Querify Labs

Building Cost-Based Query Optimizers with Apache Calcite

Vladimir Ozerov
Querify Labs, CEO

SQL use cases: technology

- “Old-school” databases (MySQL, Postgres, SQL Server, Oracle)

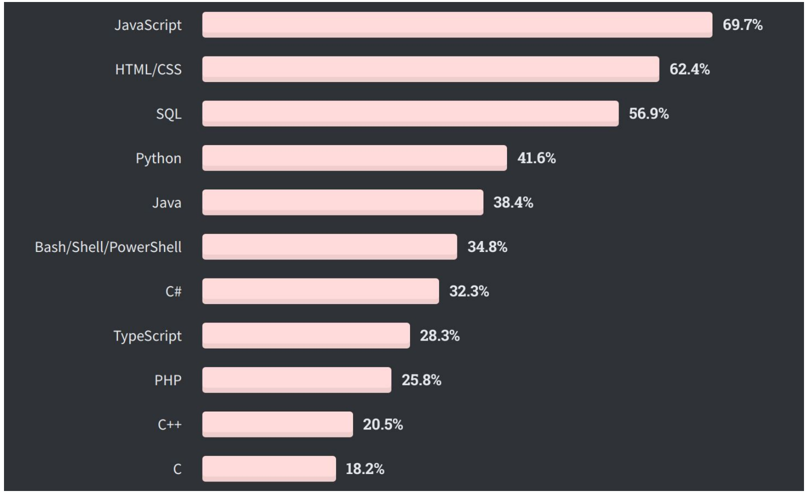
SQL use cases: technology

- “Old-school” databases (MySQL, Postgres, SQL Server, Oracle)
- “New” products
  - Relational (CockroachDB, TiDB, YugaByte)
  - BigData/Analytics (Hive, Snowflake, Dremio, Clickhouse, Presto)
  - NoSQL (DataStax*, Couchbase*)
  - Compute/streaming (Spark, ksqlDB, Apache Flink)
  - In-memory (Apache Ignite, Hazelcast, Gigaspaces)
- Rebels:
  - MongoDB
  - Redis

* Uses SQL-like languages or builds SQL engine right now

SQL use cases: technology

- “Old-school” databases (MySQL, Postgres, SQL Server, Oracle)
- “New” products
  - Relational (CockroachDB, TiDB, YugaByte)
  - BigData/Analytics (Hive, Snowflake, Dremio, Clickhouse, Presto)
  - NoSQL (DataStax*, Couchbase*)
  - Compute/streaming (Spark, ksqlDB, Apache Flink)
  - In-memory (Apache Ignite, Hazelcast, Gigaspaces)
- Rebels:
  - MongoDB
  - Redis

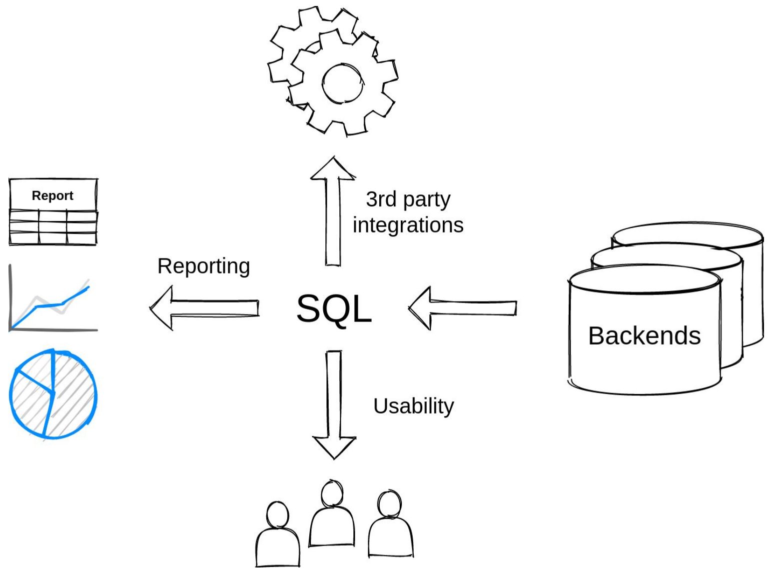


* Uses SQL-like languages or builds SQL engine right now

https://insights.stackoverflow.com/survey/2020

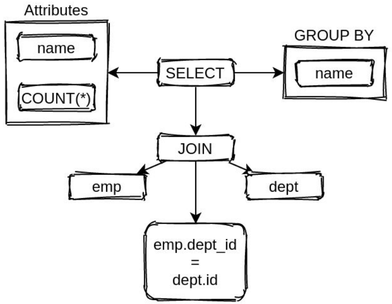
SQL use cases: applied

- Query custom data sources
  - Internal business systems
  - Infrastructure: logs, metrics, configs, events, ...
- Federated SQL - run queries across multiple sources
  - Data lakes
- Custom requirements
  - New syntax / DSL
  - UDFs
  - Internal optimizations

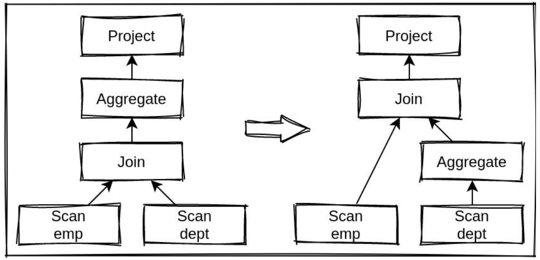


What does it take to build an SQL engine?

Syntax ans Semantic Analysis

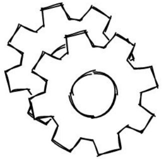


Intermediate Representation



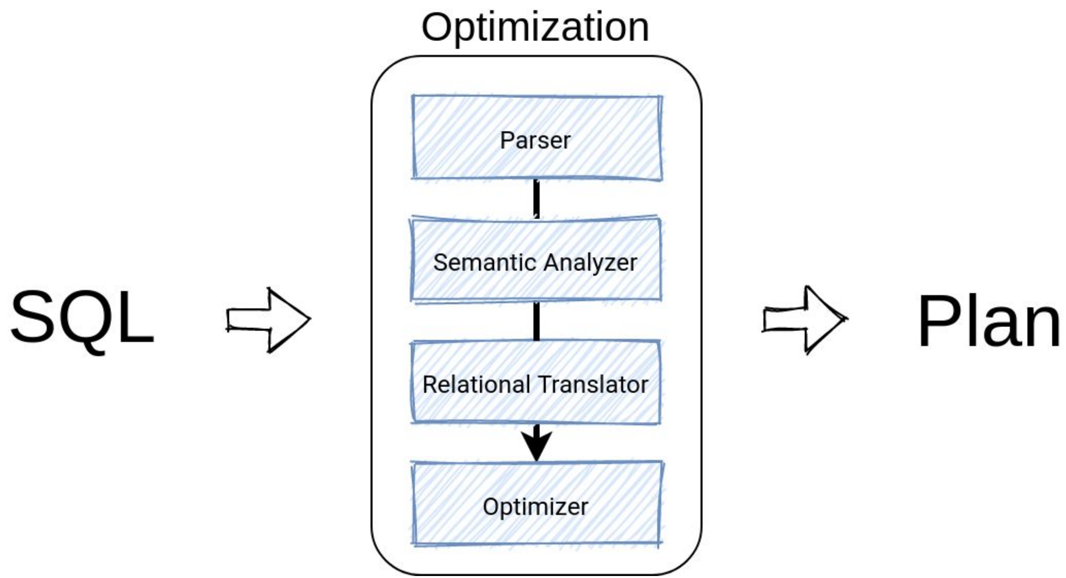

```
SELECT dept.name, COUNT(*)
FROM emp, dept
WHERE
  emp.dept_id = dept.id
GROUP BY dept.name
```

Query

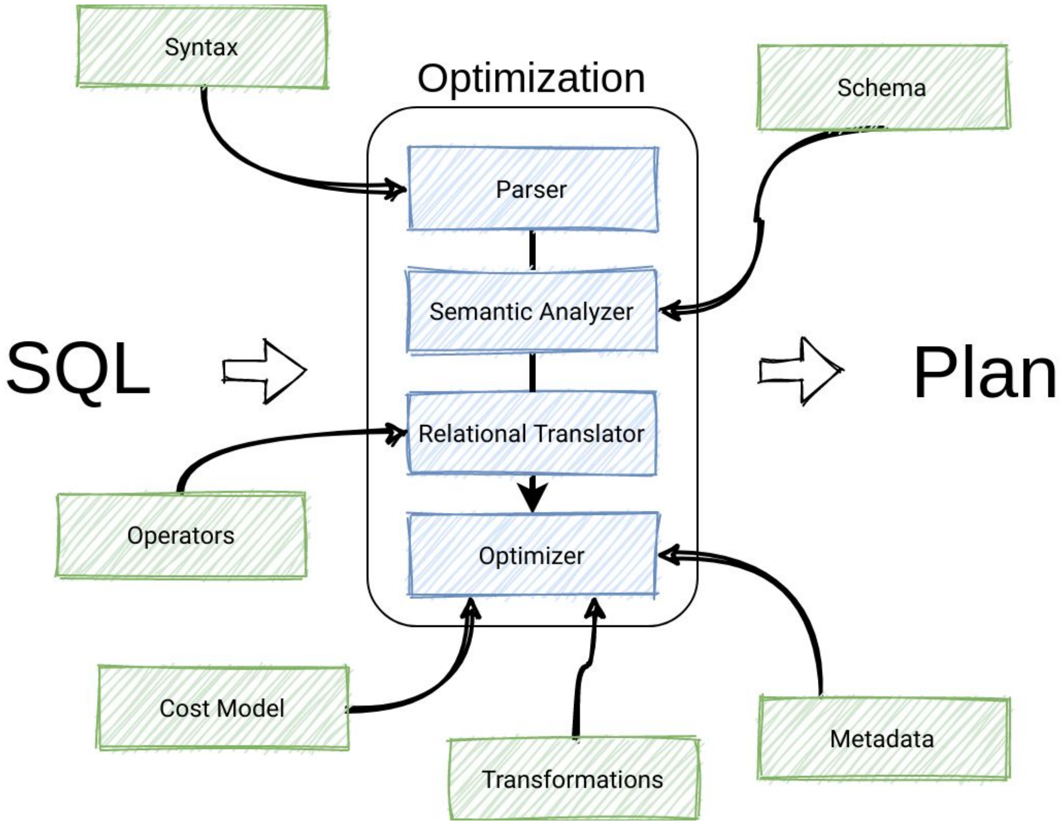


Backend

Optimization with Apache Calcite



Optimization with Apache Calcite



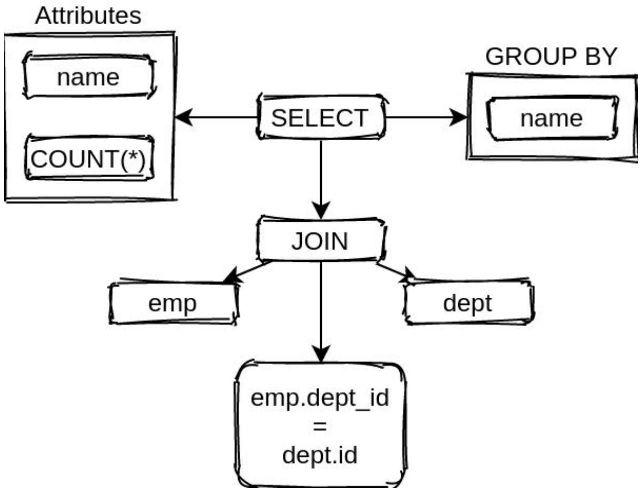
Projects that already use Apache Calcite

- Data Management:
  - Apache Hive
  - Apache Flink
  - Dremio
  - VoltDB
  - IMDGs (Apache Ignite, Hazelcast, Gigaspaces)
  - …
- Applied:
  - Alibaba / Ant Group
  - Uber
  - LinkedIn
  - …

https://calcite.apache.org/docs/powered_by.html

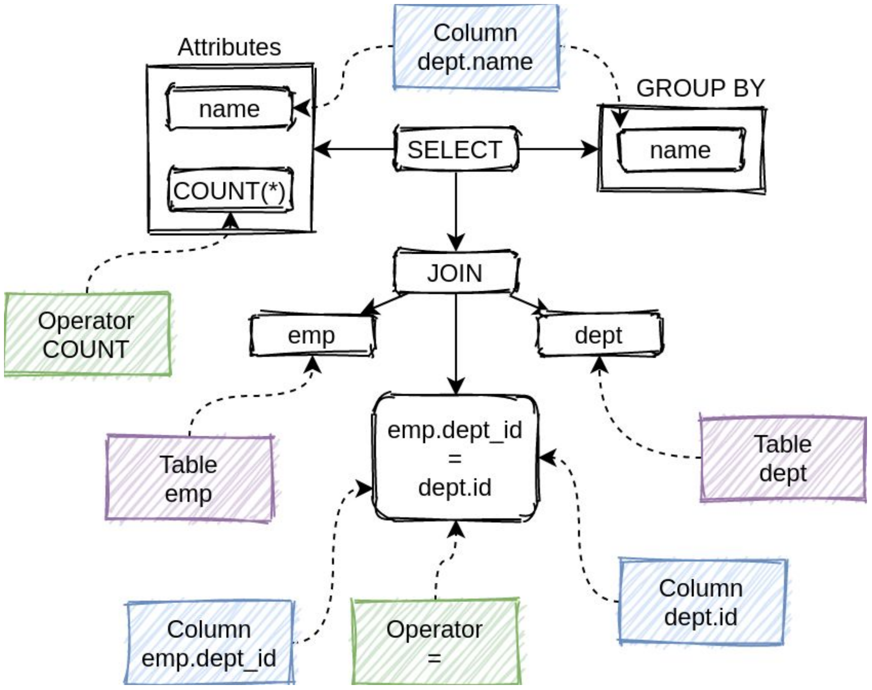
Parsing

- Goal: convert query string to AST
- How to create a parser?
  - Write a parser by hand? Not practical
  - Use parser generator? Better, but still a lot of work
  - Use Apache Calcite
- Parsing with Apache Calcite
  - Uses JavaCC parser generator under the hood
  - Provides a ready-to-use generated parser with the ANSI SQL grammar
  - Allows for custom extensions to the syntax

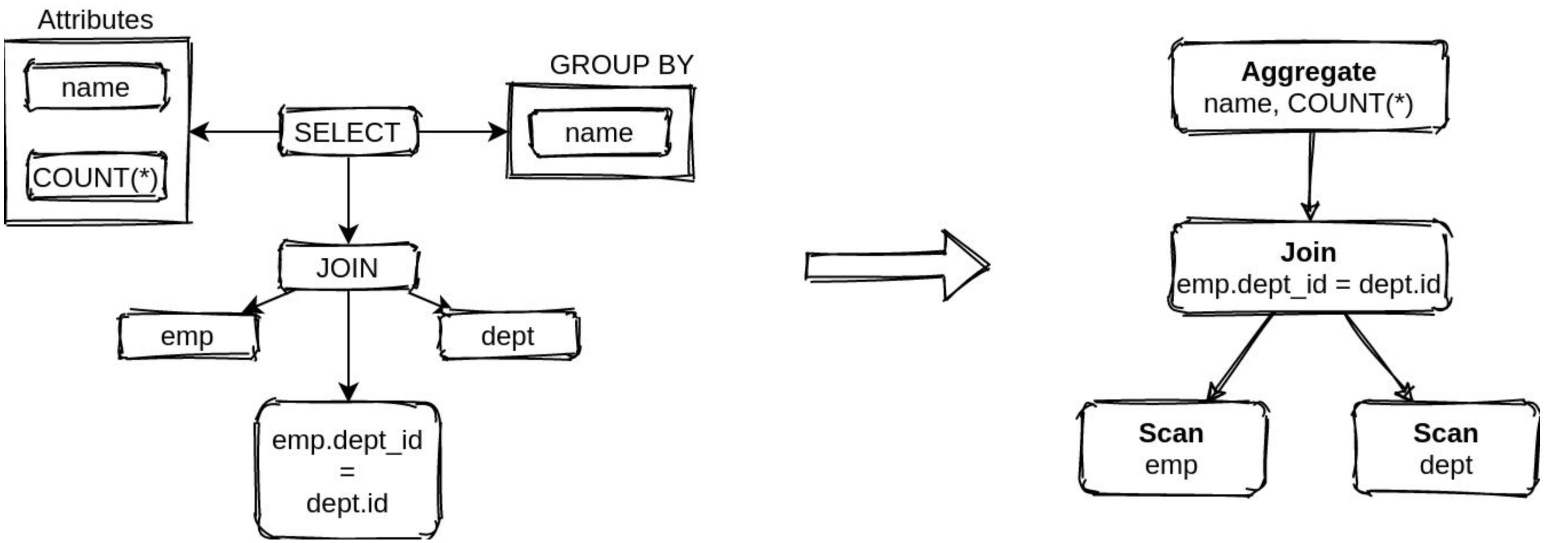


Semantic Analysis

- Goal: verify that AST makes any sense
- Semantic analysis with Apache Calcite
  - Provide a schema
  - (optionally) Provide custom operators
  - Run Calcite's SQL validator
- Validator responsibilities
  - Bind tables and columns
  - Bind operators
  - Resolve data types
  - Verify relational semantics



Relational tree

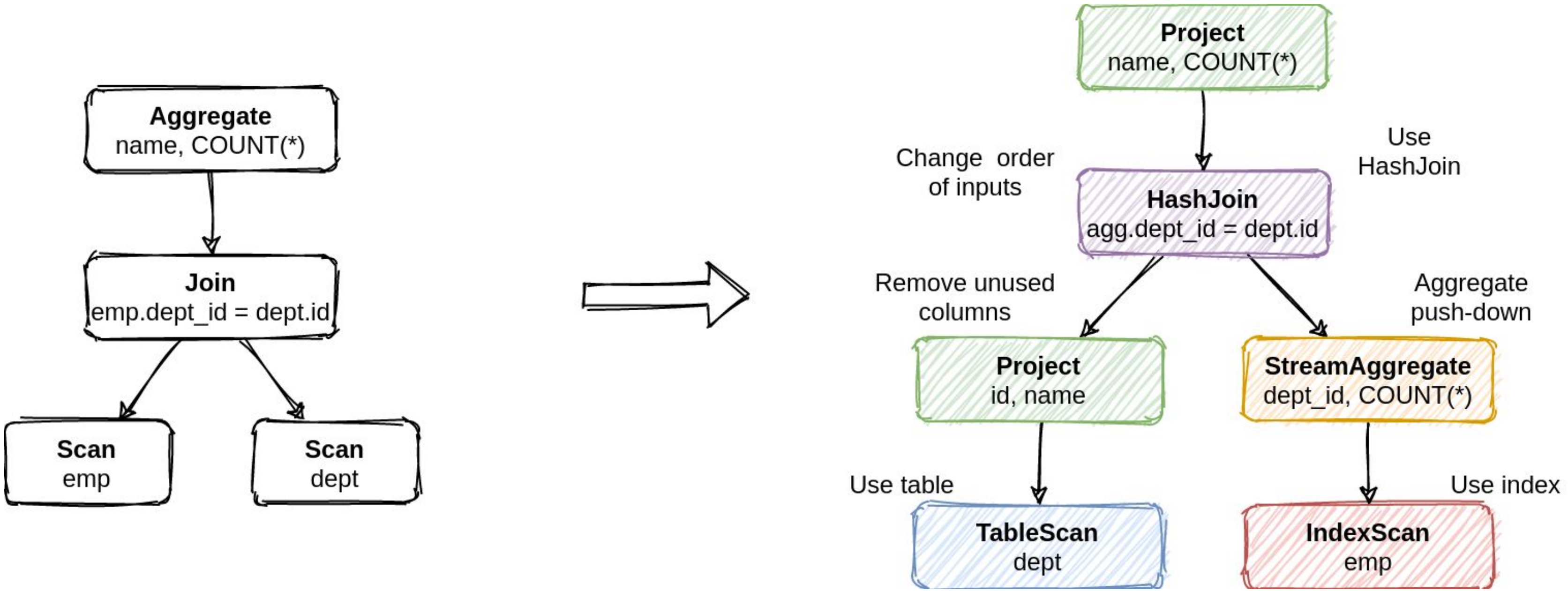


- AST is not convenient for optimization: complex operator semantics
- A relational tree is a better IR: simple operators with well-defined scopes
- Apache Calcite can translate AST to relational tree

Relational tree

Operator	Description
`Scan`	Scan a data source
`Project`	Transform tuple attributes (e.g. `a+b`)
`Filter`	Filter rows according to a predicate (WHERE, HAVING)
`Sort`	ORDER BY / LIMIT / OFFSET
`Aggregate`	Aggregate operator
`Window`	Window aggregation
`Join`	2-way join
`Union/Minus/Intersect`	N-way set operators

Transformations

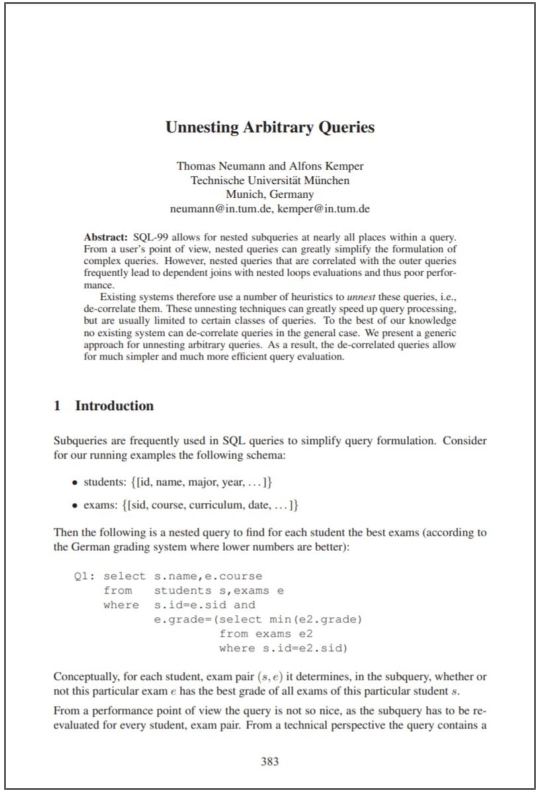


- Every query might be executed in multiple alternative ways
- We need to apply **transformations** to find better plans
- Apache Calcite: custom transformations (visitors) or **rule-based** transformations

Transformations: custom

Custom transformations implemented using a visitor pattern (traverse the relational tree, create a new tree):

- **Field trimming**: remove unused columns from the plan
- **Subquery elimination**: rewrite subqueries to joins/aggregates



Unnesting Arbitrary Queries

Thomas Neumann and Alfons Kemper
Technische Universität München
Munich, Germany
neumann@in.tum.de, kemper@in.tum.de

Abstract: SQL-99 allows for nested subqueries at nearly all places within a query. From a user's point of view, nested queries can greatly simplify the formulation of complex queries. However, nested queries that are correlated with the outer queries frequently lead to dependent joins with nested loops evaluations and thus poor performance.

Existing systems therefore use a number of heuristics to *unnest* these queries, i.e., de-correlate them. These unnesting techniques can greatly speed up query processing, but are usually limited to certain classes of queries. To the best of our knowledge no existing system can de-correlate queries in the general case. We present a generic approach for unnesting arbitrary queries. As a result, the de-correlated queries allow for much simpler and much more efficient query evaluation.

1 Introduction

Subqueries are frequently used in SQL queries to simplify query formulation. Consider for our running examples the following schema:

- students: {[id, name, major, year, ...]}
- exams: {[sid, course, curriculum, date, ...]}

Then the following is a nested query to find for each student the best exams (according to the German grading system where lower numbers are better):

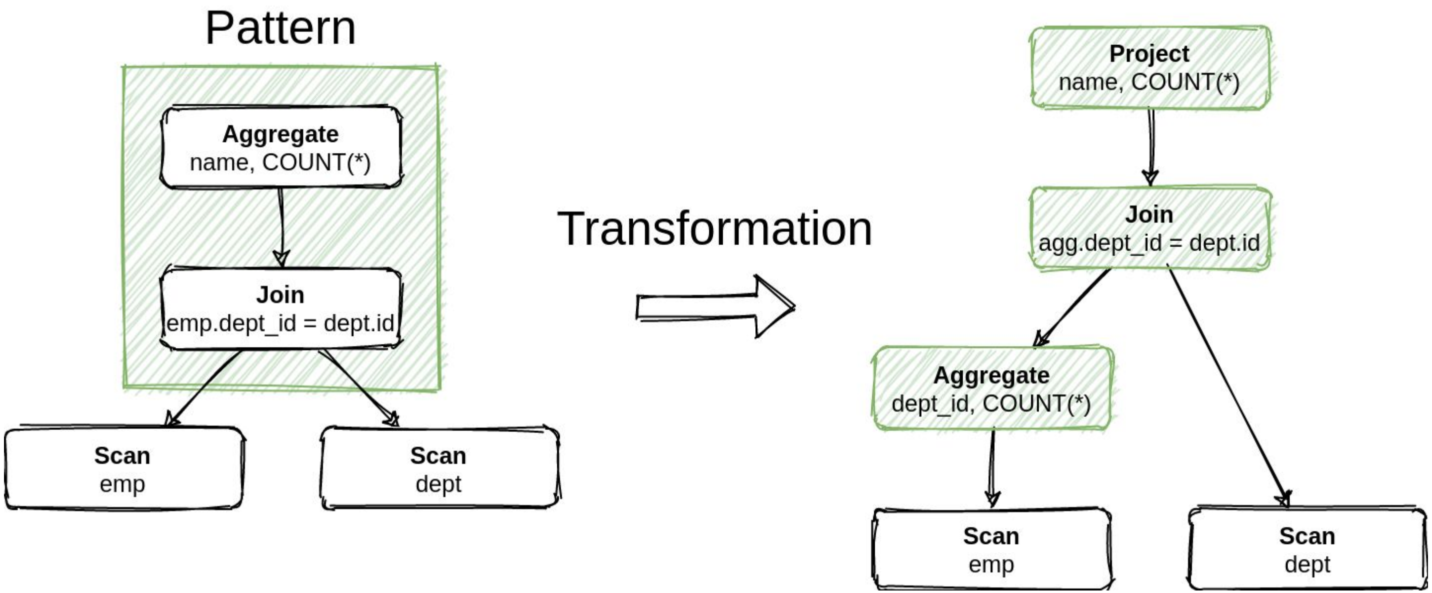
```
Q1: select s.name,e.course
    from   students s,exams e
    where  s.id=e.sid and
           e.grade=(select min(e2.grade)
                    from exams e2
                    where s.id=e2.sid)
```

Conceptually, for each student, exam pair (s, e) it determines, in the subquery, whether or not this particular exam e has the best grade of all exams of this particular student s.

From a performance point of view the query is not so nice, as the subquery has to be re-evaluated for every student, exam pair. From a technical perspective the query contains a

383

Transformations: rule-based



- A **rule** is a self-contained optimization unit: pattern + transformation
- There are **hundreds** of valid transformations in relational algebra
- Apache Calcite provides ~100 transformation rules out-of-the-box!

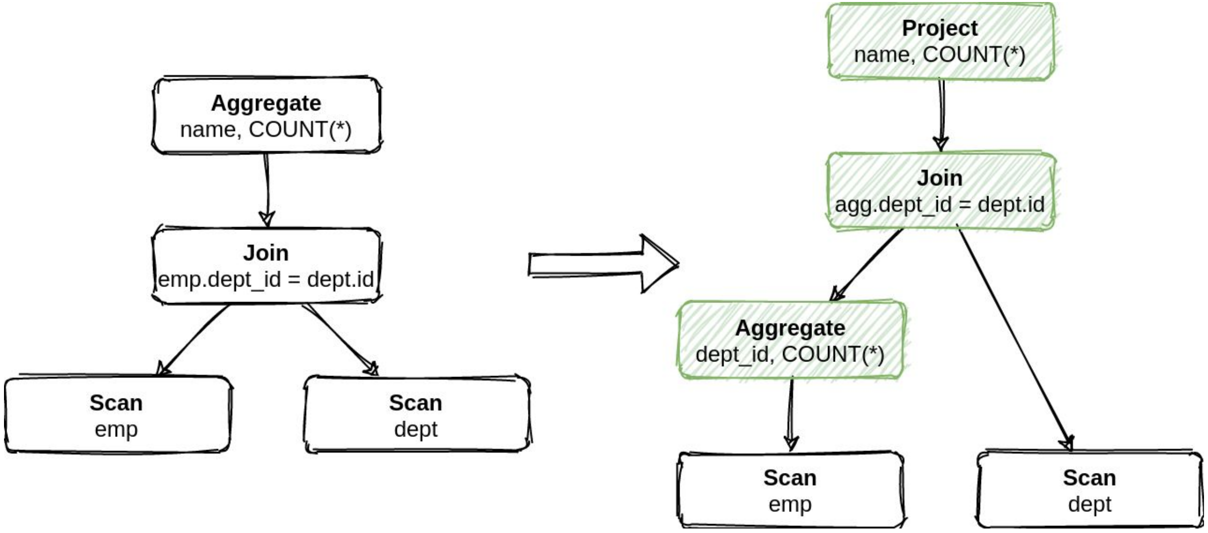
Rules

Examples of rules:

- Operator transpose - move operators wrt each other (e.g., filter push-down)
- Operator simplification - merge or eliminate operators, convert to simpler equivalents
- Join planning - commute, associate

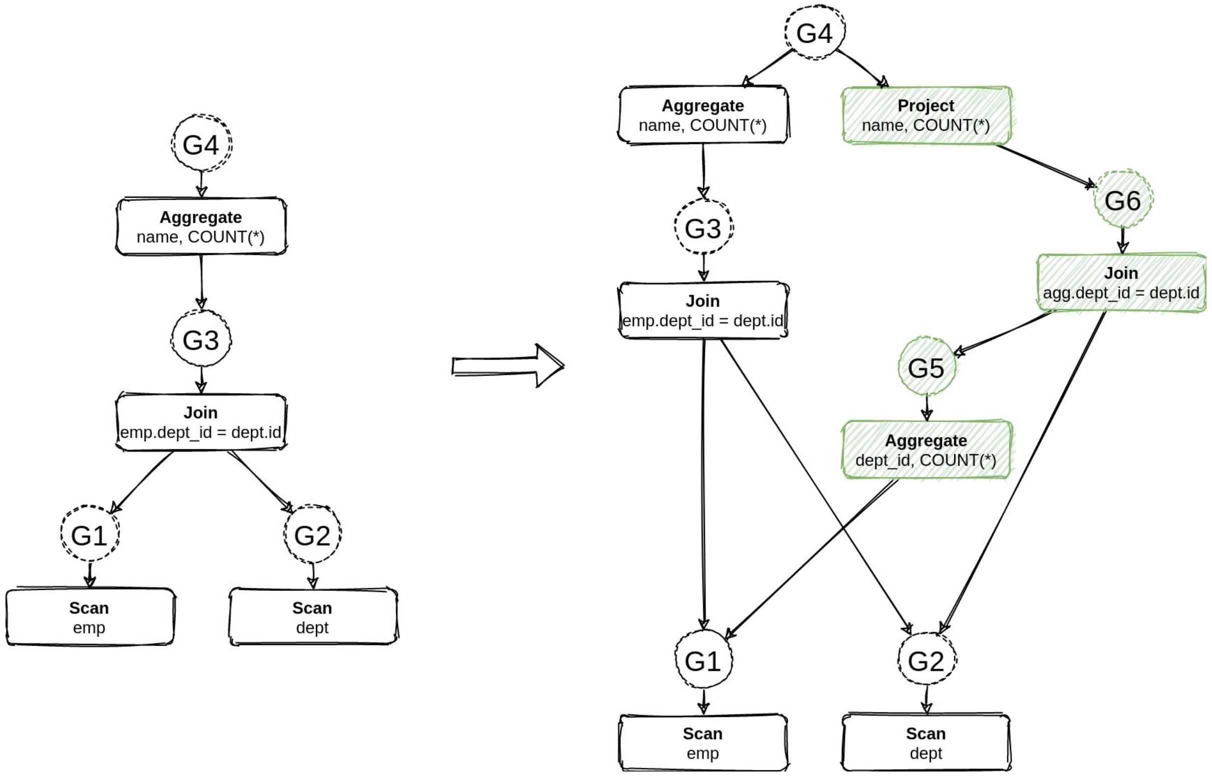
https://github.com/apache/calcite/tree/master/core/src/main/java/org/apache/calcite/rel/rules

Rule drivers: heuristic (HepPlanner)



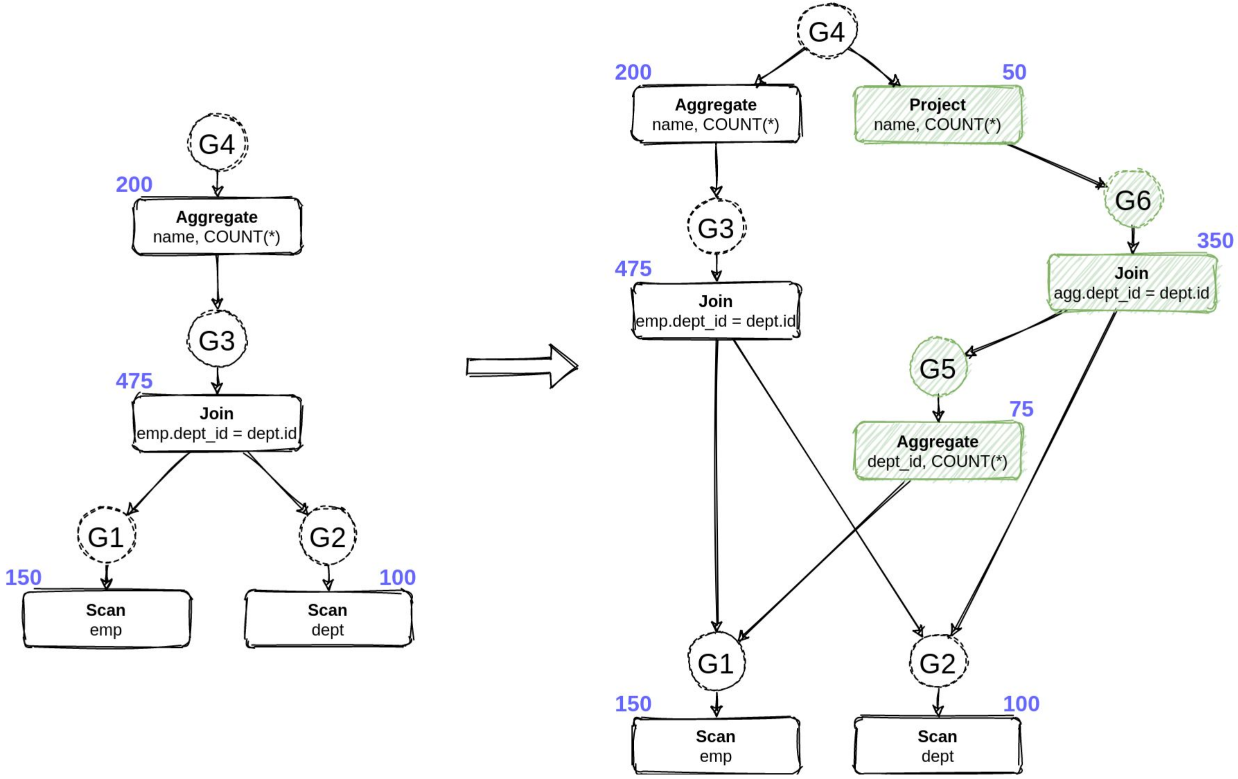
- Apply transformations until there is anything to transform
- Fast, but cannot guarantee optimality

Rule drivers: cost-based (VolcanoPlanner)



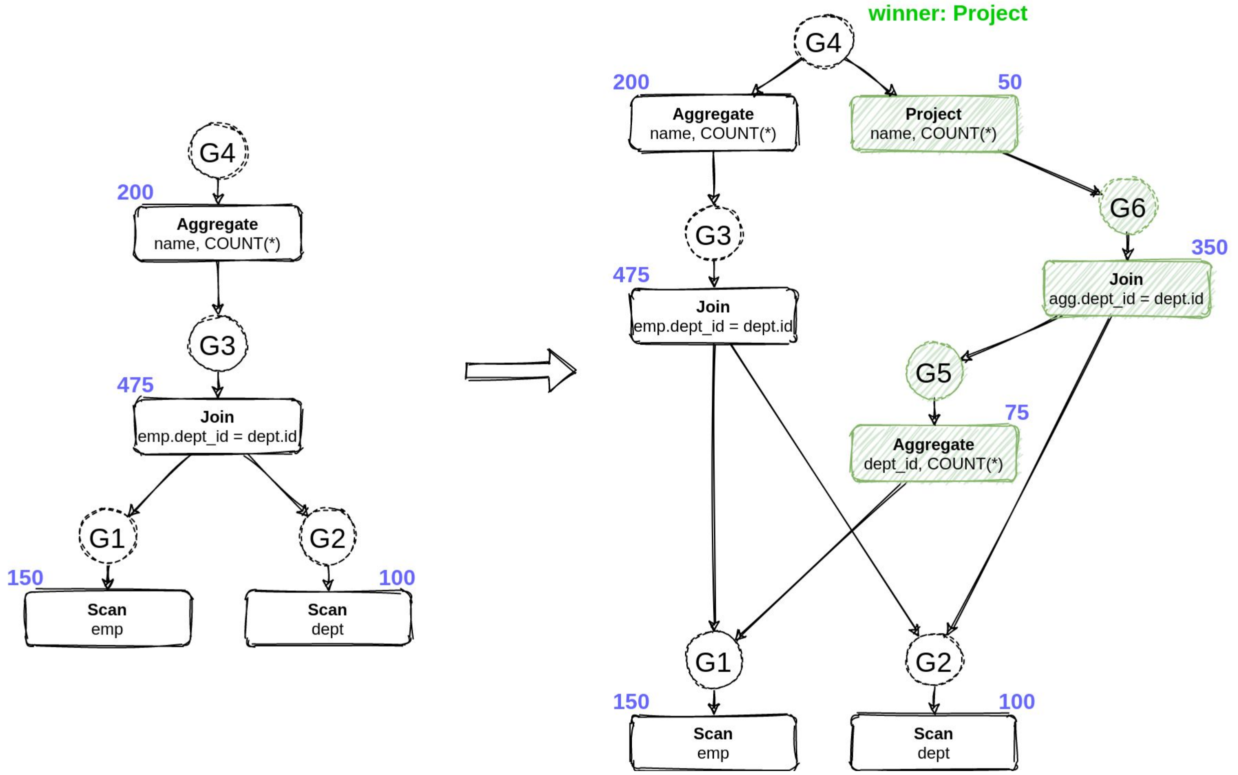
- Consider multiple plans **simultaneously** in a special data structure (MEMO)

Rule drivers: cost-based (VolcanoPlanner)



- Consider multiple plans **simultaneously** in a special data structure (MEMO)
- Assign non-cumulative costs to operators

Rule drivers: cost-based (VolcanoPlanner)



- Consider multiple plans **simultaneously** in a special data structure (MEMO)
- Assign non-cumulative costs to operators
- Maintain the winner for every equivalence group
- Heavier than the heuristic driver but guarantees optimality

Metadata

Metadata is a set of properties, common to all operators in the given equivalence group. Used extensively in rules and cost functions.

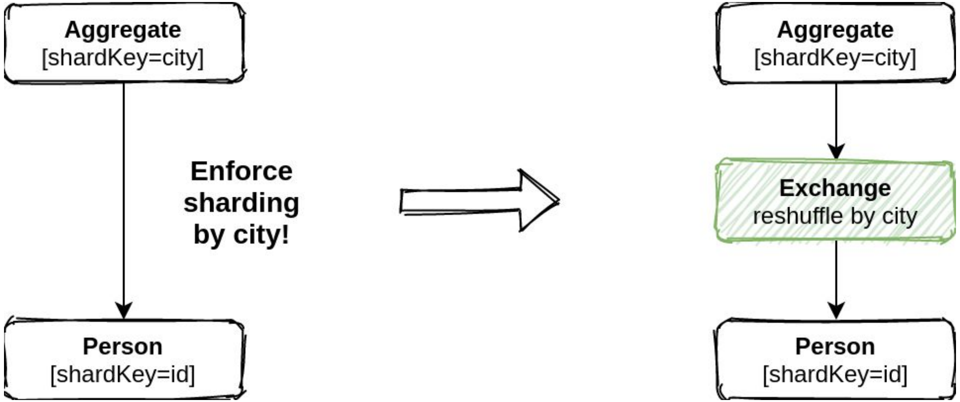
Examples:

- Statistics (cardinalities, selectivites, min/max, NDV)
- Attribute uniqueness
  - `SELECT a … GROUP BY a` -> the first attribute is unique
- Attribute constraints
  - `WHERE a.a1=1 and a.a1=b.b1` -> both `a.a1` and `b.b1` are always 1 and their NDV is 1

Implementing an operator

- Create your custom operator, extending the `RelNode` class or one of existing abstract operators
- Override the **copy** routine to allow for operator copying to/from MEMO (`copy`)
- Override operator’s **digest** for proper deduplication (`explainTerms`)
  - Usually: dump a minimal set of fields that makes the operator unique wrt other operators.
- Override the **cost function** (`computeSelfCost`)
  - Usually: consult to metadata, first of all input’s cardinality, apply some coefficients.
  - You may even provide you own definition of the cost

Enforcers



- Operators may expose physical properties
- Parent operator may **demand** a certain property on the input
- If the input cannot **satisfy** the requested property, an enforcer operator is injected
- Examples:
  - Collation (Sort)
  - Distribution (Exchange)

VolcanoOptimizer

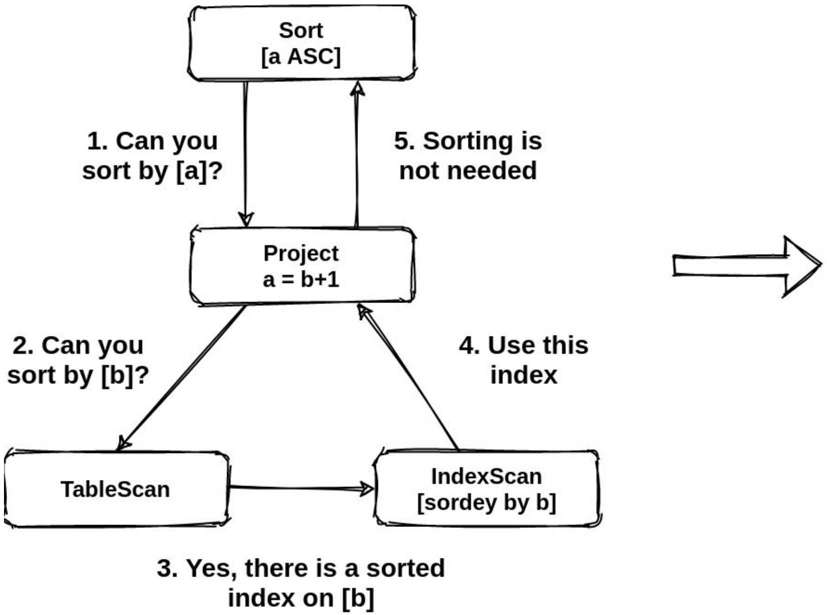
Vanilla

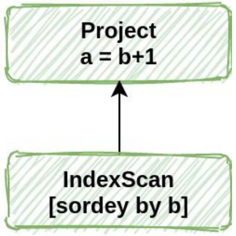
- The original implementation of the cost-based optimizer in Apache Calcite.
- Optimize nodes in an arbitrary order.
- Cannot propagate physical properties.
- Cannot do efficient pruning.

Top-down

- Implemented recently by Alibaba engineers
- Based on the Cascades algorithm: the guided top-down search.
- Propagates the physical properties between operators (requires manual implementation).
- Applies branch-and-bound pruning to limit the search space.

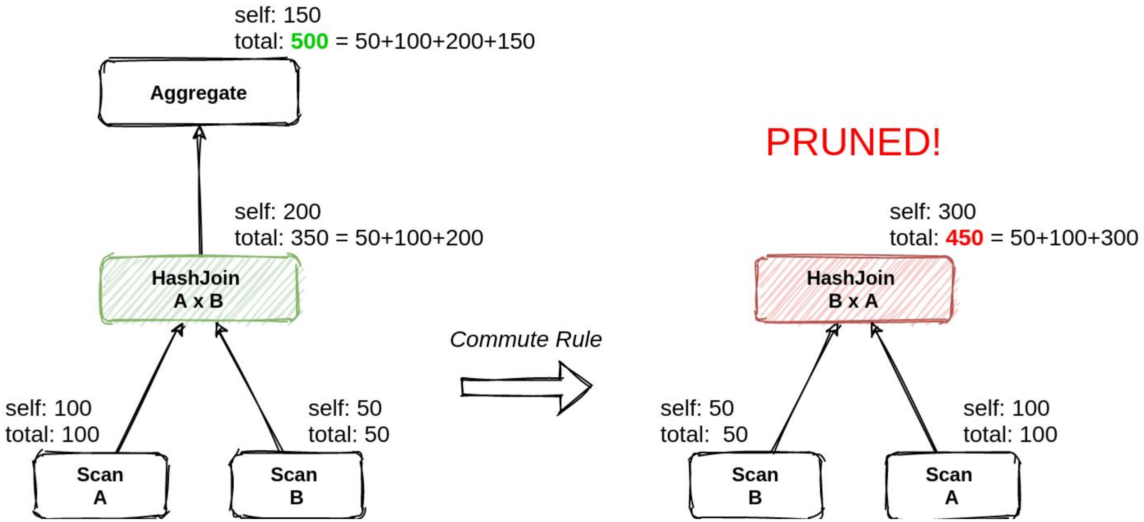
Physical property propagation





- Available only in the top-down optimizer
- **Pass-through** (1, 2, 3) - propagate optimization request to inputs
- **Derive** (4, 5) - notify the parent about the new implementation

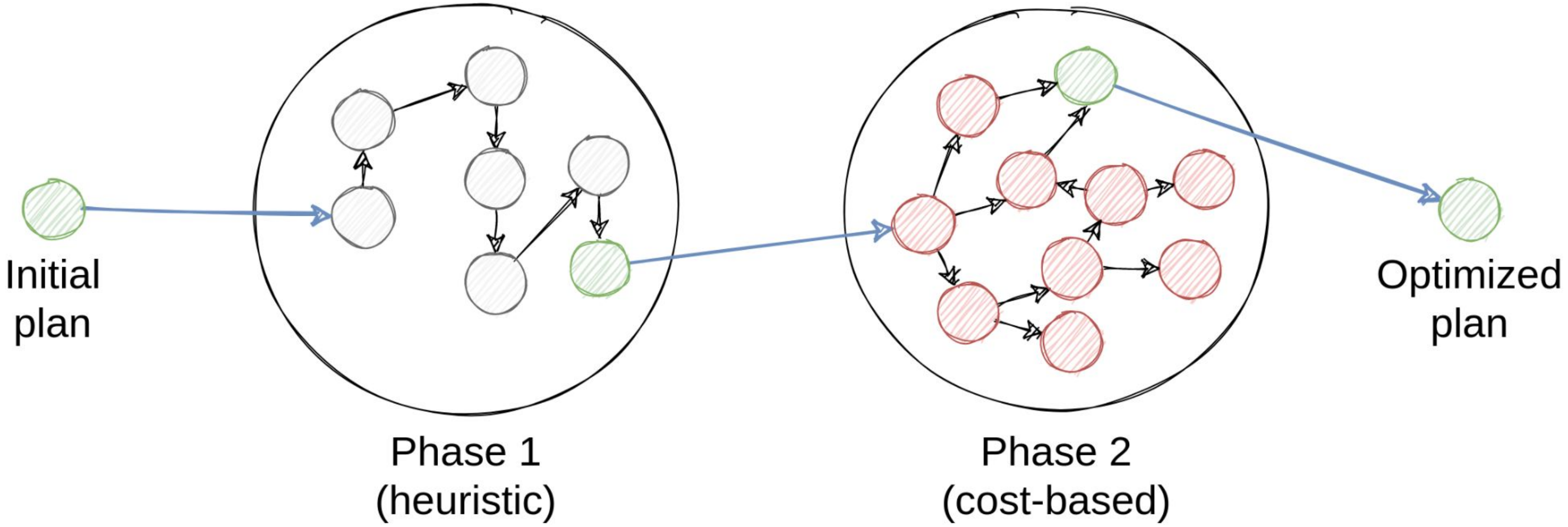
Branch-and-bound pruning



Accumulated cost bounding:

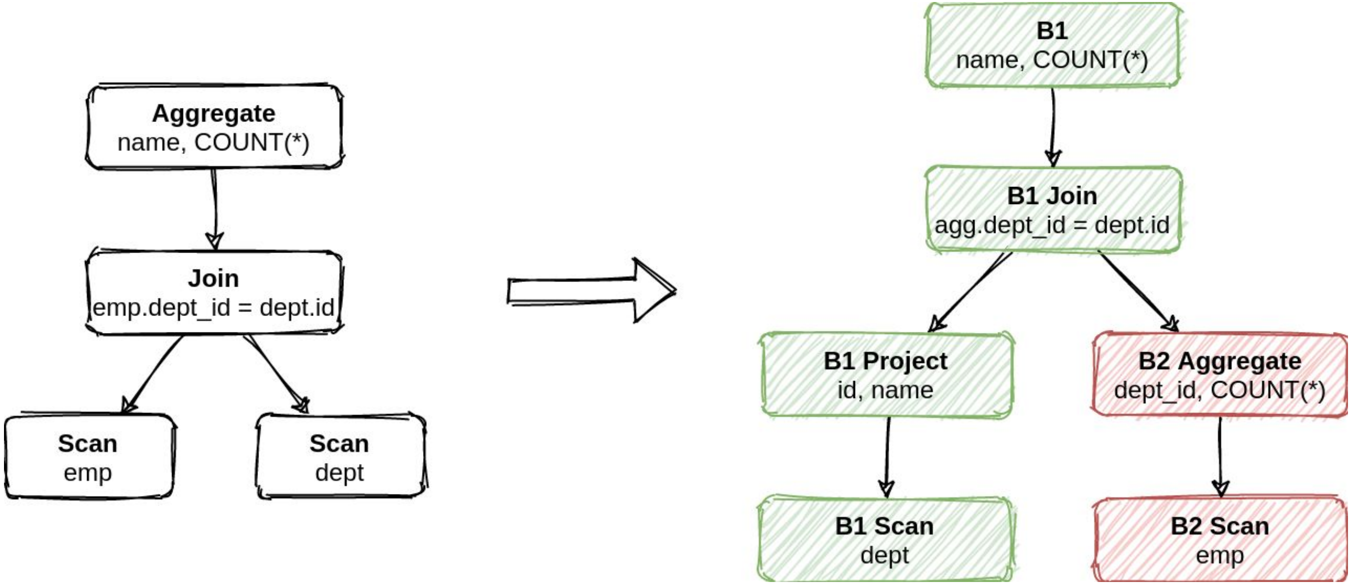
- There is a viable aggregate
  - Total cost = 500
  - Self cost = 150
  - **Input's budget** = **350**
- The new join is created
  - Self cost = **450**
  - May never be part of an optimal plan, prune

Multi-phase optimization



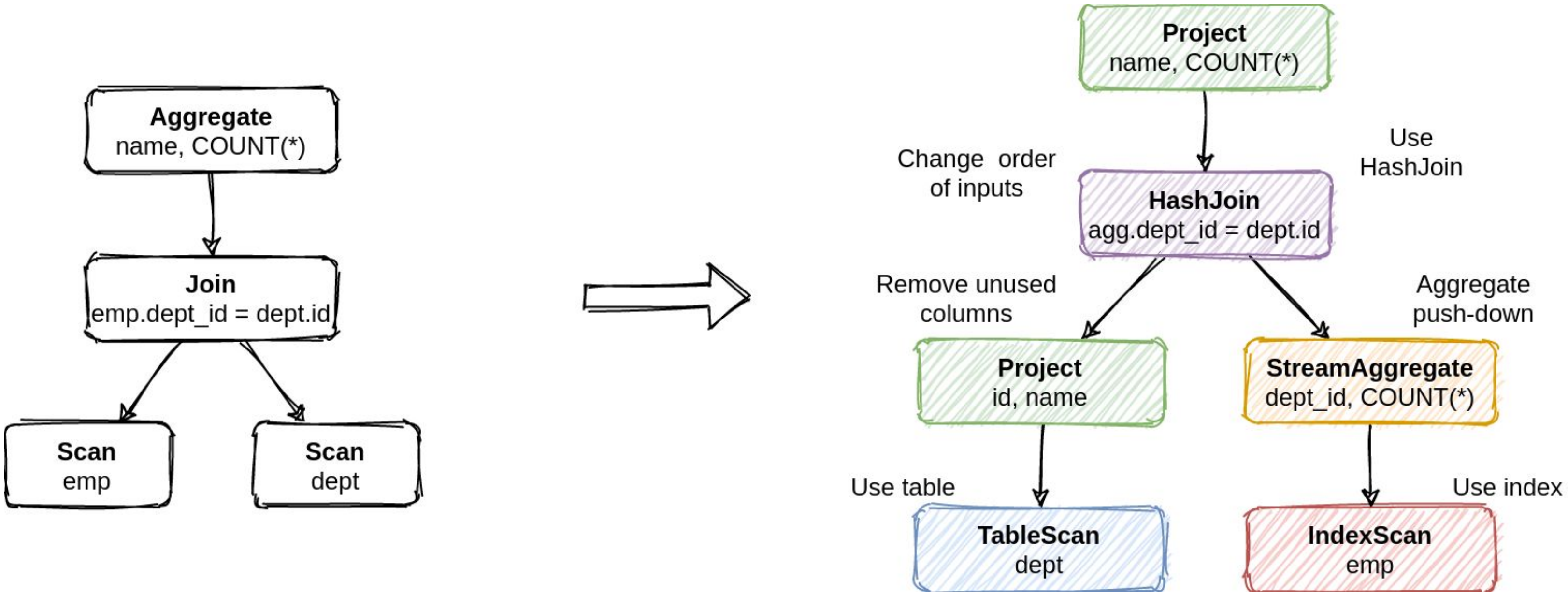
- Practical optimizers often split optimization into several phases to reduce the search space, at the cost of possibly missing the optimal plan
- Apache Calcite allows you to implement a multi-phase optimizer

Federated queries



- You may optimize towards different backends simultaneously (federated queries)
    - E.g., JDBC + Apache Cassandra
- Apache Calcite has the built-in `Enumerable` execution backend that compiles operators into a Java bytecode in runtime

Your optimizer



- Define **operators** specific to your backend
- Provide custom **rules** that convert abstract Calcite operators to your operators
  - E.g., `LogicalJoin -> HashJoin`
- Run Calcite driver(s) with the built-in and/or custom rules

Example: Apache Flink

- Custom physical batch and streaming operators
- Custom cost: row count, cpu, IO, network, memory
- The custom distribution property with an `Exchange` enforcer
- Custom rules (e.g., subquery rewrite, physical rules)
- Multi-phase optimization: heuristic and cost-based phases

https://github.com/apache/flink/tree/release-1.12.2/flink-table/flink-table-planner-blink/src/main/scala/org/apache/flink/table/planner

Summary

- Apache Calcite is a toolbox to build query engines
  - Syntax analyzer
  - Semantic analyzer
  - Translator
  - Optimization drivers and rules
  - The `Enumerable` backend
- Apache Calcite dramatically reduces the efforts required to build an optimizer for your backend
  - Weeks to have a working prototype
  - Months to have an MVP
  - Year(s) to have a solid product, **but not decades**!

Links

- Speaker
  - https://www.linkedin.com/in/devozerov/
  - https://twitter.com/devozerov
- Apache Calcite:
  - https://calcite.apache.org/
  - https://github.com/apache/calcite
- Demo:
  - https://github.com/querifylabs/talks-2021-percona
- Our blog:
  - https://www.querifylabs.com/blog