

一种数据查询方法、设备、数据库系统及存储介质

5 本公开要求于 2023 年 12 月 22 日提交中国专利局、申请号为 202311786596.8、申请名称为“一种数据查询方法、设备、数据库系统及存储介质”的中国专利申请的优先权，其全部内容通过引用结合在本公开中。

技术领域

本公开涉及数据库技术领域，尤其涉及一种数据查询方法、设备、数据库系统及存储介质。

10 背景技术

分区是一种数据库优化技术，它可以将大的逻辑表按照一定的规则分成多个小的物理表，从而提高查询和维护的效率。在分区的过程中，数据库会将数据按照分区规则分配到不同的分区中，并且可以在分区中使用索引和其他优化技术来提高查询效率。其中，大的逻辑表即为分区表，小的物理表则为分区，每一个分区在存储引擎上独立组织管理数据和索引。在进行 join 查询时，如果两个表都是分区表，并且分区键与连接键相同或相关，则可以实现分区连接(partition-wise join)。

目前，partition-wise join 技术已与多机并行查询架构相结合，基于分区信息来进行查询任务切分与调度，从而使得多个节点只需要独立负责自身所分配到的分区上的 join 工作即可，从而减少节点间不必要的数据传输功耗。

20 但是，多机并行查询架构中的单个节点中通常还启动了多条工作进程(worker)来并行处理该节点所分配到的子任务。在单个节点内部，所拉取到的分区已经被降维为普通数据表，节点内需要依赖 shuffle(重分布)机制来进行节点内的多线程调度，众所周知，shuffle 机制的性能开销非常可观，这导致节点内的查询执行效率不佳。

发明内容

25 本公开的多个方面提供一种数据查询方法、设备、数据库系统及存储介质，用以改善多机并行查询场景下的查询执行效率。

本公开实施例提供一种查询方法，适应于并行处理查询任务的多个执行节点中的任一执行节点，所述查询任务的查询对象为目标分区表，所述方法包括：

30 在接收到从所述查询任务中拆分出的子任务后，确定所述子任务在所述目标分区表中分配到的目标分区；

配所述置目标分区与所述子任务所对应的各工作进程之间的调度关系；

在拉取到所述目标分区对应的数据后，基于所述调度关系，将拉取到的数据调度至与其所属分区相关联的工作进程上，以供工作进程执行查询。

本公开实施例还提供一种执行节点，包括存储器、处理器和通信组件；

35 所述存储器用于存储一条或多条计算机指令；

所述处理器与所述存储器和所述通信组件耦合，用于执行所述一条或多条计算机指令，以用于执行前述的查询方法。

40 本公开实施例还提供一种数据库系统，包括并行处理查询任务的多个执行节点，所述多个执行节点分别接收从所述查询任务中所拆分出的子任务，单个执行节点内为自身所接收到的子任务启动有多个并行的工作进程，单个执行节点用于执行前述的查询方法。

本公开实施例还提供一种存储计算机指令的计算机可读存储介质，当所述计算机指令被一个或多个处理器执行时，致使所述一个或多个处理器执行前述的查询方法。

在本公开实施例中，在多机并行查询架构下提出了一种改进的查询方案，在向多个并行的执行节点分发子任务之后，各个执行节点可分别确定出子任务在目标分区表中分配到的目标分区，这样，可将目标分区表中的分区信息透传至各个执行节点中。在此基础上，在执行节点内部，可配置所分配到的分区与执行节点内为所分配到的子任务所启动的各工作进程之间的调度关系，并基于调度关系将拉取到的数据调度至相应的工作进程上，也即是，在执行节点内部，可充分利用获知的分区信息来进行节点内的调度。这使得，执行节点内部的调度过程不再需要依赖 shuffle（重分布）机制，可有效节省节点内调度过程中的性能开销，进而改善多机并行查询场景下的查询执行效率。

附图说明

- 10 此处所说明的附图用来提供对本公开的进一步理解，构成本公开的一部分，本公开的示意性实施例及其说明用于解释本公开，并不构成对本公开的不当限定。在附图中：
- 图 1 为本公开一示例性实施例提供的一种数据库系统的结构示意图；
- 图 2 示出了一种节点间调度的示例性方案的逻辑示意图；
- 图 3 为本公开一示例性实施例提供的一种执行节点的内部逻辑结构示意图；
- 15 图 4 为本公开一示例性实施例提供的一种配置调度关系的实现方案逻辑示意图；
- 图 5 为本公开一示例性实施例提供的一种改进的查询方案的逻辑示意图；
- 图 6 为本公开另一示例性实施例提供的一种查询方法的流程示意图；
- 图 7 为本公开又一示例性实施例提供的一种执行节点的结构示意图。

具体实施方式

20 为使本公开的目的、技术方案和优点更加清楚，下面将结合本公开具体实施例及相应的附图对本公开技术方案进行清楚、完整地描述。显然，所描述的实施例仅是本公开一部分实施例，而不是全部的实施例。基于本公开中的实施例，本领域普通技术人员在没有做出创造性劳动前提下所获得的所有其他实施例，都属于本公开保护的范围。

25 在对本公开各实施例提供的技术方案进行详细说明之前，先对本公开中涉及到的几个技术概念进行简单解释如下。

分区：是一种数据库优化技术，它可以将大表按照一定的规则分成多个小表，从而提高查询和维护的效率。在分区的过程中，数据库会将数据按照分区规则分配到不同的分区中，并且可以在分区中使用索引和其他优化技术来提高查询效率。

30 分区表：在数据库中，可以使用分区表来实现分区。分区表是指将一个大表拆分成多个小表，每个小表称为一个分区，分区表中包含多个分区，每个分区可对应一个独立的物理存储空间。

分区连接（PartitionWise Joins）：还可翻译为按分区的连接，可理解为根据连接条件（连接键），将分区表之间的连接，打散成相互匹配的分区与分区之间的连接。当正在连接两个表都在连接键上进行分区时，可以应用按分区的连接。

35 多机并行查询：可理解为多个执行节点并行处理同一个查询任务。实际应用中，可将该查询任务拆分为多个子任务并分配至多个执行节点上，多个执行节点可并行执行自身分配到的子任务，多个子任务的输出结果可进行聚合，以产生出查询任务的输出结果。单个执行节点上则可提供多个工作进程来并行处理所分配到的子任务。

40 Shuffle（重分布）：可简单理解为针对数据表中的每条数据记录都要计算哈希值，然后再基于哈希值决定应该调度到哪个节点或进程上，当查询任务涉及到多个数据表时，多个数据表都需要全表进行 shuffle。在 hash join 算子中，也会涉及到 shuffle，大致逻辑为：对需要进行 hash join 的两个数据表分别进行全表计算哈希值，针对其中一个表中的每一条数据记录分别在另一张表中执行一次全表探测，以发现是否符合 join 条件。众所周知，shuffle 的性能开销通常很大。

正如背景技术中介绍的，目前的分区连接（PartitionWise Joins）主要是在多个执行节点之间进行任务调度的过程中利用了分区信息，但是在单个执行节点内部进行调度时，则需要依赖 shuffle（重分布）机制才能实现调度，导致执行节点内部的性能开销过大，查询执行效率不佳。

- 5 为此，本公开在多机并行查询架构下提出了一种改进的查询方案，以对节点内调度进行优化，进而改善查询执行效率。

以下结合附图，详细说明本公开各实施例提供的技术方案。

- 10 图1为本公开一示例性实施例提供的一种数据库系统的结构示意图。如图1所示，数据库系统中包括多个执行节点。本实施例中，数据库系统可采用存储计算分离（存算分离）架构，例如，PolarDB等数据库。参考图1，数据库系统中还包括多个存储节点，形成存储层；而本实施例中的执行节点可理解为计算节点，形成计算层。各个执行节点中可运行数据库引擎，以负责计算工作；数据表存储在存储层中，执行节点可按计算需求从存储层中拉取所需的数据。在此，对存算分离架构不做更多说明。

- 15 本实施例中，在数据库系统中采用多机并行查询架构提供查询服务。正如前文中对多机并行查询这一概念的解释，在用户发起查询任务后，可将查询任务拆分为多个子任务，并分配给多个执行节点。另外，本实施例中，查询任务所执行的数据表为分区表。这样，可按照前述的分区连接（PartitionWise Joins）技术中提供的调度机制，来实现在多个执行节点之间分配子任务的过程。本实施例中，对这一调度过程不做限定，图2示出了一种节点间调度的示例性方案的逻辑示意图。参考图2，在该示例性方案中：基于数据库系统的逻辑结构，用户发起查询语句后，可由查询优化器对查询语句进行解析；并继续由数据库调度器根据解析结果为查询语句生成物理执行计划（作为查询任务），并基于物理执行计划拆分成多个分片计划（作为子任务），各个分片计划中的任务逻辑一致，在查询语句所指定的目标分区表中所负责处理的分区不同；之后，调度器可将分片计划分配至多个执行器中，调度的规则可以是负载均衡等，在此不作具体示例。其中，执行器由本实施例中的执行节点提供，这样，从一个查询任务所拆分出的多个子任务被分配到了执行节点上。

在现有的分区连接（PartitionWise Joins）技术中，从单个执行节点的角度来说，在其接收到子任务后，即开始将其作为普通的查询任务进行处理，执行节点对目标分区表的分区信息不再感知，而是依赖 shuffle 机制来进行节点内调度。

- 30 本实施例中，摒弃了依赖 shuffle 机制的节点内调度方式，提出了一种新的节点内调度构思：将分区表的分区信息透传至执行节点中，执行节点内部基于分区信息进行节点内调度，而不再依赖 shuffle 机制。以下对该构思进行展开说明。

- 35 为便于描述，本实施例中将从单个执行节点的视角来对该构思进行说明，应当理解的是，并行处理同一查询任务的各个执行节点均可按照该构思来实施节点内调度。图3为本公开一示例性实施例提供的一种执行节点的内部逻辑结构示意图。参考图3，本实施例中，可对执行节点中的执行器进行改造，具体可以是对执行器中的数据分发组件（例如 exchange 算子）进行改造，以使执行节点按照本实施例中提供的发明构思来进行节点内调度，进而改善查询执行效率。

- 40 参考图3，在查询任务的查询对象为目标分区表的情况下，本实施例中，执行节点可在接收到子任务后，确定子任务在目标分区表中分配到的目标分区。实际应用中，可在子任务中携带目标分区表的标识以及在目标分区表中分配到本执行节点的目标分区的标识，这样，执行节点可通过解析子任务来确定出在目标分区表中分配到的目标分区。当然，还可由图2中的调度器向执行节点推送子任务在目标分区表中分配到的目标分区的标识。本实施例中对如何支持执行节点获知在目标分区表中分配到的目标分区的实现方式不做限定。

- 45 其中，目标分区表是指查询任务所需查询的分区表，同样，从查询任务中划分出的各

个任务也会以目标分区表作为查询对象，但应当理解的是，子任务并无需查询目标分区表的全表，而是只需查询目标分区表中的目标分区即可。另外，目标分区表的数量与查询任务的任务类别有关。例如，若子任务为连接任务（也即 join 任务），则目标分区表的数据通常为两个或两个以上；若查询任务为聚合任务（也即 agg 任务），则目标分区表通常为一个。应当理解的是，查询任务对应于前文提及的物理计划，其本质可理解为计划树，这里的任务类型是基于计划树中的根节点所定义的，例如，若计划树中的根节点为 join 算子，则查询任务即描述为连接任务，又例如，若计划树中的根节点为 agg 算子，则子查询任务即描述为聚合任务。本实施例中，对计划树中的其它节点不做限定，例如，若根节点为 join 算子，其下层连接两个子节点，两个子节点又可分别连接一个或多个子节点，通常，计划树的叶子节点为扫描节点（scan 算子），用于为查询任务拉取数据，本实施例中各个子节点上的算子类型不做限定。另外应当理解的是，子任务与查询任务的任务类型通常一致。子任务的本质也是物理计划，而且，通常子任务与查询任务的计划逻辑也是一致的，只不过子任务可能只承担查询任务所需查询数据中部分数据的查询。

本实施例中，在目标分区表中分配到本执行节点的目标分区的数量，可以是一个或多个，由节点间调度过程所确定，本实施例对此不做干预。

这样，目标分区的分区信息被透传到了执行节点中。在此基础上，本实施例中提出，执行节点将获知的分区信息作为节点内调度的依据，配置出目标分区与本执行节点内为子任务所启动的各工作进程之间的调度关系。其中，这里的调度关系即用于表征哪个工作进程负责哪个目标分区上的查询工作。

本实施例中，不限定执行节点内按照何种策略来基于分区信息配置出该调度关系，这里的基本构思在于：在执行节点内部，以分区为单位进行节点内调度，而不再向传统 shuffle 机制那样以数据记录为单位进行节点内调度。

举例来说，若子任务为连接任务，且指示将分区表 A 和分布表 B 中具有相同 id 的数据记录连接起来，分配给执行节点是 id=0、id=2、id=4、id=6、id=8 和 id=10 这 6 个目标分区，则执行节点在进行节点内调度时，可配置这 6 个目标分区与执行节点内为子任务所启动的 3 个工作进程之间的调度关系，例如，每个工作进程可负责其中的两个目标分区。

在此基础上，执行节点可按照子任务的执行计划执行查询。正如前文提及的，在执行计划中会有扫描节点，用于拉取目标分区中的数据。在已经配置出目标分区与本执行节点内为子任务所启动的各工作进程之间的调度关系的基础上，执行节点可将拉取到的数据调度至与其所属分区相关联的工作进程上。承接上例，若 id=0 的目标分区与工作进程 0 之间存在调度关系，则执行节点在拉取到 id=0 的目标分区内的数据后，会自动将这些数据调度至工作进程 0 上执行查询，而不再需要进行 shuffle。

实际应用中，数据拉取工作通常是按批次进行的，也即是一批一批地拉取数据。本实施例中，执行节点可在每完成一个批次数据的拉取之后，即对将批次数据调度至合适的工作进程下，当然，也可在子任务所需的数据分多个批次全部拉取完之后，在执行数据调度操作，本实施例对数据调度操作的执行时机不做限定。优选地，可按批次执行数据调度操作，以提高查询执行效率。

综上，本实施例中，在多机并行查询架构下提出了一种改进的查询方案，在向多个并行的执行节点分发子任务之后，各个执行节点可分别确定出子任务在目标分区表中分配到的目标分区，这样，可将目标分区表中的分区信息透传至各个执行节点中。在此基础上，在执行节点内部，可配置所分配到的分区与本节点内为所分配到的子任务所启动的各工作进程之间的调度关系，并基于调度关系将拉取到的数据调度至相应的工作进程上，也即是，在执行节点内部，可充分利用获知的分区信息来进行节点内的调度。这使得，执行节点内部的调度过程不再需要依赖 shuffle（重分布）机制，可有效节省节点内调度过程中的性能开销，进而改善多机并行查询场景下的查询执行效率。

图 4 为本公开一示例性实施例提供的一种配置调度关系的实现方案逻辑示意图。在上

述或下述实施例中，可采用多种实现方式来支持执行节点配置出目标分区与本执行节点内为所分配到的子任务所启动的各工作进程之间的调度关系。图4提供了一种优选的实现方式。

5 参考图4，在该优选的实现方式中：执行节点可获取目标分区的标识；将目标分区的标识映射为本执行节点内的分区序号；建立本执行节点内的分区序号与本执行节点内各个工作进程之间的调度关系；其中，本执行节点内的分区序号采用连续编号机制。

10 其中，目标分区的标识是在目标分区表中进行分区定义时所产出的标识，例如，前文示例中的“id=1”即可作为一个目标分区的标识。在该实现方式中提出，为节点内调度过程定义节点内的分区序号。应当理解的是，对于执行节点内的执行器，具体是前文提及的数据分发组件，其在进行节点内调度的过程中认知的将是本实施例中提出的节点内的分区序号，而不再是目标分区原本的标识。

这样，采用统一的分区序号，可使得数据分发组件能够按照通用调度规则来应对不同子任务，而且采用连续编号机制还可避免节点内调度过程中发生数据倾斜等问题，造成某些工作进程被分配到过多的分区。

15 应当理解的是，在该优选的实现方式中，并未对目标分区表中的分区进行破坏，而是将目标分区中原本的分区标识转换为执行节点内部自定义的分区序号，执行节点内部将依据分区序号来实施节点内调度。

20 参考图4，目标分区表包含“split0：分区0”、“split1：分区1”、“split2：分区2”和“split3：分区4”这4个分区，经过节点间调度过程之后，“split0：分区0”和“split2：分区2”这两个目标分区被分配到node0（执行节点0）上，基于此，在node0中，可执行前述的映射操作，将“split0：分区0”映射为分区序号0，将“split2：分区2”映射为分区序号1。在node1中也执行了类型的映射操作。在此基础上，node0中可将分区序号0和分区序号1分配至节点内的并行工作进程，以建立起本执行节点内的分区序号与本执行节点内各个工作进程之间的调度关系。

25 通过图4中的示例可知，在该优选的实现方式中，采用连续编号机制还能带来另一种技术优势，就是可以体现出执行节点内所分配到的目标分区的总数。这一指标可作为节点内调度的参数之一，从而更加合理地建立起本执行节点内的分区序号与本执行节点内各个工作进程之间的调度关系。

30 基于此，进一步，在该优选的实现方案中：执行节点可基于本执行节点内因映射操作所产生的分区序号，确定本执行节点内的分区总数；根据分区总数和本执行节点内工作进程的并行度，为各个工作进程分别确定所负责的分区序号，以产生本执行节点内的分区序号与本执行节点内各个工作进程之间的调度关系。由于采用连续编号机制，因此，映射操作完成后，最大的分区序号与最小的分区序号之间的差值，即为本执行节点内的分区总数。实际应用中，在确定出分区总数之后，可在分区总数与本执行节点内工作进程的并行度之间进行除法计算，来确定出各个工作进程分别可分配到几个目标分区，之后，可依次为各个工作进程选取出所需数量的分区序号。

35 举例来说，若执行节点内产生的分区序号有0、1、2、3、4和5，执行节点内为子任务所启动的工作进程的并行度为3，则，可为每个工作进程分配2个分区序号，之后，可按照依次取号的方案，将分区序号0和1分配给工作进程0，将分区序号2和3分配给工作进程1，将分区序号4和5分配给工作进程2，以建立起本执行节点内的分区序号与本执行节点内各个工作进程之间的调度关系。

40 应当理解的是，在该优选的实现方案中，还可采用其它机制来针对子任务建立本执行节点内的分区序号与本执行节点内各个工作进程之间的调度关系，并不限于上述基于分区总数和并行度的机制，在此不做更多示例。

45 另外，在该优选的实现方案中，若子任务为连接任务，如前文提及的，子任务所指向

的目标分区表的数量将为多个, 这种情况下, 在执行前述映射操作的过程中, 需将多个目标分区表中分配到本执行节点的相同分区的标识映射为本执行节点内的同一分区序号。

举例来说, 子任务执行分区表 A 和分区表 B, 两个分区表中以 id 作为分区建, 在分区表 A 中包含 id=0、id=1、id=2、id=3、id=4 和 id=5 这几个分区, 在分区表 B 中包含 id=0、id=1、id=2、id=3、id=4 和 id=5 这几个分区。若执行节点所分配的目标分区为 d=0、id=1 和 id=2, 则分区表 A 中的 id=0 分区和分区表 B 中的 id=0 分区需映射至执行节点中的同一个分区序号, 例如映射为分区序号 0。同样, 分区表 A 和分区表 B 中其它相同分区的标识, 也应按此原则进行映射。这可保证映射之后的分区序号在目标分区表为多个的情况下, 仍保持分区之间原本的配对关系。

应当理解的是, 上述通过映射操作来支持执行节点配置出目标分区与本执行节点内为子任务所启动的各工作进程之间的调度关系的实现方式是优选的, 本实施例还可采用其它实现方式来支持该调度关系的配置, 而并不限于此, 例如, 可直接按照目标分区的标识进行分配等, 在此不做更多示例。

综上, 本实施例中, 可通过各种实现方式来支持执行节点内部配置出目标分区与本执行节点内为子任务所启动的各工作进程直接的调度关系, 从而以分区为单位, 实现节点内调度, 使得各个工作进程能够分配到互不重叠的目标分区, 从而更加高效、更加准确地并行执行子任务。

图 5 为本公开一示例性实施例提供的一种改进的查询方案的逻辑示意图。参考图 5, 在完成节点内调度之后, 各个工作进程即可按照执行计划对调度到其上的分区数据执行查询操作。

除了对执行节点中的数据交互组件 (exchange 算子) 进行前述改造之外, 本实施例中还提出, 对执行节点中的 hash join 算子也进行改造, 以优化 hash join 算子的性能。在传统的多机并行查询场景中, 执行节点内的 hash join 算子只能按照前文解释的 shuffle 机制来执行 hash join 操作, 性能开销过大, 导致 hash join 算子的执行效率不佳。

参考图 5 提供的改进的查询方案, 本实施例中, 通过对 hash join 算子进行改造, 可使得 hash join 算子中也能充分利用起分区信息, 支持针对不同分区, 分别执行独立的 hash join 操作。这样, 针对单个分区所执行的 hash join 操作将不再需要涉及到其它分区内的数据, 可避免全表 shuffle, 从而大大节省 hash join 算子的性能开销, 优化 hash join 操作效率。

基于此, 若查询子计划为 hash join 任务, 则可通过改造的 hash join 算子充分利用分区信息, 来优化 hash join 操作效率。

本实施例中, 支持单个目标分区由单个工作进程独立负责, 也支持单个目标分区由多个工作进行共同负责。以下对两种情况分别进行说明。

针对单个目标分区由单个工作进程独立负责的情况, 从单个工作进程的视角来看, 其可独立负责一个或多个目标分区。而在单个工作进程独立负责多个目标分区时, 即可通过改造后的 hash join 算子, 对其所负责不同目标分区, 分别执行 hash join 操作。针对不同目标分区所执行的 hash join 操作互不干扰。

举例来说, 以其中的目标进程为例, 若目标进程独立负责分区表 A 和分区表 B 中分区序号为 0 和 1 的两个目标分区, 则目标进程可利用改造后的 hash join 算子的创建端 (即 build 端) 分别为分区表 A 中的这两个目标分区构建独立的哈希表, 例如, 标记为哈希表 0 和哈希表 1。之后, 可利用 hash join 算子中的探测端 (即 probe 端) 基于哈希表 0 去探测分布表 B 中分区序号为 0 的目标分区; 基于哈希表 1 去探测分布表 B 中分区序号为 1 的目标分区。显然, 改造后的 hash join 算子不再需要像传统 shuffle 机制那样进行全表探测。

继续参考图 5, 考虑到 hash join 算子中需要利用到分区信息, 因此, 本实施例中,

还进一步提出了对执行器中的投影算子（如 `projet` 算子）和/或过滤器算子（如 `filter` 算子）等不会打乱分区的算子也进行改造。主要改造为，在数据经过这类算子时，对剩余数据的分区序号进行保留，以使剩余数据被传递至 `hash join` 算子时还依然携带着分区序号，进而保证 `hash join` 算子仍然可按照分区来分别执行 `hash join`。

5 发明人在研究过程中发现，在 `hash jion` 算子之前，还可能存在一些能够打乱分区的特殊算子，例如，聚合 `agg` 算子、排序 `sort` 算子以及其它 `jion` 算子等。由于数据经过这类特殊算子后已经不再符合原本的分区规则，因此，针对这种情况，本实施例中，在数据经过这类特殊算子后，可丢弃相关的分区序号，`hash join` 算子将退回传统的重分布 `shuffle` 机制执行 `hash join` 操作，以保证输出结果的准确性。

10 而针对单个目标分区由多个工作进行共同负责的情况，本实施例中也提出了针对 `hash join` 算子的改造方案。这里将被多个工作进程所共享的目标分区描述为共享分区。发明人在研究过程中发现，在数据库系统中，通常来说，`hash join` 的 `build` 过程是一个阻塞过程，也就是 `probe` 一定会发生在 `build` 完成之后，而 `probe` 过程中，并不会对哈希表 `hash table` 有更新和删除操作，所以其实是个完全的只读过程。基于此，`probe` 过程并不涉及多进程并发控制问题，为此，本实施例中将改造的重点放在了 `build` 过程。也即是提供针对 `hash join` 算子的改造方案，以解决这种情况下的多进程并发控制问题。

传统的多进程并发控制问题的解决方案通常是基于锁机制等来实现，在此不做现有方案的展开详述。本实施例中通过对 `hash join` 算子改造，实现了一种新的解决方案：

20 利用目标进程中的 `hash join` 算子获取共享分区所对应的批处理总次数以及与负责共享分区的工作进程总数；

利用 `hash join` 算子按照预置分配规则，根据批处理总次数和工作进程总数，确定目标进程自身所分配到的数据批次；

25 利用 `hash join` 算子判断当前数据批次是否为目标进程所分配到的数据批次，若是，则启动 `hash join` 算子的创建 `build` 端为当前数据批次创建用于支持 `hash join` 计算的哈希表，以使共同负责共享分区的各个工作进程协同创建共享分区对应的哈希表。

30 在该新的解决方案中，在 `hash jion` 算子中定义了基于批处理总次数和工作进程总数的预置分配规则，这样，共同负责共享分区的各个 `hash join` 算子可在获取到共享分区所对应的批处理总次数以及与负责共享分区的工作进程总数后，按照预置分配规则确定出自身所分配到的数据批次。正如前文提及的，在子任务的执行过程中，是分批次拉取数据的，而共享分区对应的批处理总次数则可在执行节点内部的调度器或者也可以在数据分发组件中来预先产生，并通知给共同负责共享分区的各个 `hash join` 算子；同样，负责共享分区的工作进程总数也可由数据分发组件通知给共同负责共享分区的各个 `hash join` 算子。这样，共同负责共享分区的各个 `hash join` 算子均可获取到这两个参数，进而均可确定出自身所应分配到的数据批次。

35 在此基础上，从单个 `hash join` 算子的角度来说，其可判断当前数据批次是否为自身所应所分配到的数据批次，若是，则启动 `hash join` 算子的创建 `build` 端为当前数据批次创建用于支持 `hash join` 计算的哈希表，这不仅可保证共同负责共享分区的各个 `hash join` 算子各自所构建出的哈希表之间不会发生重叠，而且能够保证不遗漏共享分区中的任何数据，从而有效解决多进程并发控制问题。

40 其中，在该新的解决方案中，不限定预置分配规则中的具体机制，例如，可采用负载均衡机制，使各个 `hash join` 算子分配到基本等量的数据批次，当然，也可采用抢占机制，各个 `hash join` 算子对当前数据批次进行抢占，抢占到的，即负责为当前数据批次构建哈希表。在此不做更多机制的示例，本实施例并不限于此。之后，可将共同负责共享分区的各个 `hash join` 算子各自所创建出的哈希表进行合并，从而产生共享分区对应哈希表。

45 综上，本实施例中，提出了对执行节点中除数据分发组件（`exchange` 算子）之外的更

多算子的改造方案，以在利用分区信息完成节点内调度后，能够继续充分利用分区信息，优化执行计划中多种算子的执行性能，进而改善针对子任务的执行效率。

5 发明人在研究过程中还发现，传统的多机并行查询架构中，除了在前述的节点内调度过程中以及 hash join 算子中需要依赖 shuffle 机制之外，在节点之间进行数据交互的过程中也需要依赖 shuffle 机制。也即是，执行节点内需要通过 shuffle 机制确定出哪些数据需要传输至其它节点中，这也造成了执行节点中大量的性能开销，也会拉低查询执行效率。

10 对此，本实施例中还进一步针对节点间的数据交换过程提出了优化方案。节点间的数据交换工作通常是由执行节点中的数据分发组件（如 exchange 算子）来负责的，因此，这部分优化方案可通过进一步改造执行节点中的数据分发组件（如 exchange 算子）来实现。

15 在该优化方案中提出，可由如图 2 中所示出的用于分发子任务的数据库调度器，为目标分区表构建一份分区裁剪名单，并下发给各个子任务所在的执行节点。其中，分区裁剪名单中记录了目标分区表中被裁剪后不会影响输出结果的分区的名单。数据库调度器可在分发子任务之前完成分区裁剪名单的构建及下发，以保证各个子任务所在的执行节点可及时获取到该份分区裁剪名单。

20 实际应用中，数据库调度器可根据对查询任务的查询语句所对应的解析结果发现目标分区表中被指定为需要裁剪的分区，通过目标分区表对应的属性描述信息发现目标分区表中数据为空的分区，并将这两类分区的标识记录在分区裁剪名单中。当然，分区裁剪名单中还可记录其它被裁剪后不会影响输出结果的分区的名单，而不限于此处示例出的这两类，在此不做更多示例。

25 在此基础上，在该优化方案中：执行节点可获取到针对目标分区表所构建的分区裁剪名单；若本执行节点需要与其它节点进行数据交换，则按照分区裁剪名单对目标分区表中分配到本执行节点的目标分区进行裁剪后，基于剩余分区为其它节点组建交换数据；在接收到数据交换请求时，将基于剩余分区所组建的交换数据作为响应数据。也即是，将分配到本执行节点的目标分区中位于分区裁剪名单中的那部分分区裁剪掉，而本执行节点所分配到的目标分区中的剩余分区下，可组建交换数据，被裁剪掉的目标分区将不再需要组件交换数据。

30 正如前文提及的，若子任务为连接任务，则目标分区表的数量通常为多个，这种情况下，执行节点在应用分区裁剪名单时可在多个目标分区表中进行同步裁剪。换句话说，若目标分区表的数量为多个且目标分区中存在位于分区裁剪名单内的待剪分区，则执行节点可将多个目标分区表中分配到本执行节点的待剪分区均裁剪掉。

35 举例来说，若子任务指向分区表 A 和分区 B 中的 id=0 和 id=1 分区，由于分区表 A 中的 id=0 分区中的数据为空，因此，id=0 分区包含在分区裁剪名单中。基于此，执行节点来拉取到分区表 A 和分区表 B 的 id=0 和 id=1 分区后，可将分区表 A 的 id=0 分区裁剪掉；同时，尽管分区 B 的 id=0 分区中的数据并不为空，但执行节点还是会将分区表 B 的 id=0 分区裁剪掉。

40 综上，在该优选方案中，对执行节点内部“组建交换数据的逻辑环节”不做改动，但在启动“组件交换数据的逻辑环节”之前，执行节点会分区裁剪名单对目标分区表中分配到本执行节点的目标分区进行裁剪，而裁剪之后的剩余分区则将作为“组件交换数据的逻辑环节”的操作范围。这可有效减少执行节点内部的“组建交换数据的逻辑环节”的操作范围，因此，可有效减少节点之间进行数据交互的过程中的数据传输量，从而可进一步提高查询执行效率。

45 图 6 为本公开另一示例性实施例提供的一种查询方法的流程示意图，该方法适应于并行处理查询任务的多个执行节点中的任意一个执行节点，查询任务的查询对象为目标分区

表，该方法可由该执行节点中的执行器实施。参考图 6，该方法可包括：

步骤 600、在接收到从查询任务中拆分出的子任务后，确定所述子任务在所述目标分区表中分配到的目标分区；

5 步骤 601、配置目标分区与本执行节点内为所述子任务所启动的各工作进程之间的调度关系；

步骤 602、在拉取到所述目标分区对应的数据后，基于所述调度关系，将拉取到的数据调度至与其所属分区相关联的工作进程上，以供工作进程执行查询。

在一可选实施例中，配置目标分区与本执行节点内为子任务所启动的各工作进程之间的调度关系，包括：

10 获取所述目标分区的标识；

将所述目标分区的标识映射为本执行节点内的分区序号；

建立本执行节点内的分区序号与本执行节点内各个工作进程之间的调度关系；

其中，本执行节点内的分区序号采用连续编号机制。

15 在一可选实施例中，在所述子任务为连接查询任务的情况下，所述目标分区表的数量为多个，将所述目标分区的标识映射为本执行节点内的分区序号，包括：

将所述多个目标分区表中分配到本执行节点的相同分区的标识映射为本执行节点内的同一分区序号。

在一可选实施例中，建立本执行节点内的分区序号与本执行节点内各个工作进程之间的调度关系，包括：

20 基于本执行节点内因所述映射操作所产生的分区序号，确定本执行节点内的分区总数；

根据所述分区总数和本执行节点内工作进程的并行度，为各个工作进程分别确定所负责的分区序号，以产生本执行节点内的分区序号与本执行节点内各个工作进程之间的调度关系。

25 在一可选实施例中，若所述子任务为 hash join 任务且目标进程独立负责多个目标分区，所述方法还包括：

若目标进程的执行计划中在 hash join 算子之前存在投影算子和/或过滤器算子，则在调度至所述目标进程上的数据经过所述投影算子或所述过滤器算子后，对剩余数据的分区序号进行保留；

30 在所述 hash join 算子中，针对所独立负责的不同目标分区，分别执行 hash join 操作；

其中，所述目标进程为本执行节点内的任一工作进程。

在一可选实施例中，所述方法还包括：

35 若所述目标进程的执行计划中在 hash join 算子之前存在能够打乱分区的特殊算子，则在数据经过所述特殊算子后，丢弃相关的分区序号，以使所述 hash join 算子按照重分布 shuffle 机制执行 hash join 操作。

在一可选实施例中，若所述目标进程与其它工作进程共同负责一共享分区，则所述方法还包括：

利用目标进程中的 hash join 算子获取共享分区所对应的批处理总次数以及与负责所述共享分区的工作进程总数；

40 利用所述 hash join 算子按照预置分配规则，根据所述批处理总次数和所述工作进程总数，确定所述目标进程自身所分配到的数据批次；

利用所述 hash join 算子判断当前数据批次是否为所述目标进程所分配到的数据批次，若是，则启动所述 hash join 算子的创建 build 端为所述当前数据批次创建用于支持 hash

join 计算的哈希表, 以使共同负责所述共享分区的各个工作进程协同创建所述共享分区对应的哈希表。

在一可选实施例中, 所述预置分配规则包括按照负载均衡机制进行分配。

在一可选实施例中, 所述方法还包括:

- 5 获取针对所述目标分区表所构建的分区裁剪名单, 所述分区裁剪名单中包括被裁剪后不会影响查询结果的分区的名单;

若本执行节点需要与其它节点进行数据交换, 则按照所述分区裁剪名单对所述目标分区表中分配到本执行节点的目标分区进行裁剪后, 基于剩余分区为所述其它节点组建交换数据;

- 10 在接收到数据交换请求时, 将基于剩余分区所组建的交换数据作为响应数据。

在一可选实施例中, 按照所述分区裁剪名单对所述目标分区表中分配到本执行节点的目标分区进行裁剪, 包括:

若目标分区表的数量为多个且目标分区中存在位于分区裁剪名单内的待剪分区, 则将多个目标分区表中分配到本执行节点的待剪分区均裁剪掉。

- 15 在一可选实施例中, 所述分区裁剪名单由分发所述子任务的数据库调度器生成, 所述数据库调度器用于根据对所述查询任务的查询语句所对应的解析结果发现所述目标分区表中被指定为需要裁剪的分区, 通过所述目标分区表对应的属性描述信息发现所述目标分区表中数据为空的分区, 以构建所述分区裁剪名单。

- 20 需要说明的是, 在上述实施例及附图中的描述的一些流程中, 包含了按照特定顺序出现的多个操作, 但是应该清楚了解, 这些操作可以不按照其在本文中出现的顺序来执行或并行执行, 操作的序号如 601、602 等, 仅仅是用于区分开各个不同的操作, 序号本身不代表任何的执行顺序。

- 25 图 7 为本公开又一示例性实施例提供的一种执行节点的结构示意图。如图 7 所示, 该执行节点是并行处理查询任务的多个执行节点中的任意一个, 查询任务的查询对象为目标分区表, 该执行节点可包括: 存储器 70、处理器 71 以及通信组件 72。

处理器 71, 与存储器 70 和通信组件 72 耦合, 用于执行存储器 70 中的计算机程序, 以用于:

在接收到从查询任务中拆分出的子任务后, 确定所述子任务在所述目标分区表中分配到的目标分区;

- 30 配置目标分区与本执行节点内为子任务所启动的各工作进程之间的调度关系;

在拉取到所述目标分区对应的数据后, 基于所述调度关系, 将拉取到的数据调度至与其所属分区相关联的工作进程上, 以供工作进程执行查询。

在一可选实施例中, 处理器 71 在配置目标分区与本执行节点内为子任务所启动的各工作进程之间的调度关系时, 可具体用于:

- 35 获取所述目标分区的标识;

将所述目标分区的标识映射为本执行节点内的分区序号;

建立本执行节点内的分区序号与本执行节点内各个工作进程之间的调度关系;

其中, 本执行节点内的分区序号采用连续编号机制。

- 40 在一可选实施例中, 在所述子任务为连接查询任务的情况下, 所述目标分区表的数量为多个, 处理器 71 在将所述目标分区的标识映射为本执行节点内的分区序号时, 可具体用于:

将所述多个目标分区表中分配到本执行节点的相同分区的标识映射为本执行节点内的同一分区序号。

在一可选实施例中，建立本执行节点内的分区序号与本执行节点内各个工作进程之间的调度关系时，可具体用于：

基于本执行节点内因所述映射操作所产生的分区序号，确定本执行节点内的分区总数；

- 5 根据所述分区总数和本执行节点内工作进程的并行度，为各个工作进程分别确定所负责的分区序号，以产生本执行节点内的分区序号与本执行节点内各个工作进程之间的调度关系。

在一可选实施例中，若所述子任务为 hash join 任务且目标进程独立负责多个目标分区，处理器 71 还可用于：

- 10 若目标进程的执行计划中在 hash join 算子之前存在投影算子和/或过滤器算子，则在调度至所述目标进程上的数据经过所述投影算子或所述过滤器算子后，对剩余数据的分区序号进行保留；

在所述 hash join 算子中，针对所独立负责的不同目标分区，分别执行 hash join 操作；

其中，所述目标进程为本执行节点内的任一工作进程。

- 15 在一可选实施例中，处理器 71 还可用于：

若所述目标进程的执行计划中在 hash join 算子之前存在能够打乱分区的特殊算子，则在数据经过所述特殊算子后，丢弃相关的分区序号，以使所述 hash join 算子按照重分布 shuffle 机制执行 hash join 操作。

- 20 在一可选实施例中，若所述目标进程与其它工作进程共同负责一共享分区，则处理器 71 还可用于：

利用目标进程中的 hash join 算子获取共享分区所对应的批处理总次数以及与负责所述共享分区的工作进程总数；

利用所述 hash join 算子按照预置分配规则，根据所述批处理总次数和所述工作进程总数，确定所述目标进程自身所分配到的数据批次；

- 25 利用所述 hash join 算子判断当前数据批次是否为所述目标进程所分配到的数据批次，若是，则启动所述 hash join 算子的创建 build 端为所述当前数据批次创建用于支持 hash join 计算的哈希表，以使共同负责所述共享分区的各个工作进程协同创建所述共享分区对应的哈希表。

在一可选实施例中，所述预置分配规则包括按照负载均衡机制进行分配。

- 30 在一可选实施例中，处理器 71 还可用于：

获取针对所述目标分区表所构建的分区裁剪名单，所述分区裁剪名单中包括被裁剪后不会影响查询结果的分区的名单；

- 35 若本执行节点需要与其它节点进行数据交换，则按照所述分区裁剪名单对所述目标分区表中分配到本执行节点的目标分区进行裁剪后，基于剩余分区为所述其它节点组建交换数据；

在接收到数据交换请求时，将基于剩余分区所组建的交换数据作为响应数据。

在一可选实施例中，处理器 71 在按照所述分区裁剪名单对所述目标分区表中分配到本执行节点的目标分区进行裁剪时，可具体用于：

- 40 若目标分区表的数量为多个且目标分区中存在位于分区裁剪名单内的待剪分区，则将多个目标分区表中分配到本执行节点的待剪分区均裁剪掉。

在一可选实施例中，所述分区裁剪名单由分发所述子任务的数据库调度器生成，所述数据库调度器用于根据对所述查询任务的查询语句所对应的解析结果发现所述目标分区表中被指定为需要裁剪的分区，通过所述目标分区表对应的属性描述信息发现所述目标分

区表中数据为空的分区，以构建所述分区裁剪名单。

进一步，如图 7 所示，该执行节点还包括：电源组件 73 等其它组件。图 7 中仅示意性给出部分组件，并不意味着执行节点只包括图 7 所示组件。

值得说明的是，上述关于执行节点各实施例中的技术细节，可参考前述的系统实施例中关于执行节点的相关描述，为节省篇幅，在此不再赘述，但这不应造成本公开保护范围的损失。

相应地，本公开实施例还提供一种存储有计算机程序的计算机可读存储介质，计算机程序被执行时能够实现上述方法实施例中执行的各步骤。

本公开实施例还提供一种计算机程序，当计算机程序在计算机中执行时，令计算机执行上述方法实施例中执行的各步骤。

上述图 7 中的存储器，用于存储计算机程序，并可被配置为存储其它各种数据以支持在计算平台上的操作。这些数据的示例包括用于在计算平台上操作的任何应用程序或方法的指令，联系人数据，电话簿数据，消息，图片，视频等。存储器可以由任何类型的易失性或非易失性存储设备或者它们的组合实现，如静态随机存取存储器（SRAM），电可擦除可编程只读存储器（EEPROM），可擦除可编程只读存储器（EPROM），可编程只读存储器（PROM），只读存储器（ROM），磁存储器，快闪存储器，磁盘或光盘。

上述图 7 中的通信组件，被配置为便于通信组件所在设备和其他设备之间有线或无线方式的通信。通信组件所在设备可以接入基于通信标准的无线网络，如 WiFi，2G、3G、4G/LTE、5G 等移动通信网络，或它们的组合。在一个示例性实施例中，通信组件经由广播信道接收来自外部广播管理系统的广播信号或广播相关信息。在一个示例性实施例中，所述通信组件还包括近场通信（NFC）模块，以促进短程通信。例如，在 NFC 模块可基于射频识别（RFID）技术，红外数据协会（IrDA）技术，超宽带（UWB）技术，蓝牙（BT）技术和其他技术来实现。

上述图 7 中的电源组件，为电源组件所在设备的各种组件提供电力。电源组件可以包括电源管理系统，一个或多个电源，及其他与为电源组件所在设备生成、管理和分配电力相关联的组件。

本领域内的技术人员应明白，本公开的实施例可提供为方法、系统、或计算机程序产品。因此，本公开可采用完全硬件实施例、完全软件实施例、或结合软件和硬件方面的实施例的形式。而且，本公开可采用在一个或多个其中包含有计算机可用程序代码的计算机可用存储介质（包括但不限于磁盘存储器、CD-ROM、光学存储器等）上实施的计算机程序产品的形式。

本公开是参照根据本公开实施例的方法、设备（系统）、和计算机程序产品的流程图和 / 或方框图来描述的。应理解可由计算机程序指令实现流程图和 / 或方框图中的每一流程和 / 或方框、以及流程图和 / 或方框图中的流程和 / 或方框的结合。可提供这些计算机程序指令到通用计算机、专用计算机、嵌入式处理机或其他可编程数据处理设备的处理器以产生一个机器，使得通过计算机或其他可编程数据处理设备的处理器执行的指令产生用于实现在流程图一个流程或多个流程和 / 或方框图一个方框或多个方框中指定的功能的装置。

这些计算机程序指令也可存储在能引导计算机或其他可编程数据处理设备以特定方式工作的计算机可读存储器中，使得存储在该计算机可读存储器中的指令产生包括指令装置的制造品，该指令装置实现在流程图一个流程或多个流程和 / 或方框图一个方框或多个方框中指定的功能。

这些计算机程序指令也可装载到计算机或其他可编程数据处理设备上，使得在计算机或其他可编程设备上执行一系列操作步骤以产生计算机实现的处理，从而在计算机或其他可编程设备上执行的指令提供用于实现在流程图一个流程或多个流程和 / 或方框图一个

方框或多个方框中指定的功能的步骤。

5 还需要说明的是，术语“包括”、“包含”或者其任何其他变体意在涵盖非排他性的包含，从而使得包括一系列要素的过程、方法、商品或者设备不仅包括那些要素，而且还包括没有明确列出的其他要素，或者是还包括为这种过程、方法、商品或者设备所固有的要素。在没有更多限制的情况下，由语句“包括一个……”限定的要素，并不排除在包括所述要素的过程、方法、商品或者设备中还存在另外的相同要素。

10 需要说明的是，本公开所涉及的用户信息（包括但不限于用户设备信息、用户个人信息等）和数据（包括但不限于用于分析的数据、存储的数据、展示的数据等），均为经用户授权或者经过各方充分授权的信息和数据，并且相关数据的收集、使用和处理需要遵守相关国家和地区的相关法律法规和标准，并提供有相应的操作入口，供用户选择授权或者拒绝。

以上所述仅为本公开的实施例而已，并不用于限制本公开。对于本领域技术人员来说，本公开可以有各种更改和变化。凡在本公开的精神和原理之内所作的任何修改、等同替换、改进等，均应包含在本公开的保护范围之内。

权 利 要 求 书

1、一种数据查询方法，其特征在于，适应于并行处理查询任务的多个执行节点中的任一执行节点，所述查询任务的查询对象为目标分区表，所述方法包括：

5 在接收到从所述查询任务中拆分出的子任务后，确定所述子任务在所述目标分区表中所分配到的目标分区；

配置所述目标分区与所述子任务所对应的各工作进程之间的调度关系；

在拉取到所述目标分区对应的数据后，基于所述调度关系，将拉取到的数据调度至与其所属分区相关联的工作进程上，以供工作进程执行查询。

10 2、根据权利要求1所述的方法，其特征在于，配置所述目标分区与所述子任务所对应的各工作进程之间的调度关系，包括：

获取所述目标分区的标识；

将所述目标分区的标识映射为本执行节点内的分区序号；

建立本执行节点内的分区序号与本执行节点内各个工作进程之间的调度关系；

其中，本执行节点内的分区序号采用连续编号机制。

15 3、根据权利要求2所述的方法，其特征在于，在所述子任务为连接查询任务的情况下，所述目标分区表的数量为多个，将所述目标分区的标识映射为本执行节点内的分区序号，包括：

将所述多个目标分区表中分配到本执行节点的相同分区的标识映射为本执行节点内的同一分区序号。

20 4、根据权利要求2所述的方法，其特征在于，建立本执行节点内的分区序号与本执行节点内各个工作进程之间的调度关系，包括：

基于本执行节点内因所述映射操作所产生的分区序号，确定本执行节点内的分区总数；

25 根据所述分区总数和本执行节点内工作进程的并行度，为各个工作进程分别确定所负责的分区的序号，以产生本执行节点内的分区序号与本执行节点内各个工作进程之间的调度关系。

5、根据权利要求1-4任一项所述的方法，其特征在于，若所述子任务为 hash join 任务且目标进程独立负责多个目标分区，所述方法还包括：

30 若目标进程的执行计划中在 hash join 算子之前存在投影算子和/或过滤器算子，则在调度至所述目标进程上的数据经过所述投影算子或所述过滤器算子后，对剩余数据的分区序号进行保留；

在所述 hash join 算子中，针对所独立负责的不同目标分区，分别执行 hash join 操作；

其中，所述目标进程为本执行节点内的任一工作进程。

6、根据权利要求5所述的方法，其特征在于，还包括：

35 若所述目标进程的执行计划中在 hash join 算子之前存在能够打乱分区的特殊算子，则在数据经过所述特殊算子后，丢弃相关的分区序号，以使所述 hash join 算子按照重分布 shuffle 机制执行 hash join 操作。

7、根据权利要求5所述的方法，其特征在于，若所述目标进程与其它工作进程共同负责一共享分区，则所述方法还包括：

40 利用目标进程中的 hash join 算子获取共享分区所对应的批处理总次数以及与负责所述共享分区的工作进程总数；

利用所述 hash join 算子按照预置分配规则，根据所述批处理总次数和所述工作进程

总数，确定所述目标进程自身所分配到的数据批次；

利用所述 hash join 算子判断当前数据批次是否为所述目标进程所分配到的数据批次，若是，则启动所述 hash join 算子的创建 build 端为所述当前数据批次创建用于支持 hash join 计算的哈希表，以使共同负责所述共享分区的各个工作进程协同创建所述共享分区对应的哈希表。

5

8、根据权利要求 7 所述的方法，其特征在于，所述预置分配规则包括按照负载均衡机制进行分配。

9、根据权利要求 1-8 任一项所述的方法，其特征在于，还包括：

10 获取针对所述目标分区表所构建的分区裁剪名单，所述分区裁剪名单中包括被裁剪后不会影响查询结果的分区的名单；

若本执行节点需要与其它节点进行数据交换，则按照所述分区裁剪名单对所述子任务在所述目标分区表中分配到目标分区进行裁剪后，基于剩余分区为所述其它节点组建交换数据；

在接收到数据交换请求时，将基于剩余分区所组建的交换数据作为响应数据。

15

10、根据权利要求 9 所述的方法，其特征在于，按照所述分区裁剪名单对所述子任务在所述目标分区表中分配到的目标分区进行裁剪，包括：

若目标分区表的数量为多个且目标分区中存在位于分区裁剪名单内的待剪分区，则将多个目标分区表中分配到所述子任务的待剪分区均裁剪掉。

20

11、根据权利要求 9 所述的方法，其特征在于，所述分区裁剪名单由分发所述子任务的数据库调度器生成，所述数据库调度器用于根据对所述查询任务的查询语句所对应的解析结果发现所述目标分区表中被指定为需要裁剪的分区，通过所述目标分区表对应的属性描述信息发现所述目标分区表中数据为空的分区，以构建所述分区裁剪名单。

12、一种执行节点，其特征在于，包括存储器、处理器和通信组件；

所述存储器用于存储一条或多条计算机指令；

25

所述处理器与所述存储器和所述通信组件耦合，用于执行所述一条或多条计算机指令，以用于执行权利要求 1-11 任一项所述的查询方法。

13、一种数据库系统，其特征在于，包括并行处理查询任务的多个执行节点，所述多个执行节点分别接收从所述查询任务中所拆分出的子任务，单个执行节点内为自身所接收到的子任务启动有多个并行的工作进程，单个执行节点用于执行权利要求 1-11 任一项所述的查询方法。

30

14、一种存储计算机指令的计算机可读存储介质，其特征在于，当所述计算机指令被一个或多个处理器执行时，致使所述一个或多个处理器执行权利要求 1-11 任一项所述的查询方法。

15、一种计算机程序，当所述计算机程序在计算机中执行时，令计算机执行权利要求 1-11 任一项所述的查询方法。

35