Lakehouse：统一数据仓库和高级分析的新一代开放平台

Michael Armbrust[1], Ali Ghodsi[1,2], Reynold Xin[1], Matei Zaharia[1,3]
[1]Databricks, [2]UC Berkeley, [3]Stanford University

摘要

本文认为，我们今天所知道的数据仓库架构将在未来几年内消亡，并被一种新构模式所取代，即 Lakehouse，它将 (i) 基于开直接访问Apache Parquet 等数据格式，(ii) 对机器学习和数据科学提供一流的支持，(iii) 提先进的性能。 Lakehouses 可以帮助解据仓库的几个主要挑战，包括数据陈旧性靠性、总拥有成本、数据锁定和有限的用持。我们讨论了该行业如何已经向 Lakehouses 发展，以及这种转变可能如何影响数据管理工作们还报告了使用 Parquet 的 Lakehouse 系统的结果，该系 TPC-DS 上的流行云数据仓库具有竞争力。

1 引言

本文认为，我们今天所知的数据仓库架构将在未来几年逐渐消失，并被一种新的架构模式所取代，我们称之为 Lakehouse，其特点是 (i) 开放的直接访问数据格式，例如 Apache Parquet 和 ORC，(ii) 对机器学习和数据科学工作负载的一流支以及 (iii) 最先进的性能。

数据仓库的历史始于通过将运营数据库中的数据收集中式仓库中来帮助业务领导者获得分析洞察力，然后可将其用策支持和商业智能 (BI)。这库中的数据将使用写入时模式写入，从而确保数据模型针对下游 BI 消费进行了优化们将其称为第一代数据分析平台。

十年前，第一代系统开始面临一些挑战。首先，它们通常将计算和存储耦合到本地设备中。这迫使企业为用户负载和管理数据的高峰提供和支付费用，随着数据集的增长，这变得非常昂贵。其次，不仅数据集增速，而且越来越多的数据集是完全非结构化的如视频、音频和文本文档，数据仓本无法存储和查询。

为了解决这些问题，第二代数据分析平台开始将所有原始数据卸载到数据湖中：具有文件 API 的低成本存储系统，以通用且通常开放的文件格式保存数据，例如 Apache Parquet 和 ORC [8, 9]。这种方法始于 Apache Hadoop 运动 [5]，使用 Hadoop 文件系统 (HDFS) 进行廉价存储。数据湖是一种模式读取架构，能够以低成本灵活地存储任何数据，但另一方面，解决了数据质量问题和下游数据治理。在此[1]架构中，湖中的一小部分数据稍后将被 ETL 到下游数据仓库（例如 Teradata），用于最重要的决策支持和 BI 应用程序。开放格式的使用还使数据湖数据可以直接访问广泛的其他分析引擎，例如机器学习系统 [30,37,42]。

从 2015 年开始，S3、ADLS 和 GCS 等云数据湖开始取代 HDFS。它们具有出色的耐用性（通常 > 10 个9）、地理复制，最重要的是，成本极低，可以自动进行甚至更便宜的存档存储，例如 AWS Glacier。云中的其余架构与第二代系统中的架构基本相同，具有下游数据仓库，例如 Redshift 或 Snowflake。根据我们的经验，这种两层数据湖 + 仓库架构现在在行业中占主导地位（几乎所有财富 500 强企业都使用过）。

这给我们带来了当前数据架构的挑战。虽然云数据湖和仓库架构由于单独的存储（例如，S3）和计算（例如，Redshift）而表面上宜，但两层架构对用户来说非常复杂。在第平台中，所有数据都从操作数据系接 ETL 到仓库中。在当今的架构中，数据首先 ETL 到湖中，然后再次 ELT 到仓库中，从而产杂性、延迟和新的故障模式。此外，企例现在包括机器学习等高级分析 湖和仓库都不是理想的具体来说今的数据架构通常存在四个问题：

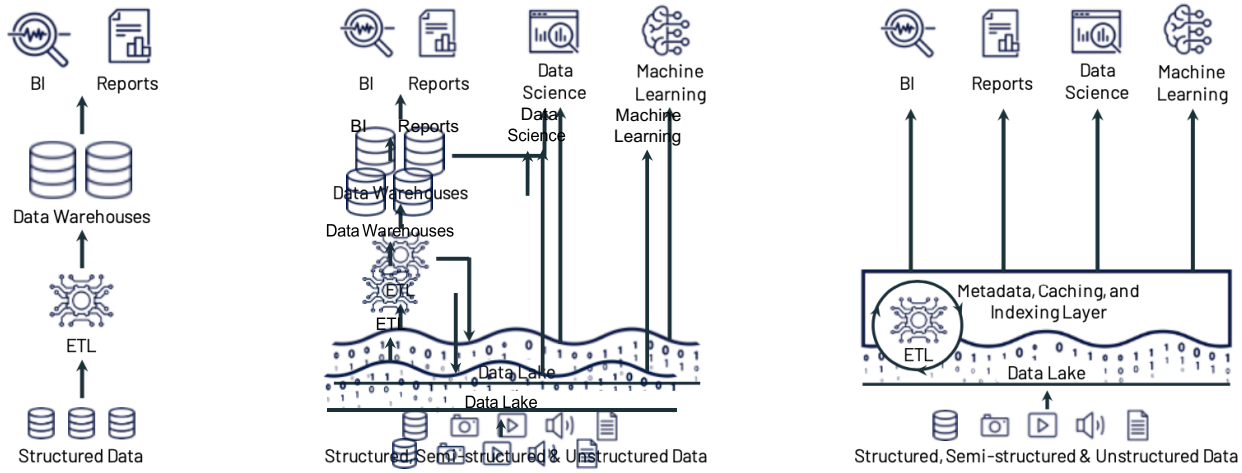
可靠性:保持数据湖和仓库的一致性既困难又昂贵。需要对两个系统之间的 ETL 数据进行持续工程，并使其可用于高性能决策支持和 BI。每个 ETL 步骤也有可能导致失败或引入降低数据质量的错误，例如，由于数据湖和仓库引擎之间的细微差异。

数据陈旧:与数据湖中的数据相比，仓库中的数据陈旧，新数据通常需要数天才能加载。与第一代分析系统相比，这是一个倒退，新的操作数据可以立即用于查询。根据 Dimensional Research 和 Fivetran 的一项调查，86% 的分析师使用过时的数据，62% 的分析师报告每月多次等待工程资源 [47]。

对高级分析的支持有限:企业希望使用他们的仓储数据提出预测性问题，例如，“我应该向哪些客户提供折扣？”尽管对 ML 和数据管理的融合进行了大量研究，但没有一个领先的机器学习系统，如 TensorFlow、PyTorch 和 XGBoost，在仓库之上运行良好。与提取少量数据的 BI 查询不同，这些系统需要使用复杂的非 SQL 代码处理大型数据集。通过 ODBC/JDBC 读取这些数据效率低下，并且无法直接访问内部仓库专有格式。对于这些用例，仓应商建议将数据导出到文件，这会进一步增加复杂性和陈旧性（添加第三个 ETL 步骤！）。或者户可以针对开放格式的数据湖数据运行这些系统而，他们随后失去了数据仓库丰富的管理功例如 ACID 事务、数据版本控制和索引。

总拥有成本: 除了为持续的ETL付费外，用户还要为复制到仓库的数据支付双倍的存储成本，而商业仓库将数据锁定为专有格式，这增加了将数据或工作负载迁移到其他系统的成本。

采用有限的稻草人解决方案是完全消除数据湖，并将所有数据存储在一个具有内置计算和存储分离功能的仓库中。我们将认为，这种方法的可行性有限，因为它仍然不支持轻松管理视频/音频/文本数据，也不支持从ML和数据科学工作负载直接快速访问。



(a) First-generation platforms. **(b) Current two-tier architectures.** **(c) Lakehouse platforms.**

图 1：数据平台架构向当今两层模型 (a-b) 和新 Lakehouse 模型 (c) 的演变.

在本文中，我们将讨论以下技术问题：是否有可能将基于标准开放数据格式（如 Parquet 和 ORC）的数据湖变成高性能系统，既可以提供数据仓库的性能和管理特性，又可以快速，来自高级分析工作负载的直接 I/O？我们认为这种类型的系统设计，我们称之为 Lakehouse（**图 1**），既可行又已经在行业中以各种形式显示出成功的证据。随着越来越多的业务应用程序开始依赖运营数据和高级分析，我们相信 Lakehouse 是一个引人注目的设计点，可以消除数据仓库的一些主要挑战。

特别是，我们认为 Lakehouse 的时机已经到来为最近解决了以下关键问题的解决方案：

1. 对数据湖的可靠数据管理： Lakehouse 需要能够存储原始数据，类似于今天的数据湖，同时支持 ETL/ELT 流程理这些数据以提高其分析质量。传统上，数据湖将数据管理为半结构化的“只是一堆文件”，因此很难提供一化数据仓库中 ETL/ELT 的关键管理功能，例如事务、回滚到旧表版本和零拷贝克隆。然而近的一系列系统，如 Delta Lake [10] 和 Apache Ice berg [7] 提供了数据湖的事务视图，并启用了这理功能。当然，组织仍然需力编写 ETL/ELT 逻辑来使用 Lakehouse 创建精选数据集，但总体而言 ETL 步骤更少，分析可以轻松高效地查询原始数据表，如果他们愿意，很像第一代分析平台。

2. 对机器学习和数据科学的支持： ML 系从数据湖格式直接读取的支持已经们处于有效访问 Lakehouse 的有利位置。此许多 ML 系统已采用 DataFrames 作为操作数据的抽象，最近的系统设计明性 DataFrame API [11]，可以对 ML 工作负载中的数据访问执行查询优这些 API 使 ML 工作负够直接受益于 Lakehouses 中的许多优化。

3. SQL 性能： Lakehouses 需要在过去十年积累的海量 Parquet/ORC 数据集之上提供最先进的 SQL 性能（或者从长远来看，一些其他标准格式已公开用问应用程序）。相比之下，经典数据仓库接受 SQL 并且可以自由优化引擎盖下的所有内容，包有存储格式。尽管如此，我们表明可用多种技术来维关 Parquet/ORC 数据集据布些现实现具有竞争力的性能。我们展 Parquet 上的 SQL 引擎（Databricks Delta Engine [19]能优于 TPC-DS 上领先的云数据仓库。

在本文的其余部分，我们详细介绍了 Lakehouse 平台的动机、潜在的技术设计和研究意义。

2 动机: 数据仓库挑战

数据仓库对于许多业务流程至关重要，但它们仍然经常以不正确的数据、陈旧性和高成本让用户感到沮丧。我们认为，这些挑战中的至少一部分是企业数据平台设计方式的“意外复杂性”[18]，而 Lakehouse 可以消除这些挑战。

首先，当今企业数据用户报告的首要问题通常是数据质量和可靠性 [47, 48]。实施正确的数据管道本质上是困难的，但如今具有独立湖和仓库的两层数据架构增加了额外的复杂性，加剧了这个问题。例如，数据湖和仓库系统可能在其支持的数据类型、SQL 方言等方面具有不同的语义；数据可能以不同的模式存储在湖和仓库中（例如，非规范化为一个）；并且跨越多个系统的 ETL/ELT 作业数量的增加增加了失败和错误的可能性。

其次，越来越多的业务应用程序需要最新数据，但当今的架构通过在仓库之前为传入数据设置单独的暂存区并使用定期 ETL/ELT 作业加载数据，从而增加了数据的陈旧性。从理论上讲，组织可以实施更多的流式管道以更快地更新数据仓库，但这些仍然比批处理作业更难操作。相比之下，在第一代平台中，仓库用户可以在与派生数据集相同的环境中立即访问从操作系统加载的原始数据。诸如客户支持系统和推荐引擎之类的业务应用程序对陈旧数据根本无效，甚至查询仓库的人类分析师也将陈旧数据报告为一个主要问题[47]。

第三，随着组织收集图像、传感器数据、文档等，现在许多行业 [22] 中的大部分数据都是非结构化的。组织需要易于使用的系统来管理这些数据，但 SQL 数据仓库及其API不要轻易支持它。

最后，大多数组织现在都在部署机器学习和数据科学应用程序，但数据仓库和湖泊并不能很好地服务这些应用程序。

如前所述，这些应用程序需要使用非 SQL 代码处理大量数据，因此它们无过 ODBC/JDBC 高效运行。随着高级分析系统的不断发展，我们相信让他们以开放格式直接访问数据将是支持他们的最有效方式。此外，ML 和数据科学应用程经典应用程序存在相同的数据管理问题如数据质量、一致性和隔离性[17、27、31]，因此，将 DBMS 功能引入他们的数据具有巨大的价值。

通往 Lakehouses 的现有步骤：当前的几个行势进一步证明了客户对两层湖+仓库模式的不满。首先，近年来，几乎所有主要的数据仓库都增加了对 Parquet 和 ORC 格式[12、14、43、46]的外部表的支持。这允许仓户也可以从同一个 SQL 引擎查询数据湖，不会使数据湖表更易于管理，也不除 ETL 复杂性、陈旧性和仓库中数据的高级分析挑战。在实践中，这些连接常也表现不佳，因为 SQL 引擎大多针内部数据格式进行了优化。其次，在直接针对数据湖存储运行的 SQL 引擎方面也有广泛的投资，例如 Spark SQL、Presto、Hive 和 AWS Athena [3, 11, 45, 50]。然而靠这些引擎并不能解决数据替代仓库的所有问题：数据湖仍然缺乏ACID事务等基本管理特性和索引等高效访问方匹配数据仓库性能。

3 Lakehouse 架构

我们将 Lakehouse 定义为基成本直接访问储的数据管理系统，它还提供传析 DBMS 管理和性能特性，例如 ACID 事务、数据版本控制、审计、索引、缓查询优化。因此，Lakehouses 结合了数据湖和数据仓库的主要优势：开放格式的低成本存储，前者的各种系统均可访问，后者具有强大的管理和优化功能。关键问题是是否可效地结合这些优势：特别是，Lakehouses 对直接访问的支味着它们放弃了数据独立性的某些方面，而直是关系 DBMS 设计的基石。

我们注意到 Lakehouses 特别适合具有独立计算和存储的云环境：不同的计用程序可以在完全独立的计算节点（例如，用于 ML 的 GPU 集群）上按需运行，同时直接访问相同的存储数据。但是，也可以在 HDFS 等本地存储系统上实施 Lakehouse。

为 Lakehouse 系统勾勒出一种可能的设计出现的三种技术理念，在整个行业中以各种形式我们一直基于此设计的 Lakehouse 平过 Delta Lake、Delta Engine 和 Databricks ML 运行时[10、19、38]构建项目 Databricks。然而，其他设计也可能是可行的，就们高级设计中的其他具体技术选择一样（例如们在 Databricks 的堆栈目前构建在 Parquet 存储格式上，但可以设计更好的格式）。我们讨论了几种替代方案和未来的研究方向。

3.1 实现 Lakehouse 系统

我们为实现 Lakehouse 提出的第一个关键想法是让将数据存储在低成本对象存储（例如 Amazon 使用 Apache Parquet 等标准文件格现事务性元数据层对象存储之上的定义了哪些对象是表版本的一部分。这允统在元数据层内实现管理功能，例如 ACID 事务或版本控制，同时量数据保留在低成本对象存储中，并允许客户用标准文件格式直接从该存储中读取对多数情况下。最近的几个系统，包括 Delta Lake [10] 和 Apache Iceberg [7]添加了管理功这种方式向数据湖例如，现数千名客户在 Databricks 大约一半的工作负载中使用 Delta Lake。

虽然元数据层增加了管理能力，足以实现良好的 SQL 性能。数据仓用多种技术来获得最先进的性能，例热数据存储在 SSD 等快速设备上、维护统计信息、构建索引等高效访问方法以及协同优化数据格式和计算引擎。在基有存储格式的 Lakehouse 中，无法更改格式，但我明可以实现其他优化保持数据文件不变，包存助数据结以据布局优化。

最后，由于声明性 DataFrame API [11, 37] 的开发，Lakehouses 既可以加速高级分析工作负载，又可以为它们提供更好的数据管理功能。

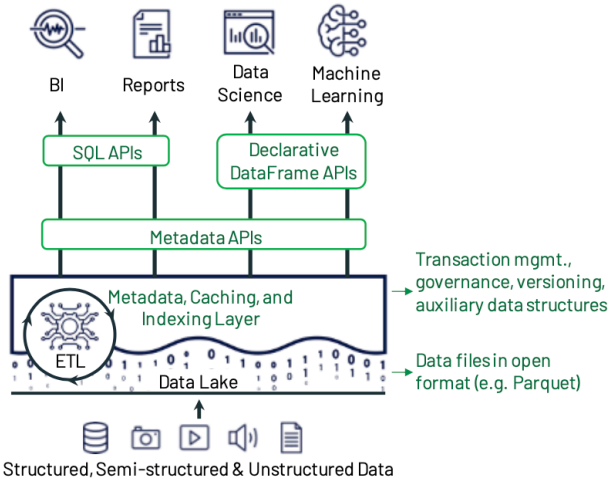


图 2：Lakehouse 系统设计示例，关键组件以绿色显示。该系统以元数据层为中心，例如 Delta Lake，它以开放格式在文件上添加事务、版本控制和辅助数据结构，并且可以使用各种 API 和引擎进行查询

许多 ML 库，例如 TensorFlow 和 Spark MLlib，已经可以读如 Parquet]据[30、37、42]。因此它们与 Lakehouse 集成的最简单方法是查数据层以确定哪些 Parquet 文件当表的一部分，然后简单地将它们传递给 ML 库。但是些系统中的大多数都支持用于数据准备的 DataFrame API，从而创造了更多的优化机会。 DataFrames 由 R 和 Pandas [40] 推广，并简单地为用户提供具有各种转换运算符的象，其中大部射到关系代数。 Spark SQL 等系过延迟评估转换并成的运算符计划传递给优化器 [11]，使该 API 具有声明性。这些 API 可以利用 Lakehouse 中的新优化功能如缓存和辅助数据）来进一步加速 ML。

图 2 显示了这些想法如何结合到 Lakehouse 系统设计中。在接下来的三个部分中，我更详细地扩展这些技术思想并讨论相关的研究问题。

3.2 用于数据管理的元数据层

我们认为启用 Lakehouses 的第一个组件是数据湖存储之上的元数据层，它可以提高其抽别以实现 ACID 事务和其他管能。 S3 或 HDFS 等数据湖存储系统供低级对象存储或文件系统接口，其中即使单的操作（例如更新跨多个文件的表不是原子操作。组织很快开这些系统上设计更丰富的数据管理层，从 Apache Hive ACID [33跟踪给定表版本中哪些数据文件是 Hive 表的一部分用 OLTP DBMS 以事务方式更新该集合。近年来，新系供了更多功能并提高了可扩展性。2016 年，Databricks 开始开发 Delta Lake [10]，它将有

关哪些对象是数据湖中表的一部分的信息存储为 Parquet 格式的事务日志，使其能够扩展到每个表数十亿个对象。始于 Netflix 的 Apache Iceberg [7] 使用类似的设计，同时支持 Parquet 和 ORC 存储。始于 Uber 的 Apache Hudi [6] 是该领域的另一个系统，专注于简化流式摄取到数据湖中，尽管它不支持并发写入器。

这些系统的经验表明，它们通常提供与原始 Parquet/ORC 数据湖相似或更好的性能，同时添加非常有用的管理功能，例如事务、零拷贝锥体和时间旅行到表的过去版本 [10]。此外，对于已经拥有数据湖的组织，它们很容易采用：例如，Delta Lake 可以将现有 Parquet 文件目录转换为零副本的 Delta Lake 表，只需添加以条目开头的事务日志引用所有现有文件。因此，组织正在迅速采用这些元数据层：例如，Delta Lake 增长到在三年内覆盖了 Databricks 上一半的计算时间。

此外，元数据层是实现数据质量执行功能的自然场所。例如，Delta Lake 实施模式强制以确保上传到表的数据与其模式相匹配，并且约束 API [24] 允许表所有者对摄取的数据设置约束（例如，国家只能是一个值列表）。达美的客户图书馆将自动拒绝违反这些期望的记录或将其隔离在特殊位置。客户发现这些简单的功能对于提高基于数据湖的管道的质量非常有用。

最后，元数据层是实现访问控制和审计日志等治理功能的自然场所。例如，元数据层可以在授予客户端从云对象存储读取表中原始数据的凭据之前检查是否允许客户端访问表，并且可以可靠地记录所有访问。

未来方向和替代设计为数据湖的元数据层是一个相当新的发展，所以有许多悬而未决的问题和替代设计。例如，我们设计 Delta Lake 将其事务日志存储在其运行的同一对象存储（例如 S3）中，以简化管理（无需运行单独的存储系统）并提供高可用性和高读取带宽到日志（与对象存储相同）。但是，由于对象存储的高延迟，这限制了它可以支持的每秒事务速率。在某些情况下，使用更快的元数据存储系统的设计可能更可取。同样，Delta Lake、Iceberg 和 Hudi 一次只支持一张表上的事务，但应该可以扩展它们以支持跨表事务。优化事务日志的格式和管理对象的大小也是悬而未决的问题。

3.3 Lakehouse的SQL性能

Lakehouse 方法最大的技术问题可能是如何提供最先进的 SQL 性能，同时放弃传统 DBMS 设计中的大部分数据独立性。答案显然取决于许多因素，例如我们有哪些可用的硬件资源（例如，我们是否可以在对象存储之上实现一个缓存层）以及我们是否可以更改数据对象存储格式而不是使用现有的标准，例如如 Parquet 和 ORC（改进这些格式的新设计不断出现 [15, 28]）。然而，无论具体设计如何，核心挑战在于数据存储格式成为系统公共 API 的一部分，以允许快速直接访问，这与传统 DBMS 不同。

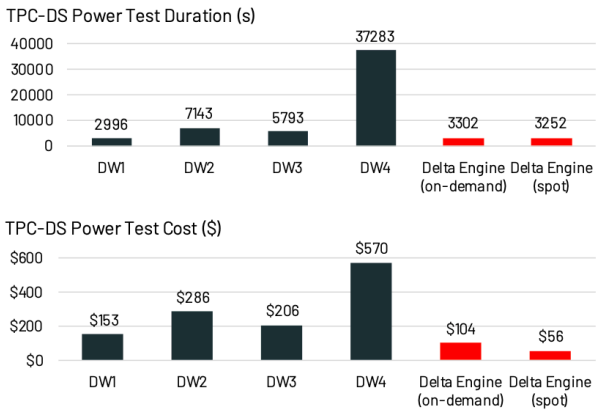


图 3：使用 Delta Engine 与 AWS、Azure 和 Google Cloud 上的流行云数据仓库在规模因子 30K 下的 TPC-DS 功率得分（运行所有查询的时间）和成本

我们提出了几种技术来在独立于所选数据格式的 Lakehouse 中实现 SQL 性能优化，因此可以应用于现有或未来的格式。我们还在 Databricks Delta Engine [19] 中实施了这些技术，并表明它们在流行的云数据仓库中产生了具有竞争力的性能，尽管还有足够的空间进行进一步的性能优化。这些与格式无关的优化是：

缓存：当使用事务元数据层（例如 Delta Lake）时，Lakehouse 系统可以安全地将云对象存储中的文件缓存到更快的存储设备上，例如处理节点上的 SSD 和 RAM。正在运行的事务可以很容易地确定缓存文件何时仍然可以读取。此外，缓存可以采用更高效的查询引擎运行的转码格式，与传统“封闭世界”数据仓库引擎中使用的任何优化相匹配。例如，我们在 Databricks 的缓存部分解压缩了它加载的 Parquet 数据。

辅助数据：即使 Lakehouse 需要为直接 I/O 公开基表存储格式，它也可以维护其他数据，以帮助优化它完全控制的辅助文件中的查询。在 Delta Lake 和 Delta Engine 中，我们在用于存储事务日志的同一个 Parquet 文件中维护表中每个数据文件的列 min-max 统计信息，这可以在基础数据由特定列聚集时进行数据跳过优化。我们还实现了一个基于布隆过滤器的索引。可以想象在这里实现各种辅助数据结构，类似于索引“原始”数据的提议 [1, 2, 34]。

数据布局：数据布局对访问性能有很大影响。即使我们修复了 Parquet 等存储格式，Lakehouse 系统也可以优化多种布局决策。最明显的是记录排序：哪些记录聚集在一起，因此最容易一起阅读。在 Delta Lake 中，我们支持使用单个维度或空间填充曲线（例如 Z-order [39] 和 Hilbert 曲线）对记录进行排序，以提供跨多个维度的局部性。人们还可以想象新的格式，支持在每个数据文件中以不同的顺序放置列，为不同的记录组选择不同的压缩策略，或其他策略 [28]。

这三种优化对于分析系统中的典型访问模式特别有效。在典型的工作负载中，大多数查询往往集中针对数据的“热”子集，Lakehouse 可以使用与封闭世界数据仓库相同的优化数据结构缓存这些数据，以提供具有竞争力的性能。对于云对象存储中的“冷”数据，性能的主要决定因素可能是每次查询读取的数据量。在这种情况下，数据布局优化（集群共同访问数据）和辅助数据结构（例如区域图）（让引擎快速确定要读取的数据文件的范围）的组合

可以使 Lakehouse 系统最小化I/O 的方式与封闭世界的专有数据仓库相同，尽管运行的是标准的开放文件格式。

性能结果:在 Databricks，我们将这三个 Lakehouse 优化与用于 Apache Spark 的新 C++ 执行引擎相结合，称为 Delta Engine [19]。为了评估 Lakehouse 架构的可行性，图 3 比较了规模因子为 30,000 的 TPC-DS 上的 Delta Engine 与四个广泛使用的云数据仓库（来自云提供商以及在公共云上运行的第三方公司），使用AWS、Azure 和 Google Cloud 上的可比集群，每个集群有 960 个 vCPU 和本地 SSD 存储[2]。1 我们报告了运行所有 99 个查询的时间以及每项服务定价模型中客户的总成本（Databricks 允许用户选择现场和现场需求实例，所以我们同时展示）。 Delta Engine 以更低的价格提供与这些系统相当或更好的性能。

未来方向和替代设计:设计高性能且可直接访问的 Lakehouse 系统是未来工作的一个丰富领域。我们尚未探索的一个明确方向是设计新的数据湖存储格式，以便在这个用例中更好地工作，例如，为 Lakehouse 系统提供更多灵活性以实现数据布局优化或索引的格式，或者只是更好的格式适合现代硬件。当然，处理引擎采用这种新格式可能需要一段时间，从而限制了可以从中读取的客户端数量，但是为下一代工作负载设计一种高质量的可直接访问的开放格式是一个重要的研究问题。

即使不改变数据格式，也有许多类型的缓存策略、辅助数据结构和数据布局策略可供 Lakehouses [4, 49, 53] 探索。确定哪些可能对云对象存储中的海量数据集最有效是一个悬而未决的问题。

最后，另一个令人兴奋的研究方向是确定何时以及如何使用无服务器计算系统来回答查询 [41] 并优化存储、元数据层和查询引擎设计以最大限度地减少这种情况下的延迟。

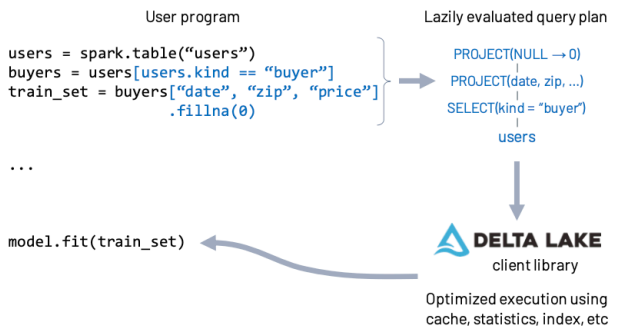


图 4: Spark MLlib 中使用的声明性 DataFrame API 的执行。用户代码中的 DataFrame 操作延迟执行，允许 Spark 引擎捕获数据加载计算的查询计划并将其传递给 Delta Lake 客户端库。该库查询元数据层以确定要读取哪些分区、使用缓存等

3.4 高效访问高级分析

正如我们在本文前面所讨论的，高级分析库通常使用不能作为 SQL 运行的命令式代码编写，但它们需要访问大量数据。有一个有趣的研究问题是如何设计这些库中的数据访问层，以最大限度地提高在上面运行的代码的灵活性，但仍然受益于 Lakehouse 中的优化机会。

我们取得成功的一种方法是提供这些库中使用的 DataFrame API 的声明式版本，它将数据准备计算映射到 Spark SQL 查询计划中，并且可以从 Delta Lake 和 Delta Engine 的优化中受益。我们在 Spark DataFrames [11] 和 Koalas [35] 中都使用了这种方法，这是一种用于 Spark 的新 DataFrame API，可提高与 Pandas 的兼容性。 DataFrames 是用于将输入传递到 Apache Spark 高级分析库生态系统的主要数据类型，包括 MLlib [37]、GraphFrames [21]、SparkR [51] 和许多社区库，因此所有这些工作负载都可以享受加速 I /O 如果我们可以优化 DataFrame 计算。 Spark 的查询计划器将用户 DataFrame 计算中的选择和投影直接推送到每个数据源读取的“数据源”插件类中。因此，在我们实现 Delta Lake 数据源的过程中，我们利用第 3.3 节中描述的缓存、数据跳过和数据布局优化来加速从 Delta Lake 读取这些数据，从而加速 ML 和数据科学工作负载，如**图 4** 所示.

然而，机器学习 API 正在迅速发展，还有其他数据访问 API，例如 TensorFlow 的 tf.data，它们不会尝试将查询语义推送到底层存储系统中。其中许多 API 还专注于 CPU 上的重叠数据加载与 CPU 到 GPU 的传输和 GPU 计算，这在数据仓库中并没有受到太多关注。最近的系统工作表明，保持现代加速器得到充分利用，特别是对于 ML 推理，可能是一个难题 [44]，因此 Lakehouse 访问库将需要应对这一挑战。

未来方向和替代设计。除了我们刚刚讨论的关于现有 API 和效率的问题之外，我们还可以探索与 ML 完全不同的数据访问接口设计。例如，最近的工作提出了将 ML 逻辑推入 SQL 连接的“因子化 ML”框架，以及可应用于 SQL [36] 中实现的 ML 算法的其他查询优化。最后，我们仍然需要标准接口让数据科学家充分利用 Lakehouses（甚至数据仓库）中强大的数据管理能力。例如，在 Databricks，我们将 Delta Lake 与 MLflow [52] 中的 ML 实验跟踪服务集成在一起，让数据科学家可以轻松跟踪实验中使用的表版本，并在以后重现该版本的数据。业界还出现了一种新兴的特征存储抽象，作为数据管理层来存储和更新 ML 应用程序中使用的特征 [26、27、31]，这将受益于在 Lakehouse 设计中使用标准 DBMS 功能，例如事务和数据版本控制。

4 研究问题和意义

除了我们在第 3.2-3.4 节中提出的未来方向的研究挑战之外，Lakehouses 还提出了其他几个研究问题。此外，数据湖功能越来越丰富的行业趋势对数据系统研究的其他领域也有影响。

还有其他方法可以实现 Lakehouse 的目标吗？可以想象其他实现 Lakehouse 主要目标的方法，例如为数据仓库构建大规模并行服务层，该服务层可以支持来自高级分析工作负载的并行读取。但是，我们认为，与让工作负载直接访问对象存储相比，此类基础架构的运行成本、管理难度和性能可能会显着降低。我们还没有看到添加此类服务层的系统的广泛部署，例如 Hive LLAP [32]。此外，这种方法解决了选择用于读取到服务层的有效数据格式的问题，并且这种格式仍然需要易于从仓库的内部格式进行转码。云对象存储的主要吸引力在于其低成本、弹性工作负载的高带宽访问以及极高的可用性；在对象存储前面有一个单

[2] 我们启动了所有系统，在适用时将数据缓存在 SSD 上，因为我们比较的一些仓库仅支持节点附加存储。然而，当使用冷缓存启动时，Delta 引擎只慢了 18%。

独的服务层，这三个都变得更糟。

除了这些替代方法的性能、可用性、成本和锁定挑战之外，还有一些重要的治理原因导致企业可能更愿意以开放格式保存数据。随着对数据管理的监管要求越来越高，组织可能需要在短时间内搜索旧数据集、删除各种数据或更改其数据处理基础设施，而开放格式的标准化意味着他们将始终可以直接访问数据不阻止供应商。软件行业的长期趋势是开放数据格式，我们相信这种趋势将持续到企业数据中。

什么是正确的存储格式和访问 API？ Lakehouse 的访问接口包括原始存储格式、直接读取这种格式的客户端库（例如，当读取到 TensorFlow 时）和高级 SQL 接口。有许多不同的方法可以在这些层中放置丰富的功能，例如通过要求读者执行更复杂的“可编程”解码逻辑来为系统提供更大灵活性的存储方案[28]。还有待观察哪种存储格式组合，元数据层设计和访问 API 效果最好。

Lakehouse 如何影响其他数据管理研究和趋势？数据湖的流行以及对它们的丰富管理接口的日益使用，无论是元数据层还是完整的 Lakehouse 设计，都对数据管理研究的其他几个领域产生了影响。

Polystore 旨在解决跨不同存储引擎查询数据的难题[25]。这个问题将在企业中持续存在，但是在云数据湖中以开放格式提供的数据越来越多，这意味着可以通过直接针对云对象存储运行来回答许多 polystore 查询，即使底层数据文件是其中的一部分逻辑上独立的 Lakehouse 部署。

数据集成和清理工具也可以设计为在 Lakehouse 上就地运行，可以快速并行访问所有数据，这可能会启用新算法，例如在组织中的许多数据集上运行大型连接和聚类算法。

通过使用其事务管理 API 将数据直接归档到 Lakehouse 系统中，HTAP 系统也许可以构建为 Lakehouse 前面的“附加”层。 Lakehouse 将能够查询数据的一致快照。

如果在 Lakehouse 上实施，ML 的数据管理也可能变得更简单、更强大。如今，组织正在构建范围广泛的 ML 特定数据版本控制和“特征存储”系统 [26、27、31]，以重新实现标准 DBMS 功能。使用内置 DBMS 管理功能的数据湖抽象来实现特征存储功能可能更简单。同时，诸如分解 ML [36] 之类的声明性 ML 系统可能会在 Lakehouse 上运行良好。

云原生 DBMS 设计（例如无服务器引擎 [41]）将需要与更丰富的元数据管理层（例如 Delta Lake）集成，而不是仅仅扫描数据湖中的原始文件，但可能能够实现更高的性能。

最后，业界一直在讨论如何组织数据工程流程和团队，例如“数据网格”[23] 等概念，其中不同的团队端到端拥有不同的数据产品，比传统的“中央数据团队”方法。 Lakehouse 设计很容易适用于分布式协作结构，因为所有数据集都可以从对象存储直接访问，而无需让用户使用相同的计算资源，无论哪个团队生产和使用数据，都可以直接共享数据。

5 相关工作

Lakehouse 方法建立在为云设计数据管理系统的许多研究工作的基础上，从早期工作开始，即使用 S3 作为 DBMS [16] 中的块存储，并在云对象存储上实现“螺栓固定”一致性 [13]。它还大量建立在研究的基础上，通过围绕固定数据格式构建辅助数据结构来加速查询处理 [1, 2, 34, 53]。

最密切相关的系统是由独立存储支持的“云原生”数据仓库 [20, 29] 和 Apache Hive [50] 等数据湖系统。 Snowflake 和 BigQuery [20, 29] 等云原生仓库已经取得了良好的商业成功，但它们仍然不是大多数大型组织的主要数据存储：大部分数据仍在数据湖中，可以轻松存储大量企业数据进入的时间序列、文本、图像、音频和半结构化格式。因此，云仓库系统都增加了对读取数据湖格式的外部表的支持 [12、14、43、46]。但是，这些系统无法像处理内部数据一样为数据湖中的数据提供任何管理功能（例如，对其执行 ACID 事务），因此将它们与数据湖一起使用仍然很困难且容易出错。数据仓库也不适合大规模机器学习和数据科学工作负载，因为与直接对象存储访问相比，从数据仓库中流式传输数据效率低下。

另一方面，虽然早期的数据湖系统为了易于实现而有意减少了关系 DBMS 的特性集，但所有这些系统的趋势一直是添加 ACID 支持 [33] 以及越来越丰富的管理和性能特性 [6、7、10]。在本文中，我们推断这一趋势以讨论哪些技术设计可以让 Lakehouse 系统完全取代数据仓库，展示针对 Lakehouse 优化的新查询引擎的定量结果，并勾勒出该领域的一些重要研究问题和设计替代方案。

6 结论

我们认为，在开放数据湖文件格式上实现数据仓库功能的统一数据平台架构可以提供与当今数据仓库系统竞争的性能，并有助于解决数据仓库用户面临的许多挑战。尽管将数据仓库的存储层限制为以标准格式打开、可直接访问的文件起初似乎是一个重大限制，但热数据的缓存和冷数据的数据布局优化等优化可以让 Lakehouse 系统获得具有竞争力的性能。我们认为，鉴于数据湖中已有大量数据，并且有机会大大简化企业数据架构，该行业可能会趋向于 Lakehouse 设计。

致谢

我们感谢 Databricks 的 Delta Engine、Delta Lake 和 Benchmarking 团队对我们在这项工作中讨论的结果所做的贡献。 Awez Syed、Alex Behm、Greg Rahn、Mostafa Mokhtar、Peter Boncz、Bharath Gowda、Joel Minnick 和 Bart Samwel 就本文中的想法提供了宝贵的反馈。我们也感谢 CIDR 审阅者的反馈。

References

[1] I. Alagiannis, R. Borovica-Gajic, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient query execution on raw data files. *CACM*, 58(12):112–121, Nov. 2015.
[2] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: a hands-free adaptive store. In *SIGMOD*, 2014.
[3] Amazon Athena. https://aws.amazon.com/athena/.
[4] G. Ananthanarayanan, A. Ghodsi, A. Warfield, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated memory caching for parallel jobs. In *NSDI*, pages 267–280, 2012.
[5] Apache Hadoop. https://hadoop.apache.org.
[6] Apache Hudi. https://hudi.apache.org.
[7] Apache Iceberg. https://iceberg.apache.org.
[8] Apache ORC. https://orc.apache.org.
[9] Apache Parquet. https://parquet.apache.org.
[10] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. undefineduszczak, M. undefinedwitakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, and M. Zaharia. Delta Lake: High-performance ACID table storage over cloud object stores. In *VLDB*, 2020.
[11] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan,

M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*, 2015.
[12] Azure Synapse: Create external file format. https://docs.microsoft.com/en-us/sql/t-sql/statements/create-external-file-format-transact-sql.
[13] P. Bailis, A. Ghodsi, J. Hellerstein, and I. Stoica. Bolt-on causal consistency. pages 761–772, 06 2013.
[14] BigQuery: Creating a table definition file for an external data source. https://cloud.google.com/bigquery/external-table-definition, 2020.
[15] P. Boncz, T. Neumann, and V. Leis. FSST: Fast random access string compression. In *VLDB*, 2020.
[16] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD*, pages 251–264, 01 2008.
[17] E. Breck, M. Zinkevich, N. Polyzotis, S. Whang, and S. Roy. Data validation for machine learning. In *SysML*, 2019.
[18] F. Brooks, Jr. No silver bullet – essence and accidents of software engineering. *IEEE Computer*, 20:10–19, April 1987.
[19] A. Conway and J. Minnick. Introducing Delta Engine. https://databricks.com/blog/2020/06/24/introducing-delta-engine.html.
[20] B. Dageville, J. Huang, A. Lee, A. Motivala, A. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, P. Unterbrunner, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, and M. Hentschel. The Snowflake elastic data warehouse. pages 215–226, 06 2016.
[21] A. Dave, A. Jindal, L. E. Li, R. Xin, J. Gonzalez, and M. Zaharia. GraphFrames: An integrated API for mixing graph and relational queries. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, New York, NY, USA, 2016. Association for Computing Machinery.
[22] D. Davis. AI unleashes the power of unstructured data. https://www.cio.com/article/3406806/, 2019.
[23] Z. Dehghani. How to move beyond a monolithic data lake to a distributed data mesh. https://martinfowler.com/articles/data-monolith-to-mesh.html, 2019.
[24] Delta Lake constraints. https://docs.databricks.com/delta/delta-constraints.html, 2020.
[25] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik. The BigDAWG polystore system. *SIGMOD Rec.*, 44(2):11–16, Aug. 2015.
[26] Data Vesion Control (DVC). https://dvc.org.
[27] Feast: Feature store for machine learning. https://feast.dev, 2020.
[28] B. Ghita, D. G. Tomé, and P. A. Boncz. White-box compression: Learning and exploiting compact table representations. In *CIDR*. www.cidrdb.org, 2020.
[29] Google BigQuery. https://cloud.google.com/bigquery.
[30] Getting data into your H2O cluster. https://docs.h2o.ai/h2o/latest-stable/h2o-docs/getting-data-into-h2o.html, 2020.
[31] K. Hammar and J. Dowling. Feature store: The missing data layer in ML pipelines? https://www.logicalclocks.com/blog/feature-store-the-missing-data-layer-in-ml-pipelines, 2018.
[32] Hive LLAP. https://cwiki.apache.org/confluence/display/Hive/LLAP, 2020.
[33] Hive ACID documentation. https://docs.cloudera.com/HDPDocuments/HDP3/HDP-3.1.5/using-hiveql/content/hive_3_internals.html.
[34] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my data files. here are my queries. where are my results? In *CIDR*, 2011.
[35] koalas library. https://github.com/databricks/koalas, 2020.
[36] S. Li, L. Chen, and A. Kumar. Enabling and optimizing non-linear feature interactions in factorized linear algebra. In *SIGMOD*, page 1571–1588, 2019.
[37] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine learning in Apache Spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, Jan. 2016.
[38] Databricks ML runtime. https://databricks.com/product/machine-learning-runtime.
[39] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. IBM Technical Report, 1966.
[40] pandas Python data analysis library. https://pandas.pydata.org, 2017.
[41] M. Perron, R. Castro Fernandez, D. DeWitt, and S. Madden. Starling: A scalable query engine on cloud functions. In *SIGMOD*, page 131–141, 2020.
[42] Petastorm. https://github.com/uber/petastorm.
[43] Redshift CREATE EXTERNAL TABLE. https://docs.aws.amazon.com/redshift/latest/dg/r_CREATE_EXTERNAL_TABLE.html, 2020.
[44] D. Richins, D. Doshi, M. Blackmore, A. Thulaseedharan Nair, N. Pathapati, A. Patel, B. Daguman, D. Dobrijalowski, R. Illikkal, K. Long, D. Zimmerman, and V. Janapa Reddi. Missing the forest for the trees: End-to-end ai application performance in edge data centers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 515–528, 2020.
[45] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner. Presto: SQL on everything. In *ICDE*, pages 1802–1813, April 2019.
[46] Snowflake CREATE EXTERNAL TABLE. https://docs.snowflake.com/en/sql-reference/sql/create-external-table.html, 2020.
[47] Fivetran data analyst survey. https://fivetran.com/blog/analyst-survey, 2020.
[48] M. Stonebraker. Why the 'data lake' is really a 'data swamp'. BLOG@CACM, 2014.
[49] L. Sun, M. J. Franklin, J. Wang, and E. Wu. Skipping-oriented partitioning for columnar layouts. *Proc. VLDB Endow.*, 10(4):421–432, Nov. 2016.
[50] A. Thusoo et al. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*, pages 996–1005. IEEE, 2010.
[51] S. Venkataraman, Z. Yang, D. Liu, E. Liang, H. Falaki, X. Meng, R. Xin, A. Ghodsi, M. Franklin, I. Stoica, and M. Zaharia. SparkR: Scaling R programs with Spark. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1099–1104, New York, NY, USA, 2016. Association for Computing Machinery.
[52] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, F. Xie, and C. Zumar. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41:39–45, 2018.
[53] M. Ziauddin, A. Witkowski, Y. J. Kim, D. Potapov, J. Lahorani, and M. Krishna. Dimensions based data clustering and zone maps. *Proc. VLDB Endow.*, 10(12):1622–1633, Aug. 2017.