optd: A Cascades-Style QO

15-799 Special Topics in Databases: Query Optimization

Sarvesh Tandon, Yuchen Liang

Outline

- Recap
- optd’s Architecture
- On-demand Exploration
- Merging
- Cascading Merge Edge case
- Calcite TPC-H Merging Results
- Domain Specific Language (DSL)
- Real-life Demo

Outline

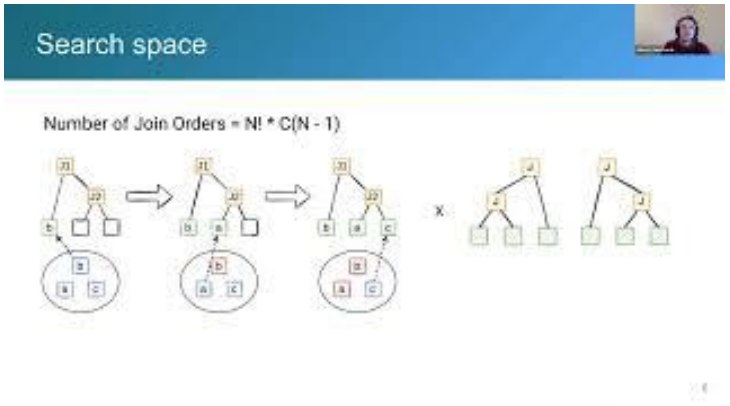
- **Recap**
- optd's Architecture
- On-demand Exploration
- Merging
- Cascading Merge Edge case
- Calcite TPC-H Merging Results
- Domain Specific Language (DSL)
- Real-life Demo

Design Goals

- Plan Quality (Return the fastest plan possible)
- Optimization Latency (Optimize Quickly)
- Plan Quality % Optimization Latency (Best Plan possible in T Time)
- Future proofness for Adaptivity (Support feedback mechanisms for in-situ streaming workloads)
- Explainability (easier to reason about rules and guidance, produce better plans with fewest amount of guidance)

What does Calcite do?

- Breaks enumeration into tasks similar to Cascades
- Lacks guidance mechanism so reduces to Volcano
- No intra-query or inter-query parallelism
- Triggers redundant work when groups are found to be equivalent (group- merging)



Talk by Alexey Goncharuk at Calcite Meetup January 2022
About Join Enumeration and Logical Space Explosion

Problem: Plan Space Explosion

Applying all the transformation and implementation rules in a search space may not give you back a good enough physical plan in timely manner.

There are approaches proven to work well in the wild:

➔ Guidance (except we don’t know how smart it is).
➔ Multi-stage optimization: QuickPlan and Heuristics first.
➔ BlackBox operator (MultiJoin), stratified optimization within the operator.

Our Approach: Can **a slightly different enumeration approach** help?

Yes! But to show you how, let’s first look at the design!

Outline

- Recap
- **optd's Architecture**
- On-demand Exploration
- Merging
- Cascading Merge Edge case
- Calcite TPC-H Merging Results
- Domain Specific Language (DSL)
- Real-life Demo

Architecture

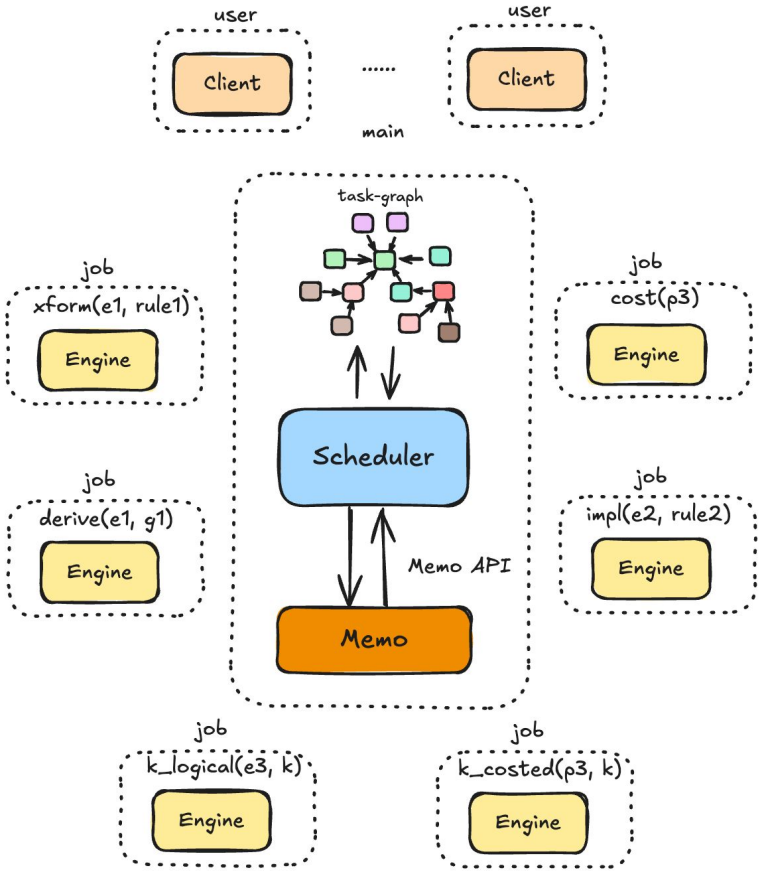
Event loop architecture; communication done through message passing.

Modeling optimization objectives as tasks

A task has other dependent subtasks.

A task can schedule discrete unit of work (job) to be executed on other coroutines.

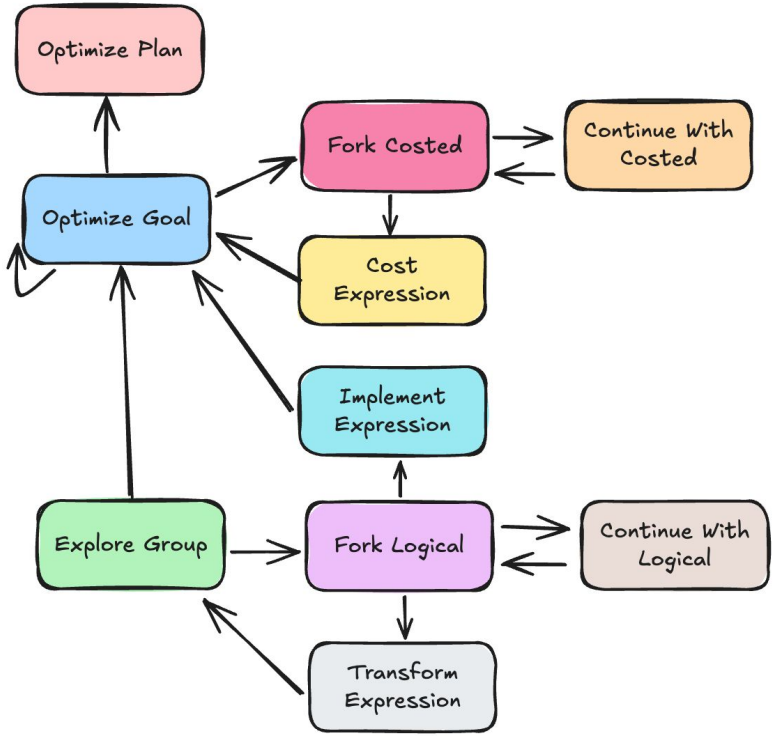
An in-memory (possibly, on disk in future) memo table



Task Graph

More tasks comparing to Cascades:

- ➔ Needed for flexible scheduling, as we might revisit a task later.
- ➔ Cascades do not need it because children goals are fully optimized before we optimize an expression that depends on them
- ➔ LIFO: Cascades

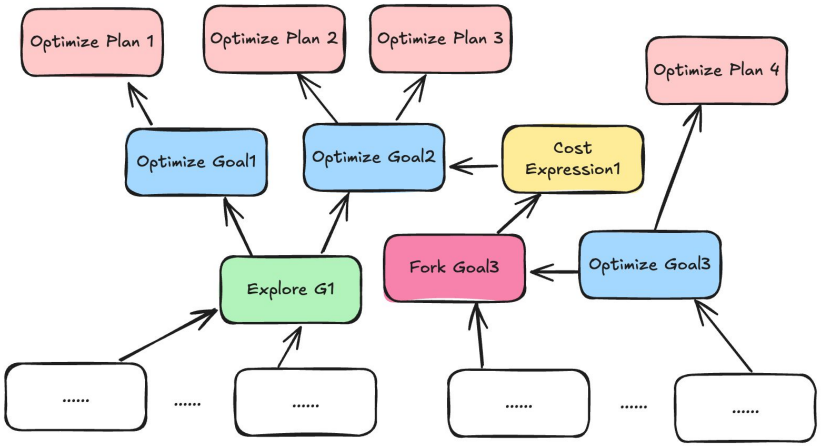


Inter-query Parallelism

Allows multiple query to share optimization results for subexpressions, not just the root plan.

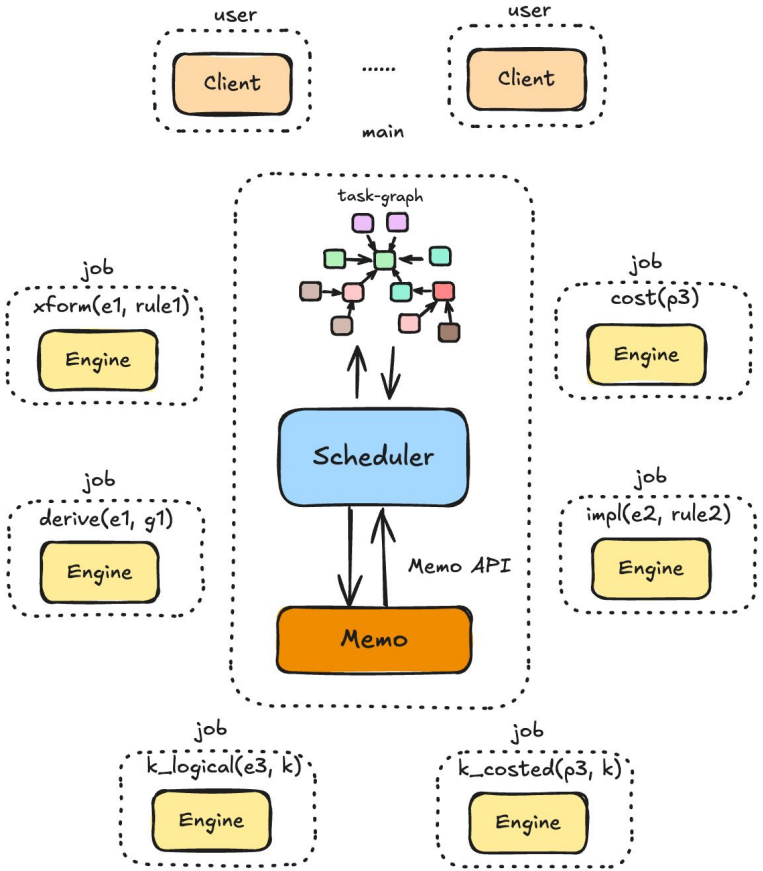
OptimizePlan:
Optimize for a specific query instance.

Query cache should be done at expression level.



Intra-query Parallelism

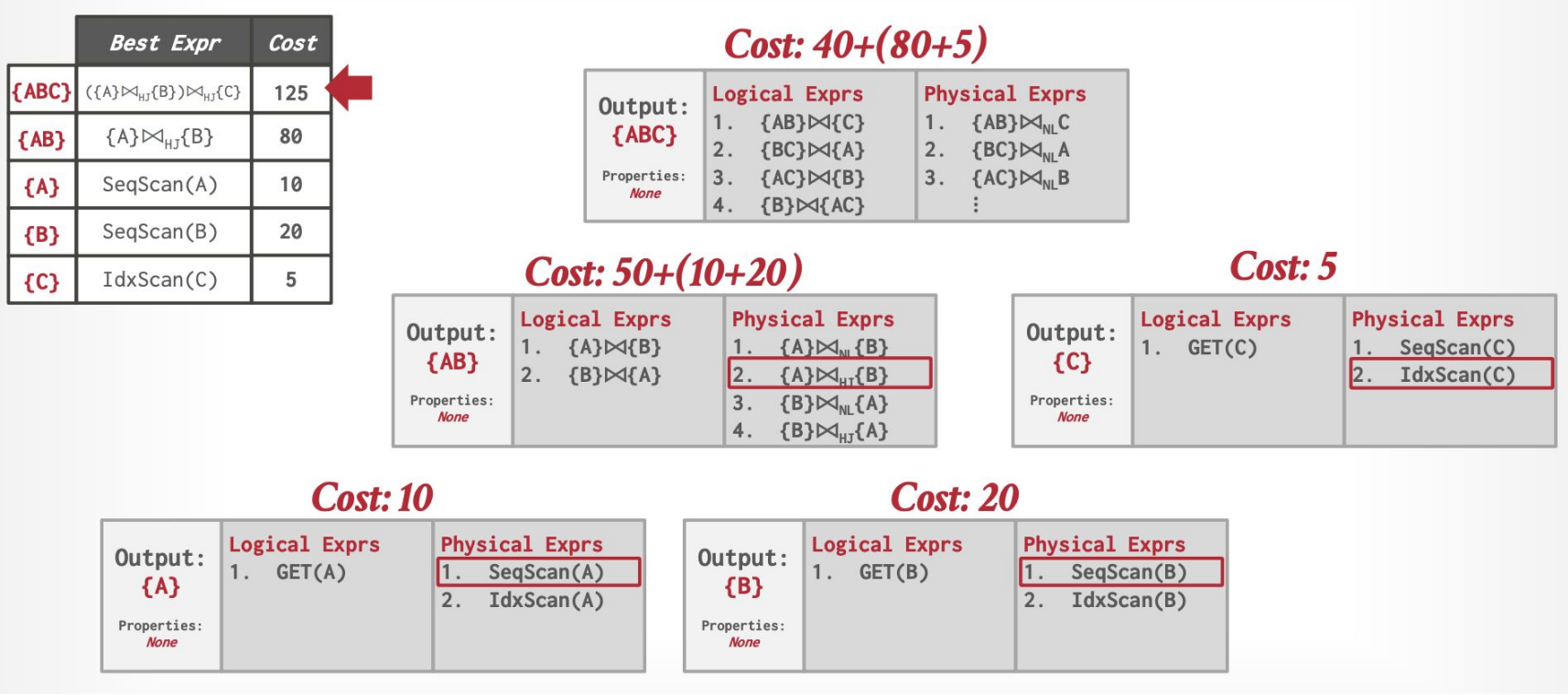
The rules and costing procedure for optimizing a single query can run in parallel in other coroutines.



Outline

- Recap
- optd's Architecture
- **On-demand Exploration**
- Merging
- Cascading Merge Edge case
- Calcite TPC-H Merging Results
- Domain Specific Language (DSL)
- Real-life Demo

On-demand Explored Cascades



Source: Andy Pavlo

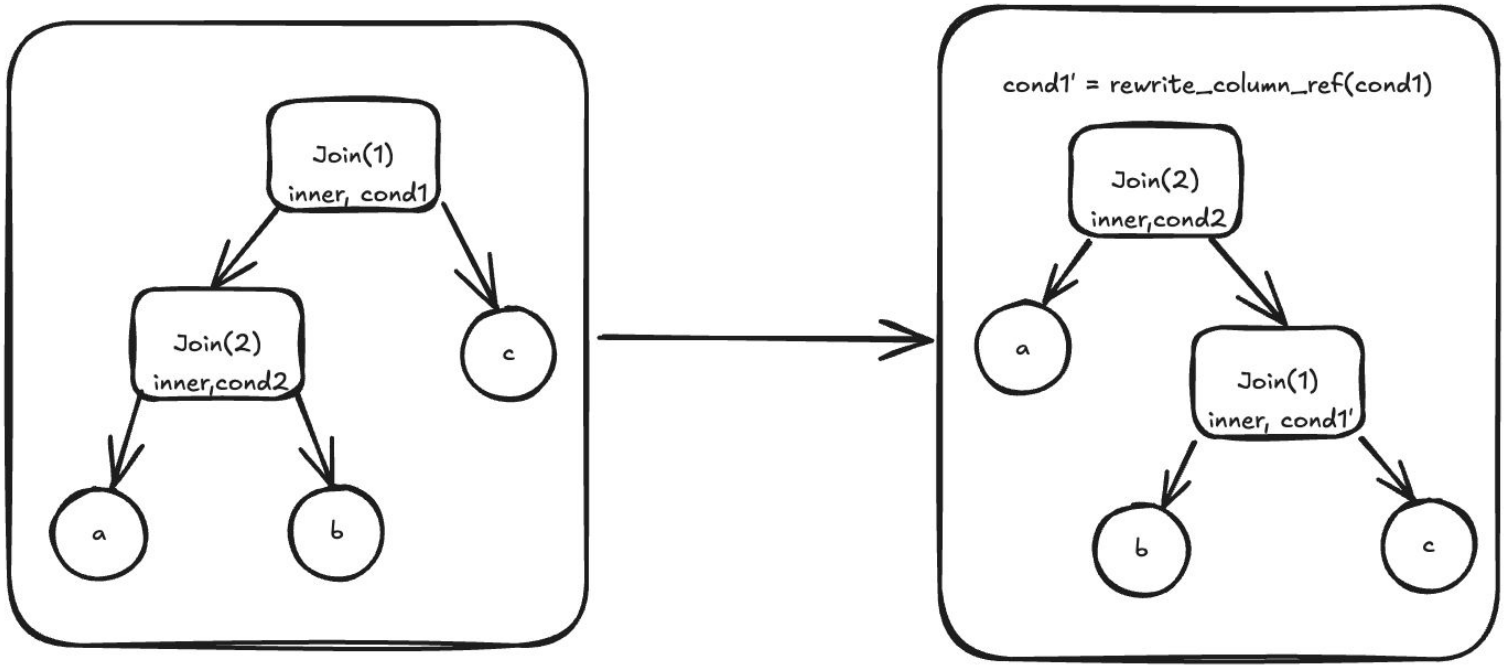
Why is On-demand Exploration Good?

Only expand the search space if the rule specifies.

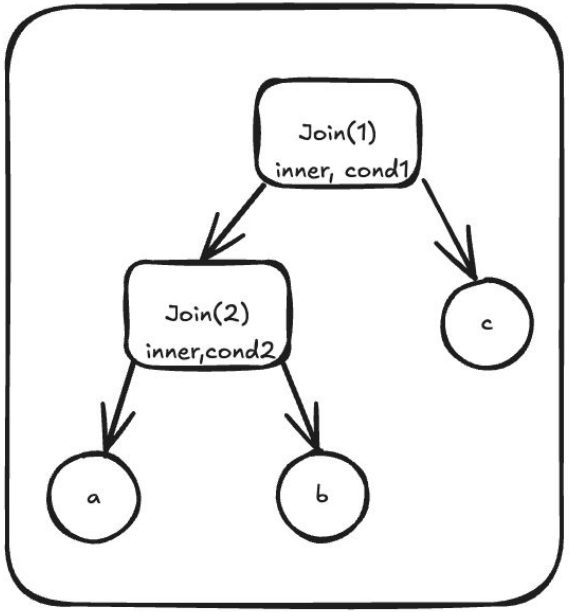
Search is directed by rules.

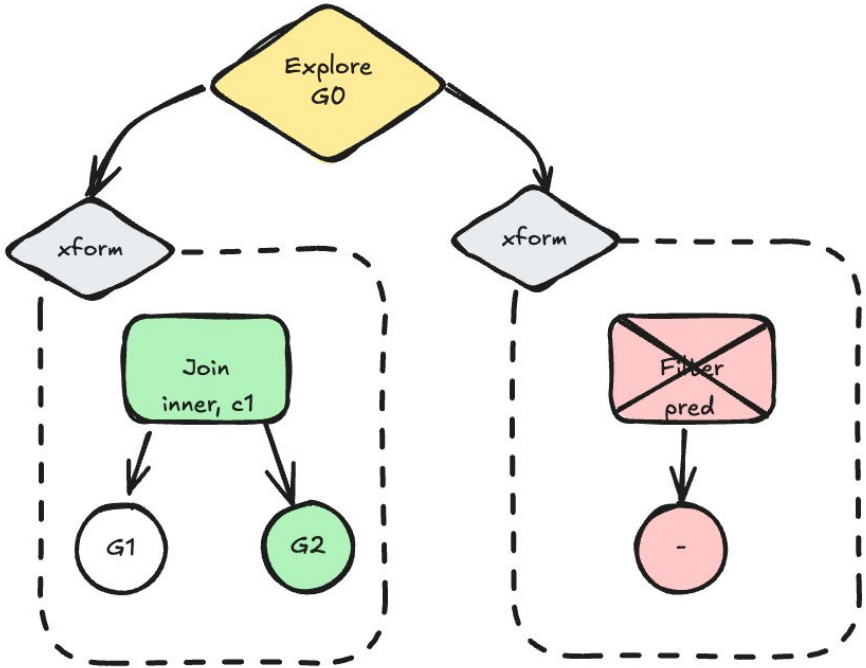
Example: JoinAssocRule

JoinAssocRule

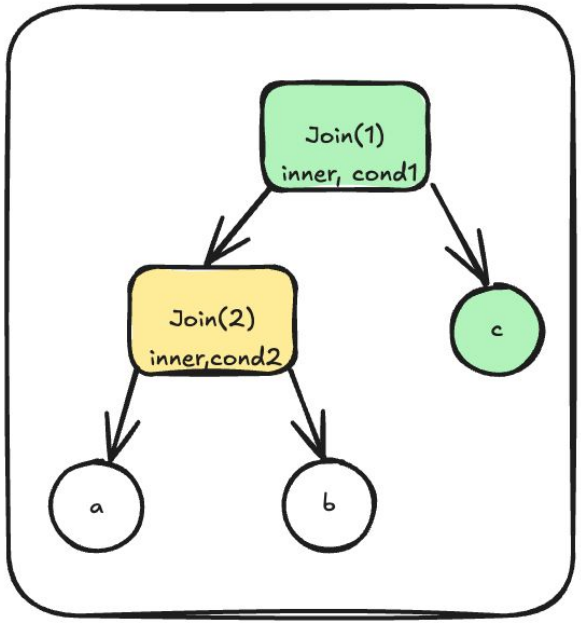


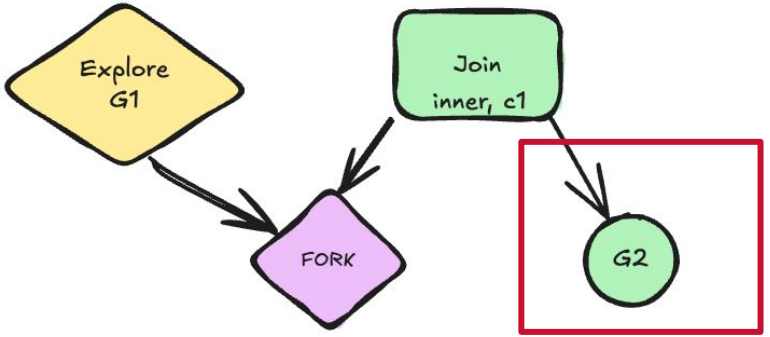
Matching Top Node Tag





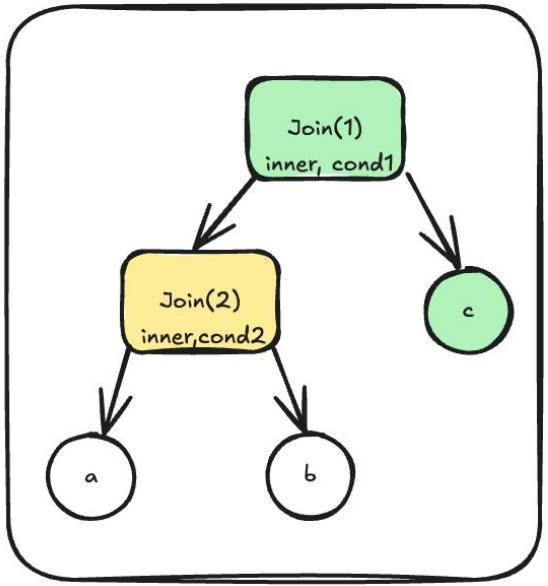
On-demand Exploration

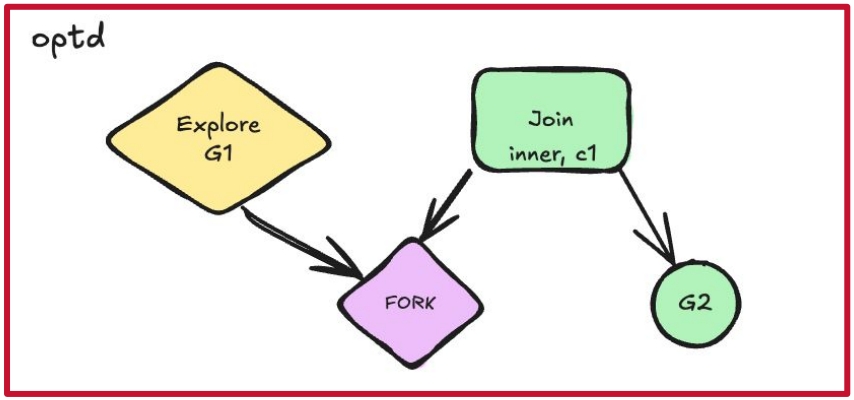


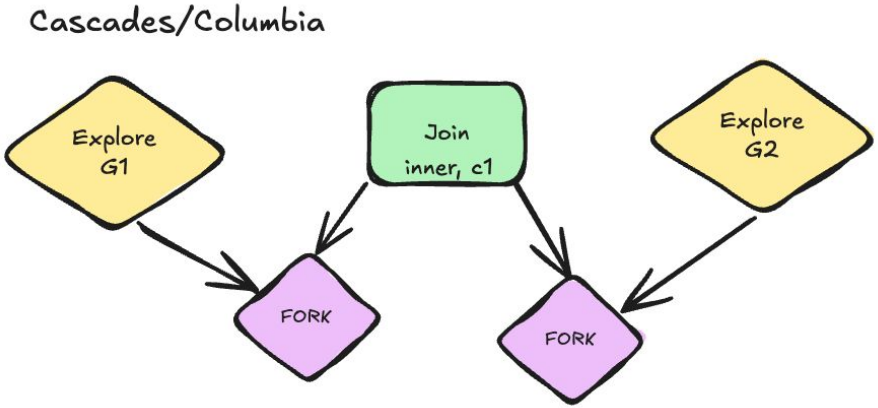


We do not expand **G2**!

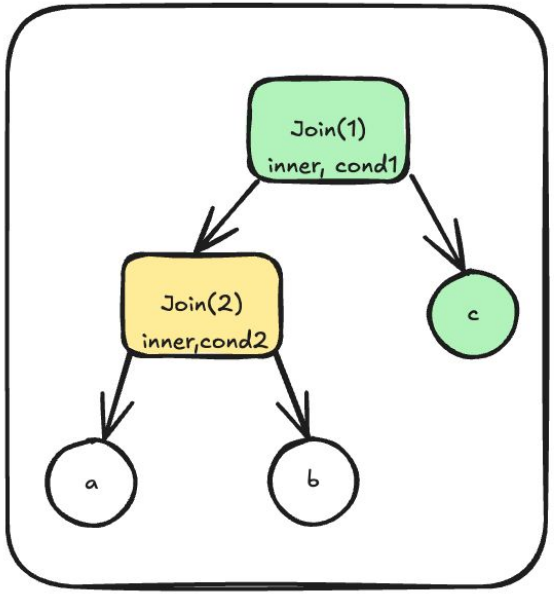
Comparison with Cascades/Columbia

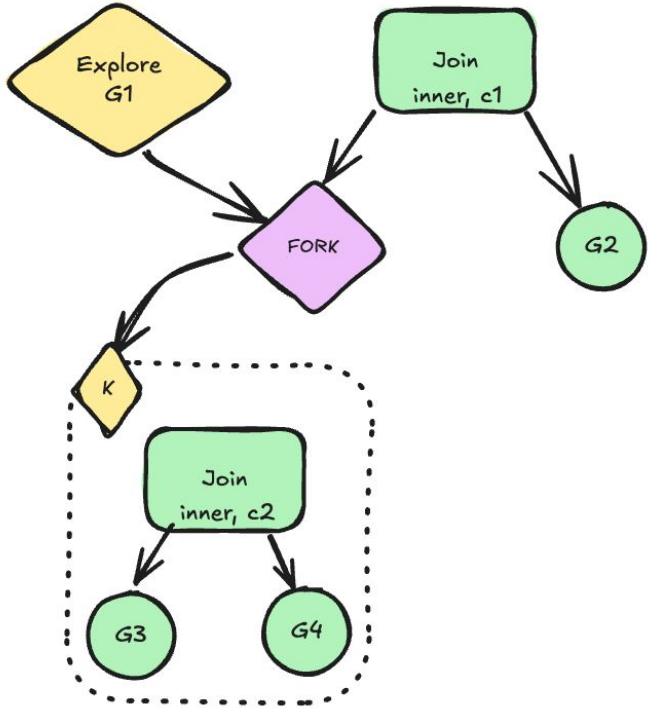






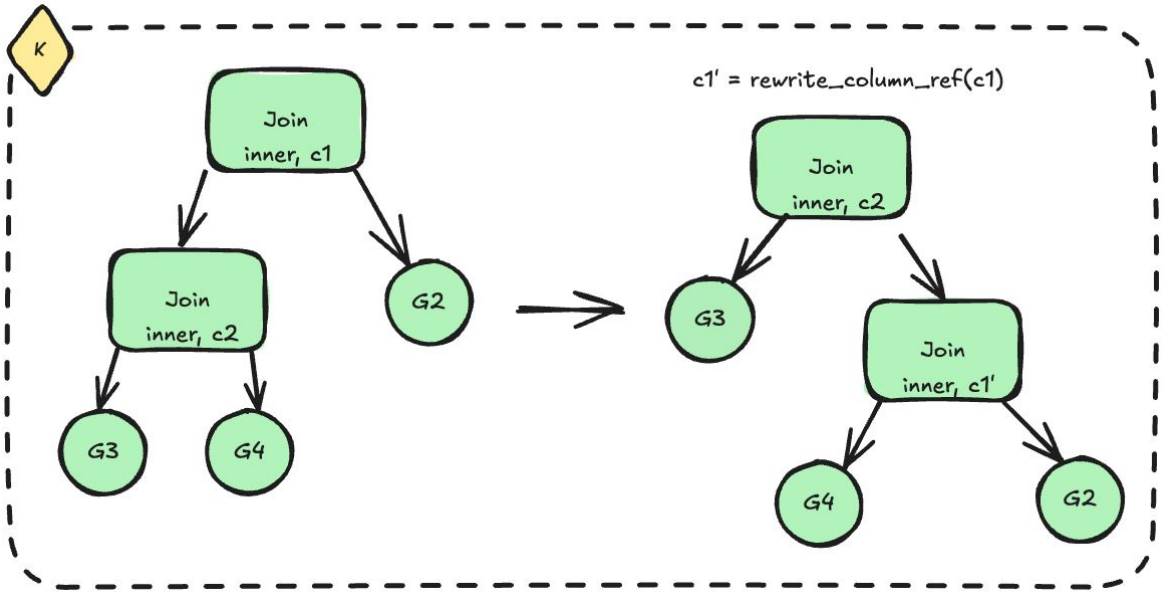
Forking Execution (CPS)





The rule engine returns a callback when it sees a **group** and resume execution on a logical expression.

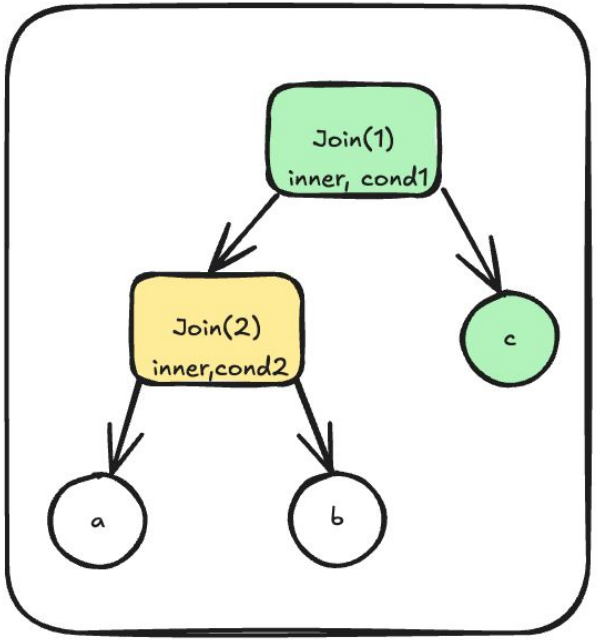
Generate Equivalent Partial Plan

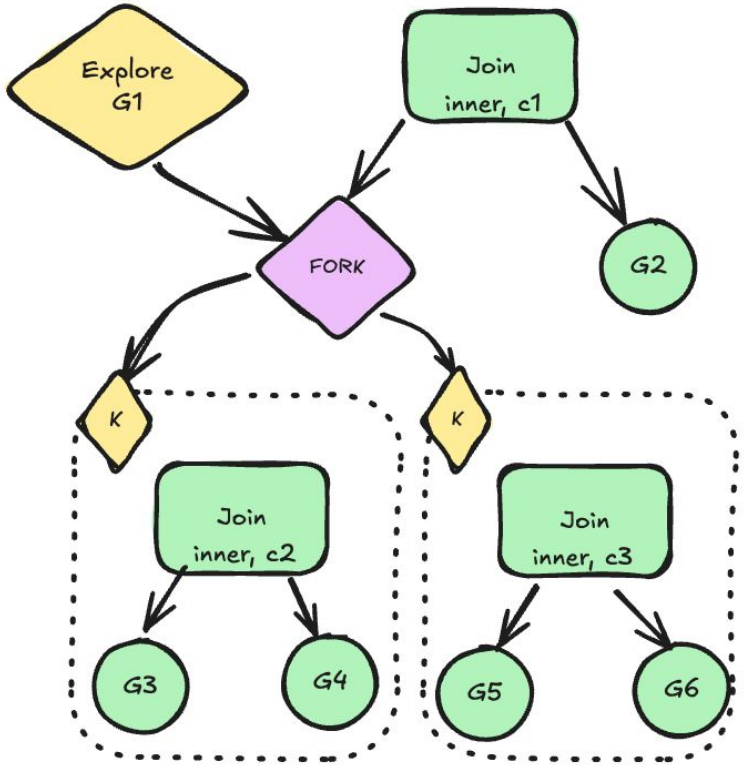


job -> main:
New Partial Logical Plan!

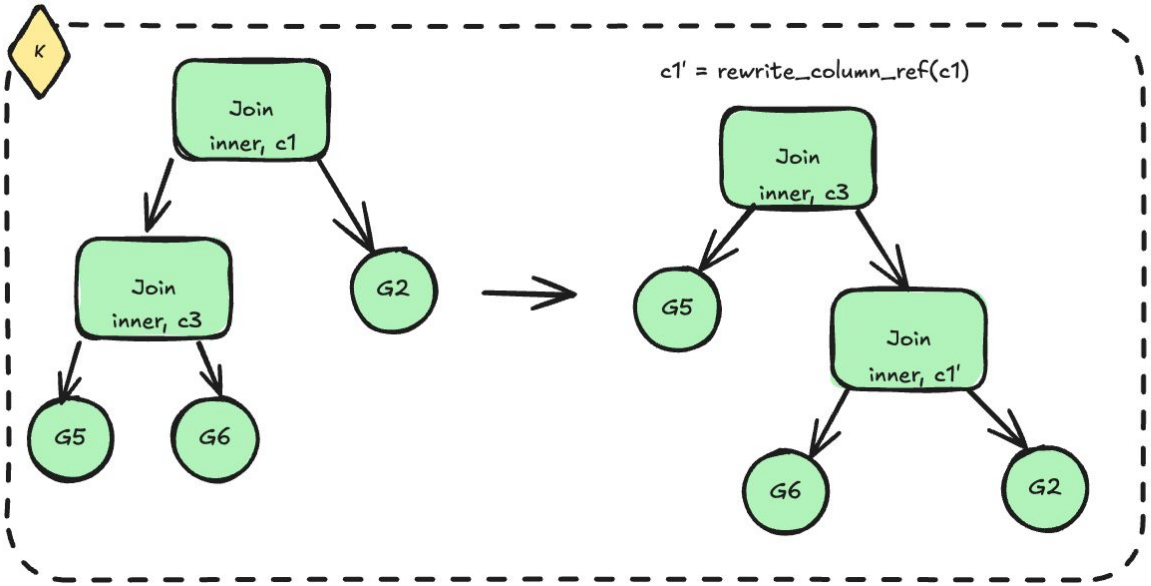
main:
- ingest the logical plan into Memo.
- create more tasks

Update on a Dependent Group, Matches





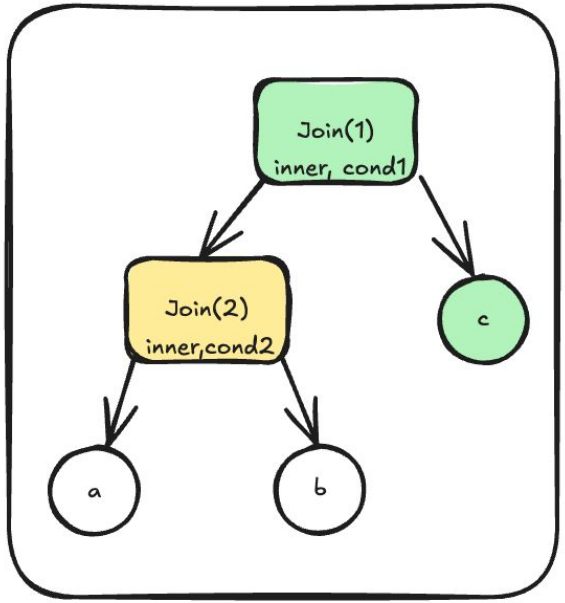
Generate an Equivalent Partial Plan

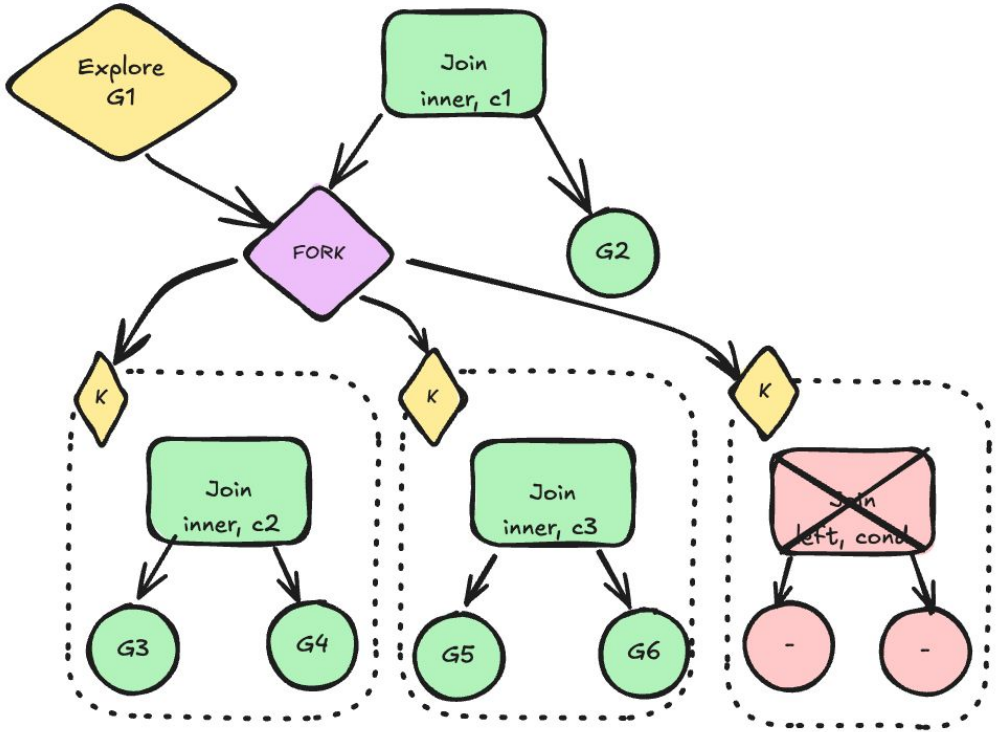


job -> main:
New Partial Logical Plan!

main:
- ingest the logical plan into Memo.
- create more tasks

Update on a Dependent Group, Does not Match





Outline

- Recap
- optd's Architecture
- On-demand Exploration
- **Merging**
- Cascading Merge Edge case
- Calcite TPC-H Merging Results
- Domain Specific Language (DSL)
- Real-life Demo

Simple Merge From the Memo table's perspective

Group ID	Expressions
4	Join (Group 1, Group 2)
3	Join (Group 2, Group 1)
2	Scan (Table A)
1	Scan(Table B)

We explore group 3 by applying Join Commute Rule to the epxression

Simple Merge From the Memo table's perspective

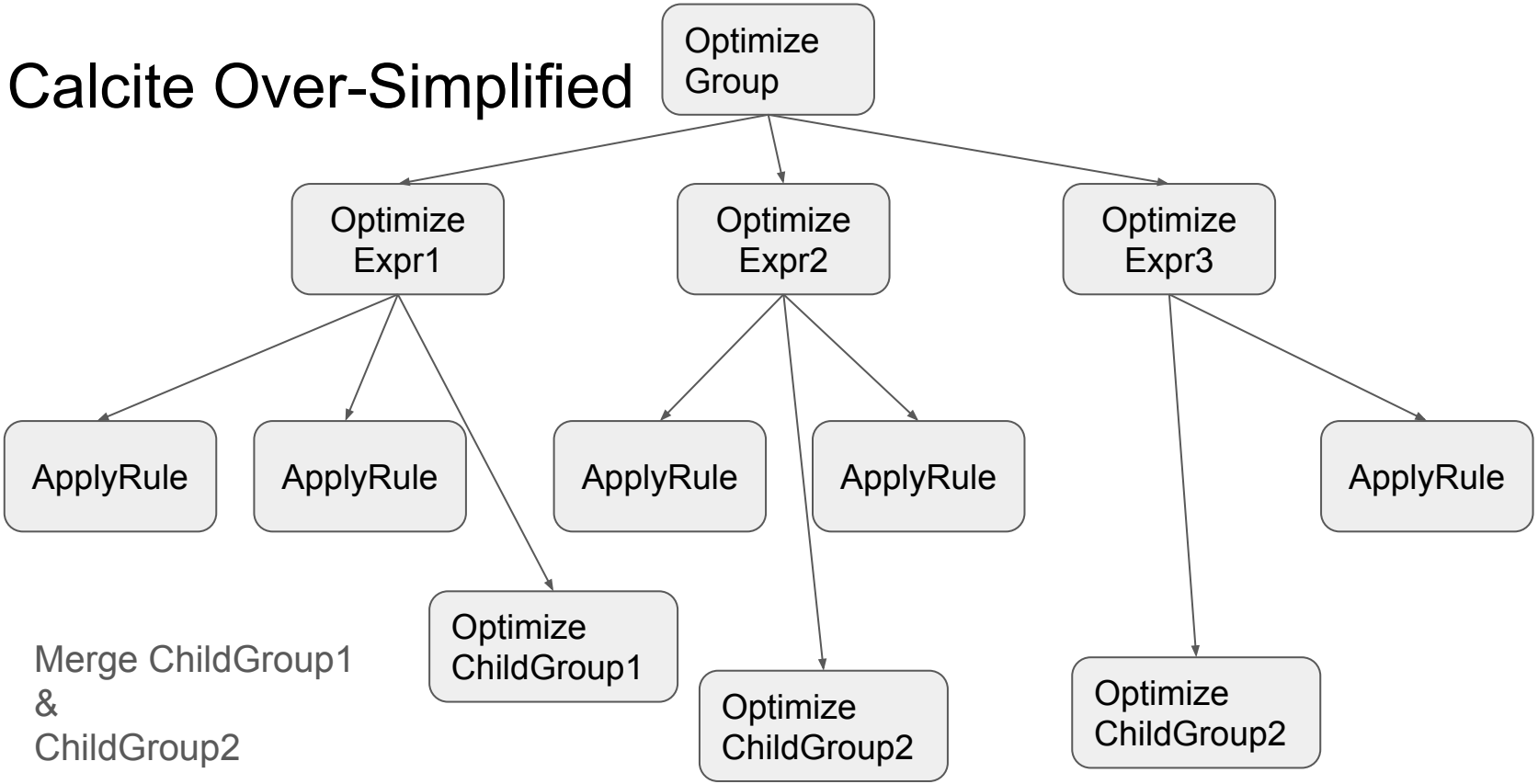
Group ID	Expressions
4	Join (Group 1, Group 2), Join (Group 2, Group 1)
3	Join (Group 2, Group 1)
2	Scan (Table A)
1	Scan(Table B)

We produce a new expression, that already exists in another group. Hence, now we know that group 3 and group 4 are equivalent and need to be merged.

What does Calcite do?

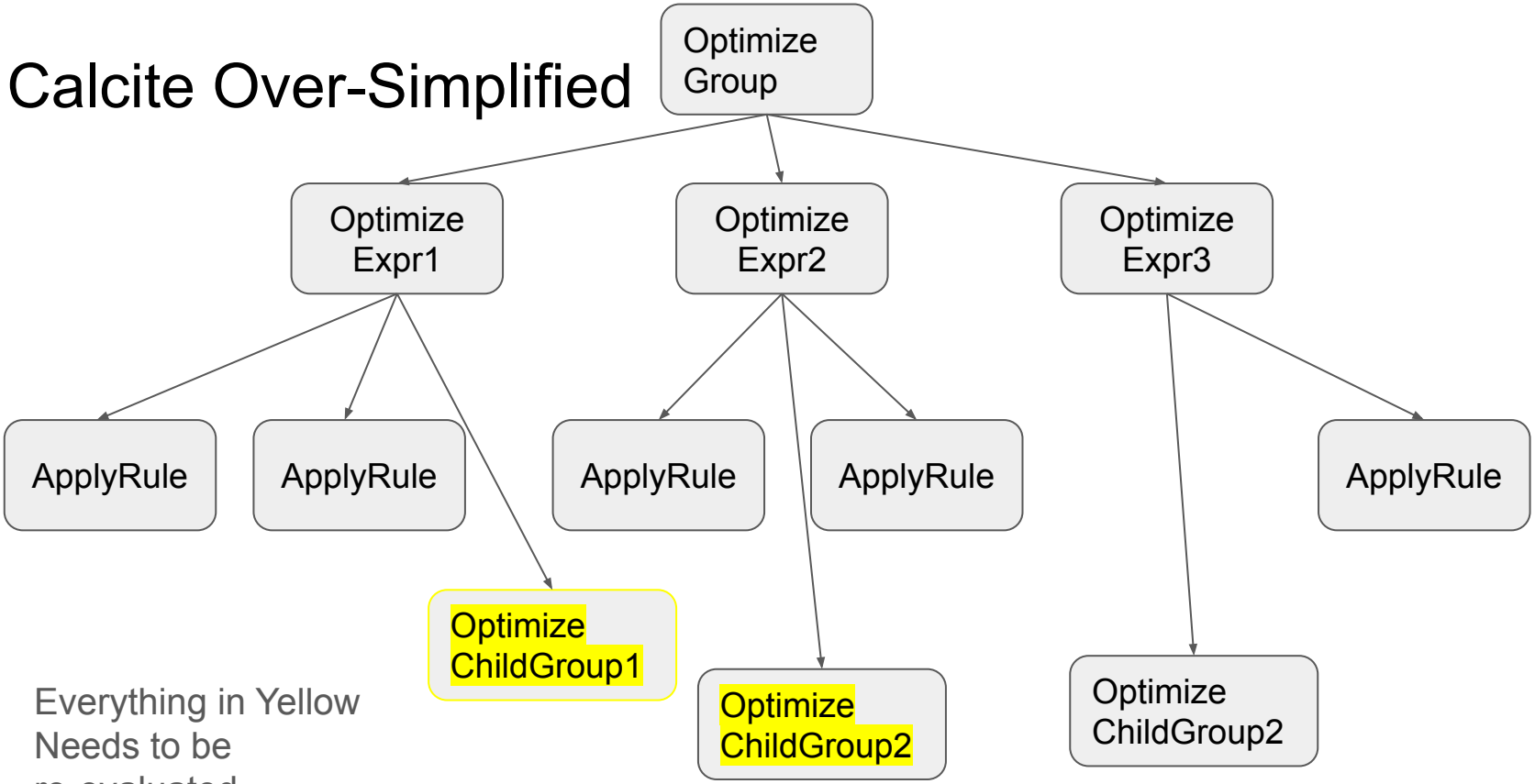
- Calcite also has a tasks-like structure where Optimize Group tasks launch Optimize Expression tasks which then launch Optimize Group task, and so on.
- When two RelSet (groups) are found to contain the same RelNode (logical expression), merging is triggered
- On merge, Calcite recursively marks everything that depends on the RelSet as unexplored and unoptimized, all the way up to the top Optimize Group task.
- It then relaunches a bunch of Apply Rule Tasks that may not need to be reevaluated.
- If the merge happens too low in the plan tree, then a lot of redundant work will be done.

Calcite Over-Simplified



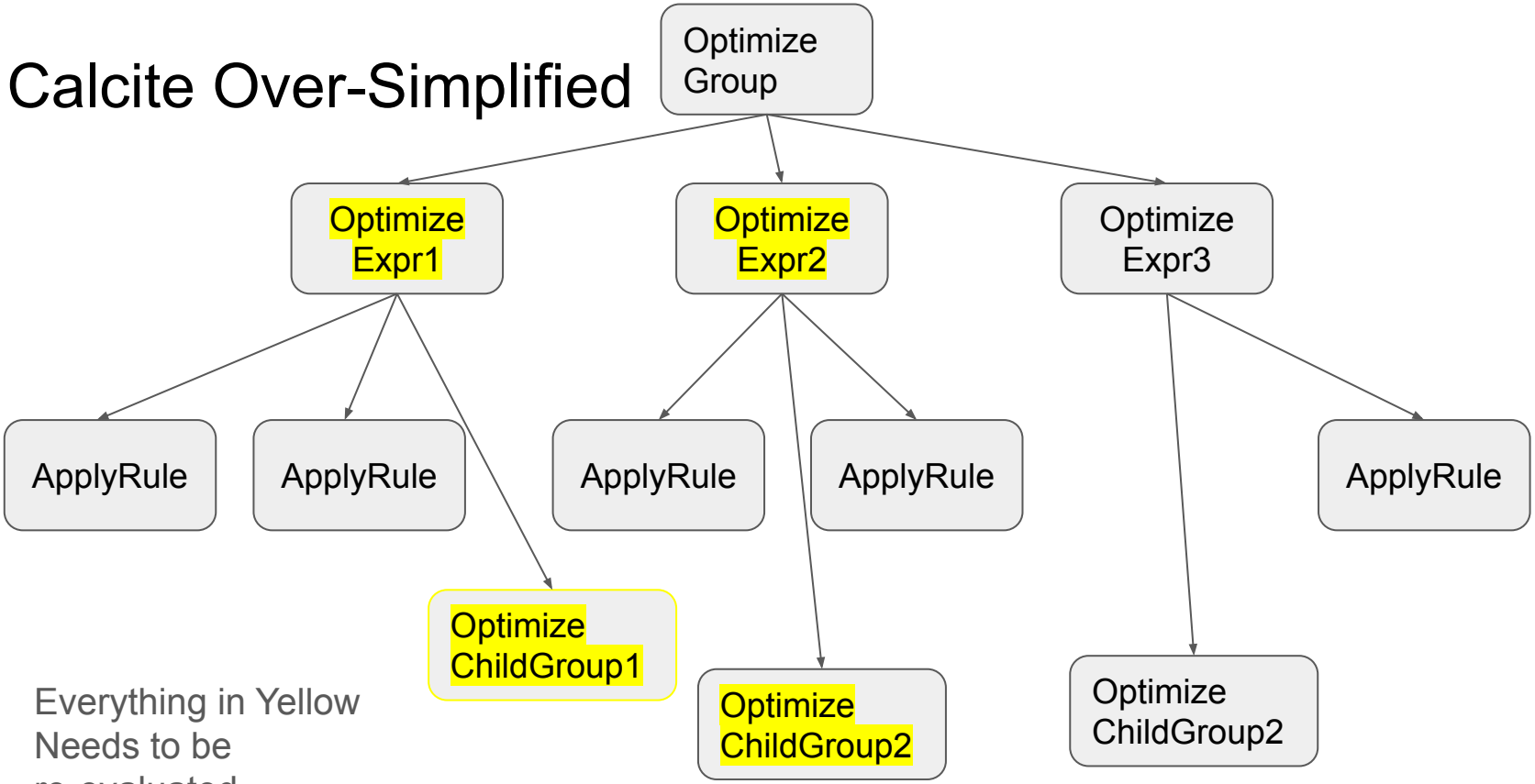
Merge ChildGroup1
&
ChildGroup2

Calcite Over-Simplified



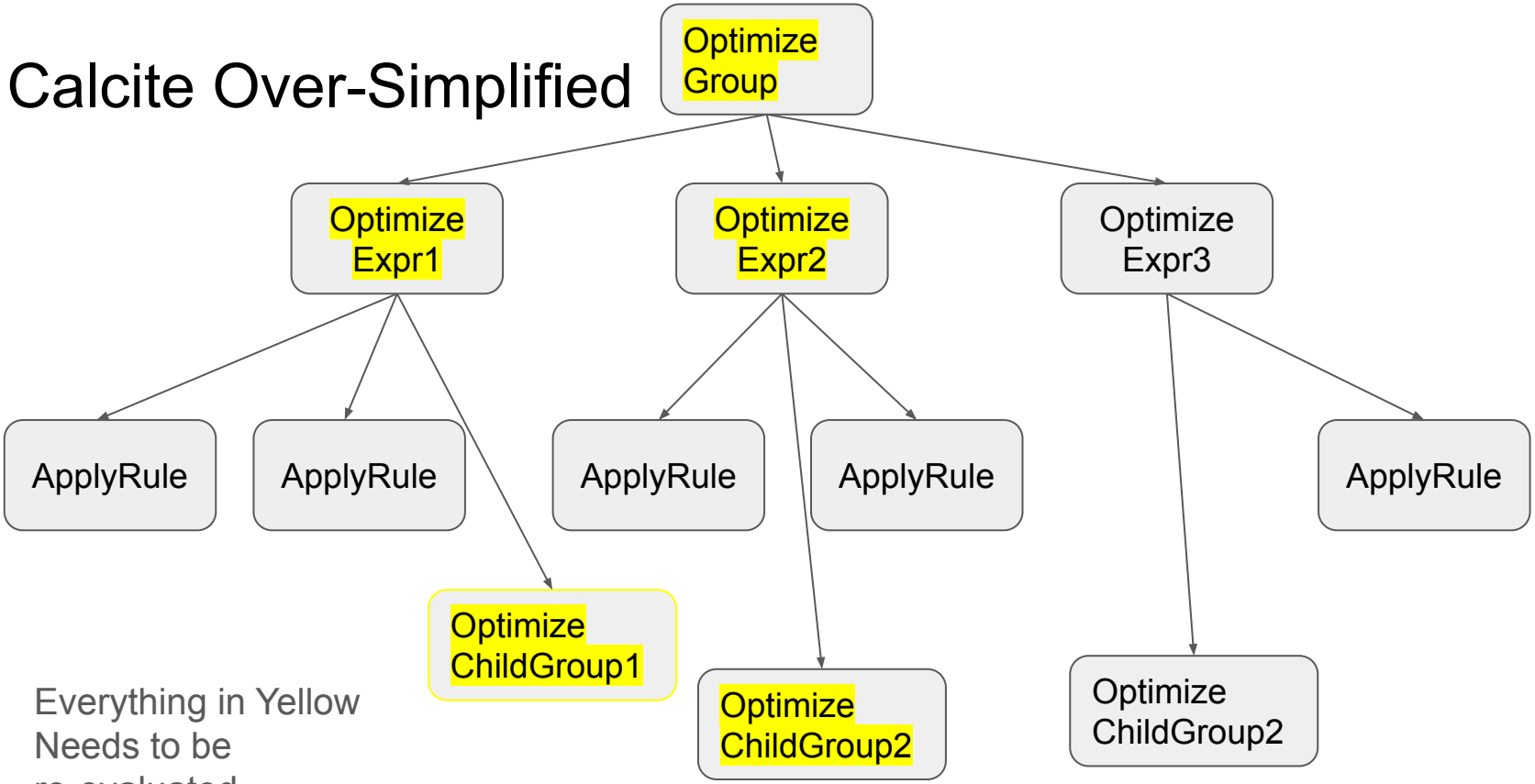
Everything in Yellow Needs to be re-evaluated

Calcite Over-Simplified



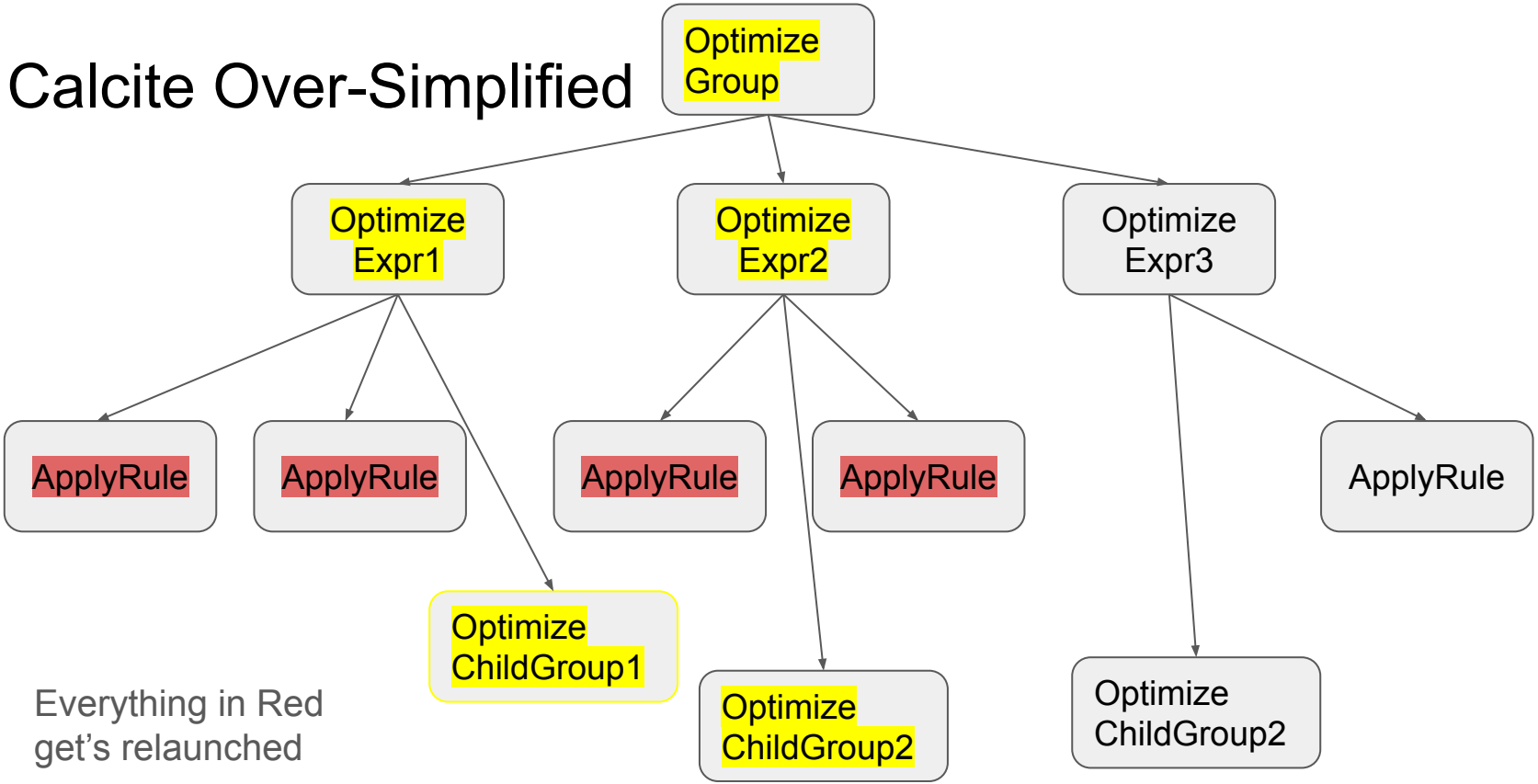
Everything in Yellow Needs to be re-evaluated

Calcite Over-Simplified



Everything in Yellow Needs to be re-evaluated

Calcite Over-Simplified



Everything in Red get's relaunched

optd Merging

- Due to On-demand exploration, we will have merging occur much more frequently; but on-demand exploration is worth doing for explainability and efficiency reasons.
- We do no relaunch anything that was marked as red unless needed.
- Furthermore, we only propagate stuff upwards when necessary

Simple Merge From the Memo table's perspective

Group ID	Expressions
4	Join (Group 1, Group 2)
3	Join (Group 2, Group 1)
2	Scan (Table A)
1	Scan(Table B)

We explore group 3 by applying Join Commute Rule to the epxression

Simple Merge From the Memo table's perspective

Group ID	Expressions
4	Join (Group 1, Group 2), Join (Group 2, Group 1)
3	Join (Group 2, Group 1)
2	Scan (Table A)
1	Scan(Table B)

We produce a new expression, that already exists in another group. Hence, now we know that group 3 and group 4 are equivalent and need to be merged.

So, how do we do this?

So, how do we do this?
We use the Union-find Algorithm

A more realistic example…

Group ID	Representative Group ID	Expression
372	372	Sort(Group 356)
…		…
4	4	Join (Group 1, Group 2, cond=(left.col1==right.col2 and right.col3 == left.col6))
3	3	Join (Group 2, Group 1, cond=(right.col1==left.col2 and left.col3 == right.col6)), Join (Group 2, Group 1, cond=(left.col3 == right.col6 and right.col1==left.col2))
2	2	Scan (Table A)
1	1	Scan (Table B)

Left and Right indicate the left group and the right group. The conditions in group 3 are equivalent but may be different in terms of their performance because of the cost of evaluating the condition for each tuple.

A more realistic example…

Group ID	Representative Group ID	Expression
372	372	Sort(Group 356)
…		…
4	4	Join (Group 1, Group 2, cond=C1)
3	3	Join (Group 2, Group 1, cond=C2), Join (Group 2, Group 1, cond=C3)
2	2	Scan (Table A)
1	1	Scan (Table B)

By default each group's representative group is itself

Apply rule…

Group ID	Representative Group ID	Expression
372	372	Sort(Group 356)
…		…
4	4	Join (Group 1, Group 2, cond=C1), Join (Group 2, Group 1, cond=C2)
3	3	Join (Group 2, Group 1, cond=C2), Join (Group 2, Group 1, cond=C3)
2	2	Scan (Table A)
1	1	Scan (Table B)

Now we do merge…

Group ID	Representative Group ID	Expression
372	372	Sort(Group 356)
…		…
4	4	Join (Group 1, Group 2, cond=C1), Join (Group 2, Group 1, cond=C2) , Join (Group 2, Group 1, cond=C3)
3	4	Join (Group 2, Group 1, cond=C2), Join (Group 2, Group 1, cond=C3)
2	2	Scan (Table A)
1	1	Scan (Table B)

Copy all expression from non-repr group to repr group

Outline

- Recap
- optd's Architecture
- On-demand Exploration
- Merging
- **Cascading Merge Edge case**
- Calcite TPC-H Merging Results
- Domain Specific Language (DSL)
- Real-life Demo

Let’s go back to simple merge

14	Join (Group 4, Group 7) …
11	Join (Group 3, Group 7) …
…	…
4	Join (Group 1, Group 2)
3	Join (Group 2, Group 1)
2	Scan (Table A)
1	Scan(Table B)

Add a new expression

14	Join (Group 4, Group 7) …
11	Join (Group 3, Group 7) …
…	…
4	Join (Group 1, Group 2), Join (Group 2, Group 1)
3	Join (Group 2, Group 1)
2	Scan (Table A)
1	Scan(Table B)

Group 3 and Group 4 merge and become equivalent

14	Join (Group 4, Group 7) …
11	Join (Group 3, Group 7) …
…	…
4	Join (Group 1, Group 2), Join (Group 2, Group 1)
3	Join (Group 2, Group 1)
2	Scan (Table A)
1	Scan(Table B)

So, these two expressions are now equivalent

14	Join (Group 4, Group 7) …
11	Join (Group 3, Group 7) …
…	…
4	Join (Group 1, Group 2), Join (Group 2, Group 1)
3	Join (Group 2, Group 1)
2	Scan (Table A)
1	Scan(Table B)

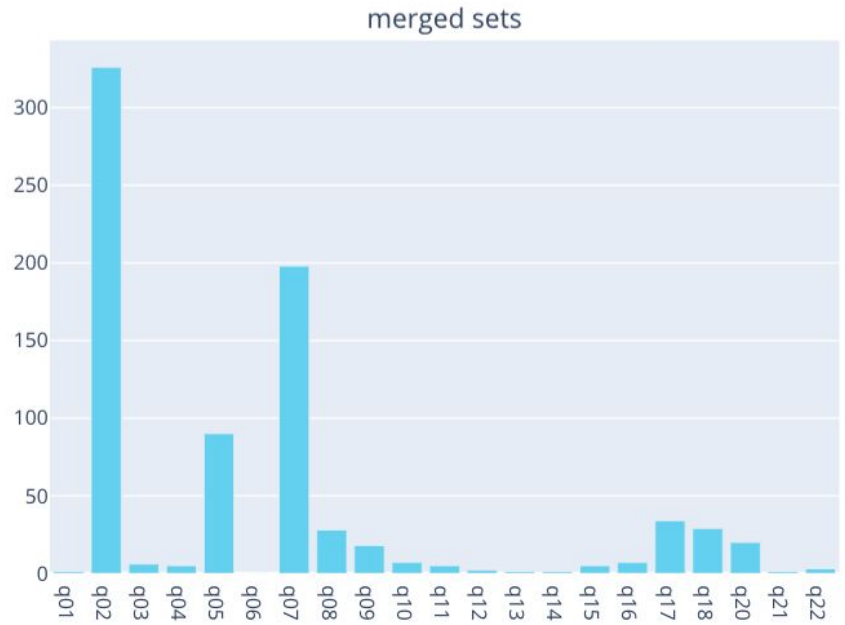
So, now group 14 and group 11 are equivalent
And must now be merged

14	Join (Group 4, Group 7) …
11	Join (Group 3, Group 7) …
…	…
4	Join (Group 1, Group 2), Join (Group 2, Group 1)
3	Join (Group 2, Group 1)
2	Scan (Table A)
1	Scan(Table B)

Outline

- Recap
- optd's Architecture
- On-demand Exploration
- Merging
- Cascading Merge Edge case
- **Calcite TPC-H Merging Results**
- Domain Specific Language (DSL)
- Real-life Demo

Merging Statistics from Calcite TPC-H



optd DSL

Designed by the optd team

Parser, Lexer, Analyzer and Rule Engine was Built-by Alexis Schlomer

Supports on-demand exploration

Turing complete

Domain Specific Language

```
data LogicalProperties
data PhysicalProperties
data Physical =
    | PAdd(left: Physical, right: Physical)
    | PMult(left: Physical, right: Physical)
  \ PConst(val: I64)

data Logical =
    | Add(left: Logical, right: Logical)
    | Mult(left: Logical, right: Logical)
  \ Const(val: I64)

[transformation]
fn (op: Logical) mult_commute() = match op
    | Mult(left, right) -> Mult(right, left)
  \ _ -> none

[transformation]
fn (op: Logical) add_to_mult() = match op
    | Add(Const(left), Const(right)) -> if left == right then Mult(Const(2), Const(left)) else none
  \ _ -> none

[implementation]
fn (op: Logical) convert(props: PhysicalProperties?) = match op
    | Add(left, right) -> PAdd(left.convert(none), right.convert(none))
    | Mult(left, right) -> PMult(left.convert(none), right.convert(none))
  \ Const(val) -> PConst(val)

fn (op: Physical) cost(): F64 =
  match op
    | PAdd(left, right) -> 2.0 + left.cost() + right.cost()
    | PMult(left, right) -> 1.0 + left.cost() + right.cost()
    \ PConst(_) -> 0.0
```

Logical Operators

```
// ------------------------
// 1. Logical Operators
// ------------------------

// Logical operators form the foundation of the language and represent relational algebra
// operations. They are declared using the following syntax:

data Logical =
    | Join(left: Logical, right: Logical, type: JoinType, predicate: Scalar)
    | Filter(child: Logical, predicate: Scalar)
    | Project(child: Logical, expressions: [Scalar])
    | Sort(child: Logical, order_by: [Bool]) // true is ascending
    \ Get(table_name: String)
```

Physical Operators

```
data Physical =
    | Scan(table_name: String)
    | PhysFilter(child: Physical, predicate: Scalar)
    | PhysProject(child: Physical, expressions: [Scalar])
    | PhysSort(child: Physical, order_by: [Bool]) // true is ascending
    \ PhysJoin =
        | HashJoin(
                build_side: Physical,
                probe_side: Physical,
                type: JoinType,
                predicate: Scalar
            )
        | MergeJoin(
                left: Physical,
                right: Physical,
                type: JoinType,
                predicate: Scalar
            )
        \ NestedLoopJoin(
                outer: Physical,
                inner: Physical,
                type: JoinType,
                predicate: Scalar
            )
```

Special Types

```
data Catalog
data LogicalProperties(schema: Schema)
data PhysicalProperties(order_by: [Bool])  // e.g., define sorting, partitioning, etc.
data CostedProperties    // e.g., define cost model, statistics, etc.
data Statistics          // e.g., define histograms, MCVs, etc.

data Schema(columns: [Column])
data Column(name: String, data_type: String, is_nullable: Bool)
```

Outline

- Recap
- optd’s Architecture
- On-demand Exploration
- Merging
- Cascading Merge Edge case
- Calcite TPC-H Merging Results
- Domain Specific Language (DSL)
- **Real-life Demo**

Real-Life Demo

Now we will show a simple DSL compiling and the optimizer optimizing a simple plan.

Future Work: Query Optimization as a Service (QOaaS)

- Make memo table persistent in order to share optimization across queries and decrease the optimization latency.
- Allow for conversions to/from for databases other than Datafusion.
- Setup infrastructure to use the feedback mechanism

Our accomplishments

- Built a working Query Optimizer that
  - Requires less guidance than other approaches to achieve the benefits of Cascades
  - Avoids the inefficiencies found in Calcite (redundant work for merging)
  - Supports intra-query and inter-query parallelism
  - Can be extended to support more operators or rules using DSL
  - Can easily be extended to support feedback based adaptivity - using new statistics in the middle of optimization
  - Can run scalar optimizations workloads

Lessons

- Equivalence expression management is needed.
- Choosing a plan representation wisely.

Cascading Merging in TPC-H Q3