# 并行数据库系统中的位置感知分区

并行数据库系统对大量结构化数据进行水平分区，以便为无共享集群中的分析工作负载提供并行数据处理能力。水平分区大量数据时的一个主要挑战是降低给定工作负载和数据库模式的网络成本。降低并行数据库系统中网络成本的一种常见技术是根据连接键对表进行共同分区，以避免昂贵的远程连接操作。 然而，现有的分区方案在这方面受到限制，因为除非表被完全复制，否则只有共享相同连接键的复杂模式中的表的子集可以被共同分区。在本文中，我们提出了一种新颖的分区方案，称为基于谓词的参考分区（简称 PREF），它允许根据给定的连接谓词对表集进行共同分区。 此外，基于 PREF，我们提出了两种自动分区设计算法来最大化数据局部性。一种算法仅需要模式和数据，而另一种算法还需要工作负载作为输入。在我们的实验中，我们表明，我们的自动化设计算法可以对不同复杂度的数据库模式进行分区，因此与现有的分区方法相比，有助于有效减少给定工作负载下的查询运行时间。

动机：现代并行数据库系统（例如 SAP HANA [5]、Greenplum [21] 或 Terradata [15]）和其他并行数据处理平台对大量数据进行水平分区，以便为分析查询提供并行数据处理能力。

水平分区数据时的一个主要挑战是在执行分析查询时实现高数据局部性，因为过多的数据传输会显着减慢商用硬件上的查询执行速度[19]。减少分析工作负载网络成本的常用技术是在 1990 年代由第一个并行数据库系统引入的，即在连接键上对表进行共同分区，以避免昂贵的远程连接操作 [8, 10]。然而，在具有许多表的复杂模式中，此技术仅限于共享相同连接键的表子集。此外，完全复制表仅适用于小型表。因此，对于现有的分区方案，对于使用不同连接键在多个表上连接路径的复杂分析查询，远程连接通常是不可避免的。

引用分区 [9]（或简称 REF 分区）是一种更新的分区方案，它将一个表与另一个由传出外键（即引用约束）引用的表进行共同分区。 例如，如图 1 所示，如果表 CUSTOMER 在其主键 custkey 上进行哈希分区，则可以使用表 CUSTOMER 表的传出外键 (fk) 对表 ORDERS 进行共同分区。 因此，使用 REF 分区，可以对通过外键链接的表链进行共同分区。 例如，表 LINEITEM 也可以按表 ORDERS 进行 REF 分区。 但是，REF 分区不支持与外键不同的其他连接谓词，甚至传入外键。 例如，图 1 中的表 SUPPLIER 不能通过表 LINEITEM 进行 REF 分区。

贡献：在本文中，我们提出了一种新颖的分区方案，称为基于谓词的参考分区（简称 PREF）。 PREF 专为批量加载数据的分析工作负载而设计。 PREF 分区方案概括了 REF 分区方案，以便可以通过引用另一个表的给定连接谓词（称为分区谓词）对表进行共同分区。 在图 1 中，表 SUPPLIER 因此可以通过表 LINEITEM 使用属性 suppkey 上的等连接谓词作为分区谓词进行 PREF 分区。 为了实现分区谓词的完整数据局部性，PREF 可能会在不同分区中引入重复的元组。 例如，当如前所述对表SUPPLIER进行PREF分区并且suppkey属性的相同值出现在表LINEITEM的多个17个分区中时，则表SUPPLIER中的引用元组将被复制到SUPPLIER的所有对应分区。 这样，使用分区谓词作为连接谓词的连接可以在每个节点本地执行。 然而，在最坏的情况下，PREF 分区方案可能会导致表的完全复制。 我们的实验表明，这只是具有大量表的复杂模式的罕见情况，可以通过我们的自动分区设计算法来避免。

此外，使用我们的 PREF 分区方案手动找到给定数据库模式的最佳分区方案以最大化数据局部性是一个难题。 现有的自动化设计算法 [14,18,20] 不知道我们的 PREF 分区方案。 因此，作为第二个贡献，我们提出了两种了解 PREF 的分区设计算法。 我们的第一个算法是模式驱动的，并假设模式中的外键代表工作负载的潜在连接路径。 我们的第二个算法是工作负载驱动的，并且还考虑了一组查询。 主要思想是首先为共享相似表集的查询子集单独找到最佳分区配置，然后增量合并这些分区配置。 在我们的实验中，我们表明，通过使用 PREF 分区方案，我们的分区设计算法优于现有的自动化设计算法，后者依赖于与数据库优化器的紧密集成（即，获取给定工作负载的估计成本）

概述：在第 2 节中，我们介绍 PREF 分区方案的详细信息，并讨论有关查询处理和批量加载的详细信息。 然后，在第 3 节中，介绍了我们的模式驱动的自动分区设计算法。 然后，第 4 节描述了工作负载驱动的算法，并讨论了减少搜索空间的潜在优化。 我们对 TPCH [3] 和 TPC-DS 基准 [2] 的综合实验评估将在第 5 节中讨论。我们选择这两个基准是因为我们想要展示我们的算法如何适用于具有均匀分布数据的简单模式（TPC -H) 以及具有倾斜数据的复杂模式 (TPC-DS)。 最后，我们在第 6 节中总结相关工作，并在第 7 节中进行总结

2. 基于谓词的引用分区

下面，我们首先介绍基于谓词的参考分区方案（简称 PREF）的详细信息，然后讨论对 PREF 分区表执行查询以及批量加载这些表的重要细节。 在符号方面，我们使用大写字母表示表（例如，表 T），使用小写字母表示各个元组（例如，元组 t ∈ T）。 此外，如果表 T 被划分为 n 个分区，则各个分区由 Pi(T) 标识（其中 1 ≤ i ≤ n）。

我们将 S 称为被引用表，将 R 称为引用表。 引用的表可以再次进行 PREF 分区。 PREF 分区表 R 的种子表是分区谓词路径中未进行 PREF 分区的第一个表 T。

如前所述，可以对引用的表使用任何分区方案（例如散列、范围、循环甚至 PREF）。 为简单起见但不失一般性，我们在本文的其余部分仅使用 HASH 和 PREF 分区方案。 此外，仅支持简单的等连接谓词（以及简单等连接谓词的合取）作为分区谓词 p，因为其他连接谓词通常会导致 PREF 分区表的完全冗余（即，一个元组可能会被 分配给 R 的每个分区）。

示例：图 2 显示了分区前（上半部分）和分区后（下半部分）的数据库示例。 在示例中，表 LINEITEM 已进行哈希分区，因此分区后没有重复项。 表 ORDERS (o) 由表 LINEITEM (l) 使用连接键 (orderkey) 上的分区谓词进行 PREF 分区； 即，ORDERS 是引用表，LINEITEM 是引用表以及种子表。 对于表 ORDERS，PREF 分区方案引入了重复项，以通过连接键 (orderkey) 实现潜在的等连接的完整数据局部性。 此外，表 CUSTOMER (c) 使用连接键 (custkey) 上的分区谓词按 ORDERS 进行 PREF 分区； 即，CUSTOMER 是引用表，ORDERS 是引用表，而 LINEITEM 是 CUSTOMER 表的种子表。 同样，PREF 对 CUSTOMER 表进行分区会导致重复。 此外，我们可以看到没有订单的客户（custkey=3）也被添加到分区表CUSTOMER中（在第一个分区中）。

因此，通过使用 PREF 分区方案，只要查询图中不存在循环，查询的给定联接路径中的所有表就可以共同分区。 然而，为给定模式和工作负载中的所有表找到最佳分区方案，以最大化 PREF 方案下的数据局部性，是一项复杂的任务，将在第 3 节和第 4 节中讨论。

对于查询处理，当 PREF 将表 R 按 S 分区时，我们创建两个附加位图索引： 第一个位图索引 dup 指示每个元组 r ∈ R 是否是第一次出现（由位图索引中的 0 指示）或者如果 r 是重复项（由位图索引中的 1 表示）。 这样，在查询处理过程中就可以轻松消除 PREF 分区产生的重复项，这将在 2.2 节中进行更详细的解释。 第二个索引 hasS 指示对于每个元组 r ∈ R 是否存在满足 p(r, s) 的元组 s ∈ S。 这样，可以优化反连接和半连接。 图 2 中的示例显示了两个 PREF 分区表的这些索引。 有关如何使用这些索引进行查询处理的详细信息将在下面的 2.2 节中讨论。

2.2 查询处理

下面，我们讨论如果给定 SQL 查询 Q 中包含 PREF 分区表，则需要如何重写查询以确保正确性。这包括添加用于消除 PREF 分区表产生的重复项的操作以及添加重新分区操作以实现正确的并行查询执行。

此外，我们还讨论了用于优化 SQL 查询的重写（例如，使用反连接或外连接来优化查询）。 所有这些重写规则都应用于查询 Q 的编译计划 P。目前，我们的重写规则仅支持 SPJA 查询（选择、投影、连接和聚合），而通过单独重写每个 SPJA 查询块来支持嵌套 SPJA 查询。+

重写过程：重写过程是一个自下而上的过程，它决定每个运算符 o ∈ P 在执行之前是否必须将不同操作或重新分区操作（即洗牌操作）应用于其输入。 运营商o.

请注意，我们的 DISTINCT 运算符与 SQL DISTINCT 运算符不同。 我们独特的运算符仅消除由我们的 PREF 方案生成的那些重复项。 可以使用析取过滤谓词消除 PREF 分区中的重复项，该谓词对（中间）结果中元组的每个 dup 属性使用条件 dup=0。 然而，仍然可以使用元组的属性来执行普通的 SQL DISTINCT 运算符来查找具有相同值的重复项。 在本文的其余部分中，我们始终参考不同运算符的语义。 此外，重新分区运算符还会在通过网络重新排列元组之前消除 PREF 分区产生的重复项。

为了决定是否必须应用不同操作或重新分区操作，重写过程为运算符 o ∈ P 的每个中间结果注释两个属性。在下文中，我们还使用变量 o 来指代中间结果 由运算符 o 产生的结果。 • Dup(o)：定义（中间）结果o 是否不含PREF 分区产生的重复项（Dup(o) = 0）或否（Dup(o) = 1）。 对于表格，我们使用相同的符号； 即，Dup(T) 定义表 T 是否因 PREF 分区而包含重复项。 对于哈希分区表 T，我们得到 Dup(T) = 0。 • Part(o)：定义（中间）结果 o 的分区方案，包括分区方法 Part(o).m (HASH, PREF, 或 NONE（如果其他方案都不成立）、分区属性列表 Part(o).A 和分区数量 Part(o).c。 同样，对于表格，我们使用相同的符号； 即 Part(T) 定义表 T 的分区方案。

下面，我们分别讨论 SPJA 查询的每种类型运算符的重写规则。 此外，我们假设计划 P 的最后一个运算符始终是投影运算符，可用于消除 PREF 分区产生的重复项。 我们不讨论选择运算符，因为不需要将额外的重复消除或重新分区运算符添加到其输入中（即，可以在不应用任何这些重写的情况下执行选择运算符）。

内部等连接 o = (oin1 ✶oin1.a1=oin2.a2 oin2)：我们应用于内部等连接的唯一重写规则是在其输入 oin1 和 oin2 上添加额外的重新分区运算符。 下面我们讨论三种不需要添加重分区运算符的情况。

(1) 如果两个输入都经过哈希分区并且它们使用相同数量的分区（即 Part (oin1).c = Part art(oin2).c 成立），则第一种情况成立。 此外，Part (oin1).A = [a1] 和 Part (oin2).A = [a2] 也必须成立（即，连接键用作分区属性）。

(2) 如果 Part(oin1).m = HASH 且 Part(oin2).m = PREF，则第二种情况成立，而连接谓词 a1 = a2 必须是 PREF 方案的分区谓词。 此外，分区方案P art(oin1) 必须是用于PREF 方案P art(oin2) 的种子表的方案。

(3) 第三种情况成立，如果 Part(oin1).m = PREF 且 Part(oin2).m = PREF，而连接谓词 a1 = a2 必须是一个输入的 PREF 模式的分区谓词（称为引用 输入）。 另一个输入称为参考输入。 此外，两个 PREF 方案必须引用相同的种子表。

否则，如果这三种情况都不成立，则重写过程应用重新分区运算符以确保两个输入都使用哈希分区方案，其中连接键是分区属性，并且两个方案使用相同数量的分区。 如前所述，重新分区运算符还消除了 PREF 方案产生的重复项。 如果一个输入已经进行了哈希分区（使用连接键作为分区属性），那么我们只需要相应地重新分区另一个输入。

在讨论了重写规则之后，我们现在介绍如何通过重写过程设置属性 Dup(o) 和 Part(o)。 如果我们像之前讨论的那样添加重新分区操作，那么我们使用重新分区运算符的哈希分区方案来初始化 Part(o) 并设置 Dup(o) = 0，因为我们消除了重复项。 如果我们不添加重新分区操作（即之前的情况 1-3），我们将 Part(o) 初始化如下： 在情况 (1) 中，我们将 Part(o) 设置为哈希值 输入之一的分区方案（请记住，两个输入使用相同的分区方案）并且 Dup(o) = 0，因为哈希分区表从不包含重复项。 在情况（2）和（3）中，我们使用参考输入的 PREF 方案来初始化 Part(o)。 此外，在情况 (2) 中，我们始终设置 Dup(o) = 0。在情况 (3) 中，如果引用的输入没有重复项，我们设置 Dup(o) = 0。 否则，我们设置 Dup(o) = 1。

其他连接：虽然等连接可以在分区输入上执行（如前所述），但其他连接（例如叉积 o = (oin1 ×oin2) 和 theta 连接 o = (oin1 ✶p oin2) ）与任意连接谓词 p 需要 作为远程连接执行，将整个较小的关系传送到所有集群节点。 对于这些连接，我们设置 Part(o).m = NONE 。 此外，我们还消除了两个输入中的重复项并设置 Dup(o) = 0。此外，外连接可以计算为内等连接和反连接的并集。 本节末尾介绍了反连接的有效实现。

聚合 o = χGrpAtts,AggF uncs(oin)：如果输入运算符 oin 的分区方案是哈希分区，并且如果条件 GrpAtts.startWith ith(Part(oin).A) （即 group-by 属性列表 以 或 开头的分区属性列表相同），那么我们不需要对输入重新分区。 否则，添加一个重新分区运算符，通过 GrpAtts 对输入进行哈希分区。 此外，如果 Dup(oin) = 1 成立，则重新分区运算符还会消除重复项（如前所述）。 最后，如果没有添加重新分区运算符，则重写过程将 Part(o) 设置为 Part(oin)。 否则，它将 Part(o) 设置为用于重新分区的哈希分区方案。 此外，在任何情况下它都会设置 Dup(o) = 0 。

图 3 显示了对图 2 所示的分区数据库进行聚合查询的示例。在该示例中，联接的输出是 PREF 分区的，并且包含重复项（如前所述）。 因此，聚合的输入必须使用 group-by 属性 c.cname 上的哈希分区方案（用作其输出的分区方案）重新分区。 此外，重新分区运算符消除了 PREF 产生的重复项。

2.3 批量加载

如前所述，PREF 方案是为批量加载新数据的数据仓库场景而设计的。 下面，我们讨论如何在引用表 S 的 PREF 分区表 R 上执行插入。我们假设引用的表 S 已经被批量加载。

为了将新的元组 r 插入到表 R 中，我们必须识别必须插入元组 r 的副本的那些分区 Pi(R)。 因此，我们需要识别引用表 S 的那些包含分区伙伴的分区 Pi(S)（即满足给定元组 r 的分区谓词 p 的元组 s）。 例如，在图 2 中，表 CUSTOMER 是引用表 ORDERS 的 PREF 分区。 将 custkey = 1 的客户元组插入到表 CUSTOMER 中时，必须将副本插入到所有三个分区中，因为 ORDERS 的所有分区都有一个 custkey = 1 的元组。

为了在不执行 R 与 S 的连接的情况下有效地实现新元组的插入操作，我们在表 S 的引用属性上创建分区索引。分区索引是基于哈希的索引，它将唯一属性值映射到分区号 i。 例如，对于图 2 中的表 ORDERS 模式，我们在属性 custkey 上创建一个分区索引，该索引映射例如 custkey = 1 到分区 1 到 3。我们在第 5 节的实验中表明，分区索引有助于有效地执行新元组的批量加载。

最后，对 PREF 分区表的更新和删除将应用于所有分区。 但是，我们不允许更新修改 PREF 方案的分区谓词中使用的那些属性（无论是在引用表中还是在引用表中）。 由于连接键在数据仓库场景中通常不会更新，因此此限制不会限制 PREF 的适用性。

3. 模式驱动的自动分区设计

在本节中，我们将介绍用于自动分区设计的模式驱动算法。 我们首先讨论问题陈述，然后简要概述我们的解决方案。 随后，我们将介绍如何最大化数据局部性同时最小化数据冗余的重要细节。

3.1 问题陈述和概述

问题陈述可以表述为以下优化问题：给定模式 S（包括引用约束）和非分区数据库 D，为每个表 T ∈ S 定义分区方案（HASH 或 PREF）（称为分区配置）

这样，就参考约束上的等连接操作而言，所得分区数据库 D P 中的数据局部性最大化，同时数据冗余最小化。 换句话说，虽然主要优化目标是在给定分区方案下最大化数据局部性，但在具有相同最高数据局部性的物化配置中，应该选择数据冗余最小的配置。

请注意，在上面的问题陈述中，我们不认为完全复制是表的可能选择。 原因是完全复制只适用于小表，而PREF可以为其他无法完全复制的表找到分区和完全复制之间的中间立场。 此外，在应用我们的设计算法之前，可以将作为完全复制候选者的小表从给定的数据库模式中排除。 为了解决前面提到的优化问题，我们的算法执行以下三个步骤。

第一步是为给定模式 S 创建一个无向标记和加权图 GS = (N, E, l(e ∈ E), w(e ∈ E))（称为模式图）。 虽然节点 n ∈ N 表示表，但边 e ∈ E 表示 S 中的参照约束。此外，标记函数 l(e ∈ E) 定义每条边的等连接谓词（从参照约束导出） ) 和加权函数 w(e ∈ E) 定义了需要在该边上执行远程连接时的网络成本。 边的权重 w(e ∈ E) 定义为与边 e 连接的较小表的大小。 这背后的直觉是，通过边缘 e 进行潜在远程连接的网络成本取决于较小表的大小，因为该表通常通过网络传输。 很明显，我们忽略了更复杂查询的选择性（使用选择运算符和多个连接），因此 w(e ∈ E) 仅表示上限。 然而，我们的实验表明，w(e ∈ E) 是一个很好的代理，可以代表总工作负载成本，即使对于具有复杂查询的工作负载也是如此。 图 4（左侧）显示了由缩放因子 SF = 1 的 TPC-H 模式的简化版本生成的模式图。

第二步，我们从 GS 中提取边 Eco 的子集，该子集可用于对 GS 中的所有表进行共同分区，从而最大化数据局部性。 对于给定的连接 GS，所需的边集 Eco 是最大生成树（或简称 MAST）。 使用 MAST 的原因是，通过从 GS 中丢弃具有最小权重的边，可以最小化潜在远程连接（即，不在 MAST 中的边）的网络成本，从而最大化上述定义的数据局部性。 此步骤的潜在结果如图 4（中）所示。

通常，对于连接的 GS，存在多个具有相同总重量的 MAST。 例如，在图 4 中，除了丢弃 SUPPLIER 和 NATION 之间的边之外，还可以丢弃 CUSTOMER 和 NATION 之间的边，因为该边具有相同的权重。 如果存在总重量相同的不同 MAST，则必须对每个 MAST 单独应用以下步骤。

最后，在最后一步中，我们枚举可应用于 MAST 的所有可能的分区配置，以找出哪种分区配置引入了最小的数据冗余。 最小化数据冗余很重要，因为这对查询的运行时间有直接影响（即使我们可以实现最大的数据局部性）。 我们在算法中枚举的分区配置都遵循相同的模式：选择 MAST 中的一个表作为使用哈希分区方案的种子表。 一般来说，它可以使用任何现有的分区方案，例如散列、循环或范围分区。 作为划分属性，我们使用边 e ∈ E 的标签 l(e) 中的连接属性，该属性连接到代表种子表的节点并具有最高权重 w(e)。 所有其他表都使用 MAST 中的边标签作为分区谓词在种子表上进行递归 PREF 分区。 图 4（右侧）显示了 MAST 的一种潜在分区配置，它使用 LINEITEM 表作为种子表。

3.2 最大化数据局部性

给定模式图 GS 的数据局部性 (DL) 和用于共同分区的边子集 Eco 定义如下：

DL = 1 意味着 Eco 包含 GS 中的所有边（即不需要远程连接），DL = 0 意味着 Eco 为空（即没有表与任何其他表共同分区）。 例如，如果我们根据主键对架构的所有表进行哈希分区，则数据局部性将为 0（只要表不共享相同的主键属性）。

为了最大化仅具有一个连接组件的给定模式图 GS 的数据局部性，我们基于给定权重 w(e ∈ E) 提取最大生成树 MAST。 MAST 中的边集表示所需的 Eco 集，因为向 MAST 添加一条边将导致循环，这意味着并非所有边都可以用于共同分区。 如果 GS 有多个连接组件，我们会提取每个连接组件的 MAST。 在这种情况下，Eco 表示所有最大生成树边缘的并集。

另一种解决方案（而不是提取 MAST）是在 GS 中复制表（即节点），以消除循环并允许一个表使用不同的分区方案。 但是，联接查询仍然可能需要远程联接。 例如，如果我们在图 4（左侧）的 GS 中复制表 NATION，则可以将 CUSTOMER 的 NATION 副本和 SUPPLIER 的 NATION 副本共同分区。 然而，使用连接路径 C − N − S 的查询仍然需要从边缘 C − N 或边缘 N − S 进行远程连接。因此，在我们的模式驱动算法中，我们根本不重复节点。

3.3 最小化数据冗余

最大化数据局部性后的下一步是为模式 S 中的所有表找到分区配置，这可以最大限度地减少分区数据库 D P 中的数据冗余。 因此，我们首先定义数据冗余（DR）如下：

定义为分区后所有表 T ∈ S 的大小之和（记为 TP）。 因此，DR = 0 表示分区后任何表均未添加数据冗余，而 DR = 1 表示分区后添加了 100% 的数据冗余（即 D 中的每个元组在 DP 中平均存在两次）。 将每个表完全复制到集群的所有 n 个节点会导致数据冗余 n − 1。

在清单 1 中，我们展示了算法的基本版本，用于枚举给定 MAST 的不同分区配置 (PC)。 为了简单起见（但不失一般性），我们假设模式图 GS 只有一个连接组件，并且只有一个 MAST。 否则，我们可以单独为每个 MAST 应用枚举算法。

枚举算法（清单 1 中的函数 findOptimalPC）获取 MAST 和非分区数据库 D 作为输入，并返回 D 中所有表的最佳分区配置。因此，该算法会分析与 MAST 中的节点一样多的分区配置（ 第 5-15 行）。 因此，我们构建遵循相同模式的分区配置（第 7-9 行）：使用一张表作为种子表，并通过其中一种种子分区方案（简称 SP）（例如哈希分区）和所有其他表进行分区 在 MAST 的边缘上递归地进行 PREF 分区（请参阅函数 addPREF）。 对于每个分区配置 newPC，我们最终估计应用 newPC 时分区数据库的大小，并将其与迄今为止的最佳分区配置进行比较（第 12-14 行）。 虽然我们的分区设计算法中的种子表从不包含重复的元组，但 PREF 分区表却包含重复的元组。 为了估计分区后数据库的大小，必须估计所有 PREF 分区表中的预期冗余。 冗余是累积的，这意味着如果 PREF 方案中的引用表包含重复项，则引用表也将继承这些重复项。 例如，在图 2 中，ORDERS 表中 orderkey = 1 的重复订单元组会导致引用 CUSTOMER 表中 custrkey = 1 的重复客户元组。 因此，为了估计给定表的大小，必须考虑直到种子（无冗余）表的所有引用表。 分区后表的大小估计的详细信息在附录 A 中进行了解释。

3.4 无冗余表

如3.3节所述，分区后给定表的最终大小由从种子表到分析表的所有边的冗余因子决定。 在具有许多表的复杂模式中，这可能会导致 PREF 分区表完全或接近完全冗余。 这是因为算法仅确定一张表为种子表，而所有其他表均进行 PREF 分区。

为了解决这个问题，我们的枚举算法还可以将用户给定的约束作为输入，这不允许单个表出现数据冗余。 因此，我们采用3.3节中描述的枚举算法如下：

(1)我们还枚举可以使用多个种子表的分区配置。 我们从具有一个种子表的配置开始，并将数量增加到 S 个种子表，直到满足用户给定的约束。 由于 MAST 的最大数据局部性随着种子表数量的增加而单调减少，因此一旦发现满足用户给定约束的分区配置，我们就可以尽早停止枚举。 该方案将是具有最大数据局部性且满足用户给定约束的分区方案。

(2) 我们尽早修剪分区配置，为用户给定的约束不允许数据冗余的表添加数据冗余。 这意味着对于我们不允许数据冗余的表，我们可以使用种子分区方案或 PREF 分区方案，其分区谓词引用没有数据冗余的表的主键。

4. 工作负载驱动的自动分区设计

在本节中，我们讨论工作负载驱动的自动分区设计算法。 同样，我们从问题陈述开始，然后简要概述我们的解决方案。 之后，我们讨论算法的细节。

问题陈述可以表述为以下优化问题：给定模式 S、工作负载 W = {Q1, Q2, ..., Qn} 和非分区数据库 D，定义分区方案（HASH 或 PREF）： 每个查询 Qi ∈ W 使用的表（称为分区配置），使得每个查询 Qi 的数据局部性最大化，而所有 Qi ∈ W 的数据冗余全局最小化。与模式驱动分区一样，主要优化目标 是在给定的分区方案下最大化数据局部性； 数据冗余只是次要的。 为了解决这个优化问题，我们的算法执行以下三个步骤。

在第一步中，我们的算法为每个查询 Qi ∈ W 创建一个单独的模式图 GS(Qi)，其中边表示查询中的连接谓词。 然后，我们计算每个 GS(Qi) 的 MAST(Qi)。 这样，每个查询 Qi ∈ W 的数据局部性被最大化，因为可以为每个查询单独生成一个最佳分区的最小数据库 D P (Qi)。 然而，这会导致非常高的数据冗余，因为各个表很可能会存在多次（对不同的查询使用不同的分区方案）。 例如，图 5（左侧）显示了工作负载 W = {Q1, Q2, Q3, Q4} 中四个不同查询产生的 MAST。 同样，如果一个查询存在具有相同总权重的不同 MAST，我们可以保留所有这些 MAST，以便在以下步骤中找到最佳解决方案。

在第二步中，我们合并各个查询的 MAST，以减少算法的搜索空间。 给定 MAST，合并函数会创建各个 MAST 中节点和边的并集。在此阶段，如果 Qj 的 MAST 完全包含在 Qi 的 MAST 中（即，我们将 MAST(Qj ) 合并为 MAST(Qi)）。 MAST(Qi) 包含具有与 MAST(Qj )) 相同标签和权重的所有节点和边。 因此，在此合并阶段不会发生循环。 合并的 MAST 表示为 MAST(Qi j ) 如果 MAST(Qj ) 完全包含在不同的 MAST 中，我们将其合并到这些 MAST 之一中。 此外，在第一个合并阶段结束时，我们确定最佳分区配置并估计每个合并的 MAST 的分区数据库的总大小（使用清单 1 中的函数 findOptimalPC）。 图 5（中）显示了第一个合并阶段的潜在结果。 该步骤有效地减少了后续合并阶段的搜索空间。

在最后一步（即第二个合并阶段）中，我们使用基于成本的方法来进一步合并 MAST。 在此步骤中，如果结果是非循环并且在减少数据冗余的同时不牺牲数据局部性，我们仅将 MAST(Qj ) 合并到 MAST(Qi) 图 5（右侧）显示了第二次合并的潜在结果 阶段。 在此示例中，Q3 和 Q4 的 MAST 被合并，因为合并后所得数据库 D P (Q3 4) 的大小小于各个分区数据库 D P (Q3) D P (Q4) 的大小之和。 对于查询执行，可以将查询路由到包含查询的 MAST，并且该查询读取的所有表的数据冗余度最小。

4.2 最大化数据局部性

为了最大化给定工作负载 W 的数据局部性，我们首先为每个查询 Qi ∈ W 创建一个单独的模式图 GS(Qi)，如前所述。 工作负载驱动算法的模式图的定义方式与第 3 节中描述的相同，即 GS = (N, E, l(e ∈ E), w(e ∈ E))，并且可以从以下查询图导出： 查询 Qi：查询图在文献中被定义为无向标记图 GQ = (N, E, l(e ∈ E))，其中每个节点 n ∈ N 代表一个表（由查询使用）。 边 e ∈ E 表示两个表之间的连接谓词，而标记函数 l(e ∈ E) 返回每条边的连接谓词。 目前，当将查询图 GQ(Qi) 转换为等效模式图 GS(Qi) 时，我们仅考虑那些使用等连接谓词作为标签的边。 请注意，这并不意味着工作负载 W 中的查询只能具有等连接谓词。 这仅意味着具有非等连接谓词的边不会添加到模式图中，因为当 PREF 用于共同分区表时，这些谓词无论如何都会导致高数据冗余，如第 2 节中所述。此外，用于创建模式图 GS，需要为GS定义一个权重函数w(e ∈ E)。 这是微不足道的，因为表大小是由非分区数据库 D 给出的，该数据库也是工作负载驱动算法的输入。 请注意，除了表大小之外，边权重 GS 还可以反映查询优化器执行连接的成本（如果提供了这些信息项）。 然而，合并函数将需要更复杂（即，节点和边的简单联合是不够的，因为相同的边可能具有不同的权重）。 下面，我们假设边权重代表表大小。 一旦为每个查询 Qi ∈ W 创建了模式图 GS(Qi)，我们就可以导出每个 GS(Qi) 的最大生成树 MAST(Qi)。 MAST(Qi) 表示可用于 Qi 中的共同分区表的边 Eco(Qi) 集合。 所有位于 Qi 查询图中但不在 MAST(Qi) 中的边都将导致远程连接。 因此，查询的数据局部性 DL 以与第 3.2 节中之前相同的方式定义为 Eco(Qi) 中的权重总和与 GS(Qi) 中所有边的权重总和的分数。 如第 3 节所示，使用 MAST 的边缘进行共同分区可以最大化数据局部性，除非我们另外允许复制表（即节点）以删除 GS(Qi) 中的循环。 此外，与模式驱动算法相比，如果连接的 GS 具有总权重相同的不同 MAST，我们的算法还会为每个 MAST 找到最佳分区配置，并估计分区数据库的大小，如清单 1 所示 对于后续的合并阶段，我们仅保留 MAST，这会产生具有最小估计大小的分区数据库。