Incorporating Partitioning and Parallel Plans into the SCOPE Optimizer

0. Introduction

此论文描述作者怎么教会 SCOPE 优化器考虑并行相关信息，生成合理的并行计划。

SCOPE 是用来做数据分析的系统，SCOPE 优化器会将用户输入的 SQL-Like 语句，转换成执行计划，并下发到集群的机器上执行。

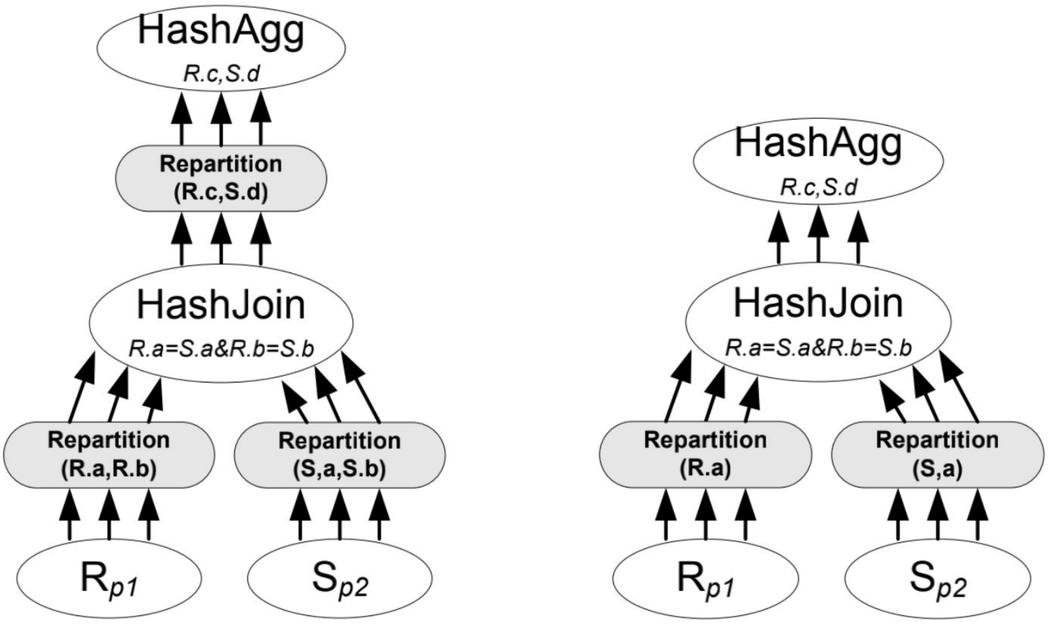
我们先想象一下，让一个原本不考虑并行信息的优化器，能够生成并行计划，最简单的方式就是在生成计划后，单独插入一个步骤（post-processing step）来考虑并行信息，把这个计划变成并行的。

比如最简单粗暴的做法就是此步骤中，在每两算子间插入数据（exchange）交换用的算子，然后把原本的算子直接并行起来。

这样简单的做法很难得到优秀的执行计划，原因是对数据做交换时，会改变原有数据性质（如有序性等），这些性质改变会深度的影响执行计划，因此，仅增加一个后续步骤来考虑并行，是远远不够的。

下面是一个例子：

```
select R.c, S.d, count(*)
from R, S
where R.a = S.a and R.b = S.b and p1(R) and p2(S)
group by R.c, S.d
```



(a) Always partition (b) Fewer partition operators

计划 a 就是最简单的做法，在所有算子中插入数据交换算子。

不同的情况下 a 和 b 各有优劣，这取决于众多因素，如：数据量大小，R.a 和 S.a 的分布，Join 的选择率等。

且计划 b 的正确性还需要一个条件来保证：R.c 函数依赖 R.a（后面会介绍）。

这些因素都应该被进行全盘的考虑，优化器需要准确的理解“并发”概念，及其相关的信息，仅仅通过一个简单的事后步骤就想处理好这个问题，是不够的。

接下来将介绍作者怎么教会 SCOPE 优化器准确的理解并发，分为这几个步骤：

1. 梳理不同的数据交换方式
2. 对 SCOPE 优化器框架做一个简单介绍
3. 形式化的对数据性质做定义及推导一些变换规则
4. 梳理不同的分片和合并方式对数据性质的影响
5. 梳理不同物理算子对数据性质的要求
6. 怎么进行数据性质的匹配

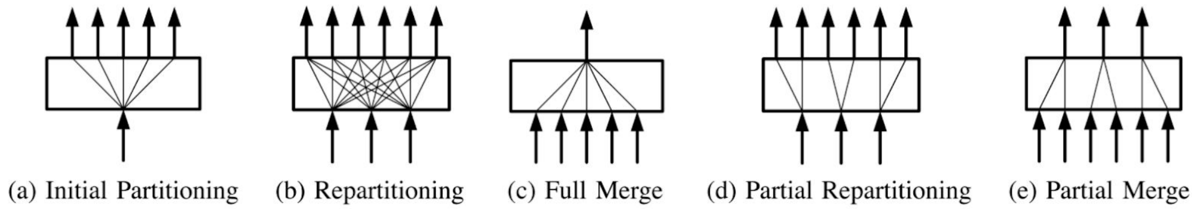
7. 为优化器加入生成并行计划的规则
8. 实验效果

其中 3~5 是理论基础，这部分定义较多，本文尽量做出解释，并给出实例，方便大家理解。

1. Parallel Plans And Exchange Operators

Exchange Topology

先简单梳理下不同的数据交换方式。



如上图，把数据交换分为了上面 5 种类型。

实现时通过组合不同的 Partition 和 Merge 算子完成。

Partitioning Schemes

Partition 算子把输入一分为多个分片，所有的 Partition 算子都是 FIFO 模式的。

Partition 有 4 种类型（物理实现）：

1. Hash Partitioning：根据某些列的 hash 值，决定输出的分片位置；
2. Range Partitioning：根据某些列进行连续的划分，如原输入为 [1, 2, 3, 4, 5, 6, 7]，划分成 3 个 partition 可能得到 {[1, 2], [3, 4, 5,], [6, 7]}；
3. Non-determinnistic Partitioning：结果没有任何性质保证，如直接随机对数据进行划分；
4. Broadcasting：把输入数据复制到每个输出口；

Merging Shcemes

Merge 算子把多个输入分片合并成一个。

Merge 有 4 种类型（物理实现）：

1. Random Merge：每次随机从某个分片读入一行进行合并；
2. Sort Merge：每次选取能读入的最小那一行，如有 2 分片 {[1, 4], [2, 8]}，输出将为 [1, 2, 4, 8]；
3. Concat Merge：随机把一个分片读取完，然后再读取另一个，如输入有 2 份为 {[1, 4], [2, 8]}，输出可能为 [1, 4, 2, 8] 或者 [2, 8, 1, 4]；
4. Sort-Concat Merge：根据每份输入的第一行数据，来决定读取顺序，然后按照 Concat Merge 的方式处理；如输入有 3 份为 {[1], [7, 10], [14, 51]}，最后结果为 [1, 7, 10, 14, 51]；

这里 Sort-Concat Merge 主要是为了和 Range Partition 进行配合，可以比较低成本的保持数据原有的有序性。

2. Property Reasoning Inside the Optimizer

这里对 SCOPE 优化器流程做一个简单的介绍，直接上流程图，然后再做解释：

Algorithm 1: OptimizeExpr($expr$, $reqd$)

```
Input: Expression expr,    ReqdProperties reqd
Output: QueryPlan plan
/*Enumerate all the possible logical rewrites  */
LogicalTranform(expr);
foreach logical expression lexpr do
    /*Try out implementations for its root
      operator                                   */
    PhysicalTranform(lexpr);
    foreach expression pexpr that has physical
    implementation for its root operator do
        ReqdProperties reqdChild =
        DetermineChildReqdProperties(pexpr, reqd);
        /*Optimize child expressions             */
        QueryPlan planChild =
        OptimizeExpr(pexpr.Child, reqdChild);
        DlvdProperties dlvd =
        DeriveDlvdProperties(planChild);
        if PropertyMatch(dlvd, reqd) then
            EnqueueToValidPlans();
        end
    end
end
plan = CheapestQueryPlan();
return plan;
```

这是一个比较典型的优化过程，流程大概如下：

1. 调用 LogicalTransform 对原有的表达式进行逻辑改写
2. 对上一步得到的每个新表达式，尝试找出合理的物理实现，步骤如下：
    1. 调用 PhysicalTransform 尝试找到他的物理实现，然后对于每一个找到的物理实现：
        1. 推导它对子节点的性质要求，得到 reqdChild
        2. 递归的优化他的子节点
        3. 根据子节点的结果，推导目前最终的性质 dlvd
        4. 判断 dlvd 是否满足父亲传入的性质要求 reqd
        5. 如果满足，则放入到计划池内

3. 在计划池内选取一个代价最低的计划，并返回

3. Property Formalism

这一部分主要是对数据性质做一个形式化的定义，并在定义上推导一些规则。

FDs, Constraints and Equivalence

先简单介绍下函数依赖和等价类的概念。

函数依赖：对于某表，如果当某些列 R{C1, C2, ... Cn} 的值相同时，另一些列 S{C1', C2', ... Cm'} 的值也一定相同，则认为 R 函数决定 S，或 S 函数依赖于 R，写作 R->S。

就相当于有这么一个函数，作用于 R 所在的列，能够得到 S 列上的值。

一个简单的例子，如果某个表有 3 列 {C1, C2, C3}，逻辑上他们的关系为 C2 = C1*2, C3 = C1%2，那就可以说 C1 -> {C2, C3}。

根据函数依赖的定义，可以推导出下面几个性质：

1. Trivial FDs: 有两个列集合 R，R'，如果 R' 为 R 的子集，则有 R -> R'；
2. Key contraints: 如果列集合 X 为某个表 T 的主键，则 X 可以函数决定其他列；
3. Column equality constraints: 经过包含 C1 = C2 的 Join 后，可以认为 {C1} 和 {C2} 相互函数决定；
4. Constant constraints: 经过包含 Ci = constant 的 Selection 算子后，可以认为空集 {} -> Ci；
5. Grouping columns: 经过某个列集合 R 的聚合后，可以认为 R 函数决定其他列，因为 R 上没有重复的值了；

上面几个性质都比较简单，不做过多论述了。

等价类：如果某些列，在某个表的所有行上，值都一样，则认为他们是等价类。

如果两列 C1, C2 属于一个等价类，则他俩可以任意互换，而不会影响查询结果。

Structural Properties

这个小节我们形式化的对数据性质（Structural Properties）做一些定义。

1. Grouping

Definition IV.1 (Grouping) A sequence of rows $r_1, r_2, \ldots, r_m$ is grouped on a set of columns $\mathcal{X} = \{C_1, C_2, \ldots, C_n\}$, if $\forall r_i, r_j, i < j, r_i[\mathcal{X}] = r_j[\mathcal{X}] \Rightarrow \forall k, i < k < j, r_k[\mathcal{X}] = r_i[\mathcal{X}]$. We denote grouping by $\mathcal{X}^g$.

大白话说就是聚合列相同的行，需要挨在一起。

如有数据 (1, 1), (2, 2), (2, 1)，现按 c1 聚合，那下面几种情况都是符合：

- [(1, 1), (2, 1), (2, 2)]
- [(1, 1), (2, 2), (2, 1)]
- [(2, 1), (2, 2), (1, 1)]
- [(2, 2), (2, 1), (1, 1)]

只要 (2, 2), (2, 1) 这两行紧挨在一起即可。

2. Sorting

Definition IV.2 (Sorting) A sequence of rows $r_1, r_2, \ldots, r_m$ is sorted on a column C in an ascending (or descending) order, if $\forall r_i, r_j, i < j \Rightarrow r_i[C] \leq r_j[C]$ (or $r_i[C] \leq r_j[C]$). We denote this ordering by C^o where $o \in \{o_\uparrow, o_\downarrow\}$.

就是所有行按照排序列，升序或者降序排序。

同样是上面的数据，按 c1 升序排列，符合这个性质的情况有：

- [(1, 1), (2, 1), (2, 2)]
- [(1, 1), (2, 2), (2, 1)]

3. Non-ordered partitioning：

Definition IV.5 (Non-ordered Partitioning) A relation $\mathcal{R}$ is *non-ordered partitioned* on columns $\mathcal{X}$, if it satisfies the condition

$$\forall r_1, r_2 \in \mathcal{R}: \ r_1[\mathcal{X}] = r_2[\mathcal{X}] \Rightarrow \mathrm{P}(r_1) = \mathrm{P}(r_2)$$

where P denotes the partitioning function used.

就是分片列相同的行，会被分到同一个分片内。

比如有数据 (1, 1), (1, 2), (2, 2), (2, 6), (3, 7), (4, 3)，按照 c1 分成 2 片：

- {[(1, 1), (1, 2), (3, 7)], [(2, 2), (2, 6), (4, 3)]}

只要 (1, 1) 和 (1, 2)，(2, 2) 和 (2, 6) 他们分别在一个分片内即可。

4. Ordered partitioning:

Definition IV.6 (Ordered Partitioning) A relation $\mathcal{R}$ is *ordered-partitioned* into partitions $\mathrm{P}_1, \mathrm{P}_2, \ldots, \mathrm{P}_m$ on columns $\{C_1^{o_1}, C_2^{o_2}, \ldots, C_n^{o_n}\}$ where $o_i \in \{o_\uparrow, o_\downarrow\}$, if it satisfies the condition in the previous definition and the additional condition

$$\forall \mathrm{P}_i, \mathrm{P}_j, i \neq j: (\forall r_1 \in \mathrm{P}_i, r_2 \in \mathrm{P}_j : r_1 <_{C_1^{o_1}, C_2^{o_2}, \ldots, C_n^{o_n}} r_2) \ or$$
$$(\forall r_1 \in \mathrm{P}_i, r_2 \in \mathrm{P}_j : r_1 >_{C_1^{o_1}, C_2^{o_2}, \ldots, C_n^{o_n}} r_2).$$

就是任意两个分片，要么其中一个的最大值小于另一个的最小值，要么相反。

还是上面数据，按照 c1 分成 2 片，下面是一个符合性质的结果：

- {[(2, 2), (2, 6), (1, 1), (1, 2)], [(4, 3), (3, 7)]}

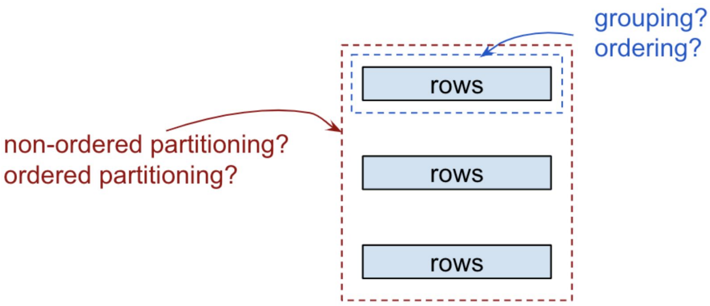
接下来是一个反例：

- {[(1, 1), (1, 2), (3, 7)], [(2, 2), (2, 6), (4, 3)]}

因为第一个分片内的最大值小于第二个分片的最小值，不符合上述条件。

5. Global/Local Properties

这里对上面提到的 4 种数据性质做一个思考，如下图：



可以总结出下面几个结论：

- 其中 grouping 和 ordering 用来描述一批“挨着”的数据，而 partitioning 用来描述多份“分散”的数据；
- 认为 grouping 和 ordering 是局部性质（local property），而 partitioning 是全局性质（global property）；
- 可以把 non-ordered partitioning 当做全局的 grouping，把 ordered partitioning 当做全局的 ordering。

6. Structural Properties

Definition IV.7 (Structural Properties) The structural properties of a relation $\mathcal{R}$ can be represented by partitioning information and an ordered sequences of actions,

$$\{\mathcal{P}^{\theta}; \{\hat{A}_1, \hat{A}_2, \ldots, \hat{A}_n\}\}$$

我们把局部性质和全局性质写在一起，用来完整的定义数据性质。

分别用 g 和 o 来表示 grouping 和 ordering；

左边部分用来表示全局的 partitioning 性质，如：

- {C1}^g：表示按照 C1 做了 non-ordered partitioning
- {C1^o, C2^o}：表示按照 C1 和 C2 做了 ordered partitioning

右边部分表示局部性质 grouping, sorting，如：

- {C3^o}：表示按照 C3 排序
- {{C1, C2}^g}：表示按照 C1, C2 聚合
- {{C1, C2}^g, C3^o}：先按照 C1, C2 聚合，然后在每个 group 内再按照 C3 排序

需要注意的是，如有多个 local 性质，需要在前一个的基础上，满足下一个。

如数据为 (1, 1, 10), (1, 1, 2), (2, 2, 1)，满足上面 case 3 的情况有：

- [(1, 1, 2), (1, 1, 10), (2, 2, 1)]
- [(2, 2, 1), (1, 1, 2), (1, 1, 10)]

可见上面两种情况中个，整体来看 C3 都不是有序的，但是在各自的 group 内，C3 是有序的。

7. Examples

Partition 1	Partition 2	Partition 3
{1,4,2}, {1,4,5}, {7,1,2}	{4,1,5}, {3,7,8}, {3,7,9}	{6,2,1}, {6,2,9}

上面是一个例子，这份数据的性质可以表示为：

$$\{\{C_1\}^g; \{\{C_1, C_2\}^g, C_3^o\}\}$$

因为他们按照 C1 做了 non-ordered partitioning。

然后在每个 partition 内，先是按照 C1, C2 做了聚合，接着按照 C3 做了排序。

Inference Rules

接下来，在我们上面的形式化定义上，推导一些转换规则；

规则 1

$$\{*; \{\hat{A}_1, \ldots, \hat{A}_{m-1}, \hat{A}_m\}\} \Rightarrow \{*; \{\hat{A}_1, \ldots, \hat{A}_{m-1}\}\} \tag{1}$$

这个规则比较简单，按照上小节说明：需要在上一个的基础上满足下一个；因此局部性质是可以后缀裁剪的；

比如 [(2, 2, 1), (1, 1, 2), (1, 1, 10)] 局部性质满足 {{C1, C2}^g, C3^o}，则肯定满足 {{C1, C2}^g}。

规则 2

$$\{\{C_1, C_2, \ldots, C_m\}^g; *\} \Rightarrow \{\{C_1, C_2, \ldots, C_m, C_{m+1}\}^g; *\} \tag{2}$$

第二个表示 non-ordered partitioning 有后缀扩展性。

相当于我们的分片函数不考虑最后一列，只要前缀相同则放入一个分片内，最后一列对分片结果无影响。

比如 {[(1, 1), (1, 2)], [(2, 2), (3, 3)]} 全局性质满足 {{C1}^g}，他也满足 {{C1, C2}^g}。

规则 3

$$\{\{C_1^o, C_2^o, \ldots, C_m^o\}; *\} \Rightarrow \{\{C_1^o, C_2^o, \ldots, C_m^o, C_{m+1}^o\}; *\} \tag{3}$$

第三个表示 ordered partitioning 有后缀扩展性，证明如下：

假设我们先按照 C1 做了 ordered partitioning，现在考虑加入 C2 列，对于任意两行 r 和 r' 分两种情况：

- 如果 C1 != C1'，那 (C1, C2) 和 (C1', C2') 的有序性和 (C1), (C1') 相同，C2 对分片间的有序性无影响，这个时候看 C1 和 C1' 就够了；
- 如果 C1 == C1'，那按照 C1 划分时这两行一定会在一个分片内，C2 列对分片间的有序性无影响，因为他们都在一个分片中了；

比如 {[(1, 1), (1, 2)], [(2, 1)]} 全局性质满足 {{C1}^o}，那也满足 {{C1, C2}^o}。

注意：

对于 grouping 性质，列的顺序是无关的，体现在写法上就是把 g 写在大括号外；

而 sorting 性质列的顺序会有影响，另外每个列可能有不同的升序/降序，写法上在每个列需要单独有一个 o。

规则 4-5

$$\{*;\{\hat{A}_1,\ldots,C^o,\ldots,\hat{A}_m\}\} \Rightarrow \{*;\{\hat{A}_1,\ldots,C^g,\ldots,\hat{A}_m\}\} \quad (4)$$
$$\{\{C_1^o,C_2^o,\ldots,C_n^o\};*\} \Rightarrow \{\{C_1,C_2,\ldots,C_n\}^g;*\} \quad (5)$$

这两条规则来自于“如果一批数据他们有 sorting 性质，那在相应列上也有 grouping 性质”。

这两条规则比较简单，口述证明下 (5)：

- “如果数据满足左边的性质，则 C1, C2 ... Cn 列相同的行，一定在同一个分片内，则也一定满足右边的性质”。

规则 6

$$\exists\ C \in \mathcal{X} : (\mathcal{X} - \{C\}) \rightarrow C, \mathcal{X}^g \Rightarrow (\mathcal{X} - \{C\})^g \quad (6)$$

这个规则表示列集合 X 满足 grouping 性质，则通过函数依赖消除（简化）内部多余的列后，也满足 grouping 性质，证明如下：

1. 如果 C 能被 (X-{C}) 函数决定，说明当 X 除 C 外其他列相同时，C 值一定相同；（结论 1）
2. 接下来用反证法，假设上述规则最右边部分不成立，则一定有如下情况：
    1. 存在 3 行 r1, r2, r3，且根据 X 做 group 时，他们的顺序为 [r1, r2, r3]；
    2. 而按照 (X-{C}) 做 group 时，r2 不应该在他们的中间顺序；

3. 接着证明 2 中的情况不存在：
    1. 根据 grouping 性质的定义，如果 r2 不应该在 r1 和 r3 中间，需要 r1 和 r3 在 (X-{C}) 列上的值相同，而和 r2 不同；
    2. 如果 r1 和 r3 在 (X-{C}) 列上的值相同（r1[X-{C}] = r3[X-{C}]），根据结论 1，则他们在 C 列上的值也相同，则 r1[X] = r3[X]；
    3. 由于根据 X 做 group 时的顺序是 [r1, r2, r3]，则有 r2[X] = r1[X]，则有 r2[X-{C}] = r1[X-{C}]；
    4. 则证明按照 (X-{C}) 做 group 时，r2 可以在 r1 和 r3 中间，上述情况不存在；

规则 7

$$
\begin{aligned}
\exists\ &\{C_1, \ldots, C_{j-1}\} \rightarrow C_j : \\
&\quad \{C_1^o, \ldots, C_{j-1}^o, C_j^o, C_{j+1}^o, \ldots\} \Rightarrow \\
&\qquad \{C_1^o, \ldots, C_{j-1}^o, C_{j+1}^o, \ldots\}
\end{aligned} \tag{7}
$$

这个规则表示在包含多个列的一组 sorting 性质中，如果某列能被前缀列函数决定，则去掉改列后 sorting 性质仍然满足，简单证明如下：

我们把包含 Cj 的那组 sorting 性质设为 P1，不包含的那组设为 P2；

对于任意两行 r1, r2，他们满足 P1 时，有下面两个情况：

1. 如果 r1[C1, ... Cj-1] != r2[C1, ... Cj-1]，则 Cj 对他们的相对顺序无影响，看前面的列就够了，于是他俩根据 {C1 ... Cj-1, Cj+1 ...} 排序时，顺序一定和根据 {C1, ... Cj, ...} 相同，也就是一定也满足 P2；
2. 如果 r1[C1, ... Cj-1] == r2[C1, ... Cj-1]，由于函数依赖，一定有 r1[Cj] == r2[Cj]，则 Cj 对他们的相对顺序无影响，和上面类似，根据 P1 和 P2 包含的列排序时，顺序一定相同，也就一定也满足 P2。

4. Deriving Structural Properties

接下来看经过不同类型的 parition 和 merge 算子后，数据性质会被怎样的改变。

Properties After a Partitioning Operator

我们假设 Partition 算子处理数据是是 FIFO 模式的，因此数据的局部顺序在处理前后是一致的，所以 Partition 算子只影响全局性质（partitioning）。
我们先看各种类型 partition 对性质的影响，再选取其中部分来做分析：

INPUT WITH PROPERTIES $\{\mathcal{X};\mathcal{Y}\}$

Scheme	Result
Hash on $C_1,\ldots,C_n$	$\{\{C_1,\ldots,C_n\}^g;\mathcal{Y}\}$
Range on $C_1^o,\ldots,C_n^o$	$\{\{C_1^o,\ldots,C_n^o\};\mathcal{Y}\}$
Non-Deterministic	$\{\emptyset;\mathcal{Y}\}$
Broadcast	$\{\top;\mathcal{Y}\}$

- 在 {C1, C2, ..., Cn} 上做 Hash，形成的 global property 为 {C1, C2, ..., Cn}^g；
- 在 {C1, C2, ..., Cn} 上做 Range，形成的 {C1^o, C2^o, ..., Cn^o}；
- Non-Deterministic 模式下，全局性是空集，得不到任何性质保证；
- Broadcast 模式下，全局性的符号表示把数据被全量复制，每个 partition 都有全量数据；

上面四种情况中，局部性质 Y 都没有被改变。

Properties After a Merge Operator

Merge 算子对数据性质的影响，取决于 Merge 的类型和数据原本的局部性质。
还是先贴结论，然后选一部分来做分析：

STRUCTURAL PROPERTIES OF THE RESULT AFTER A FULL MERGE

	Input Properties $\{\mathcal{X}^g;\mathcal{Y}\}$	Input Properties $\{\mathcal{X}^o;\mathcal{Y}\}$
Random merge	$\{\perp;\emptyset\}$	$\{\perp;\emptyset\}$
Sort merge on $\mathcal{S}^o$	1). $\{\perp;\mathcal{S}^o\}$ if $\mathcal{Y}\Rightarrow\mathcal{S}^o$ 2). $\{\perp;\emptyset\}$ otherwise	1). $\{\perp;\mathcal{S}^o\}$ if $\mathcal{Y}\Rightarrow\mathcal{S}^o$ 2). $\{\perp;\emptyset\}$ otherwise
Concat merge	1). $\{\perp;\{\mathcal{X}^g,\mathcal{Z}\}\}$ if $\mathcal{Y}\Rightarrow\{\mathcal{X}^g,\mathcal{Z}\}$ 2). $\{\perp;\emptyset\}$ otherwise	1). $\{\perp;\{\mathcal{X}^g,\mathcal{Z}\}\}$ if $\mathcal{Y}\Rightarrow\{\mathcal{X}^g,\mathcal{Z}\}$ 2). $\{\perp;\emptyset\}$ otherwise
Sort-concat merge on $\mathcal{S}^o$	1). $\{\perp;\{\mathcal{X}^g,\mathcal{Z}\}\}$ if $\mathcal{Y}\Rightarrow\{\mathcal{X}^g,\mathcal{Z}\}$ 2). $\{\perp;\emptyset\}$ otherwise	1). $\{\perp;\mathcal{Y}\}$ if $\mathcal{S}^o\Leftrightarrow\mathcal{X}^o$ and $\mathcal{Y}\Rightarrow\mathcal{S}^o$ 2). $\{\perp;\emptyset\}$ otherwise

这里都是 full merge，所以被聚合后全局性都没了，倒 T 符号表示数据处于 non-partitioned 状态。

Random merge

这个没什么好说的，局部性质都没了。

Sort Merge

Y => S^o 表示输入数据的局部性 Y 能推导出 S^o，也就是输入数据已经在 S 相关的列上有序，则最后聚合结果也满足 S^o。

比如输入数据有 2 个 partition，满足性质 {C1^g; C2^o}：

{[(1, 1), (3, 3), (1, 5)], [(2, 2), (2, 10)]}

现在按照 C2 做 Sort Merge，结果为：

[(1, 1), (2, 2), (3, 3), (1, 5), (2, 10)]

其满足 {; C2^o}

Concat Merge

简单来说就是 Y 包含 X^g 时，能保留一些性质，这是由 Concat Merge 的执行方式决定的，直接看例子吧：

比如输入数据满足 {{C1}^g; {C1^g, C2^o}}：

- {[(1, 1), (1, 2), (3, 1)], [(2, 3), (2, 5)]}

聚合的两种结果为：

- [(1, 1), (1, 2), (3, 1), (2, 3), (2, 5)]
- [(2, 3), (2, 5), (1, 1), (1, 2), (3, 1)]

都满足 {; {C1^g, C2^o}}；

Sort-Concat Merge

在输入数据是 non-partitioned 时，和 Concat Merge 一样；

但是输入数据是 ordered partitioned(range partition) 时，如果：

1. Y => S^o，表示输入数据局部性已经按照 S 相关的列有序；
2. S^o <=> X^o，输入数据按照聚合列做了 ordered partitioning；

则结果可以把原有的有序性保留下来。

主要是可以和 Range Partition 配合，保留原有的有序性，下面是一个最简单的例子：

例如输入数根据 C1 做了 range partition(ordered partition)，局部按照 C1 排序：

{[1, 2, 3], [6, 7, 8], [4, 5]}

经过 Sort-concat Merge 后结果为：

{[1, 2, 3, 4, 5, 6, 7, 8]}

C1 上的有序性得以保留。

5. Deriving Requested Properties

接下来分析每种算子对输入数据不同的性质要求。

每种算子分为两种模式 partitioned 和 non-partitioned，不同模式有不同要求。

这里先直接贴出最后的结论，然后再选几个进行分析：

Required Structural Properties of Inputs to Physical Operators

<table>
<tr><th></th><th>Non-Partitioned Version</th><th>Partitioned Version</th></tr>
<tr><td>Table Scan</td><td>$\{\perp; *\}$</td><td>$\{\mathcal{X}; *\}, \mathcal{X} \neq \emptyset$</td></tr>
<tr><td>Select</td><td>$\{\perp; *\}$</td><td>$\{\mathcal{X}; *\}, \mathcal{X} \neq \emptyset$</td></tr>
<tr><td>Project</td><td>$\{\perp; *\}$</td><td>$\{\mathcal{X}; *\}, \mathcal{X} \neq \emptyset$</td></tr>
<tr><td>Sort on $\mathcal{S}^o(\mathcal{S} \neq \emptyset)$</td><td>$\{\perp; \{\mathcal{S}^o, *\}\}$</td><td>$\{\mathcal{X}^o; \mathcal{S}^o\}, \mathcal{X} \neq \emptyset, \mathcal{S}^o \Rightarrow \mathcal{X}^o$</td></tr>
<tr><td>Hash Aggregate on $\mathcal{G}$</td><td>$\{\perp; *\}$</td><td>$\{\mathcal{X}; *\}, \emptyset \subset \mathcal{X} \subseteq \mathcal{G}, \mathcal{G} \neq \emptyset$</td></tr>
<tr><td>Stream Aggregate on $\mathcal{G}$</td><td>$\{\perp; \{\mathcal{G}^g, *\}\}$</td><td>$\{\mathcal{X}; \{\mathcal{G}^g, *\}\}, \emptyset \subset \mathcal{X} \subseteq \mathcal{G}, \mathcal{G} \neq \emptyset$</td></tr>
<tr><td>Nested-loop or Hash Join (equijoin on columns $\mathcal{J}_1 \equiv \mathcal{J}_2$)</td><td>Both inputs $\{\perp; *\}$</td><td>Pair-wise Join:
Input 1: $\{\mathcal{X}; *\}, \emptyset \subset \mathcal{X} \subseteq \mathcal{J}_1$; Input 2: $\{\mathcal{Y}; *\}, \emptyset \subset \mathcal{Y} \subseteq \mathcal{J}_2$; $\mathcal{X} \equiv \mathcal{Y}$
Broadcast Join:
Input 1: $\{\top; *\}$; Input 2: $\{\mathcal{X}; *\}, \mathcal{X} \neq \emptyset$</td></tr>
<tr><td rowspan="2">Merge Join (equijoin on columns $\mathcal{J}_1 \equiv \mathcal{J}_2$)</td><td>Input 1: $\{\perp; \mathcal{S}_1^o\}$
Input 2: $\{\perp; \mathcal{S}_2^o\}$</td><td>Pair-wise Join:
Input 1: $\{\mathcal{X}; \mathcal{S}_1^o\}, \emptyset \subset \mathcal{X} \subseteq \mathcal{J}_1$; Input 2: $\{\mathcal{Y}; \mathcal{S}_2^o\}, \emptyset \subset \mathcal{Y} \subseteq \mathcal{J}_2$; $\mathcal{X} \equiv \mathcal{Y}$
Broadcast Join:
Input 1: $\{\top; \mathcal{S}_1^o\}$; Input 2: $\{\mathcal{Y}; \mathcal{S}_2^o\}, \mathcal{Y} \neq \emptyset$</td></tr>
<tr><td colspan="2">$\mathcal{J}_1 = prefix(\mathcal{S}_1^o), \mathcal{J}_2 = prefix(\mathcal{S}_2^o)$</td></tr>
</table>

HashAgg

在 non-partitioned 模式下，数据都在一起，不需要对数据有性质要求。

在 partitioned 模式下，需要要求聚合列的值相同的行，在同一个分片中（否则需要对数据再进行一次 re-partition），这个要求的形式化表示就是：输入数据的全局性质 X 包含的列，被被聚合的列集合 G 所包含。

"X 被 G 所包含"其实这里蕴含了两个推论：

- 根据推导规则(2) + grouping 性质列顺序无关，于是按照 G 内的部分列做了 group，就已经能满足 G 所有列值相同的行都在一个分配内；
- 根据推导规则(5)，按照 X 做了 ordered-partitioning，其实也就相当于做了 non ordered-partitioning，所以这里并没有刻意标识 X 需要是 o 或者 g；

TableScan/Select/Project

他们对数据没有任何性质要求，即使在 partition 模式下也是。

StreamAgg

在 non-partitioned 模式下，需要数据在 G 上满足 grouping 性质，因此 local property 部分是 {G^g, *}；🤔

在 partition 模式下，对全局性质的要求，和 HashAgg 相同，局部性质上，还要求在 G 上满足 grouping 性质。

Join

Join 有两个子节点，为了分别处理两个子节点都 partitioned 和只有一个 partitioned 的情况，实现了两种方式：

1. Pair-wise：要求两个子节点都是 partitioned，且数据已经按照 join 列准备好，每个 Join 节点只需要处理传递上来的 partitioned 数据；
2. Broadcast：要求一个节点是 partitioned，一个不是，执行方式是把 non-partitioned 的数据广播到所有 partitioned 节点进行计算；

于是，对于 Nested-loop Join 和 Hash Join：

1. 在 non-partitioned 模式下（既两个儿子都是 non-partitioned），数据都在一起，没有任何要求；
2. partitioned 的 Broadcast 模式下（有一个是 partitioned），直接把一边的数据全量广播，没有任何要求；
3. partitioned 的 Pair-wise 模式下（两儿子都是 partitioned），要求左右儿子的数据的 partition 方式要和 join column 兼容（也就是 join column 相同的行，在一个 partition 内）；

对于 Merge Join 而言，整体和 Nested-loop Join 和 Hash Join 类似，只是对局部性新增了有序性的要求。

6. Property Matching

Property matching 主要是用来检查某个性质 P1 是否能够满足另一个性质 P2。

因为任意性质 P 都是由全局性质和局部性质组成，所以只要：

1. P1 局部性质满足 P2 的局部性质
2. P1 全局性质满足 P2 的全局性质

则认为 P1 能够满足 P2。

整个过程其实就是利用之前我们的推导规则，对性质进行转换，看能否得到另一个性质。

直接看一个例子，目前有：

1. P1 = {{C7, C1, C3}^g; {C6^o, C2^o, C5^o}}
2. P2 = {{C1, C2, C4}^g; {{C1, C2}^g}}

且有如下条件：

1. {C6, C2} --> {C3} (函数决定)
2. 有两个等价类 {C1, C6}， {C2, C7}

现在对 P1 进行转换：

1. 利用等价类，对 C7, C6 进行替换，得到 {{C2, C1, C3}^g; {C1^o, C2^o, C5^o}}；
2. 利用等价类和函数依赖，得到 {C1, C2} --> {C3}；
3. 利用规则(6)，可以剔除 C3，得到 {{C2, C1}^g; {C1^o, C2^o, C5^o}}；
4. 利用规则(2) + grouping 列顺序无关，可以得出 P1 目前全局性 {C2, C1}^g，满足 P2 的全局性质 {C1, C2, C4}^g；
5. 利用规则(1)，剔除 P1 局部性质中的 C5，得到 {C1^o, C2^o}；
6. 利用规则(5)，得到 P1 的局部有序性 {C1^o, C2^o} 满足 P2 的 {C1, C2}^g；

由于 P1 的全局和局部性质都能满足 P2，于是 P1 满足 P2。

7. Enforcer Rules

这里我们添加一条规则，用来生成并行的计划。

这条规则会被优化器在 LogicalTransform 阶段考虑，使得优化器无缝开始考虑并发的执行计划。

下面贴出这条规则的伪代码，再进行分析：

Algorithm 2: EnforceDataExchange($expr$, $reqd$)

```
Input: Expression expr,    ReqdProperties reqd
ReqdProperties reqdNew;
if Serial(reqd) then
    /*Require a serial output                              */
    AddExchange(FullMerge);
    reqdNew = GenParallel(reqd);
    Optimize(expr, reqdNew);
else
    /*Require a parallel output                            */
    /*Enumerate all possible partitioning
      properties that P_min ⊆ P ⊆ P_max                    */
    foreach valid partition schema P do
        /*Case 1: repartition                              */
        AddExchange(Repartition);
        /*Generate new partitioning requirements
          for its children; remove specific
          partitioning columns                             */
        reqdNew = GenParallel(reqd);
        Optimize(expr, reqdNew);
        /*Case 2: initial partition                        */
        AddExchange(InitialPartition);
        /*Force the child to generate a serial
          output                                           */
        reqdNew = GenSerial(reqd);
        Optimize(expr, reqdNew);
    end
end
return;
```

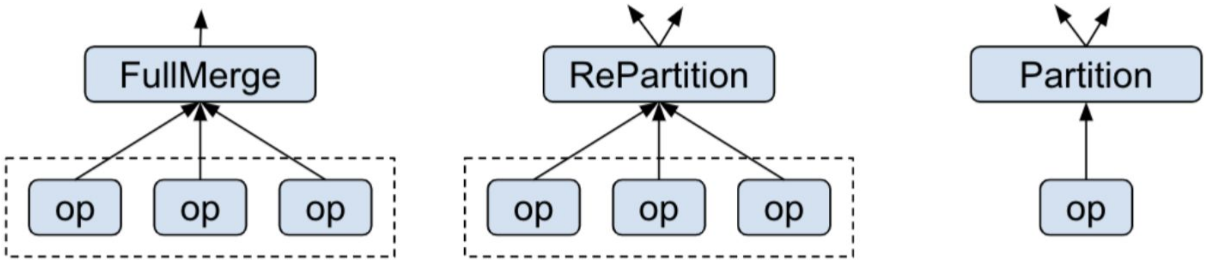
输入：1）需要被优化的表达式，2）外部需要的性质；

这条规则用来在普通算子中间，插入 Merge/Partition 算子，整个流程如下：

1. 如果要求最后的模式是 serial (non-partitioned) 的：
    1. 则在顶部加一个 FullMerge，使得对外显得是 non-partitioned 模式；
    2. 递归的优化，要求生成一个 partitioned 的方案；
2. 遍历所有可能的 Partition 方案，并做下面两个操作：
    1. 添加 repartition 算子，用于保证最后的数据满足 reqd 里要求的全局性，然后递归生成一个 partitioned 计划；

2. 添加 partition 算子，用于保证最后的数据满足 reqd 里要求的全局性，然后递归生成一个 non-partitioned 计划。

下面这个图分别对应上面的 1.1~1.2, 2.1, 2.2：



1. 1.1~1.2 使得即使外部要求为 non-partitioned，算子本身也能并行起来；
2. 2.1 使得算子并行的时候不用考虑最外部对 partition 性质的要求，如最外部要求 partition 数为 2，算子本身的并发可以任意进行，最后外部性质会被 repartition 保证；
3. 2.2 使得即使外部要求为 partitioned，算子本身也能选择非并行的方式执行；

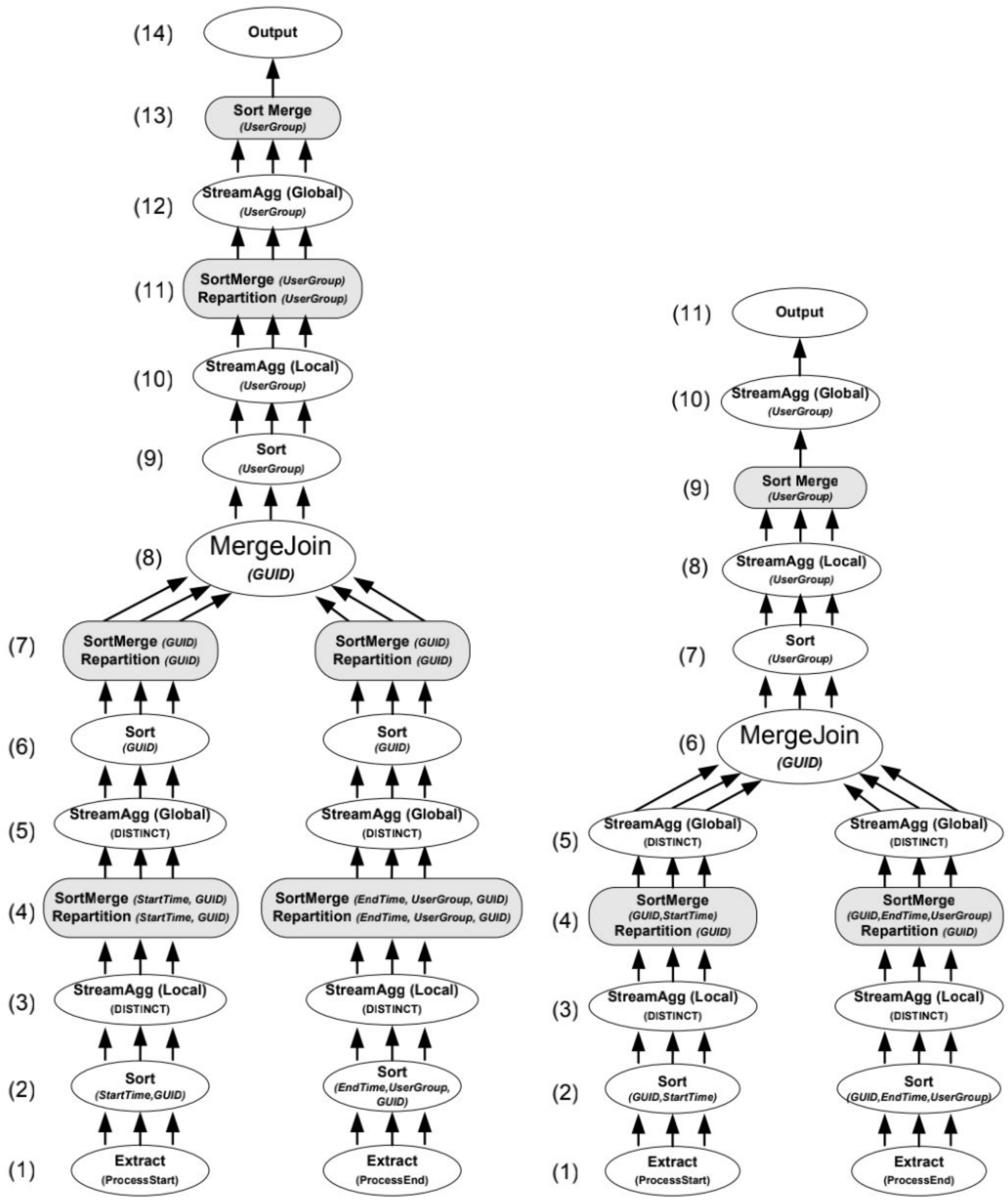
8. The Optimizer in Action

原文大概是想执行如下一条 SQL：

```
SELECT SUM(end.Time-start.Time), UserName
  (SELECT DISTINCT Time, GUID FROM ProcessStart) start,
  (SELECT DISTINCT Time, GUID, UserName FROM ProcessEnd) end
WHERE start.GUID = end.GUID
GROUP BY UserName
ORDER BY UserName;
```

就是有 start 和 end 表用来记录用户操作的开始和结束时间，需要统计这些用户的累计操作时长。

下面是优化前后的 Plan 对比：



(a) Default Plan (b) Optimized Plan

初始计划在每两个算子之间，都插入了 Repartition 算子，使得每个算子都并行了起来，最后执行了 21 分钟。

优化后，在算子 4 的地方，根据规则 (2)，可得按照 GUID 进行 partition 后，就能满足第一次 StreamAgg 对 {Time, GUID}，{Time, GUID, UserName} 的聚合性质要求，于是在后面需要按照 GUID 做 MergeJoin 时，就不需要再 repartition 一次了。

优化后的计划最终跑了 10 分钟，性能提升了一倍。