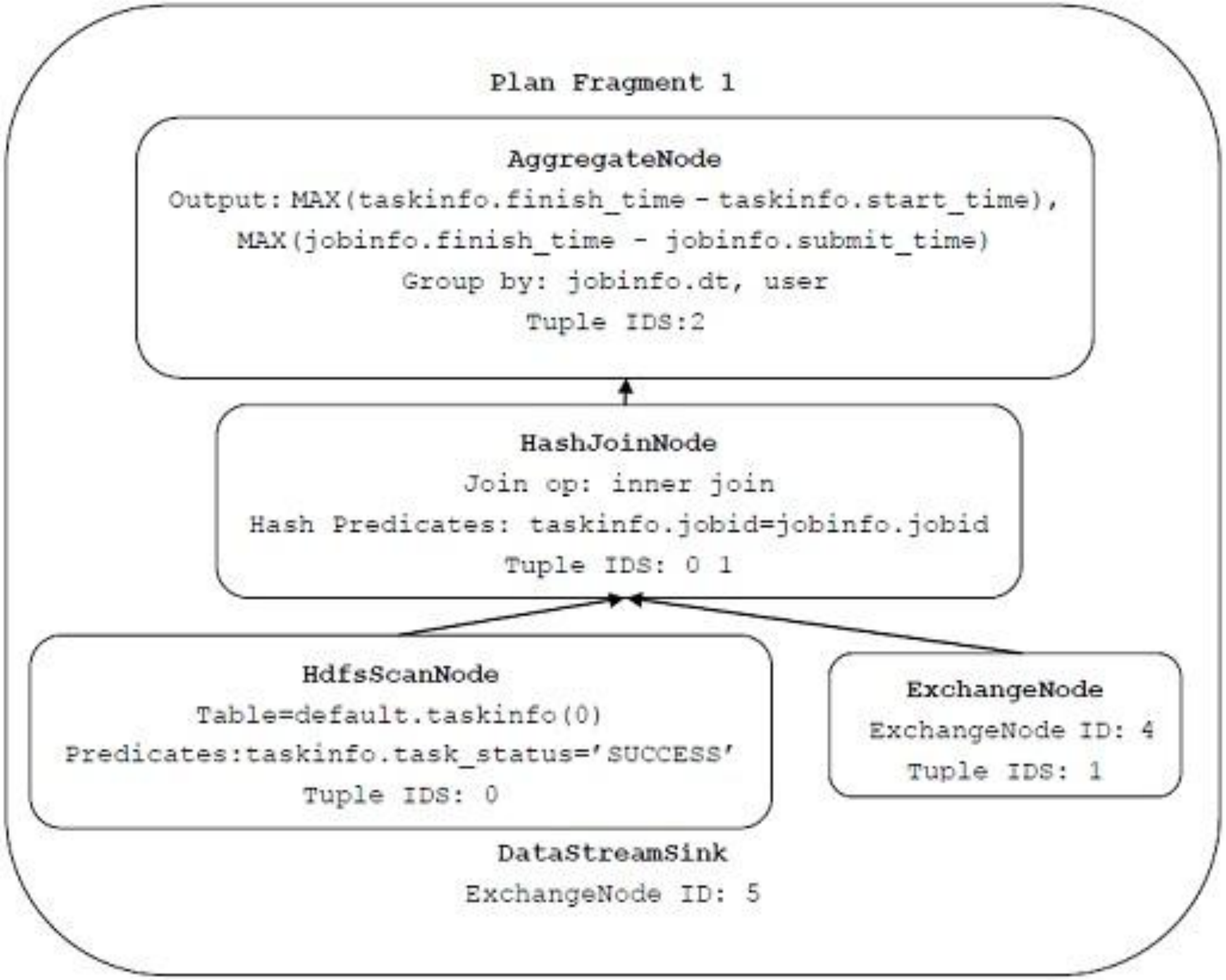
Create PlanFragments

- Now we should see how many impala backend are there. If only one, all the tree is on the same impalad; or transform it to PlanFragments.
- For different node, basically Impala uses exchange node to pass data stream.

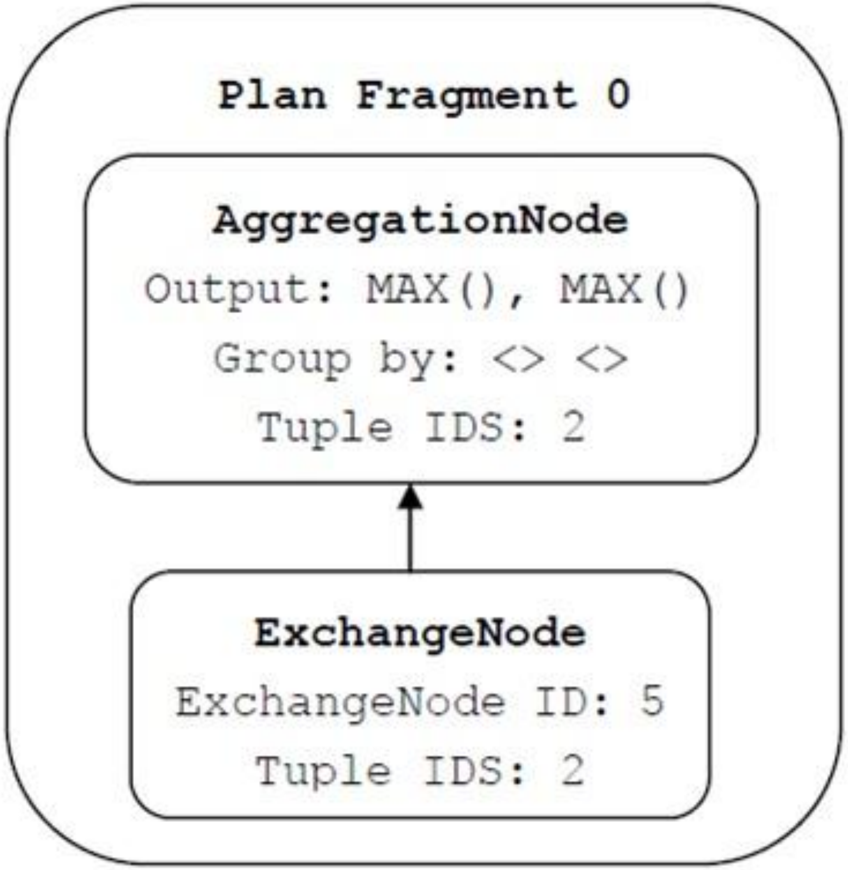
Create PlanFragments



Create PlanFragments



Create PlanFragments



TExecRequest

- In frontend.createExecRequest(), the Planner.createPlanFragments() returns an ArrayList containing the execution plan. We call PlanFragment.toThrift() to serialize it to TQueryExecRequest.
- TQueryExecRequest is a member of TExecRequest (Thrift structure).
- Finally TExecRequest is fetched by backend.

Backend

- The query goes into
- void ImpalaServer::query(QueryHandle& query_handle, const Query& query)
- It generates a QueryExecState in the SQL execution lifetime.
- Then calls Execute function to start executing.
- Starts a Wait thread to wait for the result.

Backend

- Execute() firstly request and get TExecRequest from impala frontend through JNI.
- Coordinator is responsible to distribute the query to multiple nodes to execute; backend instance is to execute query's PlanFragment. So each query has one Coordinator object and multiple backend instance. Meanwhile the query_profile_ is for the profile of query execution.

Coordinator

- Exec() is an entrance function. The execution flow is as follows:
- ComputeFragmentExecParams(const TQueryExecRequest& exec_request):
  - ComputeFragmentHosts(): for every PlanFragment, according to the node where the data exists, finds its backend instance.
  - Gives the destinations of FragmentExecParams (Data Sink), and computes the parameters of ExchangeNode.
- ComputeScanRangeAssignment():
  - Computes how much data each backend should scan.

Coordinator

- Coordinator::Exec()
  - If there is Coordinator PlanFragment: new PlanFragmentExecutor(), and then gives parameters TExecPlanFragmentParams.
  - Then it generates BackendExecState associated to each instance, request RPC to each instance:

    Issue all rpcs in parallel

    Status fragments_exec_status = ParallelExecutor::Exec(…)

PlanFragmentExecutor

- At the RPC server side, calls ImpalaServer::ExecPlanFragment() -> ImpalaServer::StartPlanFragmentExecution() to generate FragmentExecState, PlanFragmentExecutor is within it.
- PlanFragmentExecutor is the real execution function, which has Prepare()/Open()/GetNext()/Close().

ExecNode

- This is the real one that processes data in impalad, including hash-join, aggregation, scan, etc. Multiple ExecNode compose an execution tree. Firstly leaf, finally root, to be executed. It has methods Prepare(), Open(), GetNext(), Close(), CreateTree().
- Major data structures:
  - ObjectPool* pool_
  - vector<Expr*> conjuncts_
  - vector<ExecNode*> children_
  - RowDescriptor row_descriptor_

ExecNode

- **Major functions:**
  - Prepare() is called before open(), used for code generation, adding functions to the LlvmCodeGen object
  - Open() is to prepare working, and calls child nodes' GetNext(); Retrieving the JIT compiled function pointer happens in Open()
  - GetNext() return a set of row, marking eos
  - EvalConjuncts() evaluates expressions, and returns a bool value
  - CodegenEvalConjuncts() is the codegen'd function of EvalConjuncts()

PlanFragmentExecutor

- PlanFragmentExecutor::Prepare(TExecPlanFragmentParams)
  - Set the mem_limit of the query;
  - DescriptorTbl::Create(): Initialize descriptor table;
  - ExecNode::CreateTree(): Generates execution tree’s structure;
  - PlanFragmentExecutor::plan_ points to the root node of the execution tree;
  - Set the Exchange Node would receive how many senders’ data;
  - Calls plan_->Prepare(): Executes the execution tree from the root node, initializing runtime_profile (for statistics) and LLVM code generation of conjuncts (adding functions to the LlvmCodeGen object);

PlanFragmentExecutor

- PlanFragmentExecutor::Prepare(TExecPlanFragmentParams)
  - If code generation, calls runtime_state_->llvm_codegen()->OptimizeModule() to optimize.
  - Maps the ScanNode's scan range to file/offset/length;
  - DataSink::CreateDataSink();
  - Sets up profile counter;
  - Generates RowBatch to store the results.

PlanFragmentExecutor

- PlanFragmentExecutor::Open()
  - Firstly starts the profile-reporting thread, then calls OpenInternal()

1) Calls plan_->Open() and along the execution tree, calls ExecNode:: Open().

   Taking HdfsScanNode::Open() for example:

   - Calls DiskIoMgr::RegisterReader to initialize the connection with HDFS hdfs_connection_;
   - Adds the the file split to be read into HdfsScanNode's queue queued_ranges_;
   - Calls the HdfsScanNode::DiskThread driver HdfsScanNode::StartNewScannerThread()-> HdfsScanNode::ScannerThread->HdfsScanner::ProcessSplit() to read data. (One scanner thread reads one scan range currently)

PlanFragmentExecutor

- Calls IssueQueuedRanges() to send the pre-read range in queued_ranges_ to DiskIoMgr. Since the disk thread is started, so we can read data.

2) If there is sink in this PlanFragment, it needs to send all the data that this PlanFragment wants to send to other PlanFragment. Calls PlanFragmentExecutor::GetNextInternal(), and recursively calls ExecNode::GetNext() to get the result. For different operations/ExecNode, the logic of ExecNode::Open() is different, as well as GetNext(). (HdfsScanNode::GetNext() VS HashJoinNode::GetNext())

PlanFragmentExecutor

3) PlanFragmentExecutor::GetNext(RowBatch** batch)

- Get the result of ExecNode::GetNext() (plan_->GetNext). When PlanFragmentExecutor::done_==true, all the data is processed, this PlanFragmentExecutor could exit.

Code Generation

- When calling Prepare(), Impala selectively uses code generation.
- The Prepare() for each node (operation) is in folders exprs/exec/runtime.
- state->codegen()->AddFunctionToJit()

Code Generation

- For example, when evaluating arithmetic expressions, it calls Expr::Prepare() and ArithmeticExpr::Prepare().
- In Expr::Prepare, creates codegen object:
  - LlvmCodeGen* codegen = NULL;
  - codegen = state->codegen();
- For an expression tree, recursively Prepare() them. (PrepareChildren()->Prepare())

Code Generation

- **To get function type:**

```
void* fn_ptr;

Status status = LibCache::instance()->GetSoFunctionPtr("",
  fn_.scalar_fn.symbol, &fn_ptr, &cache_entry_);

compute_fn_ = reinterpret_cast<ComputeFn>(fn_ptr);
```

- **They will set fn_ and members like fn_.name.function_name**

Code Generation

- **To generate code (call stack):**

```
root->CodegenExprTree(codegen);
this->Codegen(codegen);
Function* ArithmeticExpr::Codegen(LlvmCodeGen* codegen);
```

- **Creates context and LlvmBuilder.**
- **Fill in the object according to the rules (one case):**

```
else if (fn_.name.function_name == "divide")
    result = builder.CreateFDiv(lhs_value, rhs_value, "tmp_div");
```

Code Generation

- **Uses a phi node to coalesce results (if having branches)**

```
PHINode* phi_node = builder.CreatePHI(return_type, num_paths,
  "tmp_phi");
  phi_node->addIncoming(GetNullReturnValue(codegen),
  entry_block);
  if (GetNumChildren() == 2) {
    phi_node->addIncoming(GetNullReturnValue(codegen),
  compute_rhs_block);
  }
  phi_node->addIncoming(result, compute_arith_block);
  builder.CreateRet(phi_node);
```

Code Generation

- **Finally verifies, optimizes function and add this to the JIT pool:**

```
return codegen->FinalizeFunction(function);
```

For ADD_LONG_LONG, the IR looks like:

```
define i64 @ArithmeticExpr(i8** %row, i8* %state_data, i1* %is_null) {
entry:
%child_result = call i64 @IntLiteral(i8** %row, i8* %state_data, i1* %is_null)
%child_null = load i1* %is_null
br i1 %child_null, label %ret, label %compute_rhs

compute_rhs: ; preds = %entry
%child_result1 = call i64 @IntLiteral1(i8** %row, i8* %state_data, i1* %is_null)
%child_null2 = load i1* %is_null
br i1 %child_null2, label %ret, label %arith

arith: ; preds = %compute_rhs
%tmp_add = add i64 %child_result, %child_result1
br label %ret

ret: ; preds = %arith, %compute_rhs, %entry
%tmp_phi = phi i64 [ 0, %entry ], [ 0, %compute_rhs ], [ %tmp_add, %arith ]
ret i64 %tmp_phi
```

Code Generation

- If codegen is enabled for the query, we will codegen as much of the expr evaluation as possible. This means all builtins will run through the codegen path and nothing (e.g. Exec nodes) will call GetValue(). Instead, they will call GetIrComputeFn().
- GetValue:
  - compute_fn_(this, row);
- GetIrComputeFn:
  - Calls GetWrapperIrComputeFunction
  - Gives the function pointer to fn according to codegen object.

Code Generation

- AddFunctionToJit():

  For hash-join, aggregation, hdfs-scan and native UFP expr, adds the function to be automatically JIT compiled after the module is optimized, through:

  ```
  fns_to_jit_compile_.push_back(make_pair(fn, fn_ptr));
  ```

Code Generation

Afterward, FinalizeModule() should be called at which point all codegen'd functions are optimized. After FinalizeModule() returns, all function pointers registered with AddFunctionToJit() will be pointing to the appropriate JIT'd function.

- LlvmCodeGen::FinalizeModule():
  - OptimizeModule();
  - JIT compile all codegen'd functions:
    - *fns_to_jit_compile_[i].second = JitFunction(fns_to_jit_compile_[i].first);

Code Generation

- ReplaceCallSites:
  - llvm::Function* ReplaceCallSites(llvm::Function* caller, bool update_in_place, llvm::Function* new_fn, const std::string& target_name, int* num_replaced)
  - Replaces all instructions that call 'target_name' with a call instruction to the new_fn.

- **Next time calling function 'target_name', it will call the codegen'd function.**

References

- Cloudera Impala official documentation and slides
- http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real/
- http://www.theregister.co.uk/2012/10/24/cloudera_hadoop_impala_real_time_query/
- http://yanbohappy.sinaapp.com
- http://www.tcloudcomputing.com.cn/