INTRODUCTION TO
APACHE CALCITE

JORDAN HALTERMAN

WHAT IS APACHE CALCITE?

next

What is Apache Calcite?

- A framework for building SQL databases
- Developed over more than ten years
- Written in Java
- Previously known as Optiq
- Previously known as Farrago
- Became an Apache project in 2013
- Led by Julian Hyde at Hortonworks

Projects using Calcite

- Apache Hive
- Apache Drill
- Apache Flink
- Apache Phoenix
- Apache Samza
- Apache Storm
- Apache everything…

What is Apache Calcite?

- SQL parser
- SQL validation
- Query optimizer
- SQL generator
- Data federator

Stages of query execution

01

Parse

Queries are parsed using a JavaCC generated parser

02

Validate

Queries are validated against known database metadata

03

Optimize

Logical plans are optimized and converted into physical expressions

04

Execute

Physical plans are converted into application-specific executions

COMPONENTS

Components of Calcite

- **Catalog** - Defines metadata and namespaces that can be accessed in SQL queries
- **SQL parser** - Parses valid SQL queries into an abstract syntax tree (AST)
- **SQL validator** - Validates abstract syntax trees against metadata provided by the catalog
- **Query optimizer** - Converts AST into logical plans, optimizes logical plans, and converts logical expressions into physical plans
- **SQL generator** - Converts physical plans to SQL

CATALOG

next

Calcite Catalog

- Defines namespaces that can be accessed in Calcite queries
- **Schema**
  - A collection of schemas and tables
  - Can be arbitrarily nested
- **Table**
  - Represents a single data set
  - Fields defined by a `RelDataType`
- **RelDataType**
  - Represents fields in a data set
  - Supports all SQL data types, including structs and

Schema

- A collection of schemas and tables
- Schemas can be arbitrarily nested

Schema

- A collection of schemas and tables
- Schemas can be arbitrarily nested

```
public interface Schema {

  Table getTable(String name);

  Set<String> getTableNames();

  Schema getSubSchema(String name);

  Set<String> getSubSchemaNames();

}
```

Table

- Represents a single data set
- Fields are defined by a `RelDataType`

Table

- Represents a single data set
- Fields are defined by a **`RelDataType`**

```
public interface Table {
  RelDataType getRowType(RelDataTypeFactory typeFactory);

  Statistic getStatistic();

  Schema.TableType getJdbcTableType();
}
```

RelDataType

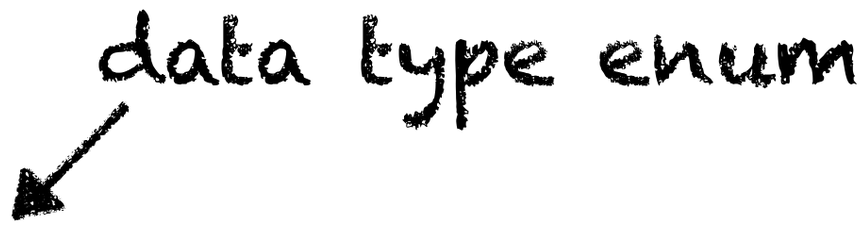
- Represents the data type of an object
- Supports all SQL data types, including structs and arrays
- Similar to Spark's `DataType`

RelDataType

```
public interface RelDataType {

  List<RelDataTypeField> getFieldList();

  boolean isNullable();

  RelDataType getComponentType();

  RelDataType getKeyType();

  RelDataType getValueType();

  Charset getCharset();

  int getPrecision();

  int getScale();

  SqlTypeName getSqlTypeName();

}
```

RelDataType

```
public interface RelDataType {

  List<RelDataTypeField> getFieldList();

  boolean isNullable();

  RelDataType getComponentType();

  RelDataType getKeyType();

  RelDataType getValueType();

  Charset getCharset();

  int getPrecision();

  int getScale();

  SqlTypeName getSqlTypeName();

}
```



Statistic

- Provide table statistics used in optimization

Statistic

- Provide table statistics used in optimization

```
public interface Statistic {
  Double getRowCount();

  boolean isKey(ImmutableBitSet columns);

  List<RelCollation> getCollations();

  RelDistribution getDistribution();
}
```

Usage of the Calcite catalog

```
SELECT id, name, CAST(created_at AS DATE)
FROM redshift.users
```

Usage of the Calcite catalog

```
SELECT id, name, CAST(created_at AS DATE)
FROM redshift.users
```

schema

Usage of the Calcite catalog

```
SELECT id, name, CAST(created_at AS DATE)
FROM redshift.users
```

schema

table

Usage of the Calcite catalog

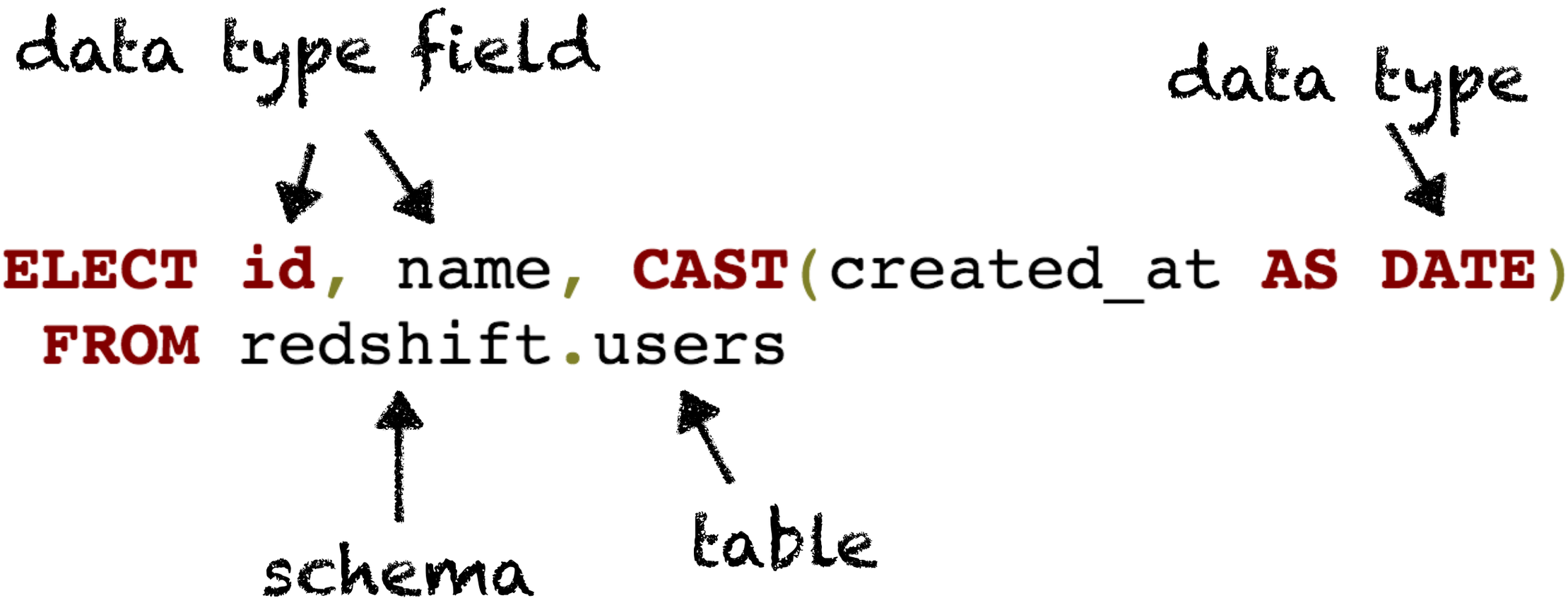
data type

```
SELECT id, name, CAST(created_at AS DATE)
FROM redshift.users
```

schema

table

Usage of the Calcite catalog



SQL PARSER

Calcite SQL parser

- LL(k) parser written in JavaCC
- Input queries are parsed into an abstract syntax tree (AST)
- Tokens are represented in Calcite by **`SqlNode`**
- **`SqlNode`** can also be converted back to a SQL string via the **`unparse`** method

JavaCC

- Java Compiler Compiler
- Created in 1996 at Sun Microsystems
- Generates Java code from a domain-specific language
- ANTLR is the modern alternative used in projects like Hive and Drill
- JavaCC has sparse documentation

JavaCC

```
CAST(created_at AS DATE)
```

CAST(created_at **AS DATE**)

```
<CAST>
<LPAREN>
e = Expression(ExprContext.ACCEPT_SUBQUERY)
<AS>
(
  dt = DataType() { args.add(dt); }
  |
  <INTERVAL> e = IntervalQualifier() { args.add(e); }
)
<RPAREN>
{
    return SqlApolloOperatorTable.CAST.createCall(
        pos.plus(getPos()), SqlParserUtil.toNodeArray(args));
}
```

JavaCC

```
CAST(created_at AS DATE)
```

tokens

```
<CAST>
<LPAREN>
e = Expression(ExprContext.ACCEPT_SUBQUERY)
<AS>
(
  dt = DataType() { args.add(dt); }
  |
  <INTERVAL> e = IntervalQualifier() { args.add(e); }
)
<RPAREN>
{
    return SqlApolloOperatorTable.CAST.createCall(
        pos.plus(getPos()), SqlParserUtil.toNodeArray(args));
}
```

JavaCC

```
CAST(created_at AS DATE)
```

tokens

```
<CAST>
<LPAREN>
e = Expression(ExprContext.ACCEPT_SUBQUERY)
<AS>
(
  dt = DataType() { args.add(dt); }
|
  <INTERVAL> e = IntervalQualifier() { args.add(e); }
)
<RPAREN>
{
    return SqlApolloOperatorTable.CAST.createCall(
        pos.plus(getPos()), SqlParserUtil.toNodeArray(args));
}
```

or →

JavaCC

CAST(created_at **AS DATE**)

tokens

```
<CAST>
<LPAREN>
e = Expression(ExprContext.ACCEPT_SUBQUERY)
<AS>
(
  dt = DataType() { args.add(dt); }
|
  <INTERVAL> e = IntervalQualifier() { args.add(e); }
)
<RPAREN>
{
    return SqlApolloOperatorTable.CAST.createCall(
        pos.plus(getPos()), SqlParserUtil.toNodeArray(args));
}
```

function call

or

JavaCC

CAST(created_at **AS DATE**)

tokens

```
<CAST>
<LPAREN>
e = Expression(ExprContext.ACCEPT_SUBQUERY)
<AS>
(
    dt = DataType() { args.add(dt); }
|
    <INTERVAL> e = IntervalQualifier() { args.add(e); }
)
<RPAREN>
{
    return SqlApolloOperatorTable.CAST.createCall(
        pos.plus(getPos()), SqlParserUtil.toNodeArray(args));
}
```

function call

or

Java code

SqlNode

- `SqlNode` represents an element in an abstract syntax tree

```
SELECT id, name, CAST(created_at AS DATE)
FROM redshift.users
```

SqlNode

- `SqlNode` represents an element in an abstract syntax tree

select

```
SELECT id, name, CAST(created_at AS DATE)
FROM redshift.users
```

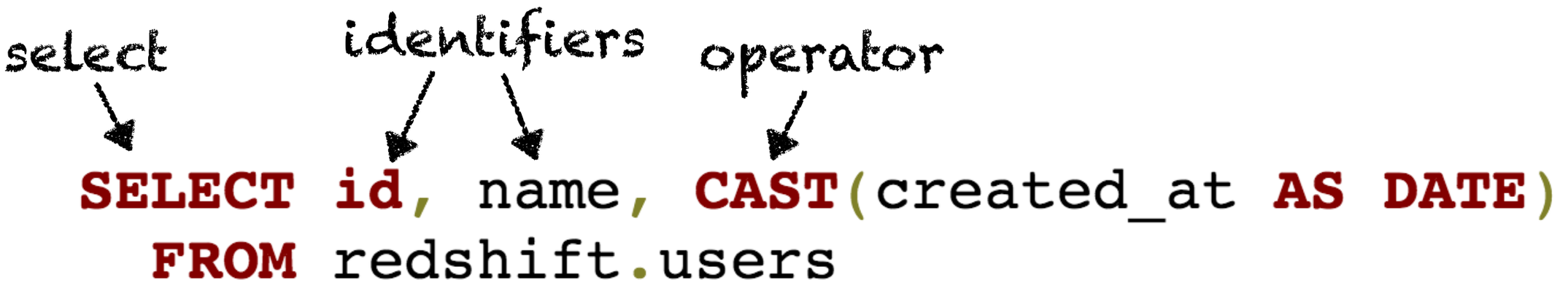
SqlNode

- `SqlNode` represents an element in an abstract syntax tree


```
SELECT id, name, CAST(created_at AS DATE)
FROM redshift.users
```

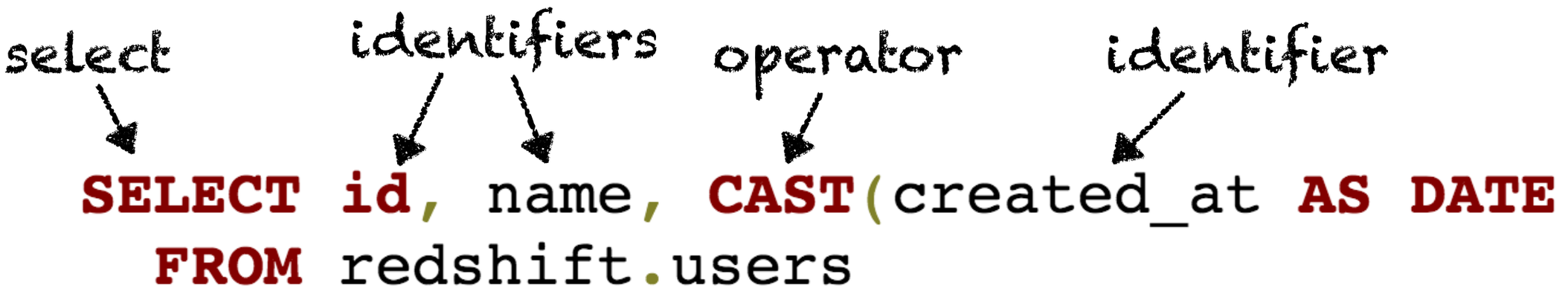
SqlNode

- `SqlNode` represents an element in an abstract syntax tree



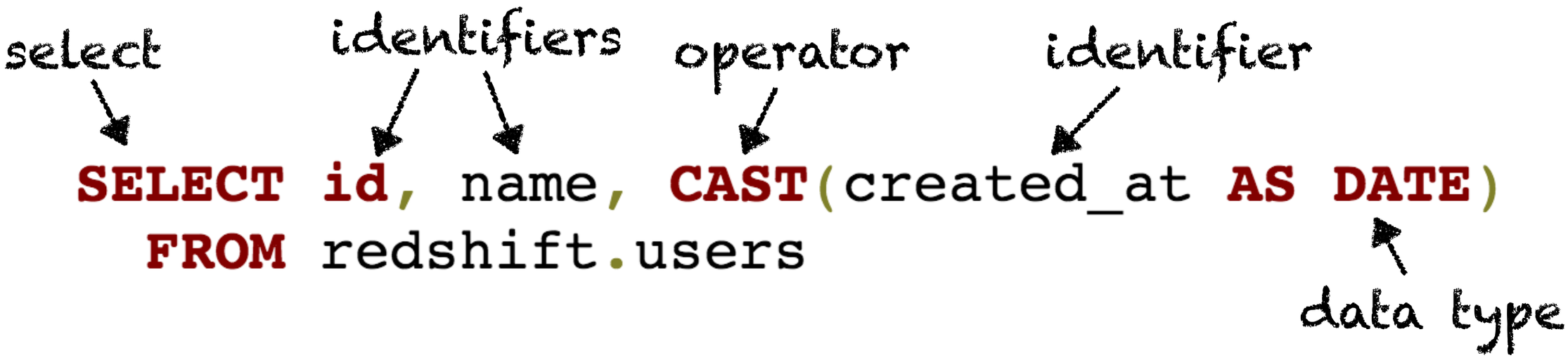
SqlNode

- `SqlNode` represents an element in an abstract syntax tree



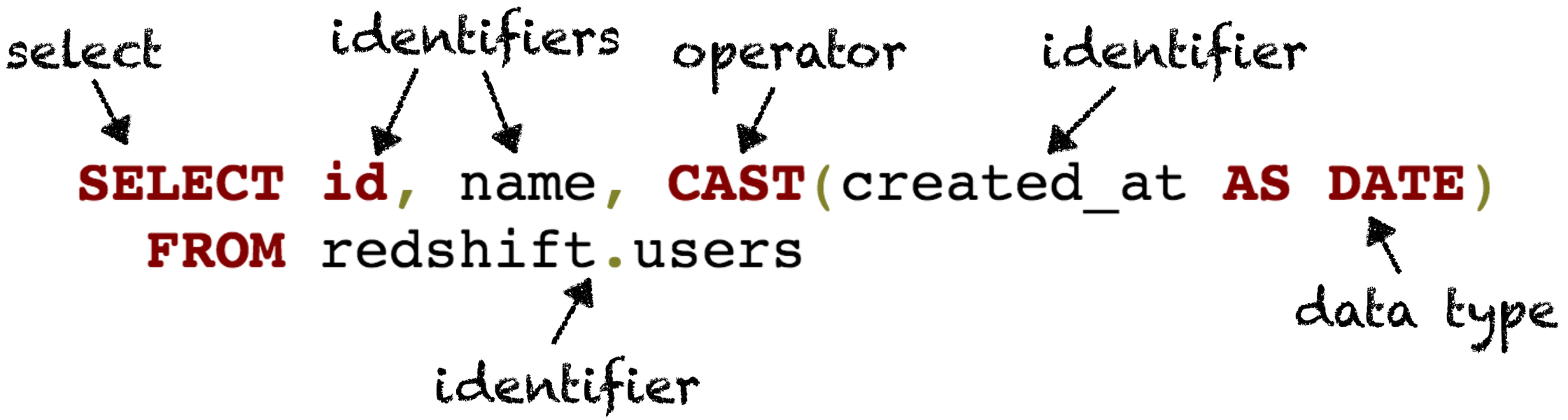
SqlNode

- `SqlNode` represents an element in an abstract syntax tree



SqlNode

- `SqlNode` represents an element in an abstract syntax tree



SqlNode

- **`SqlNode`**'s **`unparse`** method converts a SQL element back into a string

SqlNode

- **`SqlNode`**'s **`unparse`** method converts a SQL element back into a string

```
@Override
public void unparse(SqlWriter writer, int leftPrec, int rightPrec) {
  final SqlOperator operator = getOperator();
  if (leftPrec > operator.getLeftPrec()
      || (operator.getRightPrec() <= rightPrec && (rightPrec != 0))
      || writer.isAlwaysUseParentheses() && isA(SqlKind.EXPRESSION)) {
    final SqlWriter.Frame frame = writer.startList("(", ")");
    operator.unparse(writer, this, 0, 0);
    writer.endList(frame);
  } else {
    operator.unparse(writer, this, leftPrec, rightPrec);
  }
}
```

SqlNode

```
@Override
public void unparse(SqlWriter writer, int leftPrec, int rightPrec) {
  String name = typeName.getSimple();
  SqlTypeName sqlTypeName = SqlTypeName.get(name);

  // we have a built-in data type
  writer.keyword(name);

  if (sqlTypeName.allowsPrec() && (precision >= 0)) {
    final SqlWriter.Frame frame =
      writer.startList(SqlWriter.FrameTypeEnum.FUN_CALL, "(", ")");
    writer.print(precision);
    if (sqlTypeName.allowsScale() && (scale >= 0)) {
      writer.sep(",", true);
      writer.print(scale);
    }
    writer.endList(frame);
  }

  if (charSetName != null) {
    writer.keyword("CHARACTER SET");
    writer.identifier(charSetName);
  }

  if (collectionsTypeName != null) {
    writer.keyword(collectionsTypeName.getSimple());
  }
}
```

SqlNode

- `SqlDialect` indicates the capitalization and quoting rules of specific databases

SqlNode

- `SqlDialect` indicates the capitalization and quoting rules of specific databases

```
SqlDialect dialect = SqlDialect.DatabaseProduct.MYSQL.getDialect();
String sql = sqlNode.toSqlString(dialect).getSql();
```

QUERY OPTIMIZER

next

Query Plans

- Query plans represent the steps necessary to execute a query

Query Plans

- Query plans represent the steps necessary to execute a query

```
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

Query Plans

- Query plans represent the steps necessary to execute a query

```
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

table scan

table scan

Query Plans

- Query plans represent the steps necessary to execute a query

```
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

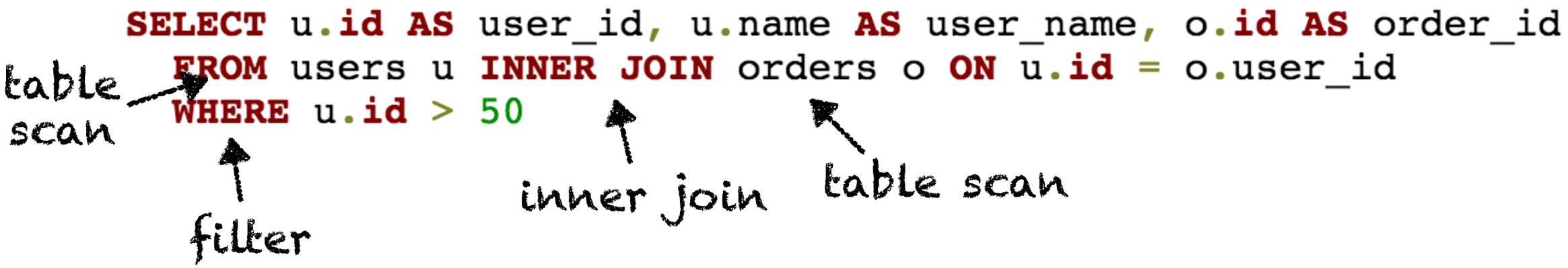
table scan

inner join

table scan

Query Plans

- Query plans represent the steps necessary to execute a query



Query Plans

- Query plans represent the steps necessary to execute a query

Query Plans

- Query plans represent the steps necessary to execute a query


```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalFilter(condition=[>($0, 50)])
    LogicalJoin(condition=[=($0, $6)], joinType=[inner])
      LogicalTableScan(table=[[USERS]])
      LogicalTableScan(table=[[ORDERS]])
```

Query Optimization

- Optimize logical plan
- Goal is typically to try to reduce the amount of data that must be processed early in the plan
- Convert logical plan into a physical plan
- Physical plan is engine specific and represents the physical execution stages

Query Optimization

- Prune unused fields
- Merge projections
- Convert subqueries to joins
- Reorder joins
- Push down projections
- Push down filters

Query Optimization

```sql
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

Query Optimization

```
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalFilter(condition=[>($0, 50)])
    LogicalJoin(condition=[=($0, $6)], joinType=[inner])
      LogicalTableScan(table=[[USERS]])
      LogicalTableScan(table=[[ORDERS]])
```

Query Optimization

```
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalFilter(condition=[>($0, 50)])
    LogicalJoin(condition=[=($0, $6)], joinType=[inner])
      LogicalTableScan(table=[[USERS]])
      LogicalTableScan(table=[[ORDERS]])
```

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalJoin(condition=[=($0, $6)], joinType=[inner])
    LogicalProject(ID=[$0], NAME=[$1])
      LogicalFilter(condition=[>($0, 50)])
        LogicalTableScan(table=[[USERS]])
    LogicalProject(ID=[$0], USER_ID=[$1])
      LogicalTableScan(table=[[ORDERS]])
```

Query Optimization

```
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalFilter(condition=[>($0, 50)])
    LogicalJoin(condition=[=($0, $6)], joinType=[inner])
      LogicalTableScan(table=[[USERS]])
      LogicalTableScan(table=[[ORDERS]])
```

push down project

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalJoin(condition=[=($0, $6)], joinType=[inner])
    LogicalProject(ID=[$0], NAME=[$1])
      LogicalFilter(condition=[>($0, 50)])
        LogicalTableScan(table=[[USERS]])
    LogicalProject(ID=[$0], USER_ID=[$1])
      LogicalTableScan(table=[[ORDERS]])
```

Query Optimization

```
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalFilter(condition=[>($0, 50)])
    LogicalJoin(condition=[=($0, $6)], joinType=[inner])
      LogicalTableScan(table=[[USERS]])
      LogicalTableScan(table=[[ORDERS]])
```

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalJoin(condition=[=($0, $6)], joinType=[inner])
    LogicalProject(ID=[$0], NAME=[$1])
      LogicalFilter(condition=[>($0, 50)])
        LogicalTableScan(table=[[USERS]])
    LogicalProject(ID=[$0], USER_ID=[$1])
      LogicalTableScan(table=[[ORDERS]])
```

push down project

push down filter

Query Optimization

```
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalJoin(condition=[=($0, $6)], joinType=[inner])
    LogicalProject(ID=[$0], NAME=[$1])
      LogicalFilter(condition=[>($0, 50)])
        LogicalTableScan(table=[[USERS]])
    LogicalProject(ID=[$0], USER_ID=[$1])
      LogicalTableScan(table=[[ORDERS]])
```

```
SparkProject(user_id=[$0], user_name=[$1], order_id=[$2])
  SparkJoin(condition=[=($0, $3)], joinType=[inner])
    JdbcToSparkBridge
      JdbcProject(ID=[$0], NAME=[$1])
        JdbcFilter(condition=[>($0, 50)])
          JdbcTableScan(table=[[acme, site, PUBLIC, USERS]])
    JdbcToSparkBridge
      JdbcProject(ID=[$0], USER_ID=[$1])
        JdbcTableScan(table=[[acme, cart, PUBLIC, ORDERS]])
```

Key Concepts

Relational algebra

Row expressions

Traits

Conventions

Rules

Planners

Programs

Key Concepts

Relational algebra	`RelNode`
Row expressions	`RexNode`
Traits	`RelTrait`
Conventions	`Convention`
Rules	`RelOptRule`
Planners	`RelOptPlanner`
Programs	`Program`

Relational Algebra

- `RelNode` represents a relational expression
- Largely equivalent to Spark's `DataFrame` methods
- Logical algebra
- Physical algebra

Relational Algebra

```
TableScan

Project

Filter

Aggregate

Join

Union

Intersect

Sort
```

Relational Algebra

`TableScan`	`SparkTableScan`
`Project`	`SparkProject`
`Filter`	`SparkFilter`
`Aggregate`	`SparkAggregate`
`Join`	`SparkJoin`
`Union`	`SparkUnion`
`Intersect`	`SparkIntersect`
`Sort`	`SparkSort`

Row Expressions

- **`RexNode`** represents a row-level expression
- Largely equivalent to Spark's **`Column`** functions
- Projection fields
- Filter condition
- Join condition
- Sort fields

Row Expressions

Input column ref

Literal

Struct field access

Function call

Window expression

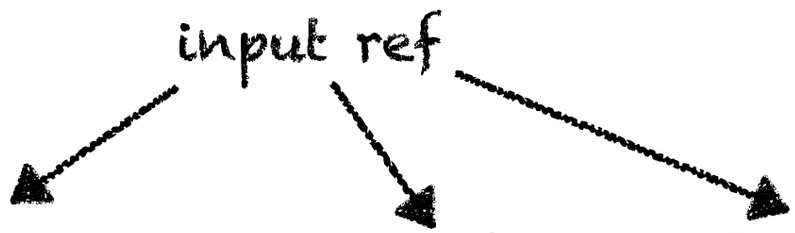
Row Expressions

Input column ref	`RexInputRef`
Literal	`RexLiteral`
Struct field access	`RexFieldAccess`
Function call	`RexCall`
Window expression	`RexOver`

Row Expressions

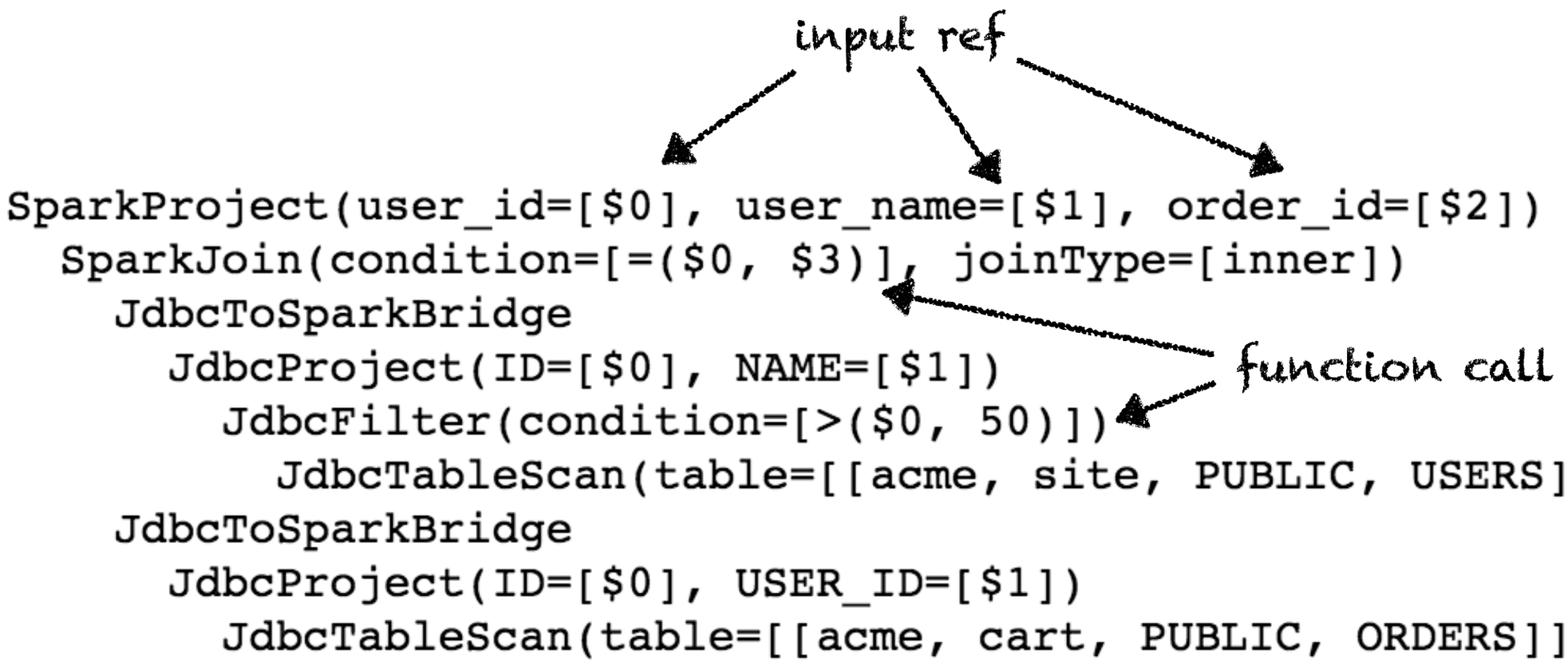
```
SparkProject(user_id=[$0], user_name=[$1], order_id=[$2])
  SparkJoin(condition=[=($0, $3)], joinType=[inner])
    JdbcToSparkBridge
      JdbcProject(ID=[$0], NAME=[$1])
        JdbcFilter(condition=[>($0, 50)])
          JdbcTableScan(table=[[acme, site, PUBLIC, USERS]])
JdbcToSparkBridge
  JdbcProject(ID=[$0], USER_ID=[$1])
    JdbcTableScan(table=[[acme, cart, PUBLIC, ORDERS]])
```

Row Expressions




```
SparkProject(user_id=[$0], user_name=[$1], order_id=[$2])
  SparkJoin(condition=[=($0, $3)], joinType=[inner])
    JdbcToSparkBridge
      JdbcProject(ID=[$0], NAME=[$1])
        JdbcFilter(condition=[>($0, 50)])
          JdbcTableScan(table=[[acme, site, PUBLIC, USERS]])
JdbcToSparkBridge
  JdbcProject(ID=[$0], USER_ID=[$1])
    JdbcTableScan(table=[[acme, cart, PUBLIC, ORDERS]])
```

Row Expressions



Traits

- Defined by the `RelTrait` interface
- Represent a trait of a relational expression that does not alter execution
- Traits are used to validate plan output
- Three primary trait types:
  - `Convention`
  - `RelCollation`
  - `RelDistribution`

Conventions

- `Convention` is a type of `RelTrait`
- A `Convention` is associated with a `RelNode` interface
- `SparkConvention`, `JdbcConvention`, `EnumerableConvention`, etc
- Conventions are used to represent a single data source
- Inputs to a relational expression must be in the same convention

Conventions

```
SparkProject(user_id=[$0], user_name=[$1], order_id=[$2])
  SparkJoin(condition=[=($0, $3)], joinType=[inner])
    JdbcToSparkBridge
      JdbcProject(ID=[$0], NAME=[$1])
        JdbcFilter(condition=[>($0, 50)])
          JdbcTableScan(table=[[acme, site, PUBLIC, USERS]])
JdbcToSparkBridge
  JdbcProject(ID=[$0], USER_ID=[$1])
    JdbcTableScan(table=[[acme, cart, PUBLIC, ORDERS]])
```

Conventions

Spark convention

```
SparkProject(user_id=[$0], user_name=[$1], order_id=[$2])
  SparkJoin(condition=[=($0, $3)], joinType=[inner])
    JdbcToSparkBridge
      JdbcProject(ID=[$0], NAME=[$1])
        JdbcFilter(condition=[>($0, 50)])
          JdbcTableScan(table=[[acme, site, PUBLIC, USERS]])
JdbcToSparkBridge
  JdbcProject(ID=[$0], USER_ID=[$1])
    JdbcTableScan(table=[[acme, cart, PUBLIC, ORDERS]])
```

Conventions

Spark convention

```
SparkProject(user_id=[$0], user_name=[$1], order_id=[$2])
  SparkJoin(condition=[=($0, $3)], joinType=[inner])
    JdbcToSparkBridge
      JdbcProject(ID=[$0], NAME=[$1])
        JdbcFilter(condition=[>($0, 50)])
          JdbcTableScan(table=[[acme, site, PUBLIC, USERS]])
```

JDBC convention

```
JdbcToSparkBridge
  JdbcProject(ID=[$0], USER_ID=[$1])
    JdbcTableScan(table=[[acme, cart, PUBLIC, ORDERS]])
```

Conventions

Spark convention

converter

JDBC convention

```
SparkProject(user_id=[$0], user_name=[$1], order_id=[$2])
  SparkJoin(condition=[=($0, $3)], joinType=[inner])
    JdbcToSparkBridge
      JdbcProject(ID=[$0], NAME=[$1])
        JdbcFilter(condition=[>($0, 50)])
          JdbcTableScan(table=[[acme, site, PUBLIC, USERS]])
    JdbcToSparkBridge
      JdbcProject(ID=[$0], USER_ID=[$1])
        JdbcTableScan(table=[[acme, cart, PUBLIC, ORDERS]])
```

Rules

- Rules are used to modify query plans
- Defined by the **`RelOptRule`** interface
- Two types of rules: converters and transformers
- Converter rules implement **`Converter`** and convert from one convention to another
- Rules are matched to elements of a query plan using pattern matching
- **`onMatch`** is called for matched rules
- Converter rules applied via **`convert`**

Converter Rule

```
public class SparkProjectRule extends ConverterRule {
  public static final SparkProjectRule INSTANCE = new SparkProjectRule();

  protected SparkProjectRule() {
    super(LogicalProject.class, Convention.NONE, SparkConvention.INSTANCE, "SparkProjectRule");
  }

  @Override
  public RelNode convert(RelNode rel) {
    final LogicalProject project = (LogicalProject) rel;
    return new SparkProject(
      rel.getCluster(),
      rel.getTraitSet().replace(SparkConvention.INSTANCE),
      convert(
        project.getInput(),
        project.getInput().getTraitSet().replace(SparkConvention.INSTANCE)),
      project.getProjects(),
      project.getRowType());
  }
}
```

Converter Rule

```
public class SparkProjectRule extends ConverterRule {
  public static final SparkProjectRule INSTANCE = new SparkProjectRule();

  protected SparkProjectRule() {
    super(LogicalProject.class, Convention.NONE, SparkConvention.INSTANCE, "SparkProjectRule");
  }

  @Override
  public RelNode convert(RelNode rel) {
    final LogicalProject project = (LogicalProject) rel;
    return new SparkProject(
      rel.getCluster(),
      rel.getTraitSet().replace(SparkConvention.INSTANCE),
      convert(
        project.getInput(),
        project.getInput().getTraitSet().replace(SparkConvention.INSTANCE)),
      project.getProjects(),
      project.getRowType());
  }
}
```

expression type

Converter Rule

```
public class SparkProjectRule extends ConverterRule {
  public static final SparkProjectRule INSTANCE = new SparkProjectRule();

  protected SparkProjectRule() {
    super(LogicalProject.class, Convention.NONE, SparkConvention.INSTANCE, "SparkProjectRule");
  }

  @Override
  public RelNode convert(RelNode rel) {
    final LogicalProject project = (LogicalProject) rel;
    return new SparkProject(
      rel.getCluster(),
      rel.getTraitSet().replace(SparkConvention.INSTANCE),
      convert(
        project.getInput(),
        project.getInput().getTraitSet().replace(SparkConvention.INSTANCE)),
      project.getProjects(),
      project.getRowType());
  }
}
```

input convention

expression type

Converter Rule

```
public class SparkProjectRule extends ConverterRule {
  public static final SparkProjectRule INSTANCE = new SparkProjectRule();

  protected SparkProjectRule() {
    super(LogicalProject.class, Convention.NONE, SparkConvention.INSTANCE, "SparkProjectRule");
  }

  @Override
  public RelNode convert(RelNode rel) {
    final LogicalProject project = (LogicalProject) rel;
    return new SparkProject(
      rel.getCluster(),
      rel.getTraitSet().replace(SparkConvention.INSTANCE),
      convert(
        project.getInput(),
        project.getInput().getTraitSet().replace(SparkConvention.INSTANCE)),
      project.getProjects(),
      project.getRowType());
  }
}
```

input convention

expression type

converted convention

Converter Rule

```
public class SparkProjectRule extends ConverterRule {
  public static final SparkProjectRule INSTANCE = new SparkProjectRule();

  protected SparkProjectRule() {
    super(LogicalProject.class, Convention.NONE, SparkConvention.INSTANCE, "SparkProjectRule");
  }

  @Override
  public RelNode convert(RelNode rel) {
    final LogicalProject project = (LogicalProject) rel;
    return new SparkProject(
      rel.getCluster(),
      rel.getTraitSet().replace(SparkConvention.INSTANCE),
      convert(
        project.getInput(),
        project.getInput().getTraitSet().replace(SparkConvention.INSTANCE)),
      project.getProjects(),
      project.getRowType());
  }
}
```

input convention

expression type

converted convention

converter function

Pattern Matching

```
public class ApolloProjectJoinTransposeRule extends RelOptRule {
  public ApolloProjectJoinTransposeRule() {
    super(
      operand(LogicalProject.class,
        operand(Join.class, any())));
  }

  @Override
  public void onMatch(RelOptRuleCall call) {
    Project newProject = null;
    // Create the new Project
    call.transformTo(newProject);
  }
}
```

Pattern Matching

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalFilter(condition=[>($0, 50)])
    LogicalJoin(condition=[=($0, $6)], joinType=[inner])
      LogicalTableScan(table=[[USERS]])
      LogicalTableScan(table=[[ORDERS]])
```

Pattern Matching

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalFilter(condition=[>($0, 50)])
    LogicalJoin(condition=[=($0, $6)], joinType=[inner])
      LogicalTableScan(table=[[USERS]])
      LogicalTableScan(table=[[ORDERS]])
```

no match :-(

Pattern Matching

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalFilter(condition=[>($0, 50)])
    LogicalJoin(condition=[=($0, $6)], joinType=[inner])
      LogicalTableScan(table=[[USERS]])
      LogicalTableScan(table=[[ORDERS]])
```

no match :-(

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalJoin(condition=[=($0, $6)], joinType=[inner])
    LogicalProject(ID=[$0], NAME=[$1])
      LogicalFilter(condition=[>($0, 50)])
        LogicalTableScan(table=[[USERS]])
    LogicalProject(ID=[$0], USER_ID=[$1])
      LogicalTableScan(table=[[ORDERS]])
```

Pattern Matching

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalFilter(condition=[>($0, 50)])
    LogicalJoin(condition=[=($0, $6)], joinType=[inner])
      LogicalTableScan(table=[[USERS]])
      LogicalTableScan(table=[[ORDERS]])
```

no match :-(

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalJoin(condition=[=($0, $6)], joinType=[inner])
    LogicalProject(ID=[$0], NAME=[$1])
      LogicalFilter(condition=[>($0, 50)])
        LogicalTableScan(table=[[USERS]])
    LogicalProject(ID=[$0], USER_ID=[$1])
      LogicalTableScan(table=[[ORDERS]])
```

match!

Planners

- Planners implement the `RelOptPlanner` interface
- Two types of planners:
  - `HepPlanner`
  - `VolcanoPlanner`

Heuristic Optimization

- `HepPlanner` is a heuristic optimizer similar to Spark's optimizer
- Applies all matching rules until none can be applied
- Heuristic optimization is faster than cost-based optimization
- Risk of infinite recursion if rules make opposing changes to the plan

Cost-based Optimization

- `VolcanoPlanner` is a cost-based optimizer
- Applies matching rules iteratively, selecting the plan with the cheapest cost on each iteration
- Costs are provided by relational expressions
- Not all possible plans can be computed
- Stops optimization when the cost does not significantly improve through a determinable number of iterations

Cost-based Optimization

- Cost is provided by each **`RelNode`**
- Cost is represented by **`RelOptCost`**
- Cost typically includes row count, I/O, and CPU cost
- Cost estimates are relative
- Statistics are used to improve accuracy of cost estimations
- Calcite provides utilities for computing various resource-related statistics for use in cost estimations

Cost-based Optimization

```
@Override
public RelOptCost computeSelfCost(RelOptPlanner planner, RelMetadataQuery mq) {
  double rows = RelMdUtil.getJoinRowCount(mq, this, getCondition());
  int leftColumns = getLeft().getRowType().getFieldCount();
  double leftRows = mq.getRowCount(getLeft());
  int rightColumns = getRight().getRowType().getFieldCount();
  double rightRows = mq.getRowCount(getRight());
  double io = Util.nLogN(leftRows) * leftColumns + Util.nLogN(rightRows) * rightColumns;
  return planner.getCostFactory().makeCost(rows, 0, io);
}
```

Cost-based Optimization

```java
@Override
public RelOptCost computeSelfCost(RelOptPlanner planner, RelMetadataQuery mq) {
  double rows = RelMdUtil.getJoinRowCount(mq, this, getCondition());
  int leftColumns = getLeft().getRowType().getFieldCount();
  double leftRows = mq.getRowCount(getLeft());
  int rightColumns = getRight().getRowType().getFieldCount();
  double rightRows = mq.getRowCount(getRight());

  // We want to make the cost of a JDBC join relative to the optimizations on the join columns.
  // If the left hand side of the join is a key column (primary key, index, distribution key, etc)
  // then the cost of the LHS in terms of I/O is nil; the same is true of the right hand side.
  // I/O cost is only added for joins on non-key columns.
  JoinInfo joinInfo = analyzeCondition();

  // Iterate through each of the left hand side join columns and determine whether the column
  // is optimized for joins.
  float leftJoinKeys = 0;
  for (int leftKey : joinInfo.leftKeys) {
    if (isJoinOnKey(left, leftKey, planner, mq)) {
      leftJoinKeys++;
    }
  }

  // Iterate through each of the right hand side join columns and determine whether the column
  // is optimized for joins.
  float rightJoinKeys = 0;
  for (int rightKey : joinInfo.rightKeys) {
    if (isJoinOnKey(right, rightKey, planner, mq)) {
      rightJoinKeys++;
    }
  }

  // Compute the cost of each side of the join based on the total row count times the percentage of
  // columns that are *not* optimized for the join.
  double leftJoinCost = leftRows * leftColumns * ((joinInfo.leftKeys.size() - leftJoinKeys) / (double) joinInfo.leftKeys.siz
  double rightJoinCost = rightRows * rightColumns * ((joinInfo.rightKeys.size() - rightJoinKeys) / (double) joinInfo.rightKe

  // The I/O cost is the sum of the cost of each side.
  return (ApolloCost)planner.getCostFactory().makeCost(rows, 0, (leftJoinCost + rightJoinCost) * 2);
}
```

Cost-based Optimization

```
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

```
SparkProject(user_id=[$0], user_name=[$1], order_id=[$2])
  SparkJoin(condition=[=($0, $3)], joinType=[inner])
    JdbcToSparkBridge
      JdbcProject(ID=[$0], NAME=[$1])
        JdbcFilter(condition=[>($0, 50)])
          JdbcTableScan(table=[[acme, site, PUBLIC, USERS]])
    JdbcToSparkBridge
      JdbcProject(ID=[$0], USER_ID=[$1])
        JdbcTableScan(table=[[acme, cart, PUBLIC, ORDERS]])
```

Cost-based Optimization

```
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

```
SparkProject(user_id=[$0], user_name=[$1], order_id=[$2])
  SparkJoin(condition=[=($0, $3)], joinType=[inner])
    JdbcToSparkBridge
      JdbcProject(ID=[$0], NAME=[$1])
        JdbcFilter(condition=[>($0, 50)])
          JdbcTableScan(table=[[acme, site, PUBLIC, USERS]])
    JdbcToSparkBridge
      JdbcProject(ID=[$0], USER_ID=[$1])
        JdbcTableScan(table=[[acme, cart, PUBLIC, ORDERS]])
```

```
SparkProject(user_id=[$0], user_name=[$1], order_id=[$2]): rowcount = 750.0, cumulative cost = {2122.
  SparkJoin(condition=[=($0, $3)], joinType=[inner]): rowcount = 750.0, cumulative cost = {1222.0 rows
    JdbcToSparkBridge: rowcount = 50.0, cumulative cost = {161.0 rows, 81.0 cpu, 80.0 io}, id = 21748
      JdbcProject(ID=[$0], NAME=[$1]): rowcount = 50.0, cumulative cost = {160.0 rows, 80.0 cpu, 80.0
        JdbcFilter(condition=[>($0, 50)]): rowcount = 50.0, cumulative cost = {120.0 rows, 80.0 cpu,
          JdbcTableScan(table=[[acme, site, PUBLIC, USERS]]): rowcount = 100.0, cumulative cost = {80
    JdbcToSparkBridge: rowcount = 100.0, cumulative cost = {161.0 rows, 1.0 cpu, 80.0 io}, id = 21750
      JdbcProject(ID=[$0], USER_ID=[$1]): rowcount = 100.0, cumulative cost = {160.0 rows, 0.0 cpu, 8
        JdbcTableScan(table=[[acme, cart, PUBLIC, ORDERS]]): rowcount = 100.0, cumulative cost = {80.
```

Cost-based Optimization

```
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

```
SparkProject(user_id=[$0], user_name=[$1], order_id=[$2])
  SparkJoin(condition=[=($0, $3)], joinType=[inner])
    JdbcToSparkBridge
      JdbcProject(ID=[$0], NAME=[$1])
        JdbcFilter(condition=[>($0, 50)])
          JdbcTableScan(table=[[acme, site, PUBLIC, USERS]])
    JdbcToSparkBridge
      JdbcProject(ID=[$0], USER_ID=[$1])
        JdbcTableScan(table=[[acme, cart, PUBLIC, ORDERS]])
```

```
SparkProject(user_id=[$0], user_name=[$1], order_id=[$2]): rowcount = 750.0, cumulative cost = {2122.
  SparkJoin(condition=[=($0, $3)], joinType=[inner]): rowcount = 750.0, cumulative cost = {1222.0 rows
    JdbcToSparkBridge: rowcount = 50.0, cumulative cost = {161.0 rows, 81.0 cpu, 80.0 io}, id = 21748
      JdbcProject(ID=[$0], NAME=[$1]): rowcount = 50.0, cumulative cost = {160.0 rows, 80.0 cpu, 80.0
        JdbcFilter(condition=[>($0, 50)]): rowcount = 50.0, cumulative cost = {120.0 rows, 80.0 cpu,
          JdbcTableScan(table=[[acme, site, PUBLIC, USERS]]): rowcount = 100.0, cumulative cost = {80
    JdbcToSparkBridge: rowcount = 100.0, cumulative cost = {161.0 rows, 1.0 cpu, 80.0 io}, id = 21750
      JdbcProject(ID=[$0], USER_ID=[$1]): rowcount = 100.0, cumulative cost = {160.0 rows, 0.0 cpu, 8
        JdbcTableScan(table=[[acme, cart, PUBLIC, ORDERS]]): rowcount = 100.0, cumulative cost = {80.
```

PUTTING IT ALL TOGETHER

Putting it all together

```
String sql = "SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id " +
  "FROM users u INNER JOIN orders o ON u.id = o.user_id WHERE u.id > 50";
```

Putting it all together

```
String sql = "SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id " +
  "FROM users u INNER JOIN orders o ON u.id = o.user_id WHERE u.id > 50";
```

```
// Parse the query
SqlParser parser = SqlParser.create(sql, parserConfig);
SqlNode sqlNode = parser.parseStmt();
```

Putting it all together

```
SqlNode sqlNode = parser.parseStmt();
```

Putting it all together

```
SqlNode sqlNode = parser.parseStmt();
```

```
// Validate the query
CalciteCatalogReader catalogReader = createCatalogReader();
SqlValidator validator = SqlValidatorUtil.newValidator(
  SqlStdOperatorTable.instance(), catalogReader, typeFactory, SqlConformance.DEFAULT);
SqlNode validatedSqlNode = validator.validate(sqlNode);
```

Putting it all together

```
SqlNode validatedSqlNode = validator.validate(sqlNode);
```

Putting it all together

```
SqlNode validatedSqlNode = validator.validate(sqlNode);
```

```
// Convert SqlNode to RelNode
RexBuilder rexBuilder = createRexBuilder();
RelOptCluster cluster = RelOptCluster.create(planner, rexBuilder);
SqlToRelConverter sqlToRelConverter =
  new SqlToRelConverter(new ViewExpanderImpl(), validator, createCatalogReader(), cluster, convertletTable);
RelRoot root = sqlToRelConverter.convertQuery(validatedSqlNode, false, true);
```

Putting it all together

```
RelRoot root = sqlToRelConverter.convertQuery(validatedSqlNode, false, true);
```

Putting it all together

```
RelRoot root = sqlToRelConverter.convertQuery(validatedSqlNode, false, true);

        // Optimize the plan.
        RelOptPlanner planner = new VolcanoPlanner();

        // Create a set of rules to apply.
        Program program = Programs.ofRules(
          FilterProjectTransposeRule.INSTANCE,
          ProjectMergeRule.INSTANCE,
          FilterMergeRule.INSTANCE,
          LoptOptimizeJoinRule.INSTANCE
        );

        // Create a desired output trait set.
        RelTraitSet traitSet = planner.emptyTraitSet()
          .replace(SparkConvention.INSTANCE);

        // Execute the program.
        RelNode optimized = program.run(planner, root.rel, traitSet);
```

Putting it all together

```
String sql = "SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id " +
  "FROM users u INNER JOIN orders o ON u.id = o.user_id WHERE u.id > 50";

// Parse the query
SqlParser parser = SqlParser.create(sql, parserConfig);
SqlNode sqlNode = parser.parseStmt();

// Validate the query
CalciteCatalogReader catalogReader = createCatalogReader();
SqlValidator validator = SqlValidatorUtil.newValidator(
  SqlStdOperatorTable.instance(), catalogReader, typeFactory, SqlConformance.DEFAULT);
SqlNode validatedSqlNode = validator.validate(sqlNode);

// Convert SqlNode to RelNode
RexBuilder rexBuilder = createRexBuilder();
RelOptCluster cluster = RelOptCluster.create(planner, rexBuilder);
SqlToRelConverter sqlToRelConverter =
  new SqlToRelConverter(new ViewExpanderImpl(), validator, createCatalogReader(), cluster, convertletTable);
RelRoot root = sqlToRelConverter.convertQuery(validatedSqlNode, false, true);

// Optimize the plan.
RelOptPlanner planner = new VolcanoPlanner();

// Create a set of rules to apply.
Program program = Programs.ofRules(
  FilterProjectTransposeRule.INSTANCE,
  ProjectMergeRule.INSTANCE,
  FilterMergeRule.INSTANCE,
  LoptOptimizeJoinRule.INSTANCE
);

// Create a desired output trait set.
RelTraitSet traitSet = planner.emptyTraitSet()
  .replace(SparkConvention.INSTANCE);

// Execute the program.
RelNode optimized = program.run(planner, root.rel, traitSet);
```

Putting it all together

```
RelNode optimized = program.run(planner, root.rel, traitSet);
```