S/C：使用有界内存加速数据物化

Zhaoheng Li、Xinyu Pi、Yongjoo Park 伊利诺伊大学厄巴纳-香槟分校的 CreateLab{zl20, xinyupi2,yongjoo}@illinois.edu

arXiv:2303.09774v1 [cs.DB] 17 Mar 2023

摘要——借助数据流水线工具和 SQL 的表达能力，管理相互依赖的物化视图（MV）变得越来越容易。每当摄取新数据时（例如每天），这些物化视图都会反复更新。数据库管理员可以通过一致的方式，针对不同类型的更新（全量更新与增量更新）以及不同的系统（单节点、分布式、云托管）观察性能指标（例如每个物化视图的刷新时间、磁盘占用）。一个错失的机会是，现有数据系统将这些物化视图更新视为彼此独立的 SQL 语句，而没有充分利用其依赖信息和性能指标。然而，如果我们知道某条 SQL 语句的结果会立即用于后续操作，那么这些后续操作就不必等到中间结果完全物化到存储中，因为这些结果已经可以直接从内存中获取。当然，这可能会带来代价，因为将这些结果保留在内存中（即使只是暂时保留）会减少可用内存；因此，我们应谨慎做出决策。

本文介绍了一种名为 S/C 的新系统，它通过高效创建和更新一组彼此具有无环依赖关系的物化视图来解决这一问题。S/C 合理利用有界内存，通过短路跳过开销高昂的读取和写入操作，缩短端到端物化视图刷新时间；S/C 的目标函数能够准确估算在特定时段将中间数据保留在内存中所节省的时间。我们的解决方案联合优化物化视图刷新顺序、要保留在内存中的数据，以及何时从内存中释放这些数据。从整体上看，S/C 仍会严格按照物化视图定义物化所有数据，因此不会影响任何服务级别协议。在使用 TPC-DS 数据集（最大为 1TB）进行的实验中，我们表明，S/C 的优化仅使用 1.6GB 内存，即可将端到端运行时间加速 **1.04**×–5.**08**× 倍。

I. 引言

为加速查询处理，数据库中常使用中间表和（物化）视图——我们统称为物化视图 [1]–[5]。当物化视图之间存在依赖关系时，更新会按照拓扑序执行，以尽量减少冗余计算；这可以借助 Google Napa [6], dbt [7], Apache Airflow [8], LookML [9], 等工具来管理。利用 SQL 的广泛应用和表达能力，我们可以为各种数据系统定义全量刷新和增量更新 [1], [10] ，涵盖从单节点数据库（例如 PostgreSQL [11]）、自行管理的分布式数据仓库（例如 Hive [12],SparkSQL [13], Presto [14]），到云托管数据服务（例如 Snowflake [15], Azure SQL [16]）。数据库管理员可以根据更新流水线的反复运行，获取实测性能指标（例如更新每个物化视图的耗时、每个物化视图的大小），而依赖关系则可以轻松地从物化视图定义中提取。

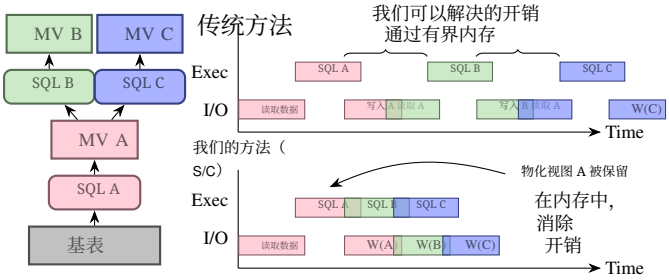


图1：我们的直觉是：利用现有的依赖关系可以缩短物化视图的整体刷新时间。如果我们知道 SQL B 依赖于 SQL A 的输出，就不必等到 SQL A 的结果完全物化后才开始。

错失的机会。 在更新一组彼此存在依赖关系的物化视图时，现有数据系统除了缓存之外，并不会引入专门的优化。我们的想法是，利用（i）物化视图之间依赖关系的明确信息，以及（ii）从以往运行中观测到的性能指标，进一步改进数据系统的操作，从而缩短端到端物化视图刷新时间。在多种可能的方法中，我们基于以下观察，探索一种重要的优化机会：虽然为了满足服务级别协议（SLAs），持久化中间数据可能不可避免，但借助有界内存（如果数据能够放入内存），可以跳过中间数据的写入和读取，让数据系统直接从内存对象读取输入数据，而不是从持久化表读取。图 1 展示了我们的想法：计算完成后，将表A 保留在内存中，这样下游任务SQLB 和SQLC 就能与表A的物化并行执行，且几乎不会产生延迟。由于内存大小有限，只有在这样做能带来显著收益时，才应将中间数据保留在内存中。这种方法具有通用性，因为我们不对计算任务的内容（即全量刷新还是增量物化视图刷新）及其执行方式（即单节点还是分布式）作任何假设。此外，由于某些数据系统支持显式的数据放置（例如，Presto [14], Spark [13], Polars [28]），我们可以利用现有功能轻松实现这项优化，无需修改其内部实现。请注意，我们的方法仍会持久化所有中间表；我们只是利用依赖关系，减少后续更新的等待时间。

表 I：S/C 与其他数据仓库优化问题的比较

主题	优化目标
增量查询处理 [17], [18]	基于不完整数据提前查询，并随着剩余数据到达进行渐进式更新
缓存预取 [19], [20]	针对预期作业的 SLO 感知预取
增量视图维护 [21], [22]	单个物化视图的更新速度更快
物化视图刷新调度 [5], [23], [24]	有些物化视图比其他物化视图更早更新（但总时间保持不变）
中间结果复用 [25]–[27]	缓存并复用中间结果，以加速未来的作业（但不进行作业重排序）
我们的方案（S/C）	通过作业重排序和数据缓存，加速一组物化视图的端到端刷新时间

挑战。遗憾的是，要实现我们的目标在技术上颇具挑战，因为朴素的方法（例如，尽可能多地贪心算法缓存数据，以及使用最近最少使用策略进行缓存淘汰）只能带来有限的性能收益（§ VI）。要获得显著提升，我们必须同时考虑多个因素。首先，我们必须优化物化视图刷新的顺序，因为这会影响哪些中间数据能够保留在内存中，以及何时释放。现成的拓扑排序算法 [29] 未必能为我们的问题生成最优顺序。其次，我们必须确定哪些中间物化视图应保留在有界内存中。根据我们的研究，贪心算法（即有可用空间时便予以保留）或随机选择的效果都不理想。第三，我们必须制定一种能够根据用户工作负载和中间数据大小进行调整的策略；如果用户的工作负载发生变化，固定的启发式策略可能会导致次优解。为应对这些挑战，我们采用一种有原则的方法，以图表示物化视图，并研究多种优化算法，具体内容将在下文介绍。

我们的方法。本文介绍了一种专用于刷新物化视图图的系统（称为 Short-Circuit 或 S/C）。S/C 利用物化视图之间的依赖关系及其观测到的性能指标，在有界内存条件下加速端到端物化视图刷新。物化视图之间的依赖关系通过有向无环图建模，其中节点表示单个物化视图更新，边表示依赖关系。为优化端到端性能，S/C 从宏观层面求解以下问题：

$$\min_{O,U} \sum_{n_i \in O} \text{data-access-time}(n_i; U), \tag{1}$$

其中，O 是物化视图刷新顺序，U 是我们临时将其结果保留在内存中的节点集合，n_i是第 i 个节点。U 必须满足可行性要求；也就是说，在更新物化视图期间的任何时刻，我们保留在内存中的所有中间数据之和都不能超过预设的上限。

为解决上述问题，我们交替优化 O和 U，具体如下：首先，在固定 O的情况下，找到能够最小化目标函数（公式 (1)）的 U；其次，找到能够降低总体内存使用量的O，期望这样的顺序能在后续迭代中得到更优的 U（§ V）。虽然这种方法可能会找到局部最优解，但我们的实证研究表明，从一种专门设计的 DFS（深度优先搜索）变体开始，可以获得高质量的解（§ VI-F）。

与其他工作的比较。本研究采用数学优化来管理数据系统中的内存和调度，重点是通过利用观测到的性能数据来缩短端到端物化视图刷新时间。尽管数据系统的优化和统计技术已应用于各种相关问题（例如，增量查询处理、 [17], [18], [30], 缓存预取 [19], [20], 索引 [31], [32]），但这些技术旨在为独立查询进行资源管理。本研究所解决的问题有显著不同，尤其是还包括对物化视图更新进行联合调度。通过增量更新 [21], [22] 和调度 [5], [23], [24] 来优化物化视图更新，与我们的工作追求相同的高层目标，因为我们的目标是改进数据准备阶段；但本研究所解决的具体问题与这些工作大体正交。关于中间结果复用的研究 [25]–[27], [33] 同样会选择性地存储条目，以加速未来的工作负载（即物化视图刷新运行）；然而，据我们所知，我们是首个同时考虑作业执行顺序（重新排序）以及在有界内存下进行中间结果缓存的研究。表 I 总结了核心差异。

我们的技术贡献有望应用于加速更广泛的一类工作负载，这类工作负载由重复作业组成，其依赖关系以有向无环图表示，例如使用 Hadoop 和 Spark 的提取-加载-转换（ETL），以及通过 Apache Airflow [8] 和 Apache Oozie [34], 等进行的作业协调 [13], 。不过，为了具体讨论，本文聚焦于物化视图刷新。

贡献。我们的贡献如下：

- 系统架构。我们提出了一种系统架构（S/C），旨在缩短数据仓库中物化视图刷新工作负载的端到端运行时间，并提供一种高效的、基于 Presto 的实现 [14] 。（§ III）
- 问题定义。我们提出并正式定义了一个针对物化视图刷新工作负载的优化问题（S/COpt），即选择要在有界内存中持久化的中间数据，以最大限度地缩短读写时间。（§ IV）
- 算法与分析。我们采用交替优化算法来求解 S/COpt。该算法将 S/COpt 分解为两个子问题，并为这两个子问题提供了高效且经验证有效的解决方案。（§ V）
- 实验分析。我们验证了S/C在提升物化视图刷新速度方面的有效性。我们发现，使用 1.6GB 内存目录时，S/C 可将端到端运行时间加速 1.04×–5.08×倍，测试数据集为 TPC-DS 数据集（最高达 1TB）。II

· 现代数据仓库中的物化

配置物化视图（即数据物化）能有效加快 OLAP 查询。在本节中，我们将介绍现代数据仓库中数据物化的典型应用，以及我们观察到的输入/输出开销。

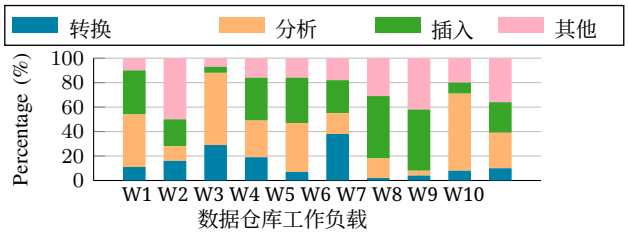


图 2：10 个真实世界工作负载（W1–W10）按查询类型划分的运行时间明细。数据来源：Amirhossein 等人 [35], [36]。

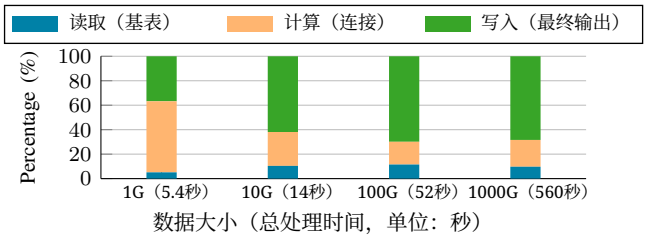


图 3：按操作划分的运行时间。将结果压缩并写入存储所需的时间要长得多。使用云数据仓库连接 TPC-H Q8 中出现的四张表（客户、订单、明细项、国家）。

A. 通过数据物化加速

现代数据仓库存储和管理海量数据。水平扩展可能无法以经济高效的方式降低分析型查询的延迟。为了获得更好的成本效益权衡，人们经常采用数据物化。

海量数据，复杂查询。 真实世界的数据仓库需要管理海量数据。根据对 40 个不同数据仓库账户中的 15.4 亿个查询和 170 万张表进行的分析 [35], [36], 12，1% 的表大于 3.625 TB，29.1% 的表介于 725 GB 和 3.625 TB 之间。大多数分析型查询都包含数据密集且耗时的操作：据报道，31% 的 SELECT 查询至少包含一个连接。32% 的时间耗费在运行时间超过 10 分钟的查询上。根据 Tableau [37], 的另一项分析，由于 BI 驱动的分析与基于 SQL 的表达式之间存在阻抗不匹配，机器生成的查询通常较为复杂。总体而言，大量数据和高成本操作会导致更长的查询延迟。

虽然水平扩展 [13], [14], [38] 和积极缓存 [39] 在一定程度上可以缓解这个问题，但它们并非经济实惠的解决方案。具体而言，为了从缓存中获益，即使查询工作负载相对较轻，用户也必须支付更多费用来维持集群运行 [40]。为了节省成本，可以根据查询工作负载弹性地扩缩集群；然而，这种做法会增加缓存未命中的可能性，因而需要通过网络传输数据 [15]。通过网络获取数据比从本地存储读取更慢。分层存储层 [41]–[43] 或许是更具成本效益、又能保持数据快速访问的解决方案；然而，快速获取数 TB 的数据仍然具有挑战性。

用于加速查询的物化。 数据物化可以通过预先计算开销较大的操作并利用其结果来降低查询延迟。物化视图已在文献中得到广泛研究。相关主题包括高效刷新/更新物化视图 [1]–[3], [44], 、为观测到的查询工作负载推荐有用的物化视图 [33], [45], [46], 、利用物化视图优化查询 [47], [48], 、近似聚合 [49]–[51], 等。虽然草图技术 [52] 和采样 [3] 也可视为物化视图的一种特殊情况，但本文不予讨论。

自动管理物化数据正日益受到关注。Napa [6] 借助物化视图实现高吞吐量数据摄取和快速查询。Keebo [53] 通过挖掘查询工作负载，自动创建物化视图。此外，还有一些工具可帮助用户手动定义数据物化策略；下一节将介绍几种常用工具。

B. 数据物化：占比显著

现实中的数据仓库会花费大量时间物化数据。通过优化物化视图刷新，我们可以降低基础设施成本和数据陈旧程度。由于现有工具提供了模板语言、可视化和基于 Web 的界面，复杂物化视图的管理正变得更加容易。

数据仓库中广泛使用。 数据物化在现代数据仓库工作负载中占据相当大的比例。图 2 展示了十个独立工作负载的统计数据 [35]。根据查询运行时间，数据物化（例如，数据定义语言、数据操纵语言）占每个工作负载的 2%-38%。有趣的是，在 W6 中，数据仓库用于物化的时间是分析所用时间的 2.2× 。

我们还与几家大大小小公司的数据工程师交流过，了解他们的数据摄取和准备流程。他们都是云托管数据仓库的客户。在这些环境中，定期（例如每天）进行数据摄取（使用 Airflow [8], Fivetran [54], 等工具）后，通常还要花数小时在数据仓库中执行重复操作，以创建和更新依赖表。[1] 仪表板会使用这些预计算表来处理面向用户的查询（而不是从基表开始一路计算）；因此，终端用户可以更快地看到结果（例如，在更改筛选条件时）。数据物化主要通过基于 Web 的工具进行管理（而不是手动重写 DDL/DML 语句并直接发送到数据仓库），我们稍后将对此进行介绍。

用于物化的工具。 借助高级管理工具，管理物化视图图变得更加容易。首先，Looker 提供了一种数据建模语言 LookML [9]。借助 LookML，用户可以表达复杂的连接和嵌套结构。基于 LookML，Looker 提供了一项名为 PDT [55] 的功能，用于将视图存储为物化表。构建完成后，Looker 会在生成的查询中使用物化表。其次，dbt [7] 是一种基于嵌套表/视图定义的数据建模工具。由于一个定义可以引用其他定义，整体结构会形成一个有向无环图，其中节点代表表/视图定义，边代表这些定义之间的依赖关系

[1]目前，物化视图并非首选，因为 Amazon Redshift、BigQuery 和 Snowflake 等热门数据仓库不支持包含连接的物化视图。

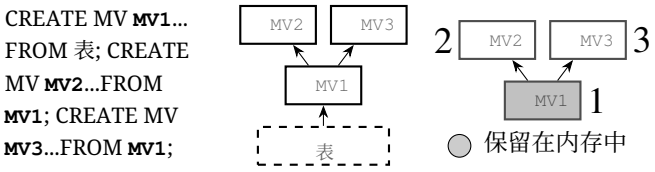


图4：示例工作负载的输入 S/C 。左：工作负载的依赖图。右：优化器执行物化视图刷新运行的计划（执行顺序、保留在内存中的节点）。

。随后，dbt 使用拓扑排序来确定表更新的顺序。需要注意的是，尽管这些工具提供了便捷的方式来构建中间表，但所有操作最终都会编译为 SQL 并发送到连接的数据仓库；数据仓库随后执行实际的计算和数据操作。因此，本文讨论的内容——加速数据仓库的内部操作——与这些工具提供的功能正交。

C. 数据物化的开销

在数据系统中，读取/写入数据往往是一个重要瓶颈。也就是说，数据转换/物化的大部分开销都花在了持久化数据上。

首先，在现代数据仓库中，读写相较于计算往往会产生较大的开销。为研究这一点，我们测量了系统花在读写操作（如序列化、压缩）上的时间，相对于计算操作（如连接、筛选）所占的比例。我们使用一家由供应商管理的商业数据仓库（匿名），运行了包含三个内连接（或等价地，连接四张表）的 CTAS 语句；这些连接出现在 TPC-H 查询 #8 中，涉及客户、订单、明细项和国家。图 3 展示了不同数据集大小下的结果，即规模因子从 1（或 1 GB）到 1000（或 1000 GB）。在 X 轴上，数据集大小旁标注了相应的总运行时间（即 1 GB 数据集为 5.4 秒）。根据这项研究，将连接结果写入持久化存储（可能涉及压缩、序列化和网络 I/O）耗费了总运行时间（每条语句）的 37%–69%。

其次，对于开源数据系统而言，读写也会耗费大量时间。我们比较了 Apache Parquet 的几种不同实现（例如 C++ Arrow [56], Rust Arrow [57]），这是一种在数据仓库中广泛使用的列式数据格式 [12]–[14], [58], [59]。在我们的环境中，使用从源代码编译的 Rust Arrow 可以实现最快的序列化速度。在此设置下，我们测试了为 TPC-DS 查询创建中间表时读写所产生的开销（见 § VI-A）。总体而言，读写耗时占计算操作（如连接、筛选等）总耗时的 85%。

III. S/C：系统概述

S/C 是一种系统，给定一组物化视图（MV）定义后，它会刷新所有物化视图（更准确地说，是为这些物化视图生成数据）。本节概述S/C 如何执行此操作：§ III-A 描述S/C的输入，§ III-B 介绍S/C的子组件及其作用，§ III-C 则概述一次物化视图刷新运行的执行过程。

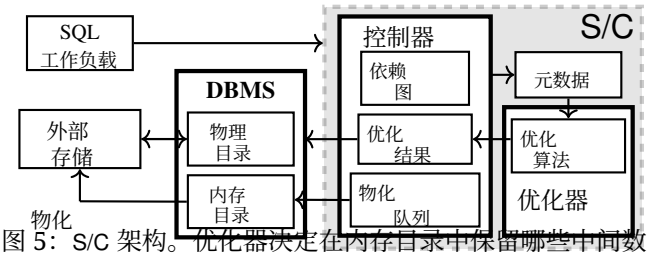


图 5：S/C 架构。优化器决定在内存目录中保留哪些中间数据。数据库管理系统可以直接访问暂时保存在内存目录中的数据进行计算，从而节省从外部存储读取数据所需的时间。

A. 工作负载规范

要更新的物化视图列表（及其依赖关系）通过依赖图正式表示。随后，执行元数据会记录有关这些物化视图的其他信息（例如磁盘占用），以用于优化。

依赖图。需要以依赖图的形式将要刷新的物化视图集合提供给 S/C （图 4，左）。每个节点对应一次单独的物化视图更新，边则对应物化视图之间的依赖关系。每个节点都包含一条用于特定物化视图更新的 SQL 语句。

执行元数据。S/C 的优化需要来自数据库管理系统端过去物化视图刷新运行中 SQL 执行的信息，即 (1) 执行每个节点中的 SQL 语句所生成输出表的估计大小，以及 (2) 将节点输出保留在内存中的预计时间节省（见 § IV）。

B. 内部架构

S/C 的控制器根据优化器生成的计划（更新顺序和内存缓存）协调物化视图更新（见图 5）。

控制器。控制器根据优化器提供的指示（见下文），安排节点（即物化视图）的物化顺序，并决定如何/在哪里存储各节点的输出（即保留在内存中还是物化到外部存储[2]）。这些指示会被编译为相应的 SQL 语句，并发送至数据库管理系统执行。控制器还负责管理保留在内存中的中间表的物化。（§ III-C）

优化器。优化器利用从数据库管理系统中 SQL 执行所收集的元数据以及我们提出的算法 S/COpt （详见 §§ IV 和 V），为控制器计算物化视图刷新顺序（例如，MV1、MV2、MV3）以及要保留在内存中的节点（图 4，右侧）。

数据库管理系统。数据库管理系统执行来自控制器的 SQL 语句（例如连接、聚合）。输入表来自内存目录（稍后介绍），或从外部存储（例如磁盘）读取。我们目前的实现使用 Presto 数据库管理系统集群 [14], ，但也可以用任何其他数据库管理系统替代。

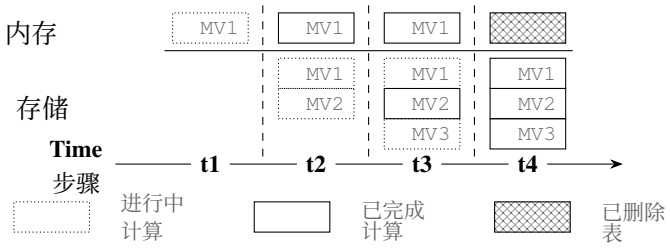


图 6：根据图 4 中给出的工作负载计划，按时间戳（t1-t4）进行物化视图刷新的示例过程。

C. 执行一次物化视图刷新运行

在物化视图刷新运行期间，控制器会根据优化器计算出的执行顺序，逐个指示数据库管理系统执行节点（即运行其 SQL 语句）。

内存管理。S/C 确保保留在内存中的节点（称为内存目录）的大小得到正确限制。设 n 是根据优化器的计划需要保留在内存中的节点：n 直接在内存目录中创建；当所有依赖于 n 的节点都完成执行后，便立即将其从内存目录中释放。

并行化计算与物化。S/C 旨在最大限度地利用数据库管理系统的计算资源和 I/O 带宽。在内存中创建的节点会在创建后立即物化到外部存储；此物化过程与下游节点的执行并行进行。

示例。基于图 4，我们在图 6 中给出一个具体示例，说明 S/C 如何在确保正确使用内存目录的同时协调物化视图刷新，并实现 I/O 与计算的并行化。根据下方时间线，物化视图按 MV1、MV2、MV3 的顺序刷新；只有 MV1 会保留在内存中：

- t1：在内存目录中创建 MV1。
- t2：MV1 已在内存中完全创建。刷新 MV2（使用新创建的 MV1）的同时，MV1 被物化到磁盘；MV2 将直接在磁盘上创建。
- t3：MV2 现已完成刷新。MV3 开始在存储上刷新，并从内存中的 MV1 读取数据。
- t4：MV3 已完全刷新，MV1 已完全物化。内存中的 MV1 已删除，物化视图刷新运行至此结束。

IV. S/COpt：问题设定

本节将正式定义 S/COpt，即在有界内存下通过存储中间数据来加速物化视图刷新运行。表 II 列出了我们在描述 S/COpt 时使用的符号。

问题输入。S/C Opt 有 3 个输入：

1. 表示物化视图刷新运行的有向无环图（DAG）$\mathcal{G}:=\{\mathcal{V},\mathcal{E}\}$。$\mathcal{V}=\{v_1,...,v_n\}$ 是 n 个独立物化视图更新（节点）的集合，$\mathcal{E}$ 是物化视图更新之间依赖关系（边）的集合。2. 每个节点产生的中间表大小集合 $\mathcal{S}=\{s_1,...,s_n\}$，我们将其简称为节点大小。

[2]我们的 S/C 实现使用网络文件系统进行存储。也可以将其替换为其他物化位置。

表 II：符号表

符号	定义
$\mathcal{G}:=\{\mathcal{V},\mathcal{E}\}$	依赖图
$\mathcal{V}=\{v_1,...,v_n\}$	$\lvert\mathcal{V}\rvert=n$ 节点集合
$\mathcal{E}$	$\lvert\mathcal{E}\rvert=m$ 依赖关系集合
$\mathcal{S}=\{s_1,...,s_n\}$	中间表（节点）大小集合
$\mathcal{T}=\{t_1,...,t_n\}$	加速分数集合
M	内存目录大小
$\tau:[1..n]\to[1..n]$	物化视图刷新（执行）顺序
$\mathcal{U}\subseteq\mathcal{V}$	保留在内存中的节点结果（标记节点）

每个 s_i 都会测量保留由 v_i生成的中间表所需的内存空间。

3. 每个节点 $\mathcal{T}=\{t_1,...,t_n\}$ 的加速分数集合，其中 t_i 衡量通过将 v_i 的输出保留在内存中（我们称之为标记该节点）所实现的物化视图刷新运行的估计加速比。

加速评分。节点 n_i 的加速评分 t_i 计算如下：

$$t_i=\Big(\sum_{(v_i,v_j)\in\mathcal{E}}\text{data-access-time}(v_j|v_i\text{ on disk})-\text{data-access-time}(v_j|v_i\text{ in memory})\Big)+\text{time}(\text{create }v_i\text{ on disk})-\text{time}(\text{create }v_i\text{ in memory})$$

标记 v_i 的加速比 t_i 是相对于顺序执行物化视图刷新操作的基准方法计算的：对于每个依赖于 v_i 的下游节点 v_j，通过从内存而非外部存储读取 v_i，可节省数据访问时间。执行 v_i 时，可通过将 v_i 并行物化到外部存储来减少写入时间，同时尽量减少对下游计算的干扰。（§ III-C）

内存使用量与执行顺序。给定内存目录大小 M，物化视图刷新运行期间任意时刻的峰值内存使用量——即同时驻留在内存中的标记节点的总大小——不得超过 M。一组标记节点的峰值内存使用量取决于执行顺序 τ：$[1..n]\to[1..n]$，节点按此顺序执行：v_i 是第 $\tau(i)^{th}$ 个执行的节点。

执行顺序决定标记节点何时可以从内存中释放——只有在其所有下游节点（子节点）都已计算完成后，才能释放该标记节点。因此，在内存目录大小约束下，执行顺序在决定一组标记节点是否可行方面起着关键作用。良好的执行顺序有助于更快地释放标记节点，进而使我们能够在整个物化视图刷新运行期间标记更多节点，从而在相同的内存目录大小约束下获得更高的总加速分数。

考虑图 7 所示的玩具示例。为简单起见，假设节点的加速评分等于其以 GB 为单位的大小；理想情况下，我们希望标记两个 100GB 节点 v_1 和 v_3，其内存目录大小为 $M=100GB$。由于执行完 v_4 后可以释放 v_1，τ_2 允许通过在 v_3之前执行 v_4，将 v_1 和 v_3 都标记起来，而 τ_1 则不允许。在 τ_2 下，通过标记 v_1 和 v_3（以及 v_6），可获得 210 的最高可能评分。这远高于在 τ_1下标记 v_1、v_5和 v_6 所能获得的 120 最高评分。

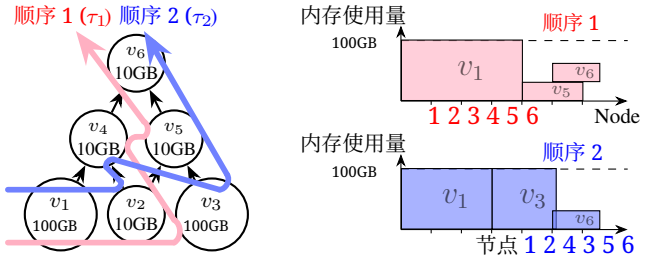


图 7：执行顺序很重要：在 τ_1 下， v_1 和 v_3 不能同时被标记，但在 τ_2 下可以。标记节点的组合大小越大越好。

问题定义。根据我们的问题设定，我们将 S/COpt 定义为对标记节点集合 $\mathcal{U}$ 和执行顺序 τ进行联合优化的问题：

问题 1. S/C Opt

输入：1. 依赖图 $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$2. 节点大小 $\mathcal{S}$3. 加速分数 $\mathcal{T}$ 4. 内存目录大小 M输出：1. 标记节点的子集 $\mathcal{U} \subseteq \mathcal{V}$ 2. 执行顺序 τ目标函数：最大化总加速分数 $\sum_{i:\ v_i \in \mathcal{U}} t_i$标记节点约束：在执行顺序 τ和内存目录大小 M条件下，标记 $\mathcal{U}$ 是可行的

V. S/COpt：联合优化

我们提出了针对S/COpt的解决方案。据我们所知，不同于以往关于一组物化视图的刷新顺序调度 [5], [23] 以及存储选定的中间数据以降低作业延迟 [18], [19], [60],的研究，我们首次考虑对这两项进行联合优化。

S/COpt 可通过将 0-1 背包问题归约至此证明为 NP 困难，因此需要使用高效的近似算法。尽管S/COpt 有两个不同的输出：标记节点 U 和执行顺序 τ，但由于在保持 U 不变时改进 τ 并不能提升总加速分数这一目标函数，因此标准的交替优化 [61] 并不适用。我们仍将S/COpt 分解为两个不同的子问题，并从初始执行顺序 τ_0 和空的标记节点集合（即 $\mathcal{U}_0 = \emptyset$）开始求解S/COpt ；然而，与标准交替优化不同，我们将S/COpt 分解为两个目标函数不同的子问题：

问题 2. S/C Opt 节点

输入： S/COpt 的输入 S/COpt，执行顺序 τ输出： 标记节点 $\mathcal{U}$目标函数： 最大化总加速分数 $\sum_{v_i \in \mathcal{U}} t_i$约束： 与 S/C Opt 相同

问题 3. S/C Opt 顺序

输入：S/COpt 的输入，标记节点 $\mathcal{U}$输出：执行顺序 τ

目标函数： 最小化平均内存使用量$\frac{1}{n}\sum_{i:v_i \in \mathcal{U}}(max_{(v_i,v_j) \in \mathcal{E}}\tau(j) - \tau(i))s_i)$约束： 与 S/COpt 相同

我们不再改进目标函数，而是通过将平均内存使用量用作 S/C Opt 顺序 的目标函数来“放宽约束”。我们将平均内存使用量定义为：假设单位作业执行时间相同，存储在内存目录中的标记节点的平均大小（图 8 中以阴影区域表示）。平均内存使用量越低，表示内存目录的使用效率越高，这是通过更及时地释放标记节点实现的，并可能使我们在交替优化的后续迭代中向$\mathcal{U}$ 中添加更多节点。

尽管峰值内存使用量是 S/COpt的约束，但我们针对 S/C Opt 顺序优化平均内存使用量，因为前者无法有效区分不同的执行顺序。图 8 展示了一个例子：节点v_1、 v_3、 v_4、 v_5 被标记。尽管两种刷新顺序的峰值内存使用量相同，我们针对 S/C Opt 顺序 (MA-DFS，§ V-B) 得出的刷新顺序达到峰值内存使用量的持续时间更短，因此平均内存使用量更低，从而在下一轮求解 S/C Opt 节点时可以额外标记一个节点。

在以下小节中，我们将介绍S/C Opt 节点 （§ V-A）和S/C Opt 顺序 （§ V-B）的解决方案。最后，我们将说明如何将二者结合起来，形成用于求解S/COpt （§ V-C）的交替优化算法。

A. S/C Opt 节点：解决方案

我们将S/C Opt 节点 精确地建模为多维 0-1 背包问题（MKP） [62]并求解。我们也考虑过贪心算法等近似解法；然而，与我们的 MKP 建模相比，这些算法的性能明显较差，而运行时间优势却微乎其微。（§ VI-F）

***MKP* 设置。**对于每个节点 v_i 和执行顺序 τ，令 $\mathcal{V}_i$表示这样一组节点：若将它们标记，则会在 v_i执行期间保留在内存中。根据S/C的内存管理方案，若节点 v_j 至少有一个子节点尚未执行，则被标记的节点 v_j 会在 v_i执行时保留在内存中：

$$\mathcal{V}_i := \{v_j | \tau(j) \leq \tau(i) \leq max_{(v_j,v_k) \in \mathcal{E}}\tau(k)\}$$

所有 $\mathcal{V}_i$构成 MKP 的约束，使得每个 $\mathcal{V}_i$ 中标记节点的总大小不超过内存目录大小 M。将加速分数 $\mathcal{T}$作为 MKP 目标函数的系数，我们将S/C OptNodes 以整数线性规划格式[3] 定义如下：

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} x_i t_i \\ \text{subject to} \quad & \sum_{j:v_j \in \mathcal{V}_i} x_j s_j \leq M, \quad i = 1, \ldots, n \\ & x_i \in \{0, 1\}, \quad i = 1, \ldots, n \end{aligned}$$

[3]我们将加速分数四舍五入到最接近的整数。

算法 1：SimplifiedMKP

输入：(1) 依赖图 $\mathcal{G}=\{\mathcal{V},\mathcal{E}\}$
(2) 节点大小 $\mathcal{S}$ 和加速分数 $\mathcal{T}$
(3) 内存目录大小 M
(4) 执行顺序 τ
输出：标记节点集合 $\mathcal{U}$

1 初始化 $\mathcal{V}_{exclude}=\{v_i|s_i>M\vee t_i=0\}$;
2 约束_集合 = 获取约束$(\mathcal{G},\mathcal{S},\mathcal{V}_{exclude},M,\tau)$;
3 初始化 $\mathcal{V}_{mkp}=\bigcup_{\mathcal{V}_i\in\text{约束_集合}}\mathcal{V}_i$;
4 初始化 $|\text{约束_集合}|=k,|\mathcal{V}_{mkp}|=l$;
5 设置利润 $P_{1\times l}=[t_{i_1},...,t_{i_l}],v_{i_1},...,v_{i_l}\in\mathcal{V}_{mkp}$;
6 设置权重 $W_{k\times l},w_{xy}=$(如果 $v_{i_y}\in\mathcal{V}_x$ 则 s_i 否则 0) $1\le x\le k,1\le y\le l$;
7 设置容量 $C_{k\times 1}=[M,...,M]$;
8 $\mathcal{U}=$ 二进制多重背包求解器(P,W,C);
9 $\mathcal{U}\leftarrow\mathcal{U}\cup(\mathcal{V}-\mathcal{V}_{mkp}-\mathcal{V}_{exclude})$;
10 返回要存储在内存中的节点集合 $\mathcal{U}$。

简化 ***MKP***。我们通过识别并移除 MKP 中冗余的变量（节点）来简化 MKP。如果满足以下任一条件，节点 v_i 就是冗余的，无需在 MKP 中进行评估：

- $s_i>M$：v_i 的大小超过内存目录的大小——单独标记 v_i 会违反约束
- $t_i=0$：存储 v_i 不会对目标函数产生贡献。我们可以将此类节点从约束集中排除 $\mathcal{V}_i$：

$$\mathcal{V}_i:=\{v_j|\tau(j)\le\tau(i)\le max_{(v_j,v_k)\in\mathcal{E}}\tau(k)\wedge v_j\notin\mathcal{V}_{exclude}\}$$
$$\text{where } \mathcal{V}_{exclude}:=\{v_i|i:s_i>M\vee t_i=0\}$$

如果满足以下任一条件，约束集 $\mathcal{V}_i$ 就是冗余的，无需在 MKP 中进行评估：

- $\mathcal{V}_i$ 不是极大的：$\exists\,\mathcal{V}_j s.t.\mathcal{V}_i\subsetneq\mathcal{V}_j$；约束集 $\mathcal{V}_i$是另一个约束集 $\mathcal{V}_j$的真子集
- $\mathcal{V}_i$ 是平凡的：$\sum_{j:\ v_j\in\mathcal{V}_i}s_j\le M$；即使将 $\mathcal{V}_i$ 中的所有节点都标记，约束 $\mathcal{V}_i$ 也不会被违反

求解 ***MKP***。我们在算法 1 中给出了S/C OptNodes 的完整解（MKP）。通过对节点进行线性扫描（第 2 行），使用 GetConstraints 可以在线性时间内高效计算出所有相关（极大且非平凡）的约束集。注意，并非每个未被排除的节点 $v_i\in\mathcal{V}-\mathcal{V}_{exclude}$ 都会出现在约束集中。如果某个节点没有出现在任何约束集中，且也不在 $\mathcal{V}_{exclude}$中，那么可以直接将其标记，因为这样不会违反内存约束；这些节点会被手动添加到 MKP 求解器返回的解中（第 9 行）。

我们选择分支定界法[4] 作为子程序 Bi-nary MKPSolver（第 8 行）。尽管该方法的最坏情况时间复杂度为指数级，但我们通过实验证实，算法 1 通常具有良好的可扩展性（§ VI-H）：其运行时间大致随依赖图中的节点数量线性增长，并且在一个包含 100 个节点的图中，平均只需 0.02 秒即可找到要标记的最优节点集合。

[4]我们的 S/C 实现使用 Google OR-Tools [63]中的分支定界求解器。

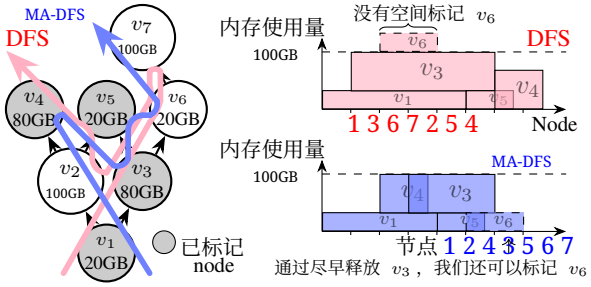


图 8：MA-DFS 背后的直觉。决定先调度 v_2 还是 v_3 时，需要进行平局判定。MA-DFS 会正确地优先调度 v_2，从而允许标记 v_6，进一步提升加速比。采用随机打破平局的 DFS 所产生的执行顺序则无法保证这一点。

B. S/C Opt 顺序：解决方案

我们通过制定一种内存感知的基于深度优先搜索的调度算法来求解 S/C Opt 顺序。我们也尝试过其他方法，例如爬山法 [64] 和递归图分隔器 [65][5]，但发现它们的效果不如该算法。（§ VI-F）

优化平均内存使用量。基于深度优先搜索的调度算法旨在通过完成一个执行分支后再开始新的分支，尽量缩短节点执行与其子节点执行之间的间隔，从而尽早释放标记节点，最大限度地降低其平均内存使用量。

然而，我们发现，现有工作中的现成基于深度优先搜索的排序方法 [67] 并不适用于我们的情况。具体而言，深度优先遍历在分支处必须选择一条路径继续前进（即进行平局判定）；若随机选择时不考虑节点大小，可能导致性能欠佳。随机选择可能会使较大的标记节点在内存中保留较长时间，从而减少可用于标记其他节点的内存空间（见图 8）。

内存感知深度优先搜索。为了解决这一问题，我们提出了一种内存感知的基于深度优先搜索的调度算法（MA-DFS），该算法在平局时优先选择实际内存消耗较低的节点：若节点已标记，其实际内存消耗等于其大小；否则为 0。MA-DFS旨在最后计算节点中体积最大的已标记依赖项，以尽量缩短这些大型依赖项在内存中的保留时间，从而释放内存空间，使更多节点有机会被标记。

图 8 中的玩具示例说明了这一思路。内存目录大小为 $M=100GB$；为简单起见，假设节点的加速评分等于其 GB 大小。当在 v_2 和 v_3 之间进行平局判定时，MA-DFS 会优先调度v_2，因为 v_3 的实际内存消耗更高；尽管 v_3比 v_2 大，v_2 却不会被标记。v_3 仅在内存中保留 3 次节点执行（v_3、v_5、v_6），从而释放内存空间，使v_6 也能够被标记。而采用以下方式则无法做到这一点：

[5]尽管存在针对该问题的整数线性规划表述，[66],但它包含$O(n^3)$个变量和约束，导致其效率过低，无法用于实时调度。我们将这一方向留待未来工作探索。

算法2：交替优化

输入：(1) 依赖图 $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$
(2) 节点大小 $\mathcal{S}$ 和加速分数 $\mathcal{T}$
(3) 内存目录大小 M
输出：(1) 标记节点集合 $\mathcal{U}$
(2) 执行顺序 τ

1 初始化 τ = 获取拓扑序(G);
2 初始化 $\mathcal{U} = \emptyset$;
3 当算法尚未收敛时执行
4 $\mathcal{U}_{new} = \text{SimplifiedMKP}(\mathcal{G}, \mathcal{S}, \mathcal{T}, M, \tau)$;
5 若 $\sum_{i:\ v_i \in \mathcal{U}_{new}} s_i \geq \sum_{i:\ v_i \in \mathcal{U}} s_i$ 则返回 $\mathcal{U}, \tau$;
6 $\mathcal{U} \leftarrow \mathcal{U}_{new}$;
7 $\tau_{new} = \text{MA-DFS}(\mathcal{G}, \mathcal{S}, \mathcal{T}, M, \mathcal{U})$
8 如果 峰值内存使用量$(\mathcal{G}, \mathcal{S}, \mathcal{U}, \tau_{new}) > M$则返回 $\mathcal{U}, \tau$;
9 $\tau \leftarrow \tau_{new}$;
10 结束

在发生平局时，使用随机打破平局的基于深度优先搜索的排序可能会优先调度 v_3 而非 v_2 。这种排序会使 v_3 在内存中保留 5 次节点执行，占用本可用于标记 v_6的内存空间。

我们会在交替优化的每次迭代中验证MA-DFS执行顺序的有效性，以满足内存约束 M。在少数情况下，若 MA-DFS 输出了不可行的执行顺序——违反了内存约束M——我们便认定交替优化上一次迭代得到的执行顺序是最优的，并终止交替优化算法。尽管我们的方法可能找到局部最优解，但我们通过实验证实，与其他方法找到的解相比，我们的方法找到的局部最优解质量仍然很高。（§ VI-F）

C. S/COpt：整合所有内容

在分别求解两个子问题后，我们在算法2中给出了用于求解S/COpt 的交替优化算法。

任何拓扑排序算法都可以用作 Get- Topological Order 子程序，以获得第 1 行的初始执行顺序。[6] 在交替优化的每次迭代中，通过对节点进行线性扫描，可在线性时间内高效计算第 8 行的峰值内存使用量。我们交替求解S/C Opt 节点和S/C Opt 顺序，直到无法再改进结果：

- SimplifiedMKP（第 5 行）未能找到更优的标记节点集合 $\mathcal{U}_{new}$
- MA-DFS 找到的执行顺序 τ_{new} 违反了内存约束 M （第 8 行）

我们的解决方案保证能够收敛，因为交替优化要继续进行， $\mathcal{U}$ 的总加速分数就必须在每次迭代中增加。对于最多包含 100 个节点的依赖图，我们的解决方案通常会在<10 次迭代内收敛。

[6]我们的S/C 实现使用 Python NetworkX 包中的拓扑_排序函数 [68]。

VI. 实验

在本节中，我们通过实证研究.....的有效性 S/C。我们提出以下主张：

1. 加快端到端物化视图刷新：S/C 能够将端到端物化视图刷新时间加快 1.04×–5.08×，相较于未优化的原始 Presto 数据库管理系统；与其他现有的现成方法相比，最多还能额外加快 2.22× 。（§ VI-B）

2. 持续节省执行时间：S/C的优化在不同数据集大小（从10GB到1TB）下均能持续节省时间。（§ VI-C）

3. 有效利用内存：S/C 能够有效利用不同大小的内存目录。S/C的内存目录可分配系统中的闲置内存，也可分配查询内存，且对加速比的损失极小。（§ VI-D） 4. 缩短表读取时间：将中间表存储在内存中，使S/C的表读取延迟降低了 $1.42\times - 1.51\times$ 。（§ VI-E） 5. S/C Opt解决方案的有效性：使用我们的方法（MKP + MA-DFS）求解 S/COpt，与消融方法相比，总时间减少了1.06×–1.23×。（§ VI-F） 6. S/C 在分布式环境中的可扩展性：无论节点数量如何，S/C都能在多节点数据库集群上实现稳定的加速比。（§ VI-G） 7. S/C 在复杂工作负载上的可扩展性：对于最多包含100个节点的复杂有向无环图，我们求解 S/COpt 的优化时间可以忽略不计 $< 0.02s$ 。（§ VI-H）

A. 实验设置

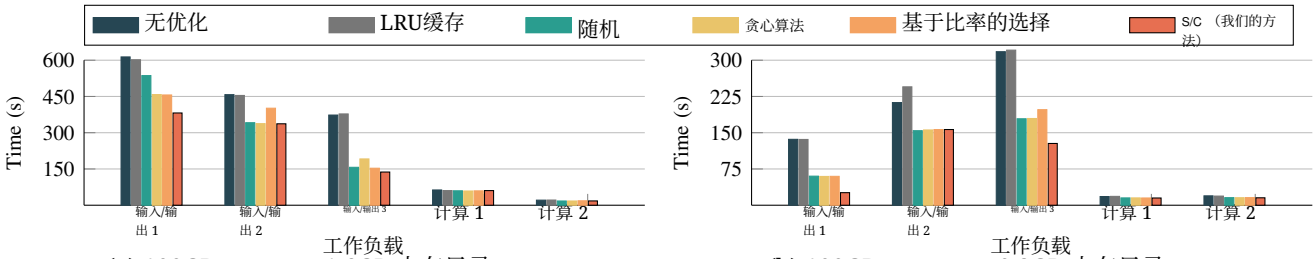
数据集。我们的实验使用 TPC-DS [69] 决策支持基准测试中包含的表生成器和查询。我们根据 5 个不同的规模因子（10、25、50、100、1000）生成 TPC-DS 数据集；规模因子决定了生成的数据集中各表的总大小（以 GB 为单位）。

对于每个规模，我们为每个数据集创建两个副本。一个是按原样生成的普通数据集（即 TPC-DS）。另一个是按日期分区的数据集，其中最大的三张表（store_sales、catalog_sales、web_sales）基于与 date_dim 表的连接，按年份进行分区（即 TPC-DSp）。

方法。我们通过将 S/COpt 框架与现成算法和此前提出的算法进行比较，评估其有效性：

- G贪心算法：按执行顺序遍历节点，如果标记某个节点不会违反内存约束，则将其标记。
- 随机：按随机顺序遍历节点，并在不会违反内存约束的情况下标记节点。
- 基于比率的选择 [60]：一种启发式方法，优先标记加速评分/节点大小比率较高的节点。
- 数据库管理系统中的 LRU 缓存会缓存查询结果。我们将 LRU 缓存的大小增加相当于内存目录大小的容量。

此外，我们还进行了消融分析，以评估我们的 S/C 解决方案的有效性。具体而言，我们评估了针对子问题 S/C OptNodes 和



(a) 100GB TPC-DS，1.6GB 内存目录　　(b) 100GB TPC-DSp，0.8GB 内存目录

图 9：100GB 数据集上的端到端物化视图刷新时间。与未优化的原始引擎相比，S/C 使用 1.6/0.8GB 内存目录可实现 1.04x–5.08x 加速比。与其他现成方法相比，额外加速比最高可达 2.22x。

表 III：工作负载汇总。输入/输出百分比根据使用 Python Polars 对等效操作进行性能分析后估算 [28]。

工作负载	TPC-DS 查询	# 节点	输入/输出比
输入/输出 1	5, 77, 80	21	51.5%
输入/输出 2	2, 59, 74, 75	19	59.0%
输入/输出 3	44, 49	26	46.6%
计算 1	33, 56, 60, 61	21	0.9%
计算 2	14, 23	16	28.3%

S/C OptOrder 的单独解决方案，方法是在交替优化过程中将其中一个子问题的解决方案替换为基线方法。

我们将自己的MKP 方案与S/C Opt 节点 进行比较，并与 Greedy 和 随机 方法进行比较。我们将自己的MA-DFS方案与S/C Opt 顺序 进行比较，并与两种基线方法进行比较：

- 模拟退火（模拟退火） [64]：一种爬山算法，通过迭代不断改进执行顺序。每次迭代中，随机选择两个可交换节点（即交换后不会违反依赖关系的节点）；如果交换能降低平均内存使用量，则执行交换。为避免陷入可能的局部最小值，也会以一定概率执行交换。我们将迭代次数设为 10,000。
- 分隔器 [70], [71]：一种分治算法，通过递归地在有向无环图中查找分隔符/切分点来划分节点。每次迭代中，将一个子图通过切分划分为两部分；当切分将原始有向无环图划分为一系列单节点时，算法停止。这些切分点定义了执行顺序。该算法还特别提供了一个下界 $O(n\log(n))$。

真实工作负载。我们根据 TPC-DS 查询构建物化视图刷新工作负载。对于查询中的每个选择-投影-连接（SPJ）单元，我们都创建一个节点/物化视图，并将具有相似中间节点和主题（即利润报告、销售分析）的 TPC-DS 查询图合并为一个更大的图。每个图表示一组在同一工作负载中一起刷新的物化视图。最终，我们得到五个包含依赖图的工作负载，每个依赖图包含 16–26 个节点（表 III）。

生成的工作负载。为研究我们的解决方案在更复杂工作负载上的可扩展性，我们创建了一个工作负载生成器，用于系统地生成包含更多节点（25–100 个）的真实依赖图。

我们的工作负载生成器由两个组件构成：

- 用于确定工作负载中各个作业之间依赖关系的有向无环图生成器 [72]。我们将生成参数设置为基于对真实世界查询结构（TPC-DS 和 Spider [73]）的实证分析。
- 一个在同一查询集上训练的马尔可夫链，用于确定节点操作（即 JOIN、聚合）。这些操作用于根据节点的输入推导其大小和加速分数。没有父节点（即直接读取基表）的节点，其大小从 100GBTPC-DS 数据集中的表大小随机采样。

环境。所有实验均在一台 Ubuntu 服务器上进行，该服务器配备 AMD EPYC 7552 48 核处理器和 1TB 内存。磁盘读取和写入速度分别为 519.8 MB/s 和 358.9 MB/s，读取延迟为 175μs。在大多数实验中，Presto 集群由 1 个工作节点组成，对于<100GB/1TB 数据集，查询内存分别为 50GB/300GB；不过，我们也在 § VI-G 中研究了分布式环境。S/C 的内存目录不受查询内存影响，除非另有说明。

实现。我们将S/C 实现为运行在 Presto 数据库管理系统之上的 Python 前端。该前端执行优化，并通过 Presto Python 连接器 [74] 向数据库管理系统发送查询；Presto 数据库管理系统构建在 Hive 元存储 [12], 之上，后者将创建的表存储在网络文件系统中。

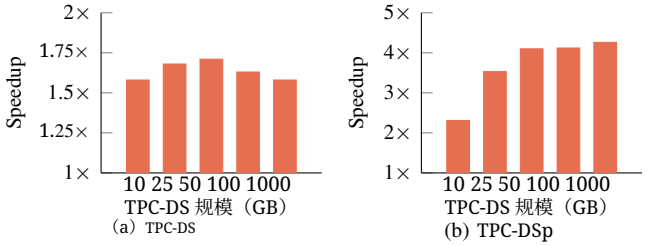
我们通过在 Hive 目录 [75] 中创建表以实现物化，或在内存目录 [76] 中创建表以将数据保留在内存中，从而控制数据库管理系统中的数据放置。基础 TPC-DS 表以 Apache Hive 原生的 ORC 格式 [77] 存储。我们以 Apache Parquet 格式创建物化视图/中间表。

时间测量。我们将端到端时间定义为从启动物化视图刷新工作负载，到该工作负载中的所有物化视图都在我们的网络文件系统上完成物化所需的时间。对于每种设置，我们测量 5 次运行的平均端到端运行时间，以减少方差，并报告中位数。

求解 S/COpt 的运行时间不计入端到端时间，因为：(i) 优化可以在数据摄取之前进行；(ii) 与端到端物化视图刷新时间相比，求解运行时间短得可以忽略不计。

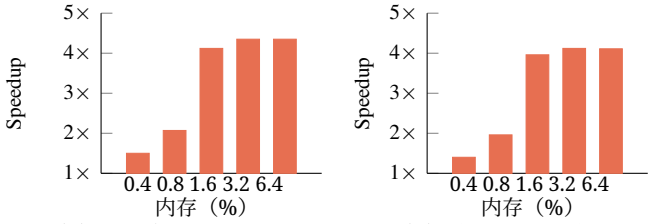
B. 更快的端到端物化视图刷新

在本节中，我们研究 S/C 的优化对端到端物化视图刷新时间的影响。为此，我们比较使用 Presto



（a）TPC-DS (b) TPC-DSp

图10：S/C 的优化在不同数据集规模下均实现了稳定的加速比。内存目录设置为数据大小的 1.6%。



（a）来自闲置内存 （b）来自查询内存

图11：S/C 在 100GB TPC-DSp 数据集上，使用更大的内存目录可实现更多节省。即使内存目录相对于数据集大小仅占很小比例（0.4%），也能显著节省资源。S/C的内存目录还可以从查询内存中重新分配，而对加速比的影响极小。

数据库管理系统，在 100GB TPC-DS 数据集上启用 S/C、不启用 S/C 以及使用其他算法时，所构建的 TPC-DS 工作负载的平均运行时间。

我们在图9中报告了结果。S/C 的设计目标是降低中间表读写所需的开销，这一点从其在 I/O 密集型工作负载中实现的加速比可见一斑。与以串行方式执行工作负载且不将任何数据保留在内存中相比，S/C 使用 1.6GB 内存目录，可将端到端工作负载执行时间加速 1.05×–2.72×。在中间表较小的数据集（100GB 按日期分区的 TPC-DS 数据集）上，节省幅度最高可增至 1.20×–5.08×；S/C 充分利用这一点，在整个工作负载执行过程中将更多数据保留在内存中。

与其他算法相比，S/C 可将端到端工作负载执行时间最多加速 2.22× 倍。具体而言，S/C 优于其他方法的原因在于：(i) S/C 还优化了工作负载的执行顺序；(ii) 贪心/启发式选择在某些工作负载上的表现可能任意糟糕。

C. S/C 处理大型工作负载

本节评估 S/C 在不同数据集大小下带来的性能提升。我们测量了 S/C 在不同大小（10GB–1TB）的常规和按日期分区的 TPC-DS 数据集上，对 5 个工作负载的总执行时间所实现的加速比；内存目录的大小为数据集大小的 1.6%。

图10报告了S/C 在不同 TPC-DS 数据集规模下实现的端到端执行时间缩短情况。若内存目录的大小与数据集规模相匹配，S/C的优化能在不同数据集规模下持续缩短执行时间：常规 TPC-DS 数据集缩短 1.58×–1.71×，按日期分区的 TPC-DS 数据集缩短 2.31×–4.26×。

表 IV：S/C 在 100GB 数据集上对各项指标的优化效果。S/C 通过将中间表存储在内存目录中，缩短了表读取时间。

数据集	延迟（CPU，秒）	无优化	0.4%	0.8%	1.6%	3.2%	6.4%
TPC-DS	表读取	4243	4308	3934	3574	3128	2884
	计算	8533	8587	8319	8283	8249	8286
	查询	12776	12895	12253	11857	11377	11170
TPC-DSp	表读取	1710	1514	1314	1106	1106	1096
	计算	2843	2756	2709	2657	2636	2644
	查询	4553	4270	4023	3763	3742	3740

D. 高效利用内存目录

本节通过评估S/C在不同内存目录大小下的性能提升，研究其对内存目录的使用情况。我们在 100GBTPC-DSp 数据集上，测量 5 个工作负载的总表读取延迟和总执行时间的加速比，并对内存目录大小进行参数扫描，范围为数据大小的 0.4% 至 6.4%。此外，我们还比较了将系统中的闲置内存用作S/C的内存目录，与从数据库管理系统中重新分配查询内存用作内存目录所实现的加速比。结果见图 11。

将中间数据存储在内存中可带来显著加速比：即使内存目录仅占数据大小的 0.4%，加速比也达到 1.50× 。进一步增大内存目录，可使加速比最高达到 4.26× ，此时内存目录占数据大小的 6.4%。

如果系统中没有可用的闲置内存作为 S/C 的内存目录，也可以从数据库管理系统的查询内存中重新分配内存目录，而对已实现的加速比影响甚微。与使用闲置内存相比，使用重新分配的内存导致的加速比下降可以忽略不计，最多仅为 0.25×。

E. S/C 缩短表读取时间

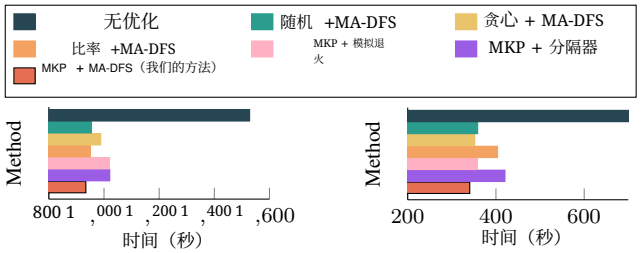
在本节中，我们考察S/C在查询层面的收益。我们在100GB数据集上运行5个工作负载，使用不同大小的内存目录，并展示 Presto 报告的 CPU 指标。

表 IV 按内存目录大小细分了节省的开销。表读取延迟的降低是S/C加速比的主要来源；当数据大小为6.4%时，加速比分别达到1.51× 和1.42×。S/C的优化对计算延迟几乎没有影响，这是预期之中的，因为这并非S/C优化的目标。

F. 我们的优化找到更高质量的解

在本节中，我们考察所提出的MKP + MA-DFS方案对S/COpt的有效性。我们比较了在相同内存目录大小下，使用 MKP + MA-DFS 求解S/COpt 所节省的时间，与消融方法——将MKP 或MA-DFS 之一与另一种方法配对——在100GB TPC-DS数据集上运行5个工作负载的总执行时间节省情况。

图 12 展示了在各种数据集上的这一比较。我们针对两个子问题提出的解决方案——针对 MKP的S/C Opt 节点和针对 MA-DFS 的S/C OptOrder——均优于各自的替代方法。将MKP 与MA-DFS结合使用，相较于Greedy，



(a) TPC-DS（16% 内存目录）(b) TPC-DS（8% 内存目录）

图12：在 100GB 数据集上，求解S/COpt 的不同方法对比。我们的方法（MKP + MA-DFS）优于消融方法，最多可额外节省 3%–11% 的工作负载执行时间。

将随机 或基于比率的选择 与MA-DFS结合使用；类似地，将MA-DFS 与 MKP 结合使用，相较于将模拟退火 或分隔器 与MKP结合使用，最高可实现 1.21× 的加速比。

值得注意的是，我们提出的各个方案克服了其他方法存在的缺陷：

- 贪心、比率和随机 标记节点时，没有考虑节点在内存中保留的时长。
- 模拟退火 和分隔器 与内存目录大小约束的兼容性较差：（i）该约束限制了模拟退火 可执行的有效节点交换次数；（ii）它无法融入分隔器，通常会导致执行顺序不可行，从而提前终止交替优化。

G. S/C 在基于集群的环境中的扩展能力

在本节中，我们展示S/C在分布式环境中实现加速的能力。我们改变 Presto 集群中的工作节点数量（每个节点的查询内存为 50GB），并研究节点数量与 100GB TPC-DS 数据集上 5 个工作负载的端到端总时间加速比之间的关系。

我们在表 V 中展示结果。尽管分布式数据库集群中的端到端总作业执行时间随着节点增加而显著缩短，但无论节点数量如何，S/C优化带来的额外节省量基本保持一致。

H. S/C处理复杂工作负载结构的能力

在本节中，我们研究工作负载依赖结构对S/C优化时间和可实现加速比的影响。我们使用工作负载生成器创建最多包含 100 个节点的有向无环图/工作负载，并（1）将S/C的优化时间与基线方法进行比较；（2）进行参数扫描，以研究生成参数与S/C估计节省量之间的关系。每种设置下我们生成 1000 个有向无环图。

优化时间。图13展示了在最多包含100个节点的有向无环图上，各方法的平均优化时间。我们提出的解决方案MKP + MA-DFS的优化时间随有向无环图中的节点数量线性增长，在包含100个节点的有向无环图上，运行时间仅为0.02秒，几乎可以忽略不计。虽然贪心 + MA-DFS、随机 + MA-DFS和基于比率的选择 + MA-DFS 的优化时间更短，但与额外带来的端到端

表 V：S/C 在数据库集群中的优化对含 1.6% 内存目录的 100GB TPC-DS 数据集的影响。

指标	1 个节点	2 个节点	3 个节点	4 个节点	5 个节点
无优化运行时间（秒）	1528	868	656	546	487
S/C运行时间（秒）	934	521	383	333	304
加速比	1.63×	1.67×	1.71×	1.64×	1.60×

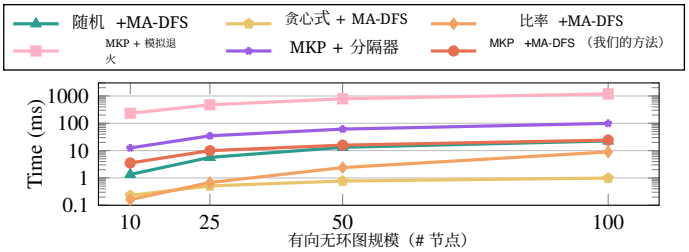


图13：比较不同方法求解合成有向无环图上的S/COpt 的运行时间。我们的方法（MKP + MA-DFS）的运行时间随有向无环图规模线性增长，且在节点数不超过 100 时可忽略不计。

工作负载执行时间缩短相比（§ VI-F），这点优势（在100个节点的图上分别为0.001秒/0.022秒/0.008秒，而对照值为0.024秒）微不足道。就优化时间而言，MKP + 模拟退火 和MKP + 分隔器 都明显慢于MKP + MA-DFS，而且在缩短工作负载执行时间方面带来的收益也更小。

参数扫描。图14报告了S/C在各种生成参数下的优化结果。时间节省以图13中标黑的参数为基准进行归一化。

- 有向无环图规模：S/C 实现的归一化节省量与有向无环图规模高度相关，但二者并非成正比。不过，这是意料之中的，因为经过重复筛选操作/投影操作后，深度嵌套的物化视图往往规模较小。

- 有向无环图高度/宽度：工作负载生成器会按照 Spark 工作负载的结构生成有向无环图：高度和宽度分别对应工作负载中的阶段数和每个阶段的节点数。由于阶段间依赖较少，节点可以按照高效的执行顺序更早地从内存目录中释放，因此，S/C 在高度/宽度比更高、结构更“细”的有向无环图上能实现更多节省。

- 节点最大出度：此参数控制有向无环图中每个节点生成的出边数量，该数量从 [0, 最大出度] 范围内均匀采样。数值越高，节省越多；这是因为单个节点的加速评分更高，其标记可降低更多下游节点的文件读取延迟。

- 阶段节点数标准差（StDev）：此参数会使有向无环图中每个阶段的节点数产生变化。随着方差增大，有向无环图结构会变得越来越不规则，但这种不规则性对加速比的影响可以忽略不计。

VII. 相关工作

高效刷新物化视图。增量视图维护 (IVM) 旨在更新物化视图 (MV)，使其反映新摄入的数据，并利用已计算的结果，以更高效的方式完成更新

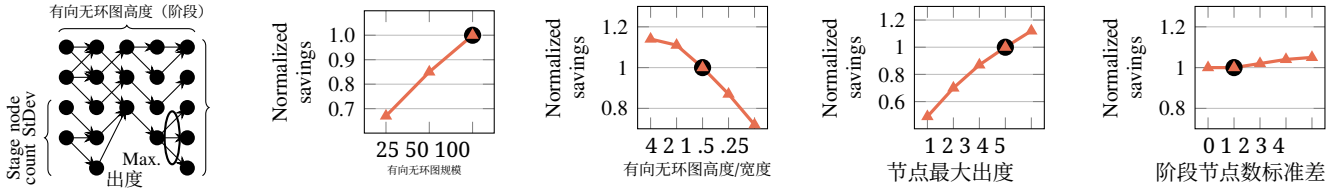


图14：合成有向无环图上 DAG 复杂度与 S/C 预测节省量的分析。节省量以具有 100 个节点、高度/宽度比为 1、最大出度为 4 、阶段节点数标准差为 1 的有向无环图为基准进行归一化（图 13 中以黑色标记的参数）。

而不是从头计算（完全刷新） [1]–[4], [6], [21], [22], [44]。增量视图维护领域已有大量研究，包括针对重复值的增量更新 [21], 、非分配性聚合函数 [22], 、高阶视图 [1], 和滑动窗口 [4]。近期研究也探讨了增量视图维护的可扩展性，并提出了与规模无关的更新 [44] 和采样视图 [3]。由于S/C 适用于任意 SQL 语句，因此S/C 与现有增量视图维护技术正交且完全兼容。

物化视图刷新调度。已有一些研究致力于物化视图集合的刷新调度，重点是解决循环依赖 [23], 、最小化加权平均陈旧度 [5],，以及优先处理在预测的未来查询中具有最高加速比的物化视图 [24]。现有研究尚未涉及通过 S/C 的调度来加速物化视图集合的端到端刷新。

有向无环图工作流调度。由具有无环依赖关系的单个作业组成的工作负载的执行是一个经过广泛研究的主题 [34], [66], [67], [78], [79]；本文研究的许多技术都可应用于物化视图刷新运行。现有的工作流调度系统（如 Apache Oozie [34], 、Apache Airflow [8], 和 Spark DAG 调度器 [78] ）会按照拓扑序自动执行用户定义的工作流。近期研究致力于寻找在峰值内存使用量 [67], [79] 和并行平台上的执行时间 [66] 方面更优的执行顺序。尽管 S/C 旨在用于物化视图刷新运行/工作负载，但作为未来可能的研究方向，我们的联合调度与优化技术也可合理地应用于一般工作负载。

中间数据缓存。一些现有的数据可视化系统会缓存用户定义的变量，以支持数据分析会话期间 [82]–[84] 数据可视化的典型增量构建 [80], [81] 。近期研究提出了一种在内存约束下管理这些缓存变量的方案：它根据神经网络预测的用户未来行为，估算各变量可节省的时间，并以贪心算法保留预计节省时间最多的变量 [60]。尽管这种方法对数据可视化很有用，但与 S/C 相比，采用贪心算法进行内存管理无法取得令人满意的结果。

中间结果复用。已有研究探讨如何存储计算产生的中间结果，以加速后续计算 [25]–[27], [85], [86]。研究主题包括通过查找连续作业的计算图之间的重叠部分来识别复用机会 [25], [27], ，以及在空间约束下，采用复用概率 [26], 、预期节省 [25], 和重新计算与存储的成本差 [85], 等启发式方法进行选择性存储，以及改写传入作业以利用已存储的中间结果 [86]。这些研究与 S/C 在内存约束下选择要存储的项目方面有相似之处，但S/C的问题设定要求其独特地同时考虑作业执行的联合（重新）排序与项目选择。

增量查询处理。当查询所需的数据尚未全部到齐时，增量处理（IQP）非常有用。与在线聚合类似， [87],查询会先基于所需数据的一个子集计算初步结果，并随着其余所需数据按照可预测的模式到达而逐步完善结果 [17], [18]。Tang 等人提出了一种动态规划方案，可在内存预算有限的情况下选择要存储在内存中的中间状态 [17]。Tempura 根据预测的数据到达模式重写查询计划，以实现更高效的执行 [18]。尽管 IQP 与 S/C 的问题设定存在相似之处，例如对有界内存的管理，S/C 还特别包含对物化视图更新顺序的联合优化。

VIII. 结论与未来工作

按照拓扑序依次刷新物化视图（MV），会产生显著开销，这是因为大量中间 I/O 用于读取和写入中间数据。本文提出了一种新系统 S/C，旨在通过在工作负载执行期间有选择地将中间数据持久化于有界内存中，减少阻塞式 I/O 开销，从而显著缩短从外部存储读取中间数据以及向外部存储写入中间数据所需的等待时间。我们提出了一种高效且可扩展的算法，联合优化（1）要持久化的中间数据子集和（2）拓扑序，以便在有界内存条件下最小化端到端物化视图刷新时间。S/C 与现有物化视图刷新技术不同且互为补充，因为我们的方法不受单个物化视图刷新操作的影响。我们证明，在 100GB 数据集上使用 1.6GB 内存时，S/C 可将端到端物化视图刷新工作负载执行时间缩短 1.04×–5.08×。此外，我们的优化算法能够轻松处理包含 100 个物化视图的复杂工作负载。

未来，我们计划将本文介绍的技术推广到非物化视图刷新类的周期性工作负载中，这类工作负载包含具有无环依赖关系的单个作业。

致谢

本研究部分由 Microsoft Azure 资助。

参考文献

[1] Y. Ahmad、O. Kennedy、C. Koch 和 M. Nikolic，《Dbtoaster：面向动态且频繁刷新的视图的高阶增量处理》，VLDB Endowment 论文集，第 5 卷，第 10 期，2012 年。[2] F. McSherry、D. G. Murray、R. Isaacs 和 M. Isard，《差分数据流》，发表于 CIDR，2013 年。[3] K. Zeng、S. Agarwal 和 I. Stoica，《Iolap：通过管理不确定性实现高效增量 OLAP》，载于《2016 年数据管理国际会议论文集》，2016 年，第 1347–1361 页。[4] L. Braun、T. Etter、G. Gasparis、M. Kaufmann、D. Kossmann、D. Widmer、A. Avitzur、A. Iliopoulos、E. Levy 和 N. Liang，《动态分析：在同一数据库中实现高性能事件处理与实时分析》，载于《2015 年 ACM SIGMOD 数据管理国际会议论文集》，2015 年，第 251–264 页。[5] L. Golab、T. Johnson 和 V. Shkapenyuk，《实时流数据仓库中的更新调度》，载于《2009 年 IEEE 第 25 届国际数据工程会议》，IEEE，2009 年，第 1207–1210 页。[6] A. Agiwal、K. Lai、G. N. B. Manoharan、I. Roy、J. Sankaranarayanan、H. Zhang、T. Zou、M. Chen、Z. Chen、M. Dai 等，《Napa：在谷歌通过稳健的查询性能支持可扩展数据仓储》，VLDB Endowment 论文集，第 14 卷，2997，2021 年 12，2986，第 – 页。[7] 《dbt——转换数据仓库中的数据》，www.getdbt.com。[8] 《Apache Airflow》，airflow.apache.org。[9] 《什么是 LookML？》 https://cloud.google.com/looker/docs/what-is-lookml。[10] J. Krueger、C. Kim、M. Grund、N. Satish、D. Schwalb、J. Chhugani、H. Plattner、P. Dubey 和 A. Zeier，《使用多核 CPU 快速更新读优化数据库》，arXiv 预印本 arXiv:1109.6885，2011 年。[11] 《PostgreSQL：全球最先进的开源关系型数据库》，www.postgresql.org。[12] A. Thusoo、J. S. Sarma、N. Jain、Z. Shao、P. Chakka、S. Anthony、H. Liu、P. Wyckoff 和 R. Murthy，《Hive：基于 MapReduce 框架的数据仓储解决方案》，VLDB Endowment 论文集，第 2 卷，第 2 期，第 1626–1629 页，2009 年。[13] M. Armbrust、R. S. Xin、C. Lian、Y. Huai、D. Liu、J. K. Bradley、X. Meng、T. Kaftan、M. J. Franklin、A. Ghodsi 等，《Spark SQL：在 Spark 中进行关系型数据处理》，载于《2015 年 ACM SIGMOD 数据管理国际会议论文集》，2015 年，第 1383–1394 页。[14] 《面向大数据的分布式 SQL 查询引擎》，prestodb.io，发布者 = （访问日期：2022年12月15日）。[15] B. Dageville、T. Cruanes、M. Zukowski、V. Antonov、A. Avanes、J. Bock、J. Claybaugh、D. Engovatov、M. Hentschel、J. Huang 等，《Snowflake 弹性数据仓库》，载于《2016 年 ACM SIGMOD 数据管理国际会议论文集》，2016 年，第 215–226 页。[16] 《 Azure SQL数据库》，azure.microsoft.com/en-us/products/azure-sql/database。[17] D. Tang、Z. Shang、A. J. Elmore、S. Krishnan 和 M. J. Franklin，《间歇式查询处理》，VLDB Endowment 论文集，第 12 卷，第 11 期，第 1427–1441 页，2019 年。[18] Z. Wang、K. Zeng、B. Huang、W. Chen、X. Cui、B. Wang、J. Liu、L. Fan、D. Qu、Z. Hou 等，《Tempura：面向增量数据处理的通用基于成本的优化器框架》，VLDB Endowment 论文集，第 14 卷，第 1 期，第 14–27 页，2020 年。[19] V. Jalaparti、C. Douglas、M. Ghosh、A. Agrawal、A. Floratou、S. Kandula、I. Menache、J. S. Naor 和 S. Rao，《Netco：面向解耦式存储分析的缓存与输入/输出管理》，载于《ACM 云计算研讨会论文集》，2018 年，第 186–198 页。[20] J. Yang、R. Karimi、T. Sæmundsson、A. Wildani 和 Y. Vigfusson，《 Mithril：挖掘稀疏关联以实现缓存预取》，载于《2017 年云计算研讨会论文集》，2017 年，第 66–79 页。[21] T. Griffin 和 L. Libkin，《含重复项视图的增量维护》，载于《1995 年 ACM SIGMOD 数据管理国际会议论文集》，1995 年，第 328–339 页。[22] T. Palpanas、R. Sidle、R. Cochrane 和 H. Pirahesh，《非分配性聚合函数的增量维护》，载于《 VLDB'02：第 28 届超大型数据库国际会议论文集》，Elsevier，2002 年，第 802–813 页。[23] N. Folkert、A. Gupta、A. Witkowski、S. Subramanian、S. Bellamkonda、S. Shankar、T. Bozkaya 和 L. Sheng，《优化一组物化视图的刷新》，载于 VLDB，2005 年，第 1043–1054 页。[24] K. Ahuja，《物化视图的 Oracle 特色》，VLDB Endowment 论文集，第 13 卷，第 12 期，第 3046–3058 页，2020 年。[25] Z. Yang、D. Jia、S. Ioannidis、N. Mi 和 B. Sheng，“多阶段并行大数据框架的中间数据缓存优化”，载于 2018 年 IEEE 第 11 届云计算国际会议（CLOUD）。IEEE，2018 年，第 277–284 页。[26] K. Dursun、C. Binnig、U. Cetintemel 和 T. Kraska，“重新审视主存数据库系统中的重用”，载于 2017 年 ACM 数据管理国际会议论文集，2017 年，第 1275–1289 页。[27] P. Michiardi、D. Carra 和 S. Migliorini，“面向数据密集型可扩展计算工作负载的多查询优化内存缓存”，载于 EDBT/ICDT 研讨会，2019 年。[28] “适用于 Rust 和 Python 的闪电般快速数据框库”，www.pola.rs，出版方 =（访问日期：2022年12月15日）。[29] “ NetworkX 拓扑排序”，networkx.org/documentation。[30] N. Sheoran、S. Chockchowwat、A. Chheda、S. Wang、R. Verma 和 Y. Park，“迈向深度在线聚合”，载于 2023 年 ACM SIGMOD 数据管理国际会议论文集，2023 年。[31] S. Chockchowwat、C. Sood 和 Y. Park，“Airphant：面向云的文档索引”，载于 2022 年 IEEE 第 38 届数据工程国际会议（ICDE）。IEEE，2022 年，第 1368–1381 页。[32] S. Chockchowwat、W. Liu 和 Y. Park，“自动寻找最优索引结构”，载于 VLDB 的 AIDB 研讨会，2022 年。[33] Y. Park、A. S. Tajik、M. Cafarella 和 B. Mozafari，“数据库学习：迈向一个每次使用都会变得更智能的数据库”，载于 2017 年 ACM 数据管理国际会议论文集，2017 年，第 587–602 页。[34] “Apache Oozie”，oozie.apache.org。[35] A. Aleyasen，“克服数据仓库重新平台化的障碍”，博士论文，伊利诺伊大学厄巴纳-香槟分校，2022 年。[36] A. Aleyasen、M. Morcos、L. Antova、M. Sugiyama、D. Korablev、J. Patvarczki、R. Mutreja、M. Duller、F. M. Waas 和 M. Winslett，“用于数据库重新平台化的智能自动化工作负载分析”，载于 2022 年数据管理国际会议论文集，2022 年，第 2273–2285 页。[37] A. Vogelsgesang、M. Haubenschild、J. Finis、A. Kemper、V. Leis、T. M¨uhlbauer、T. Neumann 和 M. Then，“面对现实：基准测试为何无法代表真实世界”，载于数据库系统测试研讨会论文集，2018 年，第 1–6 页。[38] K. Shvachko、H. Kuang、S. Radia 和 R. Chansler，“Hadoop 分布式文件系统”，载于 2010 年 IEEE 第 26 届大容量存储系统与技术研讨会（MSST）。Ieee，2010 年，第 1–10 页。[39] H. Li、A. Ghodsi、M. Zaharia、S. Shenker 和 I. Stoica，“Tachyon：面向集群计算框架的可靠高速存储”，载于 ACM 云计算研讨会论文集，2014 年，第 1–15 页。[40] A. Gupta、D. Agarwal、D. Tan、J. Kulesza、R. Pathak、S. Stefani 和 V. Srinivasan，“Amazon Redshift 及其对更简单数据仓库的倡议”，载于 2015 年 ACM SIGMOD 数据管理国际会议论文集，2015 年，第 1917–1923 页。[41] C. Wu、V. Sreekanti 和 J. M. Hellerstein，“Anna 中分层云存储的自动扩缩”，VLDB Endowment 论文集，第 12 卷，第 6 期，第 624–638 页，2019 年。[42] F. Li，“阿里巴巴的云原生数据库系统：机遇与挑战”，VLDB Endowment 论文集，第 12 卷，第 12 期，第 2263–2272 页，2019 年。[43] H. Herodotou 和 E. Kakoulli，“ Trident：大数据平台中分层存储系统上的任务调度”，VLDB Endowment 论文集，2021 年。[44] M. Armbrust、E. Liang、T. Kraska、A. Fox、M. J. Franklin 和 D. A. Patterson，“通过增量预计算实现广义规模独立性”，载于 2013 年 ACM SIGMOD 数据管理国际会议论文集，2013 年，第 625–636 页。[45] A. Shukla、P. Deshpande、J. F. Naughton 等，“多维数据集的物化视图选择”，载于 VLDB，第 98 卷，1998 年，第 488–499 页。[46] S. Agrawal、S. Chaudhuri 和 V. R. Narasayya，“SQL 数据库中物化视图和索引的自动选择”，载于 VLDB，第 2000 卷，2000 年，第 496–505 页。[47] S. Chaudhuri、R. Krishnamurthy、S. Potamianos 和 K. Shim，“使用物化视图优化查询”，载于第十一届

国际数据工程会议。IEEE，1995 年，第 190–200 页。[48] D. Calvanese、G. De Giacomo、M. Lenzerini 和 R. Rosati，“描述逻辑中的基于视图的查询回答：语义与复杂性”，《计算机与系统科学杂志》，第 78 卷，第 46 期，2012 年，第 1–26 页。[49] S. Heule、M. Nunkesser 和 A. Hall，“实践中的 HyperLogLog：先进基数估计算法的工程实现”，载于第 16 届国际数据库技术拓展会议论文集，2013 年，第 683–692 页。[50] P. Flajolet、 ´E. Fusy、O. Gandouet 和 F. Meunier，“HyperLogLog：近乎最优基数估计算法的分析”，载于《离散数学与理论计算机科学》。离散数学与理论计算机科学，2007 年，第 137–156 页。[51] K. Rong、Y. Lu、P. Bailis、S. Kandula 和 P. Levis，“利用汇总统计量为大数据工作负载进行近似分区选择”，VLDB Endowment 论文集，第 13 卷，第 12 期，2020 年，第 2606–2619 页。[52] “BigQuery 文档：物化视图简介”，cloud.google.com/bigquery/docs/materialized-views-intro。[53] “Keebo：数据学习与仪表板加速”，keebo.ai。[54]“Fivetran：自动化、可靠且安全的数据管道”，www.fivetran.com。[55] “Looker 中的派生表”，https://cloud.google.com/looker/docs/what-is-look

[[56] “C++ 实现，Apache Arrow”，arrow.apache.org/docs/cpp/index.html。[57] “Arrow - Rust”，docs.rs/arrow/latest/arrow。[58] “Amazon Athena”，aws.amazon.com/athena。[59] “Snowflake 文档：脚本：加载和卸载 Parquet 数据”，
d
o
c
s.
s
n
o
wflake.com/en/user-guide/script-data-load-transform-parquet.html。[60] D. Xin、D. Petersohn、D. Tang、Y. Wu、J. E. Gonzalez、J. M. Hellerstein、A. D. Joseph 和 A. G. Parameswaran，“利用思考时间增强数据框查询的交互性”，arXiv 预印本 arXiv:2103.02145，2021 年。[61] J. C. Bezdek 和 R. J. Hathaway，“关于交替优化的一些说明”，载于 AFSS国际模糊系统会议。Springer，2002 年，第 288–300 页。[62] H. Kellerer、U. Pferschy 和 D. Pisinger，“多维背包问题”，载于《背包问题》。Springer，2004 年，第 235–283 页。[63] “路线规划、调度、计划制定、分配、装箱、求解。OR-Tools 是一款快速且可移植的组合优化软件。”developers.google.com/optimization，发布方 = （访问日期：2022年12月15日）。[64] H. Seitz，“对最小线性排列问题的贡献”，博士论文，海德堡大学¨，2010 年。[65] M. Charikar、M. T. Hajiaghayi、H. Karloff 和 S. Rao，“ 用于顶点排序问题的 l 2 2 扩散度量 ”，《算法学》，第 56 卷，第 4 期，第 577–604 页，2010 年。[66] S. Baruah，“有向无环图调度问题的整数线性规划表示”，《实时系统》，第 58 卷，第 1 期，第 85–102 页，2022 年。[67] L. Marchal、H. Nagy、B. Simon 和 F. Vivien，“内存约束下有向无环图的并行调度”，载于 2018 年 IEEE国际并行与分布式处理研讨会（IPDPS）。IEEE，2018 年，第 204–213 页。[68] “Python 网络分析”，networkx.org，发布方 =（访问日期：2022年12月15日）。[69] “TPC-DS”，www.tpc.org/tpcds，发布方 = 15，（访问 2022
日期：）。[70] R. Ravi、A. Agrawal 和 P. Klein，“近似求解排序问题：单处理器调度和区间图补全”，载于国际自动机、语言与程序设计研讨会。Springer，1991 年，第 751–762 页。[71] S. Rao 和 A. W. Richa，“若干线性排序问题的新近似技术”，《SIAM计算杂志》，第 34 卷，第 2 期，第 388–404 页，2005 年。[72] H. Arabnejad 和 J. G. Barbosa，“基于乐观成本表的异构系统列表调度算法”，《IEEE并行与分布式系统汇刊》，第 25 卷，第 3 期，第 682–694 页，2014 年。[73] T. Yu、R. Zhang、K. Yang、M. Yasunaga、D. Wang、Z. Li、J. Ma、I. Li、Q. Yao、S. Roman、Z. Zhang 和 D. Radev，“Spider：用于复杂跨领域语义解析和文本转SQL任务的大规模人工标注数据集”，载于自然语言处理实证方法会议论文集，2018 年。[74] “Presto Python 连接器”，github.com/prestodb/presto-python-client。[75] “Presto Hive连接器”，prestodb.io/docs/current/connector/hive.html。
76] “Presto 内存连接器”，prestodb.io/docs/current/connector/memory.html。
[77] “ORC 表格式”，orc.apache.org/。
[78] “Spark DAG 调度器，”spark.apache.org/docs/1.2.1/api/java/org/apache/spark/scheduler/\DAGScheduler.ht ml
[79] G. Bathie、L. Marchal、Y. Robert 和 S. Thibault，《在共享内存平台上重新审视内存约束下的动态 DAG 调度》，载于《2020 年 IEEE 国际并行与分布式处理研讨会研讨会（IPDPSW）》。IEEE，2020 年，第 597–606 页。
[80] E. Zgraggen、A. Galakatos、A. Crotty、J.-D. Fekete 和 T. Kraska，《渐进式可视化如何影响探索性分析》，《IEEE 可视化与计算机图形学汇刊》，第 23 卷，第 8 期，第 1977–1987 页，2016 年。
[81] P. Eichmann、E. Zgraggen、C. Binnig 和 T. Kraska，《Idebench：交互式数据探索基准测试》，载于《2020 年 ACM 数据管理国际会议论文集》，2020 年，第 1555–1569 页。
[82] “Jupyter Notebook：经典笔记本界面”，jupyter.org。
[83] “RStudio 现已更名为 Posit，我们的使命仍在继续”，posit.co。
[84] “Colaboratory”，colab.research.google.com。
[85] F. Nagel、P. Boncz 和 S. D. Viglas，“流水线查询评估中的回收”，载于《2013 年第 29 届 IEEE 国际数据工程会议》（ICDE）。IEEE，2013 年，第 338–349 页。
[m]
86] A. Galakatos、A. Crotty、E. Zgraggen、C. Binnig 和 T. Kraska，“重新审视近似查询处理中的复用”，《VLDB Endowment 论文集》，第 10 卷，第 10 期，第 1142–1153 页，2017 年。
[87] J. M. Hellerstein、P. J. Haas 和 H. J. Wang，《在线聚合》，载于《1997 年数据管理国际会议论文集》，1997 年，第 171–182 页。

此图“fig1.png”可从以下位置获取“PNG”格式版本：

http://arxiv.org/ps/2303.09774v1