哈尔滨工业大学计算机科学与技术学院

《算法设计与分析》

课程报告

学号	2022113416
姓名	刘子康
班级	2203601
专业	人工智能
授课教师	张炜
报告日期	2023 年 01 月 05 日

论文题目

ConnectorX: Accelerating Data Loading From Databases to Dataframes

作者：Xiaoying Wang, Weiyuan Wu, Jinze Wu, Yizhou Chen, Nick Zrymiak, Changbo Qu, Lampros Flokas, George Chow, Jiannan Wang, Tianzheng Wang, Eugene Wu,Qingqing Zhou

刊物：VLDB

出处链接：https://www.vldb.org/pvldb/vol15/p2994-wang.pdf

1 问题定义

数据分析库，如 Pandas、 Dask 和 Modin ，在数据科学家中被广泛用于数据操作和分析，而很多企业通常将其数据存储在数据库管理系统中。因此，大多数据科学应用程序的第一步是从数据库管理系统中加载数据。然而，这种数据加载过程不仅非常慢，还会消耗过多的客户端内存，这很容易导致内存不足或性能下降。

这是一个亟待解决的问题，因为数据库读取是许多数据科学任务（如 ELT/ETL 流程和探索性数据分析）的关键，并且在实际的机器学习(Machine Learning, ML)流水线中可能占据 50%以上的时间。因此，弥合数据库和数据帧之间的差距是学术和产业上都非常关注的问题。

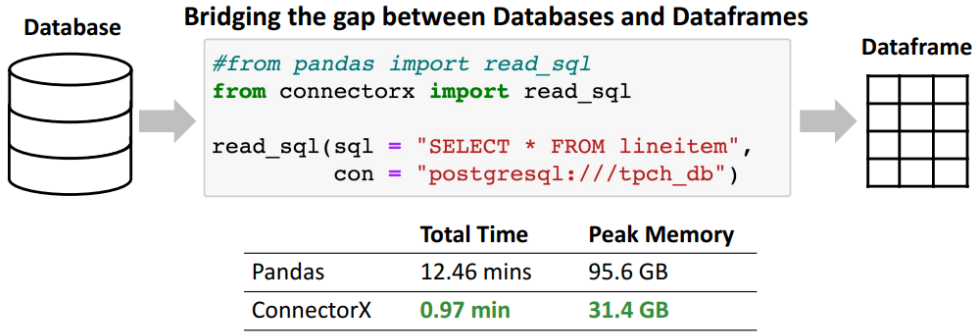

图 1：表格（CSV 格式，7.2 GB）从数据库向数据帧加载加快、内存使用量减少

1.1 现有问题

首先，实际的数据加载关键限制因素在哪里？研究团队发现 Pandas.read_sql 函数运行时间可分为两部分：服务器端运行时间（包括查询执行、序列化和网络传输），客户端运行时间（包括反序列化和转换成数据帧）。结果显示，超过 85%的时间都花在了客户端的运行，并且所有中间转换结果都在内存中实现。这表明，

客户端的优化可以显著降低数据加载成本。

其次，如何既减少运行时间和内存，又能使系统扩展到新的 DBMS ？研究团队设计运用了一种简洁的领域特定用语言(DSL)，用于将 DBMS 的结果表示映射到数据帧中，使代码行数减少了 1-2 个数量级。 ConnectorX 将 DSL 编译成流式工作系统，通过该系统将从网络接收到的字节高效地转换为内存中的对象。该工作流以流水线方式执行，并无缝结合了并行执行、字符串分配优化和高效数据表示等优化技术。

最后，广泛使用的客户端查询分区方案是否足够好？通过查询分区实现并行化是减少执行和加载时间的主要方法。包括 ConnectorX 在内的现有库在客户端进行查询分区，由于不需要对服务器进行修改，因此这种方式很受欢迎。然而，研究团队发现这会带来额外的用户负担、负载不平衡、服务器资源浪费和数据不一致。因此，团队研究了服务器端结果分区的方法，即数据库管理系统直接分区查询结果，并在不改变协议和访问方法的情况下并行发送回来，使用 PostgreSQL 制作了原型并演示了其功效。

1.2 研究背景

缩小 DBMS 与 ML 之间的差距已成为数据库领域的研究热点。ConnectorX 通过支持从 DBMS 到数据框架的高效数据加载，将这一问题纳入大局。

1.2.1 加速数据库管理系统的数据访问

（1）服务器端增强

通过元组级协议访问数据库系统的数据非常之慢。先前的研究表明，现有的线路协议存在冗余信息和成本较高的（去）序列化问题，并提出了一种新协议来解决这些问题。更多的方法倾向于利用现有的开放数据格式（例如，Parquet、ORC、Avro、Arrow，Dermel，Pickle），通过避免元组级别的访问来加快进程；Li 等人采用 Flight ，实现了以 Arrow IPC 格式导出数据时的零拷贝；Redshift 、BigQuery 和 Snowflake 等数据仓库支持直接将数据卸载到 CSV 和 Parquet 等格式的云存储（如 Amazon S3、Azure Blob Storage、Google Cloud Storage）中；数据湖和 Lakehouse 方案也主张直接访问开放格式。

并行性是另一种加速数据传输的有效方法，许多工具支持在两个文件系统（如 HDFS，S3）之间或文件系统和 DBMS 之间移动数据。Databricks 指出了单一 SQL 端点的不足，并建议使用云获取体系结构来解决这个问题，在云获取体系结构中，查询的结果被导出到多个分区中的云存储中，从而支持从客户端并行下载。其他云原生数据仓库也提供类似的支持。然而，所有这些服务器端尝试都

需要用户修改数据库服务器的源代码或切换到新的服务器，这在实际应用中通常是不可行的。即使场景支持，其中许多解决方案也需要额外的配置，例如访问特定的系统（如 S3），这在不同的供应商之间也是不同的。并且，这些工作只提供了解决方案的一部分，要获得下游任务的最终数据帧，需要从导出的结果进行额外的转换。

与这些方法不同， ConnectorX 提供利用现有 DBMS 和客户端驱动程序的单步解决方案，并通过在当前环境设置中优化客户端执行来实现最大速度。此外，ConnectorX 在从 DBMS 获取数据方面没有限制，因此可以在内部利用这些服务器端进行优化，并从中受益。

（2）客户端优化

ML 和数据分析工具倾向于采用数据帧作为数据操作的抽象，其中许多都提供本地 DBMS I/O 支持和各种优化功能。Pandas 支持分块处理，通过一次加载一个数据块来减少内存压力。MODIN、DASK、Spark 通过客户端查询分区而利用多核。第三方库 Turbodbc 通过在 ODBC 驱动程序上进行批量数据传输提供了最先进的性能。ConnectorX 在以下三个方面都优于现有的客户端库。首先，现有方法仅限于具有特定接口的客户端驱动程序（例如，Pandas、Turbodbc 和 Spark 分别需要 Python DB-API、ODBC 和 JDBC），而 ConnectorX 没有这种要求，因此能够利用最快的接口。其次，它们只针对一个或几个数据帧，相比之下，ConnectorX 是一个通用框架，只要实现相应的接口，就能轻松扩展到任何数据帧，而且性能很高。最后，与这些库相比，ConnectorX 集成了更全面的优化技术，因此可以获得最好的表现。

1.2.2 数据库管理系统与数据科学的集成

一般来说，数据科学家更熟悉数据帧操作，因此他们通常会选择将完整数据从数据库转移到客户端机器，然后使用数据科学工具进行处理。为了避免这种代价高昂的数据移动，一些系统尝试与 Pythoh 和 R 环境集成，以便在数据库引擎中运行 ML 操作；嵌入式分析系统 DuckDB 通过将 DBMS 和分析工具置于同一地址空间，避免结果集序列化和基于值的 API 的关键限制因素。然而，这些方法仍处于早期阶段，只能支持有限数量的应用场景。因此，当没有可行的集成系统时，如果仍希望允许数据科学家将数据从 DBMS 中移出，以便在数据帧抽象上进行复杂的分析和构建 ML 模型，就需要更好地设计和实现类似 ConnectorX 的 read_sql。

2 算法描述和分析

2.1 基本算法描述

2.1.1 整体工作流程

ConnectorX 借鉴了分块法，采用了流式工作流程，客户端一次加载和处理一小批数据。为了避免在最后进行额外的数据复制和连接，ConnectorX 在工作中增加了准备阶段，该阶段的目标是预分配结果数据帧，以便在执行过程中直接将解析值写入相应的最终槽。图 2 为由两个阶段组成的整体工作流程：准备阶段（①—③）和执行阶段（④—⑥）。

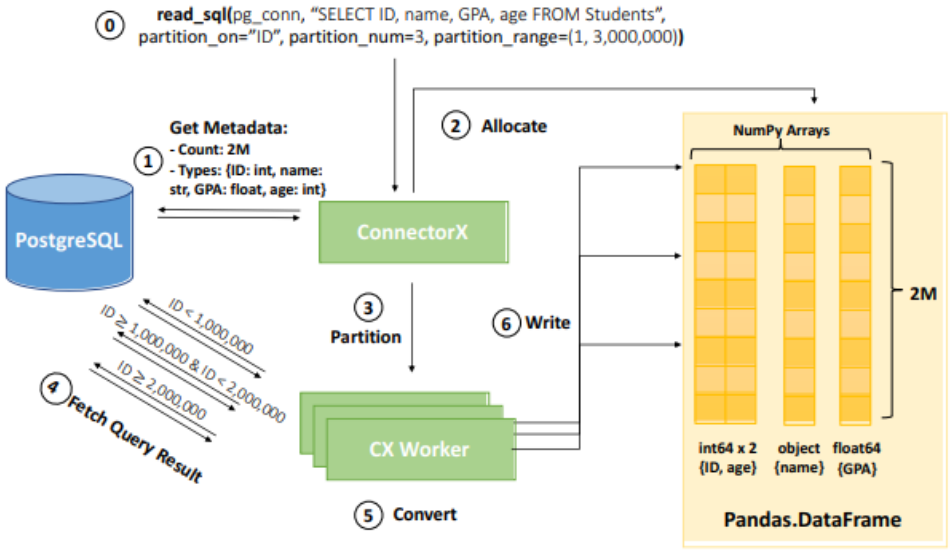

图 2：ConnectorX 的工作流程

在准备阶段，ConnectorX ① 会查询查询结果的元数据，包括行数和每列的数据类型。有了这些信息，ConnectorX 就会相应地分配 NumPy 数组，从而构建最终的 Pandas.DataFrame。为了充分利用客户端机器上的多个内核，ConnectorX 支持对查询进行分区，以便并行执行。

执行阶段以流式方式迭代进行。ConnectorX 将每个分区查询分配给一个专门的工作线程，工作线程并行地将部分查询结果从 DBMS 流式传输到数据帧中。具体来说，在工作线程的每次迭代中，它都会从 DBMS 抓取一小批查询结果，⑤ 将每个单元格转换为适当的数据格式，之后 ⑥ 将值直接写入数据帧。这一过程不断重复，直到工作线程耗尽查询结果。

（1）并行执行：如上所述，ConnectorX 利用查询分区实现并行执行。假设给定的查询以 Q 表示，用户指定了查询结果的范围分区方案，该方案由分区键、分区编号和分区范围组成。根据该方案，Q 可以被划分为一组子查询 $q_1, q_2, \ldots, q_n$。分区方案保证了 $q_1, q_2, \ldots, q_n$ 的子查询结果的联合等于 Q 的查询结果。因此，通过获取 $q_1, q_2, \ldots, q_n$ 的结果，ConnectorX 就能得到 Q 的结果。

图 2 第 0 步为分区方案：分区列 ID、分区编号 3 和分区范围（1, 3,000,000）。如果没有指定范围，ConnectorX 会通过查询 SELECT MIN(ID) 和 MAX(ID) FROM Students 自动设置范围。之后，ConnectorX 将范围平均划分为 3 个分区，

并生成 3 个子查询：

```
q1: SELECT ... FROM Students WHERE ID < 1,000,000
q2: SELECT ... FROM Students WHERE ID ∈ [1,000,000, 2,000,000)
q3: SELECT ... FROM Students WHERE ID ≥ 2,000,000
```

2.1.2 字符串分配优化

ConnectorX 预先分配了 NumPy 数组，以避免额外的数据复制。但是，字符串对象指向的缓冲区必须在知道每个值的实际长度后即时分配。此外，在 Python 中构造字符串对象并不是线程安全的。它需要获取 Python 全局解释器锁（GIL），当并行程度较高时，这会拖慢整个进程。为了减少这种开销，ConnectorX 在获取 GIL 时一次构造一批字符串，而不是单独分配每个字符串对象。为了缩短持有 GIL 的时间，在构建过程中不复制真实数据，而是在释放 GIL 后将字节写入分配的缓冲区。

假设查询结果包含 100 个字符串，每个字符串 10 字节。一种简单的方法是按需创建 Python 字符串对象。也就是说，对于从数据库驱动程序接收到的每个字符串，①获取 GIL；②分配一个 10 字节的 Python 字符串对象；③将内容复制到分配的缓冲区；④释放 GIL。与此不同的是，ConnectorX 将字符串字节暂时保存在内存中，并分批创建 Python 字符串。因此，ConnectorX 只需获取一次 GIL，而不是 100 次。此外，它还会通过交换步骤③和步骤④的顺序来提前释放锁，因为只有字符串分配才需要保持 GIL。因此，对 GIL 的竞争使用大减少了。

2.1.3 高效的数据表示

Python 的局限性使得需要一种更高效的数据表示方式。因此，研究团队使用本地编程语言来实现 ConnectorX，并选择了 Rust 语言，因为它能提供高效的性能并保证内存安全。此外，Rust 语言还为不同的数据库提供了多种高性能客户端驱动程序，ConnectorX 可以直接在此基础上进行开发。为了融入 Python 的数据科学生态系统，ConnectorX 提供了一个 Python 绑定和一个易于使用的 API。数据科学家可以使用 "pip install connectorx "下载 ConnectorX，并直接用 ConnectorX.read_sql 替换 Pandas.read_sql。

2.2 算法性能分析

2.2.1 消融实验

研究团队进行了一项消融实验，以深入了解 ConnectorX 的性能，并验证以下三种优化技术的有效性：1) 查询分区；2) 流工作；3) 字符串分配优化。改变 ConnectorX 的实现，并通过将 lineitem 表从 PostgreSQL 加载到 Pandas 来观

察性能变化，结果如图 3 所示。No-Streaming-Opt 表示禁用流工作；No-String-Opt 表示禁用字符串分配优化。就运行时间而言，ConnectorX（进行了所有优化）在有分区和无分区的情况下都是最快的。如果没有流工作，两种情况下的性能都会下降约 20%。在分区数量不同的情况下，字符串分配优化的影响也不同：在没有分区的情况下，字符串分配优化只降低了 6%，而在有分区的情况下则降低了 4.6 倍，这是因为分区越多，GIL 上的竞争使用就越多，从而导致进程变慢。在内存使用高峰期，分区对内存消耗影响不大。由于中间结果较大，No-Streaming-Opt 需要多 2.3 倍的内存。然而，与表 1 所示 Pandas 批次解决方案 95.6GB 的峰值内存使用量相比，No-Streaming-Opt（70.4GB）仍然节省了 20GB 以上的内存，这验证了使用 Rust 处理数据的高效性。

表 1：Pandas.read_sql 的内存分析

	Raw Bytes	Python Objects	Dataframe	Peak
PostgreSQL	12.4GB	52.6GB	24.4GB	95.6GB
MySQL	8.18GB	51.5GB	23.3GB[1]	99.1GB

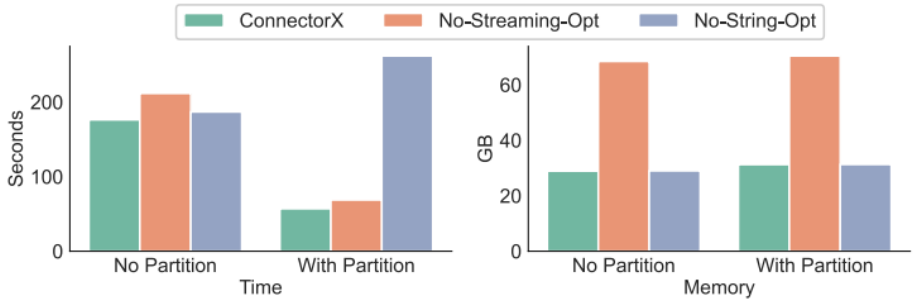

图 3：消融实验

2. 2. 2 性能比较

比较 ConnectorX 和四种将查询结果提取到 Pandas.DataFrame 中的基准模型。

1.内存比较：

图 5 评估了加载整个 TPC-H lineitem 表和 DDoS 表的峰值内存使用情况。Pandas-Chunk 为 Pandas 启用了分块处理（根据图 4，分块大小为 10K）。在 TPC-H 上，ConnectorX 和 PandasChunk 在所有 DBMS 上的内存消耗几乎相同。在客户端-服务器数据库上，它们的内存峰值始终比 Pandas 少 3 倍，在 SQLite 上少 2 倍。Dask 和 Modin 的结果与 Pandas 相似。Turbodbc 更节省内存，但仍比 ConnectorX 多出约 10GB 内存。在 DDoS 方面，ConnectorX 的表现远远优于其他基准模型，这是因为 ConnectorX 能够高效处理 DECIMAL

类型，DECIMAL 是 DDoS 中的主要类型，Python 将其扩大了 13 倍。另一个有趣的发现是，与 TPC-H 不同，ConnectorX 在 DDoS 上使用的内存比 Pandas-Chunk 少大约 2 倍。在 Pandas-Chunk 结束时连接分块数据帧时，NumPy 数组的内存将增加一倍，因为它们需要复制到一个更大的连续分块中。

图 4：通过改变块的大小来改变时间和内存

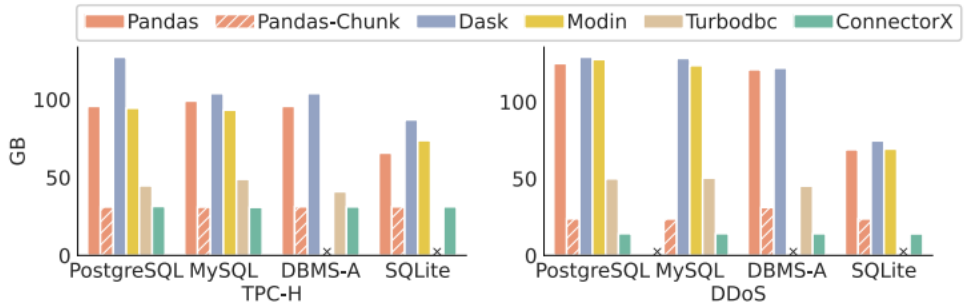

图 5：四种数据库系统的内存比较

2.速度比较：

图 6 为高带宽网络设置（10Gbit/s，SQLite 除外，它位于同一个客户端实例中）下的速度比较。为了与不支持查询分区的基准模型进行公平比较，左侧两图为 Modin、Dask 和 ConnectorX 在未应用分区时的结果；实验还测试了使用分区时的结果，以观察客户端实例在多核情况下的潜在速度提升（右侧两幅图）。

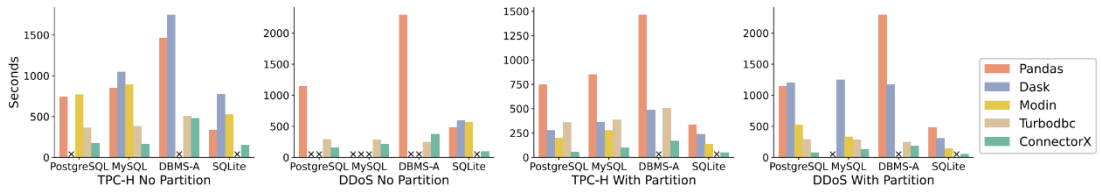

图 6：速度比较

（1）无分区：ConnectorX 几乎在所有情况下都表现最佳。在 TPC-H 上，PostgreSQL、MySQL、DBMS-A 和 SQLite 的性能分别比 Pandas 高出 4.2 倍、5.2 倍、3.0 倍和 2.3 倍；在 DDoS 上，PostgreSQL、DBMS-A 和 SQLite 的性能分别比 Pandas 高出 7.1 倍、6.0 倍和 5.2 倍。Modin 和 Dask 在从工作进程传输结果时有额外的开销。此外，由于内存不足问题，它们在很多情况下无法在未

应用分区时完成任务。与 ConnectorX 相比，Turbodbc 在 DBMS-A 上可以达到类似甚至更好的性能，但在 TPC-H / DDoS 上，PostgreSQL 和 MySQL 的速度分别为 2.0 倍/1.8 倍和 2.3 倍/1.3 倍。这种差异来自于 DBMS 的 ODBC 驱动程序的实现程度，这在很大程度上决定了 Turbodbc 的性能。

（2）有分区：ConnectorX 是所有实验中速度最快的。在基准模型中，只有 Dask 和 Modin 支持查询分区。为了更清楚地进行比较，将 Pandas 和 Turbodbc 在没有分区的情况下的结果复制到相同的图表中。使用分区后，ConnectorX 的速度进一步加快，在 TPC-H / DDoS 上分别比 Pandas 和 Turbodbc 快 12.8 倍/14.5 倍和 6.2 倍/3.8 倍；Modin 的速度也有所提高，但仍比 ConnectorX 慢 2.5 倍至 6.7 倍；Dask 也受益于分区，但稳定性较差，当达到内存限制时，它需要重新启动 Worker，这使得它比 ConnectorX 和 Modin 慢，甚至可能比 Pandas 慢。

2.3 分析结论

本文描述了 ConnectorX，这是一个快速和内存密集型的数据加载开源库，支持许多 DBMS(如 PostgreSQL、SQLite、MySQL、SQLServer、Oracle)到客户端数据帧(如 Pandas、PyArrow、modin、DASK 和 Polars)。

经过对开发人员常用的 Pandas.read_sql 函数进行了深入分析，研究团队发现了客户端执行的优化机会。开发 ConnectorX 旨在优化 read_sql 的客户端执行，而无需修改数据库服务器和客户端驱动程序的现有实现。ConnectorX 还为贡献者提供了模块化接口，以便轻松添加对更多 DBMS 和数据框架的支持。研究团队进一步确定了 ConnectorX 和其他库目前使用的客户端查询分区方法的缺点，并提出数据库系统应支持服务器端结果分区，以解决这些问题。

自 2021 年 4 月首次发布以来，ConnectorX 已被实际用户广泛采用，一年内累计下载量超过 30 万次，Github 星级超过 640 个。它已被应用于 ETL 中的数据提取和从 DBMS 中加载 ML 数据。它还被集成到 Polars、Modin 和 DataPrep 等流行的开源项目中。例如，Polars 是 Rust 中最流行的数据帧库，它使用 ConnectorX 作为从各种数据库读取数据的默认方式。实验表明，在加载大型查询结果时，ConnectorX 的性能明显优于现有库（Pandas、Dask、Modin、Turbodbc）。与 Pandas 相比，它的运行时间缩短了 13 倍，内存使用率降低了 3 倍。

示例演示[1]

①以最快和最节省内存的方式将数据从数据库加载到 Python 中：

```
1 import connectorx as cx
2 cx.read_sql("postgresql://username:password@server:port/database",
```

```
3               "SELECT * FROM lineitem")
```

②通过指定**分区字段(列)**来使用并行性加速数据加载：

```
1 import connectorx as cx
2 cx.read_sql("postgresql://username:password@server:port/database",
3               "SELECT * FROM lineitem",
4               partition_on="l_orderkey",
5               partition_num=10)
```

3 讨论和分析

3.1 算法的适用条件

ConnectorX 针对的是需要获取大型查询结果集的场景。它通过优化客户端执行，并通过并行化使网络和计算资源达到饱和，从而加速了这一过程。但是，当网络或 DBMS 上的查询执行成为关键限制因素时（例如，结果集较小的复杂查询），ConnectorX 带来的好处就会减少，有时甚至会因为获取元数据的开销而变得更慢。

3.2 算法存在的不足之处

ConnectorX 和其他库所采用的客户端查询分区方案可以通过更有效地利用高网络带宽和 CPU 资源来加速 read_sql。图 7 为 ConnectorX 在不同分区数量下的网络利用率。很明显，无分区根本无法使网络带宽达到饱和。如果有更多连接并行获取数据，就能更有效地利用带宽，从而获得更好的端到端性能（当带宽使用率降至 0 时，read_sql 终止）。

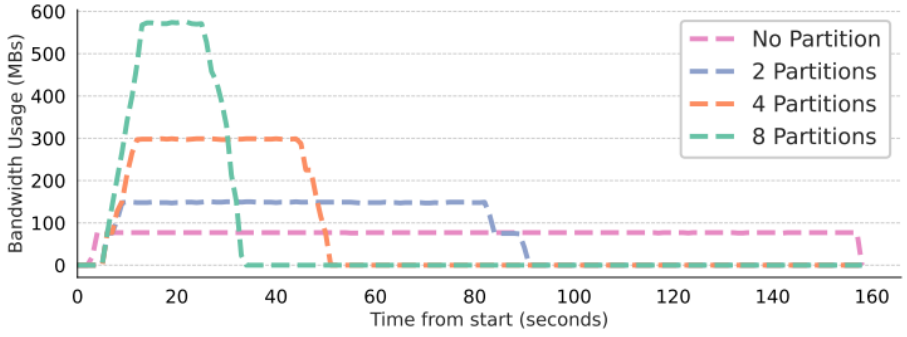

图 7：网络利用率的变化

这种方法之所以被客户端库广泛采用，是因为它不需要对数据库服务器进行任何修改。然而，它也有几个缺点：

（1）用户负担：要启用查询分区，用户必须额外花费大量时间和精力；

（2）负载不平衡：如果结果分区不均匀，可能会出现拖尾现象，从而影响整体性能；

（3）数据不一致：由于子查询是从独立的会话发送到服务器的，它们的结果可能来自不同的数据源；

（4）资源浪费：不同的子查询可能共享相同的代价高昂的子计划（例如全表扫描）。由于 DBMS 独立处理每个查询，子计划可能会重复执行多次。

3.1 算法可以改进之处

在数据库服务器端对查询进行分区可以解决上述问题。具体来说，数据库管理系统将查询结果划分为 n 个大小相等的分区，并允许客户端通过 n 个连接与现有协议并行获取这些分区。与客户端查询分区不同，服务器端结果分区不需要用户输入任何额外信息。在内部统计和成本估算器的帮助下，数据库管理系统有更好的机会对结果进行更均匀的分区。由于数据库管理系统掌握了在同一数据库快照上进行分区和执行的所有必要信息，因此它还能轻松保证数据一致性，避免资源浪费。

参考文献

[1] 邓旭东 HIT. (2021). connector-x | 让数据从 DB 高速导入到 DataFrame 中. [在线]. CSDN 博客. 取自: https://blog.csdn.net/weixin_38008864/article/details/119962735