DB internals. Третья лекция

Надуткин Федор

December 2023

Планирование порядка join

```
SELECT
    lineitem.*
FROM customer
    JOIN orders ON c_custkey = o_custkey
    JOIN lineitem on o_orderkey = l_orderkey
WHERE
    c_name = 'Ivanov'
```

Листинг 1. Пример проблемного Join

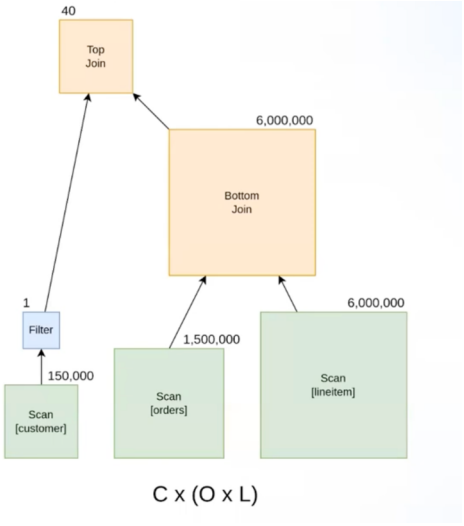

Рис. 1. Плохой вариант построения Join

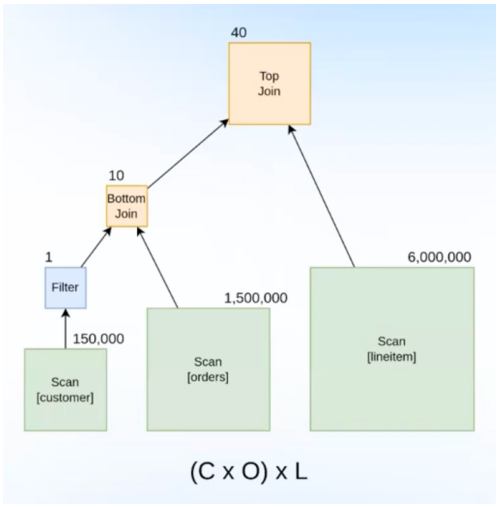

Рис. 2. Хороший вариант построения Join

Как можно видеть из картинок во втором варианте нам нужно хранить в разы меньше промежуточной информации. Получается один порядок Join будет гораздо лучше другого. Таким образом наш движок должен правильно уметь расставлять порядок Join.

Попробуем оценить сколько всего существует возможных порядков Join. Всего их $= n!$ —перебор всех возможных расстановок.

Однако вариант $((a_1 \cdot a_2) \cdot a_3) \cdot a_4 \ldots$ не всегда оптимален, и зачастую мы хотим также уметь расставлять скобки, как например $(a_1 \cdot a_2) \cdot (a_3 \cdot a_4)$.

Количество способов расставить скобки равно числу Каталана $C(n-1) = \frac{(2 \cdot n-2)!}{n! \cdot (n-1)!}$.

Таким образом количество возможных вариантов равно $n! \cdot C(n-1)$. Так для 8 таблиц у нас есть более 17 миллионов альтернатив и растёт оно всё экспоненциально. Понятно, что, перебирать все Join у нас физически нет возможности, значит надо использовать эвристики.

Не рассматривать Cross Join

Если между таблицами нет соединений, то их Join будет представлен в виде таблицы размером $table_1 \cdot table_2$, и скорее всего у нас есть более выгодные альтернативы.

Однако эта оптимизация может не всегда срабатывать.

```
SELECT *
    FROM sales
        JOIN customer ON sales.cust_id = customer.id
        JOIN sales_date on sales.date_id = sales_date.id
WHERE sales_date.date = '2024-02-04'
AND customer.city = 'Tver'
```

Согласно оптимизации мы должны сделать следующий порядок $customer \cdot sales \cdot date$, однако в таком случае придётся держать в памяти большую таблицу. С другой стороны можно в начале соединить `customer` и `date`, получив маленькую табличку, а затем связать с `sales`.

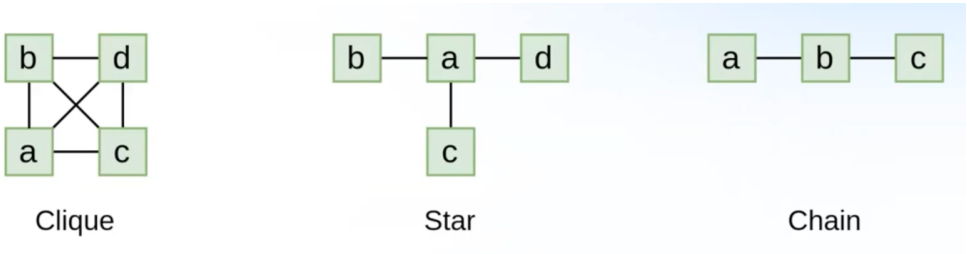

Рис. 3. Возможные топологии соединений

Данная оптимизация по-разному влияет на разные топологии. Например, клике она никак не помогает, а вот для `Chain` — количество порядков уменьшается, так для 10 таблиц надо будет рассмотреть лишь 2,5 миллиона вариантов.

Динамическое программирование

Снизу вверх

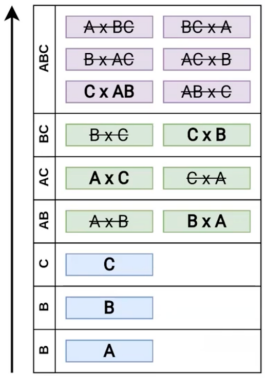

Рис. 4. Bottom up подход

```
for (i = 2 .. N) {                                          // Рассматриваем все поддеревья, содержащие 2 отношения, затем 3, ...
  for (j = 1 .. i) {                                        // Перечисляем все подгруппы
    List<Group> leftGroups = generateGroups(j)              // Сгенерировать все группы эквивалентности размером 2: AB, AC, BC
    List<Group> rightGroups = generateGroups(i - j)         // Сгенерировать все группы эквивалентности размером 1: A, B, C
    for (leftGroup : leftGroups, rightGroup : rightGroups) {  // Для каждой пары групп
      counter++
      if (connected(left, right) && notIntersecting(left, right)) { // Рассматриваем только непересекающиеся связанные группы
        connectedCounter++
        Cost leftCost = DP[leftGroup].cost                  // Получить стоимость левой группы из DP table
        Cost rightCost = DP[rightGroup].cost                // Получить стоимость правой группы из DP table
        Group group = combine(left, right)                  // leftGroup=[AB], rightGroup=[C], group=[ABC]
        order, cost = createNewOrder(group, leftCost, rightCost)  // Посчитать стоимость нового плана
        if (cost < getGroupCost(group)) {                   // Если новый порядок дешевле предыдущего, сохранить его в DP table
          DP[group] = { order, cost }
        }
      }
    }
  }
}
```

Рис. 5. DPSize

- Строим оптимальные группы для каждой из выборок размера $\leq k$.
- Переходим к $k+1$ группам, перебирая левую группу (размера $\leq k$) и автоматически подобранным к ней правым группам.

- Хорошо работает для разрешенных графов.
- Порядок обхода **A, B, C, AB, AC, BC, ABC**

Другим вариантом `Bottom up` подхода является **DPSub** — перебор масок.

```
for (i = 1 ... 2^(N-1)) {                                  // Генерируем [0,0,0,0], [0,0,0,1], [0,0,1,0], ...
  Group group = getGroupFromMask(i)                        // Выбираем группу эквивалентности по маске: [0,1,1,1] + [D,C,B,A] -> [C,B,A]
  List<Group> leftGroups = generateSubGroups(group)        // Генерируем все возможные подгруппы: A, B, C, AB, AC, BC, ABC
  for (leftGroup : leftGroups) {
    counter++
    Group rightGroup = complement(group, leftGroup)        // Генерируем комплементарную подгруппу: [ABC] - [AB] -> [C]
    if (connected(leftGroup, rightGroup)) {                // Рассматриваем только связанные группы
      connectedCounter++
      Cost leftCost = DP[leftGroup].cost                   // Получить стоимость левой группы из DP table
      Cost rightCost = DP[rightGroup].cost                 // Получить стоимость правой группы из DP table
      order, cost = findBestOrder(group, leftCost, rightCost) // Выбрать [left x right] или [right x left]
      if (cost < getGroupCost(group)) {                    // Если новый порядок дешевле предыдущего, сохранить его в DP table
        DP[group] = { order, cost }
      }
    }
  }
}
```

Рис. 6. DPSub

- Хорошо работает для графов с большим количеством рёбер.
- Порядок обхода **A, B, AB, C, AC, AB, ABC**.

Проблемы подхода:

- Всё также NP полный.
- Каждый раз когда красный счётчик увеличивается, а зелёный нет — мы делаем бесполезную работу.
- Не позволяет эффективно учитывать свойства, которые приходят сверху. Например, мы можем не сохранить сортировку для будущего Join, который был бы нам ясен выше и не получится сделать MergeJoin.

Сверху вниз

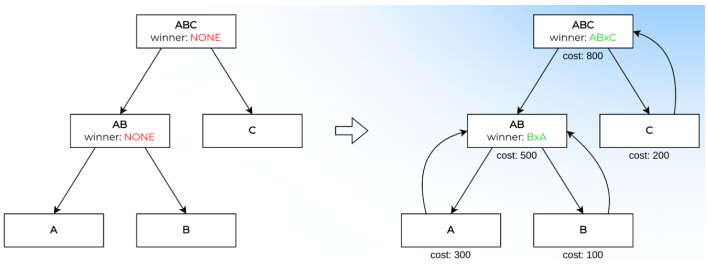

Рис. 7. Граф для top-down алгоритма.

В данном подходе мы используем MEMO.
Оптимизации:

- **Upper-bound pruning** - не добавлять оператор в MEMO, если ясно, что его стоймость уже выше оставшегося бюджета. Например, если мы знаем, что на данный момент оптимальный план стоит 800, просканировать только A, стоит 300, то мы не будем рассматривать планы, дороже 500.
- **Lower-bound pruning** - не исследовать группу дальше, если ясно, что её стоймость уже не улучшить. (Однако такое на практике такое почти нигде не сделано, так как не ясно до конца как оценить).
- Запускать параллельно, после того, как разделили на группы.

Проблемы:

- Сложнее для реализации.
- Потребляют больше памяти (из-за MEMO).

```
void optimizeGroup(group, properties) {
  if (memo[group, properties] != NULL)                              // Проверяем MEMO, что бы не оптимизировать группу дважды
    return
  List<Group> leftGroups = generateSubGroups(group)                 // Генерируем все возможные подгруппы: A, B, C, AB, AC, BC, ABC
  for (leftGroup : leftGroups) {
    Group rightGroup = complement(group, leftGroup)                 // Генерируем комплементарную подгруппу: [ABC] - [AB] -> [C]
    counter++
    if (connected(leftGroup, rightGroup)) {                         // Рассматриваем только связанные группы
      connectedCounter++
      Properties leftProperties = derive(properties, leftGroup)     // Рекурсивно оптимизируем группу слева
      optimizeGroup(leftGroup, left Properties)
      Properties rightProperties = derive(properties, rightGroup) // Рекурсивно оптимизируем группу справа
      optimizeGroup(rightGroup, rightProperties)
      order, cost = findBestOrder(group, leftGroup, rightGroup)    // Выбираем [left x right] или [right x left]
      if (cost < bestCost)
        memo[group, properties] = { order, cost }                   // Сохраняем в MEMO более дешевый план
    }
  }
}
```

Рис. 8. Наивная реализация top down алгоритма

Современная реализация алгоритма

Планирование join с помощью правил

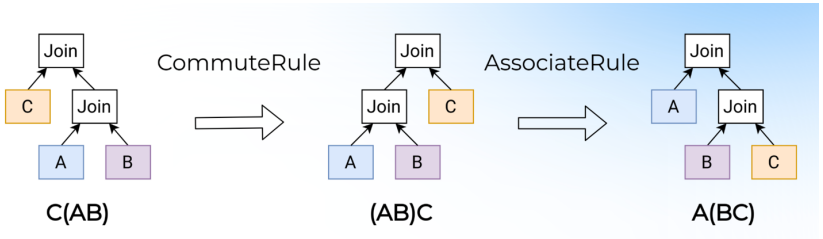

Рис. 9. Примеры правил

- `CommuteRule`: AB → BA
- `AssociateRule`: (AB)C → A(BC)
- `RotateRule`: (AB)C → (AC)B

`CommuteRule` и `AssociateRule` достаточно для генерирования остальных правил.

Однако применяя правило мы постоянно будем приходить в состояния в которых уже бывали. Причём обычно бороться с этим гораздо хуже, чем работать с предыдущими вариантами.

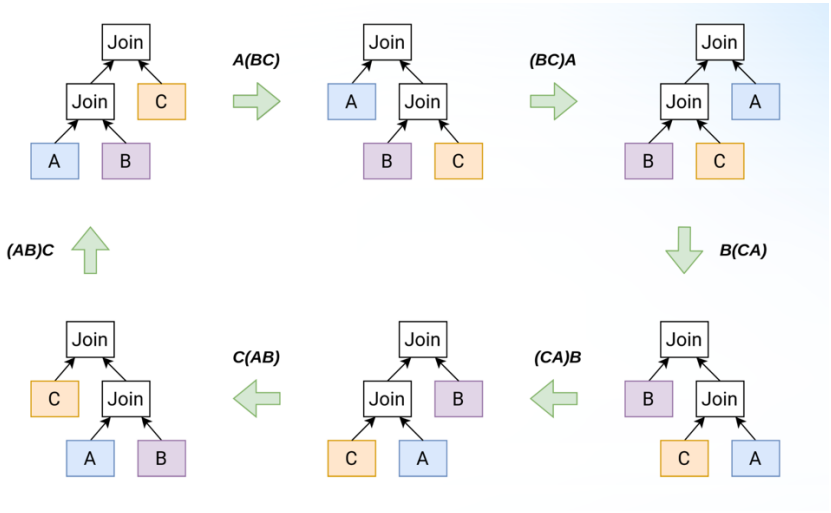

Рис. 10. Пример прихода в цикл используя правила.

Как бы хотелось

- Генерировать все возможные планы.
- Top-Down обход оптимально использует свойства.
- Top-Down эффективно реализует branch-and-bound pruning.

Как на самом деле

- Генерируем все планы для маленьких запросов.
- Рассмотрение всех свойств происходит эвристически, не всегда мы используем свойства оптимально.
- Branch-and-bound pruning мало где реализован.
- Legacy мешает.