SPARK+AI
SUMMIT 2019

How to extend Spark with customized optimizations

Sunitha Kambhampati, IBM

#UnifiedAnalytics #SparkAISummit

Center for Open Source Data and AI Technologies

CODAIT aims to make AI solutions dramatically easier to create, deploy, and manage in the enterprise.

Relaunch of the IBM Spark Technology Center (STC) to reflect expanded mission.

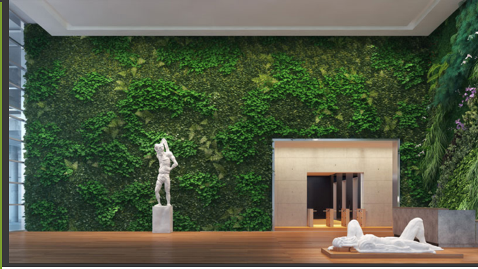
We contribute to foundational open source software across the enterprise AI lifecycle.

36 open-source developers!

https://ibm.biz/BdzF6Q

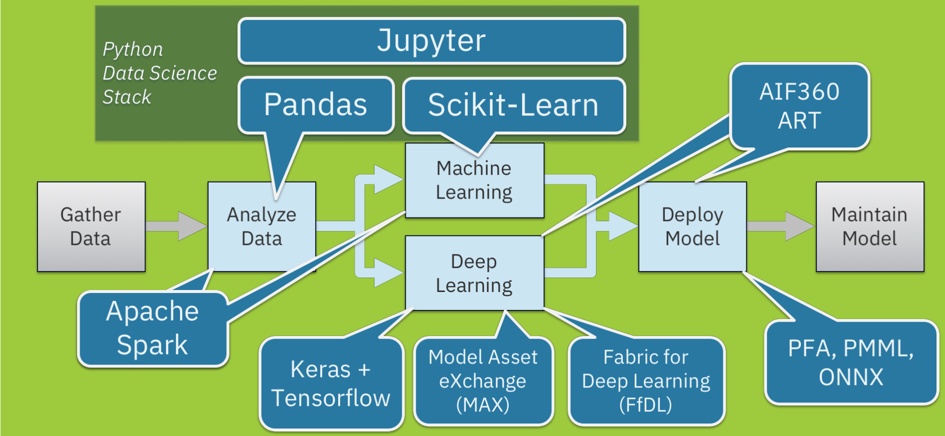
CODAIT

codait.org



IBM Watson West Building
505 Howard St.
San Francisco, CA

Improving Enterprise AI Lifecycle in Open Source



Python Data Science Stack

Jupyter

Pandas

Scikit-Learn

AIF360
ART

Gather Data

Analyze Data

Machine Learning

Deep Learning

Deploy Model

Maintain Model

Apache Spark

Keras + Tensorflow

Model Asset eXchange (MAX)

Fabric for Deep Learning (FfDL)

PFA, PMML, ONNX

Agenda

- Introduce Spark Extension Points API

- Deep Dive into the details
  - What you can do
  - How to use it
  - What things you need to be aware of

- Enhancements to the API
  - Why
  - Performance results

I want to extend Spark

- Performance benefits
  - Support for *informational referential integrity (RI)* constraints
  - Add Data Skipping Indexes
- Enabling Third party applications
  - Application uses Spark but it requires some additions or small changes to Spark

Problem

You have developed customizations to Spark.

How do you add it to your Spark cluster?

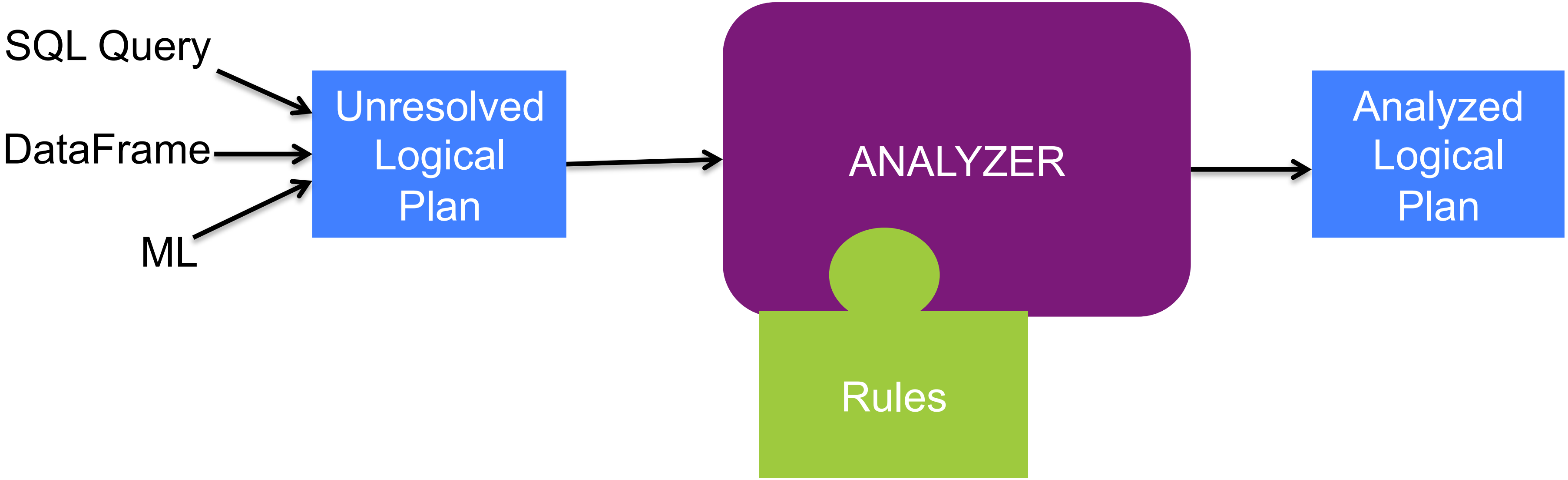
Possible Solutions

- Option 1: Get the code merged to Apache Spark
  - Maybe it is application specific
  - Maybe it is a value add
  - Not something that can be merged into Spark
- Option 2: Modify Spark code, fork it
  - Maintenance overhead
- Extensible solution: Use Spark's Extension Points API

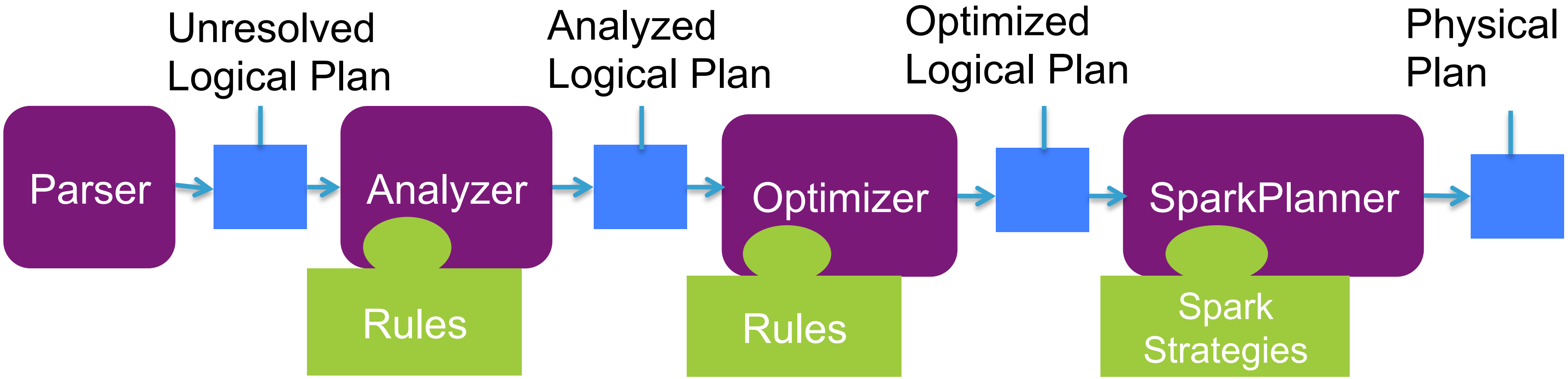
Spark Extension Points API

- Added in Spark 2.2 in SPARK-18127
- Pluggable & Extensible
- Extend SparkSession with custom optimizations
- Marked as Experimental API
  - relatively stable
  - has not seen any changes except addition of more customization

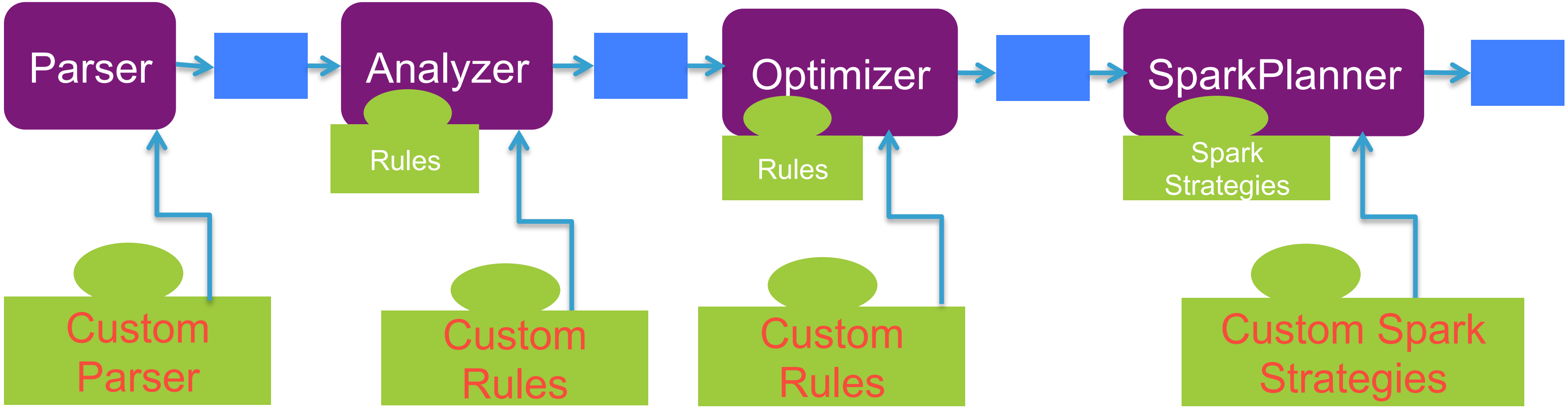
Query Execution



Query Execution



Supported Customizations



Extensions API: At a High level

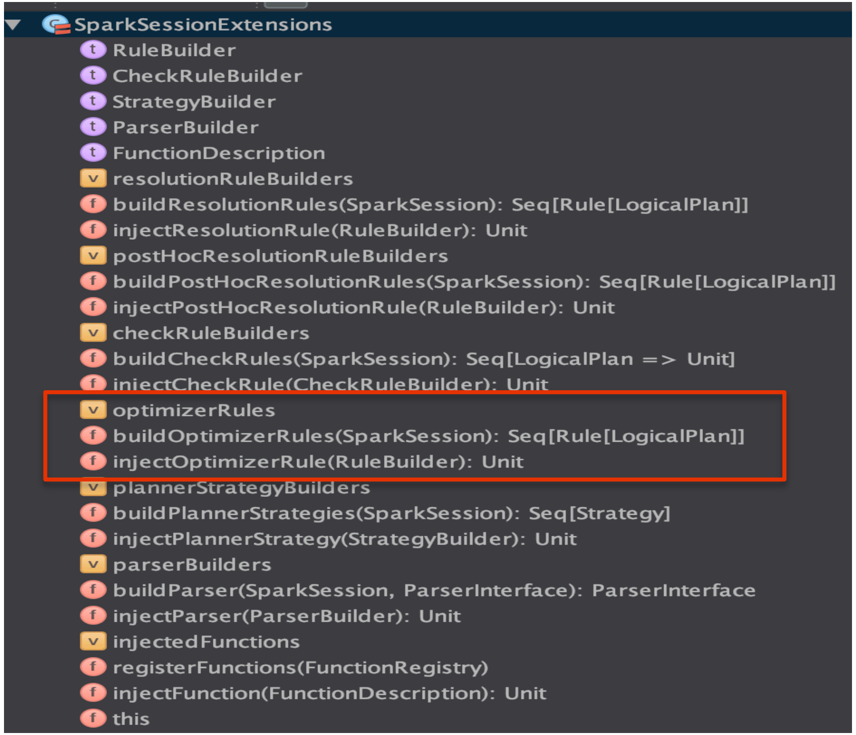
- New SparkSessionExtensions Class
  - Methods to pass the customizations
  - Holds the customizations
- Pass customizations to Spark
  - withExtensions method in SparkSession.builder

SparkSessionExtensions

- @DeveloperApi
  @Experimental
  @InterfaceStability.Unstable

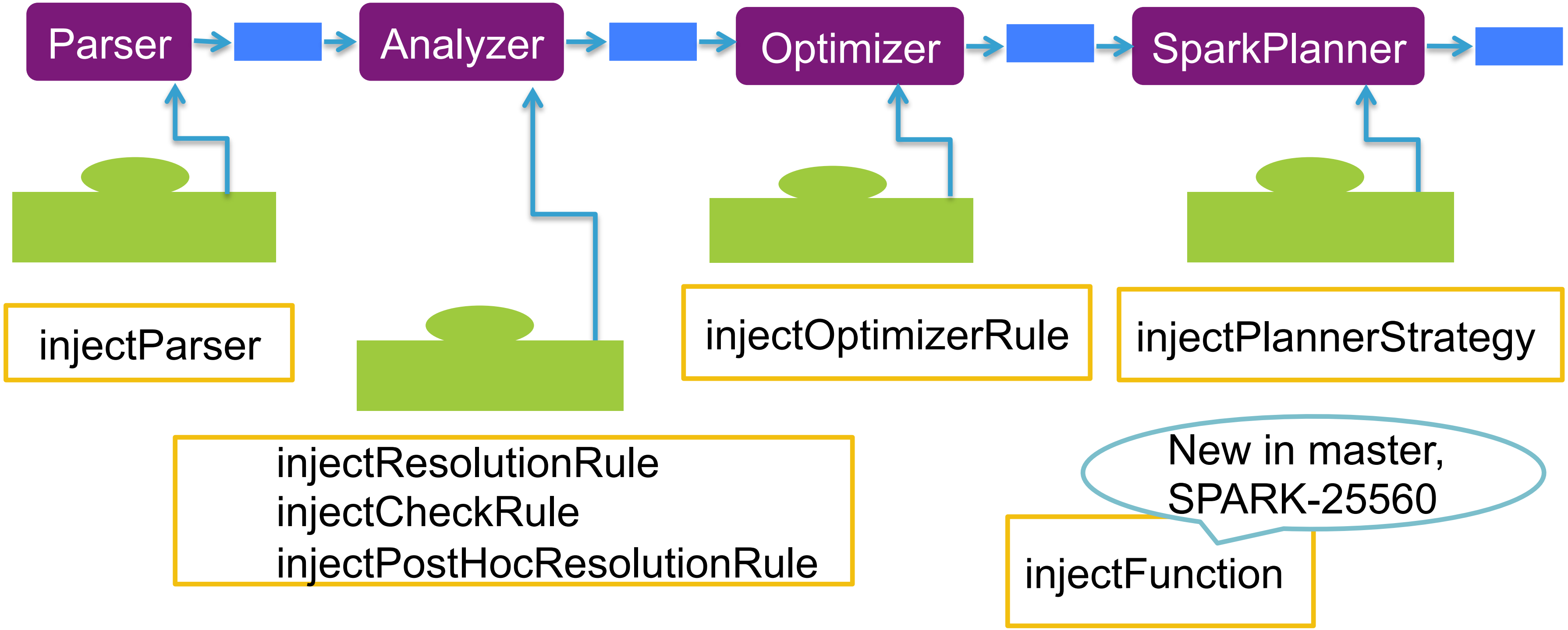
- Inject Methods
  – Pass the custom user rules to Spark

- Build Methods
  – Pass the rules to Spark components
  – Used by Spark Internals



```
SparkSessionExtensions
  t RuleBuilder
  t CheckRuleBuilder
  t StrategyBuilder
  t ParserBuilder
  t FunctionDescription
  v resolutionRuleBuilders
  f buildResolutionRules(SparkSession): Seq[Rule[LogicalPlan]]
  f injectResolutionRule(RuleBuilder): Unit
  v postHocResolutionRuleBuilders
  f buildPostHocResolutionRules(SparkSession): Seq[Rule[LogicalPlan]]
  f injectPostHocResolutionRule(RuleBuilder): Unit
  v checkRuleBuilders
  f buildCheckRules(SparkSession): Seq[LogicalPlan => Unit]
  f injectCheckRule(CheckRuleBuilder): Unit
  v optimizerRules
  f buildOptimizerRules(SparkSession): Seq[Rule[LogicalPlan]]
  f injectOptimizerRule(RuleBuilder): Unit
  v plannerStrategyBuilders
  f buildPlannerStrategies(SparkSession): Seq[Strategy]
  f injectPlannerStrategy(StrategyBuilder): Unit
  v parserBuilders
  f buildParser(SparkSession, ParserInterface): ParserInterface
  f injectParser(ParserBuilder): Unit
  v injectedFunctions
  f registerFunctions(FunctionRegistry)
  f injectFunction(FunctionDescription): Unit
  f this
```

Extension Hooks: Inject Methods



Parser

Analyzer

Optimizer

SparkPlanner

injectParser

injectResolutionRule
injectCheckRule
injectPostHocResolutionRule

injectOptimizerRule

injectPlannerStrategy

New in master,
SPARK-25560

injectFunction

Pass custom rules to SparkSession

- Use ‘withExtensions’ in SparkSession.Builder

```
def withExtensions(
      f: SparkSessionExtensions => Unit): Builder
```

- Use the Spark configuration parameter
  – spark.sql.extensions
    - Takes a class name that implements `Function1[SparkSessionExtensions, Unit]`

Deep Dive

Use Case #1

You want to add your own optimization rule to Spark's Catalyst Optimizer

Add your custom optimizer rule

- Step 1: Implement your optimizer rule

```
case class GroupByPushDown(spark: SparkSession) extends Rule[LogicalPlan]  {
    def apply(plan: LogicalPlan): LogicalPlan = plan transform {
      …. }}
```

- Step 2: Create your ExtensionsBuilder function

```
type ExtensionsBuilder = SparkSessionExtensions => Unit
val f: ExtensionsBuilder = { e => e.injectOptimizerRule(GroupByPushDown)}
```

- Step 3: Use the *withExtensions* method in *SparkSession.builder* to create your custom SparkSession

```
val spark = SparkSession.builder().master(..).withExtensions(f).getOrCreate()
```

How does the rule get added?

- Catalyst Optimizer
  - Rules are grouped in Batches (ie RuleExecutor.Batch)
  - one of the fixed batch has a placeholder to add custom optimizer rules
  - passes in the **`extendedOperatorOptimizationRules`** to the batch.

```
def extendedOperatorOptimizationRules: Seq[Rule[LogicalPlan]]
```

- SparkSession stores the SparkSessionExtensions in transient class variable *`extensions`*

- The `SparkOptimizer` instance gets created during the `SessionState` creation for the `SparkSession`
  - overrides the `extendedOperatorOptimizationRules` method to include the customized rules
  - Check the `optimizer` method in `BaseSessionStateBuilder`

Things to Note

- Rule gets added to a predefined batch
- Batch here refers to `RuleExecutor.Batch`
- In Master, it is to the following batches:
  - "*Operator Optimization before Inferring Filters*"
  - "*Operator Optimization after Inferring Filters*"
- Check the `defaultBatches` method in `Optimizer` class

Use Case #2

You want to add some parser extensions

Parser Customization

- Step 1: Implement your parser customization

```
case class RIExtensionsParser(
    spark: SparkSession,
    delegate: ParserInterface) extends ParserInterface { …}
```

- Step 2: Create your ExtensionsBuilder function

```
type ExtensionsBuilder = SparkSessionExtensions => Unit
val f: ExtensionsBuilder = { e => e.injectParser(RIExtensionsParser)}
```

- Step 3: Use the *withExtensions* method in *SparkSession.builder* to create your custom SparkSession

```
val spark = SparkSession.builder().master("…").withExtensions(f).getOrCreate()
```

How do the parser extensions work?

- Customize the parser for any new syntax to support
- Delegate rest of the Spark SQL syntax to the `SparkSqlParser`

- sqlParser is created by calling the buildParser on the extensions object in the SparkSession
  - See `sqlParser` in `BaseSessionStateBuilder` class
  - `SparkSqlParser` (Default Spark Parser) is passed in along with the `SparkSession`

Use Case #3

You want to add some specific checks in the Analyzer

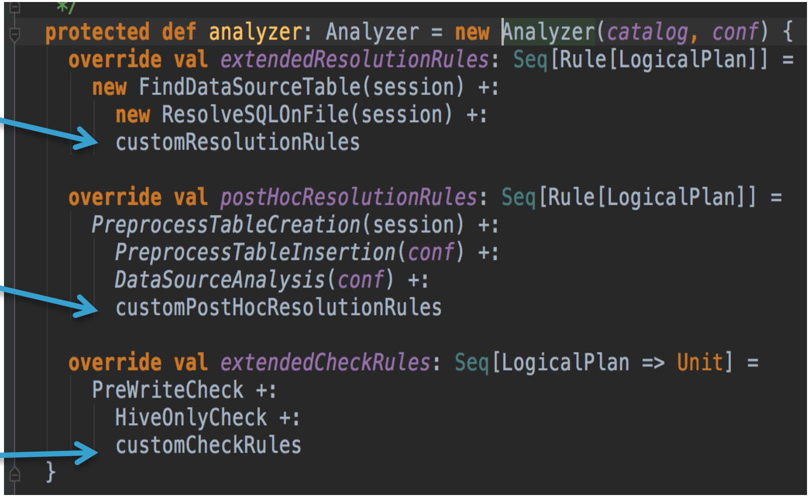
Analyzer Customizations

- Analyzer Rules

  injectResolutionRule
- PostHocResolutionRule

  injectPostHocResolutionRule
- CheckRules

  injectCheckRule



```
  */
protected def analyzer: Analyzer = new Analyzer(catalog, conf) {
  override val extendedResolutionRules: Seq[Rule[LogicalPlan]] =
    new FindDataSourceTable(session) +:
      new ResolveSQLOnFile(session) +:
      customResolutionRules

  override val postHocResolutionRules: Seq[Rule[LogicalPlan]] =
    PreprocessTableCreation(session) +:
      PreprocessTableInsertion(conf) +:
      DataSourceAnalysis(conf) +:
      customPostHocResolutionRules

  override val extendedCheckRules: Seq[LogicalPlan => Unit] =
    PreWriteCheck +:
      HiveOnlyCheck +:
      customCheckRules
}
```

Analyzer Rule Customization

- Step 1: Implement your Analyzer rule

```
case class MyRIRule(spark: SparkSession) extends Rule[LogicalPlan]  {
    def apply(plan: LogicalPlan): LogicalPlan = plan transform {
      .... }}
```

- Step 2: Create your ExtensionsBuilder function

```
type ExtensionsBuilder = SparkSessionExtensions => Unit
val f: ExtensionsBuilder = { e => e.injectResolutionRule(MyRIRule)}
```

- Step 3: Use the *withExtensions* method in *SparkSession.builder* to create your custom SparkSession

```
val spark =
SparkSession.builder().master("..").withExtensions(f).getOrCreate
```

How is the rule added to the Analyzer?

- Analyzer has rules in batches
  - Batch has a placeholder `extendedResolutionRules` to add custom rules
  - Batch “**Post-Hoc Resolution**” for **`postHocResolutionRules`**

- SparkSession stores the SparkSessionExtensions in `extensions`

- When SessionState is created, the custom rules are passed to the Analyzer by overriding the following class member variables
  - `val extendedResolutionRules`
  - `val postHocResolutionRules`
  - `val extendedCheckRules`
- Check the `BaseSessionStateBuilder.analyzer` method
- Check the `HiveSessionStateBuilder.analyzer` method

Things to Note

- Custom resolution rule gets added in the end to ‘**Resolution**’ Batch
- The `checkRules` will get called in the end of the `checkAnalysis` method after all the spark checks are done
- In Analyzer.checkAnalysis method: *`extendedCheckRules`*`.foreach(_(plan))`

Use Case #4

You want to add custom planning strategies

Add new physical plan strategy

- Step1: Implement your new physical plan Strategy class

```
case class IdxStrategy(spark: SparkSession) extends SparkStrategy {
   override def apply(plan: LogicalPlan): Seq[SparkPlan] = { ….. }
}
```

- Step 2: Create your ExtensionsBuilder function

```
type ExtensionsBuilder = SparkSessionExtensions => Unit
val f: ExtensionsBuilder = { e => e.injectPlannerStrategy(IdxStrategy)}
```

- Step 3: Use the *withExtensions* method in *SparkSession.builder* to create your custom SparkSession

```
val spark = SparkSession.builder().master(..).withExtensions(f).getOrCreate()
```

How does the strategy get added

- SparkPlanner uses a Seq of SparkStrategy
  - `strategies` function has a placeholder `extraPlanningStrategies`

- SparkSession stores the SparkSessionExtensions in transient class variable `extensions`

- The `SparkPlanner` instance gets created during the `SessionState` creation for the `SparkSession`
  - overrides the `extraPlanningStrategies` to include the custom strategy (`buildPlannerStrategies`)
  - Check the `BaseSessionStateBuilder.planner` method
  - Check the `HiveSessionStateBuilder.planner` method

Things to Note

- Custom Strategies are tried after the strategies defined in `ExperimentalMethods`, and before the regular strategies
  - Check the `SparkPlanner.strategies` method

Use Case #5

You want to register custom functions in the session catalog

Register Custom Function

- Step 1: Create a FunctionDescription with your custom function

```
type FunctionDescription =
  (FunctionIdentifier, ExpressionInfo, FunctionBuilder)

 def utf8strlen(x: String): Int = {..}
 val f = udf(utf8strlen(_))
 def builder(children: Seq[Expression]) =
 f.apply(children.map(Column.apply) : _*).expr

 val myfuncDesc = (FunctionIdentifier("utf8strlen"),
  new ExpressionInfo("noclass", "utf8strlen"), builder)
```

Register Custom Function

- Step 2: Create your ExtensionsBuilder function to inject the new function

```
type ExtensionsBuilder = SparkSessionExtensions => Unit
val f: ExtensionsBuilder = { e => e.injectFunction (myfuncDesc)}
```

- Step 3: Pass this function to withExtensions method on SparkSession.builder and create your new SparkSession

```
val spark =
SparkSession.builder().master(..).withExtensions(f).getOrCreate()
```

How does Custom Function registration work

- SparkSessionExtensions keeps track of the injectedFunctions
- During SessionCatalog creation, the injectedFunctions are registered in the `functionRegistry`
  - See class variable `BaseSessionStateBuilder.functionRegistry`
  - See method `SimpleFunctionRegistry.registerFunction`

Things to Note

- Function registration order is same as the order in which the injectFunction is called
- No check if an existing function already exists during the injection
- A warning is raised if a function replaces an existing function
  - Check is based on lowercase match of the function name
- Use the `SparkSession.catalog.listFunctions` to look up your function
- The functions registered will be temporary functions
- See `SimpleFunctionRegistry.registerFunction` method

How to exclude the optimizer rule

- Spark v2.4 has new SQL Conf: `spark.sql.optimizer.excludedRules`
- Specify the custom rule’s class name

```
session.conf.set(
"spark.sql.optimizer.excludedRules",
"org.mycompany.spark.MyCustomRule")
```

Other ways to customize

- ExperimentalMethods
  - Customize Physical Planning Strategies
  - Customize Optimizer Rules
- Use the `SparkSession.experimental` method
  - *`spark.experimental.extraStrategies`*
    - Added in the **beginning** of strategies in SparkPlanner
  - *`spark.experimental.extraOptimizations`*
    - Added **after all** the batches in SparkOptimizer

Things to Note

- ExperimentalMethods
  - Rules are injected in a different location than Extension Points API
  - So use this only if it is advantageous for your usecase
- Recommendation: Use Extension Points API

Proposed API Enhancements

SPARK-26249: API Enhancements

- Motivation
  - Lack of fine grained control on rule execution order
  - Add batches in a specific order
- Add support to extensions API
  - Inject optimizer rule in a specific order
  - Inject optimizer batch

Inject Optimizer Rule in Order

- Inject a rule after or before an existing rule in a given existing batch in the Optimizer

```
def injectOptimizerRuleInOrder(
    builder: RuleBuilder,
    batchName: String,
    ruleOrder: Order.Order,
    existingRule: String): Unit
```

Inject Optimizer Batch

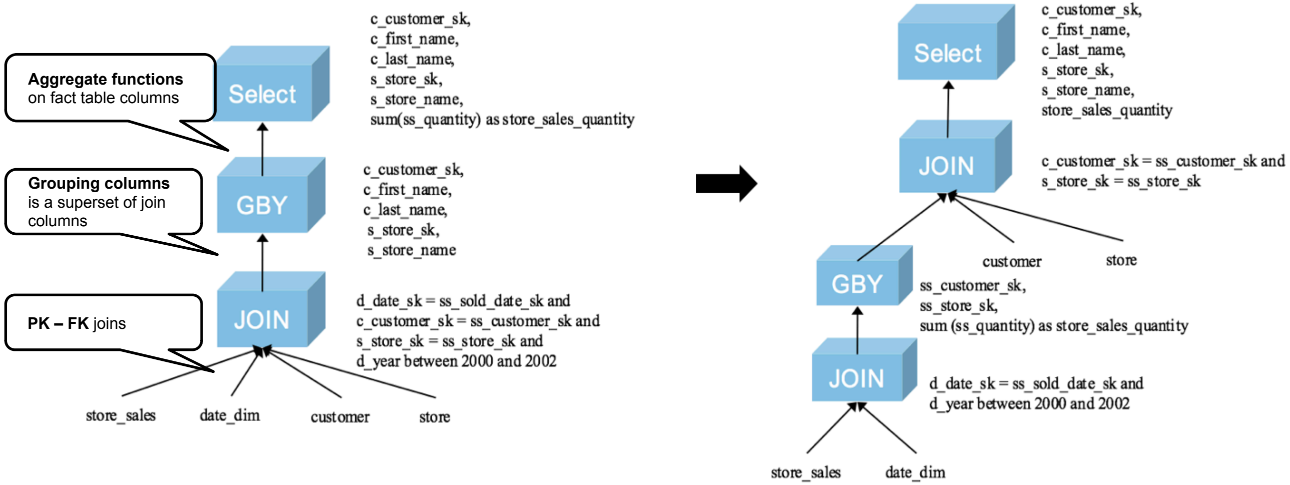
- Inject a batch of optimizer rules
- Specify the order where you want to inject the batch

```
def injectOptimizerBatch(
      batchName: String,
      maxIterations: Int,
      existingBatchName: String,
      order: Order.Value,
      rules: Seq[RuleBuilder]): Unit
```

End to End Use Case

Use case: GroupBy Push Down Through Join

- If the join is an ***RI join***, heuristically push down Group By to the fact table
  - The input to the Group By remains the same before and after the join
  - The input to the join is reduced
  - Overall reduction of the execution time



Aggregate functions on fact table columns

Grouping columns is a superset of join columns

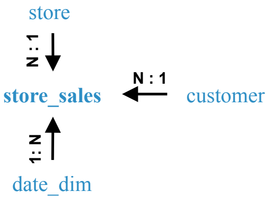
PK – FK joins

Group By Push Down Through Join

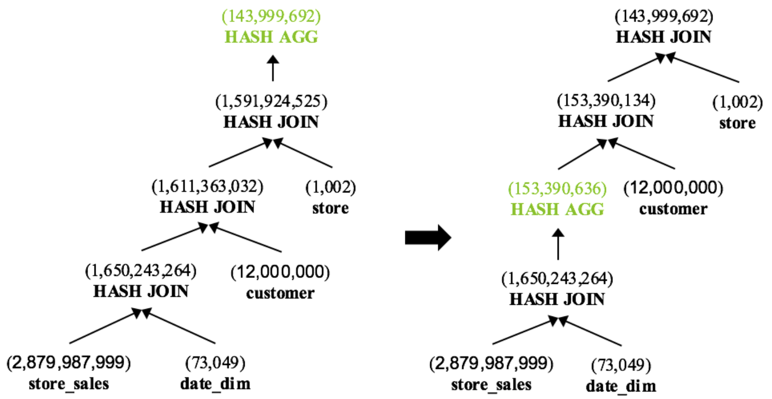
```
select c_customer_sk, c_first_name, c_last_name, s_store_sk, s_store_name,
          min(ss.ss_quantity) as store_sales_quantity
from store_sales ss, date_dim, customer, store
where d_date_sk  = ss_sold_date_sk and
          c_customer_sk  = ss_customer_sk and
          s_store_sk  = ss_store_sk and
         d_year between 2000 and 2002
group by c_customer_sk, c_first_name, c_last_name, s_store_sk, s_store_name
order by c_customer_sk, c_first_name, c_last_name, s_store_sk, s_store_name
limit 100;
```

Retrieve the minimum quantity of items that were sold between the year 2000 and 2002 grouped by customer and store information

Star schema:



Execution plan transformation:



- Query execution drops from 70 secs to 30 secs (1TB TPC-DS setup), 2x improvement

Optimized Query Plan: Explain

```
== Optimized Logical Plan ==
GlobalLimit 100
+- LocalLimit 100
   +- Sort [c_customer_sk#52 ASC NULLS FIRST, c_first_name#60 ASC NULLS FIRST, c_last_name#61 ASC NULLS FIRST, s_store_sk#70 ASC NULLS FIRST, s_store_name#75 ASC NULLS FIRST],
true
      +- Project [c_customer_sk#52, c_first_name#60, c_last_name#61, s_store_sk#70, s_store_sk#75, store_sales_quantity#0 ]
         +- Join Inner, (s_store_sk#70 = ss_store_sk#8)
            :- Project [c_customer_sk#52, c_first_name#60, c_last_name#61, ss_store_sk#8, store_sales_quantity#0L]
            :  +- Join Inner, (c_customer_sk#52 = ss_customer_sk#4)
            :     :- Aggregate [ss_customer_sk#4, ss_store_sk#8], [ss_customer_sk#4, ss_store_sk#8, min(ss_quantity#11L) AS store_ ales_quantity#0L]
            :     :  +- Project [ss_customer_sk#4, ss_store_sk#8, ss_quantity#11L]
            :     :     +- Join Inner, (d_date_sk#24 = ss_sold_date_sk#1)
            :     :        :- Project [ss_sold_date_sk#1, ss_customer_sk#4, ss_store_sk#8, ss_quantity#11L]
            :     :        :  +- Filter ((isnotnull(ss_sold_date_sk#1) && isnotnull(ss_customer_sk#4)) && isnotnull(ss_store_sk#8))
            :     :        :     +-
Relation[ss_sold_date_sk#1,ss_sold_time_sk#2,ss_item_sk#3,ss_customer_sk#4,ss_cdemo_sk#5,ss_hdemo_sk#6,ss_addr_sk#7,ss_store_sk#8,ss_promo_sk#9,ss_ticket_number#10L,ss_quantity#11L,s
s_wholesale_cost#12,ss_list_price#13,ss_sales_price#14,ss_ext_discount_amt#15,ss_ext_sales_price#16,ss_ext_wholesale_cost#17,ss_ext_list_price#18,ss_ext_tax#19,ss_coupon_amt#20,ss_net_paid
#21,ss_net_paid_inc_tax#22,ss_net_profit#23] parquet
            :     :        +- Project [d_date_sk#24]
            :     :           +- Filter (((isnotnull(d_year#30L) && (d_year#30L >= 2000)) && (d_year#30L <= 2002)) && isnotnull(d_date_sk#24))
            :     :              +-
Relation[d_date_sk#24,d_date_id#25,d_date#26,d_month_seq#27L,d_week_seq#28L,d_quarter_seq#29L,d_year#30L,d_dow#31L,d_moy#32L,d_dom#33L,d_qoy#34L,d_fy_year#35L,d_fy_quarter_seq#
36L,d_fy_week_seq#37L,d_day_name#38,d_quarter_name#39,d_holiday#40,d_weekend#41,d_following_holiday#42,d_first_dom#43L,d_last_dom#44L,d_same_day_ly#45L,d_same_day_lq#46L,d_curre
nt_day#47,... 4 more fields] parquet
            :     +- Project [c_customer_sk#52, c_first_name#60, c_last_name#61]
            :        +- Filter isnotnull(c_customer_sk#52)
            :           +-
Relation[c_customer_sk#52,c_customer_id#53,c_current_cdemo_sk#54,c_current_hdemo_sk#55,c_current_addr_sk#56,c_first_shipto_date_sk#57,c_first_sales_date_sk#58,c_salutation#59,c_first_name
#60,c_last_name#61,c_preferred_cust_flag#62,c_birth_day#63L,c_birth_month#64L,c_birth_year#65L,c_birth_country#66,c_login#67,c_email_address#68,c_last_review_date#69L] parquet
            +- Project [s_store_sk#70, s_store_name#75]
               +- Filter isnotnull(s_store_sk#70)
                  +-
Relation[s_store_sk#70,s_store_id#71,s_rec_start_date#72,s_rec_end_date#73,s_closed_date_sk#74,s_store_name#75,s_number_employees#76L,s_floor_space#77L,s_hours#78,s_manager#79,s_mar
ket_id#80L,s_geography_class#81,s_market_desc#82,s_market_manager#83,s_division_id#84L,s_division_name#85,s_company_id#86L,s_company_name#87,s_street_number#88,s_street_name#89,s
_street_type#90,s_suite_number#91,s_city#92,s_county#93,... 5 more fields] parquet
```

Group By is pushed below Join

Benefits of the Proposed Changes

- Implemented new GroupByPushDown optimization rule
  - Benefit from RI constraints
- Used the Optimizer Customization
- Injected using `injectOptimizerRuleInOrder`

```
e.injectOptimizerRuleInOrder(
   GroupByPushDown,
  "Operator Optimization before Inferring Filters",
  Order.after,
  "org.apache.spark.sql.catalyst.optimizer.PushDownPredicate")
```

- Achieved **2X** performance improvements

Recap: How to Extend Spark

- Use the Extension Points API
- Five Extension Points
- To add a rule is a 3 step process
  - Implement your rule
  - Implement your wrapper function, use right inject method

    type ExtensionsBuilder = SparkSessionExtensions => Unit
  - Plug in the wrapper function

    withExtensions method in SparkSession.Builder

Resources

- https://developer.ibm.com/code/2017/11/30/learn-extension-points-apache-spark-extend-spark-catalyst-optimizer/
- https://rtahboub.github.io/blog/2018/writing-customized-parser/
- https://github.com/apache/spark/blob/master/sql/core/src/test/scala/org/apache/spark/sql/SparkSessionExtensionSuite.scala
- https://issues.apache.org/jira/browse/SPARK-18127
- https://issues.apache.org/jira/browse/SPARK-26249
- http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf

Thank you!

IBM Cloud

Build for free on

IBM Cloud

https://ibm.biz/Bd2GbF