APACHE calcite™

Tutorial @BOSS'21 Copenhagen

Stamatis Zampetakis, Julian Hyde • August 16, 2021

BOSS

BIG DATA OPEN SOURCE SYSTEMS

APACHE calcite™

Tutorial @BOSS'21 Copenhagen

Stamatis Zampetakis, Julian Hyde • August 16, 2021

```
# Follow these steps to set up your environment
# (The first time, it may take ~3 minutes to download dependencies.)

git clone https://github.com/zabetak/calcite-tutorial.git
java -version                   # need Java 8 or higher
cd calcite-tutorial
./mvnw package -DskipTests
```

Setup Environment

Requirements

1. Git
2. JDK version ≥ 1.8

Steps

1. Clone GitHub repository
2. Load to IDE (preferred IntelliJ)
   a. Click Open
   b. Navigate to calcite-tutorial
   c. Select `pom.xml` file
   d. Choose "Open as Project"
3. Compile the project

```
git clone https://github.com/zabetak/calcite-tutorial.git
```

```
java -version
cd calcite-tutorial
./mvnw package -DskipTests
```

Draft outline

- 1. Calcite Introduction (slides) ~ 10’
- 2. Demonstration via Sqlline on CSV ~ 5’ (sql parser, explain plan, !tables)
- 3. Setup coding environment (1 slide) ~ 5’
- 4. Lucene intro & indexing (slides/code?) ~ 5’
- 5. Coding module I (type as you go) ~ 25’ (schema, typefactory, sql parser, validator, planner, rules, execution via Enumerable)
- 6. Exercise: Write more queries (with aggregation)/solve CannotPlanException ~5’
- 7. Exercise: Add more optimization rules (show row count stats) / Change option
- 8. Exercise: Use RelBuilder to construct queries ~ 5’-10’ (avoid joins, include having query: scan.filter.aggregate.filter)
- 9. Hybrid planning / multiple conventions (slides) ~
- 10. Volcano planning / queues / sets / (slides) ~
- 11. Coding module II (3 RelNodes + 3 Rules + 1 RexVisitor) ~ 30’ (slides + code)
- ~~12. Exercise: ??~~
- ~~13. Coding module III (two conventions + push filter rule + RexToLuceneTranslator) ~ 10’~~
- ~~14. Exercise: Use RexVisitor instead of hardcoded if clauses ~ 5’~~
- ~~15. Exercise Extensions to push filter rule + translator ~ 10’~~
- 16. Dialect (slides) ~
- ~~17. Custom UDF / Operator table () ~~~
- ~~18. Coding module XI (splitting rule to Enumerable/ Lucene)~~
- 19. Calcite with Spatial data (slides) ~

About us

Julian Hyde @julianhyde
Senior Staff Engineer @ Google / Looker
Creator of Apache Calcite
PMC member of Apache Arrow, Drill, Eagle, Incubator and Kylin

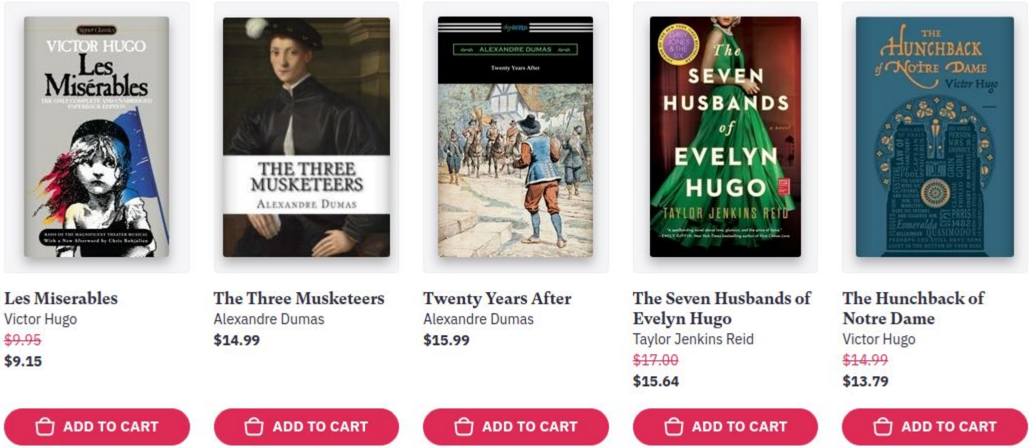
Stamatis Zampetakis @szampetak
Senior Software Engineer @ Cloudera, Hive query optimizer team
PMC member of Apache Calcite; Hive committer
PhD in Data Management, INRIA & Paris-Sud University

Outline

1. Introduction
2. CSV Adapter Demo
3. Coding module I: Main components
4. Coding module I Exercises (Homework)
5. Hybrid planning
6. Coding module II: Custom operators/rules (Homework)
7. Volcano Planner internals (optional)
8. Dialects (optional)
9. Materialized views (optional)
10. Working with spatial data (optional)
11. Research using Apache Calcite

1. Calcite introduction

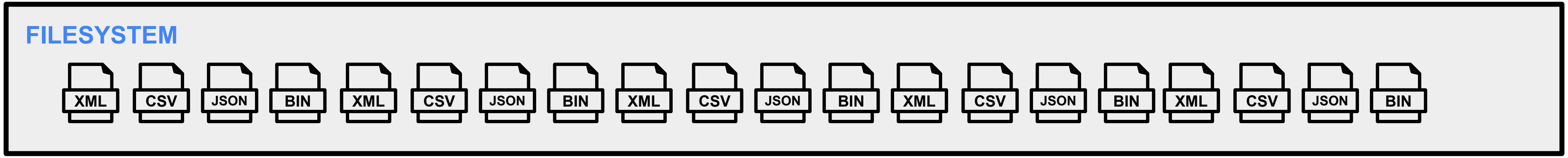
Motivation: Data views



1. Retrieve books and authors
2. Display image, title, price of the book along with firstname & lastname of the author
3. Sort the books based on their id (price or something else)
4. Show results in groups of five

What, where, how data are stored?

What, where, how data are stored?



What, where, how data are stored?

AUTHOR	
id	int
fname	string
lname	string
birth	date

author

0..1

BOOK	
id	int
title	string
price	decimal
year	int

ORACLE DATABASE

cassandra

elasticsearch

HIVE

APACHE DRILL

APACHE Spark

neo4j

hazelcast

Google BigQuery

APACHE HBASE

APACHE LUCENE

PostgreSQL

redis

Solr

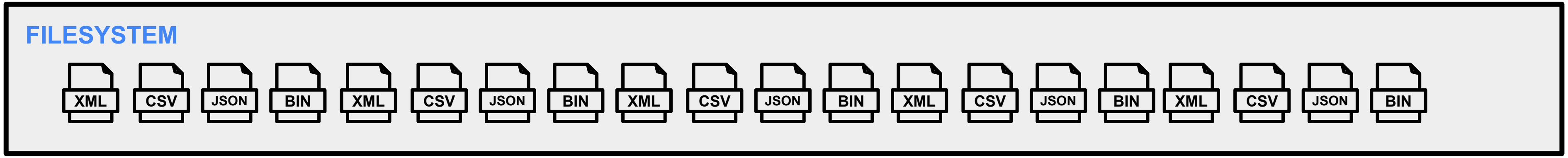
mongoDB

FILESYSTEM

XML CSV JSON BIN XML CSV JSON BIN XML CSV JSON BIN XML CSV JSON BIN XML CSV JSON BIN

What, where, how data are stored?

360+ DBMS



Apache Lucene

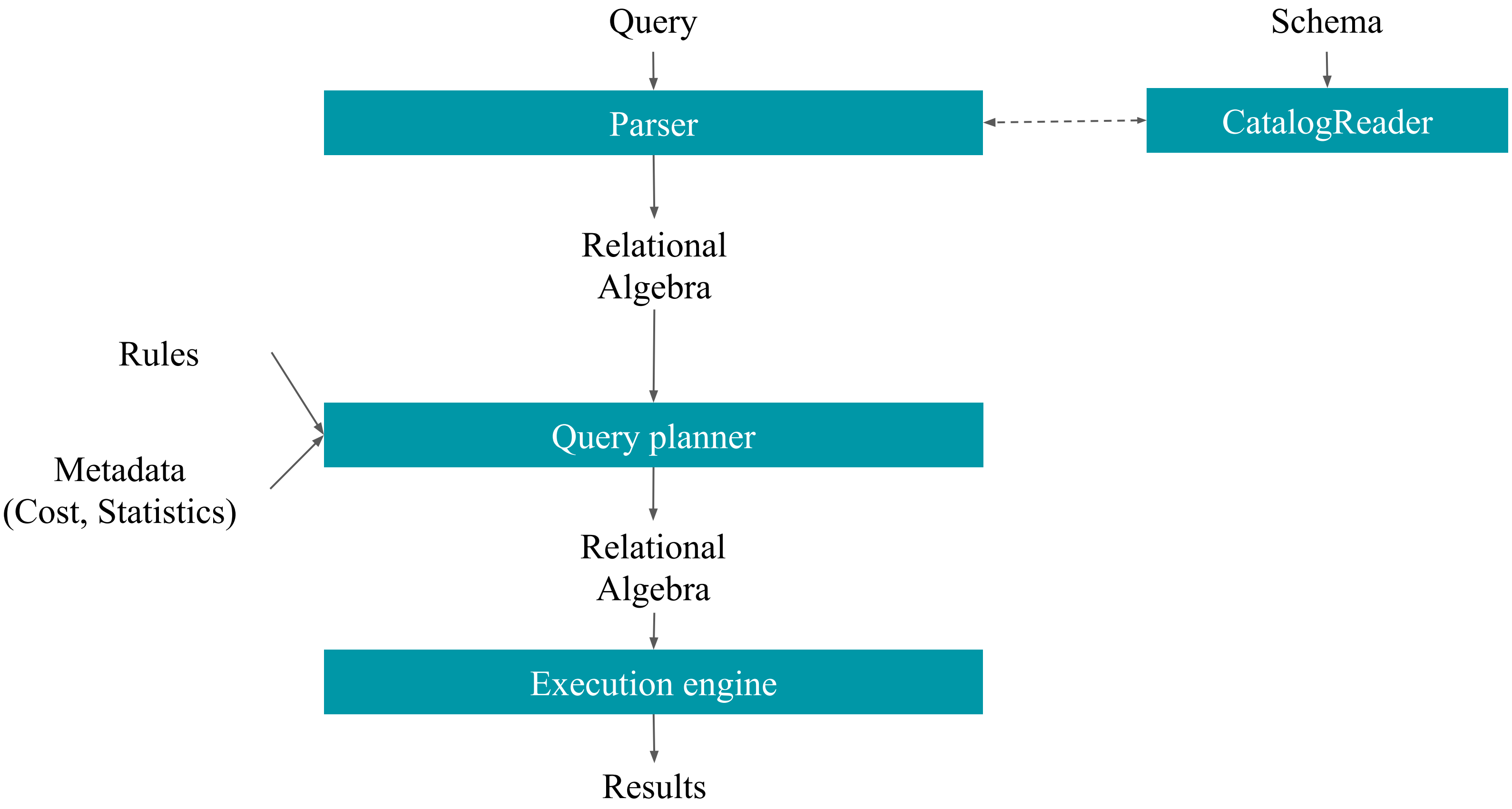
- ★ Open-source search engine
- ★ Java library
- ★ Powerful indexing & search features
- ★ Spell checking, hit highlighting
- ★ Advanced analysis/tokenization capabilities
- ★ ACID transactions
- ★ Ultra compact memory/disk format

How to query the data?

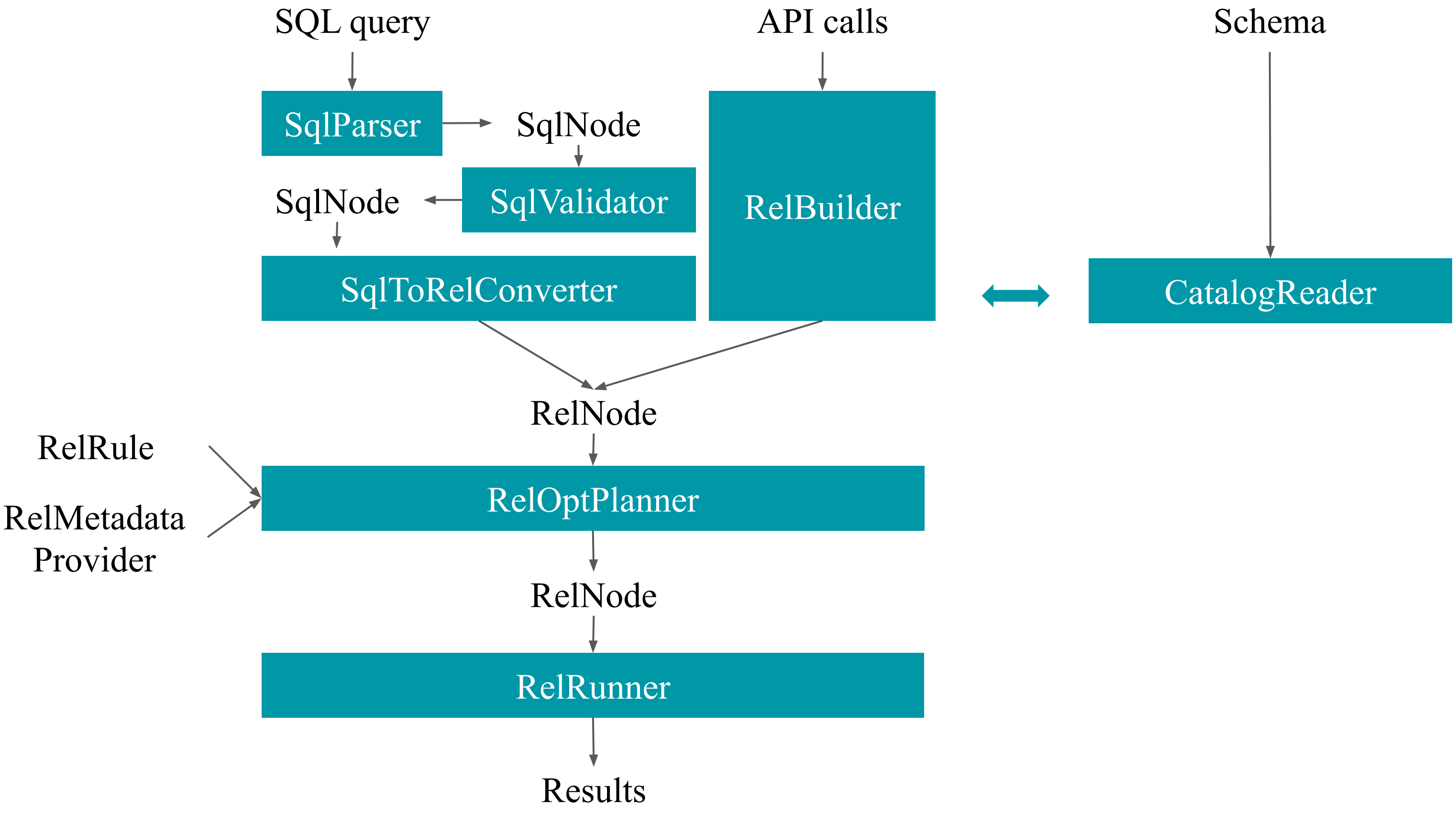
1. Retrieve books and authors
2. Display image, title, price of the book along with firstname & lastname of the author
3. Sort the books based on their id (price or something else)
4. Show results in groups of five

```
SELECT b.id, b.title, b.year, a.fname, a.lname
FROM Book b
LEFT OUTER JOIN Author a ON b.author=a.id
ORDER BY b.id
LIMIT 5
```

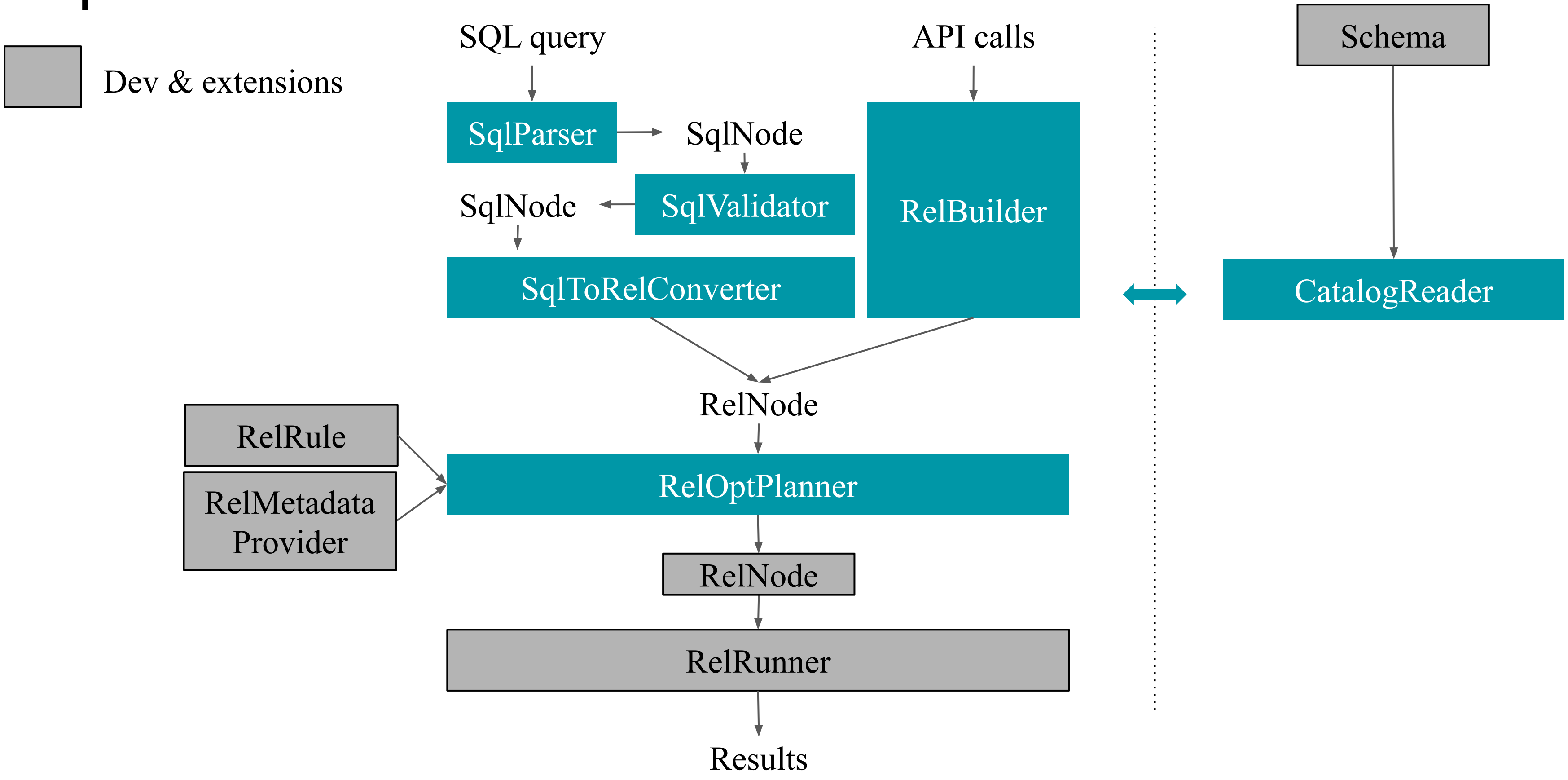
Query processor architecture



Apache Calcite



Apache Calcite



2. CSV Adapter Demo

Adapter

Implement `SchemaFactory` interface

Connect to a data source using parameters

Extract schema - return a list of tables

Push down processing to the data source:

- A set of planner rules
- Calling convention (optional)
- Query model & query generator (optional)

```
{
  "schemas": [
    {
      "name": "HR",
      "type": "custom",
      "factory":
  "org.apache.calcite.adapter.file.FileSchemaFactory",
      "operand": {
        "directory": "hr-csv"
      }
    }
  ]
]
```

```
$ ls -l hr-csv
-rw-r--r--  1 jhyde staff  62 Mar 29 12:57 DEPTS.csv
-rw-r--r--  1 jhyde staff 262 Mar 29 12:57 EMPS.csv.gz

$ ./sqlline -u jdbc:calcite:model=hr.json -n scott -p
tiger
sqlline> select count(*) as c from emp;
'C'
'5'
1 row selected (0.135 seconds)
```

Adapter

Implement `SchemaFactory` interface

Connect to a data source using parameters

Extract schema - return a list of tables

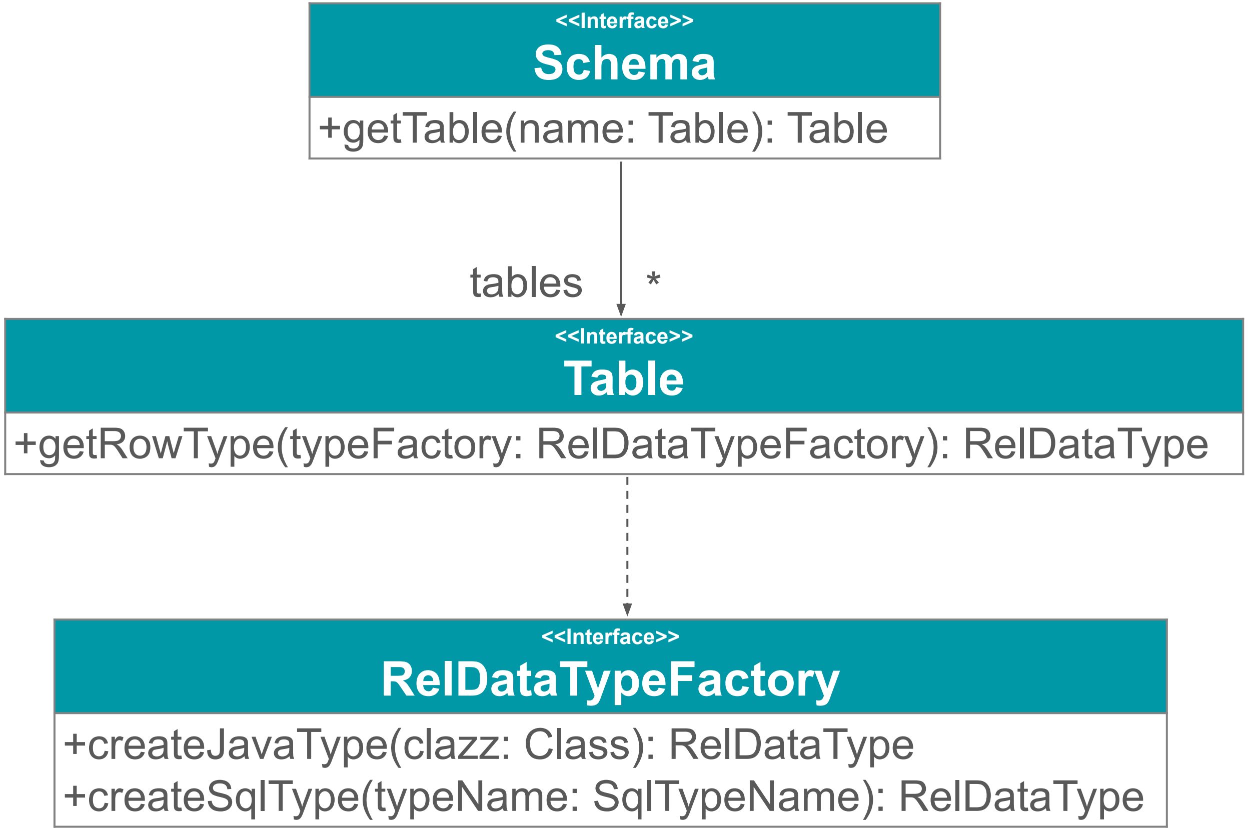
Push down processing to the data source:

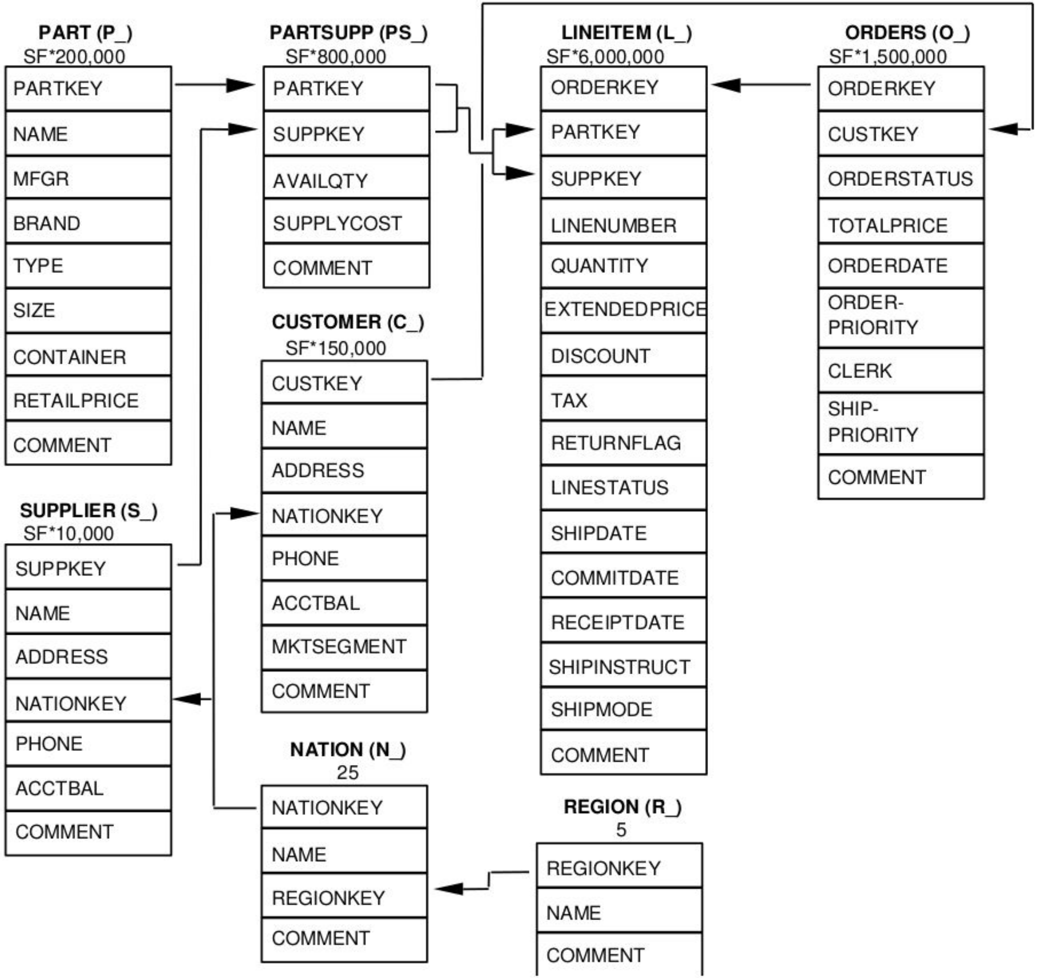
- A set of planner rules
- Calling convention (optional)
- Query model & query generator (optional)

```
{
  "schemas": [
    {
      "name": "BOOKSTORE",
      "type": "custom",
      "factory":
  "org.apache.calcite.adapter.file.FileSchemaFactory",
      "operand": {
        "directory": "bookstore"
      }
    }
  ]
]
```

3. Coding module I: Main components

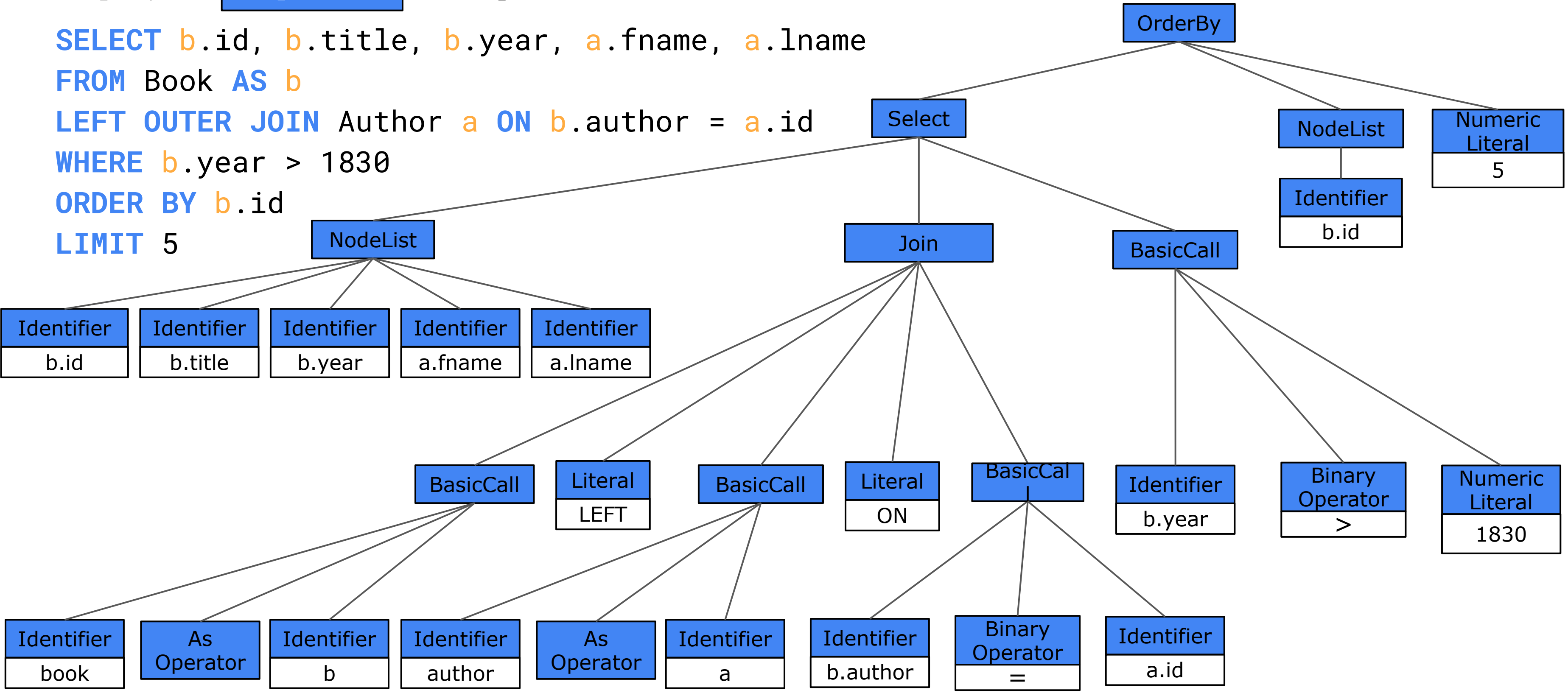
Setup schema & type factory



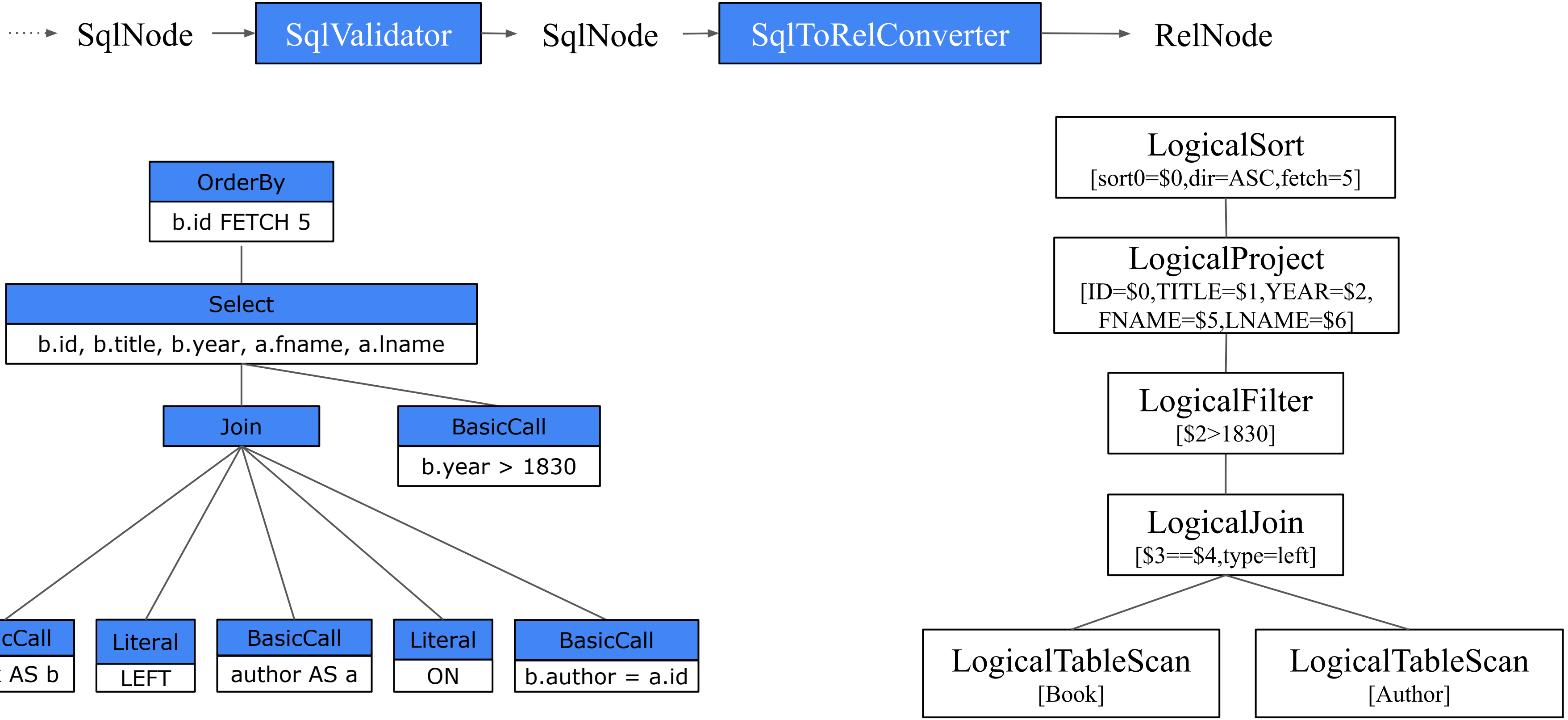

PART (P_)
SF*200,000
PARTKEY
NAME
MFGR
BRAND
TYPE
SIZE
CONTAINER
RETAILPRICE
COMMENT
PARTSUPP (PS_)
SF*800,000
PARTKEY
SUPPKEY
AVAILQTY
SUPPLYCOST
COMMENT
LINEITEM (L_)
SF*6,000,000
ORDERKEY
PARTKEY
SUPPKEY
LINENUMBER
QUANTITY
EXTENDEDPRICE
DISCOUNT
TAX
RETURNFLAG
LINESTATUS
SHIPDATE
COMMITDATE
RECEIPTDATE
SHIPINSTRUCT
SHIPMODE
COMMENT
ORDERS (O_)
SF*1,500,000
ORDERKEY
CUSTKEY
ORDERSTATUS
TOTALPRICE
ORDERDATE
ORDER-PRIORITY
CLERK
SHIP-PRIORITY
COMMENT
CUSTOMER (C_)
SF*150,000
CUSTKEY
NAME
ADDRESS
NATIONKEY
PHONE
ACCTBAL
MKTSEGMENT
COMMENT
SUPPLIER (S_)
SF*10,000
SUPPKEY
NAME
ADDRESS
NATIONKEY
PHONE
ACCTBAL
COMMENT
NATION (N_)
25
NATIONKEY
NAME
REGIONKEY
COMMENT
REGION (R_)
5
REGIONKEY
NAME
COMMENT

Query to Abstract Syntax Tree (AST)

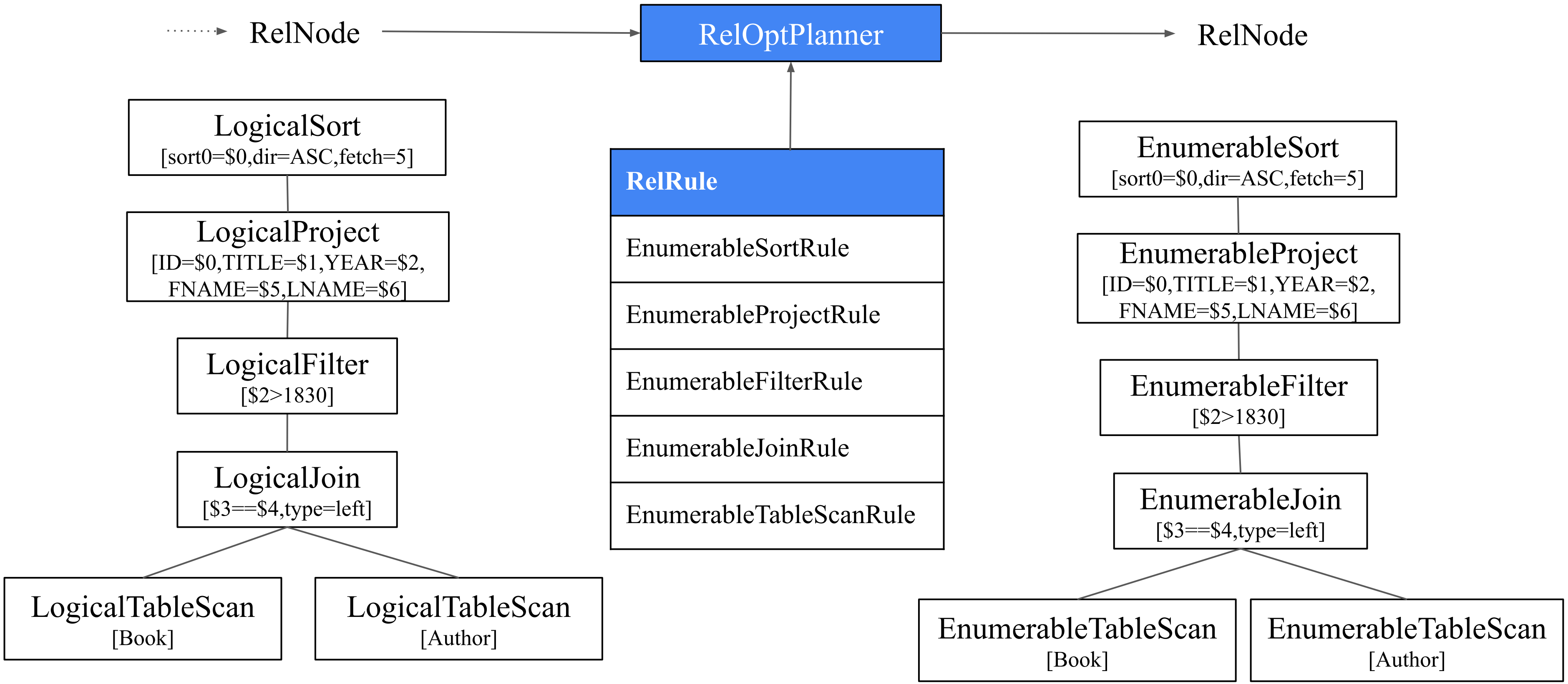

```sql
SELECT b.id, b.title, b.year, a.fname, a.lname
FROM Book AS b
LEFT OUTER JOIN Author a ON b.author = a.id
WHERE b.year > 1830
ORDER BY b.id
LIMIT 5
```



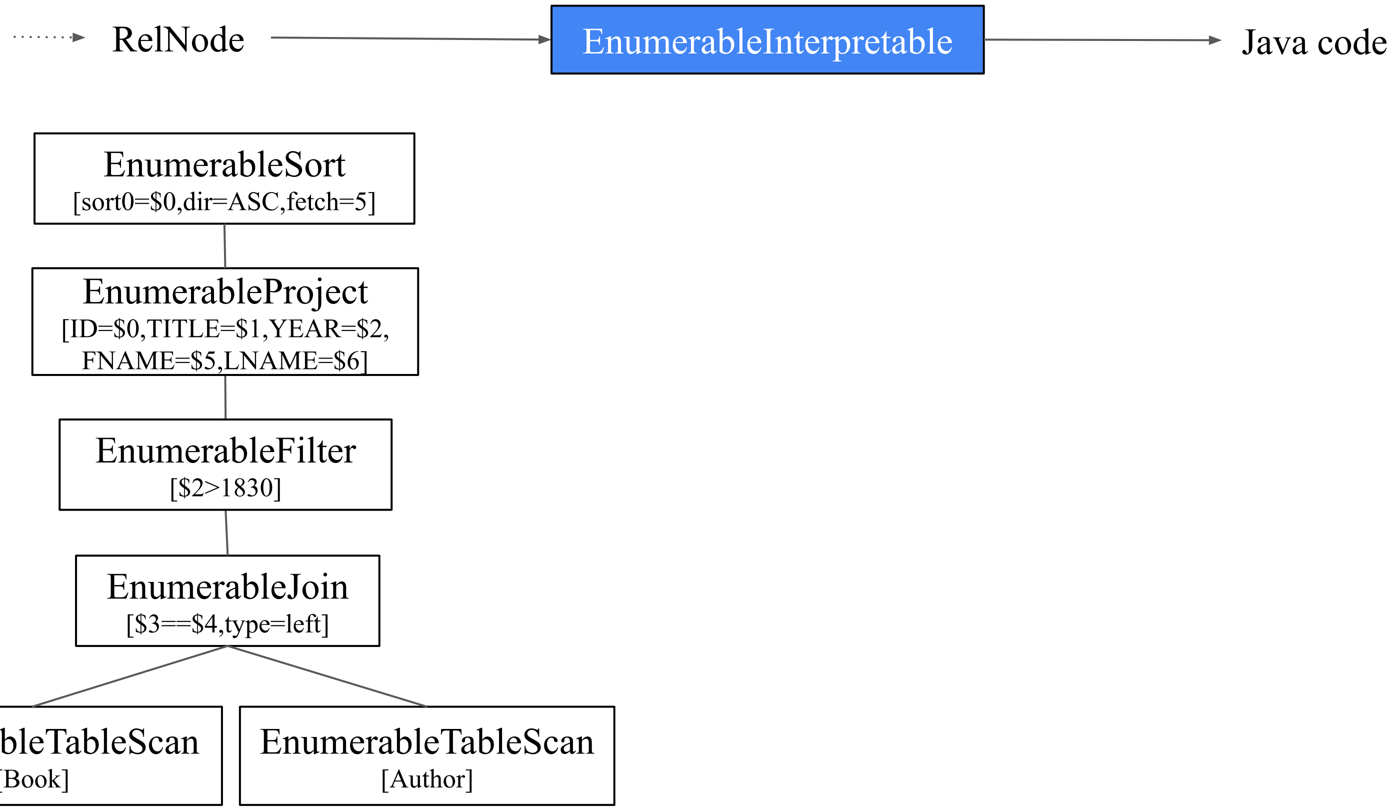
AST to logical plan



Logical to physical plan



Physical to Executable plan



4. Coding module I: Exercises (Homework)

Exercise I: Execute more SQL queries

Include GROUP BY and other types of clauses:

```
SELECT o.o_custkey, COUNT(*)
FROM orders AS o
GROUP BY o.o_custkey
```

Exercise I: Execute more SQL queries

Include GROUP BY and other types of clauses:

```
SELECT o.o_custkey, COUNT(*)
FROM orders AS o
GROUP BY o.o_custkey
```

- Missing rule to convert `LogicalAggregate` to `EnumerableAggregate`
- Add `EnumerableRules.ENUMERABLE_AGGREGATE_RULE` to the planner

Exercise II: Improve performance by applying more optimization rules

Push filter below the join:

```
SELECT c.c_name, o.o_orderkey, o.o_orderdate
FROM customer AS c
INNER JOIN orders AS o ON c.c_custkey = o.o_custkey
WHERE c.c_custkey < 3
ORDER BY c.c_name, o.o_orderkey
```

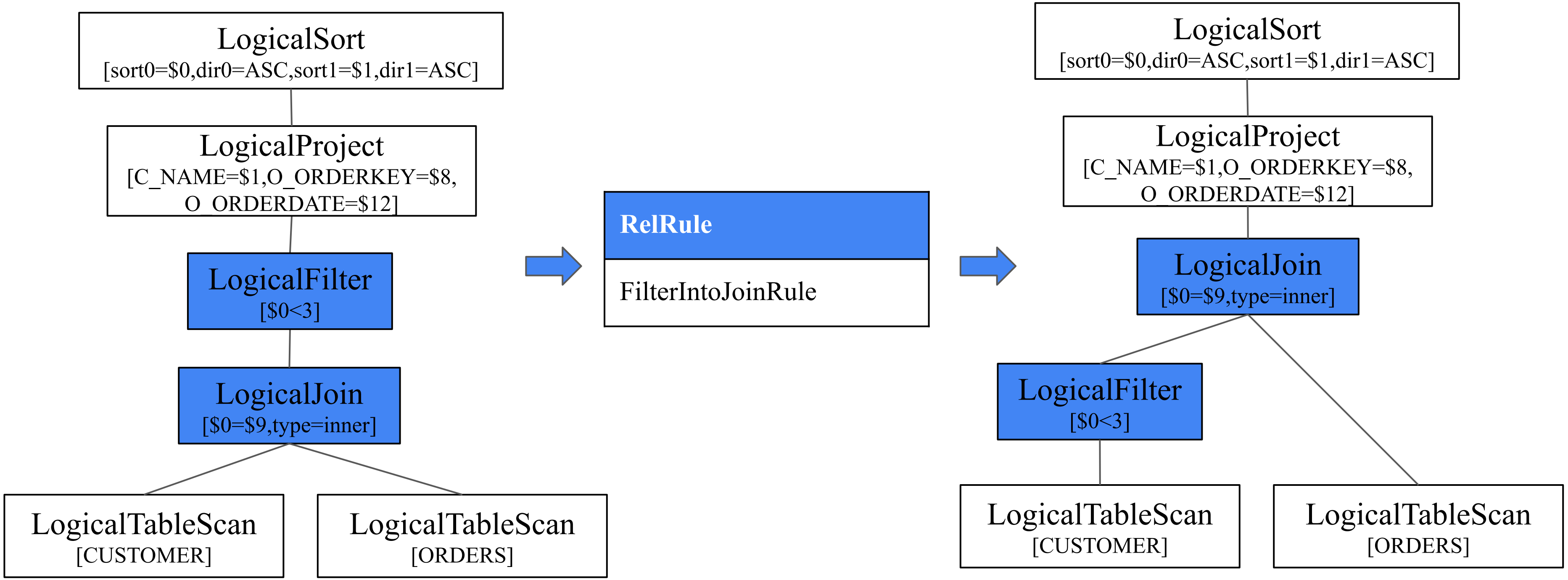
Exercise II: Improve performance by applying more optimization rules

Push filter below the join:

```sql
SELECT c.c_name, o.o_orderkey, o.o_orderdate
FROM customer AS c
INNER JOIN orders AS o ON c.c_custkey = o.o_custkey
WHERE c.c_custkey < 3
ORDER BY c.c_name, o.o_orderkey
```

1. Add rule `CoreRules.FILTER_INTO_JOIN` to the planner
2. Compare plans before and after (or logical and physical)
3. Check cost estimates by using `SqlExplainLevel.ALL_ATTRIBUTES`

Exercise II: Improve performance by applying more optimization rules



Exercise III: Use RelBuilder API to construct the logical plan

Open `LuceneBuilderProcessor.java` and complete TODOs

Q1:

```
SELECT o.o_custkey, COUNT(*)
FROM orders AS o
GROUP BY o.o_custkey
```

Q2:

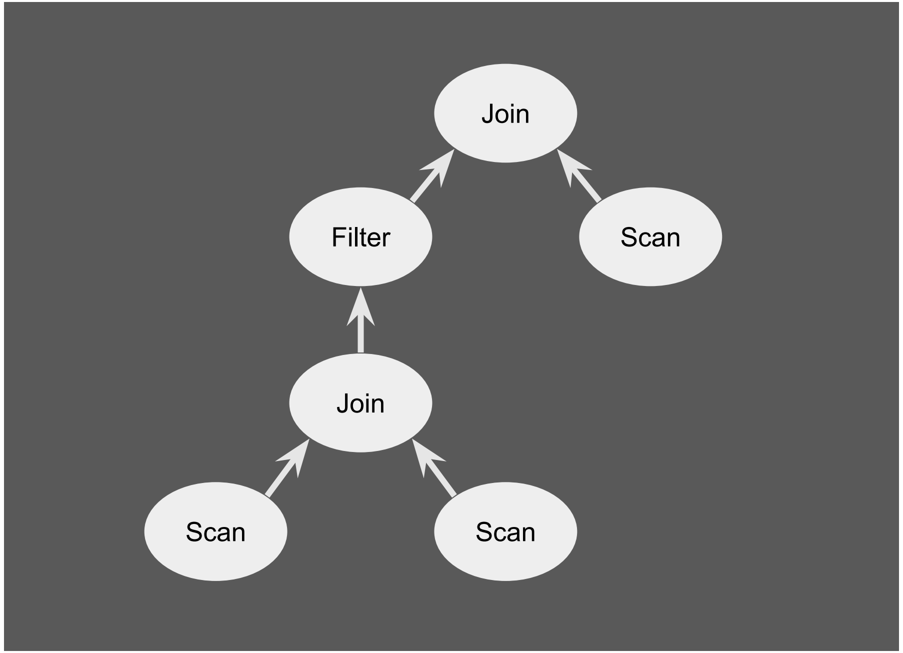
```
SELECT o.o_custkey, COUNT(*)
FROM orders AS o
WHERE o.o_totalprice > 220388.06
GROUP BY o.o_custkey
```

Exercise III: Use RelBuilder API to construct the logical plan

```
builder
    .scan("orders")
    .filter(
        builder.call(
            SqlStdOperatorTable.GREATER_THAN,
             builder.field("o_totalprice"),
             builder.literal(220388.06)))
    .aggregate(
        builder.groupKey("o_custkey"),
        builder.count());
```

5. Hybrid planning

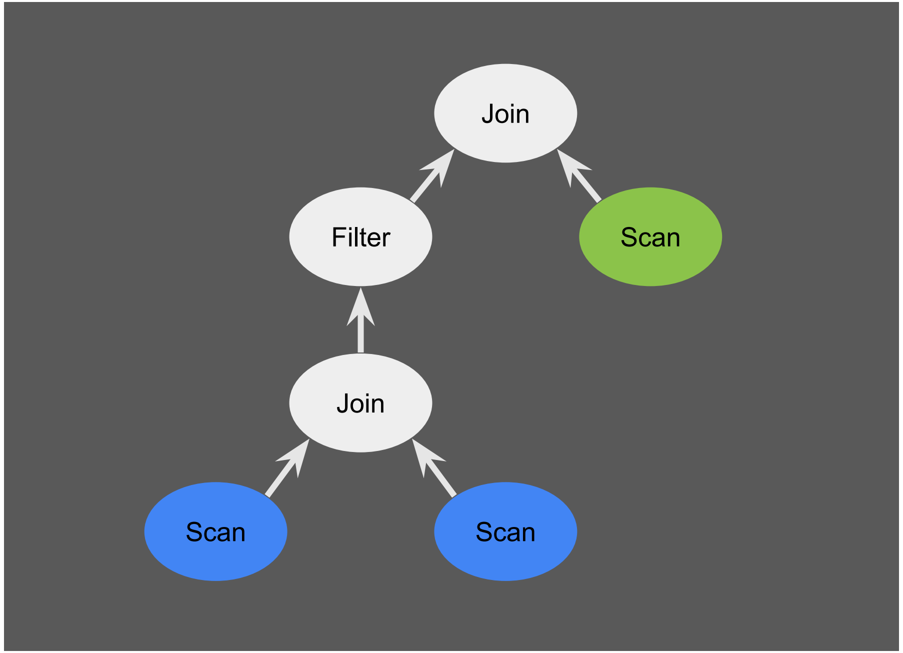
Calling convention



Initially all nodes belong to “logical” calling convention.

Logical calling convention cannot be implemented, so has infinite cost

Calling convention

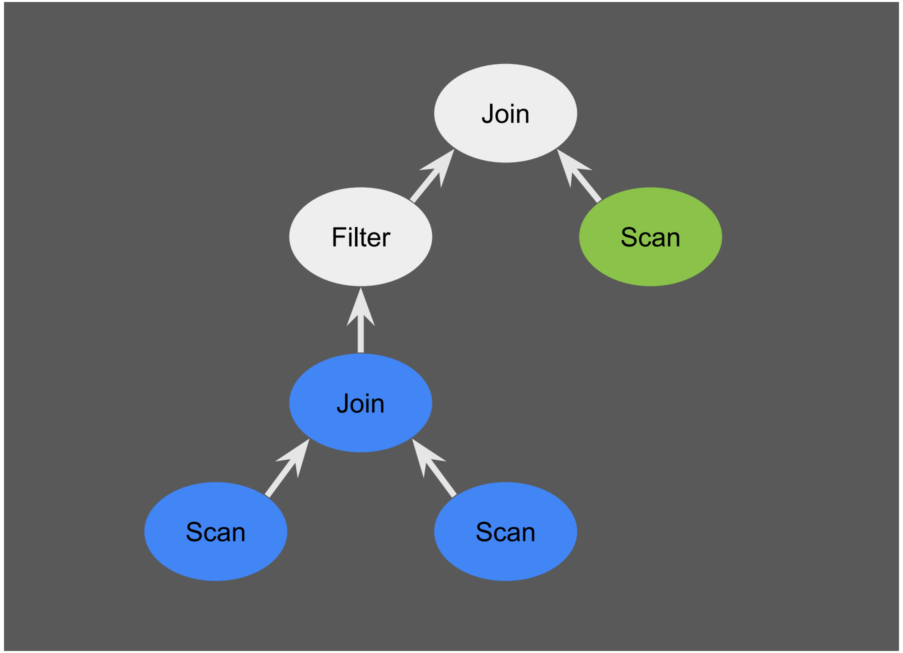


Tables can’t be moved so there is only one choice of calling convention for each table.

Examples:

- Enumerable
- Druid
- Drill
- HBase
- JDBC

Calling convention

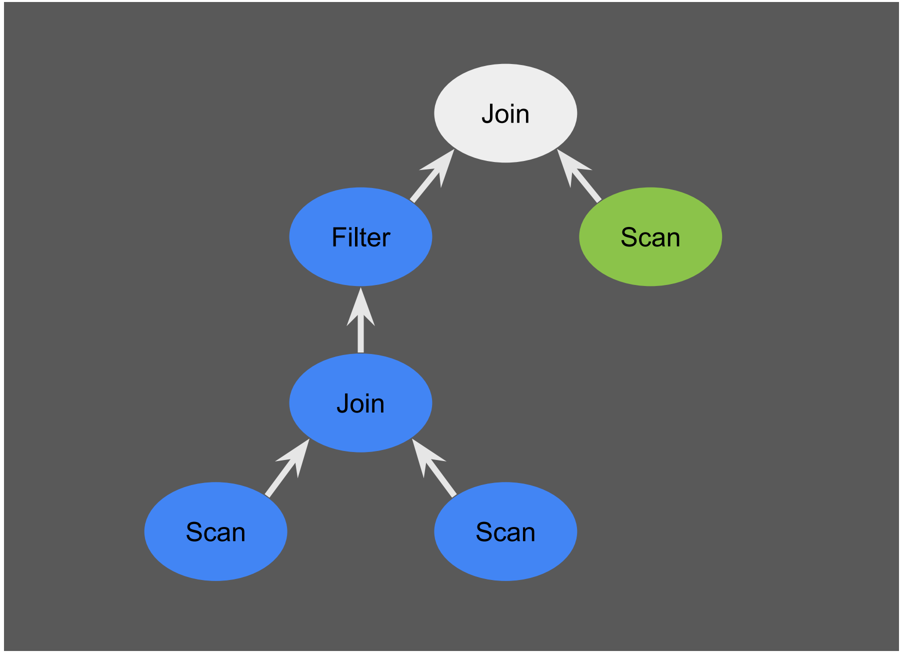


Rules fire to convert nodes to particular calling conventions.

The calling convention propagates through the tree.

Because this is Volcano, each node can have multiple conventions.

Calling convention

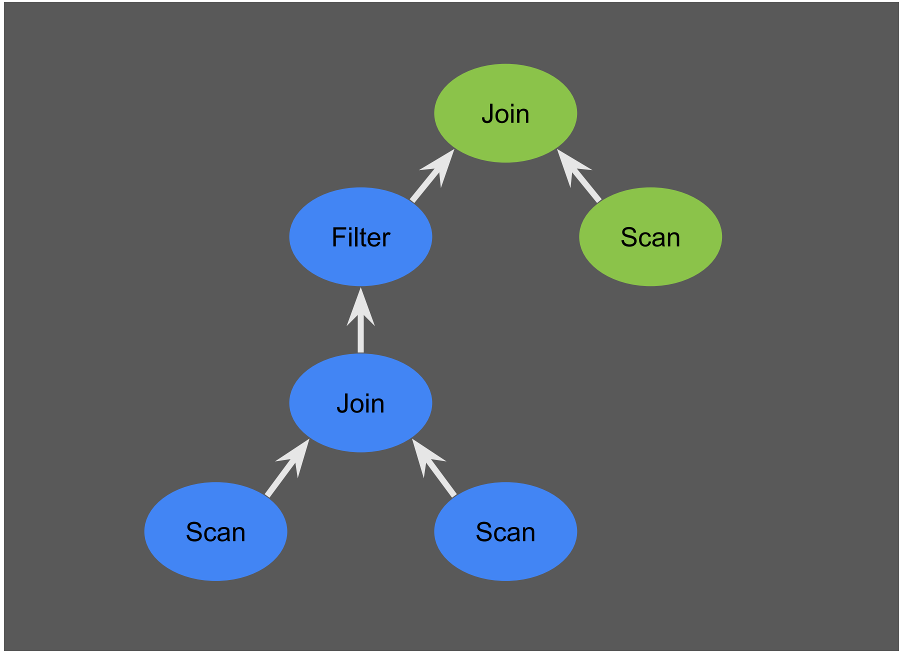


Rules fire to convert nodes to particular calling conventions.

The calling convention propagates through the tree.

Because this is Volcano, each node can have multiple conventions.

Calling convention

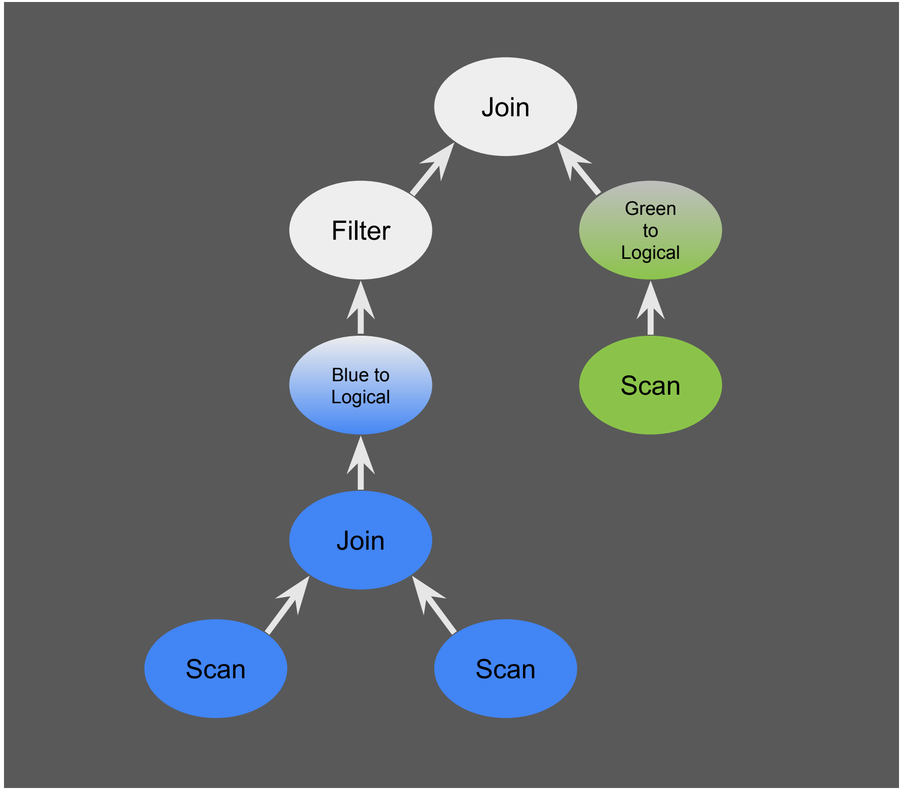


Rules fire to convert nodes to particular calling conventions.

The calling convention propagates through the tree.

Because this is Volcano, each node can have multiple conventions.

Converters



To keep things honest, we need to insert a **converter** at each point where the convention changes.

(Recall: Volcano has an enforcer for each trait. Convention is a physical property, and converter is the enforcer.)

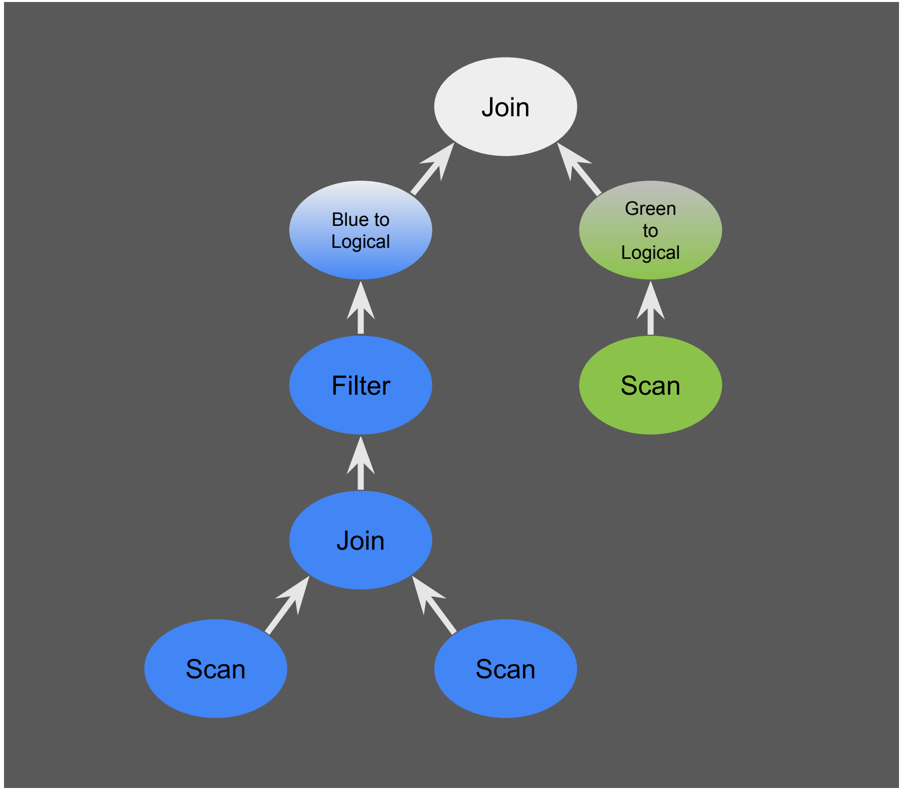
BlueFilterRule:

LogicalFilter(BlueToLogical(Blue b))

→

BlueToLogical(BlueFilter(b))

Converters



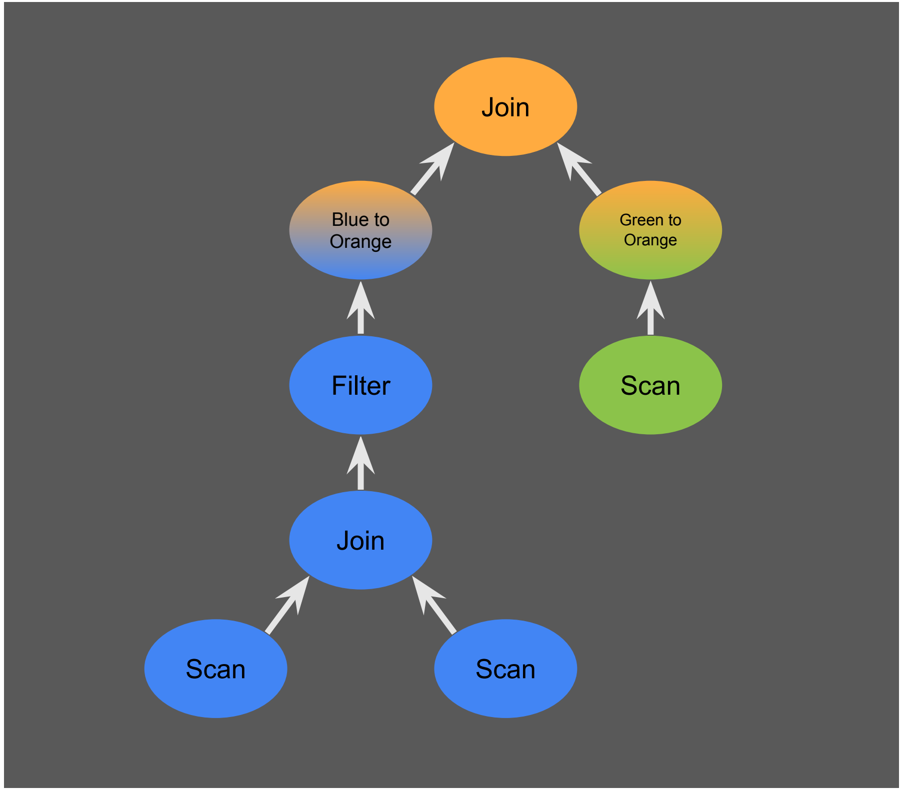
To keep things honest, we need to insert a **converter** at each point where the convention changes.

(Recall: Volcano has an enforcer for each trait. Convention is a physical property, and converter is the enforcer.)

```
BlueFilterRule:

LogicalFilter(BlueToLogical(Blue b))
  →
BlueToLogical(BlueFilter(b))
```

Generating programs to implement hybrid plans



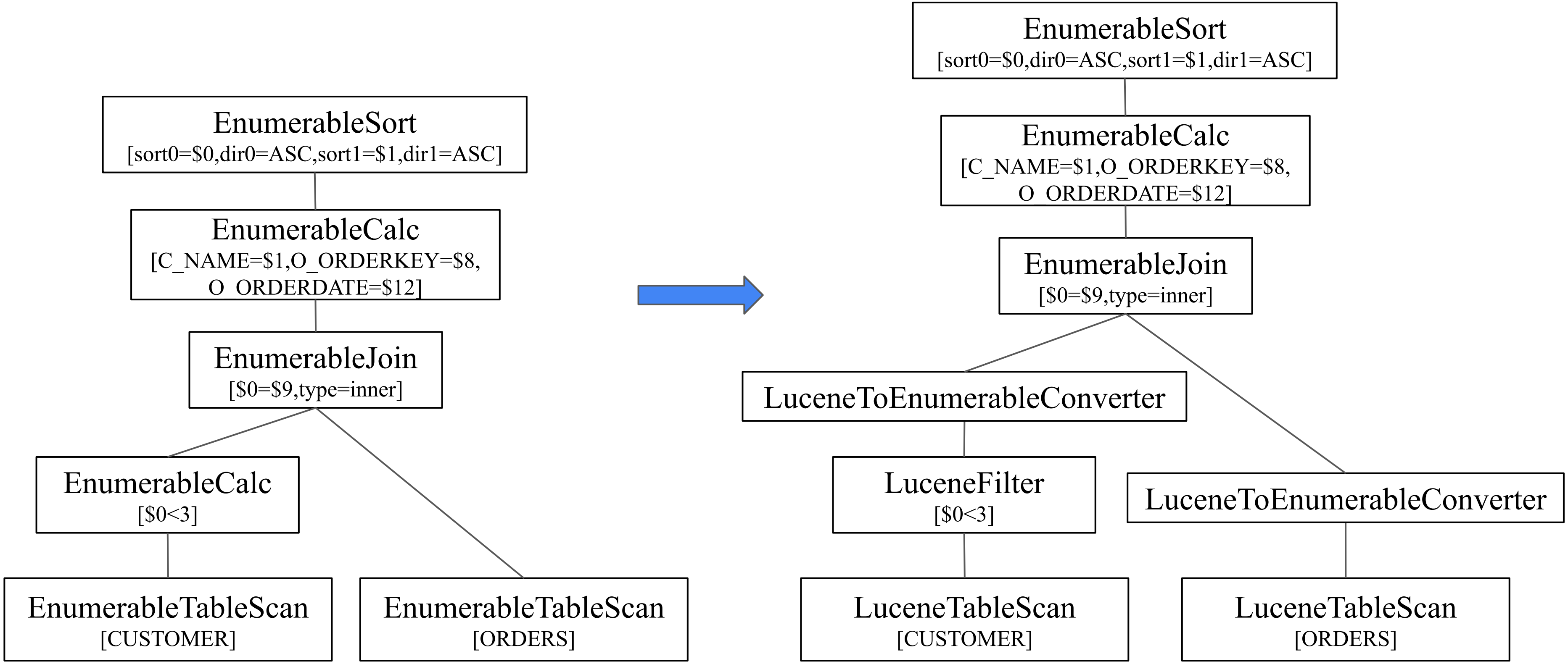
Hybrid plans are glued together using an **engine** - a convention that does not have a storage format. (Example engines: Drill, Spark, Presto.)

To implement, we generate a program that calls out to query1 and query2.

The "Blue-to-Orange" converter is typically a function in the Orange language that embeds a Blue query. Similarly "Green-to-Orange".

6. Coding module II: Custom operators/rules (Homework)

What we want to achieve?



What do we need?

Two calling conventions:

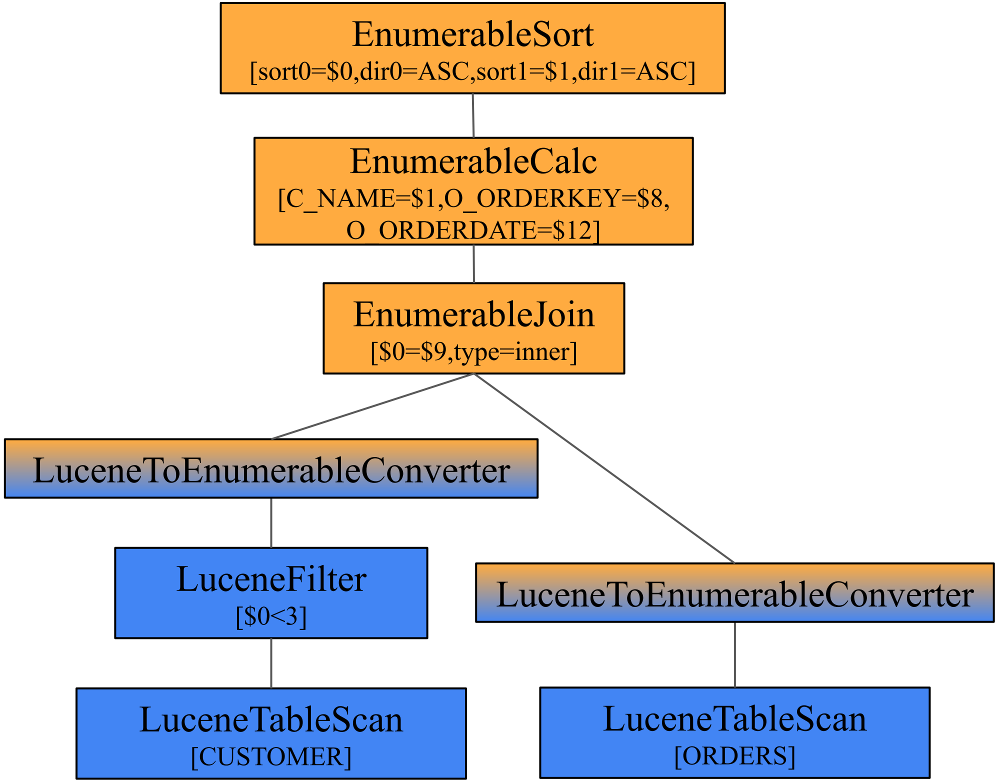
1. **Enumerable**
2. **Lucene**

Three custom operators:

1. `LuceneTableScan`
2. `LuceneToEnumerableConverter`
3. `LuceneFilter`

Three custom conversion rules:

1. `LogicalTableScan` → `LuceneTableScan`
2. `LogicalFilter` → `LuceneFilter`
3. `LuceneANY` → `LuceneToEnumerableConverter`



What do we need?

Two calling conventions:

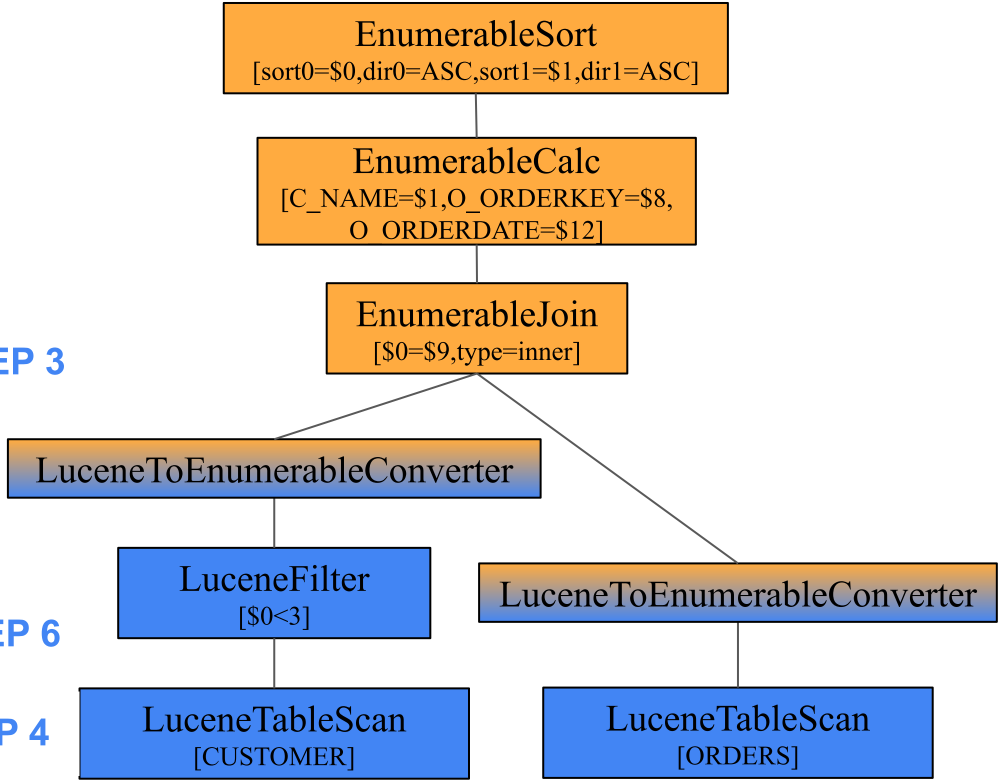
1. **Enumerable**
2. **Lucene**

Three custom operators:

1. `LuceneTableScan` **STEP 1**
2. `LuceneToEnumerableConverter` **STEP 3**
3. `LuceneFilter` **STEP 5**

Three custom conversion rules:

1. `LogicalTableScan` → `LuceneTableScan` **STEP 2**
2. `LogicalFilter` → `LuceneFilter` **STEP 6**
3. `LuceneANY` → `LuceneToEnumerableConverter` **STEP 4**



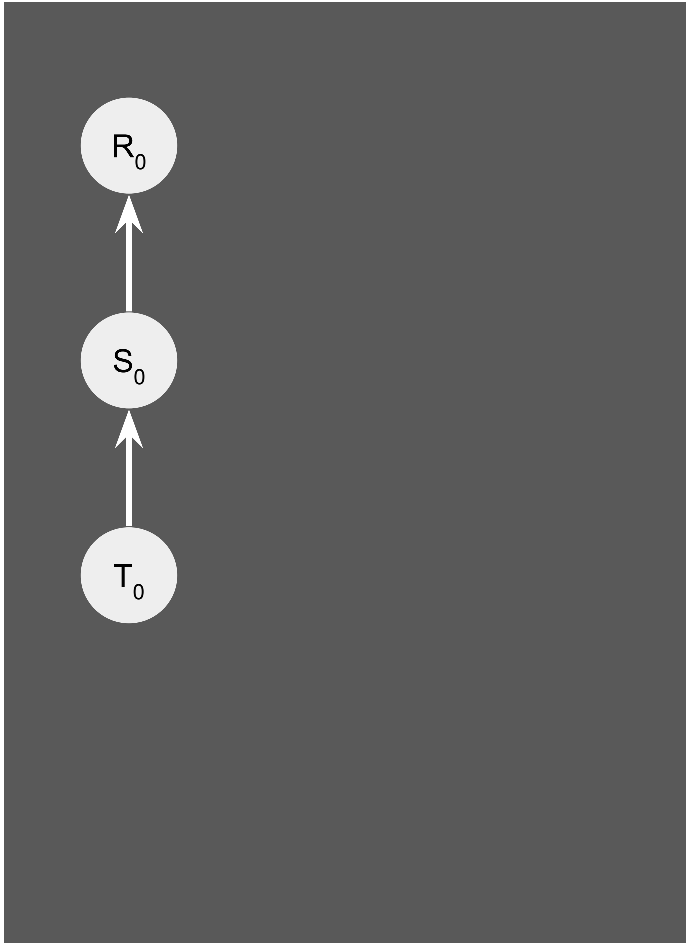
7. Volcano Planner Internals

Volcano planning algorithm

Based on two papers by Goetz Graefe in the 1990s (Volcano, Cascades), now the industry standard for cost-based optimization.

Dynamic programming: to optimize a relational expression R_0, convert it into equivalent expressions $\{R_1, R_2, \ldots\}$, and pick the one with the lowest cost.

Much of the cost of R is the cost of its input(s). So we apply dynamic programming to its inputs, too.

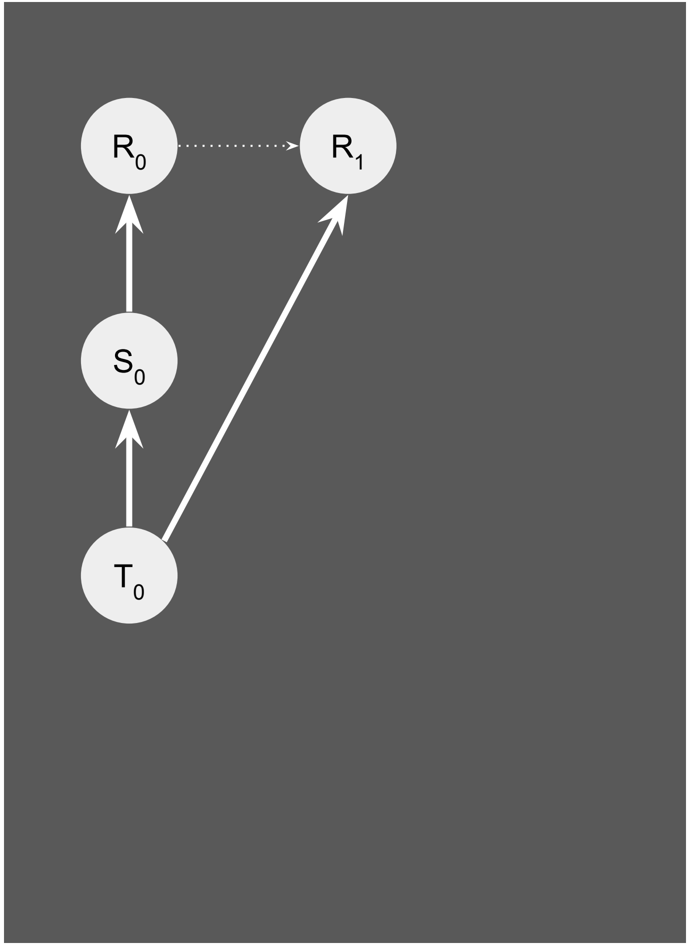


Volcano planning algorithm

Based on two papers by Goetz Graefe in the 1990s (Volcano, Cascades), now the industry standard for cost-based optimization.

Dynamic programming: to optimize a relational expression R_0, convert it into equivalent expressions $\{R_1, R_2, \dots\}$, and pick the one with the lowest cost.

Much of the cost of R is the cost of its input(s). So we apply dynamic programming to its inputs, too.

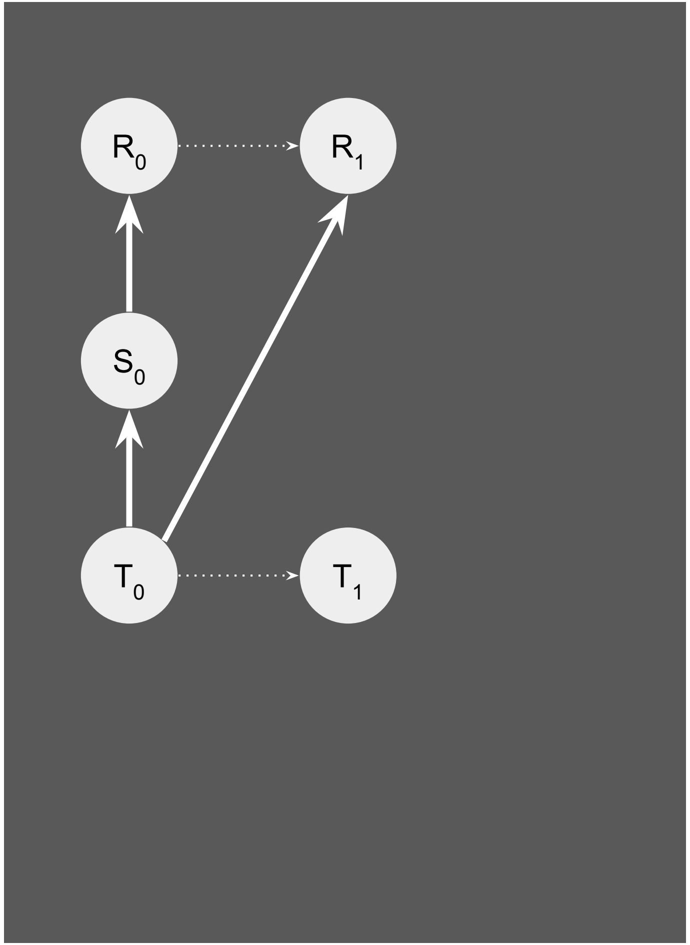


Volcano planning algorithm

Based on two papers by Goetz Graefe in the 1990s (Volcano, Cascades), now the industry standard for cost-based optimization.

Dynamic programming: to optimize a relational expression R_0, convert it into equivalent expressions $\{R_1, R_2, \ldots\}$, and pick the one with the lowest cost.

Much of the cost of R is the cost of its input(s). So we apply dynamic programming to its inputs, too.

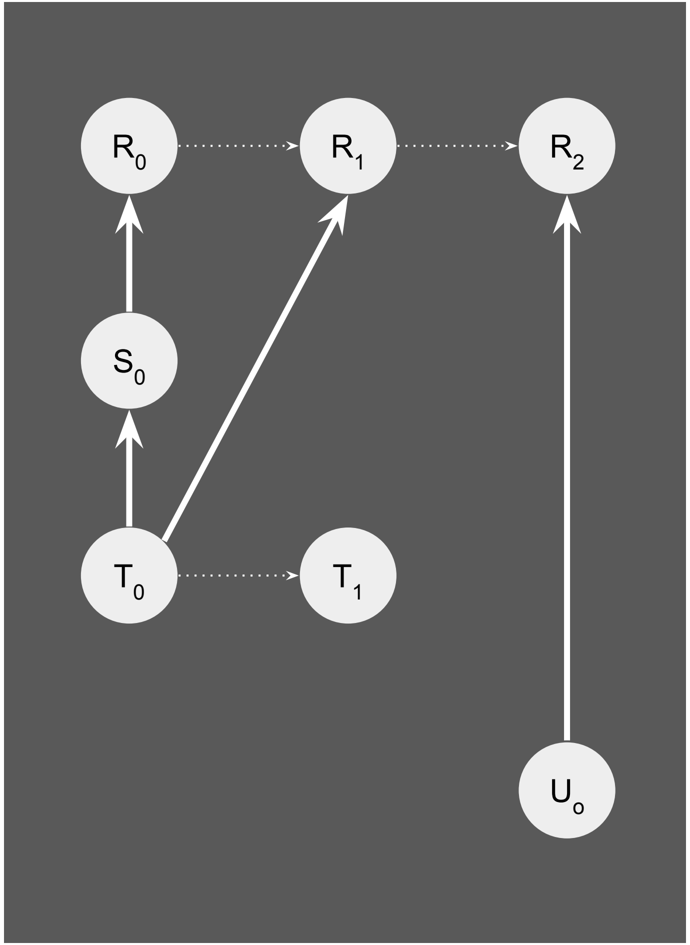


Volcano planning algorithm

Based on two papers by Goetz Graefe in the 1990s (Volcano, Cascades), now the industry standard for cost-based optimization.

Dynamic programming: to optimize a relational expression R_0, convert it into equivalent expressions $\{R_1, R_2, \ldots\}$, and pick the one with the lowest cost.

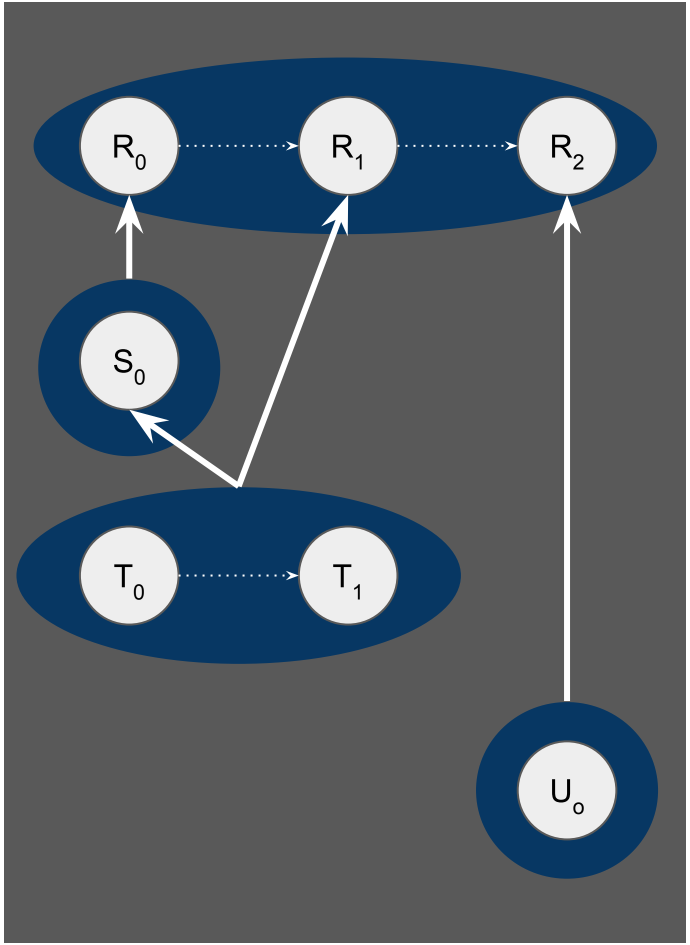
Much of the cost of R is the cost of its input(s). So we apply dynamic programming to its inputs, too.



Volcano planning algorithm

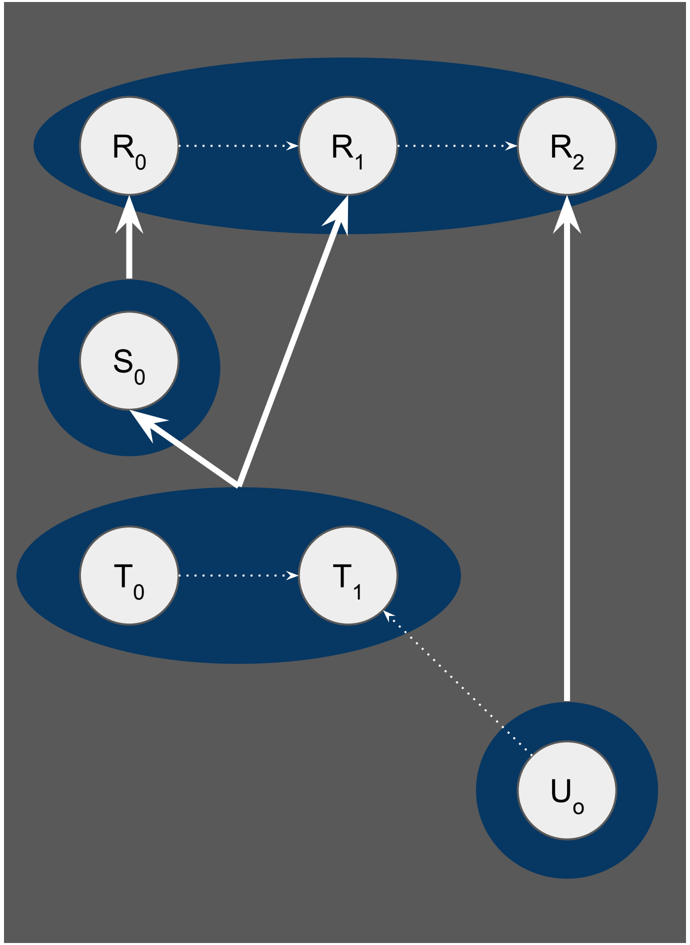
We keep equivalence sets of expressions (**`class RelSet`**).

Each input of a relational expression is an equivalence set + required physical properties (**`class RelSubset`**).



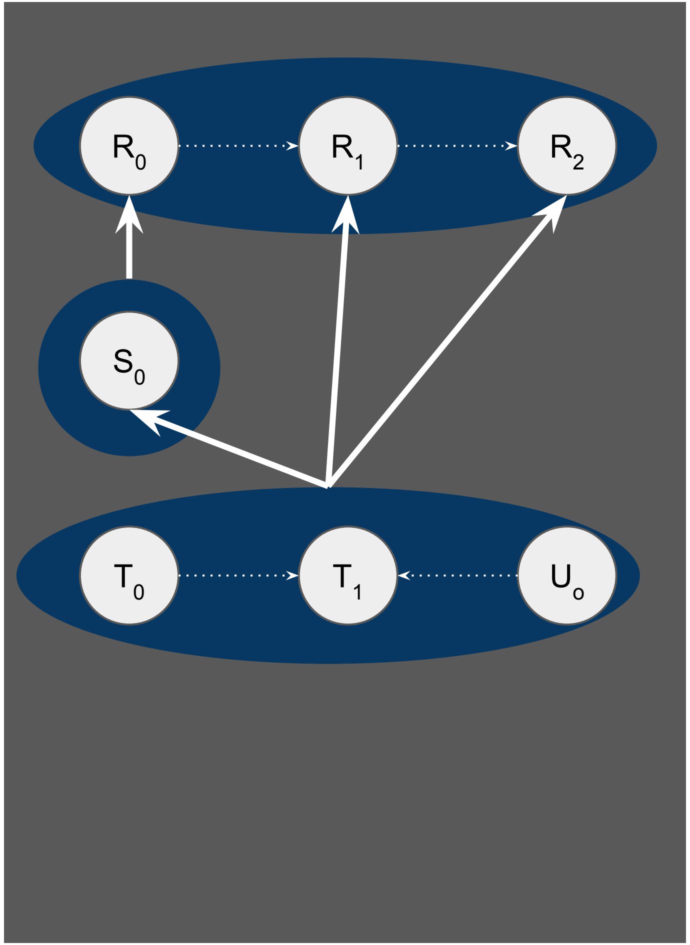
Volcano planning algorithm

Each relational expression has a memo (digest), so we will recognize it if we generate it again.



Volcano planning algorithm

If an expression transforms to an expression in another equivalence set, we can **merge those equivalence sets**.



Matches and queues

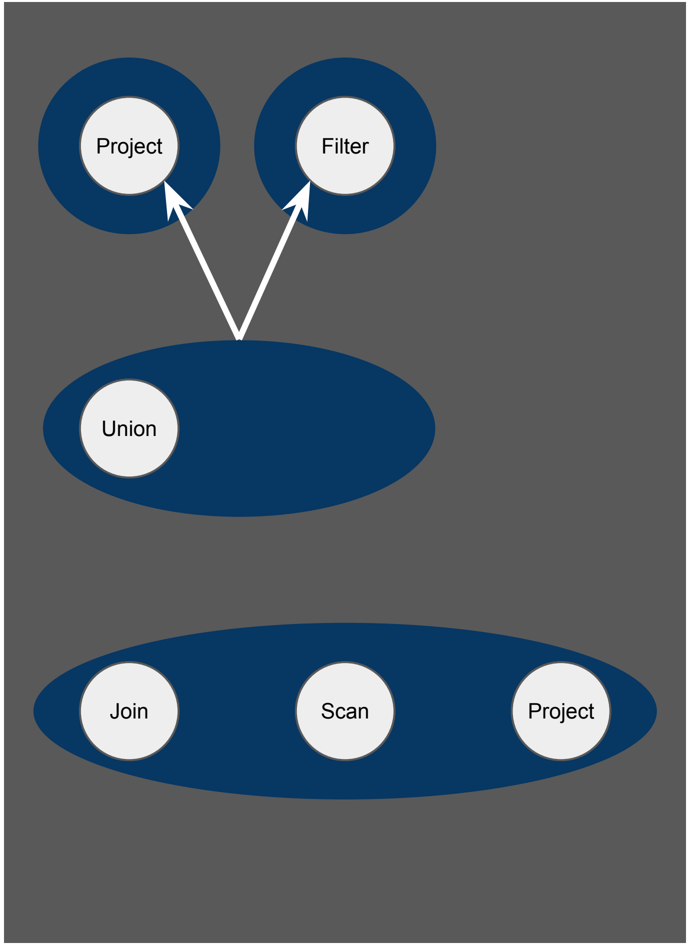
We register a new RelNode by adding it to a RelSet.

Each rule instance declares a pattern of RelNode types (and other properties) that it will match.

Suppose we have:

- Filter-on-Project
- Project-on-Project
- Project-on-Join

On register, we detect rules that are newly matched.



Matches and queues

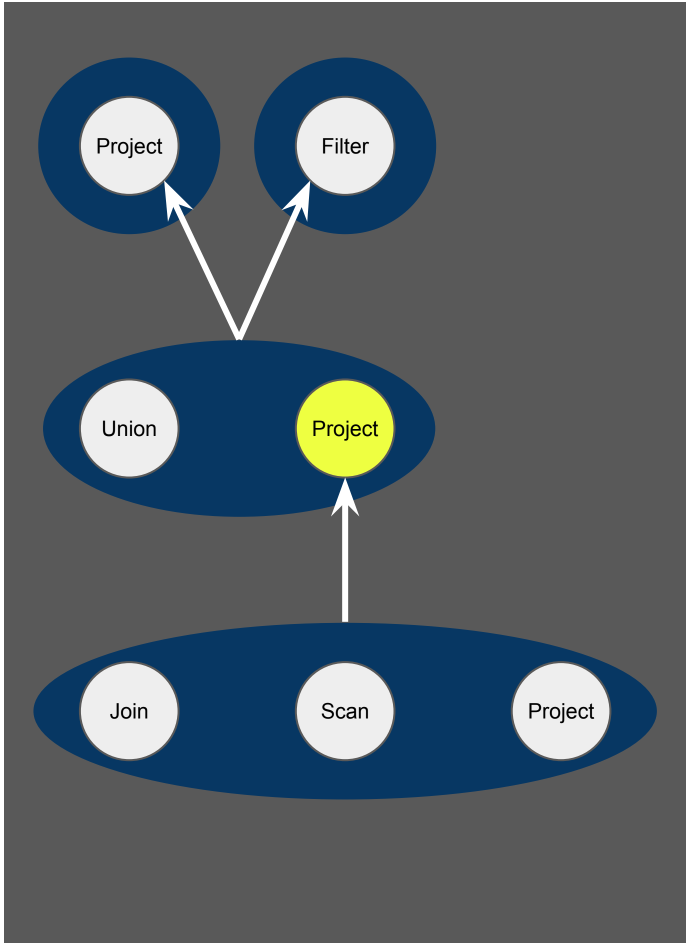
We register a new RelNode by adding it to a RelSet.

Each rule instance declares a pattern of RelNode types (and other properties) that it will match.

Suppose we have:

- Filter-on-Project
- Project-on-Project
- Project-on-Join

On register, we detect rules that are newly matched.



Matches and queues

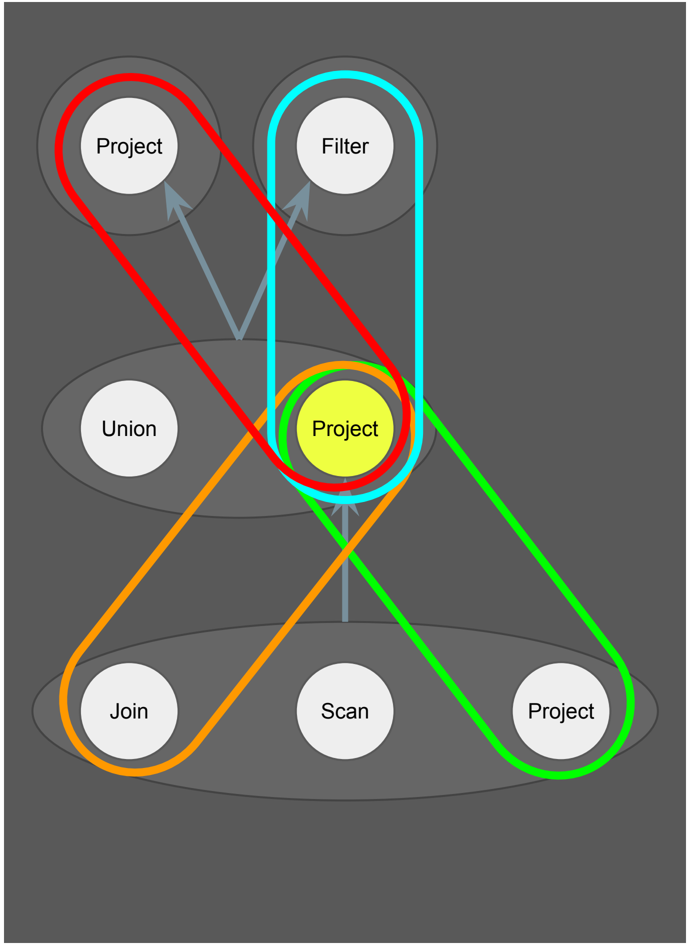
We register a new RelNode by adding it to a RelSet.

Each rule instance declares a pattern of RelNode types (and other properties) that it will match.

Suppose we have:

- Filter-on-Project
- Project-on-Project
- Project-on-Join

On register, we detect rules that are newly matched. (4 matches.)



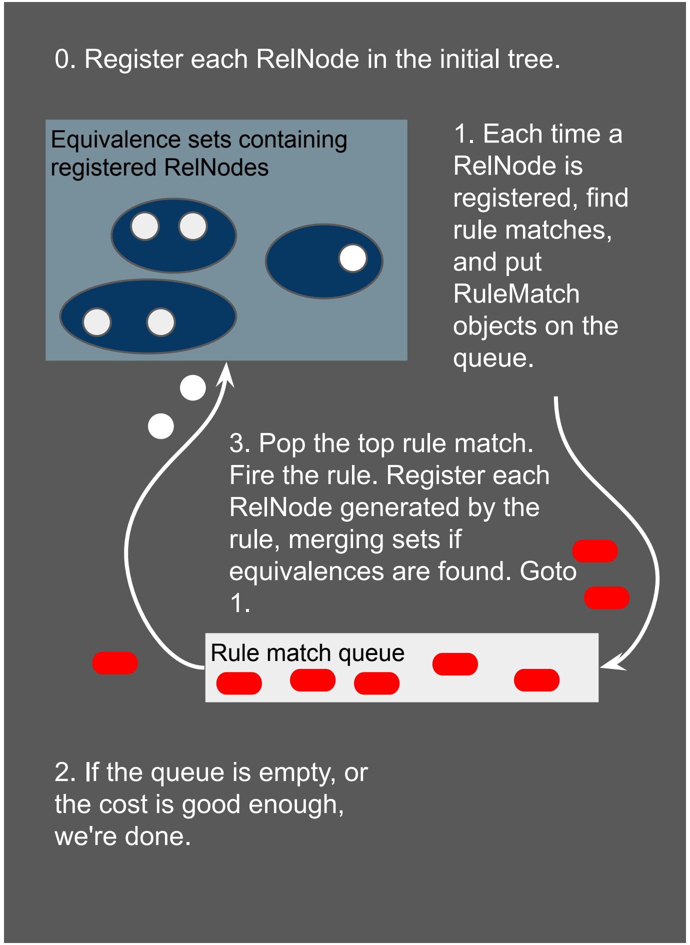
Matches and queues

Should we fire these matched rules immediately?

No! Because rule match #1 would generate new matches… which would generate new matches… and we'd never get to match #2. Instead, we put the matched rules on a queue.

The queue allows us to:

- Search breadth-first (rather than depth-first)
- Prioritize (fire more "important" rules first)
- Potentially terminate when we have a "good enough" plan



Other planner engines, same great rules

Three planner engines:

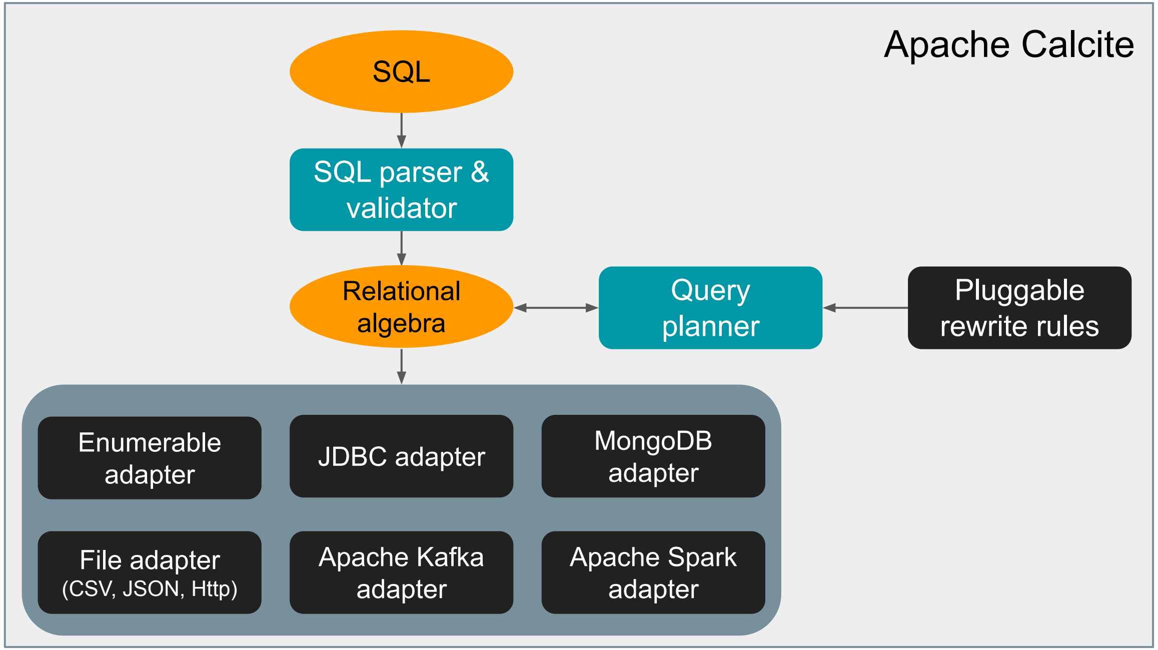
- Volcano
- Volcano top-down (Cascades style)
- Hep applies rules in a strict "program"

The same rules are used by all engines.

It takes a lot of time effort to write a high-quality rule. Rules can be reused, tested, improved, and they compose with other rules. Calcite's library of rules is valuable.

8. Dialects

Calcite architecture



At what points in the Calcite stack do 'languages' exist?

- Incoming SQL
- Validating SQL against built-in operators
- Type system (e.g. max size of INTEGER type)
- JDBC adapter generates SQL
- Other adapters generate other languages

Parsing & validating SQL - what knobs can I turn?

```
SELECT deptno AS d,
  SUM(sal) AS [sumSal]
FROM [HR].[Emp]
WHERE ename NOT ILIKE "A%"
GROUP BY d
ORDER BY 1, 2 DESC
```

`PARSER_FACTORY = "org.apache.calcite.sql.parser.impl.SqlParserImpl.FACTORY"`

`Lex.unquotedCasing = Casing.TO_UPPER`

`Lex.quoting = Quoting.BRACKET`

`Lex.quotedCasing = Casing.UNCHANGED`

`Lex.charLiteralStyle = CharLiteralStyle.BQ_DOUBLE`

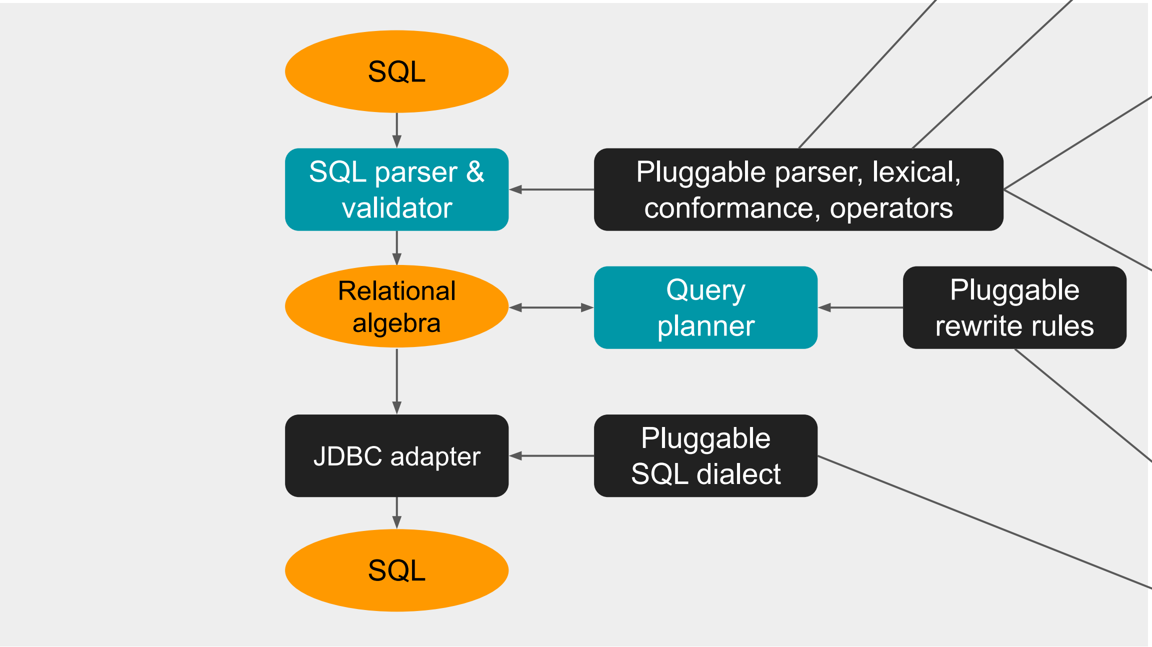
`FUN = "postgres"` (`ILIKE` is not standard SQL)

`SqlConformance.isGroupByAlias() = true`

`SqlConformance.isSortByOrdinal() = true`

`SqlValidator.Config.defaultNullCollation = HIGH`

SQL dialect - APIs and properties




```
interface SqlParserImplFactory
```

```
CalciteConnectionProperty.LEX
enum Lex
enum Quoting
enum Casing
enum CharLiteralStyle
```

```
CalciteConnectionProperty.CONFORMANCE
interface SqlConformance
```

```
CalciteConnectionProperty.FUN
interface SqlOperatorTable
class SqlStdOperatorTable
class SqlLibraryOperators
class SqlOperator
class SqlFunction extends SqlOperator
class SqlAggFunction extends SqlFunction
```

```
class RelRule
```

```
class SqlDialect
interface SqlDialectFactory
```

Contributing a dialect (or anything!) to Calcite

For your first code contribution, pick a small bug or feature.

Introduce yourself! Email dev@, saying what you plan to do.

Create a JIRA case describing the problem.

To understand the code, find similar features. Run their tests in a debugger.

Write 1 or 2 tests for your feature.

Submit a pull request (PR).

From: Charles Givre <c...@gmail.com>
To: de...@calcite.apache.org
Subject: SQL Dialect Question
Date: 2021/07/28 14:25:32
List: dev@calcite.apache.org

Hi Calcite Devs!
I'm interested in writing a SQL dialect for Apache Drill and contributing it to Calcite. What is the process for contributing a dialect? I'm asking because I didn't see any unit tests for dialects.
Thanks!
-- C

View Source Permalink Reply

From: Stamatis Zampetakis <z...@gmail.com>
Subject: Re: SQL Dialect Question
Date: 2021/07/28 14:37:44
List: dev@calcite.apache.org

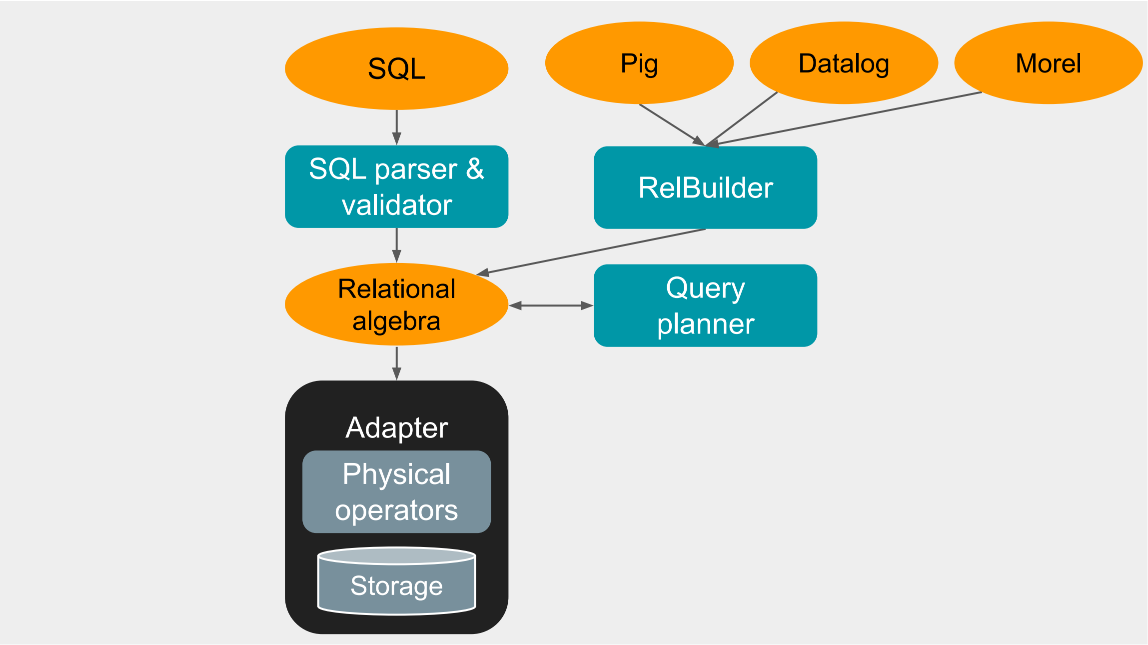
Hi Charles,

Start by creating a JIRA and then you can do more or less what was done for EXASOL dialect [1].
Tests for dialects are usually added in RelToSqlConverterTest as you can see also in [1].
If the new dialect is very similar to an existing one then maybe there is no reason to create a new one.

Best,
Stamatis

[1]
https://github.com/apache/calcite/commit/f928e073c384010c294370b63ffb748c15caab8a

Other front-end languages

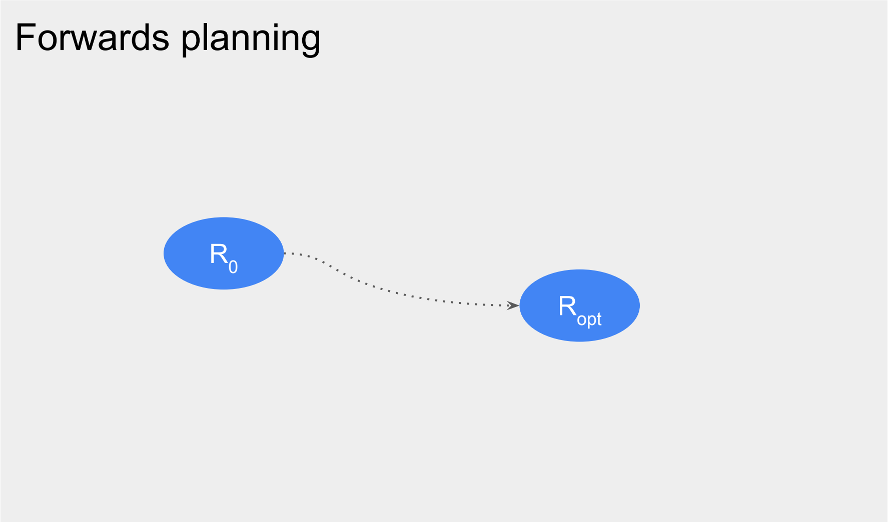


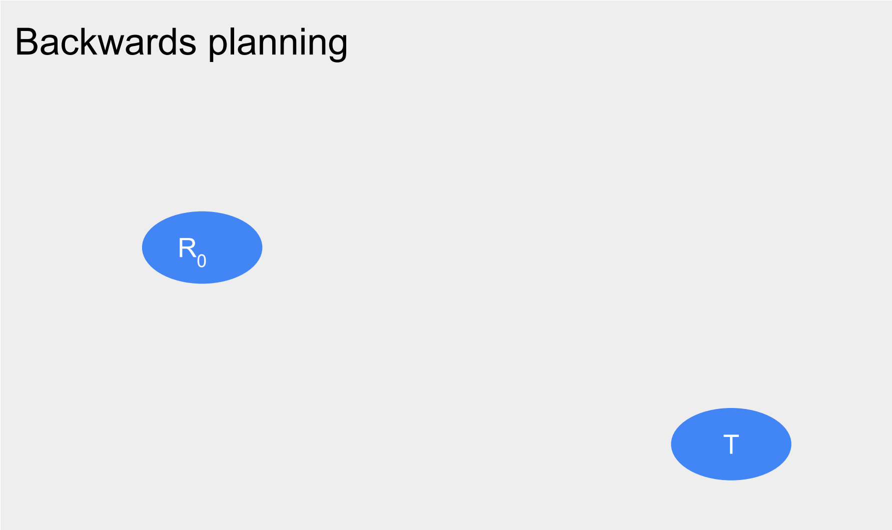
Calcite is an excellent platform for implementing your own data language

Write a parser for your language, use RelBuilder to translate to relational algebra, and you can use any of Calcite's back-end implementations

9. Materialized views

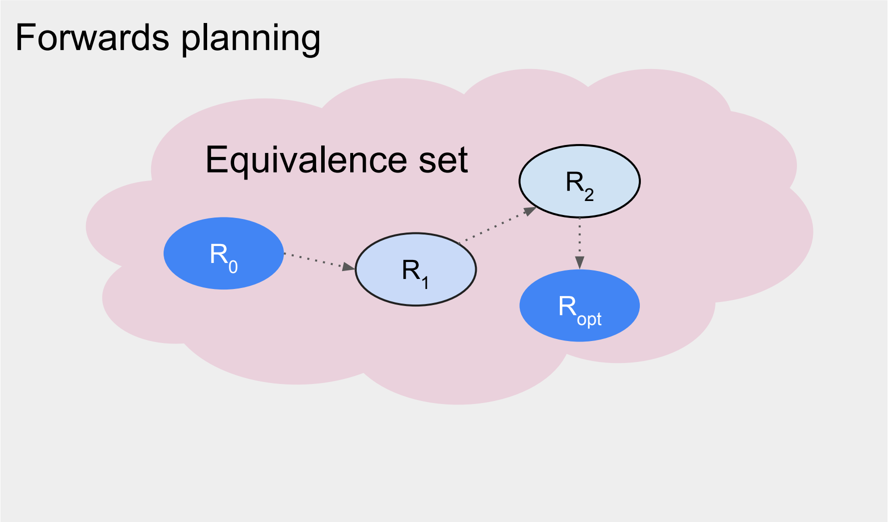
Backwards planning

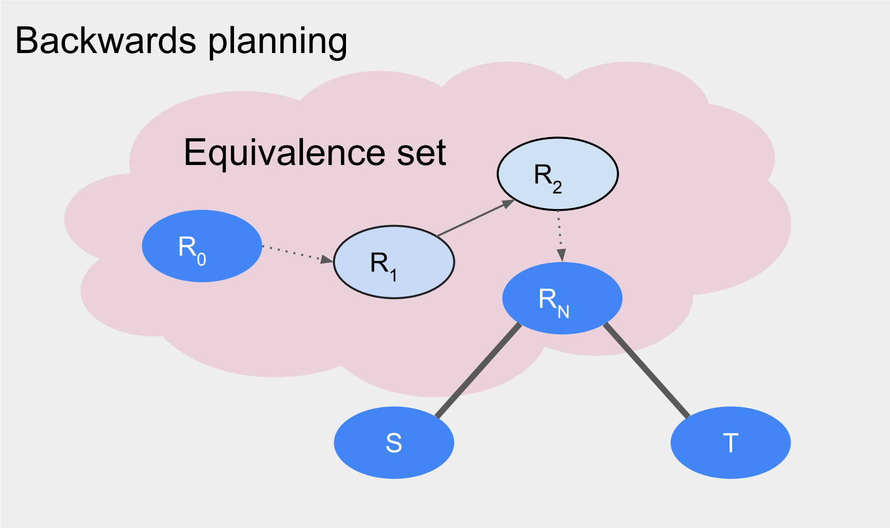




Until now, we have seen forward planning. **Forward planning** transforms an expression (R_0) to many equivalent forms and picks the one with lowest cost (R_{opt}). **Backwards planning** transforms an expression to an equivalent form (R_N) that contains a target expression (T).

Backwards planning





Until now, we have seen forward planning. **Forward planning** transforms an expression (R_0) to many equivalent forms and picks the one with lowest cost (R_{opt}). **Backwards planning** transforms an expression to an equivalent form (R_N) that contains a target expression (T).

Applications of backwards planning

Indexes (e.g. b-tree indexes). An index is a derived data structure whose contents can be described as a relational expression (generally project-sort). When we are planning a query, it already exists (i.e. the cost has already been paid).

Summary tables. A summary table is a derived data structure (generally filter-project-join-aggregate).

Replicas with different physical properties (e.g. copy the table from New York to Tokyo, or copy the table and partition by `month(orderDate)`, sort by `productId`).

Incremental view maintenance. Materialized view V is populated from base table T. Yesterday, we populated V with $V_0 = Q(T_0)$. Today we want to make its contents equal to $V_1 = Q(T_1)$. Can we find and apply a delta query, $dQ = Q(T_1 - T_0)$?

Materialized views in Calcite

```json
{
  "schemas": {
    "name": "HR",
    "tables": [ {
      "name": "emp"
    } ],
    "materializations": [ {
      "table": "i_emp_job",
      "sql": "SELECT job, empno
            FROM emp
            ORDER BY job, empno"
    }, {
      "table": "add_emp_deptno",
      "sql": "SELECT deptno,
               SUM(sal) AS ss, COUNT(*) AS c
            FROM emp
            GROUP BY deptno"
    } ]
  }
}
```

```java
/** Transforms a relational expression into a
 * semantically equivalent relational expression,
 * according to a given set of rules and a cost
 * model. */
public interface RelOptPlanner {
  /** Defines an equivalence between a table and
    * a query. */
  void addMaterialization(
    RelOptMaterialization materialization);

  /** Finds the most efficient expression to
   * implement this query. */
  RelNode findBestExp();
}

/** Records that a particular query is materialized
* by a particular table. */
public class RelOptMaterialization {
  public final RelNode tableRel;
  public final List<String> qualifiedTableName;
  public final RelNode queryRel;
}
```

You can define materializations in a JSON model, via the planner API, or via `CREATE MATERIALIZED VIEW` DDL (not shown).

More about materialized views

- There are **several algorithms** to rewrite queries to match materialized views
- A **lattice** is a data structure to model a star schema
- Calcite has **algorithms to recommend** an optimal set of summary tables for a lattice (given expected queries, and statistics about column cardinality)
- **Data profiling** algorithms estimate the cardinality of all combinations of columns

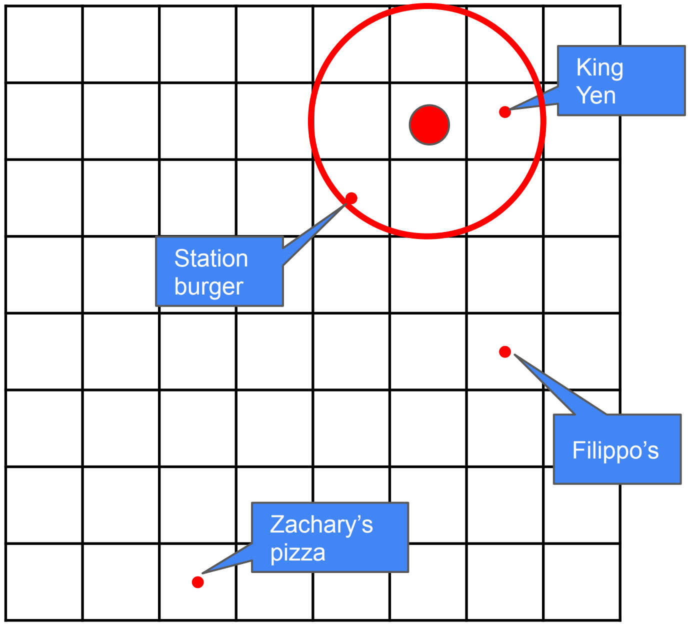
10. Working with spatial data

Spatial query

Find all restaurants within 1.5 distance units of my current location:

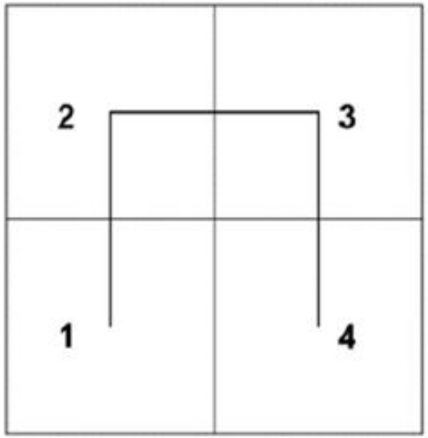
```
SELECT *
FROM Restaurants AS r
WHERE ST_Distance(
  ST_MakePoint(r.x, r.y),
  ST_MakePoint(6, 7)) < 1.5
```

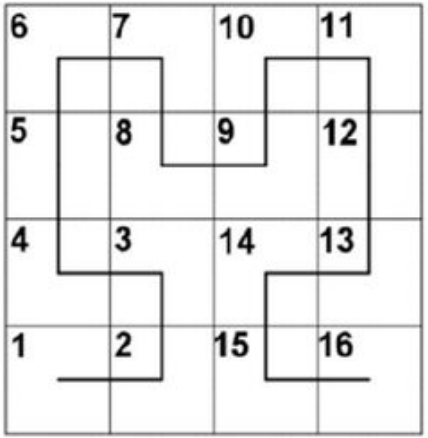
We cannot use a B-tree index (it can sort points by x or y coordinates, but not both) and specialized spatial indexes (such as R*-trees) are not generally available.

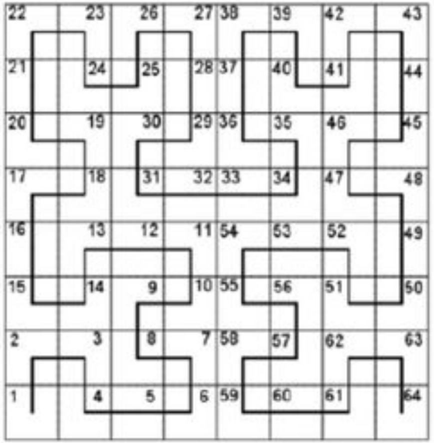


restaurant	x	y
Zachary’s pizza	3	1
King Yen	7	7
Filippo’s	7	4
Station burger	5	6

Hilbert space-filling curve







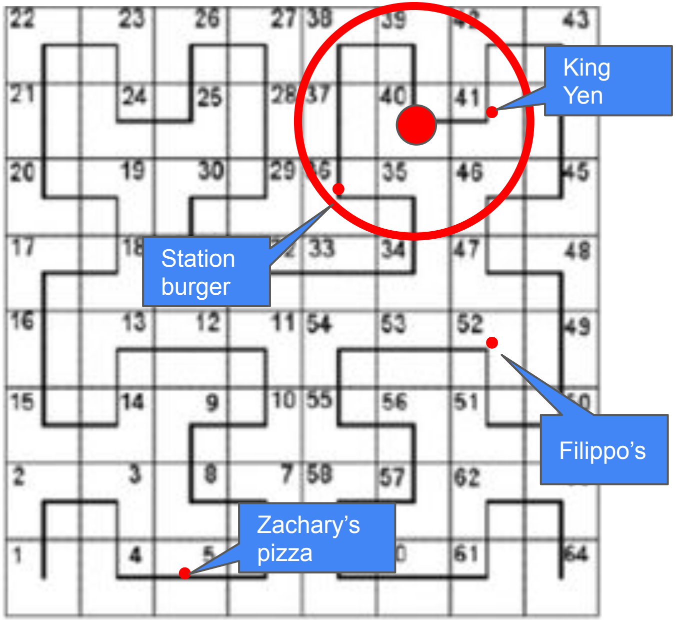
- A space-filling curve invented by mathematician David Hilbert
- Every (x, y) point has a unique position on the curve
- Points near to each other typically have Hilbert indexes close together

Using Hilbert index

Add restriction based on **h**, a restaurant's distance along the Hilbert curve

Must keep original restriction due to false positives

```
SELECT *
FROM Restaurants AS r
WHERE (r.h BETWEEN 35 AND 42
    OR r.h BETWEEN 46 AND 46)
AND ST_Distance(
  ST_MakePoint(r.x, r.y),
  ST_MakePoint(6, 7)) < 1.5
```



restaurant	x	y	h
Zachary's pizza	3	1	5
King Yen	7	7	41
Filippo's	7	4	52
Station burger	5	6	36

Telling the optimizer

1. Declare **h** as a generated column
2. Sort table by **h**

Planner can now convert spatial range queries into a range scan

Does not require specialized spatial index such as R*-tree

Very efficient on a sorted table such as HBase

There are similar techniques for other spatial patterns (e.g. region-to-region join)

```
CREATE TABLE Restaurants (
  restaurant VARCHAR(20),
  x DOUBLE,
  y DOUBLE,
  h DOUBLE GENERATED ALWAYS AS
     ST_Hilbert(x, y) STORED)
SORT KEY (h);
```

restaurant	x	y	h
Zachary's pizza	3	1	5
Station burger	5	6	36
King Yen	7	7	41
Filippo's	7	4	52

11. Research using Apache Calcite

One SQL to Rule Them All – an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables

An Industrial Paper

Edmon Begoli
Oak Ridge National Laboratory / Apache Calcite
Oak Ridge, Tennessee, USA
begoli@apache.org

Tyler Akidau
Google Inc. / Apache Beam
Seattle, WA, USA
takidau@apache.org

Fabian Hueske
Ververica / Apache Flink
Berlin, Germany
fhueske@apache.org

Julian Hyde
Looker Inc. / Apache Calcite
San Francisco, California, USA
jhyde@apache.org

Kathryn Knight
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
knightke@ornl.gov

Kenneth Knowles
Google Inc. / Apache Beam
Seattle, WA, USA
kenn@apache.org

ABSTRACT

Real-time data analysis and management are increasingly critical for today's businesses. SQL is the de facto *lingua franca* for these endeavors, yet support for robust streaming analysis and management with SQL remains limited. Many approaches restrict semantics to a reduced subset of features and/or require a suite of non-standard constructs. Additionally, use of event timestamps to provide native support for analyzing events according to when they actually occurred is not pervasive, and often comes with important limitations.

We present a three-part proposal for integrating robust streaming into the SQL standard, namely: (1) time-varying relations as a foundation for classical tables as well as streaming data, (2) event time semantics, (3) a limited set of optional keyword extensions to control the materialization of time-varying query results. Motivated and illustrated using exam-

CCS CONCEPTS

• **Information systems** → **Stream management**; **Query languages**;

KEYWORDS

stream processing, data management, query processing

ACM Reference Format:
Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. 2019. One SQL to Rule Them All – an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables: An Industrial Paper. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3299869.3314040

Tempura: A General Cost-Based Optimizer Framework for Incremental Data Processing

Zuozhi Wang[1], Kai Zeng[2], Botong Huang[2], Wei Chen[2], Xiaozong Cui[2], Bo Wang[2], Ji Liu[2], Liya Fan[2], Dachuan Qu[2], Zhenyu Hou[2], Tao Guan[2], Chen Li[1], Jingren Zhou[2]

[1]University of California, Irvine [2]Alibaba Group

[1] Irvine, United States [2] Hangzhou, China

[1]{zuozhiw, chenli}@ics.uci.edu, [2]{zengkai.zk, botong.huang, wickeychen.cw, xiaozong.cxz, yanyu.wb, niki.lj, liya.fly, dachuan.qdc, zhenyuhou.hzy, tony.guan, jingren.zhou}@alibaba-inc.com

ABSTRACT

Incremental processing is widely-adopted in many applications, ranging from incremental view maintenance, stream computing, to recently emerging progressive data warehouse and intermittent query processing. Despite many algorithms developed on this topic, none of them can produce an incremental plan that always achieves the best performance, since the optimal plan is data dependent. In this paper, we develop a novel cost-based optimizer framework, called `Tempura`, for optimizing incremental data processing. We propose an incremental query planning model called TIP based on the concept of time-varying relations, which can formally model incremental processing in its most general form. We give a full specification of `Tempura`, which can not only unify various existing techniques to generate an optimal incremental plan, but also allow the developer to add their rewrite rules. We study how to explore the plan space and search for an optimal incremental plan. We evaluate `Tempura` in various incremental processing scenarios to show its effectiveness and efficiency.

Yes, VLDB 2021!
Go to the talk!
0900 Wednesday.

PVLDB Reference Format:
Zuozhi Wang, Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Hou, Tao Guan, Chen Li, Jingren Zhou. Tempura: A General Cost-Based Optimizer Framework for Incremental Data Processing. PVLDB, 14(1): 14-27, 2021.
doi:10.14778/3421424.3421427

the adoption of the incremental processing model. Here are a few examples of emerging applications.

Progressive Data Warehouse [45]. Enterprise data warehouses usually have a large amount of automated routine analysis jobs, which have a stringent schedule and deadline determined by various business logic. For example, at Alibaba, daily report queries are scheduled after 12 am when the previous day's data has been fully collected, and the results must be delivered by 6 am sharp before the bill-settlement time. These routine analysis jobs are predominately handled using batch processing, causing dreadful "rush hour" scheduling patterns. This approach puts pressure on resources during traffic hours, and leaves the resources over-provisioned and wasted during the off-traffic hours. Incremental processing can answer routine analysis jobs progressively as data gets ingested, and its scheduling flexibility can be used to smoothen the resource skew.

Intermittent Query Processing [40]. Many modern applications require querying an incomplete dataset with the remaining data arriving in an intermittent yet predictable way. Intermittent query processing can leverage incremental processing to balance latency for maintaining standing queries and resource consumption by exploiting knowledge of data-arrival patterns. For instance, when querying dirty data, the data is usually first cleaned and then fed into a database. The data cleaning step can quickly spill the clean data but needs to conduct a time-consuming processing on the dirty data. Intermittent query processing can use incremental processing to quickly deliver informative but partial results to the issues.apache.org/jira/browse/CALCITE-4568

Dashboards Projects Issues Boards Create Search

Calcite / CALCITE-4568

Tempura: extending Calcite into an incremental query optimizer

Edit | Comment | Assign | More | Start Progress | Resolve Issue | Close Issue

Details

Type:	New Feature	Status:	OPEN
Priority:	Major	Resolution:	Unresolved
Affects Version/s:	None	Fix Version/s:	None
Component/s:	None		
Labels:	None		

Description

As discussed in the email thread, this is an attempt to extend the Calcite optimizer into a general incremental query optimizer, based on our research paper published in VLDB 2021: Tempura: a general cost-based optimizer framework for incremental data processing

To our best knowledge, this is the first general cost-based incremental optimizer that can find the best plan across multiple families of incremental computing methods, including IVM, Streaming, DBToaster, etc. Experiments (in the paper) shows that the generated best plan is consistently much better than the plans from each individual method alone.

In general, incremental query planning is central to database view maintenance and stream processing systems, and are being adopted in active databases, resumable query execution, approximate query processing, etc. We are hoping that this feature can help widening the spectrum of Calcite, solicit more use cases and adoption of Calcite.

People

Assignee:

Unassigned

Assign to me

Reporter:

Botong Huang

Votes:

0 Vote for this issue

Watchers:

6 Stop watching this issue

Dates

Created:

07/Apr/21 14:35

Updated:

2 days ago

@julianhyde @szampetak

https://calcite.apache.org

Thank you!

A. Appendix

Powered by Calcite

APACHE DRILL

neo4j

APACHE Spark

ORACLE DATABASE

cassandra

elasticsearch

HIVE

hazelcast

Google BigQuery

APACHE HBASE

APACHE LUCENE

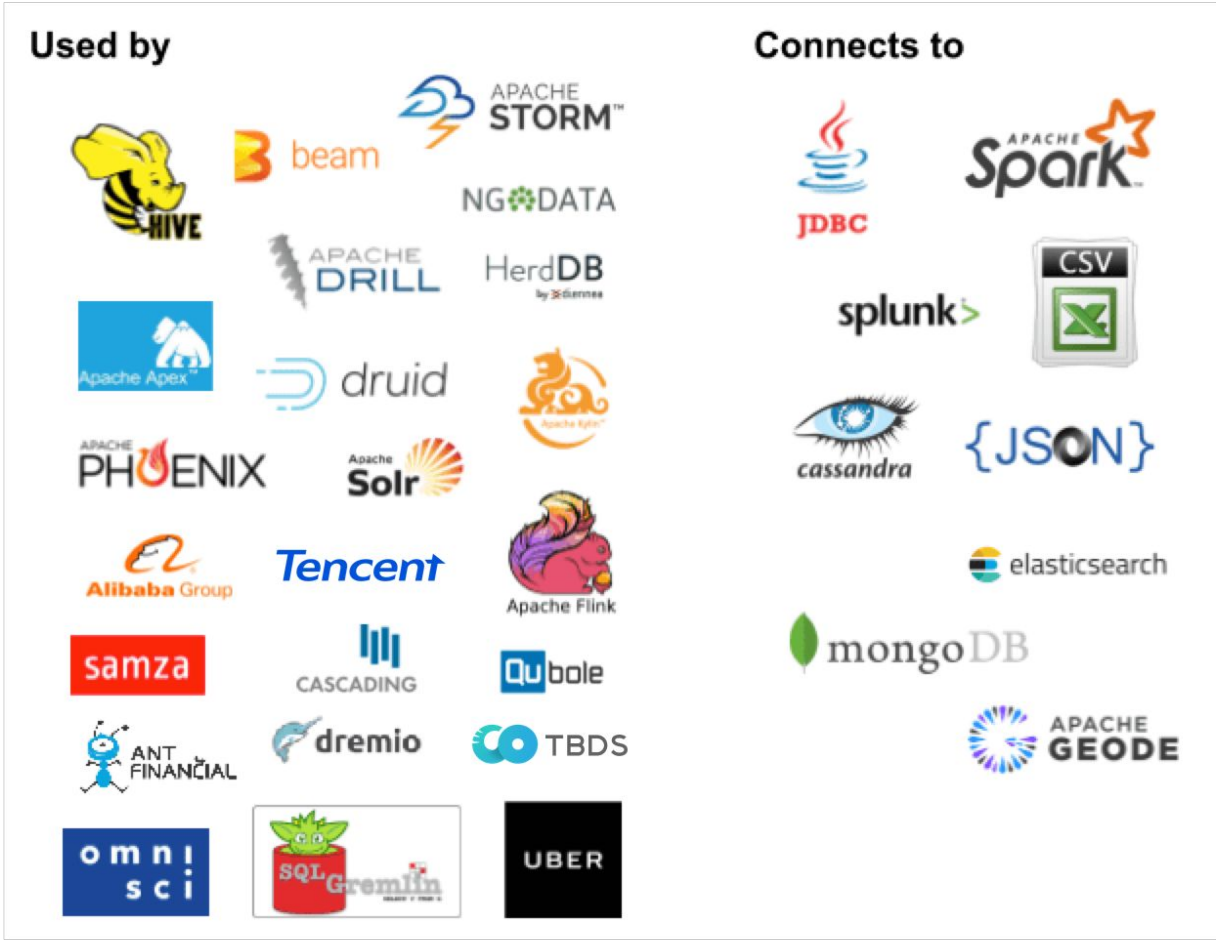
PostgreSQL

redis

Solr

mongoDB

Powered by Calcite



Resources

- Calcite project https://calcite.apache.org
- Materialized view algorithms https://calcite.apache.org/docs/materialized_views.html
- JSON model https://calcite.apache.org/docs/model.html
- Lazy beats smart and fast (DataEng 2018) - MVs, spatial, profiling https://www.slideshare.net/julianhyde/lazy-beats-smart-and-fast
- Efficient spatial queries on vanilla databases (ApacheCon 2018) https://www.slideshare.net/julianhyde/spatial-query-on-vanilla-databases
- Graefe, McKenna. The Volcano Optimizer Generator, 1991
- Graefe. The Cascades Framework for Query Optimization, 1995
- Slideshare (past presentations by Julian Hyde, including several about Apache Calcite) https://www.slideshare.net/julianhyde