Querify Labs

# Как устроено выполнение SQL-запросов в Presto/Trino

Владимир Озеров
**Querify Labs**

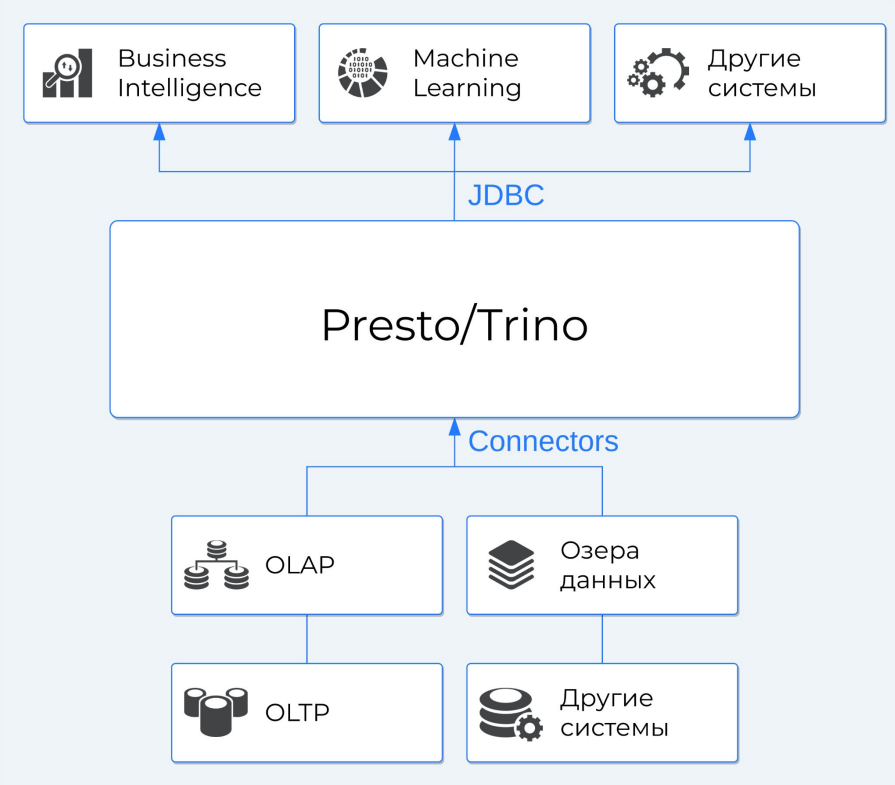
# Querify Labs

Querify Labs

cedrusdata

- Разработка новых аналитических СУБД и data management систем для технологических стартапов по всему миру (стек - Java/C++, Apache Calcite, Apache Arrow, Velox).
- Разработка российской аналитической платформы CedrusData на основе Presto.
- Контрибьютим в Apache Calcite и Apache Arrow.

# Архитектура




Presto/Trino - это распределенный SQL-движок.

Подключается к источникам данных с помощью коннекторов:

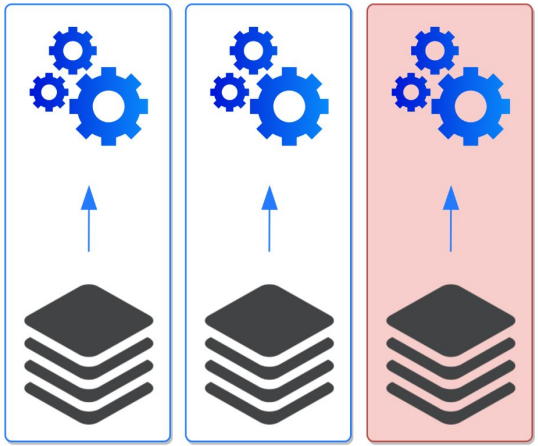
- Озера данных под управлением Hive Metastore и Apache Iceberg.
- Хранилища данных: Greenplum, ClickHouse, Apache Druid, Apache Pinot.
- Реляционные СУБД: Postgres, MySQL, Oracle, SQL Server, MariaDB.
- Нереляционные источники: Cassandra, MongoDB, Redis, Kafka, …

Отдает данные через **JDBC**.

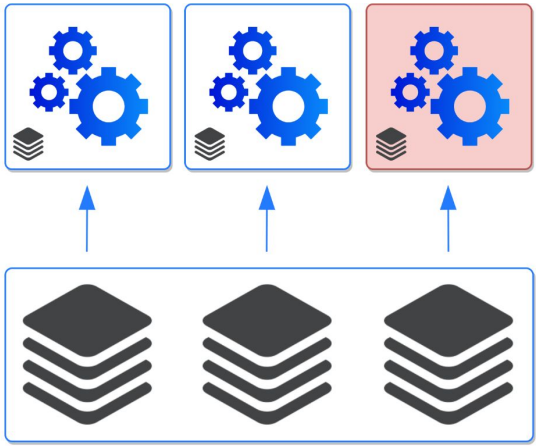
# Архитектура: плагины и коннекторы

- **Plugin** - набор расширений функционала Presto.
- **Connector** - опциональный компонент плагина, который описывает логику работы с источниками данных определенного типа.
  - Пример: Postgres.
- **Catalog** - инстанс коннектора, который работает с конкретным источником.
  - Пример: конкретный инстанс Postgres.

# Архитектура: shared storage



Shared-nothing



Shared storage

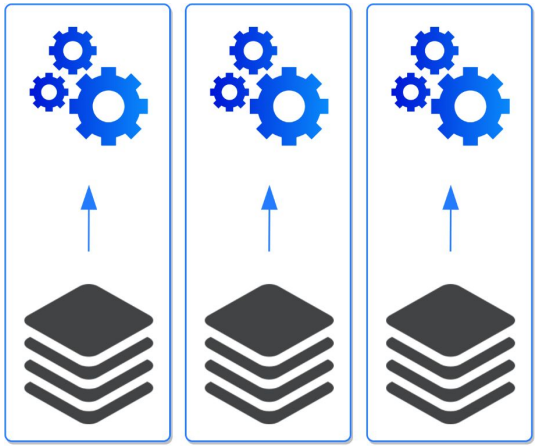
**Shared-nothing:** вычисления и данные совмещены.

- Быстро работает.
- Изменение топологии влияет и влияет на **доступность данных**.
- Тяжело масштабируется.
- Плохо подходит для облака.

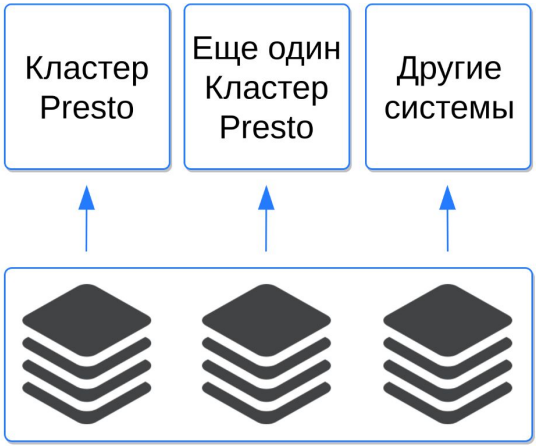
**Shared storage:** вычисления и данные разделены.

- Изменение топологии не влияет на доступность данных.
- Отлично масштабируется.
- Отлично подходит для облака.
- Быстро работает за счет гигабитных сетей и локального кэширования.

# Архитектура: shared storage



Shared-nothing




Shared storage

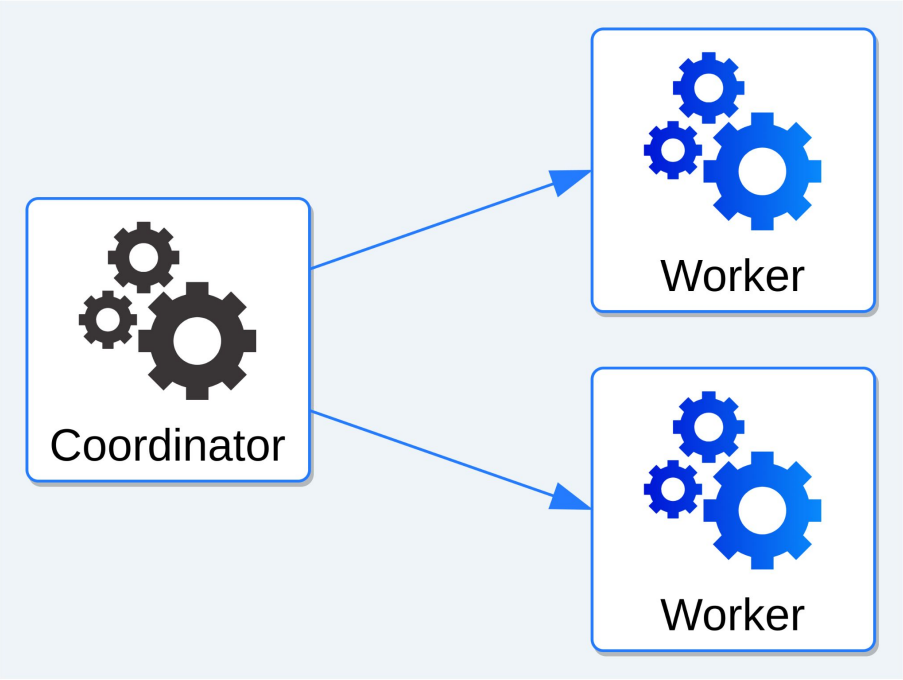
**Shared-nothing:**

- Тенденция к централизации и vendor lock-in за счет дорогой эксплуатации и хранения данных в проприетарных форматах.

**Shared storage:**

- Централизация не является обязательной, так как кластеры не влияют на целостность данных.
- С одними и теми же данными может работать много систем (напр. Presto + Spark).

# Архитектура: типы узлов



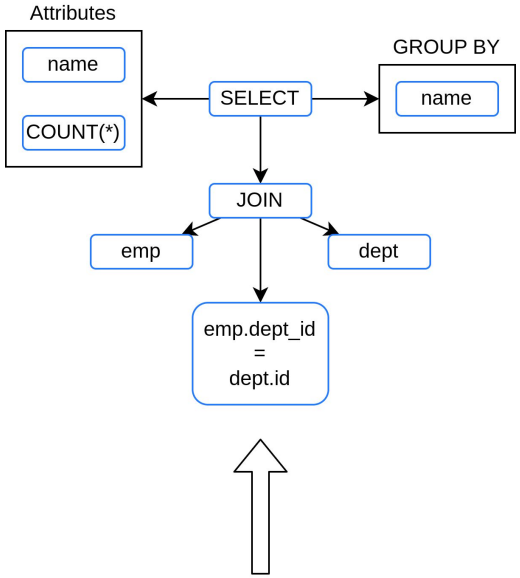

**Coordinator**:

- Получает, планирует и координирует запросы.
- Может выполнять запросы, если стоит соответствующий флаг.
- При необходимости кластер может содержать несколько координаторов.

**Worker**:

- Выполняет запросы.

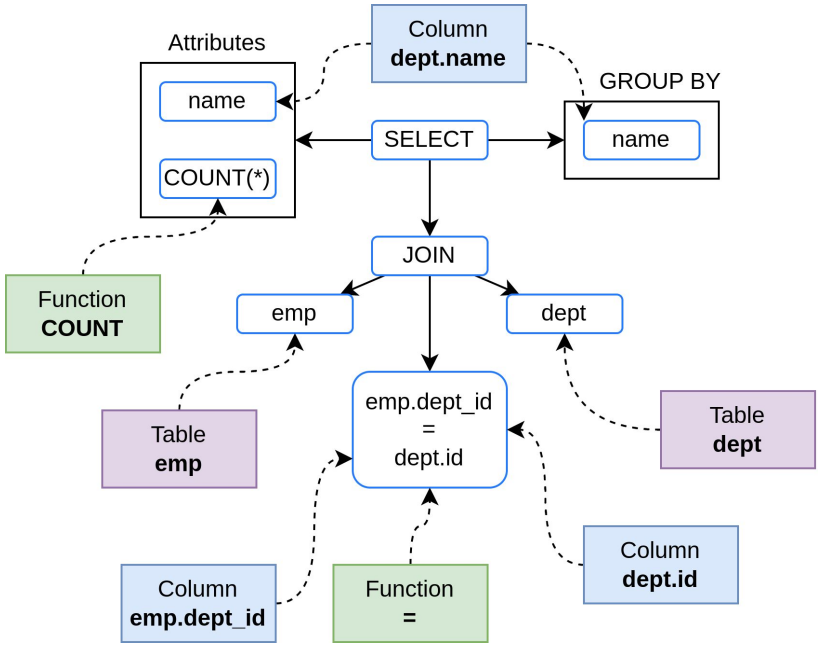
# Планирование: парсинг




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

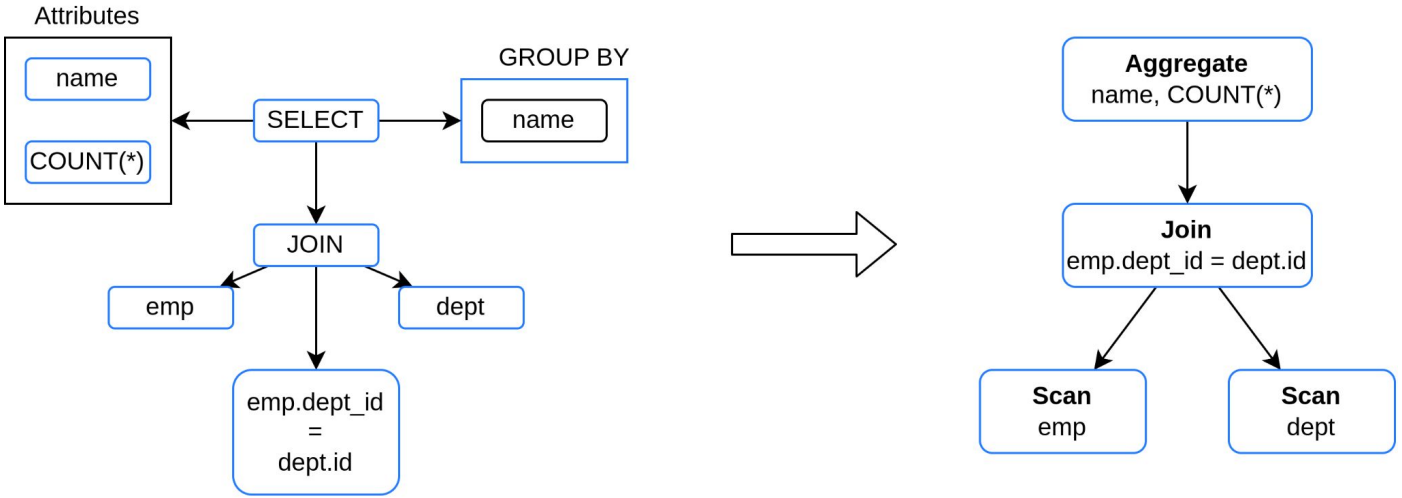
- Задача: превратить SQL-строку в синтаксическое дерево.
- Реализован с помощью ANTLR.
- См. SqlBase.g4.

# Планирование: семантическая валидация



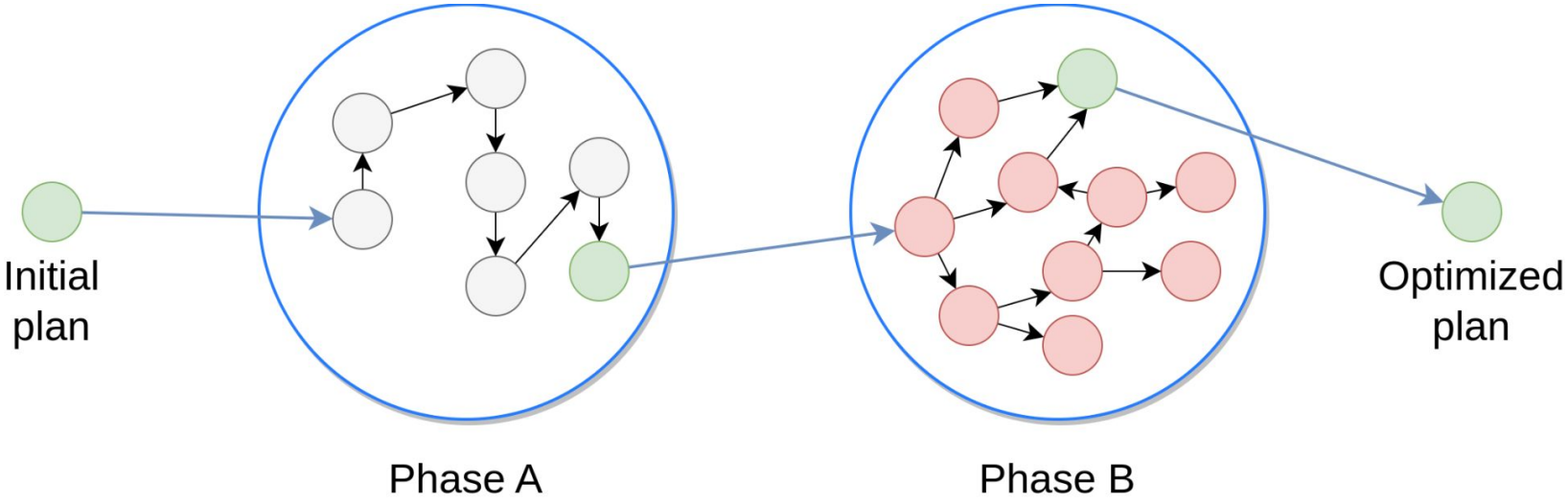

- Задача: определить объекты, участвующие в запроса; убедиться в семантической корректности.
- В отличие от синтаксического анализа, семантический анализ не поддается автоматизации. Реализован большим количеством “спагетти”-кода.
- **Коннекторы** определяют, какие объекты доступны системе (table, column, …).
- **Плагины** могут определять дополнительные функции.
- См. Analyzer.java.

# Планирование: трансляция



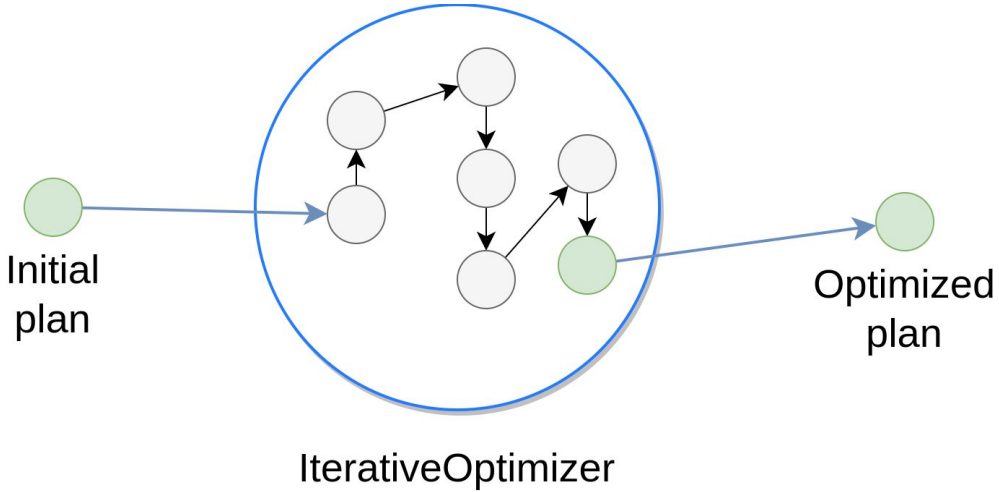

- Современные планировщики SQL-движков чаще всего работают с **реляционным** представлением.
- Presto использует реляционное представление:
  - Scan, Project, Filter, Aggregation, Join, SetOp (Union, Minus, Intersect), …
- См. PlanNode.java, RelationPlanner.java.

# Планирование: фазы



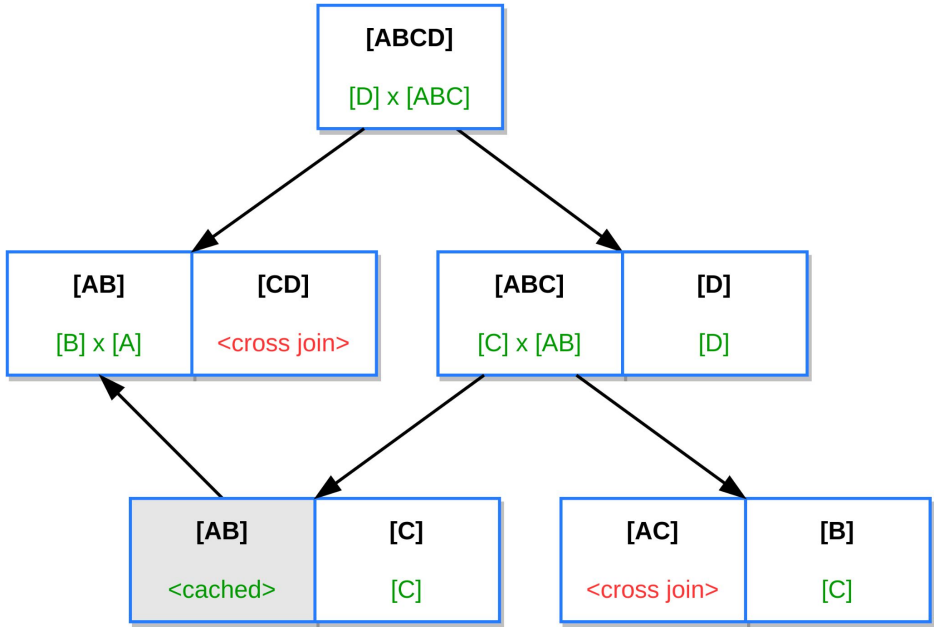

- Планирование организовано в последовательность шагов.
- На каждом шаге мы получаем на вход один план, и производим другой.
- Планирование в Presto состоит из примерно 80 шагов.
- См. PlanOptimizer.java, PlanOptimizers.java.

# Планирование: итеративный оптимизатор



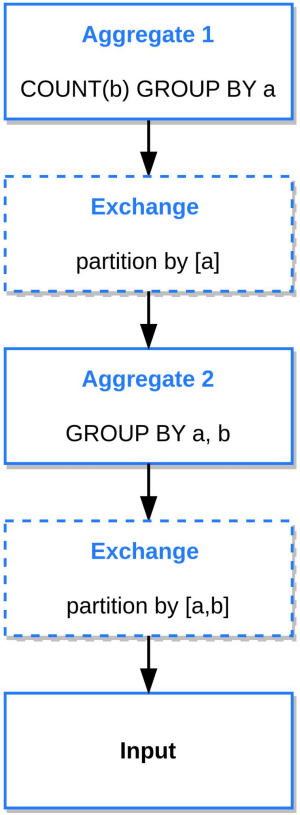

- Оптимизатор принимает начальный план и набор правил, отдает новый план. Работает пока есть возможность применять правила. Не является cost-based.
- Правило представлено паттерном и логикой трансформации. Примеры: упрощение выражений, filter pushdown, pushdown вычислений в коннектор.
- См. IterativeOptimizer.java.

# Планирование: join order




- Класс эквивалентности `[ABCD]` это все возможные перестановки отношений A, B, C, D.
- Проход вниз: разбиваем классы эквивалентности на более мелкие пары: `[ABCD]->[ABC]x[D]`.
- Проход вверх: находим оптимальный порядок для класса эквивалентности на основании стоимости.
- Стоимость это функция статистик (row count, min, max, null count, ndv).
- Для Scan операторов статистики предоставляются коннекторами. Для остальных операторов статистики вычисляются с помощью эвристических формул.
- См. ReorderJoins.java.

# Планирование: exchanges




powerSet(a)
[a]

powerSet(a,b)
[a,b], [b,a], [a], [b]

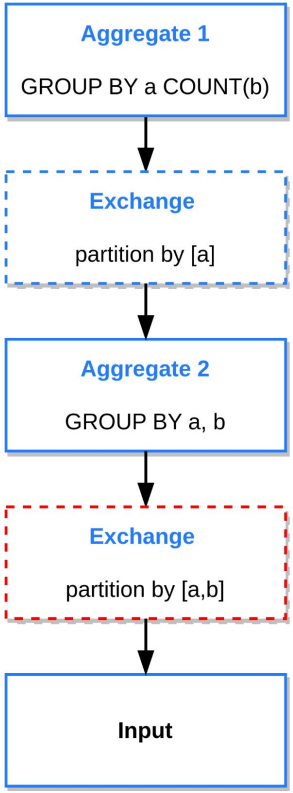
Операторы могут **требовать** определенные распределения у своих входов для обеспечения корректности. Типичные примеры:

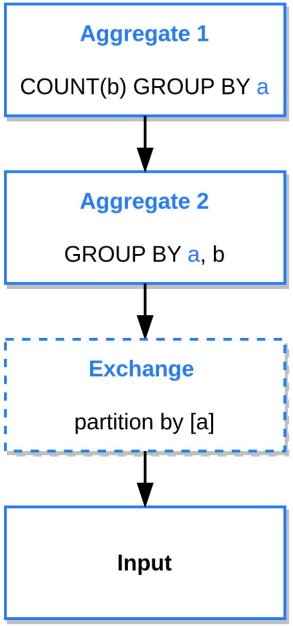
- **Hash Aggregate** - партиционирование согласно ключу группировки:
  - `GROUP BY a, b -> [a,b]`
- **Hash Join** - партиционирование согласно equi-ключу.
  - `A.a1=B.b1 AND A.a2=B.b2 -> [a1,a2] [b1,b2]`
- **Window** - партиционирование согласно `PARTITION BY`.
  - `PARTITION BY a, b -> [a,b]`

Если затребовано распределение `[a,b]`, то нас устроит любое распределение из `powerSet(a,b)`, кроме пустого.

Оператор **Exchange** моделирует перераспределение данных между потоками согласно ключу партиционирования. Оптимизатор **обязан** расставить Exchange-и для обеспечения корректности.

# Планирование: exchanges




powerSet(a)
[a]

powerSet(a,b)
[a,b], [b,a], [a], [b]

```
SELECT a, COUNT(DISTINCT b) GROUP BY a
```

Можно уменьшить количество Exchange, если поискать **общие** ключи (“overlapping exchanges”).

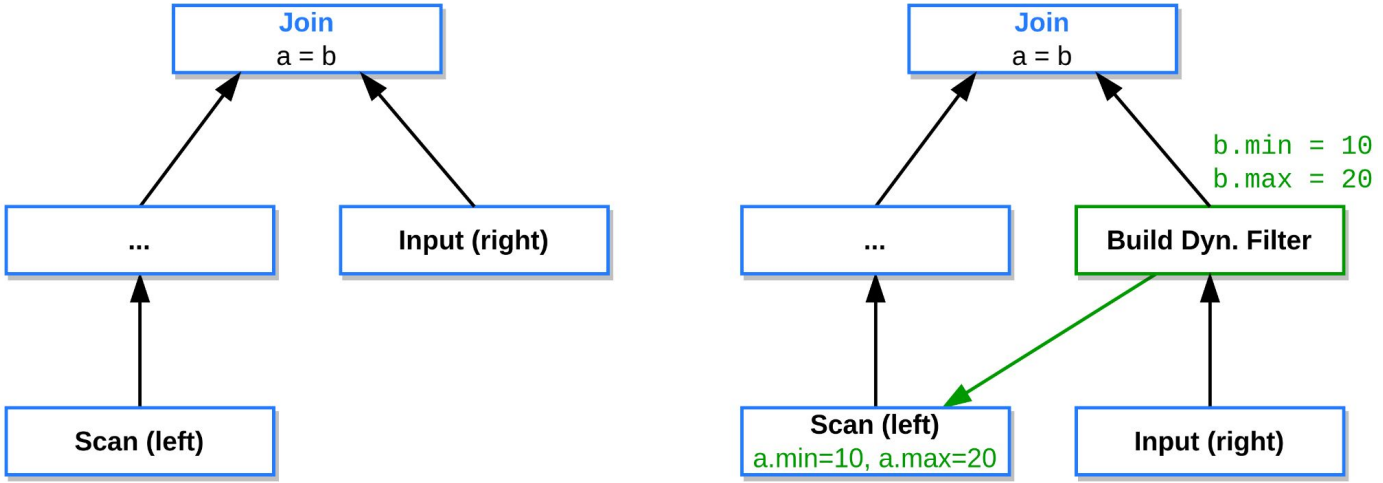
Идеальный подход:

- Перебор всех возможных порядков Join и всех возможных расстановок Exchange в **единой cost-based** фазе.

Подход Presto:

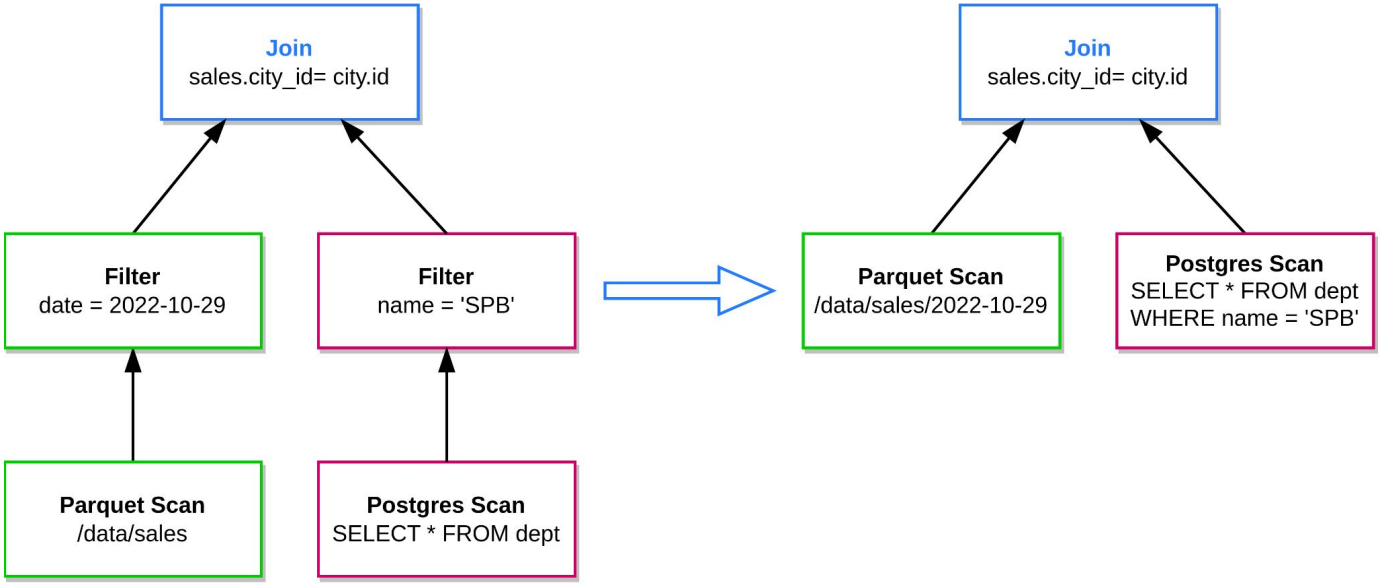
- Отдельно планируем порядок Join с эвристической расстановкой Exchange.
- Отдельно планируем эвристическую расстановку Exchange.
- Результат: достаточно хорошая расстановка Exchange, но можно лучше!

# Планирование: динамические фильтры




- Идея: посчитать в runtime предикаты одной стороны Join, и применить их к другой стороне.
- Наибольший выигрыш происходит за счет pushdown динамического фильтра в Scan.
- В Presto присутствует два типа динамических фильтров:
  - **Локальные** - вычисление и применение происходит на одном узле.
  - **Распределенные** - вычисление на координаторе, применение на воркерах.

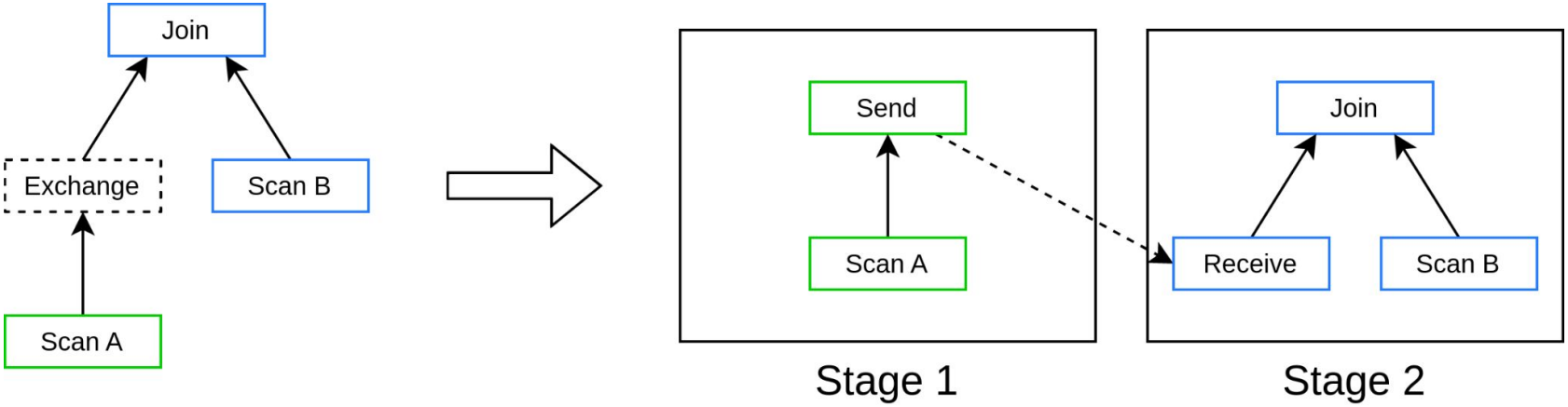
# Планирование: connector pushdown




- Коннекторы могут предоставлять свою логику для организации pushdown:
  - **Hive**: filter pushdown, partition pruning, partial aggregation (напр. MIN/MAX).
  - **JDBC**: можно запушить практически все, что угодно.

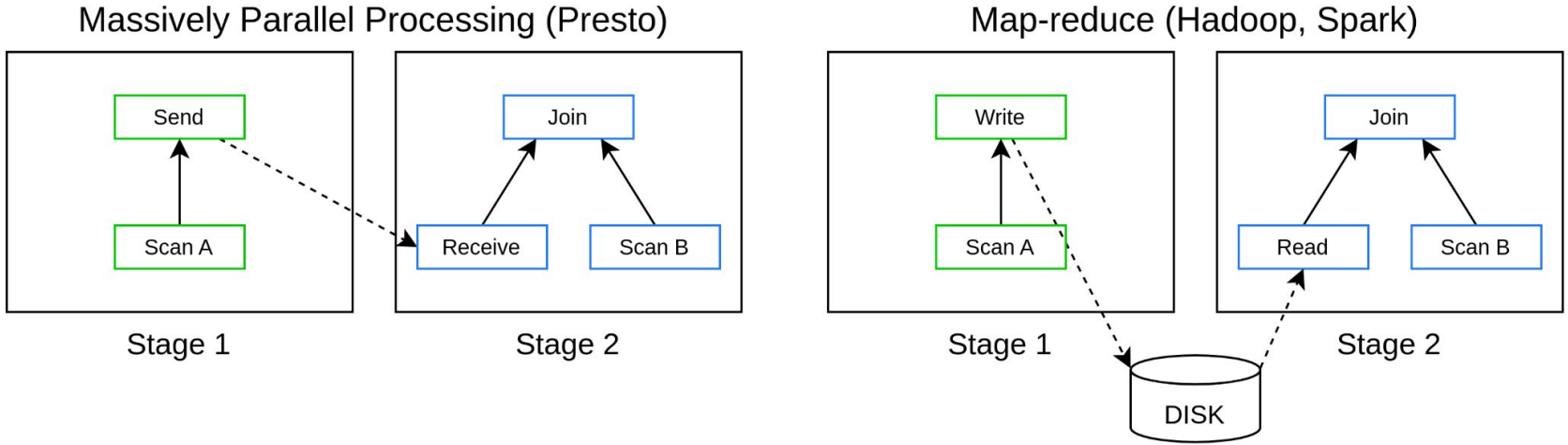
См. ConnectorPlanOptimizerProvider (Presto), ConnectorMetadata (Trino)

# Выполнение: stages



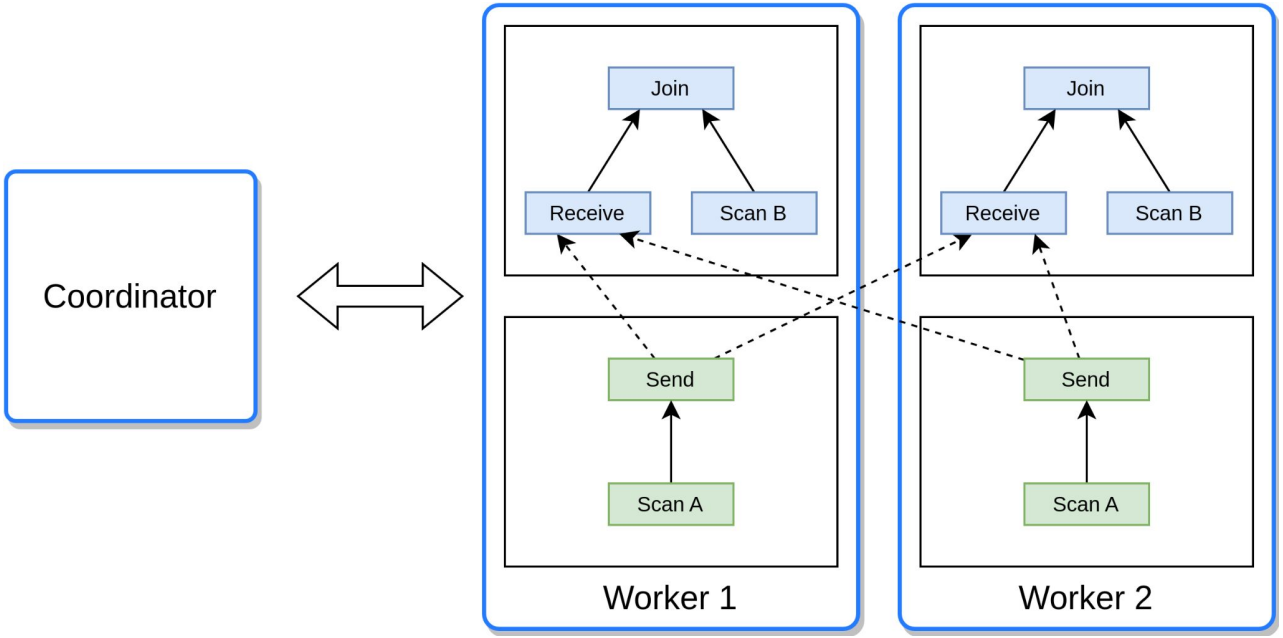

- Координатор разделяет логический план на последовательность **stages**.
- Stage - это последовательность операторов, которые могут быть выполнены локально на воркере.

# Выполнение: MPP vs MR



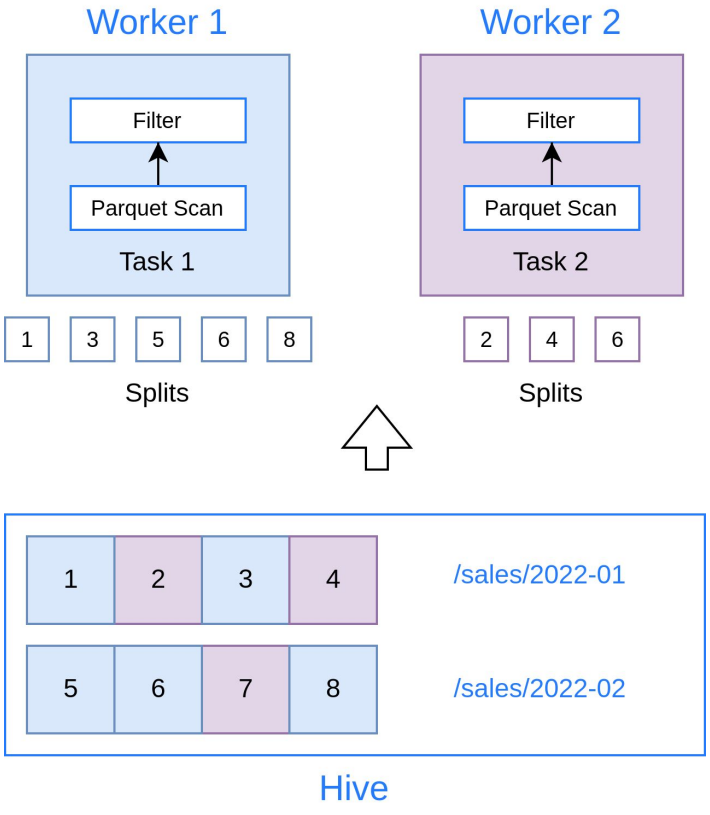

- MPP - данные передаются по сети, хранятся в памяти.
  - Быстро, но может не хватить памяти и хуже отказоустойчивость.
- MR - данные передаются через персистентное хранилище.
  - Медленно, но можно обрабатывать очень большие объемы.
- Presto это MPP с зачатками MR:
  - Умеет сбрасывать данные на диск, но не делает этого по умолчанию.
  - Инициатива Presto-on-Spark.

# Выполнение: Task




- **Stage** - это “шаблон” фрагмента запроса. У каждого Stage есть набор источников данных.
- **Task** - это инстанс stage на конкретном узле. Координатор определяет на каких узлах выполнять stage в зависимости от требований источников данных

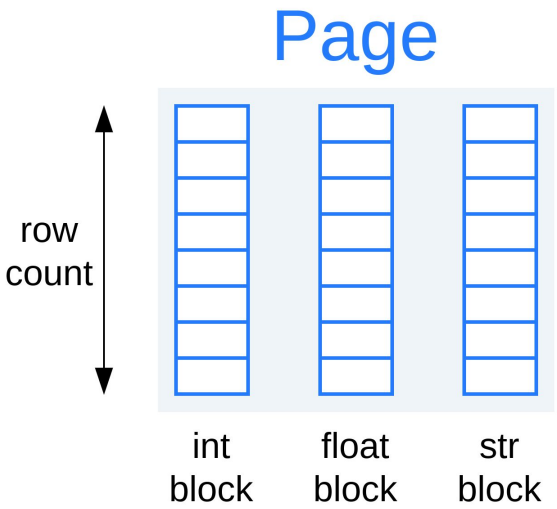
# Выполнение: Split




- **Split** - это часть данных источника. Каждый источник данных представлен конечной последовательностью Split-ов.
  - Таблицы Hive разбиваются по файлам и частям файлов.
  - JDBC-источники всегда состоят из одного сплита, представляющего собой result set выполнения запроса.
- Split может задавать требования к тому, где должна происходить его обработка:
  - **Обязательно** на конкретном узле.
  - **Желательно** на конкретном узле.
  - Без ограничений.

См. ConnectorSplitManager, ConnectorSplit.

# Выполнение: Page




Операторы получают и производят **Page** - набор кортежей в **колоночном** формате.

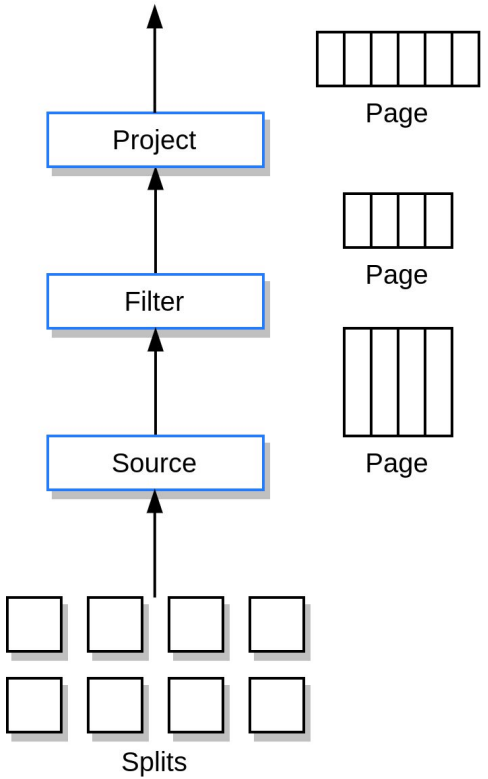
Каждый атрибут представлен объектом **Block**, который полностью инкапсулирует доступ к данным по индексу.

Пример `IntArrayBlock`:

- `int[]` - массив значений.
- `bool[]` - маркер NULL.

См. Page, Block.

# Выполнение: push



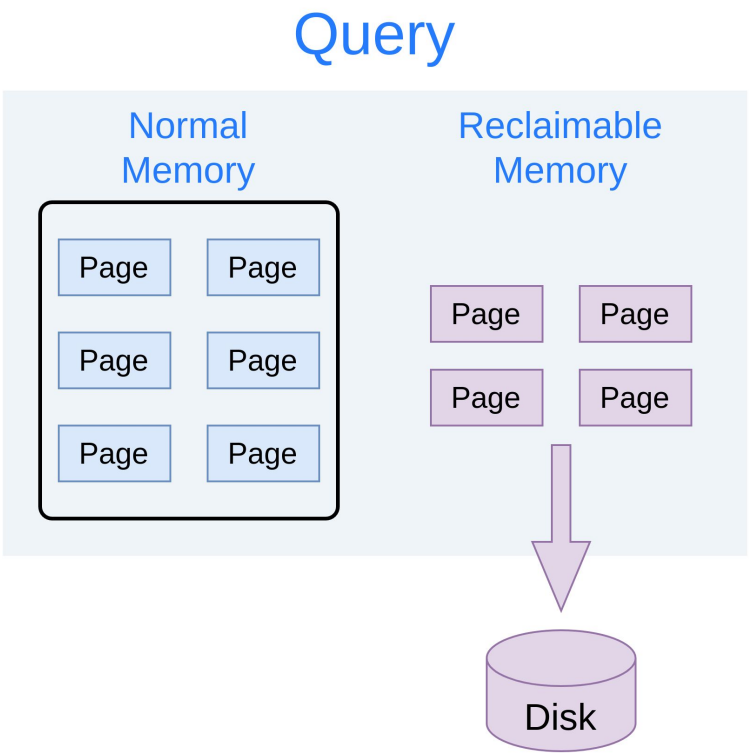

Presto реализует неблокирующую **push** модель выполнения:

- Source оператор получает split(ы).
- Оператор производит Page.
- Page передается следующему оператору (push).

Создание и поглощение page-ей отделено друг от друга:

- Оператор может быть не готов произвести Page. Например, Sort еще не получил все входные данные.
- Оператор может быть не готов принять Page. Например, Hash Join не принимает данные из probe стороны, пока не получены все данные из build стороны.

# Выполнение: управление памятью




Многоуровневое управление памятью:

- Лимит на уровне узла.
- Лимит на уровне запроса.

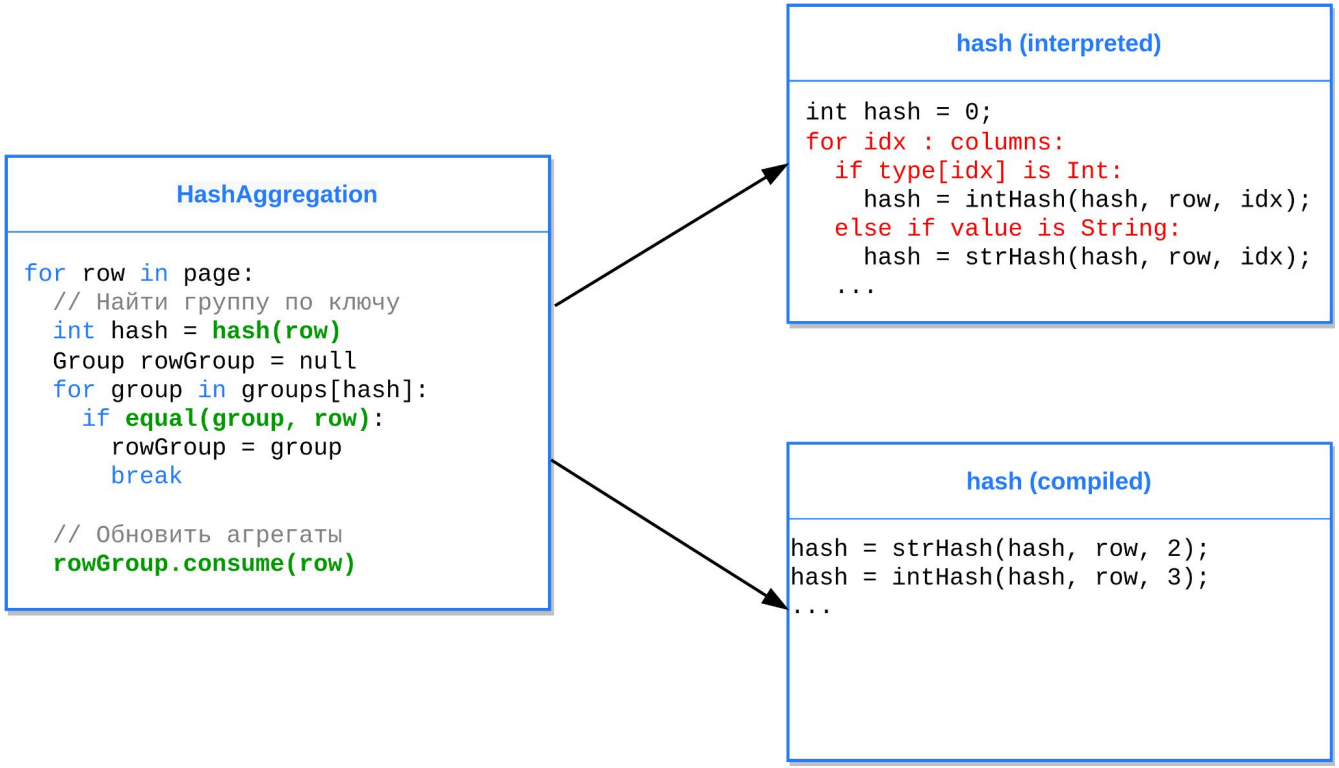
Операторы используют `Page` не только для обмена данными, но и для внутреннего состояния:

- Хэш-таблицы.
- Сортированные структуры.

Spilling:

- По умолчанию, при достижении лимита памяти запрос будет отменен.
- Операторы могут запрашивать память сверх доступного для запроса лимита. При необходимости эта память будет освобождена путем выгрузки данных на диск.

# Выполнение: компиляция




Интерпретируемый код вносит накладные расходы:

- Циклы.
- Условные переходы.
- Виртуальные вызовы.

Скомпилированный код позволяет избавиться от значительной части накладных расходов.

# Выполнение: компиляция

- Операторы содержат указатели на функции. Например:
  - `Projection/Filter` - вычисление выражений.
  - `Aggregation` - вычисление аккумуляторов.
  - `Join` - вычисление фильтров.
- Функции компилируются с помощью ASM в процессе выполнения запроса, используя информацию о конкретном операторе.

# Итого

- Presto/Trino - это распределенная MPP система, которая выполняет SQL-запросы, но не хранит данные (aka shared storage).
  - Обеспечивает высокую гибкость развертывания по сравнению с shared-nothing системами при сохранении производительности.
- Планировщик реализует все ключевые оптимизации (join order planning, exchange planning, dynamic filters, connector pushdown).
  - Большой потенциал для дальнейших улучшений за счет внедрения cost-based оптимизации.
- Runtime реализует push-модель передачи данных в колоночном формате с частичной компиляцией операторов.
  - Большой потенциал для дальнейших улучшений за счет внедрения native execution (см. Velox/Prestissimo).
- Попробовать:
  - Presto: https://prestodb.io/
  - Trino: https://trino.io/
  - CedrusData: https://www.cedrusdata.ru/