Querify Labs

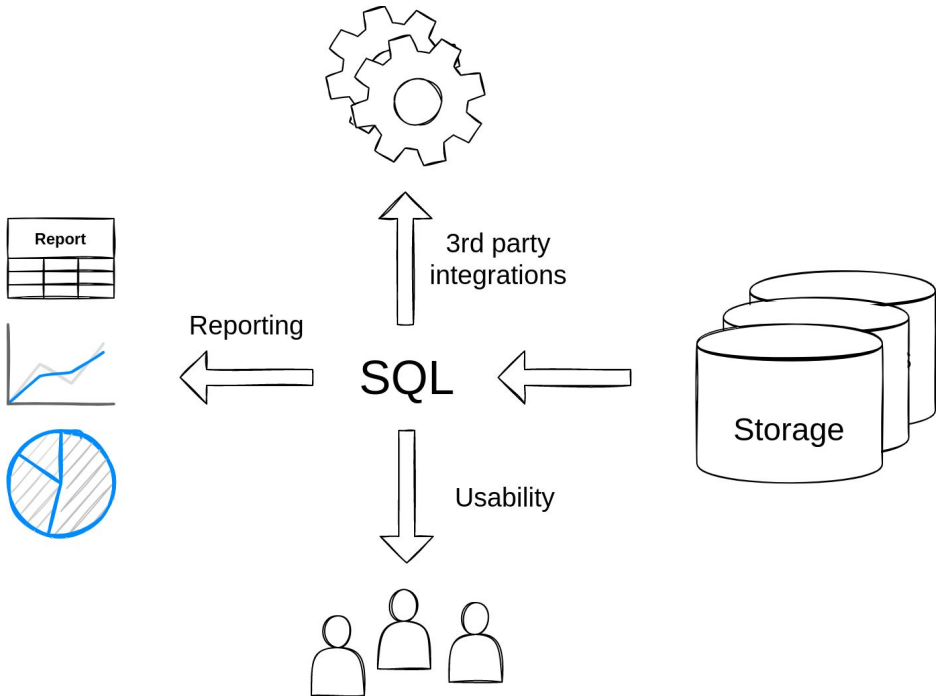
Apache Calcite: Платформа для создания продвинутых SQL-оптимизаторов на Java

Владимир Озеров
Querify Labs, CEO

О нас

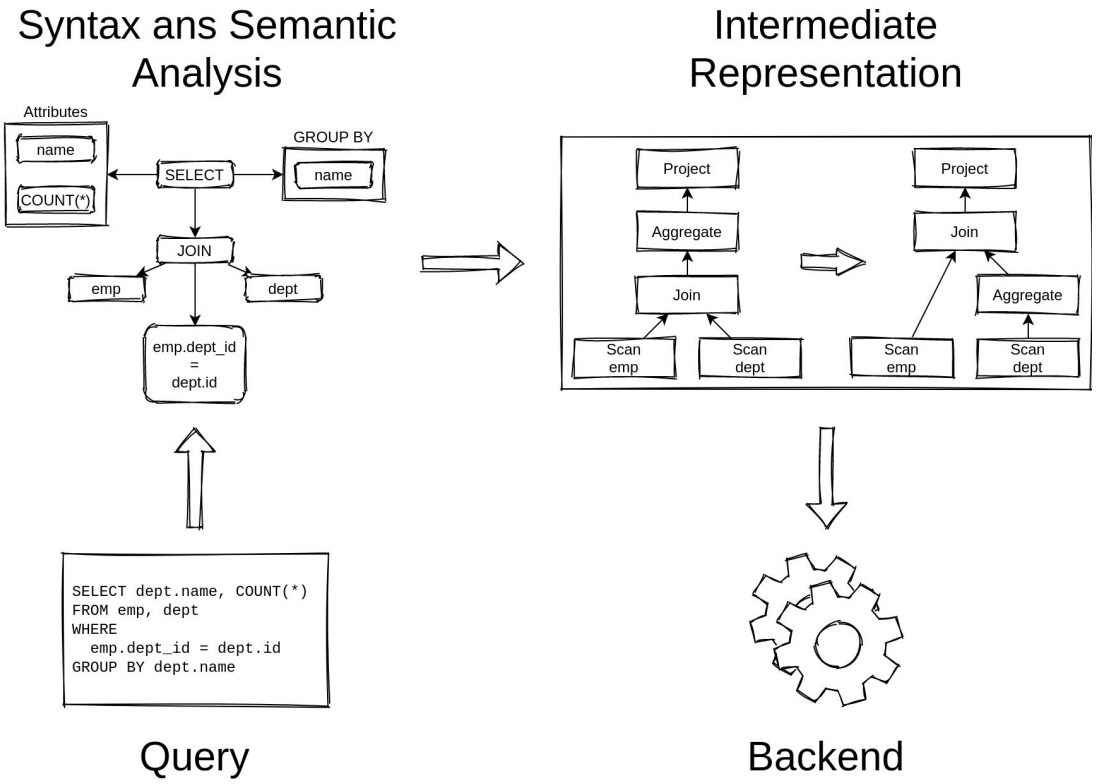
- Компания Querify Labs [1]
  - Разработка новых СУБД и data management систем: движки, оптимизаторы, storage, протоколы.
- Работал 8 лет над распределенными in-memory движками:
  - Apache Ignite
  - Hazelcast
- Контрибьютор в Apache Calcite [2] и Apache Ignite

Распределенный SQL

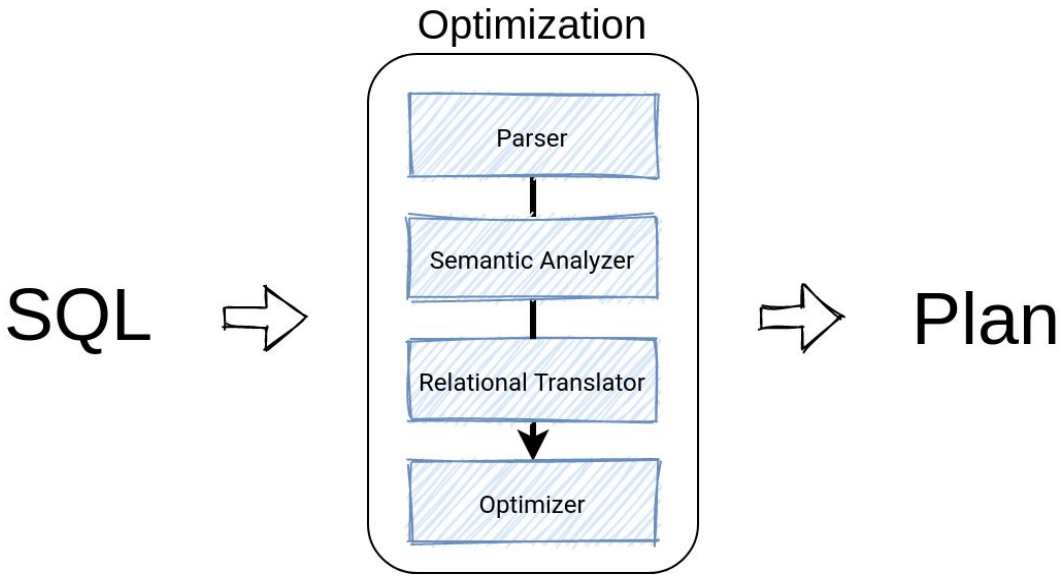


- Реляционные СУБД:
  - CockroachDB, TiDB, YugaByte
- BigData/Analytics:
  - Hive, Snowflake, Dremio, Clickhouse, Presto
- NoSQL:
  - DataStax, Couchbase
- Compute/streaming:
  - Spark, ksqlDB, Apache Flink
- IMDG:
  - Apache Ignite, Hazelcast, Gigaspaces

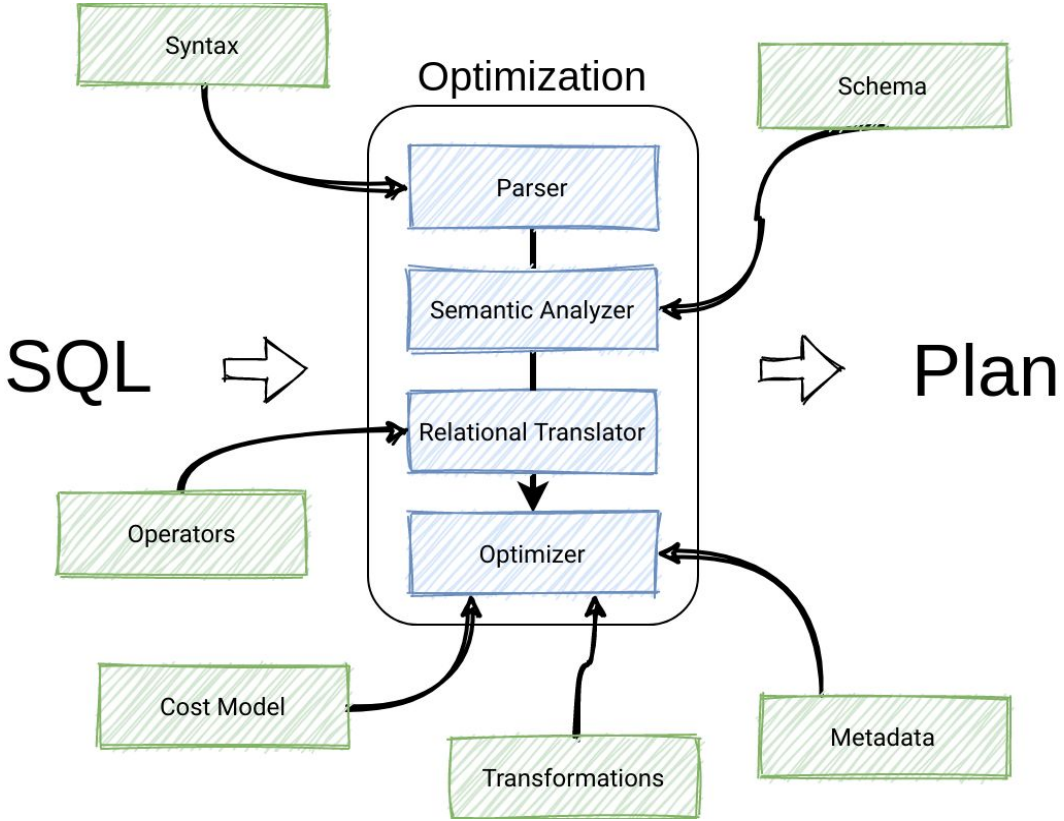
Что делает оптимизатор?



Оптимизация с Apache Calcite



Оптимизация с Apache Calcite



Проекты, которые используют Apache Calcite

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

О чем доклад?

- Что
  - Ключевые алгоритмы и проблемы оптимизаторов SQL-запросов.
  - Как они решены в Apache Calcite.
- Кому
  - Практикующим инженерам, которые используют SQL-продукты.
  - Если вам интересно внутреннее устройство СУБД, или вы хотите стать разработчиком СУБД.
  - Если вы хотите порадоваться за Java.

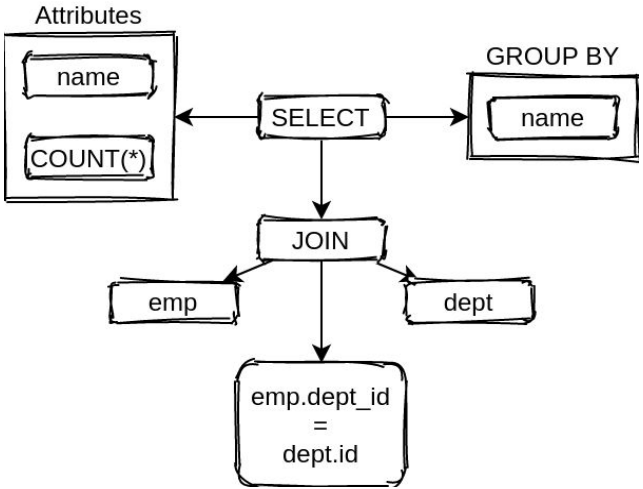
Основные стадии

- Анализ
  - Парсинг.
  - Семантическая валидация.
  - Трансляция в IR (intermediate representation).
- Оптимизация
  - Логическая - поиск абстрактных операторов (напр, Join(AxB) vs Join(BxA)).
  - Физическая - выбор реализации операторов (напр, hash join vs nested loop join).

Парсинг

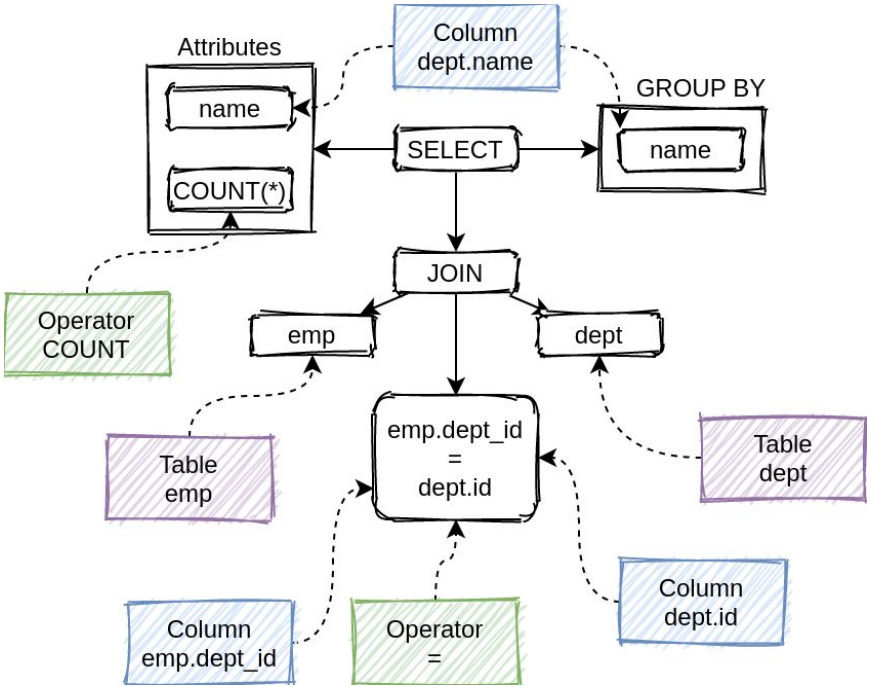
- Задача: превратить строку в синтаксическое дерево.
- Парсинг с Apache Calcite:
  - Использует JavaCC для генерации парсера [3].
  - Имеет готовый к использованию сгенерированный парсер [4].
  - Допускает расширения синтаксиса (напр., DDL [5]).

```
var sql = "SELECT ...";
var parser = SqlParser.create(sql);
SqlNode ast =parser.parseQuery();
```

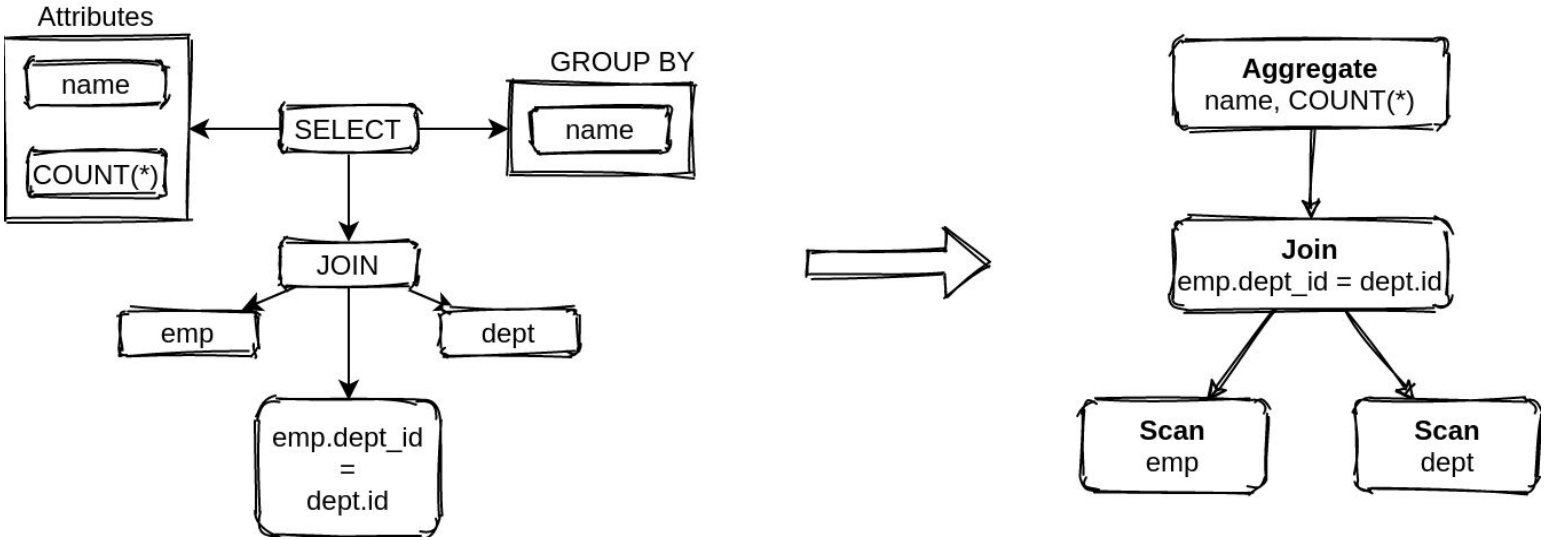


Семантический анализ

- Задача: убедиться в семантической корректности синтаксического дерева:
  - Разрешение объектов СУБД (таблицы, колонки, …).
  - Разрешение функций.
  - Вывод типов.
  - Проверка реляционной семантики.
- Семантический анализ в Apache Calcite:
  - Предоставьте схему.
  - Предоставьте кастомные функции (опционально).
  - Запустите встроенный валидатор [6].



Трансляция в реляционное дерево



- Оптимизировать синтаксическое дерево проблематично из-за высокой сложности SQL-синтаксиса.
- Для задачи оптимизации, удобнее представить запрос в виде дерева реляционных операторов **[BLOG2]** [7] [8].
- Apache Calcite имеет встроенный транслятор из AST в дерево реляционных операторов [9].
- Большинство трансформаций Apache Calcite работают с реляционными операторами.
- В других системах: Apache Spark [37], Presto/Trino [38].

RelNode

```
abstract class AbstractRelNode {
    // Доступ к дочерним узлам.
    RelNode getInput(int i);

    // Список атрибутов (“полей”) узла.
    RelDataType getRowType();

    // Сконструировать уникальную сигнатуру узла.
    RelWriter explainTerms(RelWriter pw);

    // Посчитать стоимость узла (без учета дочерних узлов).
    RelOptCost computeSelfCost(RelOptPlanner planner, RelMetadataQuery mq);
}
```

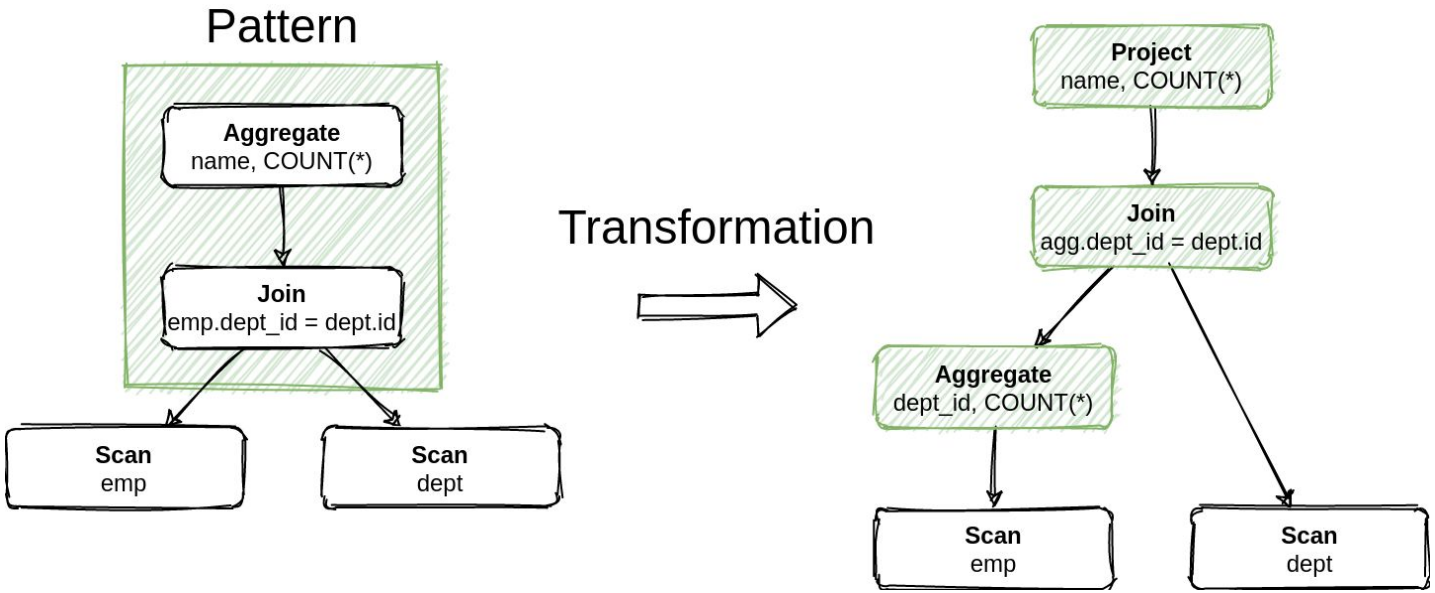
Реляционные операторы

Оператор	**Описание**
Scan	Сканировать абстрактный источник данных
Project	Трансформировать кортеж (напр., a+b)
Filter	Отфильтровать кортежи (WHERE, HAVING)
Sort	ORDER BY / LIMIT / OFFSET
Aggregate	Агрегация
Window	Оконная агрегация
Join	Классический join двух операторов
Union/Minus/Intersect	Set-операторы

Оптимизация

- Rule-based оптимизация.
- Понятие “стоимости”.
- Итеративный оптимизатор.
- Cost-based оптимизатор.

Rule-based оптимизация

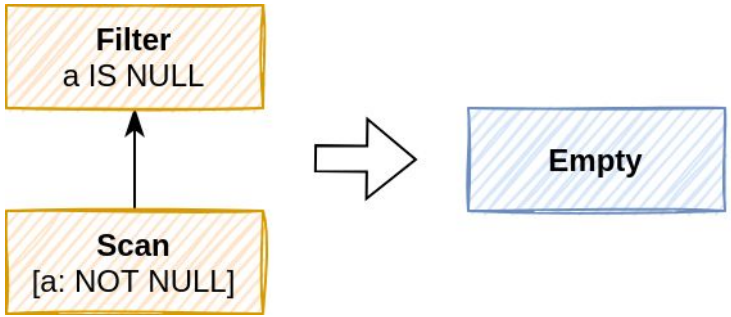


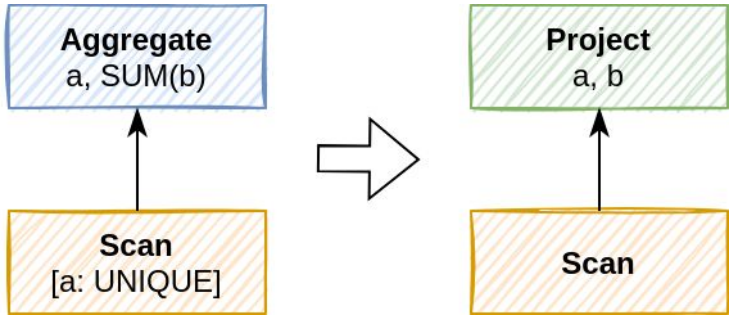
- Правило - это единица оптимизации, состоящая из паттерна и логики трансформации **[BLOG3]**.
- Значительно упрощает эволюцию и поддержку планировщика за счет декомпозиции.
- SQL-движки могут содержать сотни трансформаций.
- **Примеры**: Apache Calcite [17], Apache Spark [18], Presto [19], CockroachDB [20].

Пример правила

```
public class AggregateJoinTransposeRule extends RelRule {
  // Логика
  public void onMatch(RelOptRuleCall call) {
    Join newJoin = ...          // 400 строк сложного кода
    call.transformTo(newJoin); // Сообщить оптимизатору о новом узле
  }

  // Паттерн
  public interface Config extends RelRule.Config {
    Config DEFAULT = EMPTY.as(Config.class)
      .withOperandSupplier(
        b0 -> b0.operand(Aggregate.class)       // Aggregate ...
        .oneInput(b1 -> b1.operand(Join.class) // ... над Join ...
        .anyInputs()                           // ... а дальше все равно!
      ).as(Config.class)
  }
}
```

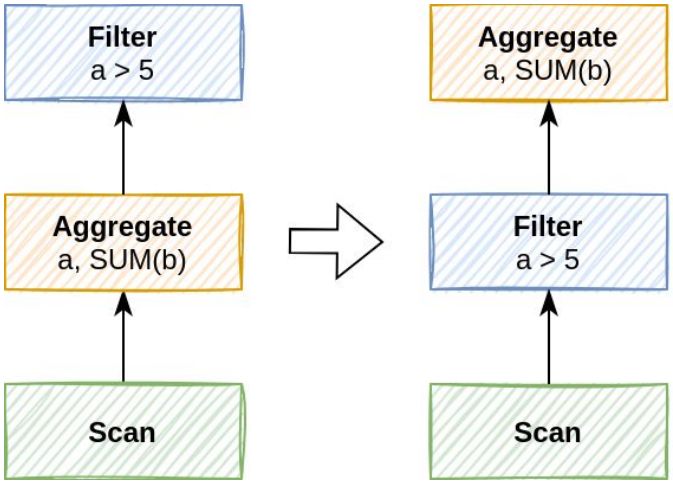
Правила: упрощение операторов

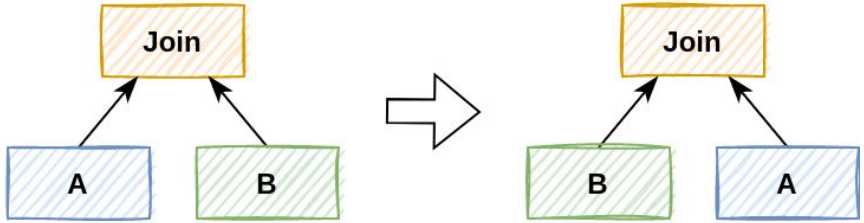




- Упрощение операторов:
  - Определение, что оператор всегда возвращает пустой результат (предикат) [25].
  - Удаление ненужных операций (напр. GROUP BY над уникальной колонкой) [26].

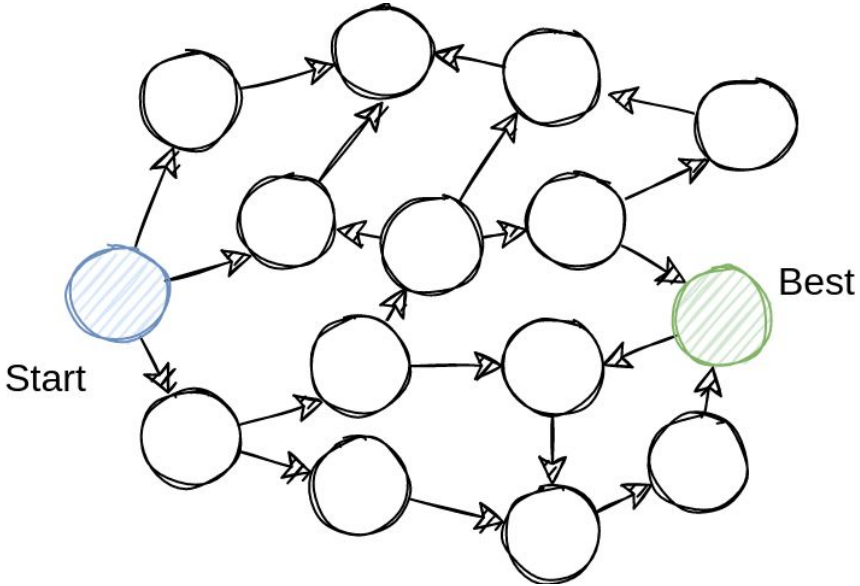
Правила: перемещение операторов





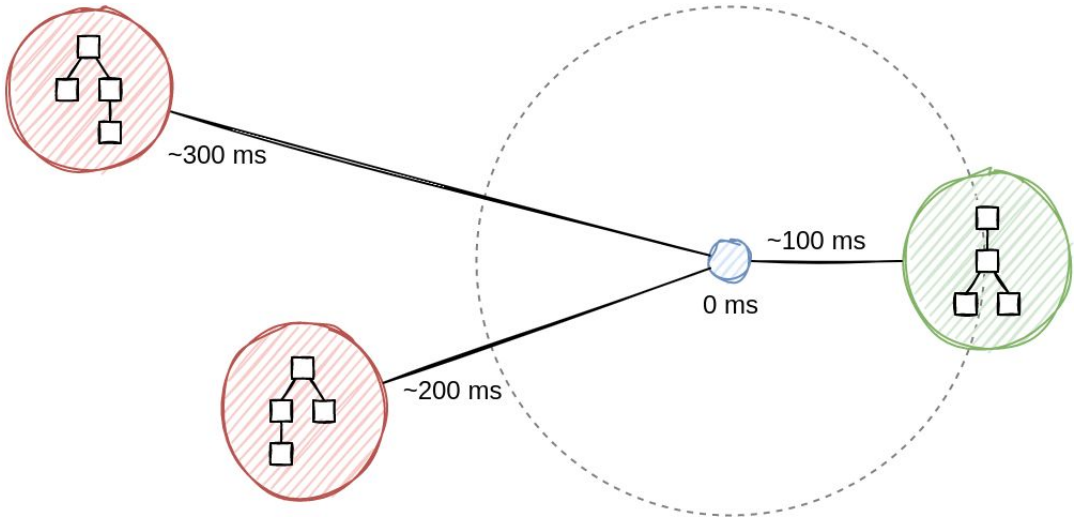
- Перемещение операторов:
  - Filter: pushdown [27], pull-up.
  - Aggregate pushdown (напр. поместить GROUP BY под Join [28]).
  - Join: commute [29], associate [30].

Что делать с правилами?



- Применяя правила, мы генерируем **эквивалентные** планы. Два плана называются эквивалентными, если они порождают одинаковые результаты на любых входных данных (*NB: без учета сортировки!*)
- Задача оптимизатора - попытаться найти оптимальный план. Проблемы:
  - Как понять, что план оптимален?
  - Как “добраться” до оптимального плана?

Что такое “стоимость”?



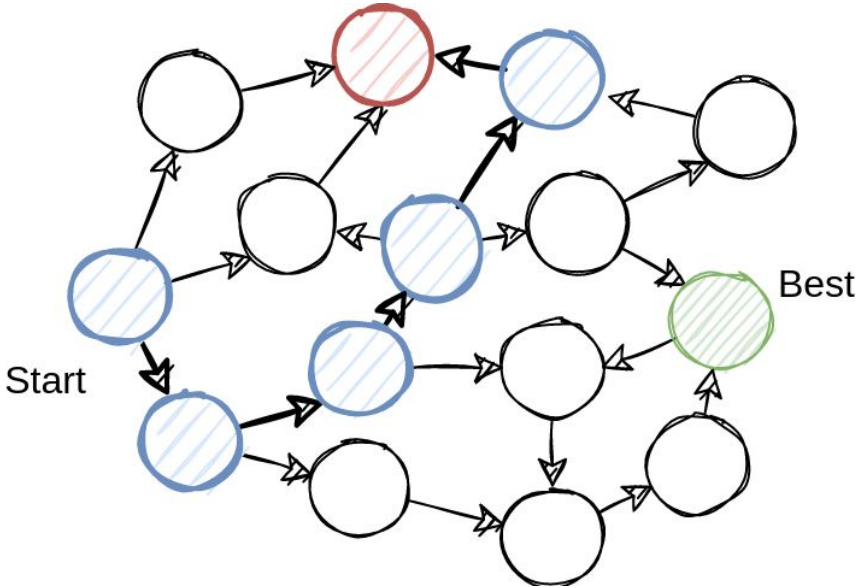
- Условная величина, которая позволяет сравнивать операторы друг с другом **[BLOG4]**.

- **Теория**:
  - Точная оценка затрат ресурсов на выполнение оператора.
- **Практика**:
  - Грубая модель, полагающаяся на неточные статистики, упрощения и магические константы [10] [11].
  - Скаляр [12].
  - ... или вектор [13] [14], который все равно сравнивается как скаляр [15] [16].

RelOptCostImpl

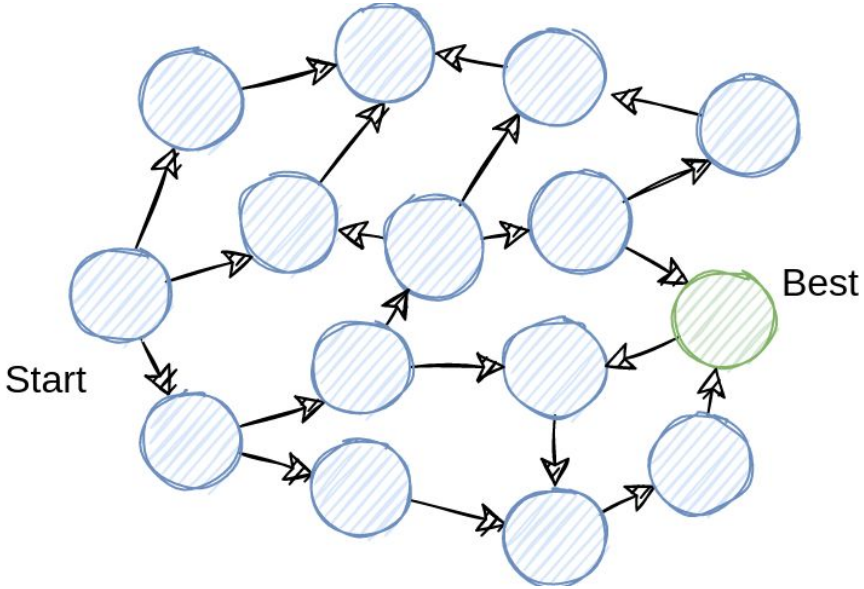
```
class RelOptCostImpl implements RelOptCost {

    private final double value;

    public boolean isLe(RelOptCost other) { ... }
    ...
}
```

Итеративный планировщик



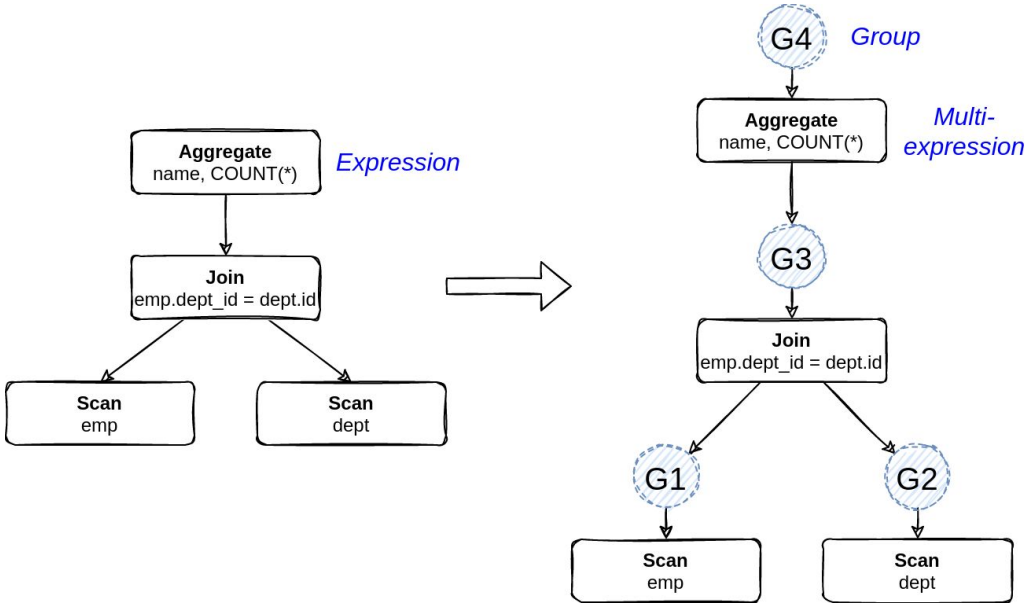
- Рассматривает один план в каждый момент времени, последовательно применяя правила.
- Быстрый, простой, но не гарантирует оптимальность и подвержен зацикливанию.
- **Примеры**: Apache Spark [21], Presto [22], Apache Calcite (Apache Flink, Apache Hive) [23]

Cost-based планировщик



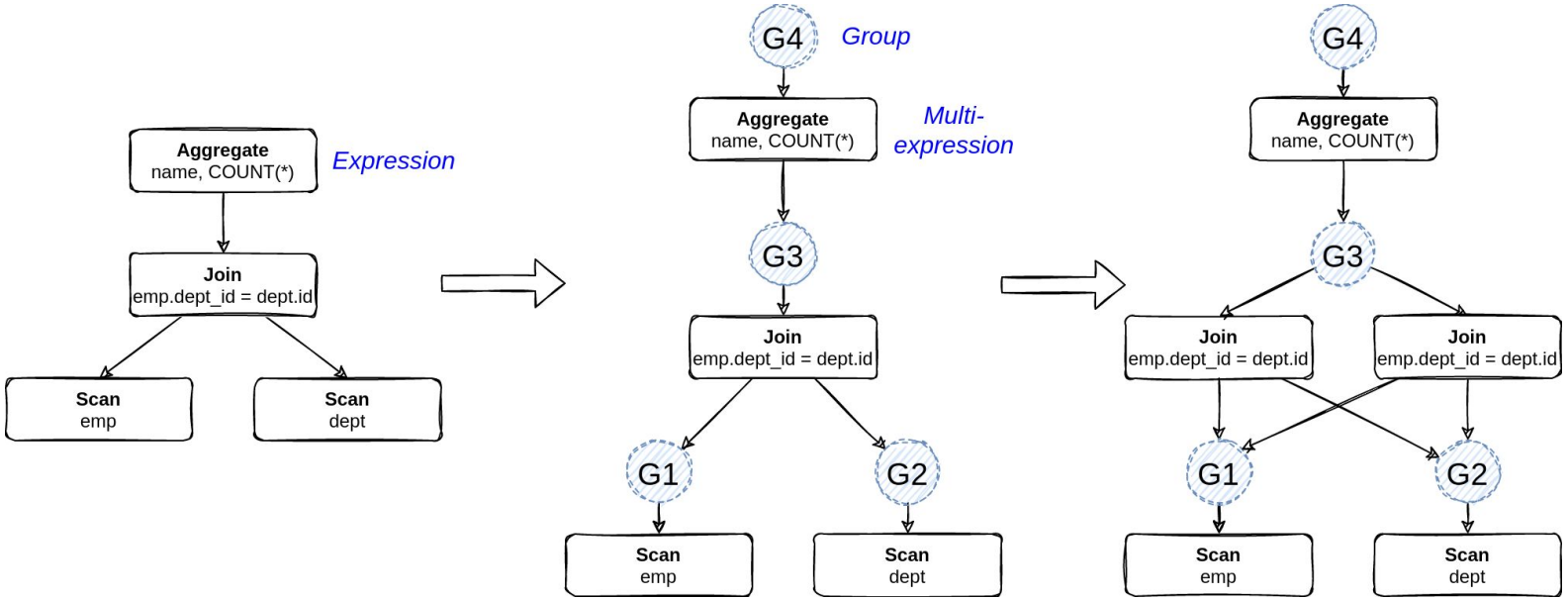
- Рассматривает много планов **одновременно**.
- Выбирает наиболее оптимальный на основе стоимости.
- **Примеры**: Apache Calcite [24], Pivotal Greenplum/Orca [31], CockroachDB [32].

Memoization



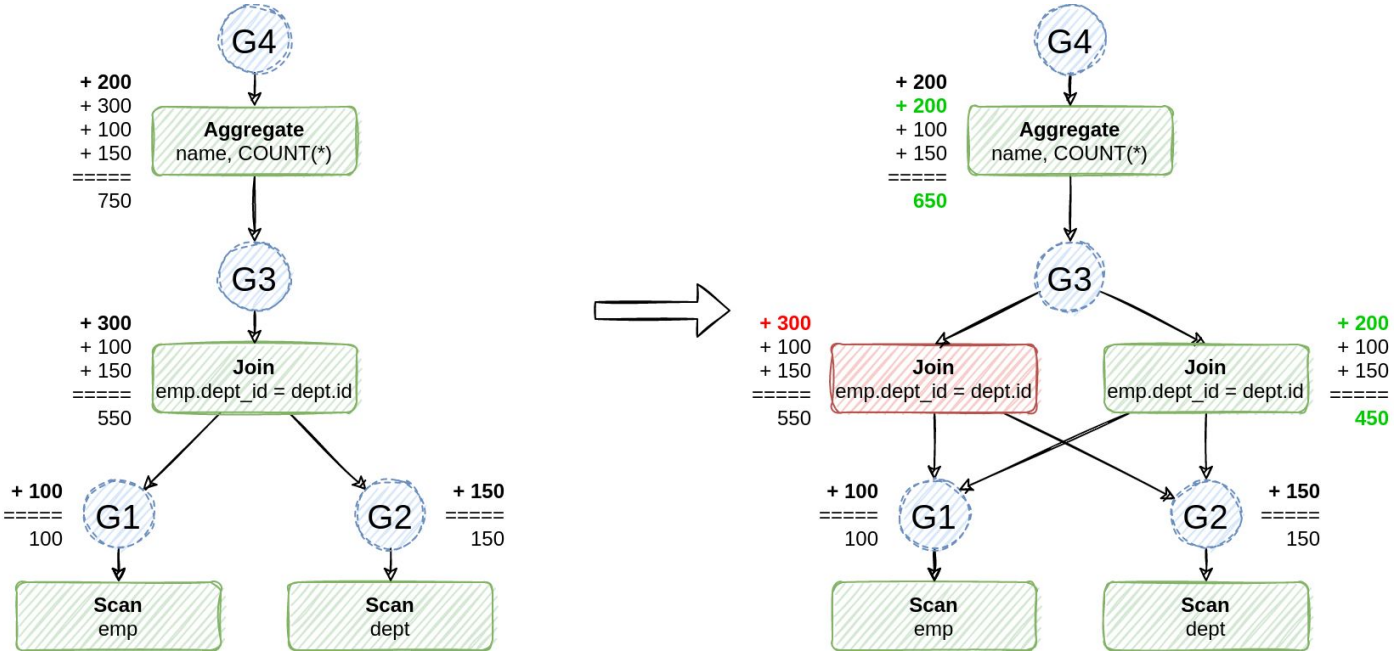
- Expression - оператор, входами которого являются другие операторы.
- Group - набор эквивалентных операторов.
- Multi-expression - оператор, входами которого являются **группы**.
- MEMO - набор groups и multi-expressions.

Memoization



- Позволяет **компактно** кодировать большое количество планов.
- **Дедуплицирует** одинаковые узлы.
- Подробнее: **[BLOG5]**

Cost-based оптимизация



- У каждого оператора есть стоимость: сумма **собственной** стоимости и стоимости входов.
- Если в группе появляется оператор с меньшей стоимостью, он объявляется **winner**, стоимость верхних операторов пересчитывается.

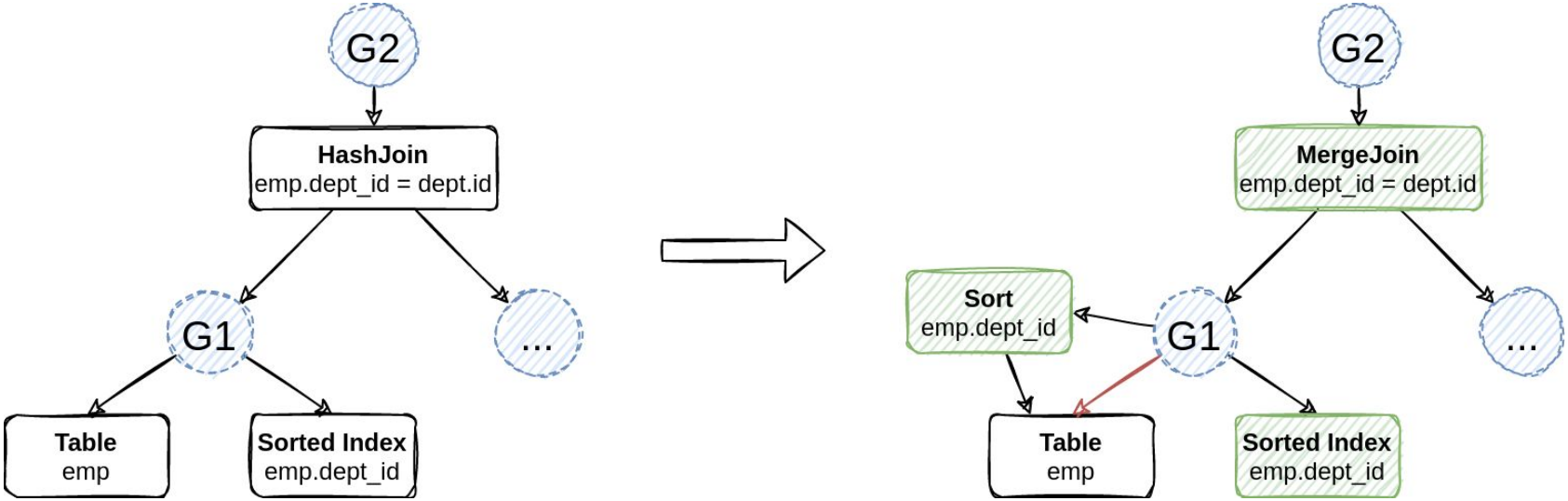
Оптимизаторы в Apache Calcite

- **HepPlanner** - итеративный оптимизатор, с итерациями и циклами [23].
- **VolcanoPlanner** - cost-based оптимизатор, с MEMO и костами [24].

Продвинутые техники

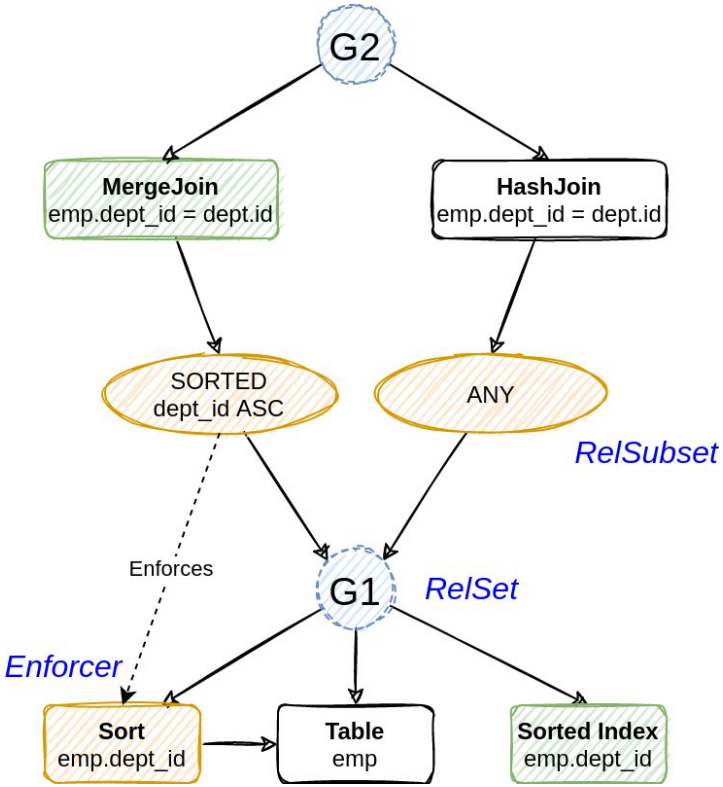
- Физические свойства
- Cascades
- Многофазная оптимизация

Физические свойства



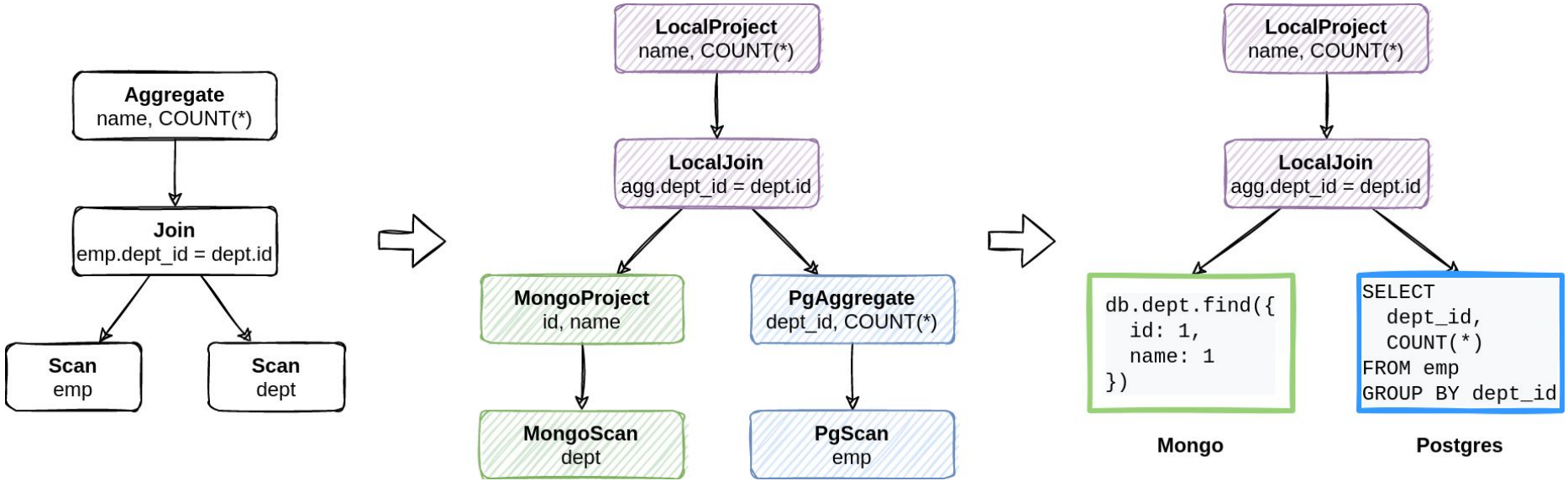
- Оператору MergeJoin необходимо, чтобы его входы были **отсортированы** в соответствии с условием.
- Как сделать так, чтобы MergeJoin учитывал только подходящие операторы?
- Если оператор не подходит, то можно ли его как-то преобразовать, что бы он стал подходящим?

Физические свойства



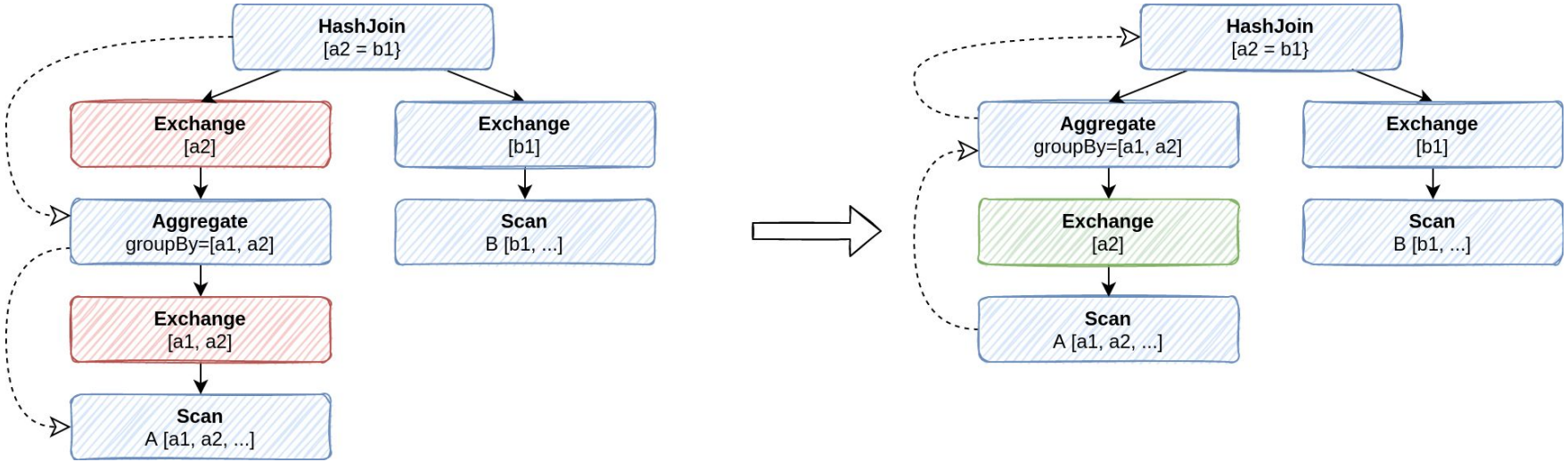
- **Физическое свойство** оператора - величина, которая не влияет на набор возвращаемых записей, но которая может оказывать влияние на другие операторы (RelTrait [33]).
- Оператор может потребовать, чтобы его входы удовлетворяли определенным свойствам (RelSubset [34]).
- Если оператор не удовлетворяет требованиям, можно автоматически вставить **enforcer**, которые преобразует физические свойства оператора.
- Свойства (и их enforcer-ы) в Apache Calcite **[BLOG6]**:
  - Collation (Sort) - сортировка по атрибутам.
  - Distribution (Exchange) - распределение данных в кластере (sharded, replicated, singleton)
  - Convention (???) - маркер, указывающий, в какой системе исполняется оператор.

Convention



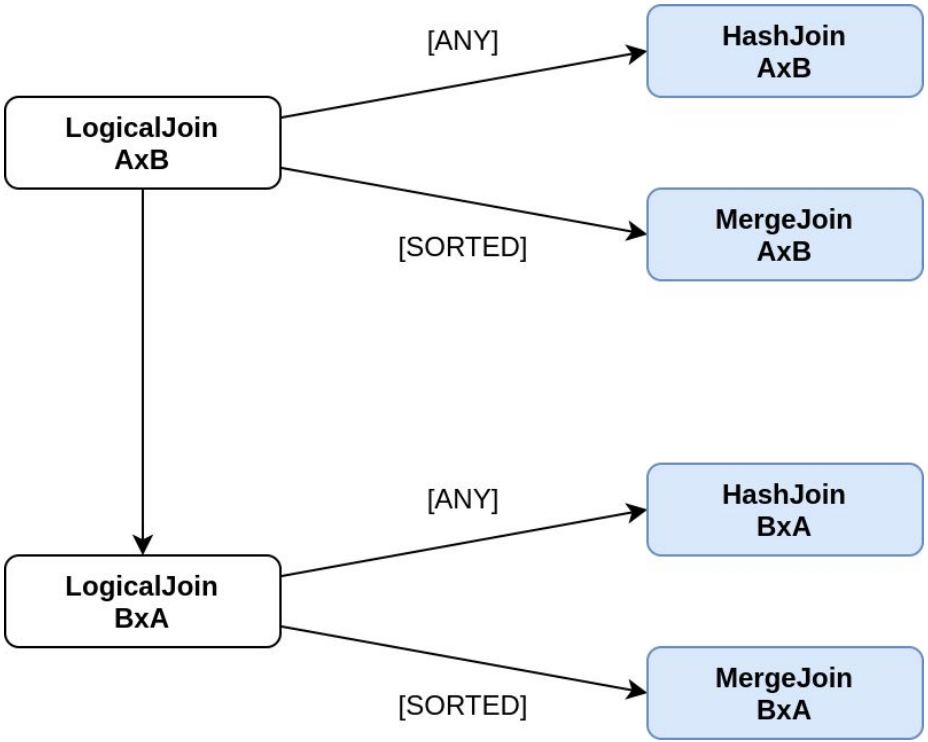
- Каждому оператору присваивается **convention** [35], т.е. целевая система выполнения.
- При переходе между convention вставляются специальные операторы-конвертеры [36] (не показано).
- Позволяет строить federated движки.

Управление свойствами



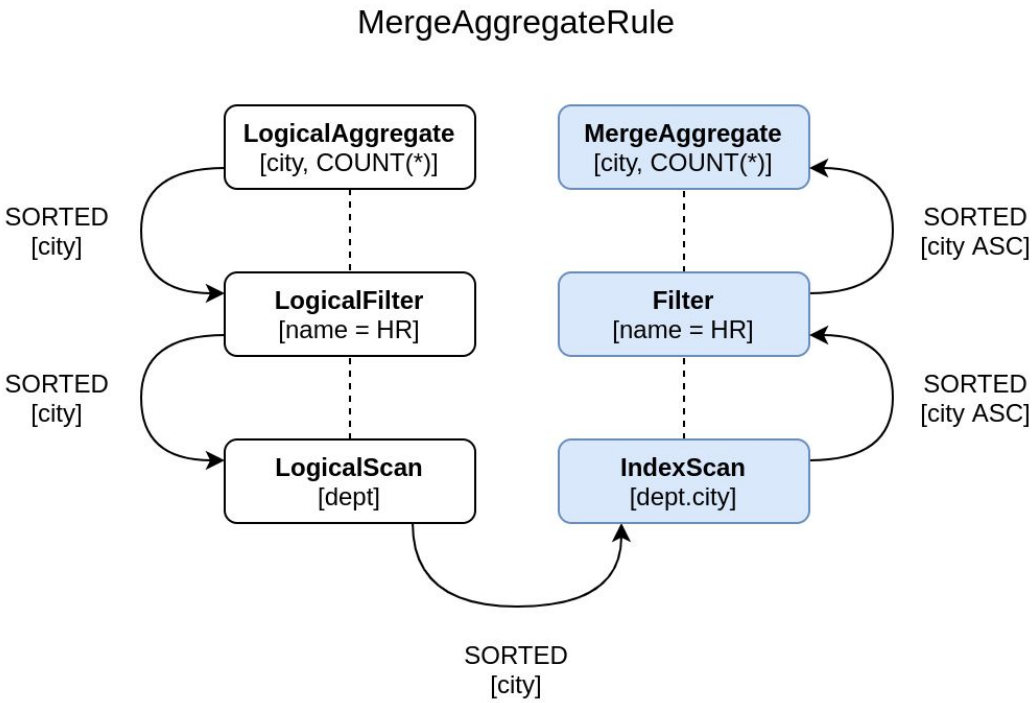
- Продвинутые оптимизаторы пытаются найти оптимальную **комбинацию операторов** на основе их свойств.
  - Упрощенные алгоритмы: Apache Flink [39].
  - Алгоритм **Cascades** [41] [42] [43]: Apache Calcite [40], Pivotal Greenplum [31].
- Катастрофическая комбинаторная сложность!

Cascades: типы операторов и правил



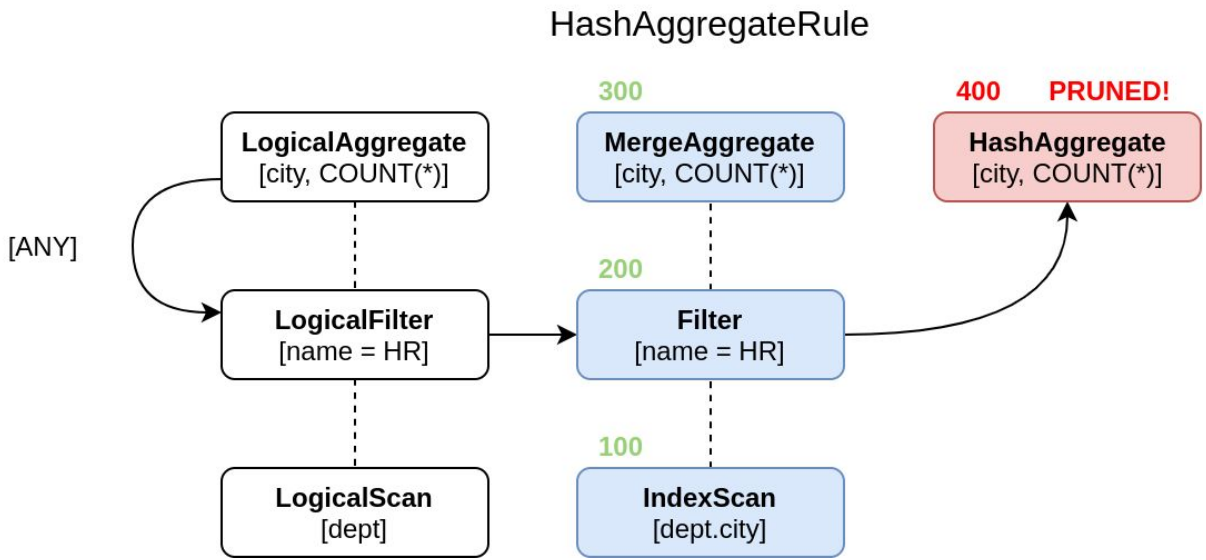
- Типы операторов
  - Logical - не имеют свойств.
  - Physical - имеют свойства.
- Трансформации:
  - Logical-Logical (exploration) - перебираем все альтернативы, так как они могут породить более оптимальные физические имплементации.
  - Logical-Physical (implementation) - перебираем только те, что запрошены.

Cascades: оптимизационные запросы



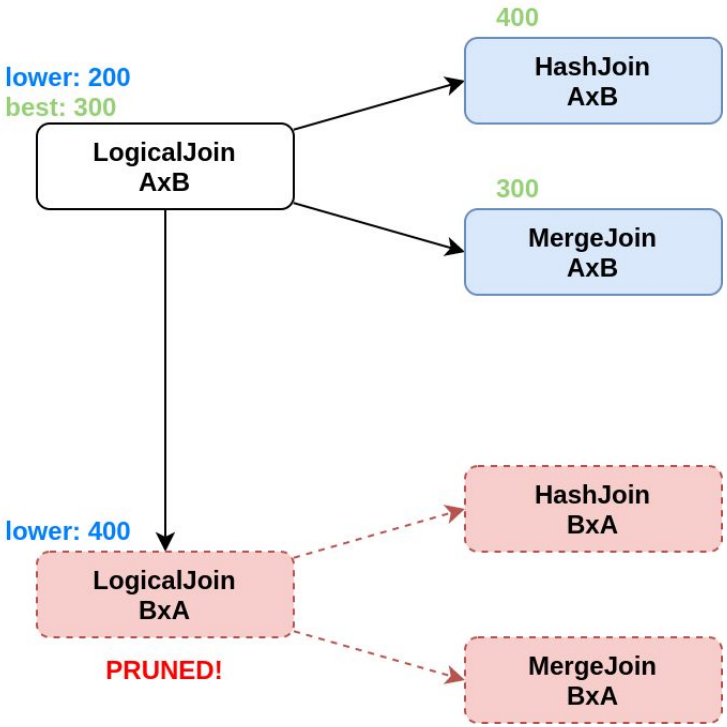
- Физический оператор “отправляет” оптимизационный запрос вниз.
- Дочерние операторы ищут способ удовлетворить этот запрос.
- Следствие: перебираем только те физические операторы, которые кому-то нужны.
- В Apache Calcite: TopDownRuleDriver [55].

Cascades: accumulated cost bounding



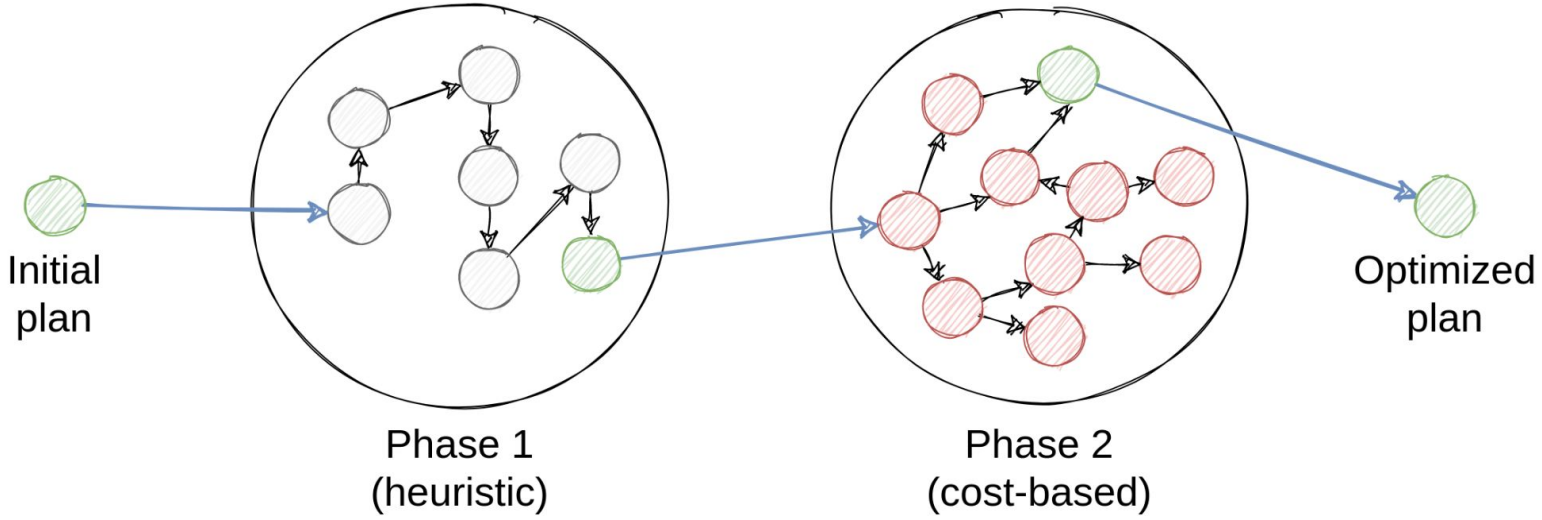
- Правило хочет сделать физический оператор HashAggregate.
- Но у нас уже есть более дешевый MergeAggregate, поэтому игнорируем HashAggregate.
- Следствие: игнорируем заведомо дорогие физические операторы.
- Работает на **практике**.

Cascades: predicted cost bounding



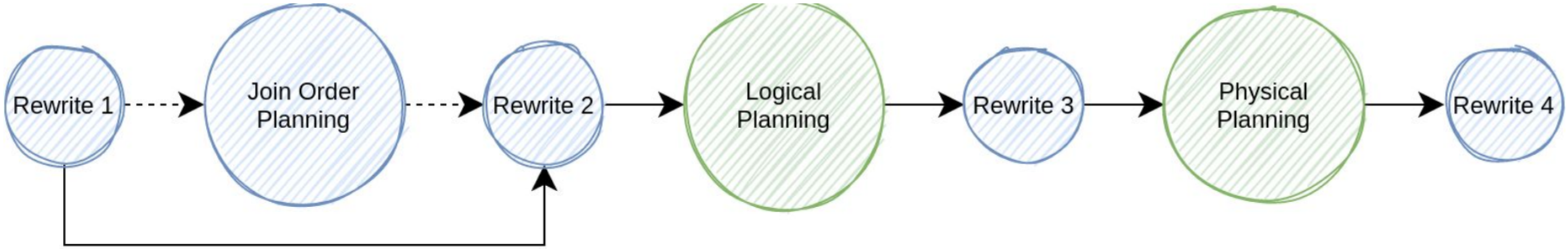
- **Lower bound** - все физические операторы, созданные из данного логического оператора будут иметь стоимость не меньше данной.
- Можно игнорировать логические операторы, у которых lower bound выше, чем стоимость лучшего физического оператора.
- Пример:
  - Для (AxB) уже найден физический оператор со стоимостью 300.
  - Для (BxA) все физические операторы заведомо имеют стоимость выше 400.
  - Из (BxA) никогда не получится оптимальный план.
- Следствие: игнорируем заведомо дорогие **группы** операторов.
- Работает в теории.
- На практике - непонятно, как посчитать lower bound.

Многофазное планирование



- Задача нахождения оптимального плана является **NP-сложной**, и в общем случае не решается за адекватное время. Поэтому оптимизаторы обычно разбивают планирование на несколько последовательных этапов.
- Примеры: Apache Flink [44] [45], Apache Spark [46], Dremio [47].

Многофазное планирование в Apache Flink



- Фазы [44]:
  - **Rewrite 1** - subquery unnesting, simplification, predicate pushdown, …
  - **Join Order Planning** - эвристическое планирование порядка join-ов (отключено по умолчанию).
  - **Rewrite 2** - transient predicates pushdown, упрощение Join, …
  - **Logical Planning** - cost-based, планирование порядка операторов aggregate pushdown, etc).
  - **Rewrite 3** - дальнейшее упрощение операторов.
  - **Physical Planning** - cost-based, выбор алгоритмов для операторов.
  - **Rewrite 4** - эвристическое добавление локальных агрегатов.
- “Join Order Planning” + “Logical Planning” + “Physical Planning” не гарантируют оптимальный план!

Что осталось за кадром?

- **Subquery unnesting/decorrelation** [48] [49] [50] - мощная техника, которая позволяет переписывать произвольные подзапросы (скалярные, коррелированные) на последовательность операторов Join/Aggregate.
- **Metadata [BLOG1]** [51] - инфраструктура для обмена метаданным между операторами для оценки их стоимости: кардинальности, селективности, и т.п..
- **Enumerable** [52] [53] - встроенный движок выполнения запросов с помощью кодогенерации (Janino). Не подойдет для серьезных проектов, но на “поиграть” - отлично!
- **Avatica** [54] - фреймворк для построения JDBC драйверов.

Итого

- Apache Calcite - платформа для построения SQL-движков.
  - Содержит парсер, семантический анализатор, оптимизатор.
- Оптимизатор:
  - Основной подход: rule-based оптимизация.
  - Планировщики: (1) итеративный, (2) cost-based.
- Практически безальтернативный выбор для новых SQL-движков.
  - Недели до прототипа.
  - Месяцы до MVP.
  - Нет предела совершенству.
- Ссылки: https://querifylabs.notion.site/Apache-Calcite-SQL-Java-be85ad4f6b5d41c48a484ad3f57d1e22
- Поработать с Apache Calcite: https://career.habr.com/companies/querify-labs/vacancies