软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(3):1027−1048 [doi: 10.13328/j.cnki.jos.006789]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向聚合查询的 Apache IoTDB 物理元数据管理*

赵东明[1], 邱圆辉[1], 康 瑞[1], 宋韶旭[1,2,3], 黄向东[1,2,3], 王建民[1,2,3]

[1](清华大学 软件学院, 北京 100084)

[2](大数据系统软件国家工程研究中心(清华大学), 北京 100084)

[3](北京信息科学与技术国家研究中心(清华大学), 北京 100084)

通信作者: 宋韶旭, E-mail: sxsong@tsinghua.edu.cn

摘 要: 时间序列数据在能源、制造、金融、气候等领域有着广泛应用, 聚合查询是相关分析场景中常见的查询需求, 快速获取海量数据的概要信息, 对于提高数据分析工作的效率具有重要意义. 通过存储元数据加速聚合查询是一种有效的提升聚合查询执行效率的手段, 但现有的时间序列数据库都使用时间窗口切分数据, 需要对数据进行实时排序和分区, 难以适应物联网场景下高并发、大吞吐量的数据写入特点. 因此, 提出了一种面向聚合查询的 Apache IoTDB 物理元数据管理方案. 该方案按照数据文件的物理存储特性切分数据, 并结合同步计算和异步计算策略, 优先保证数据的写入性能. 针对时间序列数据中普遍存在的乱序数据, 将时间范围重叠的一组文件抽象为乱序文件组并提供元数据, 聚合查询会被重写为 3 个结合物理元数据和原始数据的子查询高效执行. 多个数据集上的实验验证了该方案对聚合查询执行效率的提升效果以及不同计算策略对性能的影响.

关键词: 预聚合; 聚合查询; 查询重写; 物理元数据管理; 时间序列数据库

中图法分类号: TP311

中文引用格式: 赵东明, 邱圆辉, 康瑞, 宋韶旭, 黄向东, 王建民. 面向聚合查询的 Apache IoTDB 物理元数据管理. 软件学报, 2023, 34(3): 1027−1048. http://www.jos.org.cn/1000-9825/6789.htm

英文引用格式: Zhao DM, Qiu YH, Kang R, Song SX, Huang XD, Wang JM. Physical Metadata Management in Apache IoTDB for Aggregate Queries. Ruan Jian Xue Bao/Journal of Software, 2023, 34(3): 1027−1048 (in Chinese). http://www.jos.org.cn/1000-9825/6789.htm

Physical Metadata Management in Apache IoTDB for Aggregate Queries

ZHAO Dong-Ming[1], QIU Yuan-Hui[1], KANG Rui[1], SONG Shao-Xu[1,2,3], HUANG Xiang-Dong[1,2,3], WANG Jian-Min[1,2,3]

[1](School of Software, Tsinghua University, Beijing 100084, China)

[2](National Engineering Research Center for Big Data Software (Tsinghua University), Beijing 100084, China)

[3](Beijing National Research Center for Information Science and Technology (Tsinghua University), Beijing 100084, China)

Abstract: Timeseries data is widely used in energy, manufacturing, finance, climate and many other fields. Aggregate queries are quite common in timeseries data analysis scenarios to quickly obtain summary of massive data. It is an effective way to accelerating aggregate queries by storing metadata. However, most existing timeseries databases slice data with fixed time windows, which requires real-time sorting and partitioning. In IoT applications with high writing concurrency and throughput, these additional costs are unacceptable. This study proposes a physical metadata management solution in Apache IoTDB for accelerating aggregate queries, in which data are sliced according to the physical storage sharding of files. Both synchronous and asynchronous computing are adopted to ensure writing performance ahead of queries. Out-of-order data streams are another major challenge in IoTDB applications. This study abstracts files

* 基金项目: 国家自然科学基金(62072265, 62021002); 国家重点研发计划(2021YFB3300500, 2019YFB1705301, 2019YFB1707001); 北京信息科学与技术国家研究中心青年创新基金(BNR2022RC01011); 工信部 2020 年新兴平台软件项目

本文由“大数据治理的理论与技术”专题特约编辑杜小勇教授、杨晓春教授和童咏昕教授推荐.

收稿时间: 2022-05-15; 修改时间: 2022-07-29; 采用时间: 2022-09-23; jos 在线出版时间: 2022-10-27

with overlapping time ranges into out-of-order file groups and provides metadata for each group. Then aggregate queries will be rewritten into three sub-queries and efficiently executed on physical metadata and timeseries data. Experiments on various datasets have shown the improvement in performance of aggregate queries with the proposed solution, as well as the validity of different computing strategies.
Key words: pre-aggregation; aggregate query; query rewriting; physical metadata management; timeseries database

时间序列数据是某些指标或物理量随时间变化的采样值所组成的数据序列, 在能源、制造、金融、气候等领域有着广泛应用. 时间序列数据反映了相应指标在每一采样时间点的测量值, 随着物联网技术的蓬勃发展和应用场景的不断扩展, 各种设备和物品的实时数据通过网络共享, 并被用于监测运行状态、触发事件响应等, 数据中心收集到的海量历史数据被用于分析长周期的变化规律和发展趋势. 在时间序列数据的分析需求中, 聚合查询是常见的快速获取概要信息的手段, 如生产车间的报表中的产能、合格率等信息依赖于对生产和质量检测数据的求和汇总; 气象分析中历史同期的气温、降雨量等数据通常包含对历史数据求取平均值、最大最小值等操作; 数据中心服务器的使用状况看板需要对一段时间内各节点的负载情况求取平均值等. 数值型数据的最大最小值、总和、计数和平均值等聚合查询被广泛用于刻画数据的总体情况, 其查询性能的提升也有助于提高数据分析工作的效率.

聚合查询的加速手段在数据库领域有着广泛的研究[1], 常见的聚合函数具有可合并性, 即将数据切分为多个分片, 在其上分别计算的中间结果可以合并, 进而推算出全局数据的聚合结果. 其中, 第 1 步对数据切片并分别计算的过程需要访问数据并消耗大量计算资源, 而相应的加速手段可分为并行计算加速和存储元数据加速两类.

并行计算加速手段着眼于在聚合查询执行时对数据进行切分, 通过在各个分片之间并行计算, 充分发挥计算性能, 从而提升执行效率. PostgreSQL 的内置和自定义聚合查询函数、Spark 的自定义聚合函数和累加器均使用了类似的架构, 聚合查询将在各个集群节点之间、节点内部多个线程之间并行执行, 并逐层汇总至主节点完成计算, 它们的衍生产品[2]也具有类似的特性. 以上方法旨在提升即时计算的执行效率, 而存储元数据的加速手段则将数据分片的计算过程提前到实际查询发生之前, 并将中间结果存储为描述数据特征的元数据, 实际查询仅需利用元数据合并完成计算[3,4], 而不需要大规模读取原始数据, 关系型数据库和数据仓库中的概要表、时间序列数据库中的汇总查询等方法通过数据的逻辑特征预先切分数据, 与切分标准契合的聚合查询能够有效地利用存储的元数据加速计算, 对元数据的查询、访问和计算还能够使用索引进一步提升效率. 每条元数据所能代表的原始数据的条目, 根据实际情况从数千条到数十万条不等, 因此, 元数据加速聚合查询的效果显著优于并行计算, 并能在涉及相同数据的查询中反复使用, 保证更低的均摊计算成本.

现有的时间序列数据上使用元数据加速聚合查询的方法主要基于固定的时间窗口切分数据并预计算元数据, 其优势是元数据的检索和使用方法较为简单, 例如按天切分时间序列数据, 对某一自然月或自然年的数据执行的聚合查询只需检索相应天的元数据. 但是逻辑上的时间窗口依赖于数据的有序性, 而时间序列数据库的应用场景大多具有高并发、乱序到达的特点, 维持短期数据的有序性所需要的成本往往是不可接受的. 比如在车联网系统中, 车载传感器可以达到数百个, 对网络传输的质量要求较高, 而使用移动网络传输很容易受到网络状况影响, 城市中上下班通勤时段路上行驶车辆猛增, 大量数据并发高速写入数据库, 但同时, 车辆集中容易造成网络拥堵, 数据传输时断时续, 延迟达到的数据在维持有序性时需要频繁重写短期数据文件, 相应地, 基于时间窗口的预计算元数据也需要频繁更新.

基于数据库系统中数据的物理分片切分数据能够有效地解决以上问题. 数据库系统通常会对写入的数据进行物理分片, 划分为数据页或文件, 便于索引及修改. 通过将元数据关联物理分片, 在检索对应的原始数据时, 查询引擎即可利用数据库对物理分片的索引快速定位, 并且数据库会维持不同分片之间的相对均衡. Apache IoTDB[5]是一款面向工业物联网应用中时间序列数据收集、存储和分析需求的时间序列数据库, 在 IoTDB 中, 通过数据文件计算和维护物理元数据并支持聚合查询的主要难点有: (1) 关联数据文件并检测数据的变动, 及时生成和更新物理元数据; (2) 在高写入负载的场景下, 物理元数据的计算不能严重影响写入性能; (3) 乱序数据文件之间可能存在时间范围的重叠, 潜在的数据冲突使得对应的物理元数据不能直接使用, 聚

合查询重写需要结合物理元数据和原始数据以保证结果的正确性.

针对上述问题, 本文提出了一种面向聚合查询的 Apache IoTDB 物理元数据管理方案. 图 1 展示了所提方案在物联网时间序列数据生态中的位置, 所提方案与 IoTDB 引擎耦合, 利用有限的计算资源, 尽可能地保持物理元数据与原始数据的一致性, 并保存在外部元数据存储中. 用户在提交聚合查询时, 将自动利用物理元数据提高执行效率, 物联网中的数据采集设备和使用 IoTDB 引擎进行数据分析的应用都能无缝对接物理元数据管理框架. 本文的主要贡献有: (1) 提出了一种与 IoTDB 数据库耦合的面向聚合查询的物理元数据管理方案, 实验表明, 所提方案能够有效地提升聚合查询的执行效率; (2) 面向物联网应用场景高并发、乱序的数据特点, 设计了同步、异步和查询时计算这 3 种物理元数据计算策略, 能够在优先保证写入性能的前提下, 较好地保证元数据与原始数据的一致性; (3) 使用 ER 模型建模物理元数据与数据文件之间的关系, 将聚合查询重写为 3 个子查询, 结合元数据和原始数据, 高效准确地获得查询结果.

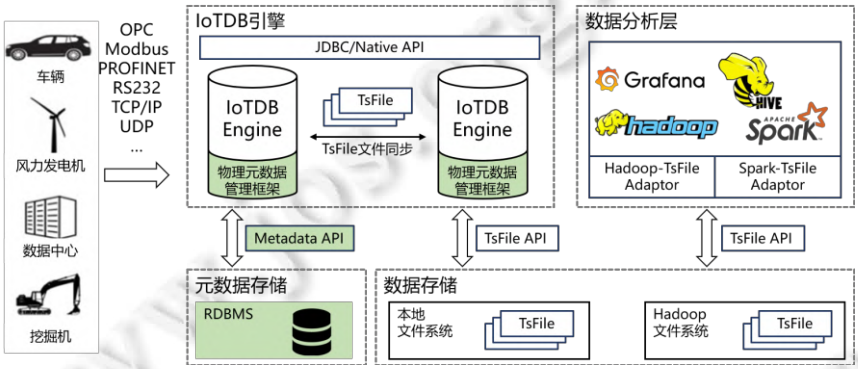


图 1 面向聚合查询的 Apache IoTDB 物理元数据管理方案框架图

本文第 1 节介绍在数据库和数据仓库领域元数据管理的相关工作. 第 2 节主要介绍理论概念和基础知识. 第 3 节详细阐释本文提出的面向聚合查询的 Apche IoTDB 物理元数据管理方案. 第 4 节通过实验验证所提方案的有效性和主要优势. 最后总结全文.

1 元数据管理相关工作

元数据在各类数据库和数据仓库系统中普遍存在, 主要应用于以下几种场景.

(1) 数据管理. 通过保存数据的逻辑模型、组织结构、数据类型等信息, 数据库能够对保存的数据进行约束和管理;

(2) 查询优化. 许多数据库会对数据进行画像, 记录不同数据的分布特征等统计信息, 以此作为评估查询计划代价的依据, 从而优化查询流程;

(3) 统计查询. 元数据能够直接提供数据的摘要信息, 辅助用户快速了解数据的重要特征, 也能够支持特定的聚合查询, 使得这些查询的效率得到显著提升.

本文主要涉及统计查询这一应用场景, 因此, 本节将主要讨论几种数据库和数据仓库中所用的元数据管理和查询技术, 并说明其使用场景和存在的问题.

1.1 概要表

得益于关系模型强大的表达力, 许多关系型数据库直接将大部分元数据以系统表的形式存储在数据库中, 如 MySQL、PostgreSQL、Oracle、SQLServer 等. 概要表是其中一类用于保存与实际数据相关的摘要信息、聚合结果等内容的系统表, 根据对应数据的列数, 又可以分为单列和多列概要表两类[6].

- 单列概要表记录了关系数据库中某一列数据的一些摘要信息, 通常包括数据类型、约束等逻辑结构信息和互异值数量、空值数量、近似分布直方图等数据摘要信息, 如 MySQL[7]的 COLUMN_STATISTICS

系统表和 PostgreSQL[8]的 pg_statistic 表. 这些系统表由数据库自动生成并定期维护, 用户也可以通过显式地调用 ANALYZE 或类似语法手动更新. 为了保证数据库的事务处理性能, 这部分元数据并不会与实际数据保持严格的一致性, 因此往往不被直接用于计算查询结果, 除非用户根据业务场景需求, 强调快速从数据库中获取摘要, 而对结果的一致性和准确性不敏感时, 手动指定从对应系统表中查询部分信息. 但它们在查询优化过程中具有重要作用, 互异值、空值计数等在 JOIN 操作中能够辅助代价模型估计操作代价, 优化实际执行流程[9], 近似分布直方图能够对 WHERE 中的筛选条件评估其选择度, 使查询优化器选择操作数据量最少的执行流程[10].

- 多列概要表是以某些列为基础对目标列进行分组聚合而形成的摘要信息, 可以视为使用 GROUP BY 子句为数据表创建的物化视图或者物理表[11,12]. 概要表的存在, 使得符合条件的聚合查询实际需要访问的数据行数大幅减少, 当查询筛选条件涉及的列是概要表的分组依据的子集时, 查询结果只需从概要表中直接读取, 或取某些行聚合得到. 例如对一张销售订单表进行分组聚合, 统计出任意地区在一个月内的营业总额, 得到如图 2 所示的概要表 Sales_Summary. 此后, 在处理某一地区的营业总额统计查询时, 执行器并不需要读取原始的 Sales 表, 而只需从概要表中进行筛选和计算. 概要表中每一行记录所代表的原始数据的行数反映了分组的粒度, 显然, 分组依据越少, 相应列的基数越小, 概要表粒度越粗, 在实际查询中被有效利用的概率越低. 比如图 2 中所示的概要表, 在查询 2020 年 1 月 6 日全北京市的营业总额时就无法被使用. 因此在实际应用中, 数据库可以针对一张数据表建立一组不同粒度的概要表, 并根据实际查询选择恰当的概要表提升执行效率.

orderID	province	year	month	day	time	price
8741	Beijing	2020	1	5	08:14:23	127
8742	Beijing	2020	1	6	11:32:46	31
8743	Tianjin	2020	1	6	15:16:29	64
8744	Beijing	2020	1	6	18:47:03	17
…	…	…	…	…	…	…

```
CREATE MATERIALIZED VIEW Sales_Summary AS
SELECT province, year, month, SUM(price) AS
turnover
FROM Sales
GROUP BY province, year, month;
```

province	year	month	turnover
Beijing	2020	1	24497
Beijing	2020	2	35068
Tianjin	2020	1	13532
Tianjin	2020	2	26573
…	…	…	…

图 2 概要表示例

在关系型数据库和数据仓库的应用中, 多列概要表被证明能够在实际查询中有效地提升查询效率, 收益远高于需要付出的预聚合计算代价和物化视图存储代价[13]. 但随着数据表列数的增加, 其概要表的可选组合会以指数形式增长, 一方面, 这会导致概要表占用空间的无限制增长和访问效率的降低, 因此, 复杂系统仅依赖领域知识和专家经验进行摘要表的构建是远远不够的[11]; 另一方面, 多个不同的概要表会导致同一个查询存在多种潜在的执行流程, 增加查询优化的难度. 但这并不影响类似的分组预计算思想被推广应用到更多数据分析平台中, 通过在特定的非关系型数据库中定义各自领域相关的数据分片方法, 诸如高维数组[14]、图像[15]、概率分布[16]等数据模型的分析工作也能使用概要表提升查询效率. Edara 等人[17]结合多列概要表中分组的思想和 BigQuery 分布式数据仓库的实际, 设计了基于数据物理分片的物理元数据 BigMetadata, 其结合分布式存储的物理特性, 将单列的概要信息在不同分片中各自计算, 不仅能够充分利用集群资源, 还提升了摘要表在包含筛选条件的复杂查询中的使用性能. 但是该方法使用分布式存储中的区块切分数据, 分片粒度较

粗, 仅适用于数据仓库场景, 且所有对已有区块的写入或修改操作均会导致整个区块的重写, 时间序列数据应用场景中乱序到达的数据将会导致频繁的区块重写和物理元数据的更新, 严重影响系统的整体性能.

1.2 汇总查询

在飞速发展的物联网应用场景下, 时间序列数据具有大体量、高并发、高吞吐量等特性, 但历史数据的信息密度相对较低, 因此, 聚合查询在时间序列数据分析工作中占有重要地位[5]. 在学术研究和工程实现中, 有多种聚合查询的优化方法和策略[4,18]: 关系型数据需要借助多列的相关关系或物理存储特性进行数据分片; 而时间序列数据本身在时间维度上有序, 借助时间窗口, 能够对数据进行有效的分组, 这种方法被称为汇总[19,20], 通常包含基于单条时间序列的降采样聚合和基于多条时间序列的分组聚合这两类技术(如图 3 所示).

- 降采样聚合将时间序列数据按照一定的时间区间长度进行分片, 并在每一个区间内, 使用特定的聚合函数给出汇总结果, 形成含有较少数据点的概要序列[21]. 一方面, 降采样聚合结果能够支持特定的聚合查询, 如在气象站采集实时气温的场景中, 查询过往一周或者一个月的每日平均气温是常见的历史数据分析需求, 此时, 通过对原始时间序列数据使用按天分片并求平均值的降采样查询, 数据库能够直接获得历史上每天的平均气温, 而不需要在查询时进行计算, 从而提升实际查询的执行效率; 另一方面, 降采样聚合获得的概要序列体量更小, 同时也包含了原始时间序列数据中的主要信息, 因此在对历史数据的精度要求不高的场景下, 使用概要序列替换原始数据能够显著减少历史数据的存储和分析压力[22].
- 分组聚合则是将多条时间序列数据中同一时间的数据点进行聚合并形成概要序列, 如在分布式集群的实时运行负载监测场景中, 通过对集群中每台服务器的实时负载求平均值, 数据库能够直接保存集群整体的负载情况. 简而言之, 降采样聚合是在时间序列内部沿时间维度进行聚合分析, 而分组聚合是在多条时间序列之间沿不同维度或指标进行聚合分析, 二者均会形成新的概要序列, 对该序列的持久化存储能够提升特定的实际查询性能, 同时, 这两种功能还可以相互协作, 满足更复杂的聚合查询需求, 如根据全市范围内多个气象监测站的气温历史数据, 对每天所有监测站的气温进行平均值汇总, 即可获得全市每天的平均气温.

seriesID	12:00	12:15	12:30	12:45	13:00	13:15	13:30	13:45
ts1	1	4	-3	8	2	-4	5	2
ts2	7	2	8	-9	4		1	1
ts3	9	3	-2	-1	6	3	8	2
ts4		2	5	2	8	5	-42	7

ROLLUP every 1h with **SUM** and **COUNT** for each series

seriesID	12:00	13:00
ts1 SUM	10	5
ts1 COUNT	4	4
ts2 SUM	8	6
ts2 COUNT	4	3
ts3 SUM	9	19
ts3 COUNT	4	4
ts4 SUM	9	16
ts5 COUNT	3	4

图 3 OpenTSDB 汇总表示例

汇总功能的设计依赖于在时间维度上对时间序列数据进行逻辑分片, 其优势是每个分片都对应具有实际意义的时间范围, 与实际查询的时间筛选条件和汇总查询需求更容易匹配. 但相应地, 由于逻辑上的时间范围与物理上的数据文件没有严格的对应关系, 在分布式环境中, 汇总功能并不能很好地利用集群资源. 如在高并发的时间序列数据采集场景中, 一段时间内的数据可能被分散存储在多个节点中, 因此, 汇总计算需要通过集群通信把数据传输到计算节点. 而基于时间序列数据在集群各个节点中的分布特性进行物理分片, 汇

总计算就能在文件存储节点上直接完成[1], 从而减轻集群的网络通信压力.

2 基础知识

2.1 时间序列数据上的聚合函数

时间序列数据中, 每一个数据点是由时间戳和其对应的数值组成的二元组, 记为 $x_t=\langle t,v\rangle$, 表示在时间 t 时刻传感器或其他设备采集到某项物理指标的值为 v. 一条时间序列或时间序列分片是一组数据点按照时间排列形成的有序集合, 记为 $X_{t_1:t_n}=[x_{t_1},x_{t_2},...,x_{t_n}]$, $t_1<t_2<...<t_n$, 从中选取 $m-1$ 个数据点 $t_1<t_{i_1}<t_{i_2}<...<t_{i_{m-1}}<t_n$, 即可将该时间序列切分为 m 条分片 $X_{t_1:t_{i_1}},X_{t_{i_1+1}:t_{i_2}},...,X_{t_{i_{m-1}}:t_n}$, 这些分片相互之间没有数据点的重复和遗漏, 按顺序拼接在一起, 与原始的时间序列是等效的, 记为 $X_{t_1:t_n}=X_{t_1:t_{i_1}}\cup X_{t_{i_1+1}:t_{i\,2}}\cup...\cup X_{t_{i_{m-1}+1}:t_n}$.

时间序列数据上的聚合函数是根据一个时间序列分片 $X_{ts:te}$ 中的所有数据计算出某一特定指标的算法, 最常见的聚合函数是针对数值类型的一系列统计函数, 如最大值、最小值、计数、求和、平均值、方差等. 尽管这些聚合函数必须读取全局数据才能得出正确结果, 但其在多个数据分片之间具有可合并性, 即通过在每个分片上计算得出的中间结果, 可以推算出全局数据的聚合结果. 简单的聚合函数如计数函数直接具备可合并性, 在任意两个数据分片上的计数值通过加和即可得到全局的计数结果; 更复杂的聚合函数如平均值函数不能直接合并, 但可以被分解为数据的总和除以计数的形式, 而求和与计数函数都具备可合并性, 表 1 列出了上述聚合函数的可合并性.

表 1 常见数值统计的聚合函数的可合并性

聚合函数	可合并性公式
最大值 max	$\max(X_{t_1:t_n})=\max\{\max(X_{t_1:t_{i_1}}),\max(X_{t_{i_1+1}:t_{i_2}}),...,\max(X_{t_{i_{m-1}+1}:t_n})\}$
最小值 min	$\min(X_{t_1:t_n})=\min\{\min(X_{t_1:t_{i_1}}),\min(X_{t_{i_1+1}:t_{i_2}}),...,\min(X_{t_{i_{m-1}+1}:t_n})\}$
计数 *count*	$count(X_{t_1:t_n})=sum\{count(X_{t_1:t_{i_1}}),count(X_{t_{i_1+1}:t_{i_2}}),...,count(X_{t_{i_{m-1}+1}:t_n})\}$
求和 *sum*	$sum(X_{t_1:t_n})=sum\{sum(X_{t_1:t_{i_1}}),sum(X_{t_{i_1+1}:t_{i_2}}),...,sum(X_{t_{i_{m-1}+1}:t_n})\}$
平均值 *avg*	$avg(X_{t_1:t_n})=\dfrac{sum(X_{t_1:t_n})}{count(X_{t_1:t_n})}$
方差 var	$\mathrm{var}(X_{t_1:t_n})=\dfrac{sum(X^2_{t_1:t_n})}{count(X_{t_1:t_n})}-(avg(X_{t_1:t_n}))^2$, $X^2_{t_1:t_n}=[x^2_{t_1},x^2_{t_2},...,x^2_{t_n}]$

基于可合并性, 我们可以将聚合函数分为 Compute-Merge-Finalize 这 3 个计算阶段: 在 Compute 阶段中, 数据会被分组, 形成一系列数据分片, 并在每一个分片上执行聚合操作, 将其中间结果存储在元数据中; 在 Merge 阶段中, 这些中间结果将被合并, 得到全量数据对应的查询中间结果; 并在 Finalize 阶段中转换为最终的聚合查询结果. 以平均值函数为例: 在 Compute 阶段, 每个分片将被分别计数与求和; 进而在 Merge 阶段, 分别对计数与总和进行合并; 在 Finalize 阶段, 使用全局数据的总和除以计数得到聚合函数的输出结果. Compute 阶段, 各个分片的计算相互独立, 能够充分利用并行化和分布式策略; 同时, 其计算过程能够预先完成并存储中间结果. 因此, 该模型能够显著提升实际聚合查询的执行效率. 类似的计算模型在 PostgreSQL 及其衍生的 TimeScaleDB、MapReduce 和 Spark 及其衍生产品[1,23−25]中均有实践.

2.2 TsFile文件格式

TsFile 是一种面向时间序列数据的文件存储格式, 在存储形式、编码压缩和读取查询方面做了许多针对性的优化[5]. 图 4 展示了 TsFile 的主要结构, 其中, 索引区与数据区分离, 单独保存在文件尾部, 避免了在数据连续读取时, 需要频繁跳过不需要的索引的问题. 数据区被划分为若干个 Chunk Group, 将逻辑上关联相对紧密的多条时间序列组织在一起, 如同一辆车上的不同传感器传回的数据, 便于管理和路径索引.

一条时间序列数据在 TsFile 中被分片组织在若干 Chunk 中, 每个 Chunk 仅包含一条序列在一段连续时间窗口内的数据, 通常情况下, 各个 Chunk 之间时间范围互不重叠且保持有序. Chunk 内部包含若干数据存储的

基本单元 Page, 这些 Page 之间严格保持数据的有序性, Page 内的数据点也按照时间戳有序存储. TsFile 文件内数据的有序性有助于在查询数据时快速对 Chunk, Page 进行筛选, 提升数据检索效率. 除此之外, TsFile 内各层级还会在头信息中保存局部数据的摘要信息, 如最大最小值、布隆过滤器等, 在处理与数值相关的筛选条件时, 能够快速跳过不符合条件的部分数据.

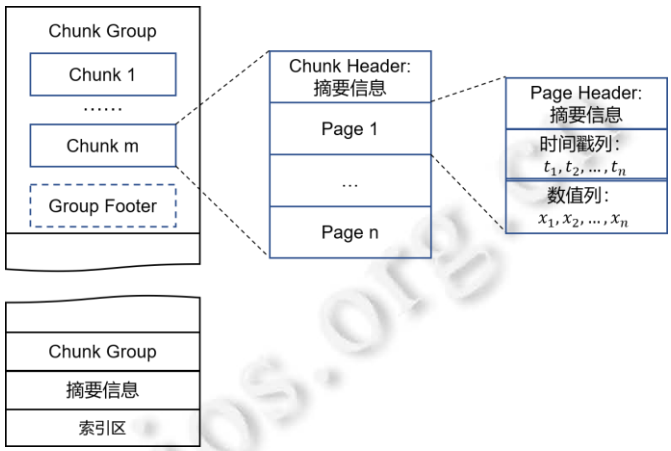


图 4 TsFile 文件格式示意图

在编码压缩方面, TsFile 将时间和值分别按列存储, 其中, 时间列使用二阶差分编码, 对物联网场景下周期性采集的传感器数据能够实现极高的压缩率, 值则可以根据实际数据的特征选择合适的编码压缩方法.

2.3 IoTDB的数据文件管理机制

为了提升服务器的数据写入性能, IoTDB 采用了类似于日志结构合并树(log structured merged tree, LSM-Tree)[26]的存储结构, 时间序列数据本身具有时间维度上的有序性, 在 LSM 树中按照数据点的时间戳有序组织的数据结构天然与查询需求相契合, 但 IoTDB 对其具体机制进行了部分修改, 用以适应时间序列数据采集和存储场景高并发、大吞吐量、乱序到达的写入特征.

时间序列数据中, 新的数据点写入后首先被存储在内存中的 MemTable 内, 为了尽可能地提升数据写入性能, 内存中的数据并不会严格保持有序性, 只在面向查询或刷写等使用需求时进行排序. 一段时间内的数据被缓存在内存中, 在完成排序后, 被批量刷写到磁盘上的列式存储文件 TsFile. 因此, 每个 TsFile 文件内部的数据是有序的, 并保存了时间范围索引用于快速检索和筛选数据.

文件之间数据的有序性有助于提高数据访问的效率, 但相应地, 在处理写入顺序与数据时间顺序严重不符的场景时, 需要付出额外的文件重写和排序代价. 因此, IoTDB 选择牺牲一定的读取性能, 将数据文件存储划分为有序空间和乱序空间两部分. 有序空间内各个数据文件之间保持有序性, 当新的数据文件与已有文件存在时间范围的重叠时, 新的文件将被转移至乱序空间中, 不再消耗额外的代价进行即时排序和整理. 以车联网应用场景为例, 上下班高峰期路上行驶车辆显著增多, 网络拥堵, 部分车辆的数据就可能出现如图 5 所示的“后发先至”的乱序情况, 即数据采集时间和送达接收的时间顺序不一致, 而数据的有序性必须以采集时间为准. 加之数据写入负载高, 数据库频繁将旧数据落盘持久化, 09:00 之前的历史数据已落盘的情况下, 延迟送达的 08:45 的数据点即被暂时保存在乱序空间中. 最终, 一整天的车联网数据在 IoTDB 中形成的数据文件可能如图 6 所示, 尽管大部分数据都是有序的, 但受到传输条件和写入负载的双重影响, 上下班高峰时段形成了乱序数据文件.

LSM 树中分层合并的机制也被相互隔离地实现在有序空间和乱序空间内, 用于将散碎的小文件收集整理为下一层的大文件, 提升数据检索和访问的效率. 特别地, IoTDB 使用跨空间的合并策略来保证数据文件的最终有序性, 即从两个空间中选择存在时间范围重叠的文件, 读取数据进行排序和文件重写. 乱序空间的存在,

使得对应于文件的物理元数据不能简单地进行合并操作，而必须充分考虑数据文件间的时间范围重叠情况.

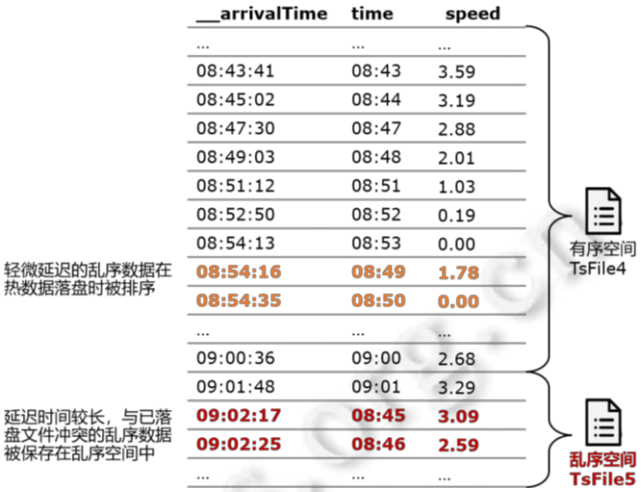


图 5 车联网应用场景中的乱序现象和乱序数据文件的产生示意图

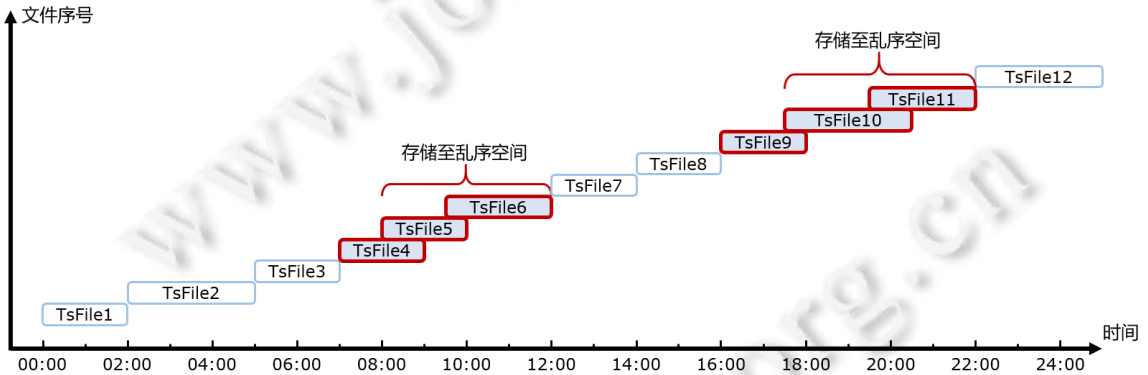


图 6 IoTDB 中的乱序文件和乱序空间的示意图

数据删除在时间序列数据的应用场景中是相对少用的操作，通常仅用于纠正部分错误数据或删除过于陈旧的历史数据. 因此, IoTDB 采用了懒删除策略以避免数据文件的频繁重写. 删除操作以日志的形式保存在数据文件附带的修改记录文件(modification file)中[5]，在读取操作中，过滤掉已经被标记删除的数据点，删除记录文件仅在文件合并时参与修改数据，无效点不会被写入到合并后的文件中.

3 基于 Apache IoTDB 的物理元数据管理框架实现方案

本文所提出的物理元数据管理方案由 3 个部分组成，其模块图如图 7 所示.

- 物理元数据存储引擎负责元数据的存储和更新，并提供高效的数据查询和读取接口. 在具体实现中，我们分别使用了基于数据文件和关系型数据库的两种存储方法.
- 物理元数据预计算引擎负责根据计算和更新物理元数据，保持与原始数据的一致性，共包含 3 种计算策略，其中，同步计算策略与 IoTDB 存储引擎的数据文件操作关联，在数据变动时，尽可能地保持物理元数据的一致性；异步计算策略通过定时任务，利用闲时资源更新缺失或过时的元数据；查询时计算策略则将查询执行过程中计算的部分中间结果更新到元数据中. 3 种策略相互协作，在优先保证数据库写入性能的同时，保证物理元数据与原始数据的最终一致性.
- 物理元数据聚合查询执行器负责回答输入的聚合查询，原始查询被重写为 3 个子查询，通过检索物理

元数据合并计算提升执行效率, 并结合原始数据确保结果的正确性.

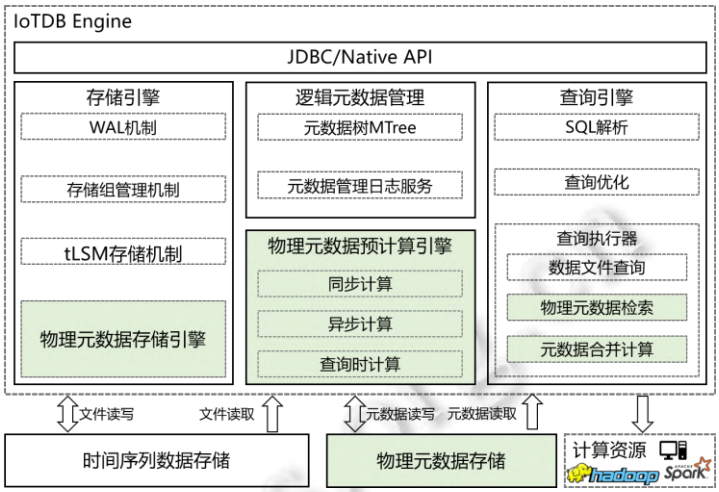


图 7 面向聚合查询的 Apache IoTDB 物理元数据管理框架图

以下将分别介绍 3 个模块的设计细节.

3.1 存储引擎

存储引擎旨在持久化存储所有预计算的元数据, 为查询引擎提供查询优化支持. 存储引擎的实体关系模型[27]如图 8 所示, 该关系模型由两个概要实体 Series, File 和两个统计实体 FileSeriesStat, FileGroupSeriesStat 组成, 其中, Series 实体对应 IoTDB 中时间序列的树状数据模型, 记录了所有时间序列在数据库中的逻辑路径; File 实体对应 TsFile 数据文件, 记录了文件的存储路径以及版本号, 版本号是用来标识文件在合并、更新等操作中的变动. 一条时间序列的数据按照 LSM 树往往被切片存储在多份数据文件中, 同时, 每份数据文件集中存储了同一设备下不同传感器对应的多条时间序列数据. 因此, Series 和 File 实体的关系才能唯一确定一个时间序列数据分片.

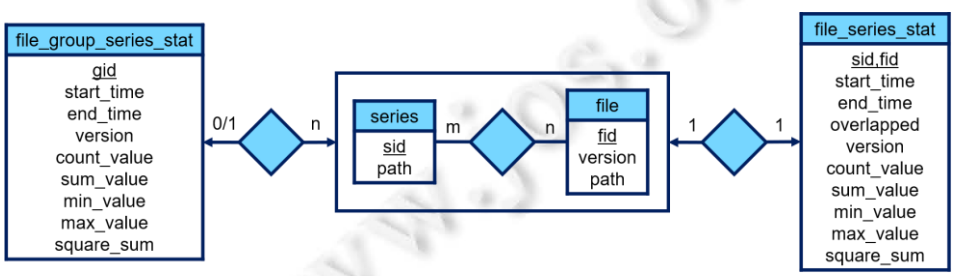


图 8 存储引擎 ER 图

FileSeriesStat 统计实体与时间序列数据分片一一对应, 存储了分片的时间范围信息和物理元数据, 其中, 时间范围信息包含分片的起始时间和终止时间, 以及是否与其他数据文件中的分片存在时间范围的重叠; 物理元数据保存了 5 个数值统计量, 用于支持最大最小值、计数、求和、平均值和方差这些聚合函数, 以及完成计算时 File 实体的版本号, 用于在查询中校验物理元数据与实际数据的一致性.

FileGroupSeriesStat 统计实体对应时间范围相互重叠的一组数据文件抽象出的乱序文件组, 与数据分片的一对多关系记录了乱序文件组与该组数据文件中的数据分片的对应关系. 与 FileSeriesStat 统计实体类似, FileGroupSeriesStat 统计实体存储了乱序文件组的起止时间范围和相应的元数据, 其中, 时间范围是该组数据文件时间范围的并集.

在存储引擎的设计中，为了保证查询的正确性，两个概要表的记录以及两个统计表中的起止时间戳始终与原始数据保持强一致性，任何时刻，其与原始数据文件的信息都是保持一致的. 同时，为了保证查询的高效性，两个统计实体中的统计量信息只保证与原始数据的最终一致性，数据库中的统计信息不一定与实际的数据文件统计信息一致. 聚合查询在使用统计量之前需要先进行统计量版本号的校验，当统计量的版本号与对应数据文件的版本号不一致时，需要先解决数据冲突，更新存储引擎中的元数据后，再利用统计量进行查询.

基于以上存储模型，本文实现了基于 TsFile 文件和基于关系型数据库的两种存储方案.

3.1.1 基于文件的元数据存储方案

基于文件的元数据存储方案将元数据与原始数据一同存储在 TsFile 文件中，在文件内部的头信息中保存了相应时间序列数据分片的起止时间以及最大值、最小值、计数和总和的统计信息. 在读取数据文件时，文件头被首先读取，并用于校验是否满足时间范围和值筛选条件；随后，在聚合查询中使用元数据进行合并计算查询结果[28,29]，而不需要完整读取内部数据.

基于文件的物理元数据存储和查询方案虽然能够通过元数据显著提升聚合查询的执行效率，并在文件格式中具有自解析性，但也存在明显的不足之处：物理元数据的结构与 TsFile 文件格式绑定，这虽然在减少磁盘 I/O、提升读取速度上有一定优势，却也限制了其自身的更新和功能扩展. 一方面，由于 IoTDB 的懒删除策略，数据文件在发生删除操作时并不会即时修改，因此，存储的元数据也无法更新从而不能保持与数据的一致性；另一方面，TsFile 文件格式已经被预先定义，用户无法在不深入理解和大规模改动源代码的基础上增加新的元数据条目，以满足不同应用场景的聚合查询需求.

3.1.2 基于关系型数据库的元数据存储方案

利用预计算的元数据来加速查询过程的关键在于保证元数据与原始数据的一致性，而关系型数据库的 ACID 特性很好地弥补了上述方案在保持元数据一致性方面的不足. 同时，考虑到元数据自身的读写性能，本文使用轻量级关系型数据库 SQLite3 实现了基于关系型数据库的元数据存储方案.

元数据与原始数据的分开存储，使得元数据的更新不再依赖于 TsFile 文件的改动. SQLite3 中，元数据信息可以在原始数据有变更时进行动态更新，从而保证了元数据的可用性，减少了查询过程中由于元数据不可用读取原始数据带来的性能损失. 同时，关系型数据库和 SQL 查询语言的特性也增强了元数据信息的可扩展性，用户只需通过 DDL 和 DML 即可向数据库中添加新的元数据条目.

3.2 物理元数据预计算引擎

本文在物理元数据预计算引擎中设计了 3 种计算策略，在物理元数据的一致性和聚合查询的执行效率之间取得平衡：同步计算策略在数据文件操作的同时对物理元数据进行更新；但在计算资源有限的条件下，部分同步计算任务无法完成会导致物理元数据缺失或过时，异步计算策略将通过定时任务扫描并更新这些元数据，更好地利用闲时计算资源提高元数据的可用率；查询时计算策略则利用执行查询需要的计算资源将中间结果写回存储引擎，在保证查询效率和结果正确性的同时，更新缺失或过时的物理元数据，加速后续涉及相同数据的聚合查询.

在系统实现中，为了限制物理元数据预计算引擎所用的计算资源，同步和异步计算策略共享线程池和任务队列. 在数据集中大量写入时，资源会被占满而无法接收新数据的物理元数据预计算任务，由此造成的物理元数据缺失或过时问题将由异步计算策略在系统负载低谷时提交计算任务. 查询时计算策略则单独占用查询执行器的计算资源，当同步和异步计算策略均无法在实际查询到达前完成物理元数据的计算和更新时，查询时计算策略是对结果正确性的最终保障手段. 物理元数据预计算引擎能够在系统后台自动地完成资源的分配和 3 种计算策略之间的协调，通常只需由数据库管理员根据实际情况设定引擎所能使用的资源上限，或在确有必要时手动提交某些时间序列的物理元数据异步计算任务，而用户在写入和查询时可以做到几乎无感.

3.2.1 同步计算策略

同步计算策略旨在实时保持物理元数据与实际数据的一致性，因此需要在数据文件发生变动时即时触发计算任务. IoTDB 中共有 3 类数据文件操作，如图 9 所示，内存中数据落盘生成新的 TsFile 文件、数据删除操

作写入修改记录文件以及多个数据文件合并为新的文件. 预计算引擎会相应地完成物理元数据的计算. 以下将分别说明 3 类文件操作对应物理元数据的具体计算和变更.

- 文件刷写操作发生时, 对应的数据仍然驻留在内存中, 预计算引擎将基于内存中的数据完成物理元数据的计算, 通过对数据按照时间进行排序, 时间序列数据分片的起止时间能够被直接获取, 而 5 项数值统计信息则可以使用流式算法在对数据的一次遍历中求出. 预计算引擎将对一份数据文件中包含的多条时间序列分别计算, 并将其物理元数据写入到存储引擎中.
- 由于物理元数据中含有与数值有关的统计信息, 因此在数据删除操作中, 同步计算模块必须重新读取对应的数据文件, 检索实际被删除的数据点, 并更新物理元数据. 其中, 计数、总和及平方和这 3 项具有增量更新的特性, 即只需从数值中减去被删除数据点的对应贡献即可完成更新, 因此在更新过程中, 模块并不需要完全扫描数据文件, 而是借助区块、页的索引快速定位到被删除数据所在位置, 仅读取对应的数据点, 对物理元数据进行增量更新. 最大值和最小值不具备以上特性, 尽管在被删除数据中不含最大值和最小值时不需要更新, 但在其他情况下, 均需要重新读取未被删除的数据点进行计算.
- 物理元数据的各项统计信息具有可合并性, 因此, 多个数据文件合并时通常其元数据也可直接合并, 不需要读取任何原始数据即可完成增量更新. 但受限于 IoTDB 对乱序文件的管理机制, 当待合并文件中存在乱序文件时, 其多份文件之间可能存在数据冲突, 因此无法直接合并其物理元数据, 而必须在读取所有数据消除冲突完成合并后, 根据新的文件重新计算.

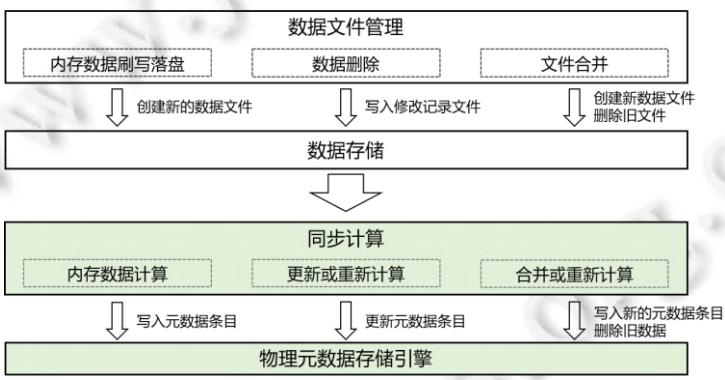


图 9 同步计算策略功能模块图

3.2.2 异步计算策略

同步计算策略牺牲了一定的数据写入和修改性能, 以换取物理元数据的一致性和可用率的最大化. 但在写入压力极大的场景下, 预计算引擎挤占计算资源可能阻塞写入线程池, 甚至导致系统崩溃. 异步计算策略则通过限制计算物理元数据所占用的资源, 降低对写入性能的负面影响, 并使用周期性扫描和更新来维持物理元数据和实际数据的最终一致性. 定时任务首先从存储引擎中获取版本号与数据文件不一致的元数据条目, 并按照对应的文件进行分组, 使用的 SQL 语句如下.

SELECT *fid*, *file_path*, *series_path*
FROM (*file_series_stat* **NATURAL JOIN** *file*) **NATURAL JOIN** *series*
WHERE *file_series_stat.version*!=*file.version*
ORDER BY *fid*;

随后, 对每一份文件的物理元数据更新过程将分别创建计算任务. 各任务之间相互独立, 因此能够在线程池中并行执行; 或在集群环境中利用分布式计算资源, 快速完成批量数据文件的物理元数据计算任务.

异步计算策略使用数据文件和对应修改记录文件的字节数作为版本号, 以此发现数据删除和文件合并带来的数据变更. 在同一份数据文件上的修改仅涉及 2 份文件的追加写入操作, 因此, 字节数作为版本号能够有

效地检测原始数据的变更. 不同于同步计算策略的触发器模式, 异步计算策略无法获取数据变更的原因, 如具体删除的数据范围、被合并的原始文件等信息, 因此必须通过读取最新的数据文件重新计算物理元数据.

3.2.3 查询时计算策略

查询时计算策略旨在更新查询过程中涉及的与原始数据不一致的元数据信息. 对于查询所涉及的数据文件, 查询引擎使用 SQL 语句检索所有满足查询条件且元数据版本与文件版本不一致的数据文件, 并交由预计算引擎进行计算和更新. 查询时计算策略既保证了在当前查询执行时物理元数据表中所有条目均与数据文件保持一致, 降低了查询重写的难度, 又有助于提升后续涉及相关文件的聚合查询的执行效率.

SELECT *file.file_path*
FROM (*file_series_stat* **NATURAL JOIN** *file*) **NATURAL JOIN** *series*
WHERE *series_path*='*root.beijing.north*401.*vehicle*'
  AND *start_time*≥*unixepoch*('2022-01-01T00:00:00')
  AND *end_time*≤*unixepoch*('2022-01-01T23:59:59')
  AND *file_series_stat.version*!=*file.version*;

乱序文件组涉及多份时间范围前后重叠的乱序文件, 组内可能含有未被当前查询所覆盖的数据, 比如在图 6 所示的数据文件上对 00:00–17:00 时间范围进行聚合查询, 完全选出乱序文件组 TsFile9-11 并更新其文件组的元数据, 会消耗额外的资源读取 17:00–22:00 时间范围内的数据. 因此, 查询时计算策略仅在查询重写完成后, 在实际计算中对被完全覆盖的部分乱序文件组的元数据进行更新. 例如在对 00:00–17:00 的数据进行聚合查询的过程中, 乱序文件组 TsFile4-6 的时间范围被完全覆盖, 因此其元数据会被校验版本并更新, 而 TsFile9-11 的元数据则不会被选中或计算. 查询首次覆盖乱序文件组时, 由于其元数据的缺失, 查询执行器必须重新读取所有数据文件, 消除潜在的数据冲突, 并使用流式算法从原始数据计算得到聚合查询的中间结果, 查询时计算策略即将这些中间结果写回元数据表中, 后续同样覆盖该乱序文件组的聚合查询即可直接读取该中间结果, 而不需要再一次读取数据文件.

3.3 聚合查询重写

结合存储引擎中已有的元数据条目, 查询执行器能够将特定的聚合查询重写为结合元数据和原始数据的查询, 充分利用预计算的结果加速查询效率. 聚合查询的执行流程如图 10 所示. 查询引擎在接收到语句输入后, 对其进行解析和查询优化, 匹配物理元数据聚合查询执行器的语句需满足两点条件: 一是包含对单个时间序列执行已声明的聚合函数的查询子句, 框架内现有的聚合函数主要涵盖了数值类型的常见统计分析函数, 包括最大最小值、计数、求和、平均值和方差; 二是仅含有时间范围筛选条件, 数值筛选条件、多序列之间运算等将破坏物理元数据的可用性. 完成查询匹配后, 查询执行器将首先使用查询时计算策略检索所有可能含有被查询数据的文件, 并更新对应缺失或过时的物理元数据, 这保证了后续执行过程中所有物理元数据都是准确、可用的, 有助于简化查询流程.

在查询实际执行过程中, 理想情况下, 查询执行器只需从物理元数据表中查找符合序列和时间范围条件的元数据条目, 进行合并和计算就能获取最终的聚合查询结果. 但是 IoTDB 中乱序文件的存在, 使得对应的物理元数据无法直接使用, 因此执行器根据数据文件的有序性和元数据信息将中间结果分为 3 部分: (1) 从乱序文件组的元数据表 *file_group_series_stat* 中查到的元数据条目; (2) 从物理元数据表 file_series_stat 中查到的有序文件的物理元数据; (3) 其他通过读取原始数据计算得到的聚合结果. 3 个子查询获得的中间结果经过合并和计算得到最终的查询结果, 不同聚合函数对中间结果的处理方式参见表 1 的可合并性公式.

以下将以查询北京市某路口一天内通过的车辆总数的聚合查询为例, 分别阐明 3 个子查询的重写原理及在样例查询上的重写结果. 该查询所涉及的数据文件及其覆盖的时间范围如图 11 所示, 其中, 底纹突出显示的文件为乱序文件, 其余为有序文件, 框线为虚线表示该文件在查询输入时, 其物理元数据存在缺失或过时问题.

SELECT *count*(*vehicle*) **FROM** *root.beijing.north*401

WHERE *time*≥2022-01-01T00:00:00 **AND** *time*≤2022-01-01T23:59:59;

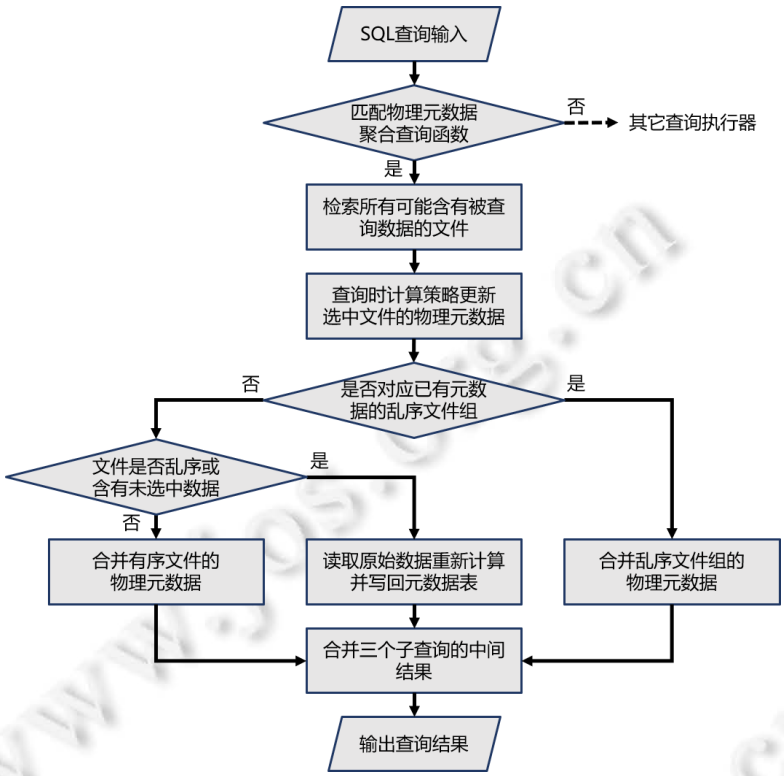


图 10　物理元数据聚合查询匹配和执行流程图

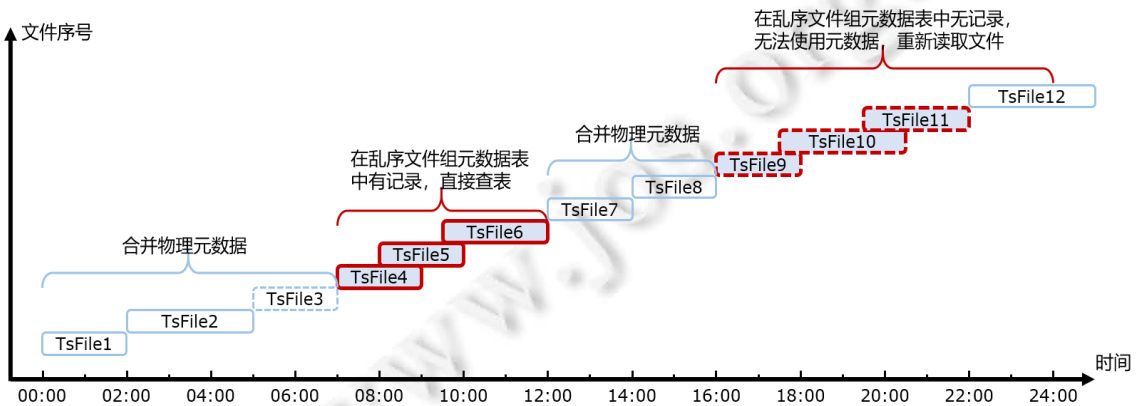


图 11　查询所涉及的数据文件及其时间范围示意图

3.3.1　乱序文件组元数据子查询

乱序文件组表示数据时间范围相互存在交集的一组乱序文件，比如图 11 中的 TsFile4-6 和 TsFile9-11. 乱序文件之间潜在的数据冲突使得其在一般查询时必须重新读取原始数据文件，排除冲突得到正确的时间序列，进而完成计算. 因此，在历史数据文件保持稳定的情况下，乱序文件组的元数据能够减少实际读取的数据量. 乱序文件组与数据文件之间存在一对多关系，根据查询到的文件列表匹配已有的乱序文件组存在一定困难，因此查询执行器使用时间范围进行匹配，从元数据表中选取时间范围被查询语句的筛选条件完全覆盖的乱序文件组，对其元数据进行合并得到子查询结果，样例查询重写后的子查询 SQL 语句如下.

```
WITH selected_group_stat AS (
  SELECT gid, start_time, end_time, count_value
  FROM file_group_stat
  WHERE sid IN (SELECT sid FROM series WHERE series_path='root.beijing.north401.vehicle')
    AND start_time≥unixepoch('2022-01-01T00:00:00')
    AND end_time≤unixepoch('2022-01-01T23:59:59')
  ORDER BY start_time, end_time)
SELECT sum(count_value) FROM selected_group_stat;
```

3.3.2 有序文件物理元数据子查询

乱序文件组的元数据仅在查询时计算策略中进行更新, 从未被历史查询完全覆盖的乱序文件组存在元数据缺失问题, 比如 TsFile9-11, 因此, 即使排除掉在乱序文件组元数据子查询中已匹配到乱序文件组的数据文件, 其余文件之间仍然可能存在时间范围的重叠, 查询执行器必须对其有序性进行辨别. 有序文件之间不存在数据冲突, 其物理元数据可以直接合并, 但是查询语句时间筛选条件边界附近的文件可能含有不符合筛选条件的数据, 比如图 11 中的 TsFile12 即含有 24:00 之后的数据, 因此, 图 10 查询流程中只有同时满足有序且不含有未选中数据的文件才会被有序文件物理元数据子查询选中, 并合并其元数据得到中间结果.

有序文件物理元数据子查询的 SQL 语句如下.

```
WITH selected_stat AS (
  SELECT fid, start_time, end_time, count_value, (
    overlapped OR start_time<unixepoch('2022-01-01T00:00:00')
    OR end_time>unixepoch('2022-01-31T23:59:59')) AS overlapped
  FROM file_series_stat
  WHERE sid IN (SELECT sid FROM series WHERE series_path='root.beijing.north401.vehicle')
    AND start_time≤unixepoch('2022-01-31T23:59:59')
    AND end_time≥unixepoch('2022-01-01T00:00:00')
    AND fid NOT IN (SELECT fid FROM file_group
      WHERE gid IN (SELECT gid FROM selected_group_stat)
 ORDER BY start_time, end_time)
SELECT sum(count_value) FROM selected_stat WHERE overlapped=FALSE;
```

特别地, 含有不符合筛选条件的数据文件, 其 *overlapped* 列也被设置为 TRUE, 这里 overlapped 表示其物理元数据不能被直接合并, 在后续处理中与乱序文件相同, 均需要读取原始数据完成计算, 这有助于保持后续操作逻辑的一致性.

3.3.3 原始数据聚合子查询

其他乱序文件及被时间筛选条件切分的数据文件需要重新读取原始数据完成计算, 这一子查询将被重写为 IoTDB 上的聚合查询, 其时间筛选条件为所需文件的时间范围的并集, 如 TsFile9-12 对应数据的时间范围应为 16:00-24:00. 乱序文件组时间范围的重叠区间合并问题可以使用排序和遍历操作解决, 将所有数据文件按照起始时间正序排列, 则重叠现象仅发生在相邻的时间范围之间, 亦即同一乱序文件组中的文件一定相邻且连续排列. 对所有文件按顺序遍历, 当前文件乱序但上一份文件有序说明一个乱序文件组的第 1 份文件被发现; 反之, 当前文件有序但上一份文件乱序说明当前乱序文件组的文件已被完全发现. 合并后, 时间范围的起始时间为第 1 份文件的起始时间, 终止时间为最后一份乱序文件的终止时间. 查询执行器利用合并后的时间范围, 即可生成对应原始数据聚合子查询的时间范围筛选条件.

以上重叠区间合并问题的算法可以通过 SQL 中的窗口函数和 GROUP BY 功能实现, 对所有选中的文件按照起始时间和终止时间排序后, 使用窗口函数访问前一行并判断 *overlapped* 是否与当前文件相同. 在不同

时, 将 *changed* 列设置为 1, 并在序号列 *range_index* 上增加 1. 这使得一组乱序文件的第 1 份文件, 如 TsFile9, 和之后的第 1 份有序文件的 *changed* 列被标记为 1, 因此, 该组文件的 *range_index* 值相同, 且与前后的有序文件各相差 1. 例如, 图 11 中文件的查询中间结果见表 2, 其中, 待合并的 TsFile9-12 对应的序号为 2.

表 2 图 11 中文件利用 SQL 合并重叠时间范围的中间结果

文件序号	*overlapped*	*changed*	*range_index*
TsFile1	false	1	1
TsFile2	false	0	1
TsFile3	false	0	1
TsFile7	false	0	1
TsFile8	false	0	1
TsFile9	true	1	2
TsFile10	true	0	2
TsFile11	true	0	2
TsFile12	true	0	2

```
WITH selected_range_index AS (
  SELECT fid, start_time, end_time, overlapped, sum(changed) OVER (
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS range_index
  FROM (
    SELECT fid, start_time, end_time, count_value, overlapped,
      overlapped!=lag(overlapped) OVER (
        ORDER BY start_time, end_time ROWS 1 PRECEDING) AS changed
    FROM selected_stat));
```

在以上中间结果中, 乱序文件组的一组文件 *overlapped* 列为 TRUE 且序号值相同, 因此其时间范围的并集可以通过分组聚合分别求取起始时间的最小值和终止时间的最大值获得, 从而在 IoTDB 中使用时间筛选条件对每个范围进行聚合查询. 这一过程可以形式化地用 SQL 语句表示为如下查询, 如果存在多个范围, 则应将多次 IoTDB 上的聚合查询结果求和汇总.

```
WITH selected_range_stat AS (
  SELECT range_index, overlapped,
    max(min(start_time), unixepoch('2022-01-01T00:00:00')) AS range_start,
    min(max(end_time), unixepoch('2022-01-31T23:59:59')) AS range_end,
  FROM selected_indexed_stat
GROUP BY range_index HAVING overlapped=TRUE)
SELECT count(vehicle) FROM root.beijing.north401 WHERE time≥range_start AND time≤range_end;
```

4 实验分析

4.1 实验设计

我们在 4 个数据集上验证所提方案的有效性. 表 3 给出了数据集的基本信息. 其中, HAR 数据集由 Stisen 等人[30]收集. 他们使用智能手机和手表采集了 9 位志愿者在 6 种不同活动过程中的运动信息, 每个数据点包括三轴加速度和陀螺仪采集到的 6 维信息; 中船数据集是 3 艘轮船在航行测试中船上系统采集的运行数据, 本实验选取了其中 27 处温度传感器的测量结果; 中车数据集是来自铁路某局的部分列车运行数据, 每辆列车仅在运行时周期性地采集相关数据, 实验中选取了其中 4 065 辆列车在一段时间内的有效数据; Gaussian 数据集是人工生成的数据集, 所有数据是从均值为 0、标准差为 100 的正态分布中随机采样得到的结果. 为提高聚合查询所能使用的数据量, 3 份真实数据集在使用中均将多列数据融合为单一的时间序列.

本文使用 Apache IoTDB 0.13.0 版本作为实验平台, 在查询性能实验中, 本文还使用了另外两款时间序列

数据库 InfluxDB 2.2.0 和 TimescaleDB 2.7.2 作为对比, 其中, InfluxDB 是一款使用 Go 语言编写的高性能时间序列数据库, 其内置的聚合函数采用流式算法实现; TimescaleDB 是一款基于关系型数据库 PostgreSQL 的时间序列数据插件, 根据时间序列数据的特性对存储和查询进行了改造和优化, 其聚合函数在执行过程中使用了 PostgreSQL 本身的并行计算加速手段, 插件提供的高级功能持续查询则通过生成物化视图实现了汇总查询, 能够进一步提升大规模数据的聚合查询执行效率. 所有数据库均部署在一台 CPU 为 Intel(R) Core(TM) i7-9700 CPU@3.00 GHz、内存 16 GB 的台式机上, 所有文件存储在一块 5 400 转/分钟的机械硬盘上.

表 3 实验数据集基本信息

名称	行数	列数	非空数据点数
HAR	16 603 407	6	49 500 000
中船	2 046 175	17	54 500 000
中车	416 049	4 065	1 340 500 000
Gaussian	–	–	2 000 000 000

本文主要测试了最大值 MAX 和均值 AVG 这两个聚合函数在不同实现方案、不同数据库之间的性能差异, 两者分别属于单列和多列元数据聚合查询, 并已在测试基准中广泛使用[31,32]. 实验中共涉及 6 种聚合函数的实现方案, 其中, File Header 和 SQLite3 分别表示在 Apache IoTDB 中将物理元数据存储在文件头和关系型数据库内两种不同的存储引擎实现方案; UDF 则是 IoTDB 提供的用户自定义函数执行框架, 使用流式算法对数据进行一次完整遍历计算出最大值和均值; InfluxDB 使用了 Flux 查询语言内置的最大值和均值聚合函数, 其同样使用了流式算法; TimescaleDB 则使用了插件提供的持续查询生成了汇总表物化视图, 在视图上使用并行聚合查询得到原始数据的最大值和均值.

实验所用的查询语句为仅包含时间范围筛选条件的单个聚合查询, 在固定查询数据点个数的情况下, 时间范围的起始时间在有效范围内随机选取, 以避免数据库的查询缓存机制影响实验结果. 而随机数的步长为固定批量的数据, 这与实际应用中按照日历天数选取起止范围的查询需求相符, 比如在车联网数据分析和可视化场景中, 数据看板通常以自然日或自然月为参考范围分析某辆车的行驶数据, 即时间范围总是倾向于从某天 0 时开始, 而随机选取其他时间作为起始时间的查询需求相对较少. 在 IoTDB 中查询语句的模板如下.

SELECT *count*(*vehicle*) **FROM** *root.beijing.north*401

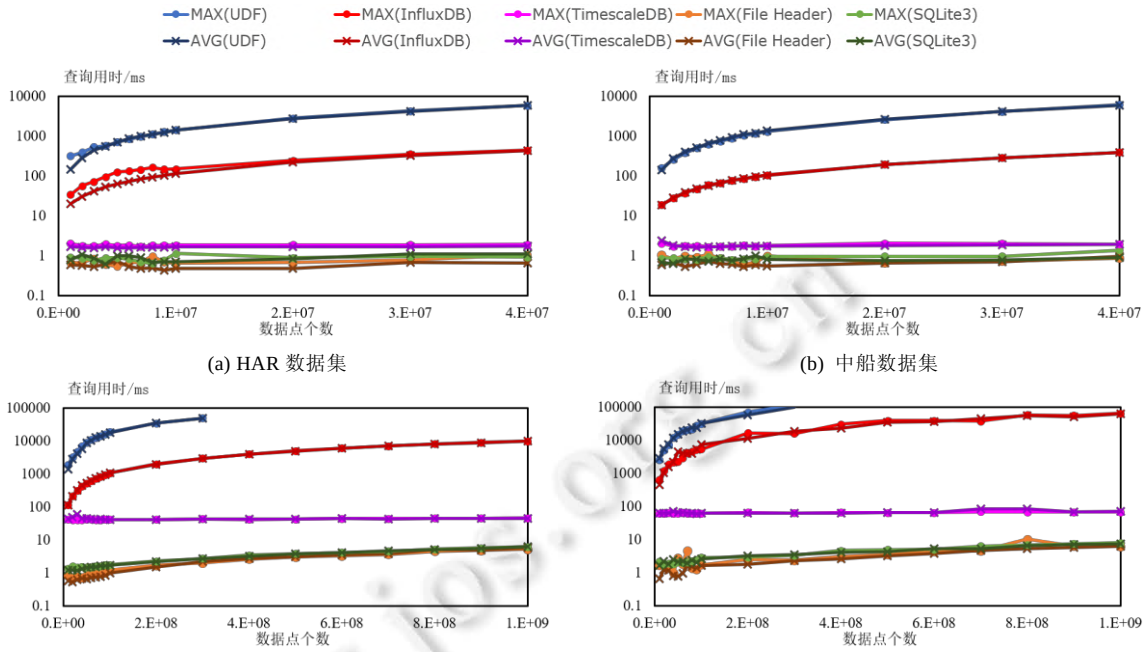
WHERE *time*≥*start_time* **AND** *time*<*end_time*+*range*;

4.2 同步计算策略

4.2.1 数据量对查询性能的影响

同步计算策略会保证在数据写入的同时完成物理元数据的计算和存储, 以在最大程度上提升聚合查询的执行效率. 因此, 本实验测试了在所有物理元数据均可用的前提下, 聚合查询性能与查询涉及的数据量之间的关系. 实验结果如图 12 所示, 结果表明, 在物理元数据全部可用的情况下, 本文提出的物理元数据管理方案的两种实现均能有效提升聚合查询的执行效率, 相比于 UDF 方法性能提升约 4 个数量级. 随着查询数据量的增长, 3 种方法的查询用时均呈现线性增长趋势, 这与理论上均为线性时间复杂度的结论相吻合. 对比另外两款时间序列数据库, InfluxDB 仅使用流式算法直接计算聚合查询结果, 因此执行效率仅高于同样采用流式算法的 UDF 方法, 两者之间约 15 倍的性能差异与数据库编程语言、文件格式等有关; TimescaleDB 在使用持续查询时执行效率约比本文所提方案低 1 个数量级, 但随着数据量的增长, 其查询用时几乎不变. 其主要原因是物化视图中元数据的数据量较小, 不足以表现出显著的性能差异.

横向对比两种不同的实现方案, 基于文件的存储方案在查询数据量小于 4E8 时平均比基于关系型数据库的存储方案性能提升约 37%; 随着查询数据量增长至 7E8 以上, 两者查询性能基本一致. 这是因为使用关系型数据库时需要额外引入连接、SQL 解析等耗时因素, 数据库内索引和连续读取的优势在数据量较小时无法弥补这一缺陷; 而数据量持续增长时, 访问大量数据文件的文件头带来的磁盘寻道和读取的压力将成为基于文件的存储方案的性能瓶颈, 关系型数据库在数据访问方面的优势使得二者的性能差异逐渐缩小至 10%以内.



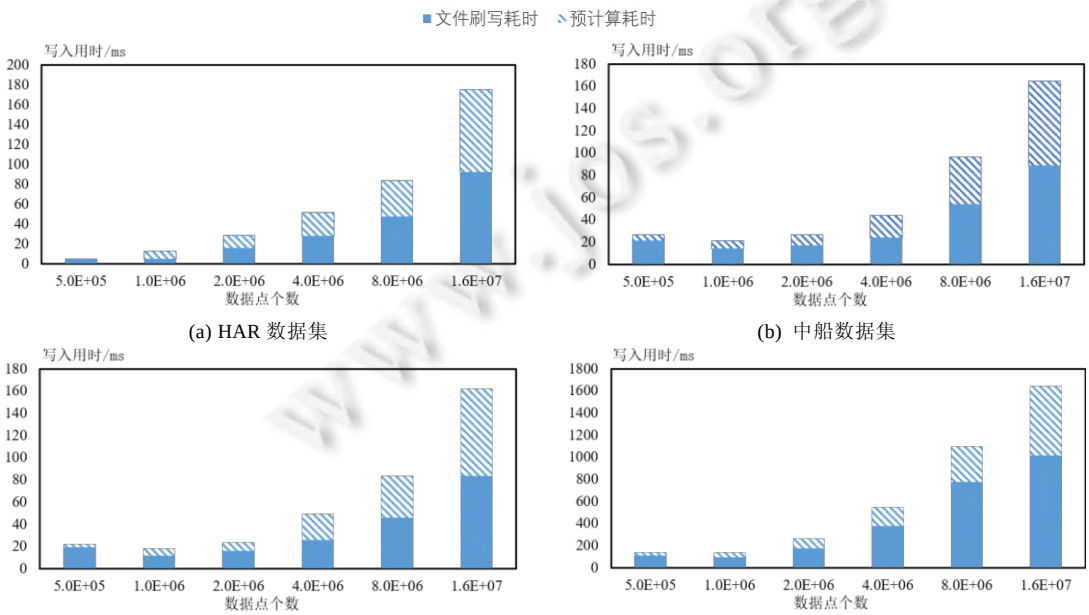
(a) HAR 数据集 (b) 中船数据集

(c) 中车数据集 (d) Gaussian 数据集

图 12 数据量对查询性能的影响

4.2.2 同步计算对数据写入性能的影响

内存数据刷写至磁盘时，元数据的同步计算在带来查询性能提升的同时，也提高了数据的磁盘写入代价. 本实验测量了基于关系型数据库的元数据存储方案在不同数据量下，元数据预计算时间与文件 I/O 读写时间在内存数据写入磁盘时的占比，结果如图 13 所示.



(a) HAR 数据集 (b) 中船数据集

(c) 中车数据集 (d) Gaussian 数据集

图 13 同步计算对数据写入性能的影响

在数据量较小时，与数据刷写至磁盘的 I/O 开销相比，元数据统计信息的计算以及 SQLite3 数据库写入开销较小，元数据的预计算在整体写入耗时中占据约 10%–20%. 数据量增大后，随着元数据计算开销和数据库写入开销的增大，预计算时间占比逐渐增加至 50%–60%. 这说明在数据写入磁盘时，利用同步计算策略更新元数据的方案带来的磁盘写入代价是不可忽视的；且在大数据量下，对写入性能的影响更为明显. 同时，也印证了利用异步计算策略和查询时计算策略更新元数据的必要性.

4.2.3 文件删除比例对查询性能的影响

当 TsFile 涉及文件有修改时，基于文件的元数据存储方案由于元数据无法及时同步，在之后的聚合查询中，需要重新读取数据文件并计算相关统计信息. 而基于关系型数据库的元数据存储方案则会利用同步计算策略对元数据进行更新，从而保持元数据与原始数据的一致性. 为了比较两种方案及 UDF 方案在原始数据有删除时的查询性能，本实验在 4 个原始数据集的基础上，分别随机挑选了一定比例的 TsFile 文件，随机删除其中 10%的时间序列数据并测试查询性能，实验结果如图 14 所示. 其中，中车数据集和 Gaussian 数据集由于数据量较大, UDF 方案查询时间过长，未参与比较.

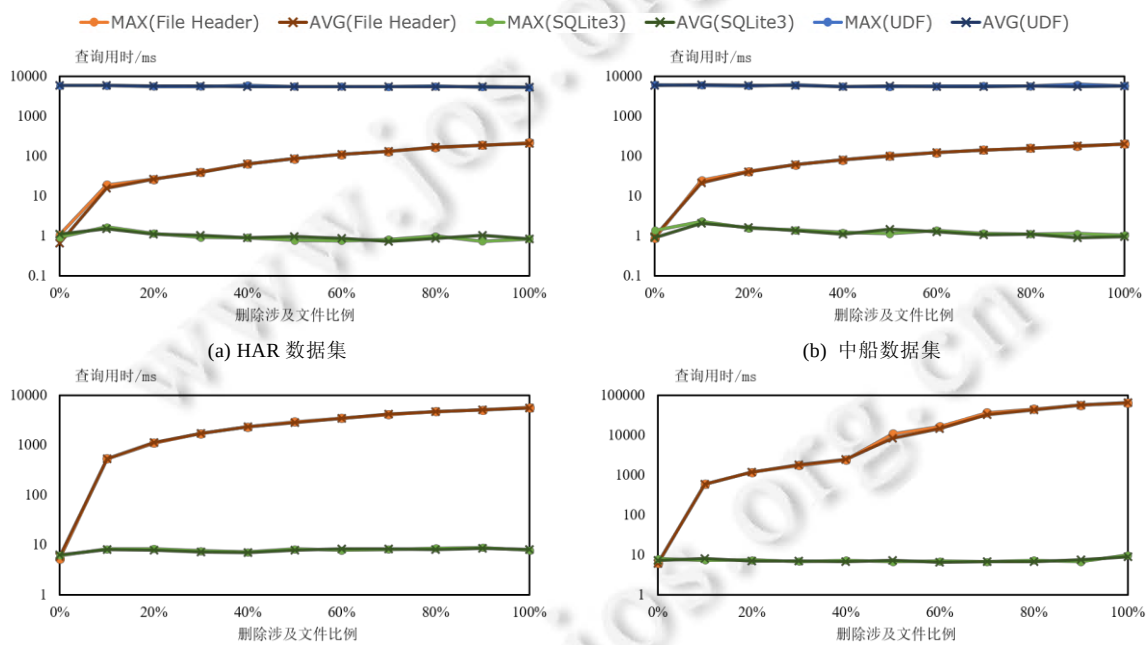


图 14 删除涉及文件比例对查询性能的影响

实验结果表明，本文实现的两种元数据管理方案在数据有删除时均能有效地提高查询效率，性能与 UDF 方案相比提高了 3–4 个数量级. 随着数据删除涉及的 TsFile 文件比例的增加，UDF 方案由于需要在查询时读取全部数据并计算统计信息，其查询用时与删除涉及文件比例无明显变化.

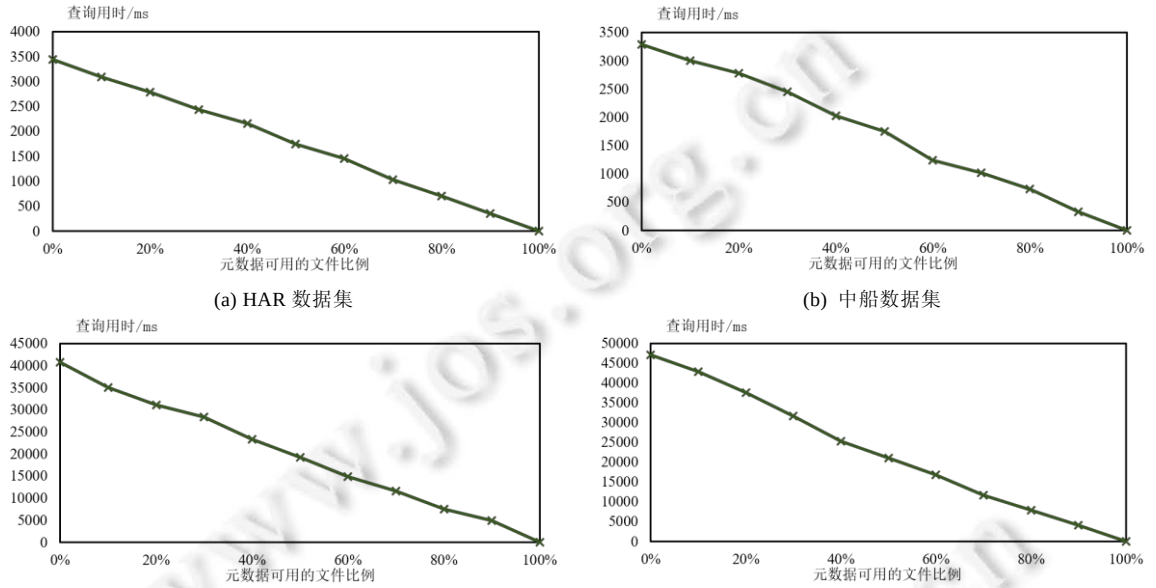
基于文件的元数据存储方案在删除涉及文件比例从 0 增长至 10%时，查询性能显著下降约 2 个数量级；随后，查询性能随删除涉及文件比例的增长呈现近似线性下降. 这是由于在数据没有删除记录时，基于文件的存储方案在进行聚合查询时可以直接读取预计算的元数据返回查询结果；而当数据有删除时，由于其元数据无法及时同步，对于有删除记录的数据文件，该方案需要重新读取原始数据进行计算，因此查询性能与没有删除记录时相比会有突变. 随后，当删除记录继续增多时，其查询用时会随删除涉及文件比例近似线性增长.

基于关系型数据库的存储方案在数据删除时会同步更新元数据，在查询过程中无须读取原始数据，因此查询用时随删除记录的增多并无明显变化. 其查询性能在数据无删除记录时与基于文件的存储方案相当，且

随着删除涉及文件比例的增大, 前者性能比后者提高了 2–4 个数量级.

4.3 异步计算策略

异步计算的优势在于, 能够在不影响数据写入和查询性能的前提下定期更新元数据. 图 15 展示了在不同数据集下, 基于关系型数据库的存储方案 AVG 聚合查询性能随可用元数据比例的变化, 横坐标代表元数据可用的 TsFile 文件占比.



(a) HAR 数据集 (b) 中船数据集

(c) 中车数据集 (d) Gaussian 数据集

图 15 异步计算比例对查询性能的影响

在可用元数据信息较少时, 大部分查询都需要读取原始数据文件, 查询用时较长, HAR 和中船数据集中的查询用时达到了 3.5 s 左右; 而在中车数据集及 Gaussian 数据集中, 由于数据量较大, 查询用时更是达到了 45 s 左右. 随着元数据可用的 TsFile 文件占比越高, 聚合查询能够利用的元数据信息越多, 因此查询效率也越高, 查询用时随可用元数据比例的增加呈现近似线性下降. 当可用元数据比例达到 100%时, 查询用时与第 4.2.1 节中同等数据量下的实验结果相近, 收敛至 10 ms 以内.

4.4 查询时计算策略

查询时计算策略能够利用查询中间结果反哺物理元数据存储, 加速后续涉及相同数据的聚合查询. 为了验证策略的有效性, 本实验在禁用了同步计算和异步计算策略的情况下, 从关系型数据库中删除所有已存储的物理元数据, 在数据集上随机生成一组各自覆盖约 1/20 数据的聚合查询, 仅使用查询时计算策略更新所涉及数据文件对应的物理元数据, 图 16 展示了依次执行 100 次查询的用时变化.

由于每次聚合查询是从整个数据集中随机选取了一段固定长度的时间范围, 涉及的数据文件是否被此前的聚合查询所覆盖也是随机的, 因此查询用时呈现出剧烈的不稳定, 但整体上仍表现出明显的下降趋势. 查询 ID 小于 40 的查询在执行时, 存储引擎中已被查询时计算策略更新的物理元数据有限, 随机选取的时间范围较难命中已有条目, 查询用时较长, 中车和 Gaussian 数据集上的用时与图 15 中相同数据量下元数据可用的文件比例为 0%的结果接近. 而随着查询的进行, 物理元数据表被逐步完善, 查询 ID 大于 80 的查询用时基本收敛在毫秒级别, 这与第 4.2.1 节中物理元数据均可用的实验结果一致. 以上实验结果验证了查询时计算策略在更新物理元数据上的有效性和对查询性能的提升效果.

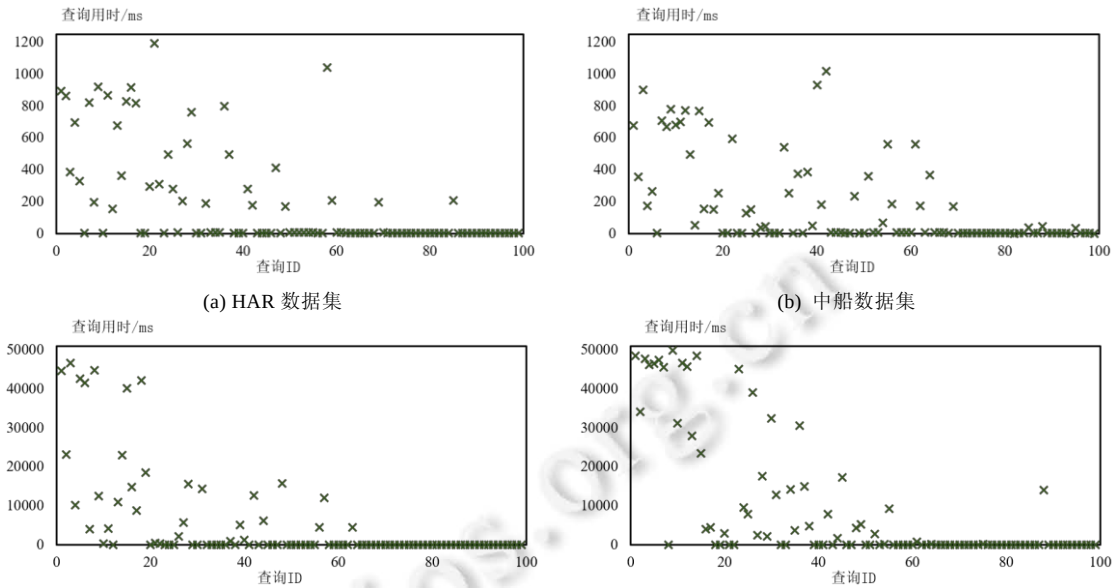


图 16 查询时计算对查询性能的影响

5 总结和展望

聚合查询在时间序列数据的分析场景中有着广泛应用，使用元数据加速聚合查询能够显著提升执行效率. 本文提出了一种面向聚合查询的 Apache IoTDB 物理元数据管理方案，针对时间序列数据采集和存储场景中写入负载高、计算资源有限以及数据在时间维度上分布不均匀、非有序等问题，本文提出了基于数据库中数据文件管理机制切分数据的物理元数据方案，并结合同步计算、异步计算和查询时计算策略，降低对数据写入性能的影响，维持元数据的最终一致性. 本文使用关系模型对物理元数据进行建模，将时间序列数据上的聚合查询重写为物理元数据表上的聚合查询语句和读取部分原始数据的子查询，多个数据集上的查询实验证明了本文所提出方案对聚合查询执行效率的提升效果和多种计算策略的有效性.

在本文工作的基础上，许多方向仍然值得进一步研究: 首先是目前同步计算策略在文件刷写阶段仍然会对写入性能产生一定的负面影响，通过流水线作业、并行计算等方法能够避免直接阻塞刷写线程，但仍然局限于单台计算机的可用资源，实际生产环境往往部署在集群中，数据存储节点和计算节点也可能不同，利用分布式架构或许能够进一步减轻数据写入阶段的压力; 另外，本文仍然只涉及最大最小值、计数、求和、平均值、方差等常见的数值统计函数，但与已有工作相比，本文所用的聚合函数模型和存储的 ER 模型具有良好的可扩展性，后续应进一步探索支持更多聚合函数乃至用户自定义聚合函数，以满足实际数据分析场景中多样的聚合查询需求.

References:

[1] Huang XD, Zheng LF, Qiu MM, *et al*. Time-series data aggregation index. Journal of Tsinghua University (Science and Technology), 2016, 56(3): 229−236, 245 (in Chinese with English abstract).

[2] Jensen SK, Pedersen TB, Thomsen C. Modelardb: Modular model-based time series management with spark and Cassandra. Proc. of the VLDB Endowment, 2018, 11(11): 1688−1701.

[3] Wang D. Research on Summary query based on database [Ph.D. Thesis]. Shenyang: Northeastern University, 2013 (in Chinese).

[4] Sheng J, Fang J, Guo XQ, Wang CD. Implementation of multidimensional aggregate query service for time series data. Journal of Chongqing University, 2020, 43(7): 121−128 (in Chinese with English abstract).

[5] Wang C, Huang X, Qiao J, *et al*. Apache IoTDB: Time-series database for internet of things. Proc. of the VLDB Endowment, 2020, 13(12): 2901−2904.

[6] Naumann F. Data profiling revisited. ACM SIGMOD Record, 2014, 42(4): 40−49.

[7] Widenius M, Axmark D, Arno K. MySQL Reference Manual: Documentation from the Source. O'Reilly Media, Inc., 2002.

[8] Momjian B. PostgreSQL: Introduction and Concepts. New York: Addison-Wesley, 2001.

[9] Mannino MV, Chu P, Sager T. Statistical profile estimation in database systems. ACM Computing Surveys (CSUR), 1988, 20(3): 191−221.

[10] Poosala V, Haas PJ, Ioannidis YE, *et al*. Improved histograms for selectivity estimation of range predicates. ACM Sigmod Record, 1996, 25(2): 294−305.

[11] Czejdo B, Taylor M, Putonti C. Model of summary tables selection for data warehouses. In: Proc. of the Int'l Conf. on Advances in Information Systems. Berlin, Heidelberg: Springer, 2000. 1−13.

[12] Zhang Q. Research on key technology of materialized view based on aggregation function [Ph.D. Thesis]. Nanjing: Nanjing University of Science and Technology, 2010 (in Chinese).

[13] Teschke M, Ulbrich A. Using materialized views to speed up data warehousing. Technical Report, IMMD 6, University of Erlangen-Nuremberg, 1997.

[14] Gutiérrez AG. Applying OLAP pre-aggregation techniques to speed up aggregate query processing in array databases [Ph.D. Thesis]. IRC-Library, Information Resource Center der Jacobs University Bremen, 2012.

[15] Gutiérrez AG, Baumann P. C omputing aggregate queries in raster image databases using pre-aggregated data. In: Proc. of the ICCSA. 2008. 447−479.

[16] Timko I, Dyreson CE, Pedersen TB. Pre-aggregation with probability distributions. In: Proc. of the 9th ACM Int'l Workshop on Data Warehousing and OLAP. 2006. 35−42.

[17] Edara P, Pasumansky M. Big metadata: When metadata is big data. Proc. of the VLDB Endowment, 2021, 14(12): 3083−3095.

[18] Babu S, Widom J. Continuous queries over data streams. ACM Sigmod Record, 2001, 30(3): 109−120.

[19] Larsen C, Sigoure B, Ligus O, *et al*. OpenTSDB: The distributed, scalable time series database. 2018. https://opentsdb.net/

[20] Hawkins B, Sabin J, Nicolaie A, *et al*. KairosDB: A fast time series database on Cassandra. 2016. https://kairosdb.github.io/

[21] Klemm S, Nordström E, Arye M, *et al*. TimescaleDB: The modern Postgres for time-series. 2020. https://www.timescale.com/

[22] Pelkonen T, Franklin S, Teller J, *et al*. Gorilla: A fast, scalable, in-memory time series database. Proc. of the VLDB Endowment, 2015, 8(12): 1816−1827.

[23] Kornacker M, Behm A, Bittorf V, *et al*. Impala: A modern, open-source SQL engine for hadoop. In: Proc. of the 7th Biennial Conf. on Innovative Data Systems Research (CIDR 2015). Asilomar, 2015. Article No.5.

[24] Choi H, Lee KY. Efficient processing of an aggregate query stream in MapReduce. KIPS Trans. on Software and Data Engineering, 2014, 3(2): 73−80.

[25] Peng D, Duan K, Xie L. Improving the performance of aggregate queries with cached tuples in MapReduce. Int'l Journal of Database Theory and Application, 2013, 6(1): 13−24.

[26] O'Neil P, Cheng E, Gawlick D, *et al*. The log-structured merge-tree (LSM-tree). Acta Informatica, 1996, 33(4): 351−385.

[27] Chen PPS. The entity-relationship model—Toward a unified view of data. ACM Trans. on Database Systems (TODS), 1976, 1(1): 9−36.

[28] Huang XD, Wang JM, Wong R, *et al*. Pisa: An index for aggregating big time series data. In: Proc. of the 25th ACM Int'l Conf. on Information and Knowledge Management. 2016. 979−988.

[29] Qiao JL, Huang XD, Wang JM, *et al*. Dual-Pisa: An index for aggregation operations on time series data. Information Systems, 2020, 87: Article No.101427.

[30] Stisen A, Blunck H, Bhattacharya S, *et al*. Smart devices are different: Assessing and mitigating mobile sensing heterogeneities for activity recognition. In: Proc. of the 13th ACM Conf. on Embedded Networked Sensor Systems. 2015. 127−140.

[31] Mostafa J, Wehbi S, Chilingaryan S, *et al*. SciTS: A benchmark for time-series database in scientific experiments and industrial Internet of things. arXiv:2204.09795, 2022.

[32] Liu R, Yuan J. Benchmarking time series databases with IoTDB-benchmark for IoT scenarios. arXiv:1901.08304, 2019.

附中文参考文献:

[1] 黄向东, 郑亮帆, 邱明明, 等. 支持时序数据聚合函数的索引. 清华大学学报: 自然科学版, 2016, 56(3): 229–236, 245.

[3] 王丹. 基于数据库的 Summary 查询研究 [博士学位论文]. 沈阳: 东北大学, 2013.

[4] 盛家, 房俊, 郭晓乾, 等. 时序数据多维聚合查询服务的实现. 重庆大学学报, 2020, 43(7): 121–128.

[12] 张茜. 基于聚合函数的物化视图关键技术的研究 [博士学位论文]. 南京: 南京理工大学, 2010.

赵东明(1998－), 男, 硕士生, 主要研究领域为时序数据库.

邱圆辉(2000－), 男, 硕士生, 主要研究领域为时序数据库.

康瑞(1998－), 男, 博士生, 主要研究领域为时序数据查询与优化.

宋韶旭(1981－), 男, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为数据库, 数据质量, 时序数据清理, 大数据集成.

黄向东(1989－), 男, 博士, 助理研究员, CCF 会员, 主要研究领域为工业数据管理, 分布式存储系统.

王建民(1968－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为数据库, 工作流, 大数据, 知识工程.