CS 744: BIG DATA SYSTEMS

Shivaram Venkataraman
Fall 2018
- Assignment 1 grades up, Assignment 2 in progress
- Midterm review session on Nov 2 at 5pm
- Course Project Proposal (5%)
SQL: STRUCTURED QUERY LANGUAGE
SQL IN BIG DATA SYSTEMS

- **Scale**: How do we handle large datasets, clusters?

- **Wide-area**: How do we handle queries across datacenters?

- **Hardware**: Making efficient use of hardware?
SPARK SQL: ARCHITECTURE
lines = sc.textFile(“users”)  
csv = lines.map(x =>  
    x.split(‘,’))  
young = csv.filter(x =>  
    x(1) < 21)  
println(young.count())

ctx = new HiveContext ()  
users = ctx.table(“users”)  
young = users.where(  
    users(“age”) < 21)  
println(young.count())
OPERATORS ➔ EXPRESSIONS

Projection (select), Filter, Join, Aggregations take in Expressions

```
employees.join(dept,
    employees ("deptId") === dept ("id ")
)
```

Build up Abstract Syntax Tree (AST)
OTHER FEATURES

1. Debugging: Eager analysis of logical plans
2. Interoperability: Convert RDD to Dataframes
3. Caching: Columnar caching with compression
4. UDFs: Python or Scala functions
Goal: Extensibility to add new optimization rules
CATALYST DESIGN

Library for representing **trees** and **rules** to manipulate them

Pattern match → replace sub-trees

Only applied in sub-trees that match

Run in batches till **fixed point**

tree. transform { 
    case Add(Literal(c1),Literal(c2)) => Literal(c1+c2) 
    case Add(left , Literal(0)) => left 
    case Add(Literal(0), right) => right 
}
LOGICAL, PHYSICAL PLANS

1. **Analyzer** Lookup relations, map named attributes, propagate types
2. **Logical Optimization**
   - Constant folding
   - Predicate push-down
   - Project pruning …
3. **Physical Planning**
   - Select between plans using cost (join algorithm)
   - Pipeline multiple projection, filter into map
CODE GENERATION

CPU bound when data is in-memory
Branches, virtual function calls etc.

```python
def compile(node: Node): AST = node match {
    case Literal(value) => q"${value}"
    case Attribute (name) => q"${row.get($name)}"
    case Add(left, right) =>
        q"${compile(left)} + ${compile(right)}"
}
```

- **Literal(1)** becomes `1`
- **Attribute(“x”)** becomes `row.get(“x”)`
- Directly access Java field `row.x`
EXTENSIONS

Data sources
- Define a BaseRelation that contains schema
- TableScan returns RDD[Row]
- Pruning / Filtering optimizations

User-Defined Types (UDTs)
- Support advanced analytics with e.g. Vector
- Users provide mapping from UDT to Catalyst Row
SCHEMA INFERENCE

Common data formats: JSON, CSV, semi-structured data

JSON schema inference
- Find most specific SparkSQL type that matches instances
  e.g. if tweet.loc.latitude are all 32-bit then it is a INT
- Fall back to STRING if unknown
- Implemented using a reduce over trees of types
SUMMARY, TAKEAWAYS

Relational API
- Enables rich space of optimizations
- Easy to use, integration with Scala, Python

Catalyst Optimizer
- Extensible, rule-based optimizer
- Code generation for high-performance

Evolution of Spark API
QUESTIONS / DISCUSSION?