arXiv:2304.08532v1 [cs.DB] 17 Apr 2023

基于磁盘的列存储中的混合物化

Evgeniy Klyuchikov、Elena Mikhailova、George Chernishev 圣彼得堡国立大学 俄罗斯圣彼得堡 evgeniy.klyuchikov@gmail.com、egmichailova@mail.ru、chernishev@gmail.com

摘要——在面向列的查询处理中，物化策略决定了轻量级位置（行标识符）何时转换为元组。这是列存储架构的重要组成部分，因为它定义了所支持的查询计划类型，并因此影响整体系统性能。

本文继续研究分布式基于磁盘的列存储的物化策略。我们首先展示现有方法会对最终系统性能造成根本性限制的情况。随后，为解决这些问题，我们提出一种新的混合物化模型。混合物化的主要特点是能够同时处理位置和值。这样，查询引擎便能灵活地结合所有现有策略的优势，并支持一类新的查询计划。此外，混合物化使查询引擎能够灵活定制各个属性的物化策略。

我们阐述了如何在列式系统中实现混合物化。作为示例，我们采用 PosDB——一种分布式、基于磁盘的列存储。我们介绍所需的数据结构、混合算子的内部机制，并描述此类算子的代数。基于这一实现，我们在多个基于 TPC-H 查询的场景中评估了晚期、超晚期和混合物化策略的性能。实验表明，混合物化的速度几乎是其他策略的两倍，同时提供了更灵活的查询模型。

I. 引言

列存储以其在分析型工作负载上的出色性能而闻名 [1], [2]。其成功的主要因素之一，是能够处理单独的列，因此只有在绝对必要时才从磁盘读取数据。此外，这种处理方式还能让查询执行器受益于压缩和向量化计算 [3]–[5]。

处理单独的属性值时，仍有必要引用这些值所属的表元组子集。为此，列存储通常使用

行号、命名位置或行标识符 [6]。使用位置对查询引擎来说很方便，但最终必须将其转换为属性值并拼接在一起，形成结果元组。

然而，大多数现代列存储与先驱者所设想的愿景大相径庭 [7]–[9]。00 年代的研究原型支持流畅的数据位置处理，使查询引擎能够在查询执行期间显式使用位置。这带来了许多优化和技术，为查询处理提供了各种好处。遗憾的是，大多数主流系统并未遵循最初的理念，而是转而采用更简单的做法（因此得名朴素列存储 [6]）。因此，我们将“支持位置”的列存储与其他列存储区分开来：前者能够从其引擎内部的显式位置操作中获益。

在支持位置的列存储中，从位置转换为值的过程称为物化。因此，物化策略决定了在查询计划中的何时以及如何物化值。物化策略是支持位置的列存储架构的重要组成部分，并且会显著影响系统性能。

在我们先前的研究 [2] 中，我们全面探讨了各种系统用于物化数据的基本原理。我们比较了现有的提前物化和晚期物化策略，并提出了一种新的超晚期物化策略，以更快地处理分析型工作负载。

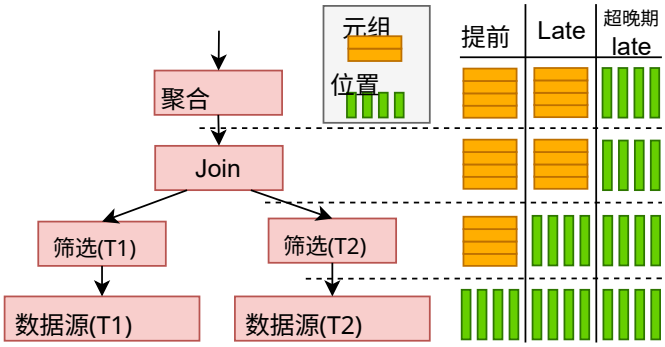


图1：带有物化点的选择-投影-连接查询

为了整体了解各种物化策略，请看图1。图中展示了列存储中的一个选择-投影-连接查询计划，同时说明数据表示在不同查询阶段的演变。提前、晚期和超晚期策略在查询计划底部使用位置，并在某个物化点 将位置转换为属性值。之后，查询执行将继续处理元组，与传统行存储类似。

提前物化会在筛选条件之前构造元组，晚期物化会在连接之前构造元组，而超晚期物化会在聚合之前构造元组。我们的研究—

表 I：物化策略的各个方面

物化	快速谓词	谓词期间重新读取	连接前预读	连接期间重新读取	乱序探测
提前	No	No	Yes	No	No
Late	Yes	Yes	Yes	No	No
超晚期	Yes	Yes	No	Yes	Yes
混合	?	?	?	?	?

表明，超晚期物化可以受益于在过滤连接后读取属性数据，从而在星型模式基准测试等工作负载上实现超过两倍的加速 [2] [10]。

然而，我们认为，这三种策略都受到其设计的限制。它们在任何给定时刻要么使用位置，要么使用元组（但不会同时使用两者），因此无法灵活地物化单个属性。因此，本文提出了一种新的混合物化策略，克服了其他策略的缺点。

混合物化的理念是让位置和值同时通过算子。这样，我们就能结合其他所有策略的优势。与超晚期物化类似，我们可以在聚合之前读取查询的某些属性。然而，如果读取某些属性的开销很大 [11], ，我们就可以灵活地更早缓存这些特定属性，并将它们与位置一起传递。

本文将探讨基于磁盘、支持位置操作的列存储系统中的混合物化。我们将分析提前、晚期和超晚期策略，展示这些策略因一步物化而存在根本局限的情形。我们将阐述如何在 PosDB[1] 这样的基于磁盘的列存储系统中实现混合物化。我们将介绍混合数据结构、混合算子的内部结构以及整体混合查询模型。最后，我们将展示采用混合物化的查询计划，并通过实验评估其性能与晚期物化和超晚期物化相比的表现，以验证我们的发现。

本文的主要贡献如下：
1）对提前、晚期和超晚期物化策略进行定性分析。我们重点指出可通过混合位置-值处理加以改进的情形。
2）提出一种混合查询模型。我们介绍混合数据结构、混合算子内部结构以及混合算子代数，并说明如何将混合策略安全地引入采用提前、晚期或超晚期物化的系统。
3）使用经修改的 TPC-H 基准测试查询 [12]，对混合物化与晚期物化和超晚期物化进行实验评估。

本文组织结构如下。第二节概述现有物化策略的基础，说明混合物化在哪些方面能够发挥作用，并讨论相关工作。第三节根据我们在 PosDB 中的经验，介绍如何在基于磁盘的列存储中实现混合物化。第四节将更详细地讨论混合物化在本地和分布式场景中的优势。随后，第五节对各种物化策略进行实验评估。最后，第六节和第七节介绍我们的未来计划，并对论文作出总结。

II. 背景与相关工作

A. 背景

为更详细地说明混合物化的动机并概述相关研究，我们首先介绍列存储如何物化数据。为此，我们简要回顾现有物化策略的主要方面，并说明一步物化在哪些方面存在局限。

我们使用选择-投影-连接（SPJ）类查询进行分析和实验。这类查询足够简单，同时又包含性能显著依赖于所选物化策略的算子。具体而言，我们重点关注连接和筛选条件（谓词）。

如图1所示，提前、晚期和超晚期物化策略的区别在于将位置转换为元组的时机。因此，这些策略在算子如何处理内存中的数据，以及如何从磁盘加载这些数据方面也有所不同。

表 I 总结了各种物化策略在数据处理方面的五个重要特性。

早期物化使用列式存储仅读取必要的属性，然后切换到元组处理。采用这种策略时，列存储与常规行存储类似。

晚期物化在执行计划中进一步使用位置，通过按需加载同质列式数据进行处理，从而加快谓词运算。这能显著提升性能，但某些属性可能会被多次读取。

超晚期物化将位置进一步向后推进，以加速连接，方式与谓词类似。通过对位置进行操作，连接可以使用更小的哈希表，并仅在必要时加载属性。对于只保留少量记录的“筛选”连接工作流，后一种方式尤其高效 [2]。

超晚期物化有两个已知缺点：连接过程中容易发生重复读取，而且还可能被无序的位置列表迫使读取属性数据。后一种情况需要耗费大量资源进行随机磁盘访问，称为乱序探测问题 [13]。

混合物化可能是对现有所有物化策略的一种折衷。其核心思想是在查询计划的某些部分同时传递位置和值。因此，查询引擎既能利用位置，又能提前物化那些重新访问成本高昂的属性。这样，混合物化便使

[1]https://pos-db.com/

查询引擎能够针对每个属性或表，分别自定义五个方面中的每一项。因此，它能为给定查询提供尽可能最佳的策略。

B. 相关工作

本节将简要讨论有关物化策略或混合位置-值处理的现有论文。

“物化策略”这一术语最初由 C-Store [1]，（一种基于磁盘的列存储）的作者提出，用于讨论如何将磁盘上的列数据转换为内存中的结果元组。他们介绍了在查询处理过程中物化部分数据的概念，并解释了相关成本。不过，他们主要关注提前物化和晚期物化及其性能。

有趣的是，作者介绍了几个从磁盘读取属性数据的算子，其中一个实际上同时使用位置和值 [1]。这意味着 C-Store 实际上支持混合物化，但仅限于一个特定算子。他们还提到，在连接算子 [6], 中，如果左子树使用位置而右子树使用元组，混合方法可能会很有用。然而，他们没有继续研究混合物化，也没有讨论在本地和分布式场景下如何将其用于整个查询。

MonetDB 系统的作者也探索了各种物化策略 [7]，[8]。其算子使用二元关联表，这是一种由一列或两列组成的特殊表。这些列包含位置或值，查询引擎可以按需使用它们物化数据。之所以采用这种特定表示方式，是因为 MonetDB 被设计为内存系统，其所有算子都针对内存中的快速处理进行了优化。它也只支持本地查询处理。因此，MonetDB 的理念只能部分用于分布式基于磁盘的列存储中的混合物化。

另一系列关于混合位置-值处理的研究源自 PAX [14]。PAX 旨在通过改变数据在磁盘和内存中的数据页内的组织方式，提升经典行存储引擎的速度。其核心思想是将数据页中的元组数组替换为一系列微页，每个属性对应一个微页。因此，每个属性的数据都单独存储，就像在列存储中一样；所有微页都包含同一范围内的表元组数据，也就是说，它们具有一些虚拟行标识符。这种混合存储可供查询引擎使用，在一定程度上同时通过位置和值处理数据。

许多系统都是基于 PAX 模型构建的。MonetDB 的下一代系统 Vectorwise 使用 PAX 来支持表的总大小超过随机存取存储器容量限制的工作负载 [4]。不过，混合格式仅用于磁盘，而 Vectorwise 查询引擎则更侧重于向量化计算、并发查询处理和数据缓存。

作为较新的尝试，HyPer 为混合在线分析处理和在线事务处理工作负载提供了一个高性能的主存数据库，并引入了 DataBlocks [15]。DataBlocks 以如下方式扩展了 PAX 的理念。首先，每个页面都包含若干可按字节寻址的属性数据压缩块。因此，它们与列存储系统中的压缩列类似。其次，HyPer 引入了位置小型物化聚合（PSAM），这是一种存储在每个页面中的特殊轻量级索引。PSAM 是一个条目表，其中每个属性范围 $[b, e)$ 都指定了可能匹配的元组，即相应压缩块中的位置。HyPer 利用这种混合存储显著加快了扫描（对属性的谓词操作）。尽管 DataBlocks 的理念很有意思，在思路上也与混合物化相似，但 HyPer 是一个主存数据库，专为混合工作负载而设计。因此，它与我们所讨论的分布式磁盘列存储系统有所不同。

最后，有人尝试使用 PAX 来提升 PostgreSQL 在现代固态硬盘上的性能 [16]。作者提出了一组专用算子，其中的 FlashJoin 运用了与混合物化非常相似的技术。在一系列 FlashJoin 中，位置和值可以共存，从而实现高效的数据访问。在最后一个 FlashJoin 中，常规元组会被重建，查询处理随后以经典的行存储方式继续进行。

由此可见，针对基于磁盘的列存储系统中的混合物化，相关研究仍然不足。在我们此前的工作 [2], 中，我们曾简要提及混合物化是分布式查询中一种非常理想的优化，但并未对此进行深入研究。本文将继续深入探讨混合物化，以及它优于其他策略的各种场景。

III. 混合查询模型

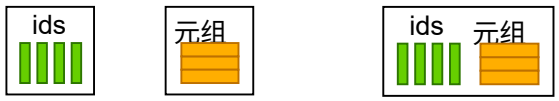
为了支持一种新的物化策略，列存储需要提供一种新的数据表示和若干算子。对于混合物化，主要

难点在于构建一个足够灵活、同时又能支持高效实现的模型。

我们在研究论文中没有找到关于列存储混合查询模型的详细讨论。因此，我们基于 PosDB 中混合物化的实现，提出了我们的构想。

A. 数据表示

PosDB 使用基于数据块的 Volcano 查询模型 [17],，其中每个查询计划都由交换数据块的算子组成。因此，我们需要一种新的混合数据块，以便同时存储位置和值，见图 2c。



(a) 位置型 (b) 元组 (c) 混合

图 2：PosDB’23 中的数据块结构

一个混合块由一组位置列和一个元组数组组成（两者都可以为空）。一组

位置列实际上就是一个位置块，在物化点之前供提前、晚期和超晚期物化使用，见图 2a。它能高效执行列专属操作，例如按位置读取属性值，并根据谓词筛选属性值 [6], [18],。

从元组块中取出一个元组数组，在物化点之后供上述物化策略使用，见图 2b。对于值密集型操作，元组表示更为合适，因为在这种情况下，算子可以受益于空间局部性 [18]。

B. 混合算子内部结构

晚期物化和超晚期物化使用一个特殊的物化点，将位置块转换为元组块。混合物化采用了不同的方法：在许多算子中，属性会按需物化。

这意味着混合算子既要实现其主要算法，也要处理数据转换。我们考虑三种主要情况：

1) 在算子之前物化一个新属性。2) 将部分输入位置和值传递通过算子。3) 将算子的结果作为位置列或元组中的附加属性添加到输出混合块中。

可以将核心算子算法和所有数据转换都实现在同一个实体中，但这样会使查询计划分析和优化变得困难。因此，我们采用 FlashJoin [16] 的思路，将每个混合算子拆分为三个部分：获取、核心和合并，并将它们实现为独立的物理算子。

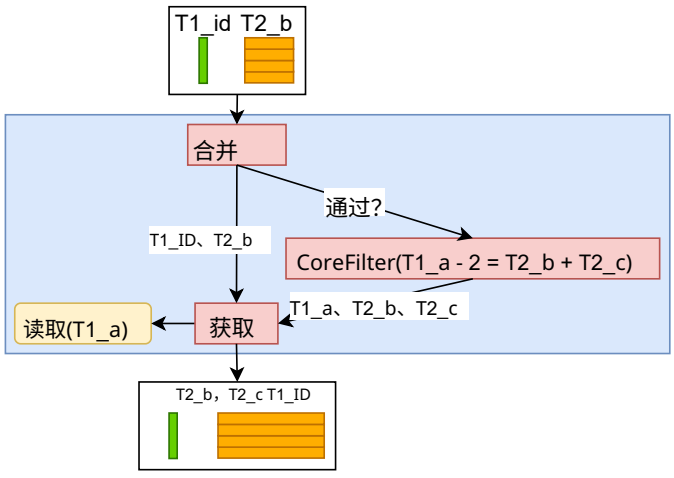


图 3：混合自连接算子的内部结构

图 3 展示了一个示例混合自连接在物理层上的组织方式。为方便起见，我们将 $T1$ 的位置记为 $T1$ id 列，后文也采用类似记法。获取会物化 $T1.a$，并提供对 $T2.b$ 和 $T2.c$ 的访问。随后，CoreFilter 检查 $T1.a - 2 = T2.b + T2.c$ 是否成立；如果成立，合并就会将 $T1$ 和 $T2.b$ 的位置添加到结果混合块中。

请注意，只要能提供所需的值，核心部分就可以与任何获取实现协同工作。同样，核心生成数据的方式与合并的实现方式无关。因此，我们可以实现任意数量的专用获取算子和合并算子，并在查询优化和物理计划创建期间选择效率最高的算子。

例如，图 4 展示了三个专用的 Fetch-Combine 配对。如果我们知道只会用到位置或元组，就可以简化块处理，采用经典的列模型或行模型（图 4a、图 4b）。这样，混合物化就能高效模拟晚期或超晚期物化。初步基准测试显示，模拟版本与“原始”物化策略之间存在几个百分点的差异（这很可能是由具体实现因素造成的）。

作为一个更有趣的例子，我们可以创建一个特殊的 CombineHashJoin，在执行混合哈希连接时构建混合块，如图 4c 所示。CoreHashJoin 使用左子树的获取算子在内存中构建哈希表。然后，它按顺序处理右子树的获取算子返回的值，并调用 CombineHashJoin 来构建结果块。CombineHashJoin 了解哈希连接的具体细节，并同时使用右侧获取算子返回的数据和连接算子内部的哈希表。此外，对于只需使用哈希表数据或右侧获取算子数据的情况，我们也可以设计特殊的 CombineHashJoin。

根据系统架构，我们可以提供数十种获取-合并特化，因为它们不会改变核心算子。我们只需为查询规划器添加启发式规则，以找到最佳特化。

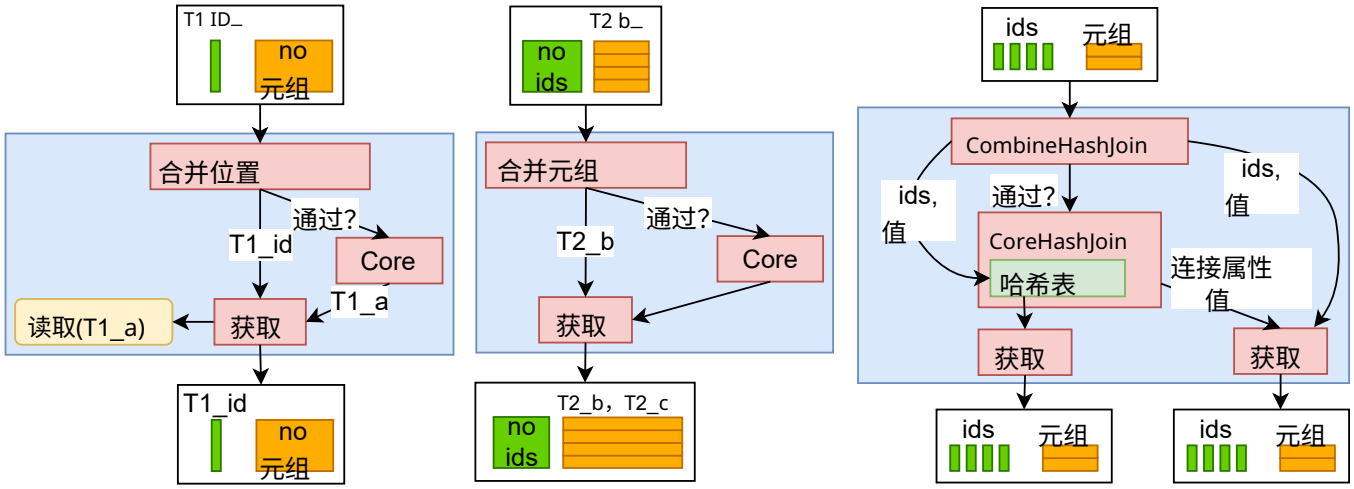
C. 混合算子代数

我们开发了一组最小的混合算子，足以处理简单的 SPJ 查询。每个算子都接受特定类型的块作为输入（可能有多个参数），并返回其他类型的块。在以下列表中，输入块的类型列在 $\rightarrow$ 之前，结果块的类型列在其后。总的来说，我们实现了以下算子：

1) HYDataSource：Nil $\rightarrow$混合块；2) HYFilter、HYProject、HYMaterialize：混合块 $\rightarrow$混合块；3) HYHashJoin、HYNestedLoopJoin：{混合块1、混合块2} $\rightarrow$ 混合块；4) HYToTuple：混合块 $\rightarrow$ 元组块。

这里，HYDataSource 为表生成位置，随后 HYMaterialize、HYProject 和 HYFilter 按需添加、筛选和移除属性值。HYHashJoin 和 HYNestedLoopJoin 连接两个表；所有数据都物化后，HYToTuple 会将混合块转换为元组块。

我们有意省略了聚合算子，因为我们专注于不涉及复杂聚合的 SPJ 查询。简单的选择可以由 HYMaterialize 和 HYProject 支持，也可以由超晚期物化模型中的聚合算子支持，如下一节所示。



（a）对于位置　　（b）对于元组　　（c）对于哈希连接

图 4：获取-合并特化

每个 HYOperator 都由获取、合并和核心部分组成。为简化起见，我们省略获取和合并部分的细节，并假设它们在每个查询计划中都经过了高效的专门化处理。根据所需的细节程度，我们还会同时使用 HYOperator 表示法以及详细的获取-合并-核心表示法。

D. 转换到混合模型及从混合模型转换出去

使用早期物化、晚期物化和超晚期物化时，在任意时刻我们都只能处理位置或元组。借助混合物化，我们可以灵活地合并所有物化模型，按需读取值，并按照需要将其缓存任意长的时间。

例如，我们可以让左树使用超晚期物化、右树使用早期物化进行连接，如图 5a 所示。左树向我们提供一组位置列，我们将其视为不包含元组的混合块。右树向我们提供一组元组，我们将其视为不包含位置的混合块。然后，我们可以执行常规的混合连接，并继续在混合模型中执行。

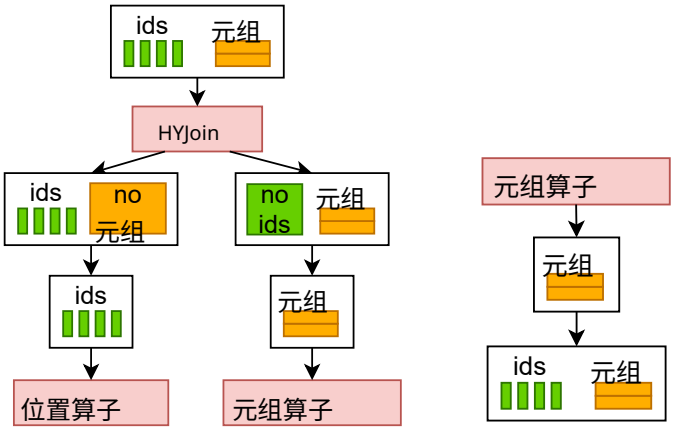
同样，如图 5b 所示，一旦所有数据都已物化，我们就可以轻松地从混合模块切换到经典的基于元组的处理。

如果我们拥有高度优化的列处理引擎或行处理引擎，并希望通过混合物化加速查询计划的某些部分，那么这种转换会很有用。当少量或部分数据以元组形式从外部源接收时，采用晚期物化和超晚期物化也会很有用。

E. 查询计划示例

为总结混合模型的讨论，我们通过图 6 展示一个使用混合物化执行的简单查询。

首先，HYFilter 使用位置和读取器应用谓词，与晚期物化中的做法相同。然后，两个提取算子获取 $T1$ a、 $T1$ b 和 $T2$ c 的数据，从而完全物化 $T1$。



（a）切换到混合模式　　（b）从混合模式切换

图 5：物化策略之间的转换

混合嵌套循环连接在包含 $T1$ b 和 $T2$ id 的混合块中合并 $T1$ 和 $T2$。连接完成后，HYMaterialize 使用 $T2$ id 物化 $T2$ b，并将生成的元组发送给用户。

有了这样的混合计划，问题是：它们在何时以及为何会比采用晚期或超晚期物化的计划更高效?

IV. 混合物化的优势

比较多种算法的一种经典方法是建立理论模型，找出有趣的情况，并通过实验加以证明。为物化策略建立一个完整的查询模型需要占用相当多的篇幅，因此我们不妨根据它们的缺点进行非正式分析：

1）晚期物化：

- 读取随后在后续算子（例如连接）中被丢弃的额外数据，

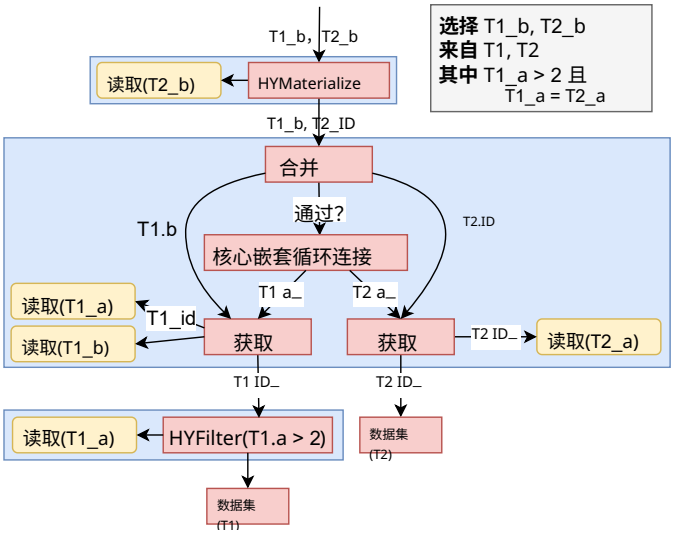


图 6：混合查询模型中的简单查询

- 将数据传递给所有这些算子。

2）超晚期物化：

- 多次读取属性值（重复访问），
- 以随机访问方式从磁盘读取属性值（即所谓的乱序探测问题）。

基于这些缺点，我们将分析本地和分布式情形，以确定混合物化何时能够取得更好的性能。

A. 本地情形。超晚期物化与混合物化

先前研究 [2] 已证明，超晚期物化对分析型工作负载相当有效。在大多数情况下，Volcano 模型和现代缓冲区管理器使重复访问属性数据的成本很低。然而，在某些情况下，超晚期物化会导致从磁盘多次重新读取，或引发乱序探测。

假设有一个查询，其中两个大表 $T1$ 和 $T2$ 之间执行嵌套循环连接，如图 7 所示。为简化起见，假设：

1) 对于 $T2$ a 中的每个值，$T1$ a 中都有一个与之对应（相等）的值；2) 嵌套循环连接会保留右子树中记录的顺序。

混合物化和超晚期物化以相似的方式处理嵌套循环连接：对于每个 $T2$ a，找到对应的 $T1$ a 并构建下一条结果记录。主要区别在于，混合物化可以在连接之前读取 $T1$ b，此时 $T1$ 的位置仍然是顺序的。与此同时，超晚期物化会迫使查询引擎在连接之后、物化算子中读取 $T1$ b。

在最坏情况下，超晚期物化中的嵌套循环连接会产生连续的 $T2$ ids，但与之对应的 $T1$ ids 会稀疏地分布在磁盘上的多个 $T1$ b页中，且不会按顺序排列。例如，假设连接返回了以下位置对：$\{T2\ id = 1, T1\ id = 5\}$、$\{T2\ id = 2, T1\ id = 1000000\}$、$\{T2\ id = 3, T1\ id = 3000\}$。此类 $T1$ ids 需要通过昂贵的随机访问来读取 $T1$ bs，这被称为探测外问题 [13]。另一方面，混合物化会在连接之前按顺序读取 $T1$ b，因此效率很高。

如果我们将嵌套循环连接替换为哈希连接，也会出现类似的问题。在超晚期物化的情况下，哈希连接会为 $T1$ id 创建一个哈希表，随后生成包含 $T2$ id，$T1$ id 的配对，其 $T2$ id 按顺序排列，但 $T1$ id 则随机分布。

即使列本身能够装入内存，乱序探测也可能在许多情况下出现。这可能是因为数据库管理系统必须将部分内存用于哈希表、外部排序以及其他并发运行的查询。

B. 本地情形。晚期物化与混合物化

在本地情形中，混合物化相较于晚期物化，具有与超晚期物化相同的优势。具体而言，晚期物化会迫使查询引擎：

1）即使连接会进行筛选（即丢弃大多数行），也会在连接前读取大量数据；2）在将值传递给后续算子的过程中复制更多数据；3）增大数据结构（例如哈希表）的规模。

最后两点会带来不那么明显、但很重要的缺点：

1）缓存效率（局部性）降低；2）getNext() 调用次数增加；3）数据库管理系统的整体查询带宽降低，例如，系统将有更少的内存用于并发运行的查询所使用的哈希表。

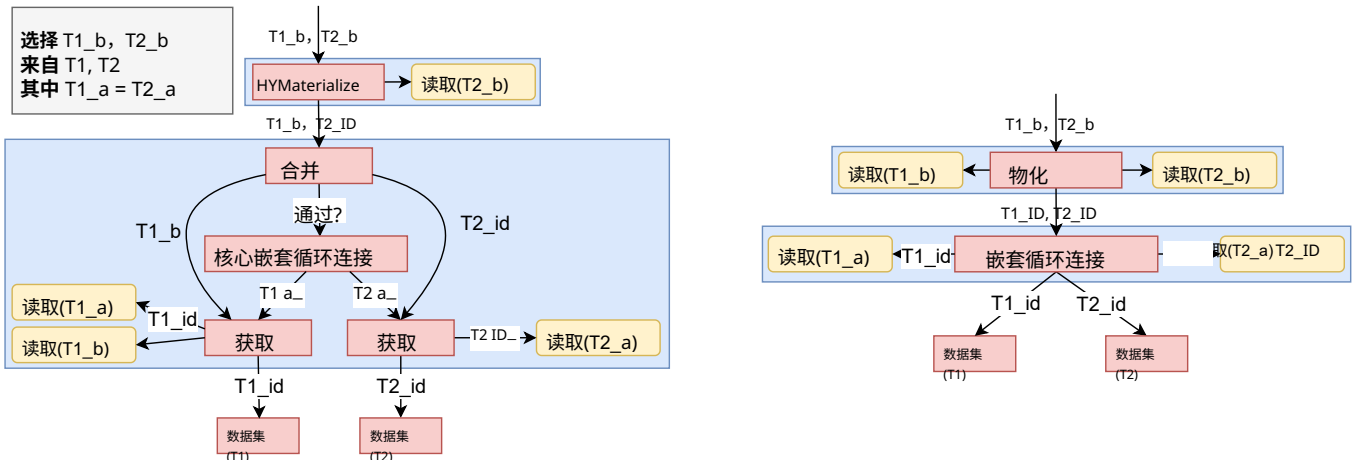
C. 分布式场景。超晚期物化与混合物化

对于分布式查询，即使不存在乱序探测问题，重复访问的成本通常也很高。假设我们想在 Node1 上执行某个算子，并使用 $T1$ 位置读取 $T1$ a，而 $T1$ a 位于 Node2 上。在这种情况下（参见 [2]中的第 4.4 节），我们需要执行以下操作：

1）将 $T1$ 位置发送回 Node2；2）等待 Node2 从磁盘读取 $T1$ a；3）等待 Node2 将 $T1$ a 传回 Node1。

这个过程开销相当大，因为步骤 1 和 3 受网络延迟和带宽的限制。我们可以尝试让它们异步执行，但要读取 $T1$ a，我们需要知道从 Node1 上的前序算子接收到的 $T1$ 位置。因此，我们无法轻易提前请求 $T1$ a。

在混合物化的情况下，查询引擎可以将 $T1$ a 与 $T1$ 位置一并发送到 Node1。因此，$T1$数据会从 Node2 直接单向发送到 Node1，无需额外循环。这能加快查询执行速度，并且由于数据流更简单，更便于分析和优化计划。



(a) 混合物化　　(b) 超晚期物化

图 7：两个大表之间的连接

D. 分布式场景：晚期物化与混合物化

在位置上进行操作时，晚期物化与上文所述的超晚期物化有同样的缺点。不过，晚期物化只在谓词中使用位置，而谓词通常在节点内执行。因此，晚期物化的重复访问成本在本地和分布式情况下应大致相同。

然而，采用晚期物化的分布式查询还有另一个缺点，即在算子间传递额外数据。假设有一个查询：

1) 在 Node1 上根据 $T1_a$ 筛选记录，Node1 本地有 $T1$ 数据；2) 使用 $T1$ 通过若干分布式连接连接一些表，其中一个连接谓词使用 $T1\ a$；3) 将包含 $T1\ b$、$T1\ c$、$T1\ d$等属性的记录返回给 Node2 上的用户。

采用晚期物化时，我们必须在 Node1 上应用筛选条件后读取所有$T1_a$、$T1\ b$、$T1\ c$和 $T1\ d$，并将它们传递经过所有分布式算子。具体增加的数据量取决于属性大小，可能会使通过网络发送的数据量增加数倍。网络通信开销很大，通常是分布式系统中的瓶颈，因此网络流量增加数倍会显著减慢查询处理速度。

使用混合物化时，如果 Node2 有 $T1$ 数据，查询引擎可以仅在 Node1 上执行筛选后物化 $T1\ a$，并在之后读取 $T1\ b$、$T1\ c$、$T1\ d$。这样可以降低网络负载并加快查询处理。请注意，由于连接，晚期物化计划必须为每个结果行发送 $T1\ b$、$T1\ c$ 和 $T1\ d$。即使 $T1$ 的行基数很小，计划中传递的行数也可能非常多。当参与连接的其他表很大，且连接属性中包含许多匹配值时，就可能出现这种情况。

E. 总结与要点

我们已经证明，与晚期物化和超晚期物化相比，混合物化能够加快本地和分布式场景下的查询处理。混合查询模型具有灵活性，因此具备提升性能的潜力；在该模型中，查询引擎可以在查询执行过程中较早或较晚地物化单个属性。因此，混合物化既能克服超晚期物化的乱序探测问题，也能解决晚期物化中读取和处理额外数据的问题。

我们认为，混合物化值得实现并进一步研究。它在分布式环境中尤其值得关注，因为分布式系统需要通过网络发送大量数据，容易受到网络速度缓慢的影响。我们指出了分布式查询可从混合物化中获益的两种情形，但此类情形应该不止这些，而且还可能与分布式连接算法有关。

V. 实验评估

正如我们之前指出的，混合物化对分布式和集中式场景都可能有用。作为第一步，本文决定重点开展集中式实验。许多系统只支持本地查询执行，这种场景也更稳定，不受网络子系统优化、网络配置、网络延迟和带宽等实现细节的影响。对于分布式场景，我们已在第IV-C节和第IV-D节讨论了混合物化的潜在优势。详细的分布式实验值得作为未来的一项独立研究课题。

我们的研究基于 TPC-H 基准测试 [12]，其规模因子（比例因子）处于 [1, 50] 范围内。表 II 展示了各个 TPC-H 表中确切的行数，以便于实验分析。

我们已将此前使用的 SSB [2], [10] 替换为 TPC-H，因为它的查询具有更复杂的数据流，

表 II：TPC-H 表大小

表名	元组数量
区域	5
国家	25
供应商	比例因子 * 10 000
客户	比例因子 * 150 000
part	比例因子 * 200 000
零件供应	比例因子 * 800 000
订单	比例因子 * 1 500 000
订单明细	比例因子 * 6 000 000

同时仍然足够简单，便于比较执行引擎，而非比较优化器的输出。

具体而言，我们首先从 Q5 的简化版本入手，用它来展示采用不同物化策略的查询计划，并比较它们的性能。接着，我们基于 Q9 中经过少许修改的选择-投影-连接部分，构造一个更贴近实际（也更复杂）的查询，其中包含多个数据表之间的连接。

所有实验均在以下硬件配置的台式机上运行：AMD Ryzen 9 3900X、GIGABYTE X570 AORUS ELITE、Kingston HyperX FURY Black HX434C16FB3K2/32 32GB、512 GB SSD M.2 Patriot Viper VPN100-512GM28H。使用的软件版本如下：Ubuntu 22.04 LTS、GCC 11.3.0。每次实验运行前均未进行预热，并且在每次启动前清空操作系统页缓存。由于每次运行得出的数据几乎相同，我们展示的是十次运行的平均结果。

A. 分析型工作负载

混合物化可以与晚期物化和超晚期物化共存，也可以通过特殊的 Fetch-Combine 配对来模拟两者，如第 III-B 节所述。

这意味着，如果在 SSB 上超晚期物化的速度是晚期物化的两倍以上，那么混合物化的性能将接近超晚期物化。本文旨在展示这样的情形：晚期和超晚期策略都存在根本性局限，这些局限源于其执行流程，而非实现细节。因此，我们决定不再采用 SSB，因为它可支持的查询类型相当有限，并不适合我们的研究目的。SSB 的问题在于，它考虑的是一种特定情形：一张大型事实表和若干非常小的维度表。

B. 混合物化策略

在混合查询模型中，提前物化某个属性的成本，与稍后物化该属性时将位置传递给后续计划算子的成本之间存在权衡。对于特定查询中的某些属性，提前读取数据是避免乱序探测和高昂远程访问成本的有效方式。然而，对于同一查询中的其他属性，我们可以稍后再物化数据，从而减少内存数据结构的大小以及传递数据的成本

通过算子处理数据块。

为混合物化构建成本模型和功能完备的优化器是一个有趣的课题。此类优化器至少应考虑每个单独属性的物化点（时间）、各种表的大小以及节点上可用随机存取存储器的容量。然而，本文旨在展示列存储如何支持混合物化，并证明在某些情况下，混合查询模型具有相当大的优势，因此值得进一步研究。

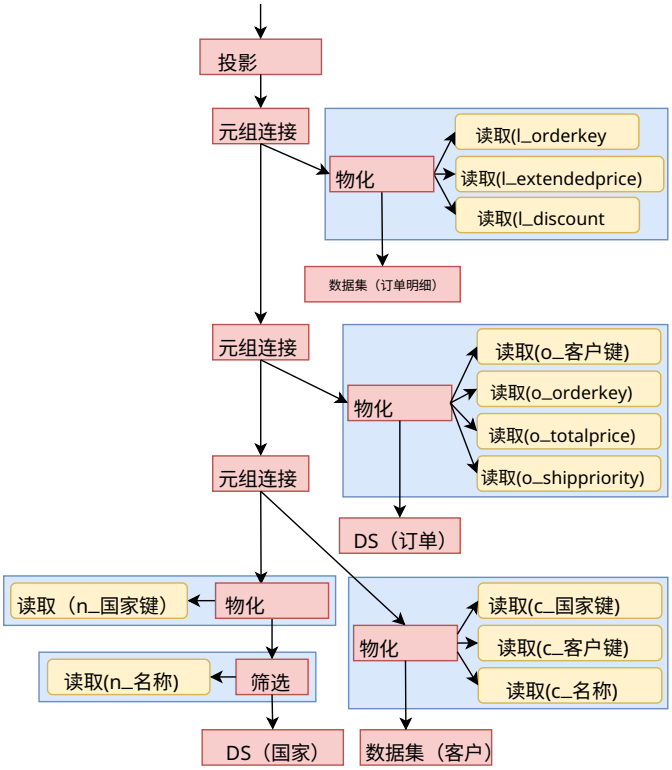


图 8：PosDB'22 中的晚期物化

因此，我们没有构建查询优化器，而是采用一种简单的启发式方法：在合理可行的情况下尽可能晚地物化数据。这意味着我们以类似于超晚期物化的方式创建混合物化计划，然后提前物化部分属性，以缓解乱序探测和重复访问带来的问题。

C. 实验*1*

在第一个实验中，我们使用了 TPC-H Q5 中经过修改的选择-投影-连接部分。我们有意省略了聚合和子查询部分，因为物化策略主要影响连接和谓词。我们还添加了与经过筛选的国家表的连接，并对 c name 列进行选择：

选择
c_名称，o 总价，o 发货优先级，l 订单键，l 扩展价格
*(1 – l 折扣)
FROM
国家，客户，订单，订单明细
其中
n_名称 = 阿尔及利亚**并且**n 国家键 =c 国家键 **并且**c 客户键 =o 客户键 **并且**o 订单键 =l 订单键;

我们依据以下原则设计了此查询：
1）三个中大型表之间进行三次连接。2）随机存取存储器足够大，可以使用哈希连接。3）第四次连接使用固定大小的表 nation，使其中一次连接具有 1/25 选择率，从而起到“筛选”作用（“阿尔及利亚”筛选）。这使晚期物化代价高昂，因为它必须处理 orders 和订单明细的所有记录及属性。4）连接链使得在超晚期物化中读取 c_名称值时，客户位置（c_标识符）处于无序状态。由于需要进行乱序探测，超晚期物化的代价因此很高。

图 8 和图 9 展示了晚期物化和超晚期物化模型中的具体查询计划，PosDB’22 使用的正是这两种模型。

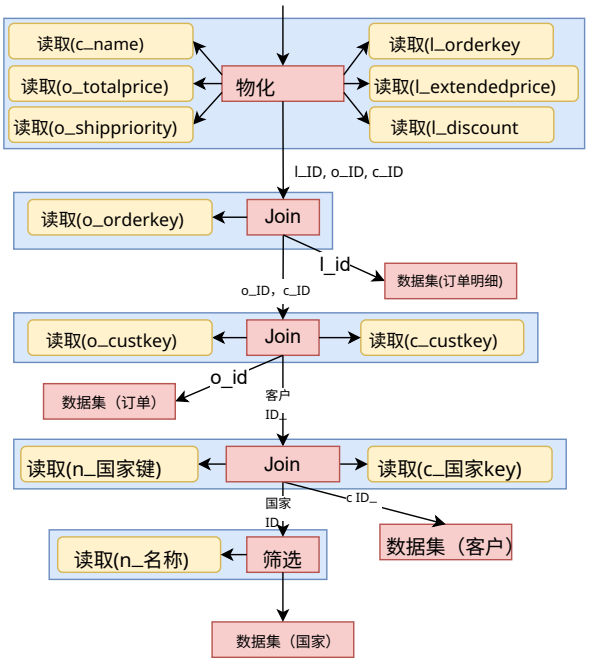


图 9：PosDB’22 中的超晚期物化

混合物化可以结合这两种方法的优势：在最终的 HYMaterialize 中物化 orders 和订单明细数据，同时在客户位置仍有序时缓存 c_名称。这样可以减少处理的数据量，并避免乱序探测。图 10 展示了 PosDB’23 中采用混合物化的查询计划。

为了证明我们的观点，我们首先在规模因子（SF）为 1 到 50 的 TPC-H 上评估了所有策略，如图 11a 所示。可以看出，在较小的 SF 下，超晚期物化和混合物化的性能大致相同，并且明显快于晚期物化。在规模因子较大时，由于无法轻易将整个 c_名称列与所有哈希表一同放入随机存取存储器，乱序探测问题会更加明显。因此，超晚期物化的开销越来越大。相反，混合物化表现出适度的增长，最终在 SF 50 时比超晚期物化快近两倍。

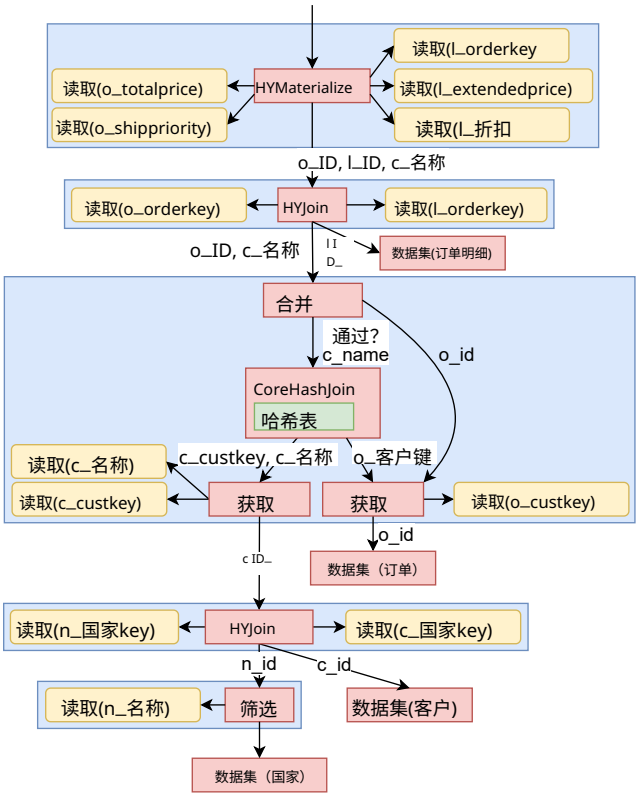


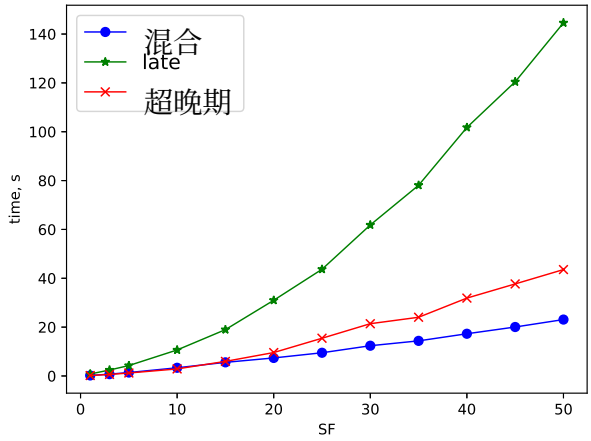
图 10：PosDB’23 中的混合物化

我们再次重复了相同的实验，这次查询的 SELECT 子句中不包含 c_name。这样便消除了乱序探测。正如图 11b 所示，在所有比例因子下，混合物化和超晚期物化都比晚期物化更快。由于实现开销，混合物化比超晚期物化慢了几个百分点。

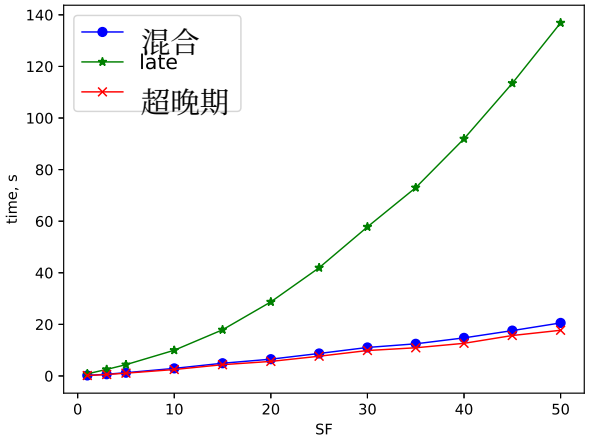
D. 实验 *2*

在第二个实验中，我们使用 TPC-H Q9 中的选择-投影-连接部分，这为我们提供了一个更贴近实际的查询示例，其连接结构也更复杂。为专注于连接和谓词，我们再次省略子查询和聚合部分：

```
SELECT
    n_名称，o_总价，l_扩展价格 * (1 – l_折扣) —ps_供应成本
* l_数量来自零件，供应商，订单明细，零件供应，订单，国家
其中s_供应商键 =l_供应商键并且ps_供应商键 = l_供应商键并且
ps_零件键 = l_零件键并且p_零件键 = l_零件键并且o_订单键 =
l_订单键并且s_国家键 = n_国家键并且 p_尺寸 > 25;
```



(a) 使用乱序探测



(b) 不使用乱序探测

图11：TPC-H 的修改版 Q5

为简化起见，我们将 extract(year from o_orderdate) 替换为 o_totalprice，并将 p_name like '\%:?\%' 替换为 p_size > 25。

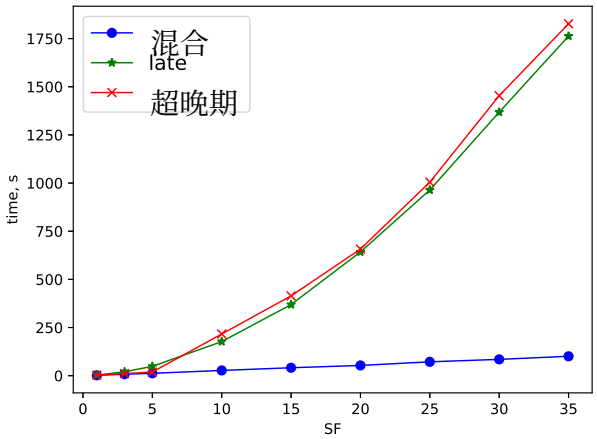


图12：修改后的 TPC-H Q9

如图12所示，我们研究了 TPC-H 上这一查询在所有物化策略下的性能，比例因子（SF）范围为 1 到 35。可以看出，混合物化的性能最佳，速度提升超过两倍。晚期物化速度较慢，因为 p_size 谓词与零件、零件供应和订单明细之间的连接，会筛选掉大量记录，其效果类似于过滤连接。超晚期策略速度较慢，因为它必须通过随机访问读取 ps_supplycost（即执行乱序探测）。

E. 总结

在本研究中，我们重点开展了基于 TPC-H 基准测试的集中式实验，并比较了各种物化策略。

我们的研究结果表明：

1）当需要乱序访问时，超晚期物化会变慢。2）当查询计划包含过滤连接时，晚期物化会变慢。3）混合物化结合了这两种方法，几乎可以将速度提升两倍。

VI. 未来计划

我们计划从多个方向扩展对混合物化的研究：

1）将混合物化与晚期物化进行比较，并考察旁路信息传递等特定优化 [11]。2）深入研究分布式案例，发掘更多有利于混合物化的数据流模式，并通过实验加以研究。3）为混合查询模型中的每个属性制定选择物化点的最优策略。

VII. 结论

本文继续了我们先前对基于磁盘的列存储中物化策略的研究 [2]。首先，我们回顾了现有的提前、晚期和超晚期物化策略及其主要特征。我们指出，所有这些策略都缺乏灵活性，而且每种策略在某些情况下都会遇到根本性局限。

在此基础上，我们提出了一种混合物化策略，它能够发挥所有现有策略的优势，既延迟数据物化，又避免代价高昂的重复数据访问。已有研究论文曾简要提及混合位置-值处理，但尚未对混合物化进行深入研究。

因此，我们介绍了混合块结构、混合算子内部结构，以及混合算子的整体代数。

接着，我们展示了四种本地和分布式场景，在这些场景中，混合策略相较于晚期物化和超晚期物化具有根本优势。最后，我们在 PosDB（一种分布式、基于磁盘的列存储系统）中实现了我们的方法。

使用 TPC-H [12] 基准中的 Q5 和 Q9 修改版本进行的实验评估表明，混合物化使速度几乎提升了一倍。

致谢

我们感谢 Anna Smirnova 在论文准备过程中提供的帮助。

参考文献

[1] D. J. Abadi、D. S. Myers、D. J. DeWitt 和 S. R. Madden，“面向列的数据库管理系统中的物化策略”，载于*2007*年*IEEE*第*23*届数据工程国际会议，2007年，第466–475页。[2] G. A. Chernishev、V. Galaktionov、V. V. Grigorev、E. Klyuchikov 和 K. Smirnov，“基于磁盘的列存储晚期物化策略综合研究”，载于第*24*届大数据设计、优化、语言与分析处理国际研讨会（*DOLAP*）论文集（与第*25*届数据库技术扩展国际会议及第*25*届数据库理论国际会议（*EDBT/ICDT 2022*）同期举行），爱丁堡，英国，*2022*年*3*月*29*日，CEUR研讨会论文集系列，K. Stefanidis 和 L. Golab 编，第3130卷。CEUR-WS.org，2022年，第21–30页。 [在线]。地址：http://ceur-ws.org/Vol-3130/paper3.pdf [3] D. Abadi、S. Madden 和 M. Ferreira，“在面向列的数据库系统中集成压缩与执行”，载于*2006*年*ACM SIGMOD*数据管理国际会议论文集，SIGMOD ’06系列。纽约，纽约州，美国：计算机协会，2006年，第671–682页。 [在线]。地址：https://doi.org/10 1145 1142473 1142548 。/ 。M. Zukowski 和 P. A. Boncz，[4] “Vectorwise：超越列存储”，*IEEE*数据工程简报，第35卷，第1期，第21–27页，2012年。 [在线]。地址：http://sites.computer.org/debull/A12mar/vectorwise.pdf [5] A. Slesarev、E. Klyuchikov、K. Smirnov 和 G. A. Chernishev，“重新审视列存储中的数据压缩”，载于模型与数据工程——第*10*届国际会议，*MEDI 2021*，塔林，爱沙尼亚，*2021*年*6*月*21*日至*23*日，论文集，计算机科学讲义系列，J. C. Attiogb´e 和 S. B. Yahia 编，第12732卷。施普林格，2021年，第279–292页。 [在线]。地址：https://doi.org/10 1007 978 3 030 78428 7 22 。/ - - - - [6] D. Abadi、P. Boncz 和 S. Harizopoulos，现代面向列数据库系统的设计与实现。汉诺威，马萨诸塞州，美国：Now Publishers Inc.，2013年。[7] S. Idreos、F. Groffen、N. Nes、S. Manegold、K. S. Mullender 和 M. L. Kersten，“MonetDB：面向列数据库架构的二十年研究”，*IEEE*数据工程简报，第35卷，第1期，第40–45页，2012年。 [在线]。地址：http://sites.computer.org/debull/ A12mar/monetdb.pdf [8] P. A. Boncz 和 M. L. Kersten，“用于查询碎片化世界的MIL原语”，*VLDB*期刊，第8卷，第2期，第101–119页，10月1999 [ ] 10 1007 007780050076 。在线。地址：http://dx.doi.org/ 。/s [9] M. Stonebraker、D. J. Abadi、A. Batkin、X. Chen、M. Cherniack、M. Ferreira、E. Lau、A. Lin、S. Madden、E. O’Neil、P. O’Neil、A. Rasin、N. Tran 和 S. Zdonik，“C-Store：一种面向列的数据库管理系统”，载于第*31*届超大型数据库国际会议论文集，VLDB ’05系列。VLDB基金会，2005年，第553–564页。[10] “P. E. O’Neil、E. J. O’Neil 和 X. Chen。《星型模式基准测试（SSB）》。 ”http://www.cs.umb.edu/~poneil/StarSchemaB.PDF，2009年，访问日期：10/09/2017。[11] L. Shrinivas、S. Bodagala、R. Varadarajan、A. Cary、V. Bharathan 和 C. Bear，“Vertica分析型数据库中的物化策略：经验总结”，载于*2013*年*IEEE*第*29*届数据工程国际会议（*ICDE*），2013年，第1196–1207页。[12] “TPC-H基准测试。2.17.3版”，http://www.tpc.org/tpch。[13] S. Harizopoulos、D. Abadi 和 P. Boncz，“面向列的数据库系统，VLDB 2009教程。”2009年。 [在线]。地址：nms.csail.mit. edu/~stavros/pubs/tutorial2009-column stores.pdf

[14] A. Ailamaki、D. J. DeWitt、M. D. Hill 和 M. Skounakis，“为缓存性能编织关系”，载于第*27*届超大型数据库国际会议论文集，VLDB ’01系列。旧金山，加利福尼亚州，美国：Morgan Kaufmann 出版社，2001年，第169–180页。[15] H. Lang、T. M¨uhlbauer、F. Funke、P. A. Boncz、T. Neumann 和 A. Kemper，“数据块：在压缩存储上结合向量化和编译的混合在线事务处理与在线分析处理”，载于*2016*年数据管理国际会议论文集，SIGMOD ’16系列。纽约，纽约州，美国：计算机协会，2016年，第 311 326 [ ] 10 1145 2882903 2882925 – [16] 。在线。地址：https://doi.org/ 。/ 。D. Tsirogiannis 、 S. Harizopoulos 、 M. A. Shah、J. L. Wiener 和 G. Graefe，“面向固态硬盘的查询处理技术”，载于*2009*年*ACM SIGMOD*数据管理国际会议论文集，SIGMOD ’09系列。纽约，纽约州，美国：计算机协会，2009年，第59–72页。 [在线]。地址：https://doi.org/10 1145 1559845 1559854 。/ 。G. Graefe，“大型数据库[17] 的查询求值技术”，*ACM*计算调查，第25卷，第2期，第73–169页，1993年6月。 [在线]。地址：http://doi.acm.org/10 1145 152610 152611 。/ 。M. Zukowski、N. [18] Nes 和 P. Boncz，“DSM与NSM：面向块的查询处理中CPU性能权衡”，载于第*4*届新型硬件数据管理国际研讨会论文集，DaMoN ’08系列。纽约，纽约州，美国：ACM，2008年，第47–54页。