facebook

Hive Anatomy

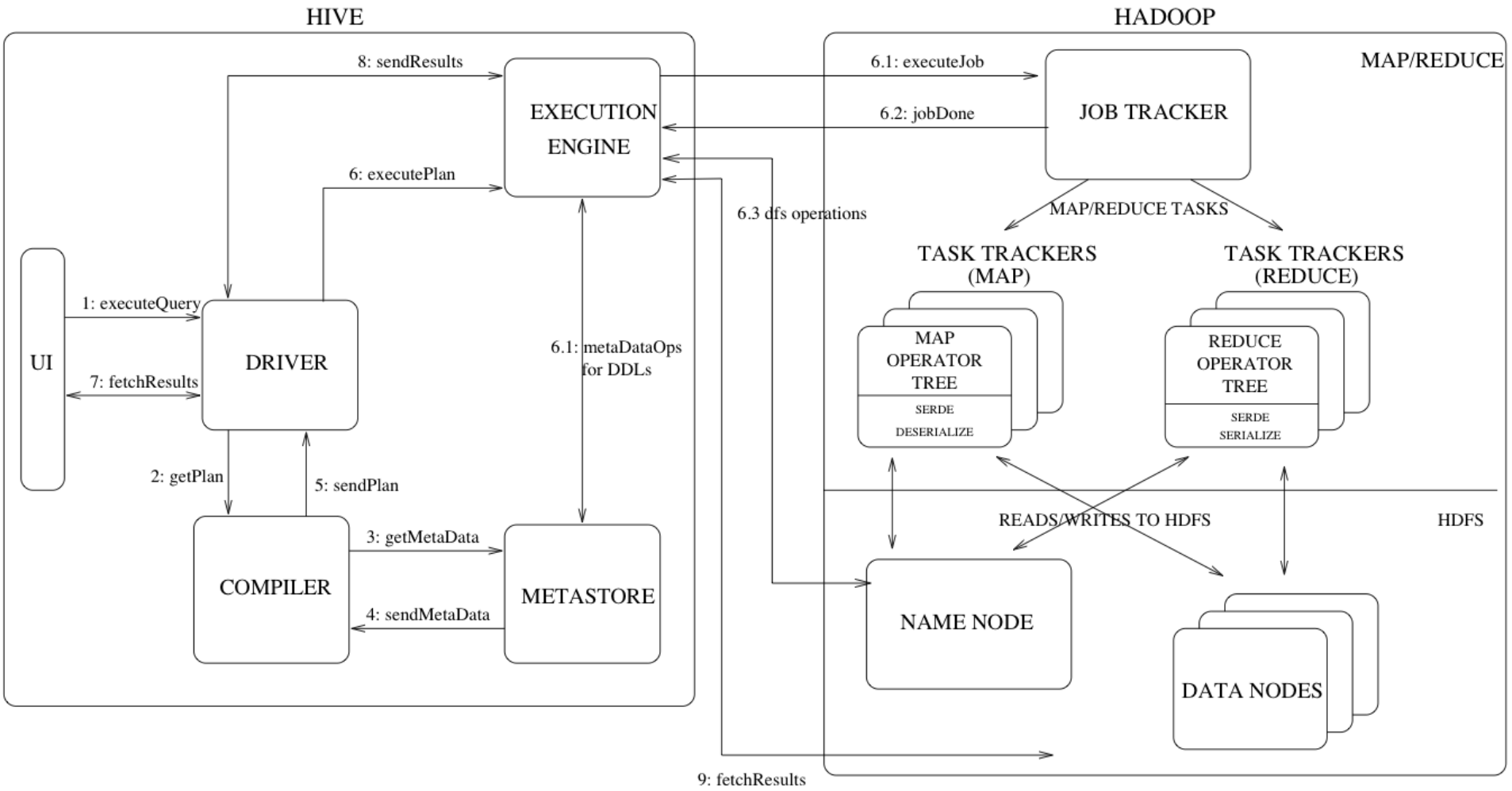
Data Infrastructure Team, Facebook
Part of Apache Hadoop Hive Project

Overview

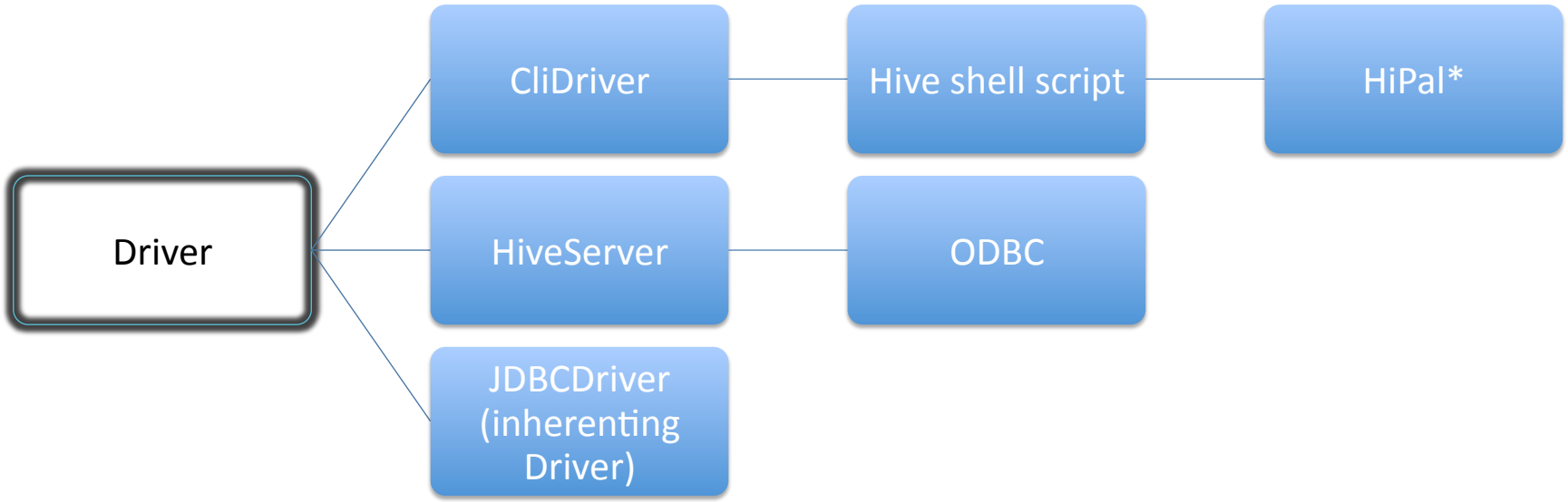
- Conceptual level architecture
- (Pseudo-)code level architecture
    - Parser
    - Semantic analyzer
    - Execution
- Example: adding a new Semijoin Operator

Conceptual Level Architecture

- Hive Components:
  - Parser (antlr): HiveQL → Abstract Syntax Tree (AST)
  - Semantic Analyzer: AST → DAG of MapReduce Tasks
    - Logical Plan Generator: AST → operator trees
    - Optimizer (logical rewrite): operator trees → operator trees
    - Physical Plan Generator: operator trees -> MapReduce Tasks
  - Execution Libraries:
    - Operator implementations, UDF/UDAF/UDTF
    - SerDe & ObjectInspector, Metastore
    - FileFormat & RecordReader

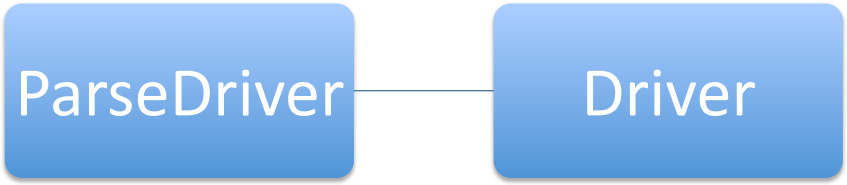

HIVE
HADOOP
MAP/REDUCE
8: sendResults
6.1: executeJob
6.2: jobDone
EXECUTION ENGINE
JOB TRACKER
6: executePlan
6.3 dfs operations
MAP/REDUCE TASKS
TASK TRACKERS (MAP)
TASK TRACKERS (REDUCE)
1: executeQuery
UI
DRIVER
7: fetchResults
6.1: metaDataOps for DDLs
MAP OPERATOR TREE
SERDE DESERIALIZE
REDUCE OPERATOR TREE
SERDE SERIALIZE
2: getPlan
5: sendPlan
READS/WRITES TO HDFS
HDFS
3: getMetaData
COMPILER
METASTORE
4: sendMetaData
NAME NODE
DATA NODES
9: fetchResults

Hive User/Application Interfaces



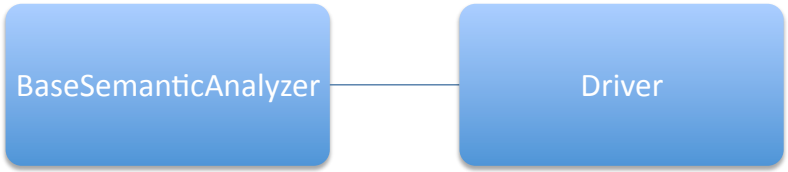
*HiPal is a Web-basedHive client developed internally at Facebook.

Parser



- ANTLR is a parser generator.
- $HIVE_SRC/ql/src/java/org/apache/hadoop/hive/ql/parse/Hive.g
- Hive.g defines keywords, tokens, translations from HiveQL to AST (ASTNode.java)
- Every time Hive.g is changed, you need to ‘ant clean’ first and rebuild using ‘ant package’

Semantic Analyzer



- BaseSemanticAnalyzer is the base class for DDLSemanticAnalyzer and SemanticAnalyzer
  - SemanticAnalyzer handles queries, DML, and some DDL (create-table)
  - DDLSemanticAnalyzer handles alter table etc.

Logical Plan Generation

- SemanticAnalyzer.analyzeInternal() is the main funciton
  - doPhase1(): recursively traverse AST tree and check for semantic errors and gather metadata which is put in QB and QBParseInfo.
  - getMetaData(): query metastore and get metadata for the data sources and put them into QB and QBParseInfo.
  - genPlan(): takes the QB/QBParseInfo and AST tree and generate an operator tree.

Logical Plan Generation (cont.)

- genPlan() is recursively called for each subqueries (QB), and output the root of the operator tree.
- For each subquery, genPlan create operators "bottom-up"* staring from FROM→WHERE→GROUPBY→ORDERBY → SELECT
- In the FROM clause, generate a TableScanOperator for each source table, Then genLateralView() and genJoinPlan().

*Hive code actually names each leaf operator as "root" and its downstream operators as children.

Logical Plan Generation (cont.)

- genBodyPlan() is then called to handle WHERE-GROUPBY-ORDERBY-SELECT clauses.
  - genFilterPlan() for WHERE clause
  - genGroupByPalnMapAgg1MR/2MR() for map-side partial aggregation
  - genGroupByPlan1MR/2MR() for reduce-side aggregation
  - genSelectPlan() for SELECT-clause
  - genReduceSink() for marking the boundary between map/reduce phases.
  - genFileSink() to store intermediate results

Optimizer

- The resulting operator tree, along with other parsing info, is stored in ParseContext and passed to Optimizer.
- Optimizer is a set of Transformation rules on the operator tree.
- The transformation rules are specified by a regexp pattern on the tree and a Worker/Dispatcher framework.

Optimizer (cont.)

- Current rewrite rules include:
  - ColumnPruner
  - PredicatePushDown
  - PartitionPruner
  - GroupByOptimizer
  - SamplePruner
  - MapJoinProcessor
  - UnionProcessor
  - JoinReorder

Physical Plan Generation

- genMapRedWorks() takes the QB/QBParseInfo and the operator tree and generate a DAG of MapReduceTasks.
- The generation is also based on the Worker/ Dispatcher framework while traversing the operator tree.
- Different task types: MapRedTask, ConditionalTask, FetchTask, MoveTask, DDLTask, CounterTask
- Validate() on the physical plan is called at the end of Driver.compile().

Preparing Execution

- Driver.execute takes the output from Driver.compile and prepare hadoop command line (in local mode) or call ExecDriver.execute (in remote mode).
  - Start a session
  - Execute PreExecutionHooks
  - Create a Runnable for each Task that can be executed in parallel and launch Threads within a certain limit
  - Monitor Thread status and update Session
  - Execute PostExecutionHooks

Preparing Execution (cont.)

- Hadoop jobs are started from MapRedTask.execute().
  - Get info of all needed JAR files with ExecDriver as the starting class
  - Serialize the Physical Plan (MapRedTask) to an XML file
  - Gather other info such as Hadoop version and prepare the hadoop command line
  - Execute the hadoop command line in a separate process.

Starting Hadoop Jobs

- ExecDriver deserialize the plan from the XML file and call execute().
- Execute() set up # of reducers, job scratch dir, the starting mapper class (ExecMapper) and starting reducer class (ExecReducer), and other info to JobConf and submit the job through hadoop.mapred.JobClient.
- The query plan is again serialized into a file and put into DistributedCache to be sent out to mappers/ reducers before the job is started.

Operator

- ExecMapper create a MapOperator as the parent of all root operators in the query plan and start executing on the operator tree.
- Each Operator class comes with a descriptor class, which contains metadata passing from compilation to execution.
  - Any metadata variable that needs to be passed should have a public setter & getter in order for the XML serializer/deserializer to work.
- The operator's interface contains:
  - initilize(): called once per operator lifetime
  - startGroup() *:called once for each group (groupby/join)
  - process()*: called one for each input row.
  - endGroup()*: called once for each group (groupby/join)
  - close(): called once per operator lifetime

Example: adding Semijoin operator

- Left semijoin is similar to inner join except only the left hand side table is output and no duplicated join values if there are duplicated keys in RHS table.
  - IN/EXISTS semantics
  - SELECT * FROM S LEFT SEMI JOIN T ON S.KEY = T.KEY AND S.VALUE = T.VALUE
  - Output all columns in table S if its (key,value) matches at least one (key,value) pair in T

Semijoin Implementation

- Parser: adding the SEMI keyword
- SemanticAnalyzer:
  - doPhase1(): keep a mapping of the RHS table name and its join key columns in QBParseInfo.
  - genJoinTree: set new join type in joinDesc.joinCond
  - genJoinPlan:
    - generate a map-side partial groupby operator right after the TableScanOperator for the RHS table. The input & output columns of the groupby operator is the RHS join keys.

Semijoin Implementation (cont.)

- SemanticAnalyzer
  - genJoinOperator: generate a JoinOperator (left semi type) and set the output fields as the LHS table's fields
- Execution
  - In CommonJoinOperator, implement left semi join with early-exit optimization: as long as the RHS table of left semi join is non-null, return the row from the LHS table.

Debugging

- Debugging compile-time code (Driver till ExecDriver) is relatively easy since it is running on the JVM on the client side.
- Debugging execution time code (after ExecDriver calls hadoop) need some configuration. See wiki http://wiki.apache.org/hadoop/Hive/DeveloperGuide#Debugging_Hive_code

facebook

Questions?