Querify Labs

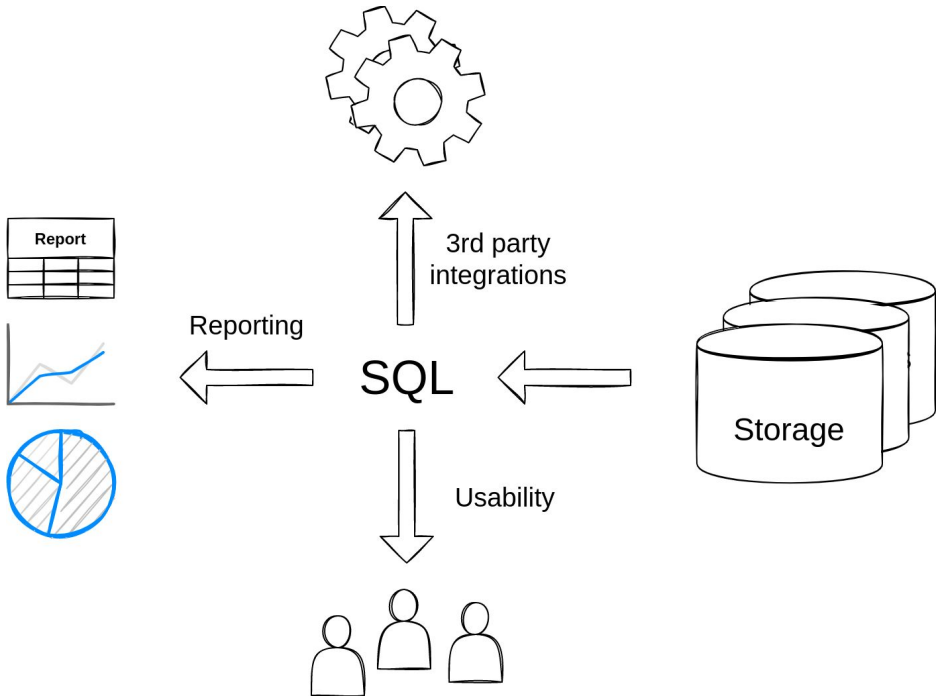
Архитектура Высокопроизводительных Распределенных SQL-движков

Алексей Гончарук, Владимир Озеров
Querify Labs

О нас

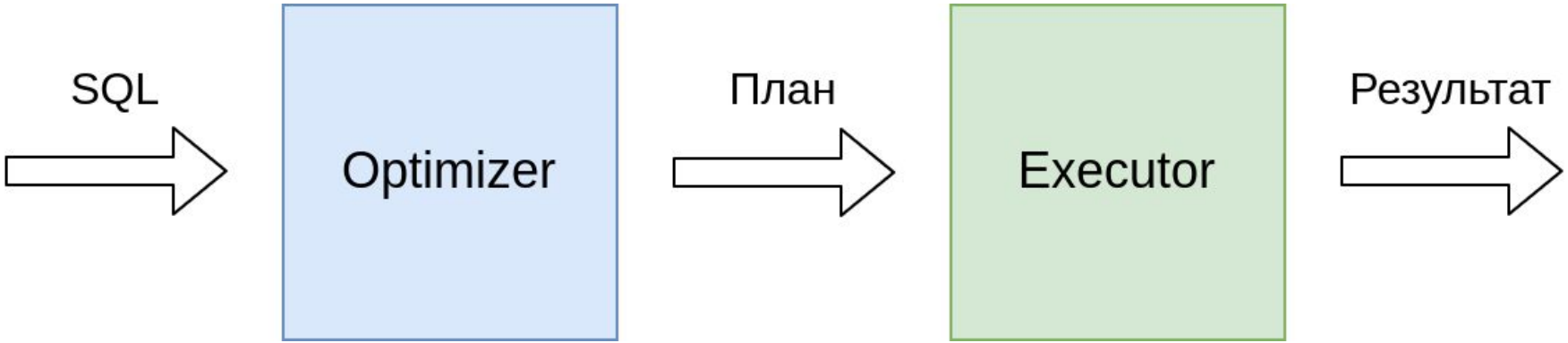
- Компания Querify Labs [1]
  - Разработка новых СУБД и data management систем: движки, оптимизаторы, storage, протоколы.
- Работали много лет над распределенными in-memory движками:
  - Apache Ignite
  - Hazelcast
- Контрибьюторы в Apache Calcite [2] и Apache Ignite

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

Архитектура

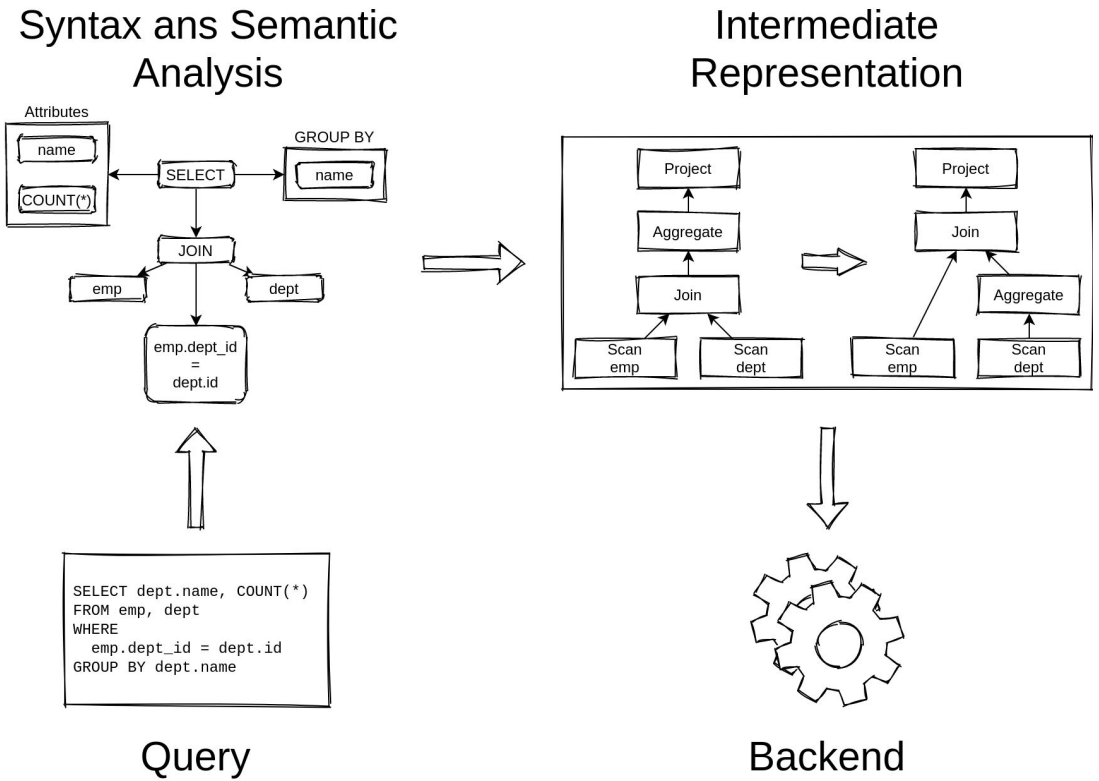


- **Optimizer** - находит (предположительно) оптимальный план выполнения запроса с учетом распределения данных в кластере.
- **Executor** - выполняет план, дополняет и корректирует решения оптимизатора.

Основы планирования запросов

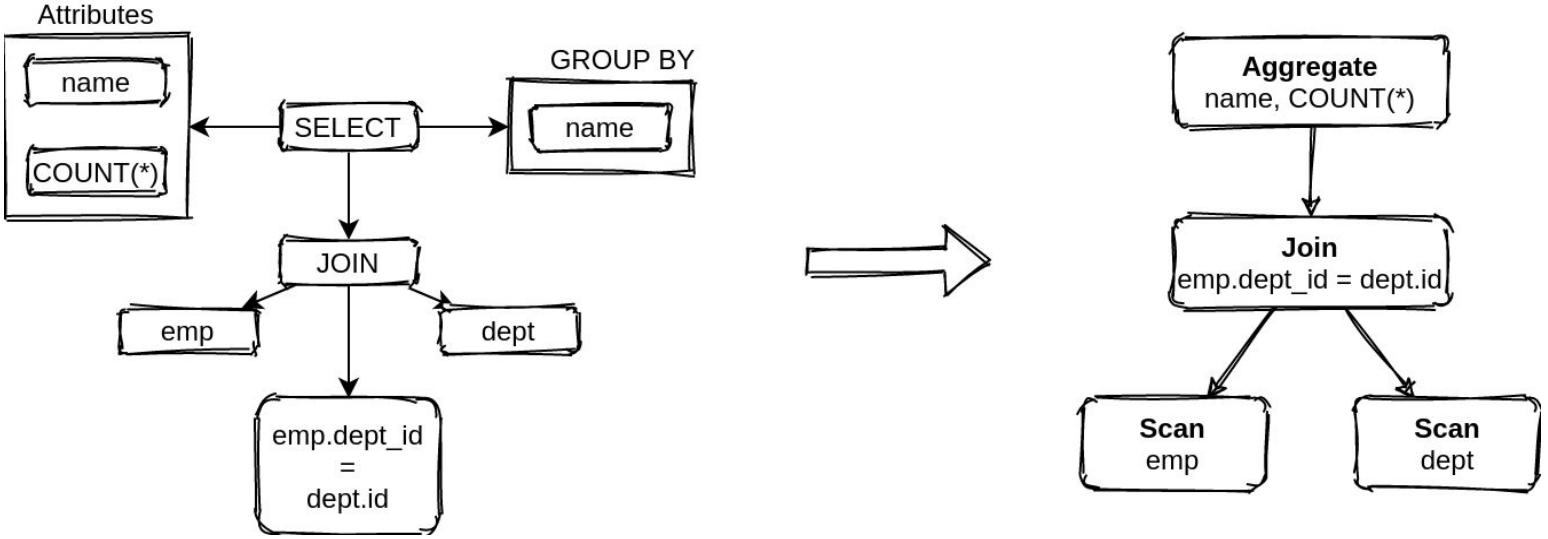
- Промежуточное представление (IR)
- Rule-based оптимизация
- Итеративное и cost-based планирование
- Примеры правил

Планирование



- SQL - декларативный язык.
- Запрос может быть исполнен множеством способов.
- Задача планировщика - найти оптимальный план.
- “Оптимальность” зависит от задачи:
  - Скорость
  - Деньги

Intermediate Representation

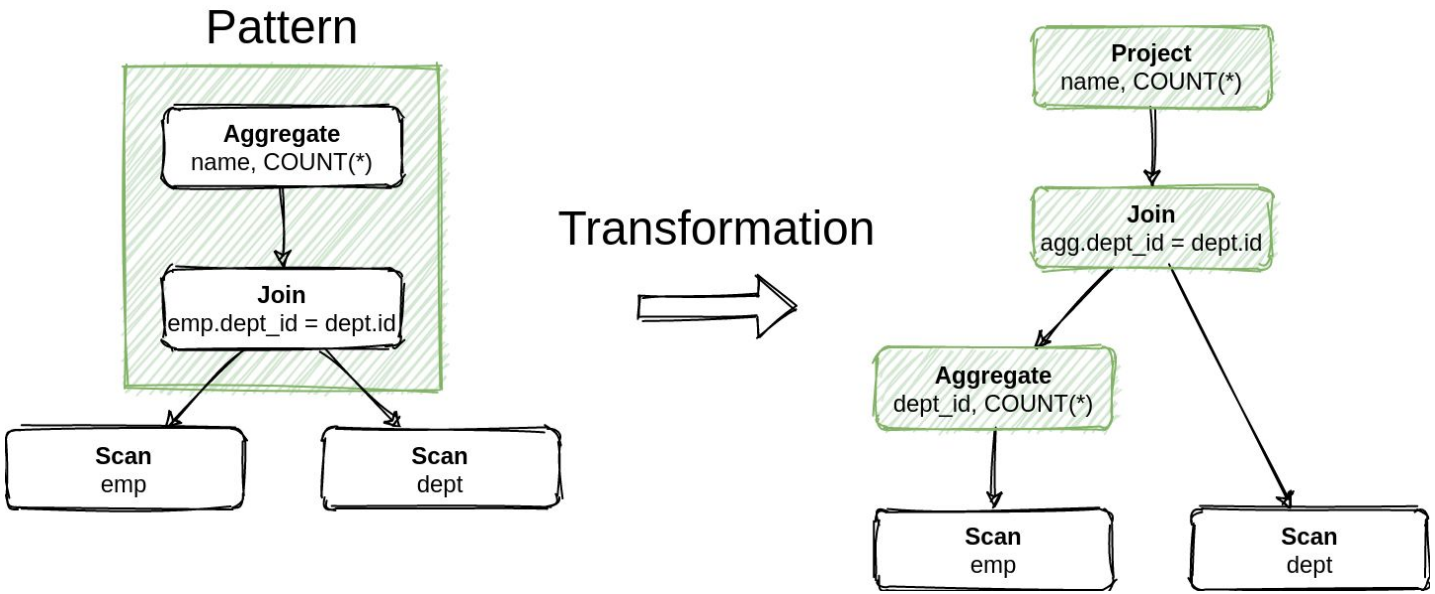


- Оптимизировать синтаксическое дерево проблематично из-за высокой сложности SQL-синтаксиса.
- Удобнее представить запрос в виде дерева (относительно) простых реляционных операторов.
- Реляционные операторы - типичное промежуточное представление (IR) во многих движках:
  - Apache Calcite (Apache Flink, Apache Hive, Dremio) [3], Apache Spark [4], Presto [5], Greenplum, CockroachDB, ...
- Правильный дизайн IR критически важен для раскрытия возможностей оптимизатора
  - Пример: Apache Calcite с большим трудом планирует порядок Join из-за особенностей его IR [6].

Intermediate Representation

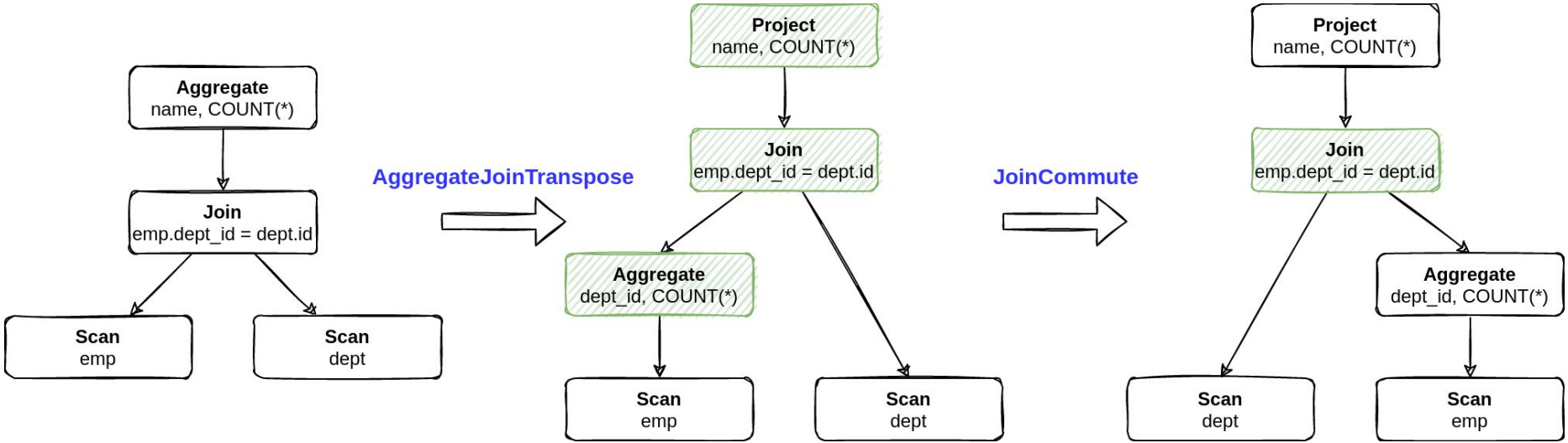
Оператор	**Назначение**
Scan	Сканирование источника данных
Project	Трансформация атрибутов
Filter	WHERE, HAVING
Sort	ORDER BY
Aggregate	GROUP BY
Window	Window-функции
Join	Соединение двух источников
Union/Minus/Intersect	SET-операторы

Rule-based трансформации



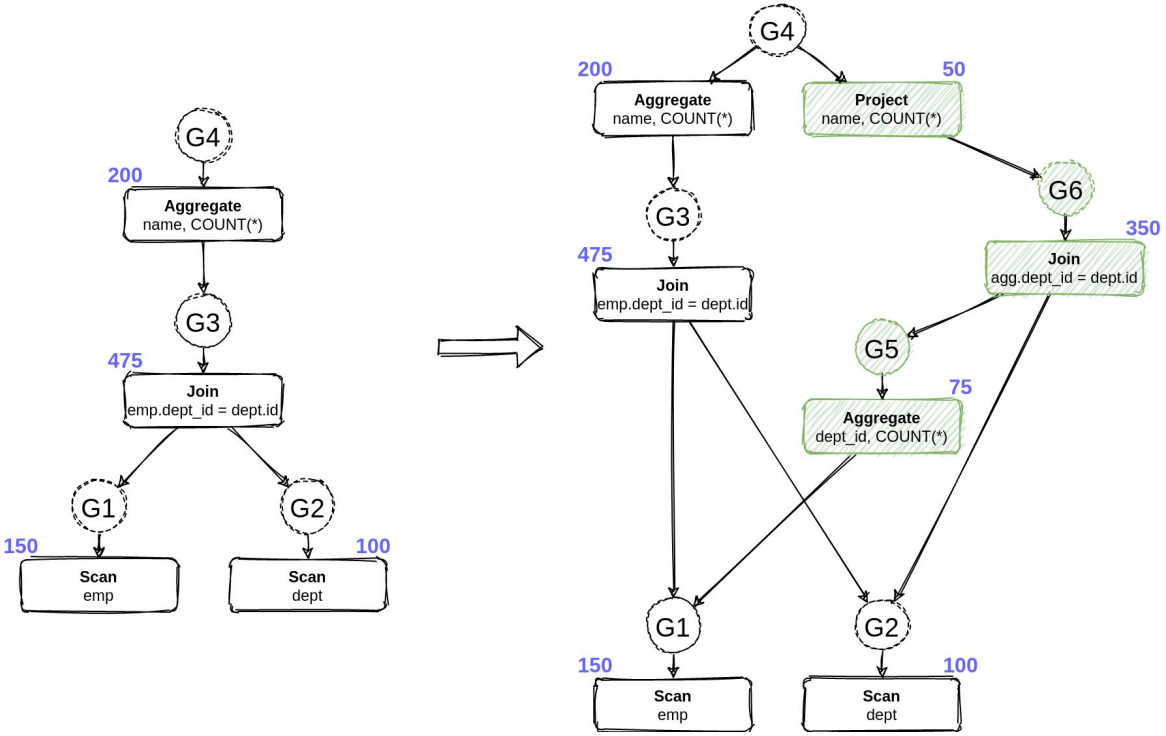
- Правило - это единица оптимизации, состоящая из паттерна и логики трансформации.
- Значительно упрощает эволюцию и поддержку планировщика за счет декомпозиции.
- Распределенные SQL-движки могут содержать сотни трансформаций.
- **Примеры**: Apache Calcite (Apache Flink, Apache Hive) [7], Apache Spark [8], Presto [9], CockroachDB [13].

Итеративное планирование



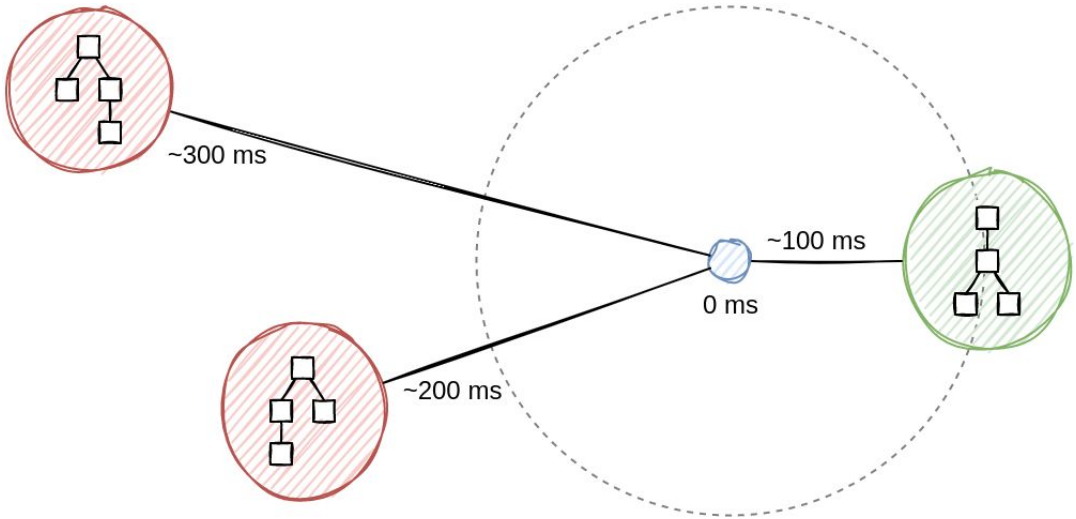
- Последовательно применяет правила к плану.
- Рассматривает один план в каждый момент времени.
- Быстрый, простой, но не может гарантировать оптимальность.
- **Примеры**: Apache Spark [10], Presto [11], Apache Calcite (Apache Flink, Apache Hive) [12]

Cost-based планирование



- Планировщик рассматривает несколько планов запросов **одновременно**. Для этого операторы, возвращающие идентичные результаты, организуют в группы эквивалентности.
- Планировщик использует cost-модель для назначения **стоимости** операторам.
- В общем случае, рассматривает большее количество планов, чем итеративные планировщики.
- **Примеры**: Apache Calcite [14], Greenplum [15], CockroachDB [16].

Что такое “стоимость”?



- Условная величина, которая позволяет сравнивать операторы друг с другом.

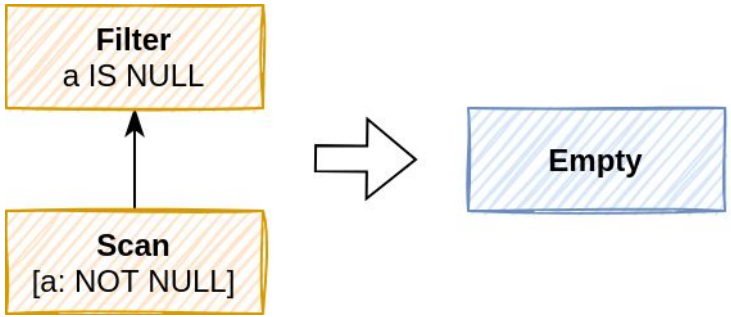
- **Теория**:
  - Точная оценка затрат “железных” ресурсов на выполнение оператора.
- **Практика**:
  - Грубая модель, полагающаяся на неточные статистики, магические константы и “хаки” [17] [18].
  - Скаляр [19]. Или вектор [20] [21], который все равно сравнивается как скаляр [22] [23].

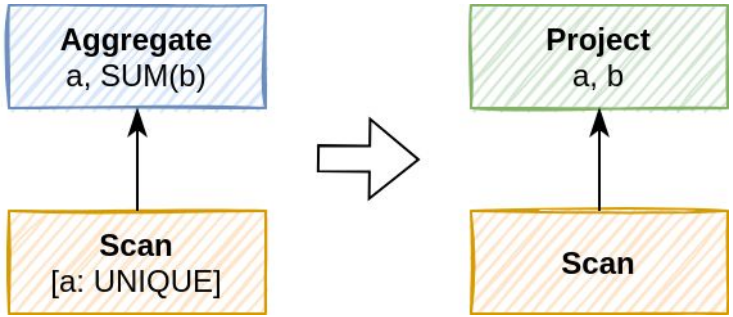
- **Совет**: “доверяй, но проверяй”. Уделяйте внимание статистикам и хинтам.

Типичные трансформации

- Упрощение операторов.
- Перемещение операторов.
- Планирование порядка join-ов.
- Планирование **распределенных операторов.**

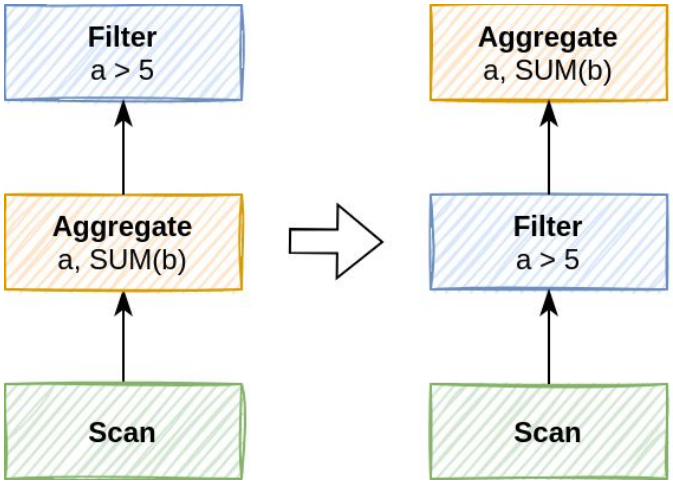
Упрощение операторов

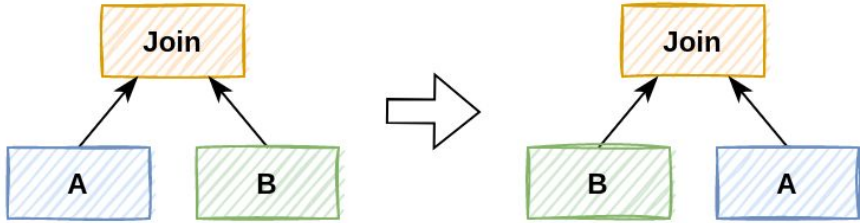




- Упрощение операторов:
  - Определение, что оператор всегда возвращает пустой результат (предикат) [24].
  - Удаление ненужных операций (напр. GROUP BY над уникальной колонкой) [25].

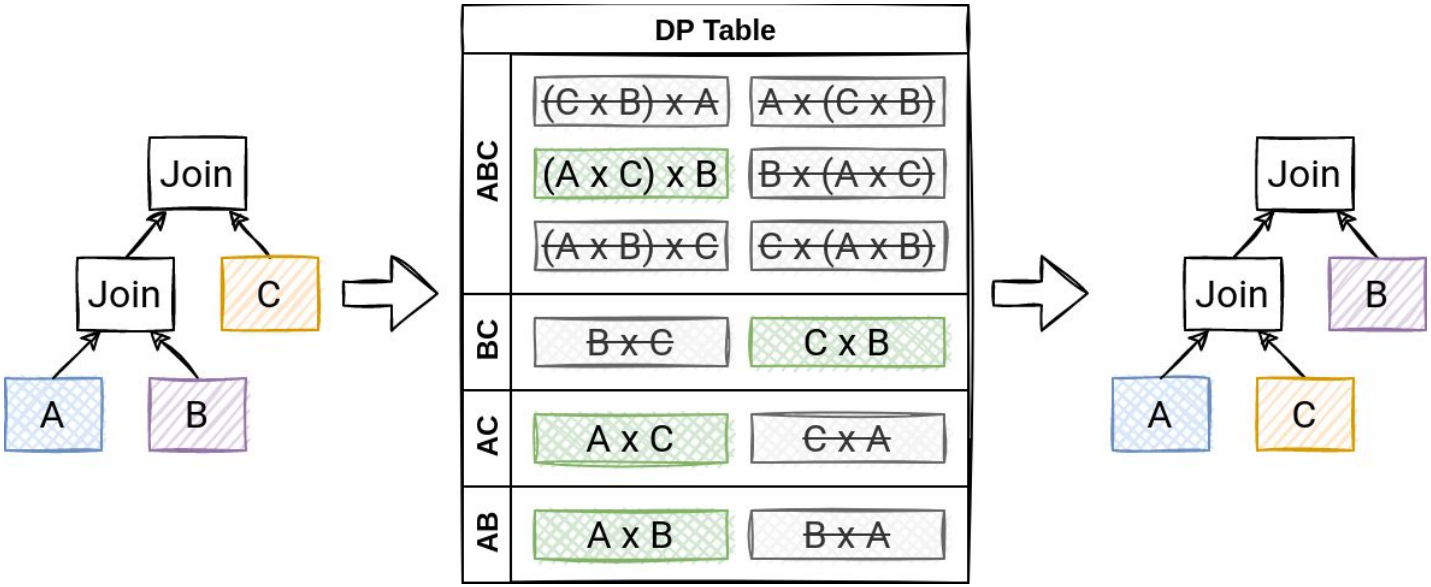
Перемещение операторов





- Перемещение операторов:
  - Filter: pushdown [26], pull-up.
  - Aggregate pushdown (напр. поместить GROUP BY под Join [27]).
  - Join: commute [28], associate [29].

Планирование порядка join-ов

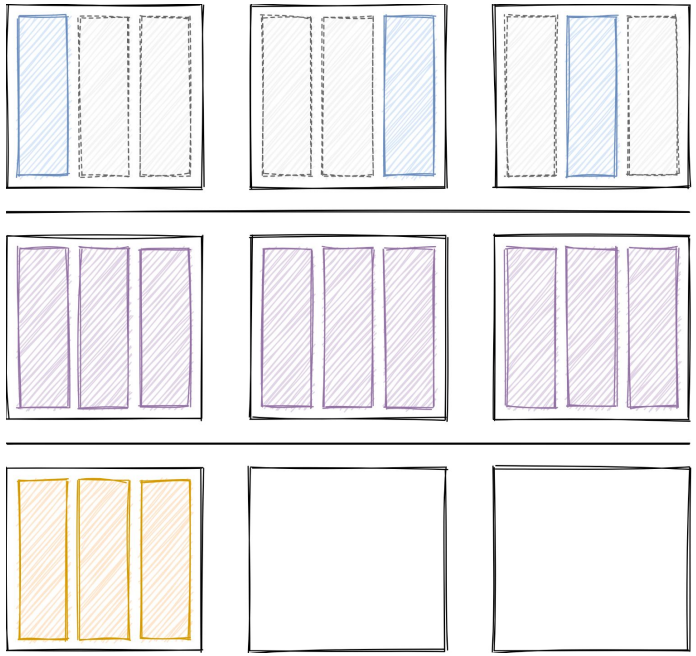


- Выделяем поддерево операторов Join.
- Перебираем допустимые варианты
  - Dynamic programming: теория [30], CockroachDB [31], Presto [32].
  - Top-down enumeration with memoization: теория [33].

Планирование распределенных операторов

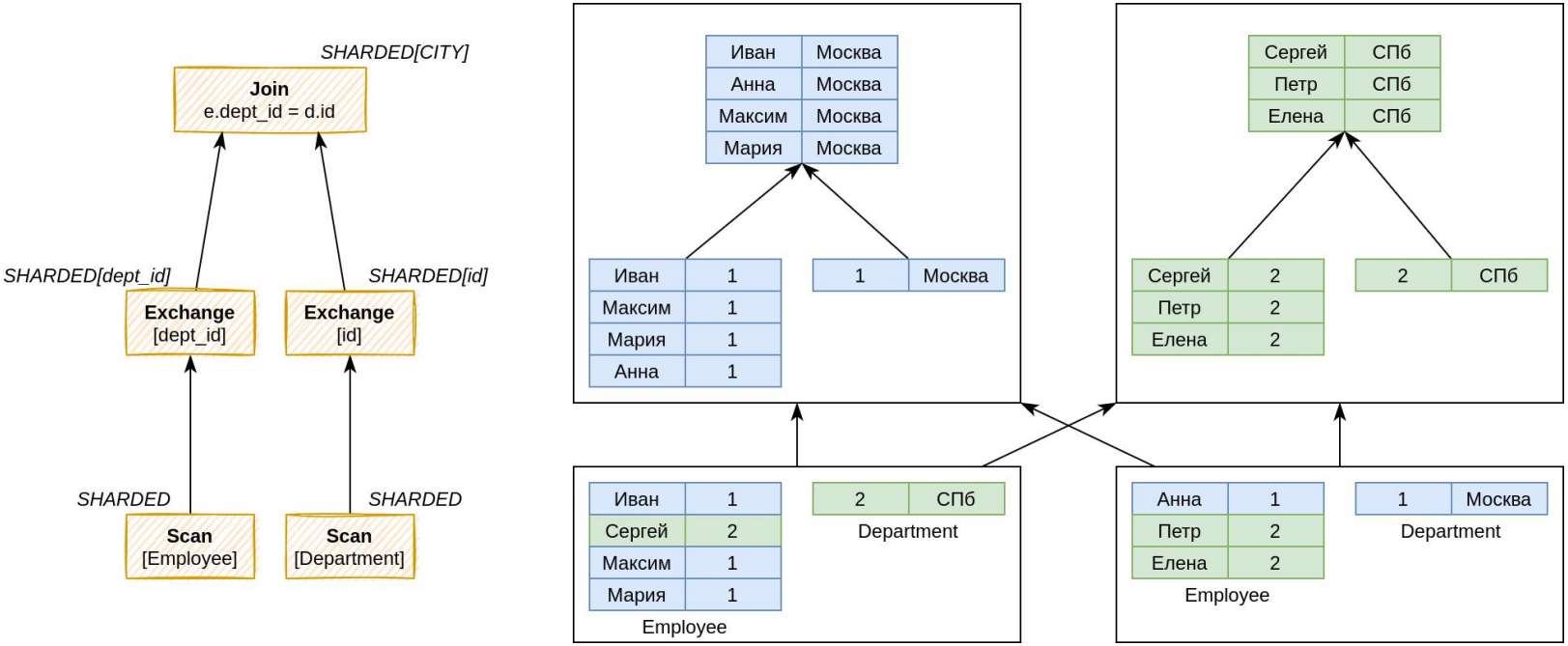
- Ключевые понятия: distribution, exchange.
- Планирование Join.
- Планирование Aggregate.

Распределение данных



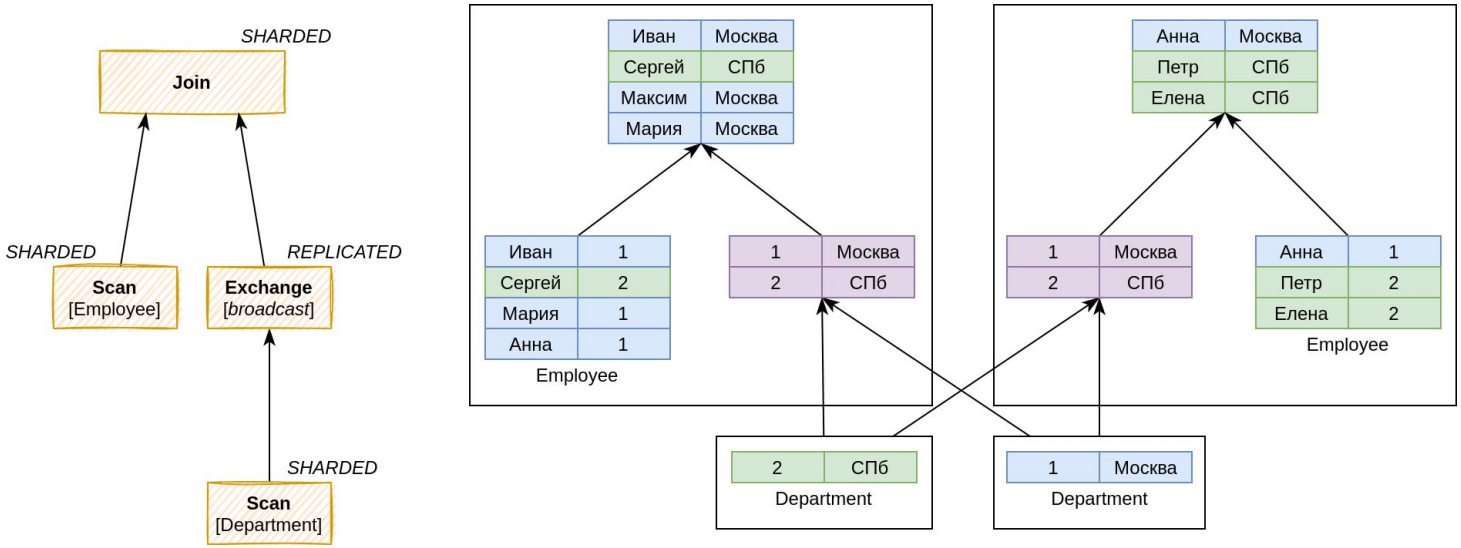
- Оператору присвоено **распределение**:
  - SHARDED - данные распределены по узлам.
  - REPLICATED - полная копия данных на всех узлах.
  - SINGLETON - данные на одном узле.
  - Apache Flink [34], Presto [35].
- Оператор задает **требование** к распределению своих входов.
- На основе этих требований планировщик вставляет операторы **Exchange**, которые моделируют пересылку данных между узлами.
  - Apache Flink [36].
- Обычно есть **несколько** альтернативных способов выполнить распределенный оператор. Поэтому чаще используют cost-based оптимизацию.

Join: unicast



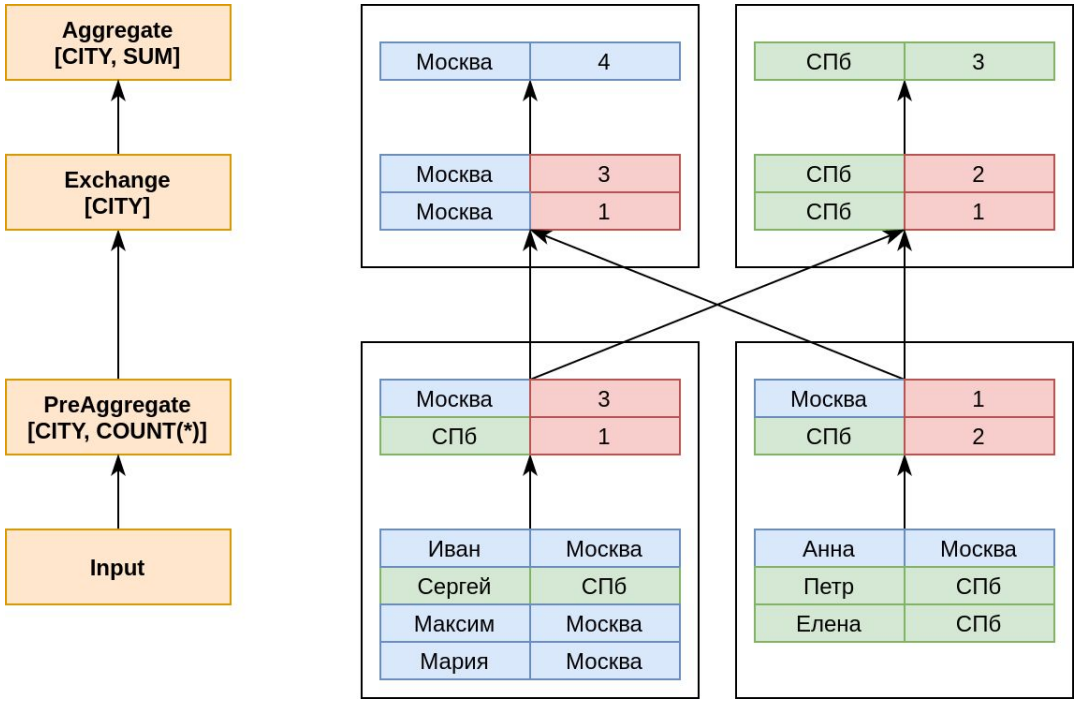
- Для equi-join можно решардировать данные по equi-условию.

Join: broadcast



- Альтернативно, можно форсированно реплицировать один из входов.
  - Выгодно, если реплицируемый вход имеет небольшой размер.
  - Обязательно для outer join и non-equi join.
- Пример:
  - `table.optimizer.join.broadcast-threshold` в Apache Flink [37].

Агрегаты

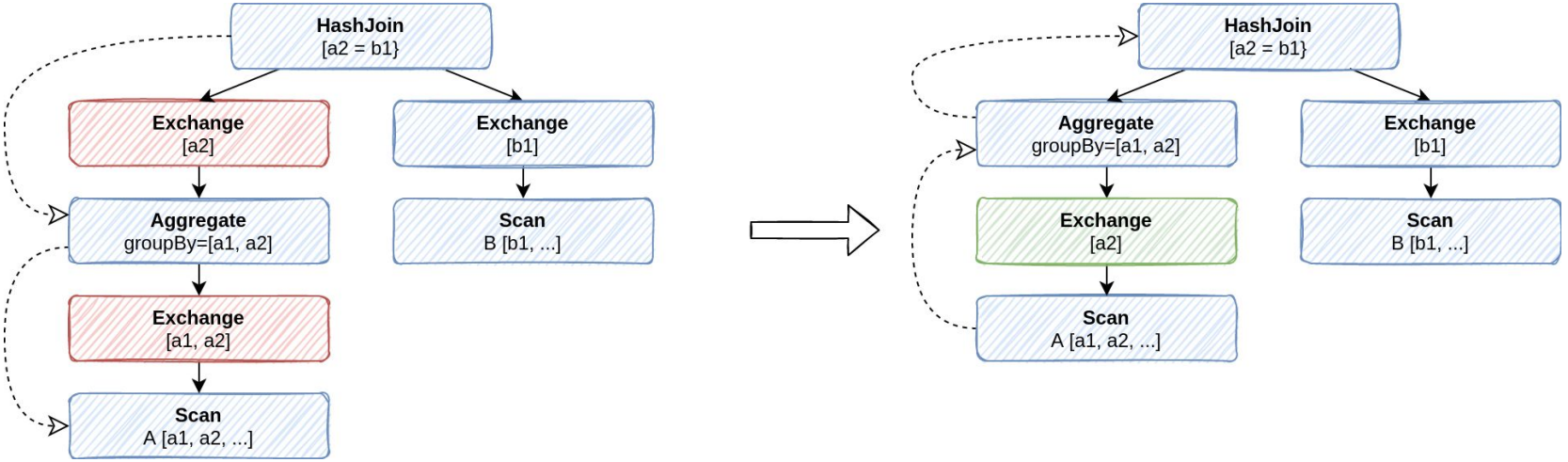


- Если данные шардированы по ключу группировки, то агрегат можно посчитать на каждом узле независимо.
- В противном случае, данные необходимо решардировать.
- Большинство систем делает локальную пре-агрегацию для минимизации количества передаваемых по сети данных.
- **Совет 1**: для ускорения агрегаций шардируйте таблицы по ключу группировки (или части ключа).
- **Совет 2**: убедитесь, что система умеет решардировать данные; ознакомьтесь с ограничениями.

Продвинутые техники

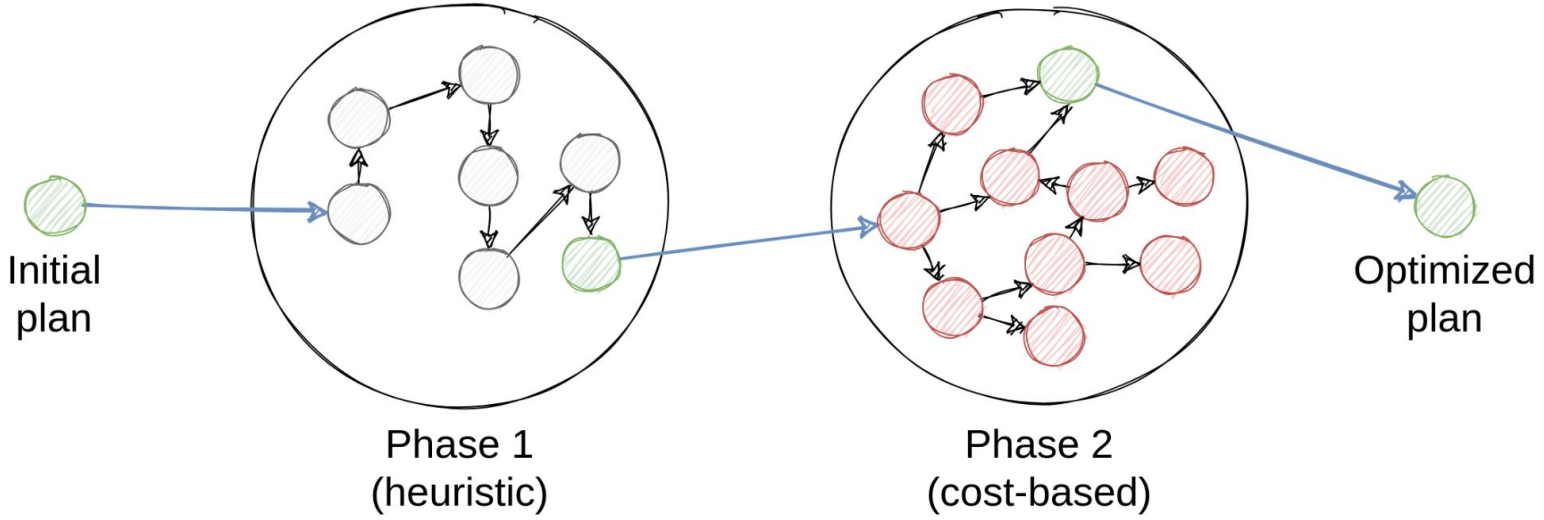
- Планирование физических свойств.
- Многофазное планирование.

Физические свойства



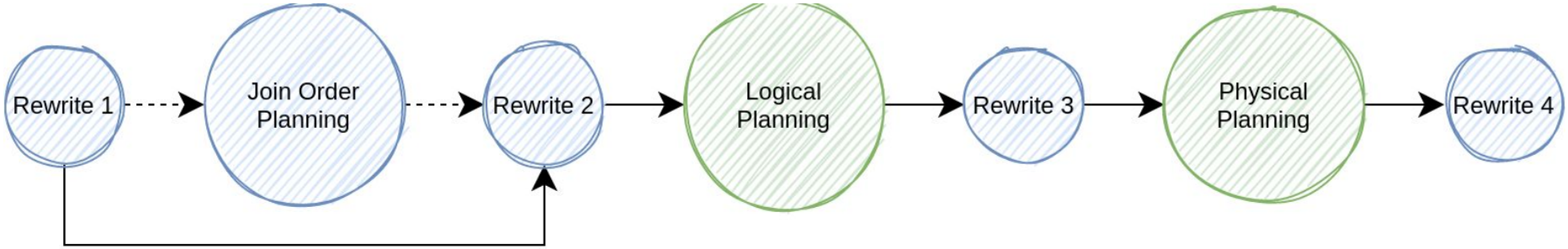
- Наблюдение: Sort и Exchange меняют только **свойства** отношений (сортировка и распределение), но не меняют сами кортежи.
- Продвинутые оптимизаторы пытаются найти оптимальную **комбинацию операторов** на основе их свойств.
  - Упрощенные алгоритмы: Apache Flink [38].
  - Алгоритм **Cascades** [40] [41] [42]: Apache Calcite [39], Pivotal Greenplum [15].
- Катастрофическая комбинаторная сложность!

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.
  - Apache Flink [43] [44], Apache Spark [45], Dremio [46].
- **Совет :** Внимательно ознакомьтесь с хинтами, так как они обычно появляются в тех местах, где ошибки оптимизатора наиболее вероятны (планирование join-ов, выбор индексов, выбор broadcast, и т.д.).

Многофазное планирование в Apache Flink

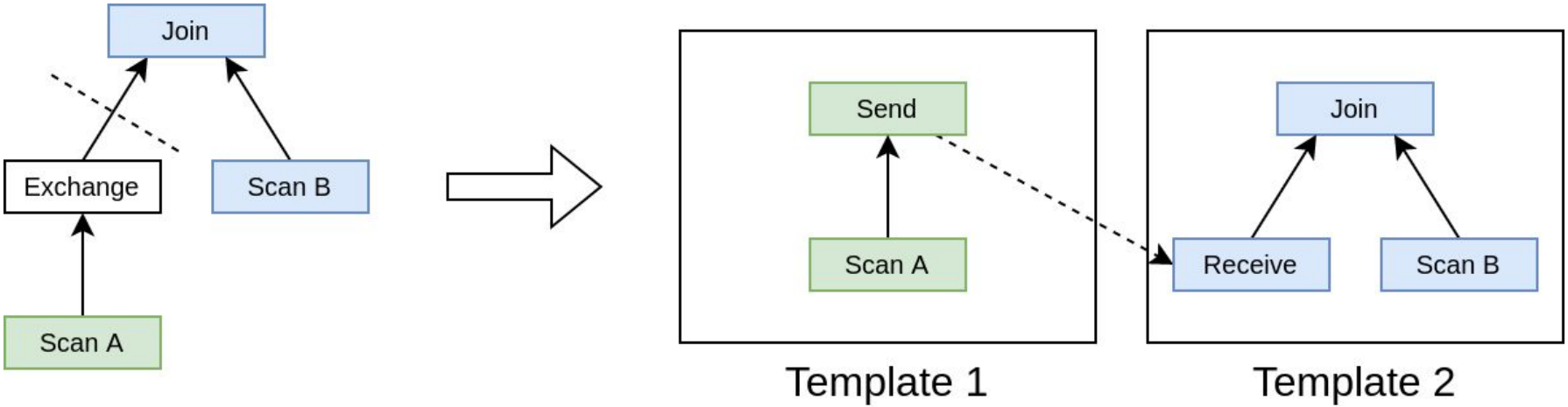


- Фазы [43]:
  - **Rewrite 1** - subquery unnesting, simplification, predicate pushdown, …
  - **Join Order Planning** - эвристическое планирование порядка join-ов (отключено по умолчанию).
  - **Rewrite 2** - transient predicates pushdown, упрощение Join, …
  - **Logical Planning** - cost-based, планирование 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!

Запуск запросов

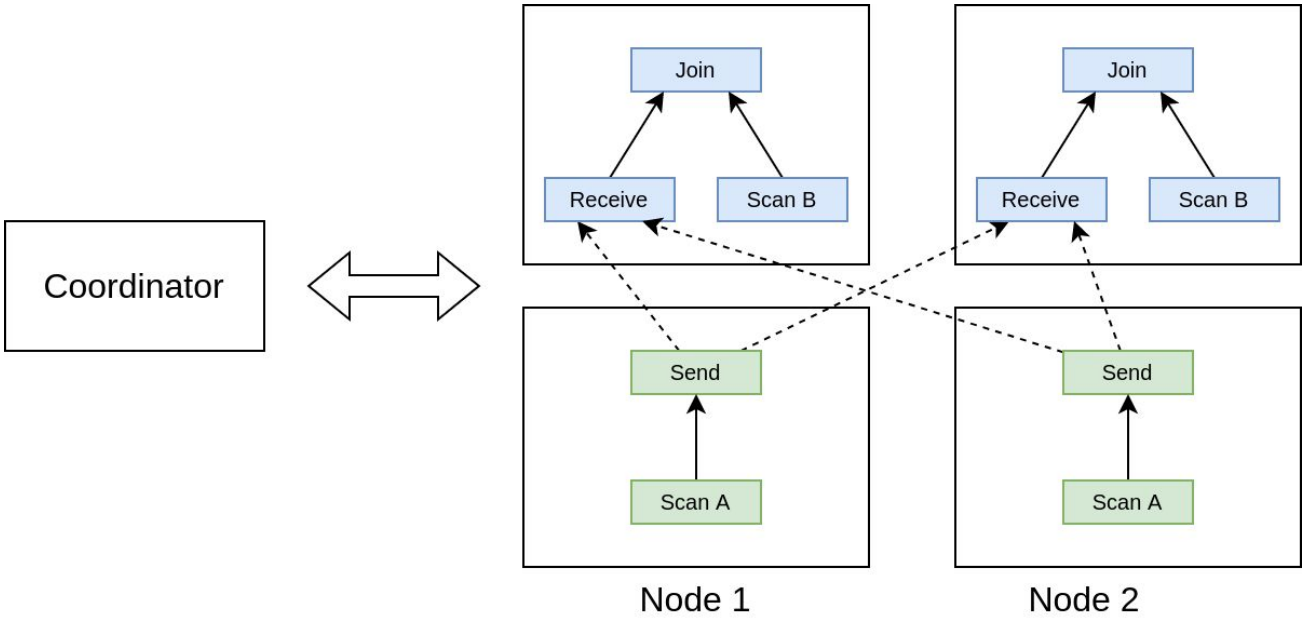
- Фрагменты.
- Акторы: координатор, участник.

Фрагменты



- План разбивается на несколько **фрагментов** (шаблонов), которые будут отправлены на узлы.
- Разные узлы могут выполнять разные фрагменты, в зависимости от особенностей реализации:
  - Вертикальный и горизонтальный параллелизм.
  - Требования к операторам (например, скан может требовать наличие конкретного шарда на узле).

Акторы



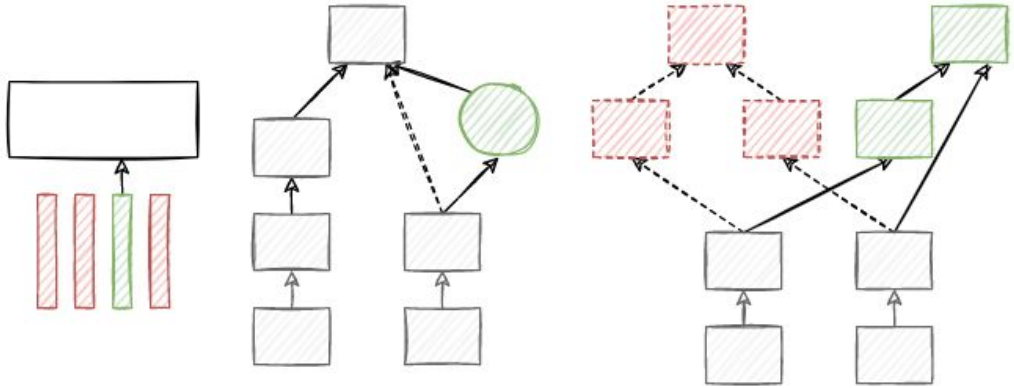
- **Координатор** управляет жизненным циклом запроса в кластере: планирование, запуск, отмена, fault tolerance.
- **Узлы** выполняют операторы, обмениваются данными.
- TODO: Практика - управление параллелизмом

Runtime оптимизации

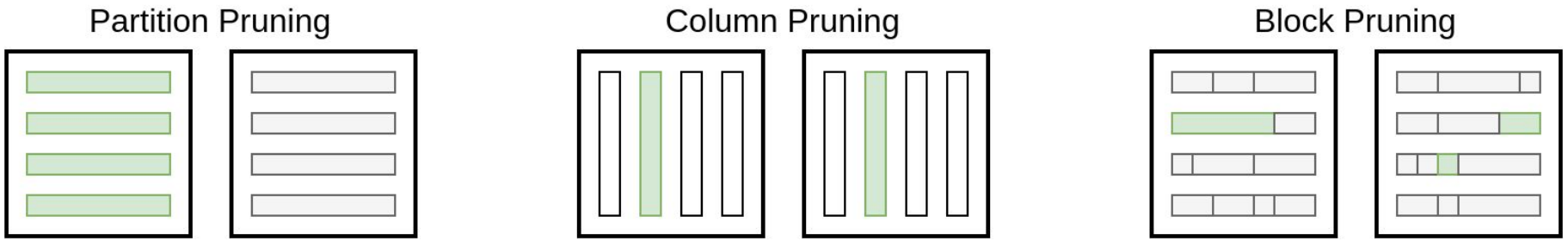
- Прунинг
- Динамические операторы
- Замена и реоптимизация плана во время исполнения

Runtime оптимизации

- Оптимизатор может принимать неоптимальные решения
- Собранную во время исполнения информацию можно использовать для улучшения производительности выполняющегося запроса
- Может быть выгодно поменять план исполнения исходя из имеющихся на данный момент ресурсов



Pruning



- Идея: игнорировать части источника, которые заведомо не содержат необходимых данных.
  - **Partition pruning** - игнорируем ненужные шарды [47], [48].
  - **Column pruning** - игнорируем ненужные колонки (больше применимо для колоночных систем) [49].
  - **Block pruning** - игнорируем отдельные блоки источника [50]
    - Используя статистики блоков в колоночных СУБД
    - Bitmap Scan в Postgres [51]
- Реализация:
  - Планирование - совмещаем Scan, Filter (предикаты) и Project (запрошенные колонки), выводим ограничения.
  - Выполнение - если ограничение динамическое, применяем его в runtime (block pruning).

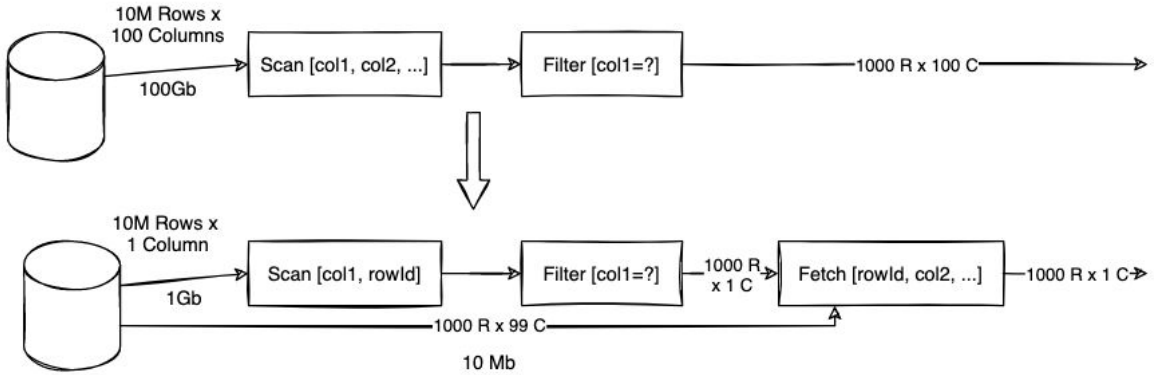
Pruning

- **Пример**: partition pruning
  - `PARTITION BY month(ts)`
  - `SELECT sum(amount) FROM orders WHERE ts` **`BETWEEN ‘2020-10-01’ and ‘2021-02-01’`**
- Данные оптимизации обычно требуют, чтобы предикат был **sargable!**
  - Сравнение, AND/OR, LIKE с фиксированным префиксом, и т.д.

Runtime: Поздняя материализация

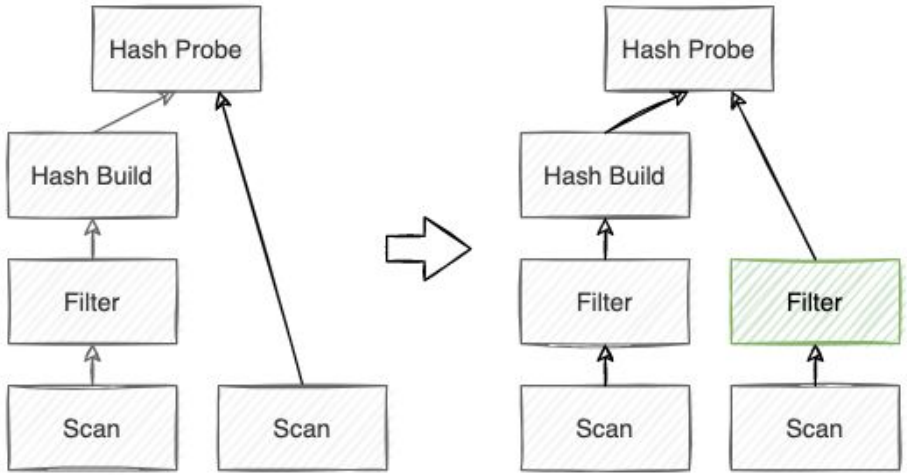
```
SELECT col1,…, col100 from table where col1 = “SelectiveId”
```

- Фильтруем по маленькому числу колонок, возвращаем большое число колонок
- Большая часть данных будет игнорирована
- Column store + rowId: читаем только нужную колонку и дочитываем оставшиеся колонки уже после фильтрации
- Пример: VectorWise, Vertica [52]



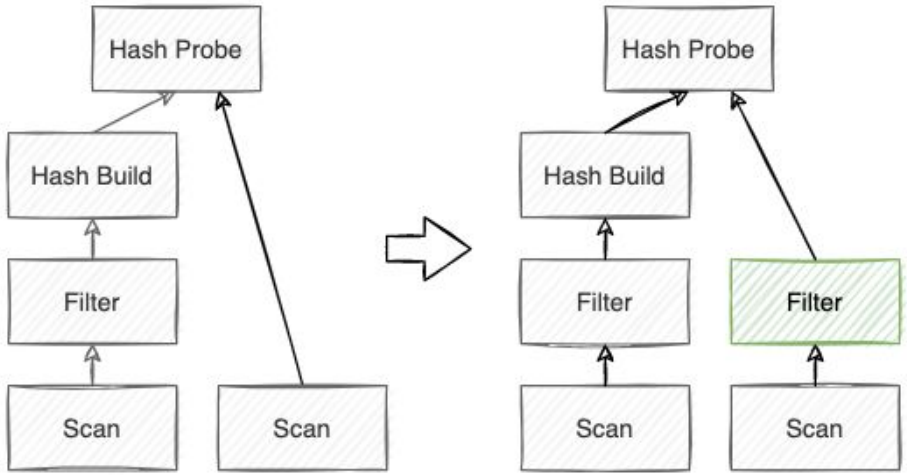
Runtime: Динамическая фильтрация в Join

- Используем состояние хеш-таблицы для ранней фильтрации данных в probe плече
- Probe нужно начать выполнять только после того, как полностью выполнился hash build
  - Требуется поддержка со стороны планировщика
- **Пример: Presto** [53]**, Spark** [54]**, Dremio** [55]**, Snowflake** [56]



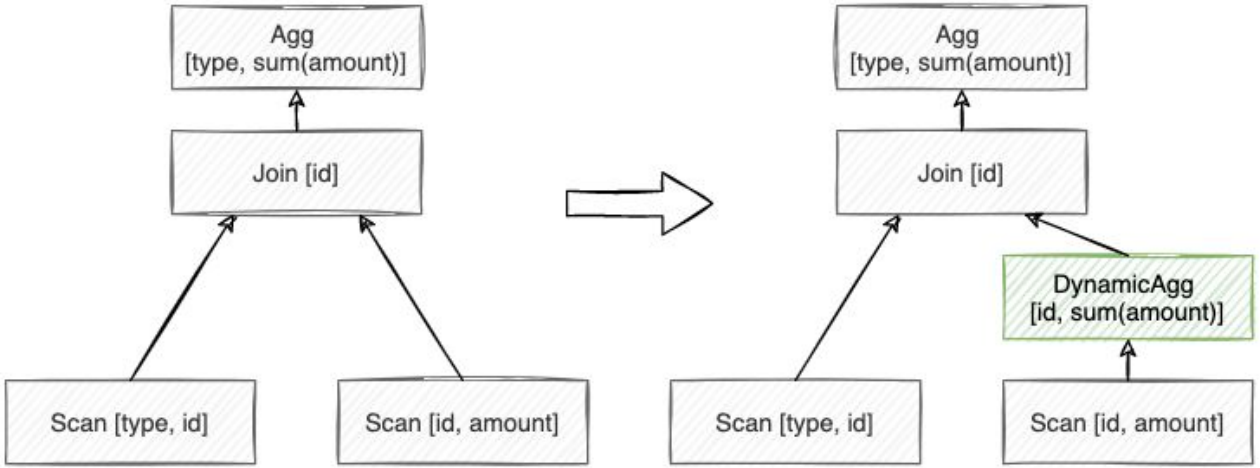
Runtime: Динамическая фильтрация в Join

- Построение фильтра требует дополнительных ресурсов
- Совет: проверьте значение threshold для выключения динамической фильтрации
- **Presto/Trino** [57]:
  - `dynamic-filtering.large-broadcast.*`
  - `dynamic-filtering.large-partitioned.*`



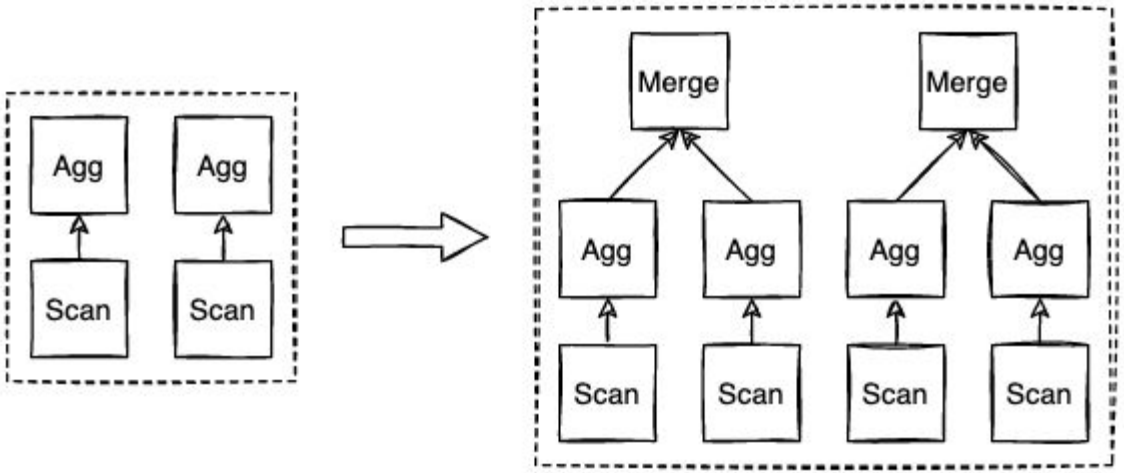
Runtime: Aggregate pushdown

- Агрегат можно продублировать и запушить за Join для снижения кардинальности
- Выгодно только если промежуточный агрегат снижает кардинальность
- “Отключение” промежуточного агрегата не влияет на корректность
- **Пример: Snowflake** [56]



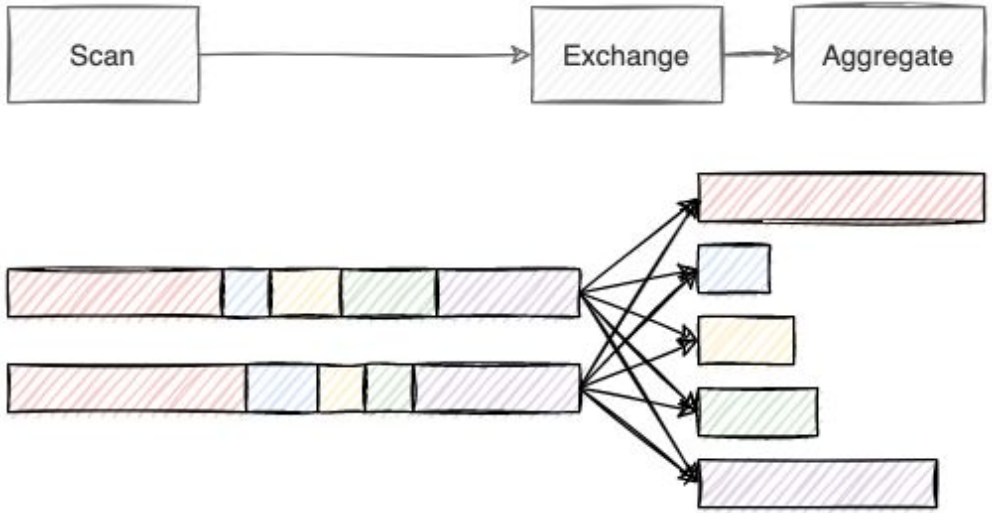
Runtime: пре-агрегация

- Часто есть возможность читать данные параллельно даже из одного шарда
- Разбиваем шард на части, для каждой части делаем локальную пре-агрегацию
- Результат агрегации каждой из частей комбинируем
- Количество частей можно выбирать в зависимости от доступности ресурсов
- Компенсирует data skew [58],[59]



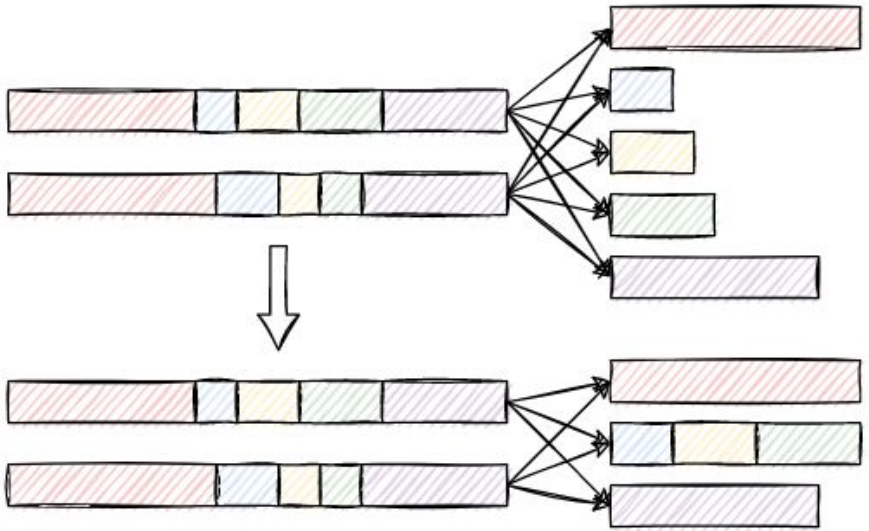
Runtime: Динамическое количество шардов

- При перераспределении данных количество шардов основано на кардинальности
- Ошибки в оценке кардинальности => непредсказуемое время исполнения
  - Использование слишком маленьких шардов увеличивает накладные расходы
  - Использование слишком больших снижает параллелизм



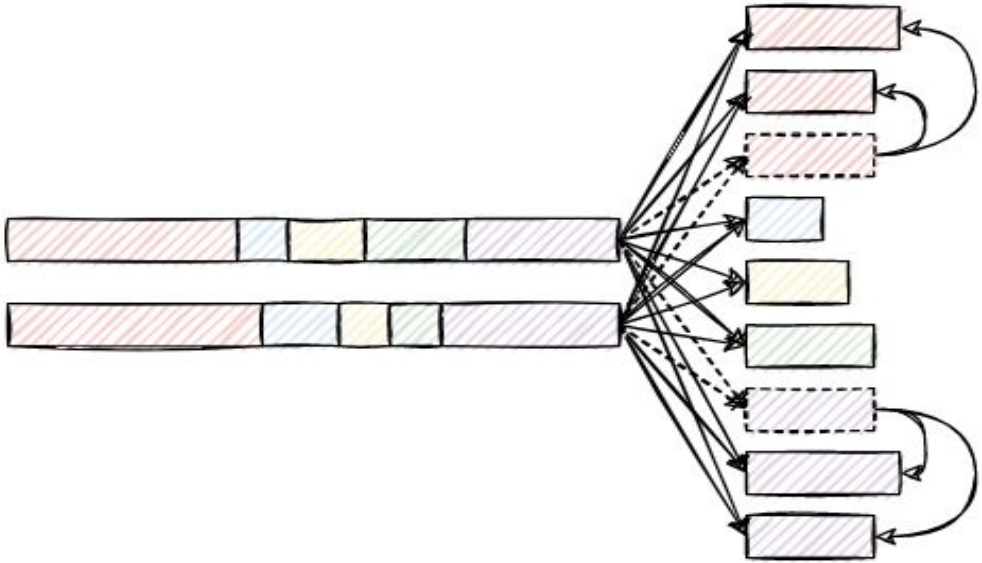
Runtime: Динамическое количество шардов

- Маленькие шарды - совместить
- **Spark** [60], [61]
  - `spark.sql.adaptive.enabled`
  - Реоптимизация происходит между точками материализации



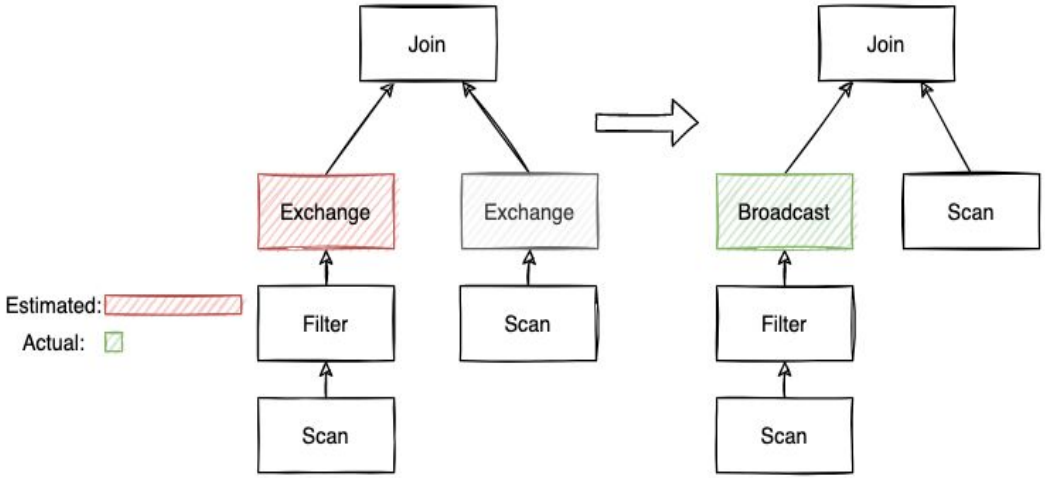
Runtime: Динамическое количество шардов

- Большие шарды - разделить
- **BigQuery** [58]
  - Контроль размера шарда во время исполнения
  - Перераспределение уже накопленного результата в случае превышения порога



Runtime: Замена алгоритма join

- Ошибка в оценке кардинальности, одна из сторон Join оказывается меньше, чем broadcast threshold



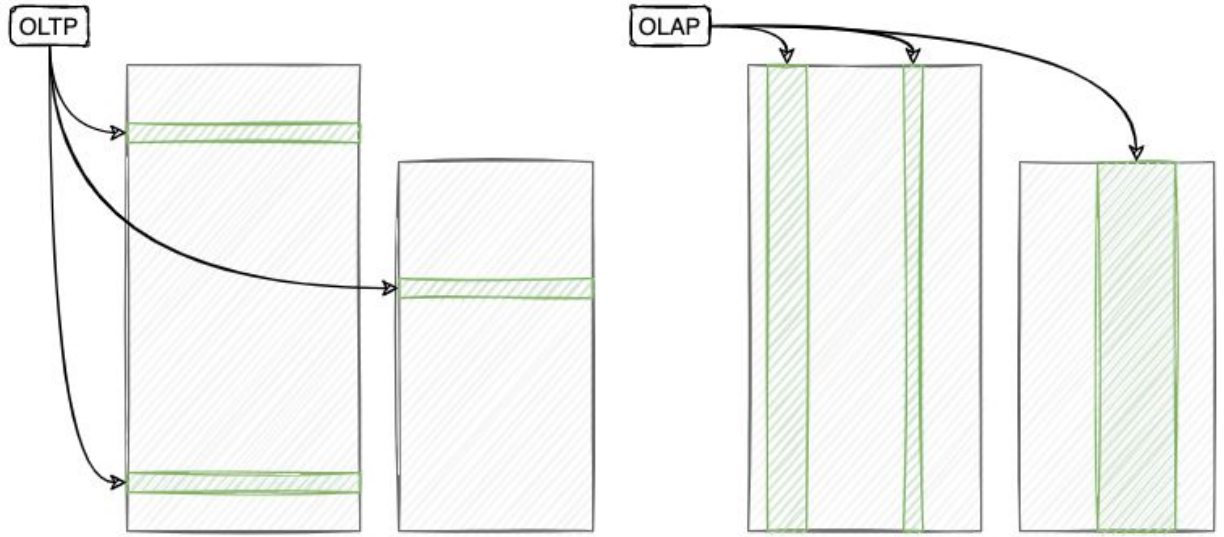
- **Spark** [60], [61]:
  - `spark.sql.adaptive.enabled`
  - Merge => Broadcast
  - Materialization
- **BigQuery** [58]:
  - Shuffle => Broadcast
  - Cancel & Re-Execute

Эффективное исполнение

- Векторизация
- Компиляция запросов
- Operator fusion

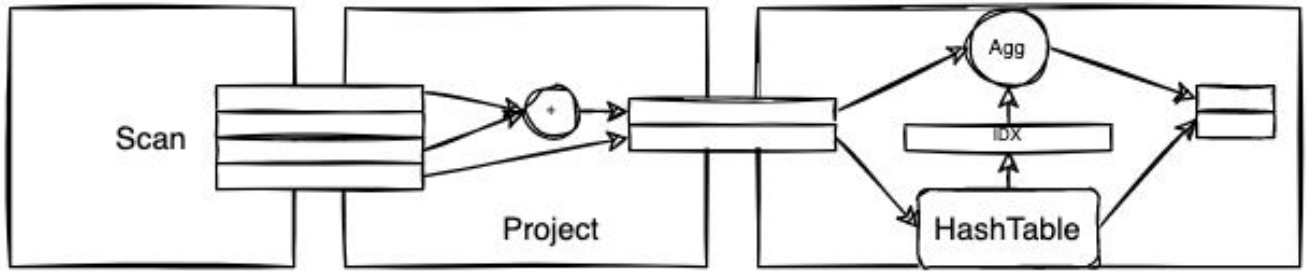
Advanced: эффективное исполнение

- OLTP: точечные запросы, широкий набор колонок
  - Относительно небольшое количество строк и трансформаций
  - Критичен выбор оптимального метода доступа (скан vs Index, выбор оптимального индекса)
- OLAP: “длинные” запросы, узкий набор колонок



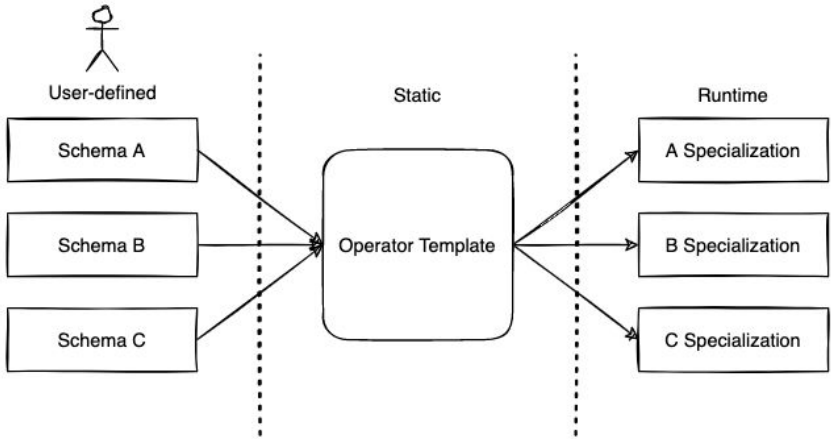
Advanced: векторизация

- OLAP - критически важна эффективная утилизация ресурсов
- Обработка данных в компактном цикле блоками, помещающимися в кэш процессора
- Векторизированный оператор [52]
  - Элемент обработки - набор кортежей
  - Примитивная операция работает над одной колонкой
  - ClickHouse, Actian Vector (ex. VectorWise), Vertica, Presto/Trino, CockroachDB [62],[63]
  - Apache Arrow-based (Dremio, Flink)[64]



Advanced: компиляция

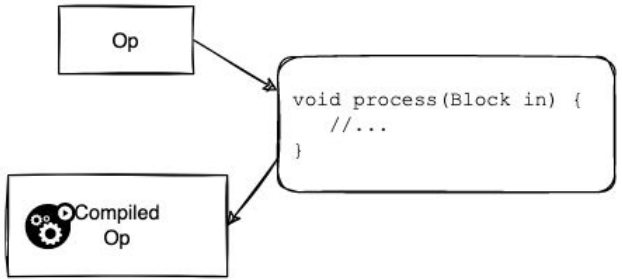
- Интерпретация: большие накладные расходы на контроль типов и виртуальные вызовы
- Специализация: для конкретной схемы код каждого оператора можно существенно упростить [65]
- Схема определяется пользователем => произвольное количество специализаций
  - Генерация кода при исполнении
- Время компиляции может быть больше времени исполнения запроса
  - Кэширование, совмещение интерпретации и компиляции



Компиляция: подходы

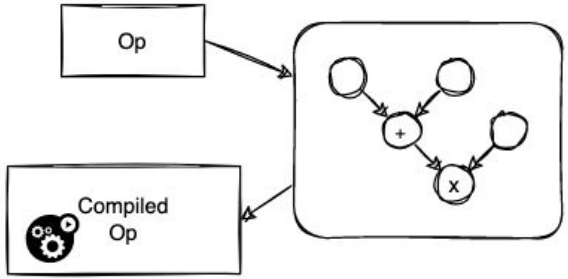
Генерировать исходный код

- e.g. Janino, clang
  - Apache Spark [66], MS Hekaton, MemSQL, [62]
- + Более привычное представление
- + Легче отладка
- - Существенное время компиляции



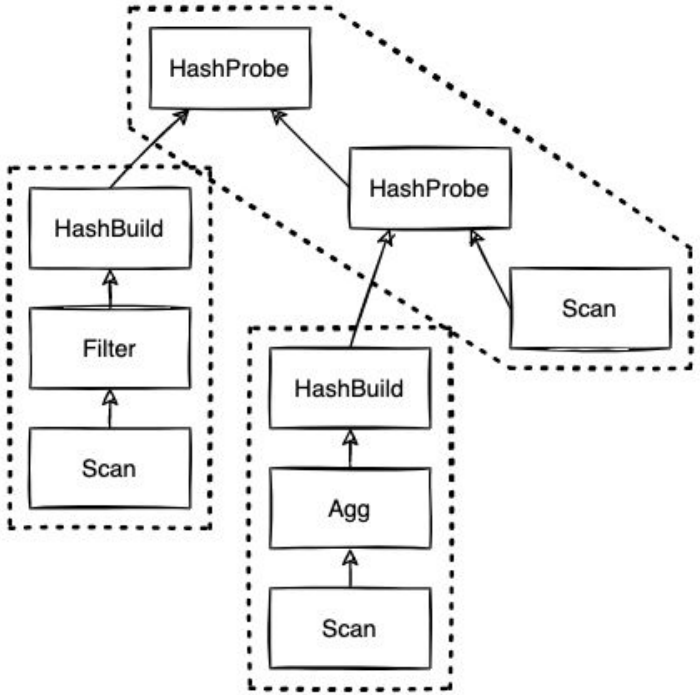
Генерировать IR компилятора

- LLVM
  - Apache Arrow [67], MemSQL (2016+) [62]
- Truffle
- ASM
  - Splice Machine, Presto
- + Потенциально более быстрая компиляция
- - Высокая сложность поддержки



Advanced: Operator Fusion

- Использование абстракции отдельного оператора - не самый эффективный подход
- Пайплайнинг: векторное выполнение “непрерывного набора операторов” [68]
- Цель - держать содержимое кортежа в регистрах процессора как можно дольше



Итого

- Главная особенность distributed SQL-движков - необходимость эффективного выполнения распределенных операторов.

- Оптимизатор:
  - Планирование классических реляционных операторов (rule-based, iterative/cost-based, несколько фаз).
  - Планирование Exchange операторов.

- Executor:
  - Корректировка решений оптимизатора (перемещение операторов, repartitioning, broadcast).
  - Pruning для уменьшения количества сканируемых данных.
  - Агрессивные промежуточные агрегации для уменьшения количества передаваемых по сети данных.
  - Компиляция и векторизация для лучшей утилизации железа.

- Ссылки: https://www.notion.so/querifylabs/SQL-0dbb77b8964d4eb6933349eb2b07ac32