计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202330536
61(3): 614−638, 2024

时序数据库关键技术综述

刘 帅[1,2] 乔 颖[1,2] 罗雄飞[2] 赵怡婧[1,2] 王宏安[1,2]
[1](中国科学院大学计算机科学与技术学院 北京 100049)
[2](人机交互北京市重点实验室(中国科学院软件研究所) 北京 100190)
(liushuai@iscas.ac.cn)

Key Techniques of Time Series Databases: A Survey

Liu Shuai[1,2], Qiao Ying[1,2], Luo Xiongfei[2], Zhao Yijing[1,2], and Wang Hong'an[1,2]
[1](*School of Computer Science and Technology*, *University of Chinese Academy of Sciences*, *Beijing* 100049)
[2](*Beijing Key Laboratory of Human-computer Interaction* (*Institute of Software*, *Chinese Academy of Sciences*), *Beijing* 100190)

Abstract With the continuous development of the industrial Internet of things (IIoT), an increasing number of devices and sensors are being connected to networks, resulting in a massive influx of time series data. The explosive growth of time series data presents new challenges for database management systems: continuous high-throughput data ingestion, low-latency multidimensional data queries, high-performance time series indexing, and cost-effective data storage. In recent years, time series database technology has become a hot research topic in the field of databases. Some scholars have conducted in-depth research on time series database technology, while specialized time series databases have emerged for managing time series data and have been applied in various fields. These databases have become essential components in IIoT. The existing reviews of time series databases primarily focus on the comparison of functionalities and performance, as well as providing selection recommendations for specific domains. There is a lack of research on key technologies such as data persistence, querying, computation, and indexing in time series stores. Additionally, these reviews appeared earlier and lacked research on modern time series database technologies. We conduct a comprehensive investigation and research analysis of both academic research on time series data storage and industrial time series databases. We take a deep dive into four key technologies in time series databases: 1) time series index optimization techniques; 2) in-memory data organization techniques; 3) high-throughput data ingestion and low-latency data query techniques; 4) cost-effective storage techniques for massive historical data. We also analyze and summarize existing TSDB benchmarks. Finally, we present future development directions for the key technologies in time series databases.

Key words industrial Internet of things; time series data; time series database; time series data compression; time series data store

摘 要 随着工业物联网(industrial Internet of things, IIoT)的不断发展,越来越多的设备和传感器开始连接到网络中,产生了大量的时间序列数据(简称“时序数据”),时序数据爆炸式的增长给数据库管理系统带来了新的挑战:持续高吞吐量数据摄取、低延迟多维度数据查询、高性能时间序列索引以及低成本数据存储.近年来时序数据库技术已经成为一个研究热点,一些学者对时序数据库技术进行了深入的研究,同时出现了一些专门用于管理时序数据的时序数据库,并且已经被应用在多个领域,成为工业物联网中不

收稿日期:2023-06-21;**修回日期**:2023-12-01
基金项目:中国科学院战略性先导科技专项 (XDC02030300)
This work was supported by the Strategic Priority Research Program of Chinese Academy of Sciences (XDC02030300).
通信作者:乔颖 (qiaoying@iscas.ac.cn)

可缺少的关键组成. 现有的时序数据库相关综述侧重于时序数据库的功能和性能比较，以及在特定领域中对时序数据库的选择建议，缺少对时序数据库持久化存储、查询、计算和索引等关键技术的研究，同时这些综述工作出现的时间较早，缺少对现代时序数据库关键技术的研究. 对学术界时序数据存储研究和工业界时序数据库进行了全面的调查和研究，凝练了时序数据库的 4 类关键技术：1）时间序列索引优化技术；2）内存数据组织技术；3）高吞吐量数据摄取和低延迟数据查询技术；4）海量历史数据低成本存储技术. 同时分析总结了时序数据库评测基准. 最后，展望了时序数据库关键技术在未来的发展方向.

关键词 工业物联网；时序数据；时序数据库；时序数据压缩；时序数据存储

中图法分类号 TP311

近年来，随着工业物联网（industrial Internet of things, IIoT）技术的迅猛发展，工业制造开始进入数字化时代，各种设备和传感器产生的时间序列数据（简称“时序数据”）正在爆炸式地增长，成为数字经济的核心生产要素. 在工业物联网中，时序数据通常包括传感器的测量数据、设备状态记录和生产过程参数，时序数据记录了生产环境在不同时间点上的变化趋势，通过采集和分析时序数据，可以实时监控设备状态、预测故障和优化生产过程. 工业物联网中的时序数据具有产生频率高、数据规模大、数据维度复杂的特点，例如，三一重工在生产的每台重型机械上安装了多种类型的传感器以搜集各种参数，如机械的开关信号、GPS 信息、发动机参数等，用以监控重型机械的运转情况[1]，这对海量时序数据的管理带来了挑战，海量时序数据管理已经成为一个难题. 虽然一些数据库管理系统，如关系数据库管理系统（relational database management system, RDBMS）也可以用来管理时序数据，但是时序数据的工作负载完全不同于关系数据，随着时序数据规模的不断增长，这些通用的数据库管理系统的性能会急剧下降，无法处理大规模传感网络产生的时序数据. 为了存储和分析大规模的时序数据，专门的时序数据库（time series database, TSDB）被开发出来，以克服通用数据库管理系统在时序数据管理方面的局限性. 在 20 世纪 90 年代末就已经出现了时序数据库 RRDtool[2]，用于处理网络带宽、温度监测等产生的时序数据，但是直到物联网开始快速发展，越来越多的设备被接入到网络中，时序数据库才进入快速发展时期，尤其是在最近 5 年，根据 DB-Engines[3] 排名，从 2018 年开始，时序数据库的关注度开始迅速增长. 虽然已经有一些关于时序数据库的综述工作，但是这些工作的重点集中在对时序数据库的功能和性能的比较，缺少对时序数据库技术的调查和研究，且这些综述工作出现的时间较早，缺少对现代时序数据库的研究.

本文系统地对时序数据库系统现有的工作进行了调查和综述. 首先，介绍了相关的综述性工作以及背景知识，并分析了时序数据库在管理工业物联网中的海量时序数据时面临的挑战. 其次，根据存储架构的不同，将时序数据库分为 4 大类，包括：1）内存型时序数据库；2）基于关系数据库的时序数据库；3）基于 KV（key-value）存储的时序数据库；4）原生时序数据库. 并介绍了每一类的代表系统，比较了不同存储架构的数据模型和管理时序数据的优缺点. 然后，重点研究时序数据库的 4 类关键技术，包括：1）时间序列索引优化技术；2）内存数据组织技术；3）高吞吐量数据摄取和低延迟数据查询技术；4）海量历史数据低成本存储技术. 同时还介绍了时序数据库的评测基准. 最后，本文对时序数据库关键技术在未来的发展方向进行了展望，包括面向工作负载的自适应时序数据存储技术、面向新硬件优化的时序数据存储技术，以及使用云原生技术和人工智能技术的时序数据存储技术.

本文的关注点是时序数据库关键技术，包括时序数据缓存、压缩、存储、查询、计算和索引等，不涉及对时序数据库分布式技术的讨论. 我们调查了 2013—2023 年学术界和工业界的研究成果，以及一些开源时序数据库. 根据这些要求，我们建立了综述工作的纳入标准和排除标准.

纳入标准：

1）这项工作是专门针对时序数据管理的；

2）这项工作在管理时序数据上有创新性；

3）这项工作必须包含时序数据缓存或持久化存储；

4）这项工作在当前时间正在活跃地进行.

排除标准：

1）这项工作的关注度很低，或者没有创新性；

2）该数据库是基于其他数据库研发的，在管理时序数据上没有值得关注的创新点；

3)这项工作是讨论时序数据库分布式技术的;

4)这项工作相对久远,早于2013年,除非该工作在时序数据库技术中有重要的地位,否则该工作只会用在背景介绍中.

为了高效地研究时序数据库关键技术,我们只选择与研究问题最相关的论文和时序数据库进行调查综述,从具有数据库技术、存储主题的会议和期刊中查找论文,同时还在Google Scholar, ACM, DBLP直接使用关键词搜索论文.在查找时序数据库时,我们使用了DB-Engines[4]提供的服务,在时序数据库管理系统分类Top20中,找到开源且得分较高或排名在上升的时序数据库.注意,我们的选择结果截止期为2023年4月.

本文不仅调查了最近10年学术界和工业界的时序数据库关键技术的研究成果,同时研究了一些关注度高的开源时序数据库,本文的贡献有3个方面:1)对现有的时序数据库技术进行了全面的调查和综述;2)对时序数据库的4类关键技术进行了详细的研究,并介绍了时序数据库的评测基准;3)展望了时序数据库关键技术在未来的发展方向,并提出了一些预测和建议.

1 相关工作

在本节中,我们将介绍一些与时序数据库相关的综述工作,现有综述工作侧重于不同时序数据库的功能、性能比较或特定场景中对时序数据库的选择建议.

Jensen等人[5]对时序数据库管理系统进行了全面的调查,首先按照存储架构将这些系统分为3类:1)内置时序数据存储系统,如WearDrive[6];2)外置时序数据存储系统,如Gorilla[7];3)通过扩展关系数据库实现时序数据存储,如TimeTravel[8].其次,从功能、架构和性能3个方面对这些时序数据库进行了全面的介绍.然后,提出了时序数据库在管理时序数据时遇到的挑战,比如数据压缩、索引、查询.最后讨论了时序数据库技术在未来的研究方向,如基于模型的分布式近似查询处理(approximate query processing, AQP)、面向特殊查询模式的优化技术.文献[5]完成于2017年,研究所涉及的时序数据库在现在看来很多都已经过时,一些时序数据库不再继续开发或维护,缺少对2017年以后出现的时序数据库的研究,而2017年至今是时序数据库发展最快的时期.

Bader等人[9]检索了83个开源的时序数据库,重点比较了12个时序数据库,包括InfluxDB[10], OpenTSDB[11], TimescaleDB[12], Apache Cassandra[13]等,从系统架构、数据模型、数据摄取和查询性能等方面对每个时序数据库进行了全面评估,还讨论了这些开源时序数据库的局限性和未来的发展趋势.这是一篇较新的且较全面的时序数据库调查文献,但是缺少对时序数据库技术细节的研究.

Sanaboyina[14]从能源效率的角度对InfluxDB[10]和OpenTSDB[11]进行了比较,首先讨论了数据中心能源消耗的重要性和对数据库节能的需求,然后根据InfluxDB和OpenTSDB在负载和空载条件下的能耗来评估它们的性能.

还有一些针对特定领域的时序数据库调查,Fadhel等人[15]对DB-Engines[4]上排名前8的可用于存储和查询水质数据的时序数据库(包括InfluxDB[10], OpenTSDB[11], Prometheus[16]等)进行了比较分析,评估各种时序数据库纳入低成本水质监测系统的能力.Grzesik等人[17]在低功耗传感器网络边缘计算背景下对时序数据库进行比较分析,重点研究了TimescaleDB, InfluxDB, Riak TS[18]三个时序数据库以及PostgreSQL[19]和SQLite[20]两个关系型数据库的性能,并在Raspberry Pi计算机上进行了性能测试.

可以发现,以上的综述工作有2个特点:1)研究工作集中在5年前,缺少对最近5年内的时序数据库技术的研究;2)研究内容集中在时序数据库的功能和性能的比较,以及在特定领域中对时序数据库的选择建议,缺少对时序数据库存储、查询、计算、索引等技术的研究.然而,过去几年里时序数据库技术正在迅速发展,至今为止还没有一篇涵盖时序数据库关键技术的完整的综述文章.本文将对时序数据库的关键技术,包括时序数据库持久化存储、查询、数据压缩、索引与计算技术进行全面的调查和研究.注意,本文不涉及对时序数据库分布式技术的讨论.

2 背景知识

在本节中,我们首先分析时序数据的特点,然后概述时序数据库的基本知识,最后介绍在时序数据库技术中最常使用的2种数据结构.

2.1 时序数据

时序数据是一组按照时间顺序索引的数据点,这些数据通常由同一来源在一个固定的时间间隔内的连续测量组成,用于跟踪随时间而产生的变化[21].时序数据由测量指标(metric)、标签集(tag set)、测量

字段集(field set)和时间戳(*timestamp*)构成. 以气象监测数据为例, 测量指标是每个气象气球需要采集的气象(*weather*)数据, 每个气象气球有 2 个静态数据: 位置(*location*)和气球 *id*, 每分钟采集温度(*temperature*, *temp*)和湿度(*humidity*)2 个气象数据, 如表 1 所示.

Table 1 Weather Monitoring Data

表 1 气象监测数据

指标	标签集	测量字段	时间戳
weather	*location*=southeast1, *id*=1	*temp*=9.6, *humidity*=51	1 672 531 200
weather	*location*=west2, *id*=2	*temp*=7.6, *humidity*=20	1 672 531 200
weather	*location*=southeast2, *id*=1	*temp*=9.6, *humidity*=53	1 672 531 200
weather	*location*=southeast1, *id*=1	*temp*=9.6, *humidity*=52	1 672 531 260
weather	*location*=west2, *id*=2	*temp*=7.5, *humidity*=22	1 672 531 260

时序数据有 4 个特点:

1)数据是结构化的, 带有一个时间戳, 数据在时间上是有序的;

2)数据以写入为主, 极少情况下有更新和删除操作;

3)数据规模巨大, 例如, GoldWind 风力发电公司有超过 2 万个风力发电机, 每个风机上有 120~510 个传感器, 数据采集频率可达 50 Hz, 整个公司每秒产生接近 5 亿个时序数据点[22];

4)数据的查询与分析通常以时间范围为基础, 需要多个维度的过滤查询分析, 如 t_1~t_2 时间内, 位置为 southeast1、气球 *id* 为 1 的气象气球监测到的最高温度和最低温度.

大部分时序数据库使用 KV 模型的列式存储, 由时序数据的特点可知, 时序数据是带有时间戳的特殊的 KV 数据. 例如, 1 条时序数据“*weather*, *location*=southeast1, *id*=1, *temp*=9.6, *humidity*=51, *timestamp*=1672531200”可以被表示为 2 条 KV 数据:

key_1=*weather*, *location*=southeast1, *id*=1, *temp*;

$value_1$=〈9.6,1 672 531 200〉;

key_2=*weather*, *location*=southeast1, *id*=1, *humidity*;

$value_2$=〈51, 1 672 531 200〉.

这里的键(key_1, key_2)被称为序列键(series key), 用于描述采集值的静态属性, 值($value_1$, $value_2$)由采集值和时间戳组成.

2.2 时序数据库基本知识

时序数据库是用于管理时序数据的一类数据库, 至少应该具有写入(write)和查询(query)操作, 有些时序数据库也提供更新(update)和删除(delete)操作. write 将一个或者一批时序数据点写入时序数据库; query 从时序数据库中查询数据, 需要支持基于时间范围和标签组合的过滤查询; update 更新具有相同标签值和时间戳的时序数据; delete 根据时间范围、标签值和字段从数据库中删除数据. 一些时序数据库还提供了数据分析服务, 比如 InfluxDB 提供了 SUM, MIN, MAX, AVG 等聚合计算函数, 以及 HOLT_WINTERS 等时间序列预测方法[23], 用于预测未来的趋势和季节性变化, 这些特性在预测和趋势分析至关重要的金融和投资领域尤其有用. 时序数据库需要面对每秒数百万数据点的写入, 不断增加的时序数据导致存储成本快速增加, 一些时序数据库通过数据生命周期管理和数据压缩来减小存储成本.

值得注意的是, 时序数据库通常不需要支持事务, 工业物联网应用对传感数据通常没有强一致性要求, 但是在金融行业如股票交易, 需要保证每次操作的原子性和数据的强一致性, 一些时序数据库, 如 DolphinDB[24] 通过 2 阶段提交(two-phase commit, 2PC)[25-26] 和多版本并发控制(multi version concurrency control, MVCC)[27] 实现对事务的支持, 被广泛应用在量化金融交易中.

随着边缘计算的兴起, 时序数据库需要被部署在边缘设备中, 与数据中心相比, 边缘计算存在资源受限的问题, 而时序数据库通常需要消耗大量的内存、CPU 和磁盘资源. 为了适应边缘环境, 一些时序数据库提供了在边缘侧的数据同步功能, 如 InfluxDB 的边缘数据复制(edge data replication)[28], 也有一些为充分利用边缘节点的资源专门面向边缘环境研发的时序数据库, 如 EdgeDB[29].

时序数据库有多种实现方式, 不同的时序数据库采用了不同的数据结构, 针对时序数据的特点使用各种优化方法, 其最终目的是实现高吞吐量数据摄取和低延迟复杂查询. 时序数据是写密集型, 写入数据的频率远高于读取数据的频率, 多数时序数据库使用对写密集型工作负载友好的数据结构, LSM-

tree[30]是时序数据存储中最常使用的数据结构,在2.3节中我们将会详细介绍LSM-tree.在时序数据库中,索引也是重要的数据结构,它是一种根据条件获取相关信息的方法和结构,索引的作用有2个:1)对时间序列的元数据索引,实现多维度复杂查询;2)对写入磁盘文件的时序数据索引,以快速定位文件中的历史数据.通常情况下,时间序列元数据的索引会保留在内存中,对磁盘文件中数据的索引会部分或者全部保存在磁盘中.随着时间序列元数据的增加,驻留在内存中的索引大小会急剧增加,查询效率也会变慢,如何高效地管理时间序列元数据是时序数据库需要解决的一个关键问题.为实现高效的复杂查询,不仅需要对文件中的时序数据索引,同时可能需要对多个文件中的数据重新整理写入新的文件,使文件中的数据更有序,提高查询效率.数据的重复读取和写入所需的额外I/O导致读放大[31]和写放大[32],写放大会严重影响系统数据摄取吞吐量,还可能带来额外的危害,如导致SSD寿命降低[33],写放大和读放大已经成为时序数据库技术中的重要问题.

2.3 时序数据库中的数据结构

时序数据库内部使用了多种数据结构管理时间序列元数据和时序数据.不同的数据结构有不同的优缺点,没有任何一个数据结构是最优的,时序数据库使用了多种优化方法,同时作出权衡,使这些数据结构在管理时序数据时有更好的表现.在本节中,我们首先概述不同存储介质的特点,然后介绍一些时序数据库中常用到的数据结构.

通常情况下,机械硬盘(hard disk drive, HDD)[34]在时序数据存储中占据主要地位,HDD由磁头、盘片等组成,在工作时盘片会以每分钟几千转的速度高速旋转,磁头可以沿着盘片半径的方向移动,在盘片的指定位置进行数据写入和读取操作,因此大部分数据结构和算法都针对HDD的这些物理特性进行了优化,通过顺序写入和读取[30,35-36]获取更高的性能.然而,基于闪存(flash)技术的固态硬盘(solid state drive, SSD)的价格越来越便宜,数据中心已经开始大范围使用固态硬盘,SSD完全由半导体芯片构成,没有移动部件,在数据写入和读取时没有机械部件的移动,对数据访问模式不敏感,具有数据并行处理能力[37],且SSD具有有限的生命周期[38],因此,大多数面向HDD的优化方法并不适用于SSD.除了SSD,非易失性内存(non-volatile memory, NVM)[39]技术的出现对存储系统提出了新的要求,NVM在掉电后数据不会消失,具有字节可寻址(byte-addressable)特性,存储密度比DRAM更高,相比SSD,读写延迟降低到1/100,带宽提高了5~10倍[39-40],这些特性使得NVM成为替代SSD的合适选择.

存储是数据库的核心功能,数据的特点、工作负载以及存储器的固有特性共同决定了数据库在设计存储引擎时选择的数据结构.为了充分利用磁盘顺序写的特点,数据库通常采用缓存(cache)技术,在主内存(RAM)中累积更多的数据,然后批量写入磁盘.查询时使用近似成员查询过滤器加速确定某个时间序列的数据是否在磁盘文件中,减少昂贵的I/O操作.在时序数据库持久化存储中,有一种被广泛使用的数据结构:日志结构合并树(log structured merge tree, LSM-tree)[30],LSM-tree是一种针对写入密集型工作负载优化的数据结构,非常适合时序数据写入频率高、读取频率低的场景,大多数时序数据库使用了LSM-tree,并针对时序数据的特点采用了很多优化方法.在介绍LSM-tree之前我们首先介绍可用来提高查询效率的过滤器(filter).

2.3.1 近似成员查询过滤器

近似成员查询过滤器(approximate membership query filter, AMQ-Filter)用于快速估算一个元素在一个大的数据集合中是否存在,减少不必要的I/O操作[41],AMQ-Filters可以帮助减小读放大.AMQ-Filters有一个重要特点,它可能导致假阳性(false positives),但是不会导致假阴性(false negatives).假阳性是指当元素不在集合中时,查找结果为真;假阴性是指集合中存在元素时,查找结果为假.

布隆过滤器(Bloom filter)[42]是最常见的近似成员查询过滤器,由一个m位的比特数组和k个哈希函数组成,每个哈希函数将元素映射到数组的一个位置,将该位置的比特位设置为1.查询时需要计算所有k个哈希函数,如果比特数组中所有的对应位置都被设置为1,那么返回真.布隆过滤器需要在存储空间和误判率之间权衡,增加比特数组的位数和哈希函数的数量可以降低误判率,但这会占用更多的存储空间.一些时序数据库使用布隆过滤器加速查询磁盘文件是否存在指定时序的数据,减少不必要的磁盘I/O.除了布隆过滤器,还有一些常见的AMQ-Filters,如cuckoo过滤器[43]和xor过滤器[44].

2.3.2 日志结构合并树

LSM-tree最初是由O'Neil等人[30]在1996年提出,是面向写密集型工作负载设计的数据结构,其写入性能比传统的B tree/B+ tree以及其他树型结构更好,被广泛应用在数据库系统中.LSM-tree的核心思想是

将小的随机写转换成大的顺序写，充分利用磁盘顺序写的优势. 2011 年 Google 开源了基于 LSM-tree 的 KV 数据库 LevelDB[36]，推动了 LSM-tree 在数据库领域的应用，其架构如图 1 所示.

LSM-tree 由 3 部分组成：

1）预写日志（write ahead log, WAL）. 以追加的方式将数据写入日志文件中，用于系统异常重启后的数据恢复.

2）内存表（memtable）. 分为可变内存表（mutable memtable）和不可变内存表（immutable memtable），是 LSM-tree 的内存模型，可以是哈希表（Hash table），也可以使用其他缓存技术，LevelDB 使用跳表（skip list）[45] 作为 memtable 的数据结构.

3）排序字符串表（sorted string table, sstable）.KV 数据持久化存储的文件格式，每个 sstable 包含了一组键值对（key-value pair），按照 key 的字典顺序排序，允许高效地顺序访问.

我们首先按照数字顺序分析 LSM-tree 的数据传播过程：

1）写入或更新数据，每条 KV 数据以追加的方式写入 WAL，WAL 会保证数据的持久性和一致性[46]. 在数据恢复阶段，WAL 中的数据会被重新写入内存表.

2）数据写入 WAL 后，相同的数据被再次写入可变内存表中，该内存表是 KV 数据的缓存区，避免了小的随机写.

3）当可变内存表的大小达到阈值后，会变成不可变内存表，同时创建一个新的可变内存表缓存新到来的数据.

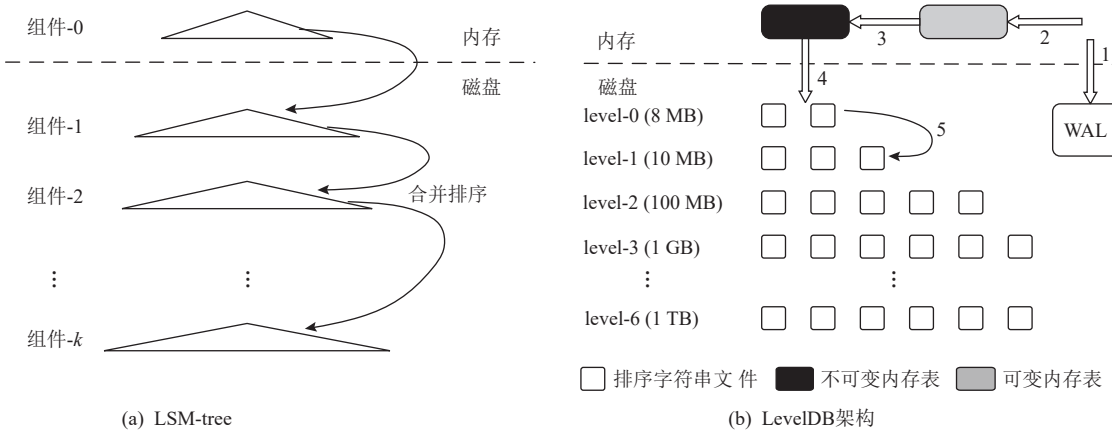
4）当不可变内存表生成后，首先对其缓存的数据按照 key 的字典顺序排序，然后将排序后的 KV 数据以数据块的形式顺序写入磁盘 sstable 文件，当文件大小超过阈值后，将关闭该文件，同时创建一个新的 sstable 文件继续写入.

5）每个层级（level）只能容纳一定大小的 sstable 文件，这个大小随着层级的增加而呈指数增加，例如，LevelDB 在默认情况下将增长因子维持在 10 倍的水平之间[36]. 由不可变内存表写入磁盘的 sstable 文件处于 level-0，由于持续地写入，level-0 中 sstable 文件数量快速增加，不同文件之间可能存在 key 范围重叠的情况，这时会触发合并（compaction）操作，将当前层级中的 1 个或者多个 sstable 文件与下一层级中存在 key 范围重叠的 sstable 文件合并写入一个新的 sstable 文件，放入下一层级. 合并结束后，删除参与合并的 sstable 文件. sstable 文件会不断地被合并到下一层级，直到最高层级.

由于合并操作的存在，KV 数据会从 level-0 不断向下一层级移动，直到到达最高层级，即相同的数据会在不同层级之间重复写入，这就产生了写放大[32]，在 LevelDB 中写放大可以达到 50 倍[47]，写放大问题会导致数据的摄取吞吐量降低.

在查询过程中，会从内存表和不同层级的 sstable 文件中查询数据，当进行大范围合并操作时，还会导致尾延迟（tail latency），尾延迟也被称为“高百分比延迟”，是相对较少但是响应时间较长的一小部分请求的延迟，是客户端很少看到的高延迟.

基于 LSM-tree 的时序数据存储，key 是由测量指标、标签组合、测量字段键（field key）构成，value 是由测量字段值（field value）和时间戳构成，与基于 LSM-tree 的 KV 存储相比，基于 LSM-tree 的时序数据存储需要分别处理测量值和时间戳. 时序数据通常具有多个标签组合，当标签集的基数增加时，基于标签组合的 key 的数量会急剧膨胀，而 key 通常是在内存中索引的，内存资源占用也会急剧增加. 一些时序



(a) LSM-tree (b) LevelDB架构

Fig. 1 LSM-tree and LevelDB architecture[31]

图 1 日志结构合并树和 LevelDB 架构[31]

数据库将 2.3.1 节提到的 AMQ-Filter 与 LSM-tree 一起使用, 以减小读放大. 没有任何一个数据结构是完美的, 有很多优化方法被应用在时序数据库中来提高性能, 大多是通过在各种性能之间权衡来实现的, 也有一些更新颖的方法被提出, 例如 Pebblesdb[32] 提出了 FLSM(fragmented log structured merge tree), 与 RocksDB[48] 相比, 写放大从 42 减小到 17, 同时写入吞吐量提高了 6.7 倍.

3 工业物联网海量时序数据管理面临的挑战

工业物联网中的时序数据具有产生频率高、数据规模大、数据维度复杂的特点, 同时与工业物联网相关的应用服务在时序数据管理上产生了新的工作负载, 其中包括 3 个方面.

1)高吞吐量数据摄取. 时序数据库需要 24 h×365 天全年稳定地处理每秒数百万时序数据点的写入请求, 由于工业环境的复杂性, 时序数据经常因为设备故障、网络拥堵等原因出现数据丢失或者无法有序到达, 对时序数据高速摄取带来了困难.

2)低延迟多维度复杂查询. 工业物联网通常需要 2 种形式的查询: 一是查询传感器的最新值实现对设备的实时监控; 二是基于时间窗口和标签值过滤的多维度复杂查询, 通过分析多个时间序列的关联信息, 对设备进行故障诊断和预测分析.

3)大规模历史数据存储. 工业物联网产生的数据量非常大, 例如 GoldWind 风力发电公司 2 万个风力发电机每秒产生接近 5 亿个时序数据点, 这些数据通常需要长期存储, 甚至永久存储, 以便对生产设备全生命周期管理以及事故追溯.

时序数据库是解决海量时序数据高效管理的有效途径, 面对工业物联网中海量的时序数据, 时序数据库会面临 3 个挑战, 如表 2 所示. 这 3 个挑战分别是: 1)需要高效地管理复杂的时间序列元数据; 2)需要应对工业物联网特殊的工作负载; 3) 需要降低海量历史数据存储的成本.

Table 2 Challenges, Key Technologies and Descriptions of Time Series Databases

表 2 时序数据库面临的挑战、关键技术和描述

挑战	关键技术类别	描述
时间序列元数据管理	时间序列索引优化技术	通过优化时间序列索引, 解决时序数据高基数问题.
特殊工作负载	内存数据组织技术	通过新颖的内存数据组织技术, 在有限的内存资源中缓存更多的时序数据, 高效地处理热数据.
	高吞吐量数据摄取技术	面向写入密集型工作负载, 优化时序数据存储方法, 减小时序数据存储的写放大问题, 提高写入吞吐量.
	低延迟数据查询技术	通过过滤器、索引等技术, 减小复杂查询中的读放大问题, 降低数据查询延迟.
海量历史数据存储	海量历史数据低成本存储技术	通过数据生命周期管理、数据压缩和分级存储技术, 减小海量历史数据的存储成本.

3.1 时间序列元数据管理

工业物联网中的每个传感器都有多个与之关联的标签, 这些标签表示传感器的静态属性, 如所属的厂区、设备、制造商等. 在数据写入和查询时, 首先需要获取数据对应的时间序列元数据, 通过对时间序列建立索引, 可以提高定位元数据的速度.

但是当时序数据集存在大量的标签集合时, 时序数据库会对每一个标签组合创建一个时间序列, 并在内存中建立索引, 随着时间的推移, 时序数据的基数越高, 索引就越大, 内存占用越大, 建立索引以及从索引中查询会占用更多的 CPU 和内存资源, 在数据摄取和查询时, 大量的资源和时间被消耗在对时间序列索引的建立和查找中.

3.2 工业物联网特殊工作负载

工业物联网中的时序数据是写入密集型, 数据的写入频率远高于读取频率, 同时具有时间局部性(temporal locality)[49], 数据冷热分明, 热数据是最近接入被频繁访问的数据, 冷数据是访问频次较低的历史数据, 热数据的价值远高于冷数据, 需要被频繁访问和分析. 时序数据库需要设计面向写入密集型工作负载的存储引擎, 同时需要使用优化方法在内存中高效地管理热数据.

3.3 海量历史数据存储

工业物联网持续产生的海量历史数据最终需要被写入磁盘持久化存储, 存储成本随着数据的不断写入而持续增加, 时序数据库需要通过各种方法减小海量历史数据存储的成本.

4 时序数据库的分类和特点

从广义上来说, 任何针对时序数据存储进行优化的数据库都应该属于时序数据库系统的范畴. 关系数据库系统也可以用来管理时序数据, 关系数据库通常基于 B+tree, 这种数据结构在处理单个时间序

列的批量数据写入时具有很高的性能，但是在同时处理数千数万个时间序列的批量数据写入请求时，数据写入最终看起来更像是随机写入，而不是仅追加写入，因此随着时序数据规模的不断增长，关系数据库性能会急剧下降，不适用于真实世界时序数据场景. 与关系数据库相比，一些 KV 存储系统更适合管理时序数据，特别是基于 LSM-tree 的 KV 数据库，如 LevelDB[36]，InfluxDB 在第 1 个版本中使用 LevelDB 作为持久化存储引擎[50]，LevelDB 中的 key 空间是有序的，在处理时序数据时，只有时间戳在 key 中，才能够快速查询指定时间范围内的数据. LevelDB 将数据分割成许多小文件，随着时序数据规模的增加，包含时间戳的 key 的数量急剧膨胀，单个进程打开的文件句柄快速增加，最终可能导致操作系统的资源被耗尽，这是基于 LSM-tree 的 KV 数据库在处理时序数据时无法解决的问题，因为 LSM-tree 在处理时序数据时没有考虑到时序数据的特点（每一个数据点都与一个时间戳关联）和特殊工作负载（基于时间范围的数据查询）.

本文的研究范围是针对时序数据存储进行优化的时序数据库，这类数据库不仅能够高效地存储大规模时序数据，而且能够满足时序数据场景下复杂的多维查询和计算要求. 本文按照存储架构对时序数据库进行了分类，每一类时序数据库的存储架构如图 2 所示. 按照存储架构，时序数据库可以分为 4 类：1）内存型时序数据库；2）基于关系数据库的时序数据库；3）基于 KV 存储的时序数据库；4）原生时序数据库. 表 3 列出了每一类的代表性系统以及数据结构，并对时序数据读写性能分级和历史数据压缩性能分级，其中每个分级通过高、中、低 3 个等级衡量.

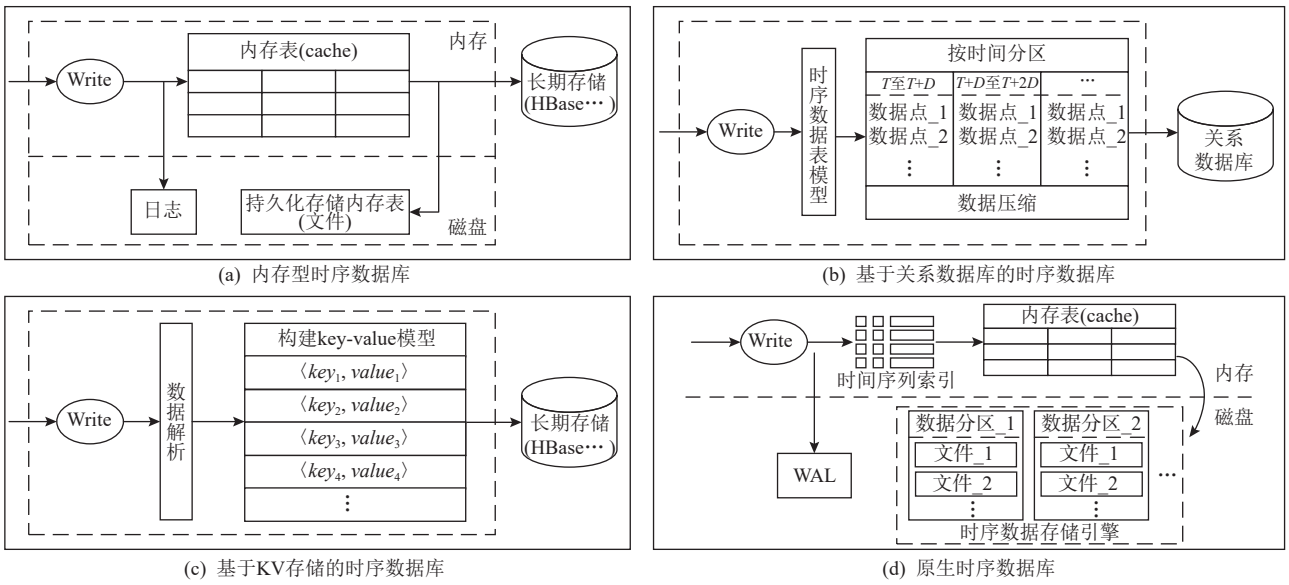


Fig. 2 Four types of storage architecture diagram of time series database

图 2 时序数据库的 4 类存储架构图

Table 3 Classification and Comparison of Storage Architectures for Time Series Databases

表 3 时序数据库的存储架构分类与比较

存储架构	代表性系统	数据结构	持久化存储	数据模型丰富度	性能	数据压缩
内存型时序数据库	Gorilla[7]	TSmap	本地简单存储	低	高	高
基于关系数据库	TimescaleDB[12]	B-tree	PostgreSQL[19]	高	高	低
基于 KV 存储	OpenTSDB[11]	LSM-tree	HBase[51]	中	中	中
	KairosDB[52]	LSM-tree	Cassandra[13]，H2[53]	中	中	中
原生时序数据库	InfluxDB[10]	LSM-tree	内部存储	低	高	高
	IoTDB[54]	LSM-tree	内部集成	低	高	高

4.1 内存型时序数据库

内存型时序数据库使用内存作为存储介质，在内存中缓存持续摄取的时序数据，与基于磁盘存储的时序数据库相比，内存型时序数据库具有更高的

写入吞吐量和更低的访问延迟. Gorilla[7]是一个内存型时序数据库, 在将监控数据写入 HBase[51]之前, 起到高速缓存的作用. Gorilla 在内存中使用 TSmap (timeseries map)数据结构缓存时序数据, 这些数据通过多种编码方法压缩以减少存储空间. 为了应对单节点系统故障, Gorilla 使用了 3 种磁盘文件: 日志文件(log file)、完整数据块文件(complete block file)和检查点文件(checkpoint file). 摄取的时序数据以追加(append-only)的方式写入日志文件, 用于在系统异常崩溃重启后恢复内存中丢失的数据, 考虑到不提供事务支持, Gorilla 没有使用预写日志文件(WAL), 而是当数据被缓存到 64 KB 后触发写日志操作, 在系统因异常崩溃时可能会丢失少量的数据, 但是与预写日志相比, 这种方式具有更高的写入吞吐量. 每 2 h, Gorilla 将缓存的时序数据拷贝到磁盘完整数据块文件, 一旦 1 个文件写入完成, Gorilla 就会触发 1 个检查点文件并删除相应的日志. 为了提高查询效率, Gorilla 使用了多种压缩算法压缩时间戳和浮点数据, 与传统的基于 HBase 的时序数据库相比, 极大地减小了查询时延, 同时提高了写入吞吐量.

内存型时序数据库通常用作时序数据高速缓存, 在内存中实现数据的写入、查询和聚合, 适用于对热数据访问频次高的场景.

4.2 基于关系数据库的时序数据库

基于关系数据库研发的时序数据库, 通过扩展关系数据库以优化时序数据存储, 继承了关系数据库的生态, 例如原生支持标准 SQL.TimescaleDB[12]通过扩展 PostgreSQL[19]实现时序数据管理, 可以在整个 PostgreSQL 实例中运行, TimescaleDB 通过在 PostgreSQL 的查询计划器、数据模型和执行引擎的深处添加钩子, 高度定制化扩展层, 基于该扩展模型, TimescaleDB 可以利用 PostgreSQL 的多个属性, 例如可靠性、安全性以及丰富的第三方工具. 基于关系数据库的时序数据库提供了全部的 SQL 功能, 同时支持事务, 适用于数据分析高度依赖 SQL 功能以及对数据一致性要求高的场景. 但是由于关系数据库使用行存储模型, 数据压缩效率与采用列存储的时序数据库相比有较大的差距, 且部署和运维复杂度更高.

4.3 基于 KV 存储的时序数据库

基于 KV 存储的时序数据库, 通过扩展 NoSQL 数据库实现时序数据存储, 将摄取的时序数据构建成 KV 模型, 底层以 KV 形式将数据持久化在分布式文件系统(distributed file system, DFS)上, 由于使用了分布式文件系统, 这类时序数据库具有很高的扩展性, 但是存在部署复杂、运维困难的问题.

4.4 原生时序数据库

原生时序数据库是面向时序数据存储全新研发的时序数据库, 该类型时序数据库不依赖第三方存储, 使用列存储模型, 提供极致的数据摄取、查询和压缩能力, 部署和运维更加简单. 但是原生时序数据库支持的数据模型有限, 通常只支持整型、布尔型、浮点型和字符串 4 种数据类型, 数据模型丰富度以及对 SQL 的支持程度不如基于关系数据库的时序数据库, 且通常不支持事务, 难以处理对数据一致性要求高的任务, 更适合工业物联网场景.

5 时序数据库关键技术

与其他类型的数据相比, 时序数据的工作负载具有一些截然不同的特性, 包括持续高吞吐量数据摄取、多维度复杂元数据管理、基于时间范围和标签值的多条件复杂查询、数据生命周期管理和历史数据管理. 从时序数据工作负载的特性中可以凝练出 4 类关键技术: 时间序列索引优化技术、内存数据组织技术、高吞吐量数据摄取和低延迟数据查询技术、海量历史数据低成本存储技术. 其中, 时间序列索引优化技术是与解决时序数据高基数问题相关的技术(见 5.1 节). 内存数据组织技术是时序数据库面向时序数据的特殊工作负载在内存中组织管理数据的技术(见 5.2 节). 高吞吐量数据摄取和低延迟数据查询技术是关于如何减小写放大和读放大的技术(见 5.3 节). 海量历史数据低成本存储技术是关于如何减小海量时序数据存储成本的技术(见 5.4 节). 表 4 列出了每一类技术的关键技术.

5.1 时间序列索引优化技术

当时序数据的基数很大时, 时序数据库写入和查询的性能会急剧降低, 因为大量的内存和 CPU 资源被用在时间序列索引的建立和查找中, 时序数据高基数已经成为时序数据库不得不面对的难题. 时间序列索引优化技术是为了解决时序数据高基数问题, 主要分为 2 类: 1)面向索引模型和架构的优化技术; 2)面向基数分析的索引优化技术.

5.1.1 高基数的定义

对于时序数据而言, 高基数问题变得更加复杂. 时间序列高基数问题[61]是指时序数据库在处理复杂的时间序列元数据时遇到的资源占用以及性能问题. 在数据库中, 基数是指存储在数据库中的唯一数据集的数量, 具体来说, 它指的是在一个表列中可能的

Table 4 Overview of Key Technologies of Time Series Databases

表 4 时序数据库关键技术总览

技术类别	关键技术	代表性工作	主要优点	主要缺点
时间序列索引优化技术	面向索引模型和架构的优化技术	InfluxDB TSI[55] TimescaleDB[12] TDengine[56] QuestDB[57] DolphinDB[24]	性能高	复杂度高
	面向基数分析的索引优化技术	VictoriaMetrics[58] TimescaleDB[12]	定位准确	需手动执行
内存数据组织技术	基于内存压缩的数据组织技术	ByteSeries[59] DolphinDB[24]	压缩比高	复杂度高，需要数据解压缩
	对齐时间序列	IoTDB[54]	复杂度低	内存占用减小有限
高吞吐量数据摄取和低延迟数据查询技术	LSM-tree：减小写放大	IoTDB 分离策略 Cassandra TWCS[60]	提高写入性能	影响查询性能
	LSM-tree：减小读放大	BloomFilter[42] 聚合计算索引	提高查询性能	影响写入性能
海量历史数据低成本存储技术	数据生命周期管理	InfluxDB[10] IoTDB[54] TDengine[56]	-	-
	数据压缩	IoTDB[54] InfluxDB[10] VictoriaMetrics[58]	-	-
	分级存储	TDengine[56]	-	-

注：“-”表示该技术没有显著的优缺点对比.

唯一值的总数. 时间序列基数表示了时间序列元数据的复杂程度，时序数据是和描述这些数据的元数据相关联的，且通常会被索引以提高多维度查询性能. 时序数据的基数是由每个单独索引列的基数的叉积来定义. 例如，一个省的街区视频监控与 1 000 个街道 ID（*street_id*=1, 2, …, 1 000）、5 家供应商 ID（*vendor_id*=1, 2, …, 5）、1 000 台设备 ID（*device_id*=1, 2, …, 1 000）共 3 个标签关联，那么总基数为 1 000×5×1 000=5 000 000，通过对每个设备索引，可以快速地找到 *street_id*=5, *vendor_id*=2, *device_id*=300 的摄像装置. 当增加了一个新的标签值时，如 *vendor_id*=6，那么基数会呈指数级别的增长变成 1 000×6×1 000=6 000 000. 因此，时序数据的高基数由 2 个原因导致[62]：1）1 个时间序列有多个索引列；2）每个索引列包含多个唯一值.

高基数会带来下面 2 个问题：1）高基数增加了定位唯一值的时间，导致写入吞吐量降低，查询性能降低；2）高基数同时导致在内存中索引的唯一列的数量呈指数增长，内存占用急剧增加. 解决时序数据的高基数问题，需要解决 2 个具体的问题：1）在内存中索引时间序列，提高写入和查询性能；2）当增加标签时，避免数据集基数呈指数级增长. 多种方法被用来解决时序数据高基数问题，包括时序数据模型优化、索引结构优化和元数据存储架构优化等.

5.1.2 面向索引模型和架构的优化技术

InfluxDB[10] 设计了专属的时间序列索引（time series index, TSI）[55] 管理时间序列元数据，TSI 使用了基于 LSM-tree 的 key-value 模型，key 由用于定位唯一时间序列的测量（measurement）指标、标签组合和测量字段构成. TSI 对 key 进行哈希运算，将 key 映射成无符号 64 b 整数，将冗长的字节数组转换成数值索引在内存中，大大减小了时间序列索引的内存占用，TSI 在高基数情况下的效果更好. 通过 Bitmap 进行基于标签值的多条件过滤查询，并使用了 LRU Cache[63] 缓存活跃的时间序列 ID，将不活跃的时间序列索引存储在磁盘文件，并动态更新 LRU Cache. TSI 可以支持千万级别的时间序列，官方给出基数上限为 3 000 万. 但是当持续增加标签或标签值时，由测量指标、标签集和测量字段唯一组合构造的 key 的数量呈指数级增加，TSI 的写入和查询性能开始明显降低. 为了解决时序数据高基数问题，InfluxDB 重新设计了新的存储引擎 IOx[64]，与 TSI 将每个时间序列作为一列不同，IOx 引擎将每个标签（tag）和字段（field）存储为一个列，这样大大减少了列的总数，基数降低，从而提高了性能.

TimescaleDB[12] 是基于 PostgreSQL[19] 的时序数据库，首先按照时间范围对数据进行分区（chunk），然后在每个数据分区上默认使用 B-tree[65] 数据结构对时

序数据标签建立索引. 索引是在分区级别上, 因此索引的大小与该分区的时间范围内时序数据的基数相同. TimescaleDB 可以根据工作负载手动添加和删除索引, 在离散或者连续字段上创建索引, 同时支持对多个列创建联合索引. 与 InfluxDB 中基于 LSM-tree 的 TSI 索引不同, TimescaleDB 采用了 B-tree 数据结构, 因此在改变索引时不需要重写所有的历史数据, 且基于 B-tree 的索引对关系运算(如<, <=, =, >=, >)更友好.

QuestDB[57] 使用 HashMap 索引唯一列, 为了解决高基数问题, QuestDB 对索引列上的 HashMap 操作进行了大规模并行化处理, 同时在 SQL 引擎中使用 SIMD[66] 处理繁重的工作, 这样可以尽可能并行地执行与索引和 HashMap 查找相关的工作.

TDengine 从数据模型、元数据存储架构 2 个方面来应对时序数据高基数问题[61]. 首先, TDengine 为每个数据源创建一个表, 通过一致性哈希将表分布在多个虚拟节点中, 当表的数量增加时, 可以通过创建更多虚拟节点水平扩展, 将存储和查询元数据的压力分摊到多个虚拟节点中, 减少定位表的延迟, 保证标签过滤操作具有较小的延迟. 其次, TDengine 引入了超级表的概念, 超级表将一组标签集与对应的每个表关联, 同时将超级表数据与时序数据分开存储, 使用 B-tree[65] 来索引标签存储元数据, 在通过标签值过滤查询时, 从元数据存储中搜索满足过滤条件的表, 然后从时序数据存储中查询关联的数据并完成聚合.

5.1.3 面向基数分析的索引优化技术

VictoriaMetrics[58] 使用 mergeset 数据结构存储时间序列元数据, mergeset 是一种以 LSM-tree 为基础的数据结构, 为数据写入、查询做了特殊的优化, 同时支持按字符串前缀快速进行范围扫描. 为了解决时序数据高基数问题, VictoriaMetrics 提供了基数分析(cardinality explorer)[67] 功能, 可以识别并移除高基数的来源.

DolphinDB[24] 发现在真实世界中存在很多不活跃的时序序列, 这些被索引的不活跃的时间序列是导致高基数的主要原因, 与 VictoriaMetrics 对高基数问题的处理方法不同, DolphinDB 允许自定义标签值组合在索引中的映射关系, 可以将多个不活跃的时间线序里的标签值组合映射到同一个索引列中, 大大降低了被索引列的基数. 查询时, 将这些被映射到同一个索引列的数据全部加载到内存中过滤. 根据 DolphinDB 的实验, 通过将多个不活跃时间序列映射到一个索引列, 可以很好地解决时序数据高基数问题, 写入性能大幅度提升, 而点查询性能仅仅慢了 2 ms 左右.

理论上时序数据的基数可能是无限的, 例如, 当标签中引入了时间戳, 随着时间的推移, 这个数据集的最大基数是无限大的. 现有的优化方法都是在某些方面作出权衡, 如资源占用与查询效率之间的权衡, 将能管理的时序数据基数不断提高, 但是仍然无法解决基数无限增长的问题, 高基数仍然是时序数据库技术中需要研究的重要问题.

5.2 内存数据组织技术

时序数据具有时间局部性(temporal locality)[49], 数据冷热分明, 热数据是最近接入被频繁访问的数据, 冷数据是访问频次较低的历史数据. 在时序数据场景中, 热数据的价值远高于冷数据, 需要被频繁访问和分析. 在内存中缓存最近接入的数据是处理热数据的重要方法, 但是随着缓存的数据量增加, 内存资源占用会持续增加, 同时时序数据高基数导致需要更多内存资源索引时间序列, 因此需要采取适应性的内存数据组织技术. 该类技术使用较少的内存资源高效地处理高热度的时序数据以应对倾斜的工作负载, 以及减少高基数时间序列索引的内存资源占用. 本节将介绍 2 类内存数据组织技术, 包括: 1)基于内存压缩的数据组织技术; 2)对齐时间序列.

5.2.1 基于内存压缩的数据组织技术

1)压缩索引. ByteSeries[59] 提出了一种压缩倒排索引(compressed inverted index), 如图 3 所示, 该技术可以有效地压缩元数据, 元数据的内存占用减少了 60% 且保持了多维查询的高效性.

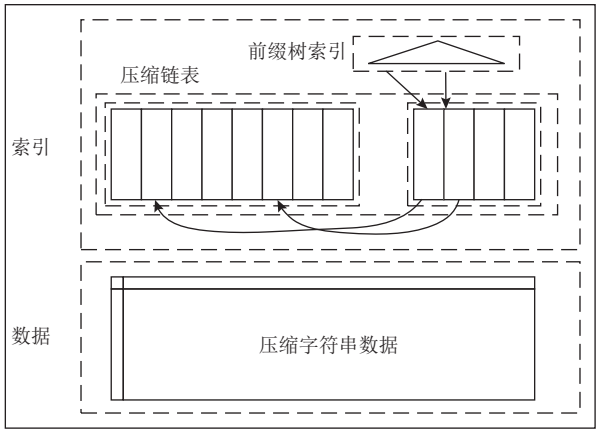


Fig. 3 Compressed inverted index[59]

图 3 压缩倒排索引[59]

压缩倒排索引分别对索引中的键和值进行压缩. 首先根据标签组合构建倒排索引, 倒排索引由键和

值2部分数据组成，将所有的值数组串联成一个数组，然后使用另一个数组记录每个倒排索引数组的值在该串联数组中的起始偏移量，同时将倒排索引中每个键对应的值更改为对应的偏移量. 最后，用前缀树[68]压缩倒排索引的键，并将串联数组和记录偏移量的数组压缩成一个数组(*compressed_list*). 在标签过滤查询时，首先从前缀树中找到每个标签键对应的偏移量；然后将 *compressed_list* 解压缩为倒排索引数组和偏移量数组，从偏移量数组获得倒排索引的位置；最后从倒排索引数组中读取倒排索引，并对查询的结果进行聚合，完成查询.

ByteSeries 在内存中对每个时间序列的数据点进行了压缩，同时为了避免数据压缩的开销影响数据摄取吞吐量，使用分段内存方法(segmented memory approach)来平滑这种影响，分段内存结构如图4所示.

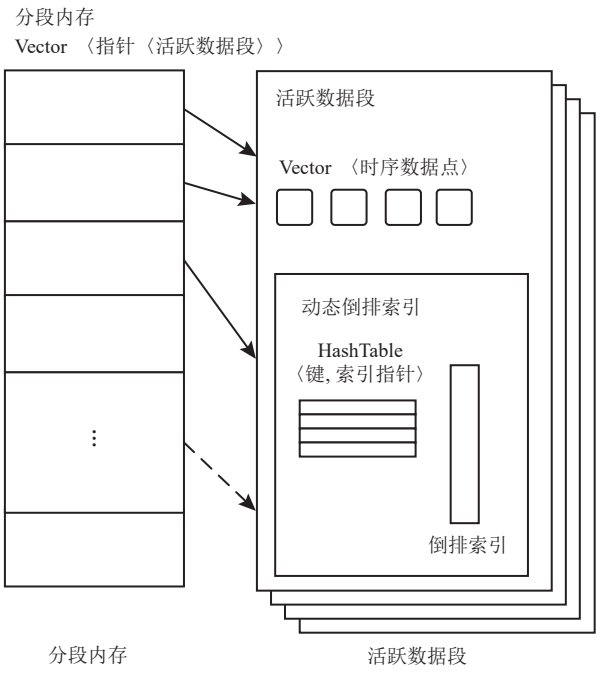


Fig. 4 Segment memory and active segment[59]

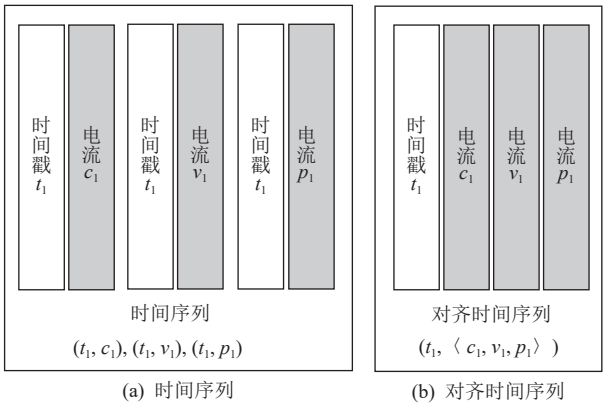
图4 分段内存和活跃数据段[59]

内存中有2种数据结构：活跃缓冲区(active buffer)和静态缓冲区(static buffer)，活跃缓冲区被划分为多个数据段(segment)，数据段使用了简单的数据结构来实现高速摄取目标，不执行数据压缩. 数据首先写入活跃缓冲区，通过计算序列键的哈希值确定写入的数据段. 当数据段大小达到阈值时，使用 Gorilla[7] 算法对时序数据压缩，并转换成静态缓冲区的格式，最后对静态缓冲区的数据进行合并存储. 通过对静态缓冲区分片(shard)，数据压缩和转换操作可以在不同数据段上并行执行，在数据段级别上进行数据压缩和转换，内存开销与参与压缩和转换的数据段的大小相同，小于整个活跃缓冲区的大小，同时避免了数据压缩和转换操作导致的系统读写操作短暂暂停，在保持高压缩率的同时减小对数据摄取吞吐量的影响.

2)压缩内存表. DolphinDB[24] 发现在 HDD 中 4 KB 随机读的时延为 5~15 ms，SSD 的 4 KB 随机读时延为 100~200 μs，而在内存中 4 KB 的 Snappy[69] 算法解压缩只需要 1~2 μs，远小于 HDD 和 SSD 的 4 KB 随机读时延. Dolphin 提出了压缩内存表，对缓冲区内的每列时序数据使用 Snappy 算法压缩，将更多的时序数据缓存在内存中，在相同大小的内存中可以多缓存5倍的数据.

5.2.2 对齐时间序列

在真实场景中，1个实体的多个测量值通常被同时采样，这些测量值会具有相同的时间列，IoTDB 提出了对齐时间序列模型[70]，并将这样一组时间序列称为对齐时间序列，如图5所示. 通过创建对齐时间序列，时间戳列在内存中只需要被存储1次，减少了内存占用.



(a) 时间序列 (b) 对齐时间序列

Fig. 5 Time series and aligned time series

图5 时间序列和对齐时间序列

5.3 高吞吐量数据摄取和低延迟数据查询技术

物联网场景下的工作负载通常是写密集型，时序数据的写入频率远高于读取频率，因此多数时序数据库使用对写密集型工作负载友好的 LSM-tree 数据结构. 但是 LSM-tree 天生具有写放大和读放大的问题，写放大[32]是指存储系统执行的总写入 I/O 相对于用户总写入数量增加的倍数，写放大导致器件频繁磨损，对硬件的寿命产生危害[71]，同时也会降低写吞吐量[72]，增加系统的读写时延，例如，在 RocksDB 中，写放大会导致写吞吐量降低到读吞吐量的 10%[73]. 读放大[31]是指在查询时，实际读取的数据大小大于

用户实际获得的数据大小, 与写放大一样, 读放大也会增加系统延迟, 但是对于写密集型工作负载, 写放大的危害更大. 减小写放大和读放大是提高时序数据库写入吞吐量以及减小查询延迟的关键, 在写密集型工作负载中, 减小写放大比减小读放大更重要. 本节首先分析写放大和读放大产生的原因以及二者的权衡关系, 然后介绍减小写放大和读放大的技术.

5.3.1 写放大和读放大的权衡

持续摄取的时序数据最终会被存储在磁盘文件中, 同一个时间序列的数据可能分布在多个文件中, 为了提高数据的压缩比和查询效率, 存储引擎会对多个文件进行合并, 尽可能将同一个时间序列的数据存储在同一个文件中, 使文件中的数据更有序. 文件合并时数据的重复写入所需的额外 I/O 导致了写放大, 而从多个文件中查询一个时间序列的数据时产生的额外 I/O 导致了读放大.

基于 LSM-tree 的时序数据存储, 将离散的随机 I/O 转换为批量的顺序 I/O, 充分利用磁盘顺序写的特点, 显著提高了写性能, 但是同时带来了写放大和读放大. 3 个原因导致了写放大: 1)时序数据首先写入 WAL, 相同的数据会被再次写入缓存, 并最终写入磁盘文件, WAL 导致数据重复存储; 2)LSM-tree 会在相邻层级之间合并文件, 相同的数据可能会从 level-0 到最高层级重复写入, 这是产生写放大的主要原因; 3)1 个实体可能在同一时间点产生多个采集值, 如 1 个智能电表 1 次需要采集电流、电压 2 个值, 以〈电流, 时间戳〉,〈电压, 时间戳〉的形式存储, 每次会重复存储时间戳. 而读放大的原因有 2 个: 1)1 个时间序列的数据可能分布在多个文件中, 查询数据时, 需要从 level-0 层级中的文件开始查找, 直到最高层级, 有些文件并不包含目标数据, 这是产生读放大的主要原因, 虽然可以通过一些技术方案, 如布隆过滤器或文件级别的索引降低从无关文件中读取数据的概率, 但是无法消除这些不必要的读取操作; 2)从磁盘文件中读取数据时, 通常是以数据块 (block) 为读取单位, 如一个数据块的大小为 4 KB, 可能会将不需要的数据加载到内存, 这也会导致读放大.

在以 LSM-tree 为基础的时序数据存储中, 写放大和读放大是一种典型的权衡. 为了提高读性能, LSM-tree 通过合并操作, 将 level-i 层级与 level-i+1 层级中存在时间序列范围重叠的多个文件合并成有序的文件并放入 level-i+1 层级, 数据的重复写入导致了写放大; 为了提高写性能, LSM-tree 通过缓存数据, 将随机 I/O 转换成顺序 I/O, 使各层级中的文件存在时间序列范围重叠, 查询时需要从多个层级的文件中读取数据, 导致读放大.

5.3.2 减小写放大

在 LSM-tree 中, 每层会存在至少 1 组按照 key 有序排列的文件, 不同文件之间没有 key 范围重叠, 这一组文件被称为一个 sorted run. 按照每层 sorted run 的数量, LSM-tree 的合并策略分为 Leveled 和 Tiered 两种, 如图 6 所示. 对于 Leveled 合并策略, 每一层级只能有一个 sorted run, 当触发合并操作时, 会将当前层级的文件与下一层级存在键范围重叠的文件合并为有序文件, 放到下一层级, 合并操作涉及 2 个层级的文件, Leveled 合并策略对顺序写有很好的优势, 因为顺序写保证了不会有 key 范围重叠的数据, 那么每个 sorted run 中的文件也不会存在 key 范围重叠的问题, 但是当多个文件存在 key 范围重叠时, 合并操作可能会涉及到当前层级与下一层级的所有文件, 导致写放大问题严重影响写性能. 而 Tiered 合并策略只会在当前层级内将存在 key 范围重叠的文件合并为一个新的有序文件, 并放入下一层级, 合并操作的范围不会跨层级, 因此写放大问题小于 Leveled 合并策略, 写性能更高. 多数时序数据库, 如 InfluxDB, IoTDB 采用 Tiered 合并策略, 减小写放大问题, 同时提高写性能.

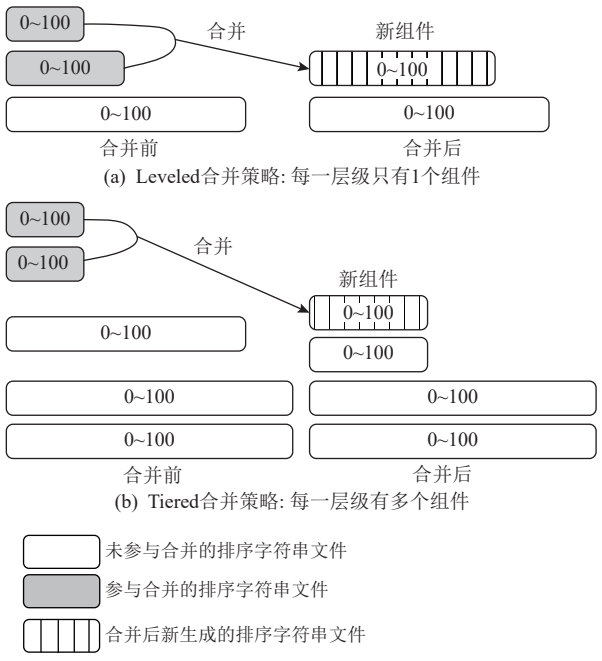


Fig. 6 LSM-tree merge policies[74]

图 6 日志结构合并树合并策略[74]

时序数据库通常采用 WAL 的方法, 在数据库因异常重启后恢复缓存中丢失的数据, 当数据写入

WAL 后，操作系统并不会直接将数据写入持久性存储设备中，而是写入 RAM 的页面缓存（page cache）中，除非使用 fsync[75] 系统调用明确告诉操作系统将最近写入的数据从页面缓存刷新到持久性存储设备. 但是 fsync 系统调用是一个非常慢的过程，频繁地触发 fsync 系统调用会严重影响系统性能. WAL 默认是单线程的，且可能会在每次写入数据时触发小数据块的落盘操作，这使 WAL 成为影响系统写入性能的主要原因之一. WAL 不仅降低了系统写入吞吐量，而且带来了写放大问题. VictoriaMetrics 放弃使用 WAL，直接在 RAM 中缓冲数据，定期（硬编码 1 s）将数据原子刷新到磁盘文件中持久化存储[76]，但这样会存在一个问题，当数据库异常关闭时会丢失最后 1 s 没有存储的数据，VictoriaMetrics 通过放弃严格的数据安全性来提高写入性能并减小写放大问题.

Cassandra[13] 是基于 LSM-tree 的 NoSQL 数据库系统，提出了一种专门面向时序数据存储的时间窗口压缩策略（time window compaction strategy, TWCS）[60]，通过一系列的时间窗口将 SSTables 文件分组，通过 SSTable 的最大时间戳决定文件所属的时间窗口，在触发合并操作时，TWCS 会在最新的时间窗口内使用基于大小的分层合并策略（sized tiered compaction strategy, STCS）[60] 合并该时间窗口内的 SSTables，在该时间窗口结束时，TWCS 将该时间窗口内所有的 SSTables 合并成一个 SSTable，一旦合并操作完成，这部分数据就不会被再次压缩，一个时间窗口内的 SSTables 不会与其他时间窗口内的 SSTables 一起压缩合并，从而减小写放大. 然而，TWCS 不适用于乱序数据的写入，因为一旦一个时间窗口内的 SSTables 合并完成，就不会再次被压缩合并.C*DynaConf[77] 提出了一种基于 Cassandra TWCS 的自动调优数据压缩合并机制，动态地调整压缩合并参数，最大化吞吐量并最小化响应时间.

TDengine[56] 也采用了 LSM-tree 架构，使用了 WAL、skip list 缓存等技术，但是去掉了 LSM-tree 的层级结构，将数据按照时间段分区，然后按表分块写入文件. 在 TDengine 的时序数据文件中，1 个数据块只能存储 1 个表的数据，每个块的块级索引（block range index, BRIN）信息在相关联的索引文件中以表为分组，按照时间顺序递增，形成索引块. 查询时将数据文件相关联的索引文件加载到内存中，直接找到数据文件中的时序数据. 通过将标签数据和时序数据分离存储，减少了标签数据的存储空间，同时去掉了 LSM-tree 的层级结构，不需要对文件进行频繁地合并，减小了写放大.

在真实场景中，时序数据并不总是有序的，它们会因为网络延迟[78]、时钟偏差[79] 等各种原因产生不同的延迟[80]. 如果新到达的时序数据的时间戳比磁盘中所有时序数据的的时间戳都要晚，那么它是有序的数据点，否则是无序的数据点. 在 LSM-tree 模型中，当内存表大小达到阈值，会将其缓存的数据写入磁盘 SSTables 文件，当存在无序时序数据时，会导致文件合并时某一时间段的数据被多次重写，增加写放大. 张凌哲等人[81] 提出了一种在保持面向大块数据的高效查询的基础上实现对最新写入的时序数据的低延迟查询的 2 阶段 LSM 合并框架，将文件的合并过程分为少量乱序文件快速合并与大量小文件合并这 2 个阶段，在每个阶段内提供多种文件合并策略，并在 Apache IoTDB 上分别实现传统的 LSM 合并策略以及 2 阶段 LSM 合并框架和测试，实验结果表明，与传统的 LSM 相比，2 阶段的文件合并模块在提升策略灵活性的情况下使即席查询读盘次数大大降低，并且使历史数据分析查询性能提升了约 20%. Kang 等人[82] 研究了 Apache IoTDB 中的分离策略（separation policy）对写放大的影响，建立鲁棒模型评估不同工作负载以及乱序内存表和有序内存表的容量对写放大的影响，在 Apache IoTDB 中实现了一个延迟分析器，以选择更好的策略来减小写放大，最后在一个真实的工业案例中使用该模型评估分离策略对减小写放大的有效性.

IoTDB 通过创建对齐时间序列[70]，实现一个实体的多个测量值的时间戳列在磁盘上只存储 1 次，减少了磁盘空间的占用，同时在文件合并时减小写放大.

5.3.3 减小读放大

时序数据最终会被持久化到磁盘文件，1 个时间序列的数据可能分布在多个文件中，查询引擎在处理 1 个查询请求时，需要从多个文件中查询，有些文件并不包含目标数据，这是导致读放大的主要原因. 时序数据库使用了多种优化方法，如过滤器、更高级别的索引来减小读放大.

InfluxDB 等使用了文件级别的布隆过滤器，提高查询命中概率，但是从文件中读取布隆过滤器以及布隆过滤器导致的假阳性，还是会带来无效的磁盘 I/O. IoTDB 等提供了数据块级别的预聚合信息，在将时序数据写入磁盘文件之前，预先计算数据块中时序数据的开始时间、结束时间以及最小值、最大值等聚合计算信息，然后将该预先计算的信息一同写入文件，在处理聚合计算请求时，通过对数据块聚合

信息过滤查询,直接得到数据块的聚合计算结果,无需将数据块全部加载到内存中解压缩和计算,通过预聚合计算的方式减小读放大.

黄向东等人[1,83]提出了一种用于分段聚合的高效索引:分段聚合的持久化索引(persistent index for segmented aggregation, PISA),该索引结合了概要表和线段树,通过计算直接定位出待查询时间序列的聚合计算信息,减少了大量磁盘 I/O,在提高聚合计算性能的同时减小了读放大.Qiao 等人[84]在文献 [1,83] 基础上提出了 Dual-PISA,可以容忍一定程度的乱序时序数据聚合. 赵东明等人[85]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,针对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 3 种物理元数据的策略,能够在乱序数据摄取场景中维持元数据的最终一致性,减小聚合查询时磁盘 I/O 次数.

5.4 海量历史数据低成本存储技术

有 3 种技术可用于降低海量历史数据的存储成本,包括:1)数据生命周期管理;2)数据压缩;3)数据分级存储. 下面我们将详细介绍这 3 种技术.

5.4.1 数据生命周期管理

不是所有的时序数据都需要被永久存储,不同的场景对数据生命周期的要求不同,当数据的生命周期结束时需要及时地删除过期数据. InfluxDB[10]基于保留策略[86]创建多个数据分片组,每个分片组只存储 1 个固定时间范围内的数据,摄取的时序数据会根据时间戳被写入相应的分片组,当一个分片组的生命周期结束后该分片组会被删除,释放硬件空间.图 7 展示了一个具有 3 天生命周期和分片组持续时间为 1 天的保留策略.

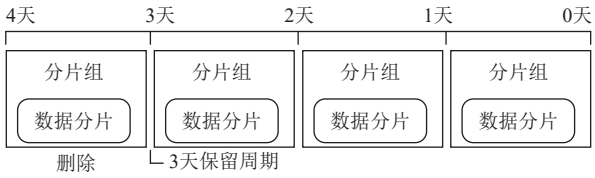


Fig. 7 Data shard group and lifecycle management[86]

图 7 数据分片组与生命周期管理[86]

5.4.2 数据压缩

除了数据生命周期管理,数据压缩也是显著减小存储成本的方法. 几乎所有的时序数据库都采用列式存储,将同一类型的数据作为一列存储在文件中,时序数据的特性决定了每一列数据可以使用不同的压缩算法,包括无损编码(lossless encoding)、有损编码(lossy encoding)以及通用压缩,近年来学术界将机器学习算法用在时序数据压缩中,取得了可观的成果.

5.4.2.1 无损编码

无损编码是一种不损失数据完整性和准确性的数据压缩方法,可以保证原始数据在解压缩后能完整地还原,并且保持原有的质量,无损编码被广泛应用在时序数据压缩中.

Gorilla[7]发现绝大多数时序数据点以固定时间间隔到达,少数数据点时间戳可能会提前或延迟 1s,即大多数相邻数据点时间戳之差为 0,少数相邻数据点时间戳之差不为 0,基于这些特点,Gorilla 提出了用于时间戳压缩的 2 阶差分 delta-of-delta 算法,大约 96% 的时间戳可以被压缩到 1 b;Gorilla 还提出了“XOR”编码算法对双精度浮点类型数据进行压缩,大约 51% 的值可以被压缩到 1 b. 这 2 种编码算法适用于波动较小的数据,当时间序列发生剧烈变化时这些方法可能会失败.

Xiao 等人[87]分析了可能影响编码性能的时序数据特征,包括规模(scale)、增量(delta)、重复(repeat)和增长(increase),并详细介绍了 Apache IoTDB 中使用的时序数据编码方式:1)用于数值型数据的差分编码(differential encoding),如 TS_2DIFF[88];游程编码(run-length encoding, RLE)[89],如带有位压缩(bit-packing)的游程编码和 PAKE 编码[90];差分和游程相结合的编码算法,如 RLBE[91], SPRINTZ[92].2)用于文本压缩的字典编码(DICTIONARY)[93]、霍夫曼编码(HUFFMAN)[94],并使用生成数据和真实的工业数据集对各种编码算法进行基准测试,通过压缩比、插入时间、查询时间 3 个指标评估各种编码算法的性能.

InfluxDB[10]使用了一种自适应编码方式进行时间戳编码,基于被编码的时间戳的结构,使用不同的编码方式. 在对时间戳编码时,首先使差分编码(delta encoding)将值转换为更小的整数,然后自适应地选择游程编码、simple8b[95]编码或不压缩. 对于整数类型的数据,首先使用 zig-zag 编码,将有符号整数转换成无符号整数,然后根据编码后值的范围使用 simple8b 编码或不压缩,这种方式对于频繁保持不变的值非常有效. 对于布尔类型的数据,InfluxDB 使用位压缩编码,字符串数据使用 Snappy[69]压缩.

5.4.2.2 有损编码

有损编码通过丢弃一些不重要的信息实现数据压缩,压缩后的数据不能被重新恢复为原始数据,只能重建为原始数据的近似值. 与无损编码相比,有损

编码可以得到更高的压缩比，但会损失部分精度.

PI Server[96] 是 OSISoft 公司（后被 AVEVA 公司收购）研发的实时数据库系统，用于管理工业中产生的时序数据，PI Server 使用了旋转门压缩（swinging door trending, SDT）[97] 算法，SDT 是一种有损压缩算法，使用线性拟合的方法压缩数据，计算复杂度较低，误差可控. Apache IoTDB 也支持 SDT 算法，在数据写入磁盘文件之前使用 SDT 算法对缓存的数据进行压缩，将压缩后的数据刷新到磁盘. SDT 算法在一定条件下可以用来跟踪数据的变化趋势，但是当数据存在噪声时，SDT 会对过程趋势作出错误的判断，压缩比降低，且 SDT 无法判断和处理异常数据. Feng 等人[98] 对 SDT 算法进行了改进，通过使用自适应记录限制和异常值检测规则处理异常值和适应实际数据的波动，实现了更高的压缩比.

5.4.2.3　其他压缩算法

时序数据被编码之后，还可以使用通用压缩算法进行 2 次压缩，例如 GZIP[99], LZ4[100], Snappy[69]. TDengine 支持 2 阶段压缩[101]，在对时序数据编码后，再次使用通用压缩算法进行压缩，获得更高的压缩效率.

除了工业界时序数据库中广泛使用的成熟的时序数据压缩算法，学术界也提出了一些针对时序数据的压缩算法. ModelarDB[102] 提出了一种在线自适应多模型压缩算法，可以动态地选择最佳模型. Eichinger 等人[103] 提出了一种基于分段回归计算的时序数据有损压缩算法，通过自定义最大偏差控制压缩精度和压缩比. Chiarot 等人[104] 对时序数据压缩算法进行了全面的调查研究，通过分类的方法分析时序数据压缩的整体方法和特征，并分析了这些算法的性能.

最近几年，随着人工智能的发展，机器学习算法也被应用到数据压缩中，Yu 等人[105] 提出了一种 2 级压缩模型，为每个时间序列选择合适的压缩算法，通过基于强化学习的方法来自动学习压缩参数，有效处理多样化的数据模式，该算法根据数据点周围局部上下文的情况动态调整压缩参数，可以有效地压缩真实世界中模式多变的时序数据. 实验评估表明，这种基于强化学习的压缩算法比 Snappy, Gorilla, MO[106] 具有更高的压缩比，将压缩比提高了 50% 以上，且压缩效率更高，适合并行计算.

5.4.3　分级存储

正如 5.2 节所述，时序数据具有时间局部性[49]，热数据被访问的频次远高于冷数据. 热数据的价值更高，需要被快速地访问和计算，而冷数据具有规模大、访问频次低的特点，随着时间的推移，热数据的访问频次会逐渐降低，呈现出由热到冷的变化趋势，不同时间范围内时序数据访问特性的不一致性对底层存储的性能要求也不同，一些时序数据库提出了分层存储架构. TDengine 通过冷热数据自动分级存储[107]，将不同热度的数据存储在不同的地方，例如，最新的数据存储在 SSD 上，1 周以上的数据存储在 HDD 上，4 周以上的数据存储在价格更低的网络设备上，从而降低了存储[108] 成本，同时保证了数据的高效访问.

我们以 InfluxDB 和 TimescaleDB 基准测试的部分关键指标的对比为例，直观展示这 4 类关键技术对时序数据库性能的影响. 图 8 是 TimescaleDB 官方发布的 TimescaleDB 与 InfluxDB 在写入性能、查询性能以及磁盘占用 3 个指标的基准测试对比[109].

测试结果表明，在低基数的工作负载中，InfluxDB 的插入性能比 TimescaleDB 表现更好，但是随着数据基数的增加，InfluxDB 的性能会急剧降低，高基数情况下，TimescaleDB 的插入性能是 InfluxDB 的 3.5 倍，这是因为 InfluxDB 使用了基于 LSM-tree 的时间序列索引，无法高效地管理复杂的时间序列元数据，而 TimescaleDB 使用 B-tree 数据结构对时序数据标签建立索引，同时允许在数据集上创建多个索引，这有利于处理高基数数据集. 对于查询而言，TimescaleDB 的查询性能总体优于 InfluxDB，在复杂查询中，TimescaleDB 的性能远远超过 InfluxDB，InfluxDB 的查询性能同样受限于基于 LSM-tree 的时间序列索引和时序数据存储，这种限制在时序数据高基数情况下更为明显. 在数据压缩存储测试中，InfluxDB 通过列式存储实现了更高的压缩比，这是 TimescaleDB 使用 B-tree 数据结构的行存储无法比拟的.

本节系统地研究了时序数据库的 4 类关键技术，并详细分析了每一类技术的代表性工作和优缺点. 这 4 类关键技术不是独立地存在，而是互相作用，共同影响时序数据存储的性能. 时间序列索引直接影响内存数据的组织方法、数据摄取和查询的性能，是时序数据库最核心的关键技术. 内存数据的组织方式依赖于时间序列索引，索引性能直接决定了在内存中定位唯一时间序列的效率，以及数据在内存中的模型. 基于分片的数据生命周期管理可以减小历史数据的存储成本，但是分片的时间跨度会影响数据摄取、查询的性能以及存储引擎的资源占用，每一个分片都包含一个独立的时序数据存储引擎，较小的分片时间跨度能使大批量的数据并行写入到多个

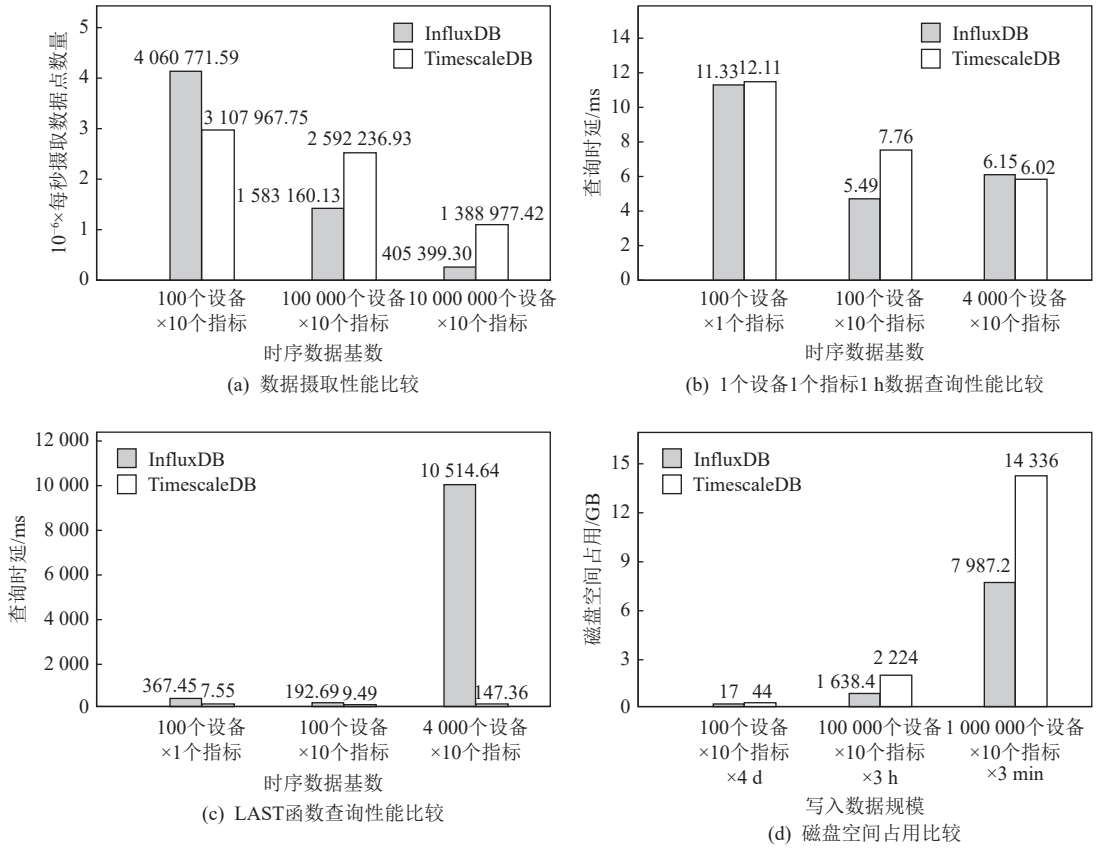


Fig. 8 Performance comparison of InfluxDB and TimescaleDB

图 8 InfluxDB 与 TimescaleDB 的性能比较

分片中，有助于提高写入吞吐量，同时跨多个分片的历史数据查询可以并行执行，提高查询效率，但是随着分片的数量增加，时序数据库对 CPU、磁盘 I/O 等资源的消耗成比例地增加，较高的资源占用又会对数据写入吞吐量和查询效率产生负面影响. 通常情况下，数据压缩有助于提高读写性能，同时减小写放大和读放大，但是数据压缩的效果与数据的波动情况密切相关，当数据波动较大时，数据压缩比较低，压缩和解压缩会占用大量的 CPU 资源，影响时序数据库的读写性能. 因此，在设计时序数据存储时，需要充分考虑到这 4 类关键技术的互相作用关系，并作出权衡.

6 时序数据库评测基准

数据库基准测试工具是用来测试与衡量数据库系统性能的重要工具，对推动数据库技术的不断发展具有重要意义，同时有助于研究人员或开发人员找到最适合工作负载的数据库系统. 评测基准对时序数据库技术的发展也至关重要，近年来，随着时序数据库技术的快速发展，出现了一些用于时序数据库基准测试的工具，但是目前可用的标准和工具还存在一些问题，无法系统地测试时序数据库的性能，主要原因是物联网以及工业物联网的场景复杂多变，不仅数据规模庞大，而且经常因为设备故障、网络拥堵等原因导致时序数据异常、乱序到达以及工作负载发生变化，现有的基准测试工具在模拟真实世界的物联网和工业物联网场景时存在不足. 本节将具体介绍 3 个接受度较高的时序数据库评测基准.

6.1 TSBS

TSBS(time series benchmark suite)[110] 是当前认可度最高、使用范围最广泛的时序数据库评测基准，从 2018 年开始，已有多个开源时序数据库厂商将 TSBS 作为基准测试平台[111-113]. 该评测基准具有可扩展性，可对多种时序数据库的读写性能进行基准测试，支持多种类别的典型查询，并能自动汇总测试结果. 当前 TSBS 支持 2 种场景：DevOps 和 IoT.

在 DevOps 场景下，有 2 种类型的测试用例：1)完整形式的测试用例用来生成、插入和测量来自 9 个系统的数据，这些系统可以在真实的开发环境中被监控(如 CPU、内存、磁盘等)；2)另一种测试用例只关注 CPU 指标，cpu-only 数据模型更简单，每个设备

具有 10 个测量值. 除了监测指标数据, 数据集同时包含标签数据, 每一个标签组合用于标识数据集中的 1 台主机, 不同主机的数量规模可以通过参数设置.

在 IoT 场景下, TSBS 模拟生成卡车公司的一组卡车的数据流, 测试用例模拟了每辆卡车的诊断数据和指标. 与 DevOps 场景下的测试用例不同, IoT 测试用例引入了真实世界中存在的影响因素, 如乱序数据、缺失数据或空数据的摄取, 数据集同时包含了卡车的元数据.

使用 TSBS 进行基准测试包括 3 个过程:

1)数据与查询生成. 使用 TSBS 进行基准测试, 需要预先生成测试数据和查询用例, 以避免即时生成数据和查询用例对测试结果产生影响.IoT 数据集会模拟生成乱序数据、缺失数据或空数据, 查询用例支持多种聚合计算, 如 MAX, AVG, LAST, LOW, HIGH, GROUP BY.

2)数据加载与写入. 该过程将加载过程 1)中预先生成的模拟数据, 写入待测试数据库, 同时也支持即时模拟和加载数据.

3)查询执行. 该过程用于评估待测数据库的查询性能, 需要先加载过程 1)中预先生成的模拟数据和查询用例, 可以通过参数设置并行查询线程数量. 最后将查询用例的执行结果输出.

6.2 iot-benchmark

iot-benchmark(原名 IoTDB-Benchmark)[108, 114] 是用来对时序数据库进行基准测试的工具, 该工具主要关注工作负载场景更加复杂的工业场景. Liu 等人[114] 详细介绍了 IoTDB-Benchmark 的架构以及数据模拟、测试工作负载、系统资源监控等过程, 并使用该工具对 4 个流行的时序数据库 InfluxDB[10], KairosDB[52], OpenTSDB[11], TimescaleDB[12] 进行基准测试, 演示了这些时序数据库在不同工作负载下的性能比较.

iot-benchmark 具有 5 个特点: 1)可模拟真实场景中的不同应用的数据分布, 包括但不限于具有可控高斯噪声的方波、正弦波和锯齿波, 通过多种数据分布测试时序数据库存储引擎中使用的多种数据编码方法; 2)可模拟有序数据和乱序数据; 3)可进行混合操作以模拟复杂的现实工作负载; 4)可记录系统资源消耗情况, 如 CPU 使用情况和内存使用情况; 5)该工具提供了一种管理基准测试结果数据的通用方法, 包括持久化测试配置和测试数据, 以及分析测试结果.

6.3 TPCx-IoT

TPCx-IoT[115] 是一个专门设计用于测量物联网网关系统性能的物联网基准, 可以直接比较不同的软件和硬件系统. TPCx-IoT 使用发电站中 200 种不同类型传感器的数据, 每条数据包含系统 ID、传感器名、采集值和时间戳, 并将数据填充到 1KB 大小, TCPx-IoT 使用 3 个指标作为评测基准:

1)性能指标 $IoT_{ps}=SF/T$. 该指标表示被测系统的有效吞吐能力, 其中 SF(scale factor)是摄取数据的数量, T 是摄取经过的时间(单位为 s).

2)性价比指标 $\$/IoT_{ps}=P/IoT_{ps}$. 该指标表示被测系统每单位性能($IoT_{ps}$)的总成本, 其中 P 是被测系统的总成本.

3)可用性指标. 该指标反映了价格配置的所有项通常可用的日期, 即对任何客户可用的日期, 保证了基准测试系统是可以购买的生产系统, 而不是仅为运行基准测试而实现的实验系统.

Poess 等人[116] 详细介绍了 TPCx-IoT, 以及该基准评测关键要素背后的设计决策, 并通过实验分析 TPCx-IoT 如何衡量物联网网关系统的性能. TPCx-IoT 自称是第一个专门面向时间序列场景的评测基准, 但是提供的测试场景不适合物联网中的实际应用, 例如, 在真实世界中, 数据并不总是有序到达, 存在乱序数据摄取的情况.

还有一些其他评测基准被用于时序数据库性能测试, influxdb-comparisons[117] 是使用 Go 语言开发的基准测试工具, 用于比较 InfluxDB 与其他时序数据库的性能, 支持 Cassandra, OpenTSDB, TimescaleDB 等数据库. Hao 等人[118] 提出并开发了 TS-Benchmark, 可以模拟真实世界中的时序数据库工作负载, 即风力发电厂中的数据管理和运行分析, 将工作负载分为 2 类: 1)与数据库性能无关的上层应用工作负载; 2)需要由数据库完成的工作负载. TS-Benchmark 更关注时序数据库本身的数据摄入和数据读取性能. 表 5 列出了上述时序数据库评测基准的对比.

时序数据库在不同的数据集上可能会表现出较大的性能差异, Shah 等人[120] 使用 3 种真实数据集(包括物联网数据集、金融数据集和分析型数据集)以及 1 种合成数据集, 对 InfluxDB[10], TimescaleDB[12], Druid[121], Cassandra[13] 进行了基准测试, 实验表明时序数据库的写入性能和查询性能在不同类型的真实数据集以及合成数据集上会有很大的差异. 时序数据库评测基准都在尽可能地模拟真实场景中的数据集和工作负载, 以获得时序数据库在真实世界中的表现, 然而当前已存在的基准评测无法覆盖不同类型的时序数据场景, 且不能较好地模拟真实场景中复

Table 5 Comparison of Benchmark for Time Series Databases

表 5 时序数据库评测基准比较

评测基准	模拟场景/数据	评价指标	主要优点	主要缺点
TSBS[110]	DevOps(CPU、内存等监控数据) IoT(一组卡车的数据流)	写入性能 查询性能	模拟数据考虑了真实世界存在的影响因素，如乱序数据、缺失数据和空数据.	需要手动进行数据正确性验证.
iot-benchmark[108]	工业 (模拟多种数据分布，包括具有可控的高斯噪声)	写入性能 查询性能 系统资源占用	可以模拟具有噪声的数据，以及乱序数据，支持数据正确性验证，支持生成测试报告及可视化.	模拟的数据不能保证真实世界的数据分布.
TPCx-IoT[115]	工业 (发电站传感器数据)	摄取吞吐量 性价比 可用性	考虑了基准评测的现实意义，即经济性和可用性.	缺少乱序数据场景.
TS-Benchmark[118]	工业 (风力涡轮机监控数据)	写入性能 查询性能	基于 DCGAN[119] 模型，通过对真实种子数据训练，生成高质量的测试数据.	缺少乱序数据场景.
influxdb-comparisons[117]	DevOps(CPU、网络等监控数据) IoT(天气、智慧家居等数据)	写入性能 查询性能	模拟的数据类别多.	缺少乱序数据场景.

杂多变的数据和工作负载，时序数据库评测基准应该提供不同场景的数据集，同时重点关注因各种故障、异常导致的时序数据缺失、时序数据质量变化、乱序数据以及工作负载突变.

7 时序数据库关键技术的未来发展方向与预测

时序数据库伴随着物联网的快速发展而进入新的发展阶段，尤其是最近 5 年出现了大量新的时序数据库，多种解决方法被提出以应对管理工业物联网海量时序数据面临的技术挑战，从这些方法中我们可以对时序数据库关键技术的发展方向做一些预测.

7.1 面向工作负载的自适应时序数据存储

不断提高性能是时序数据库研发人员在未来需要考虑的首要工作，但是没有任何一个存储方法可以同时实现最优写、最优读，每个存储方法的表现受到实际工作负载的影响，当工作负载变化时，时序数据库可能会出现很大的性能波动.

在大多数情况下，时序数据都是写密集型，因此大部分时序数据库使用对写密集型工作负载友好的基于 LSM-tree[30] 的数据结构. 然而，工作负载可能会根据实际场景的变化而变化，从写密集型变成读密集型，这时面向写密集型设计的时序数据库会出现性能波动.LSM-tree 的可调节性是很值得研究的，可以实现给定工作负载的最佳权衡，根据不同的工作负载，LSM-tree 可以进一步优化，如 Pebblesdb[32] 提出了 FLSM，通过读放大来换取更低的写放大，以适应写密集型工作负载，Mei 等人[122] 提出的 LSM-Forest 也是通过读放大换取更低的写放大. Luo 等人[74] 详细研究了基于 LSM-tree 的存储技术，包括写放大优化、合并策略优化、2 级索引、面向硬件和工作负载的优化. 在设计基于 LSM-tree 的时序数据存储时可以从这里得到启发. 在 LSM-tree 和 LSM-Forest[32,122] 之间并没有不可逾越的鸿沟，二者是可以根据参数互相转换的，时序数据存储可以设计成根据工作负载情况自适应地选择 LSM-tree 或者 LSM-Forest，以应对不同的工作负载. 如果存在特定的访问模式，如数据倾斜，那么需要研究与这些访问模式相关的优化技术，以应对特殊的工作负载.

7.2 面向新硬件的时序数据存储

面向新硬件优化的时序数据存储正在成为一个新的趋势. 基于 Flash 的 SSD 已经在消费级产品中取代了 HDD，SSD 几乎占领了个人电脑市场，而且随着 SSD 价格的不断下降，SSD 在大型数据中心所占的比重正在不断上升. SSD 的硬件特性与 HDD 有非常大的差异，SSD 内部具有大的并行数据处理能力，且对数据访问模式不敏感[37]，Chen 等人[123-124] 系统地研究了基于闪存的固态硬盘的内部并行性，揭示了在高速数据处理中，充分利用固态硬盘的内部并行性的重要作用. 然而，目前时序数据存储的大多数技术方案都是针对 HDD 硬件优化的，这些优化方法对 SSD 无效，甚至会对 SSD 造成损害[123]，因此面向 SSD 硬件优化的时序数据存储是一个很重要的研究方向，时序数据存储充分利用 SSD 的内部并行性，可以显著提高时序数据库的读写性能. 在其他类型的数据存储中，面向 SSD 的优化已经成为一个研究热点，已经有一些面向 SSD 优化的 KV 存储技术被提出，Lu 等人[31] 提出了 WiscKey，这是一个基于 LSM-tree 的持久性 KV 存储，具有面向性能的数据布局，它将键和值分开，以尽量减少写放大，并在设计时充分考虑了面向 SSD 的优化，利用了 SSD 设备的顺序和随机

I/O性能特性. Doekemeijer等人[33]详细地调查了面向SSD块接口(block interface)优化的KV持久化存储技术,这些技术对设计时序数据存储很有启发性,因为采用列式存储模型的时序数据存储可以被看作是一种特殊的KV存储. 除了面向SSD数据块级别接口的KV存储研究,还有一些面向Open channel SSD和非易失性内存(non-volatile memory, NVM)优化的KV存储研究, Wang等人[125]提出了一种基于Open Channel SSD的LSM-tree, 通过利用SSD内部多个通道(channel)以及优化I/O请求并发调度策略,提高数据并行处理,将系统的吞吐量提高4倍以上. Kannan等人[126]提出了可充分利用非易失性内存特性基于LSM-tree的NoveLSM, 通过利用NVM的字节可寻址性(byte-addressability)来减少读写延迟,从而实现更高的吞吐量. 游理通等人[127]提出了一种基于日志结构的非易失性内存KV存储系统TinyKV, 该系统通过减少对NVM的写入与缓存刷新指令,降低写入延迟. 文献[31,33,37,123−127]研究对设计面向新硬件优化的时序数据存储有很大的参考价值.

7.3 Cloud+AI

工业物联网的快速发展离不开云计算技术的发展,越来越多的公司开始将数据中心迁移到云平台,包括公有云和私有云. 云基础设施的虚拟化、弹性分配、高可用等特点,为时序数据库提供了按需分配、高可用、低成本存储等优势,一些基于云原生技术的时序数据库快速发展起来,采用计算和存储分离的架构,实现计算资源和存储资源的独立分配和扩展以充分利用云基础设施弹性伸缩的特点. 然而,云原生技术对时序数据库提出了新的要求,时序数据库的数据组织、存储管理、查询优化、故障恢复等都需要为适应云环境而重新设计和优化. 云原生技术已经在其他类别的数据库中发展壮大,例如在OLTP和OLAP数据库,董昊文等人[128]深入分析了云原生数据库系统的架构和技术,探讨了云原生OLTP和云原生OLAP数据库的架构设计和关键技术. 云原生技术也适合时序数据场景,充分利用云原生技术的时序数据库可以实现更高的写入吞吐量以及更低的存储成本.

云计算提供的强大算力同时带动了人工智能(artificial intelligence, AI)的快速发展,与云原生技术一样,人工智能技术与时序数据库技术的结合可能带来新的发展方向. 例如, Yu等人[105]使用强化学习技术压缩时序数据,不仅可以应对真实世界中模式多变的时序数据,同时可以实现更高的压缩比. 除了时序数据压缩,人工智能技术可以在2个方面给时序数据库赋能: 1)AI使时序数据库更加智能,通过深度强化学习实现存储引擎自适应调优,建立AI驱动的索引、故障检测与恢复机制,使时序数据库能够应对更加复杂的工作负载,实现更高效的时序数据存储和更快的故障恢复. 孟小峰等人[129]对机器学习化数据库系统进行了调查和综述,包括存储管理、查询优化的机器学习化研究以及自动化的数据库管理系统,并指出了机器学习化数据库系统的未来研究方向及可能面临的问题与挑战. 2)人工智能为时序数据库提供了更先进的数据分析能力,例如,基于海量的历史数据,使用时间序列预测算法来预测一个数据集在未来可能的价值,或通过时间序列异常检测发现系统中可能存在的异常,对保障系统安全、设备平稳运行具有重要的意义. 总之,人工智能技术不仅可以使时序数据库更加智能,还可以充分挖掘海量实时、历史时序数据的价值,提供更先进的数据分析能力以满足越来越复杂的分析场景.

8 总　　结

本综述对最近10年时序数据库关键技术进行了完整的调查、总结、分类和研究. 本文分析了时序数据库在管理工业物联网海量时序数据时面临的挑战,包括: 1)如何高效地管理复杂的时间序列元数据; 2)如何应对工业物联网特殊的工作负载; 3)如何降低海量时序数据存储成本. 围绕这3个挑战,本文首先对时序数据库进行了分类和比较,然后重点研究了时序数据库的4类关键技术: 1)时间序列索引优化技术; 2)内存数据组织技术; 3)高吞吐量数据摄取和低延迟数据查询技术; 4)海量历史数据低成本存储技术. 本文总结了每一类技术下具体的关键技术,并对每一个关键技术进行了详细的研究. 这些关键技术决定了时序数据库在管理工业物联网大规模复杂时序数据时的写入吞吐量、查询时延、对关键数据计算响应的实时性以及存储成本,对提高工业物联网的实时性、安全性和优化制造流程具有重要的实际价值,是数据库设计人员需要着重考虑的关键技术.

本文还概述了时序数据库的主流评测基准,包括TSBS, iot-benchmark, TPCx-IoT,详细介绍了这些评测基准的数据生成特点、执行过程和评价指标. 最后,本文展望了时序数据库关键技术在未来的发展方向,并提出了一些预测和建议.

通过调查和研究可知,工业物联网产生的时序

数据正在不断地增加,时序数据库面临的挑战也在不断地增加,随着时序数据规模的膨胀,现有的解决方案可能会力不从心,迫切需要面向新的场景、新的环境和新的硬件研究时序数据库关键技术.

作者贡献声明:刘帅负责调研并完成论文撰写;乔颖负责论文审阅,并给出详细的修改指导意见;罗雄飞、王宏安参与了论文审阅,并提出指导意见;赵怡婧参与了方案讨论、技术支持等工作.

参 考 文 献

[1] Huang Xiangdong, Zheng Liangfan, Qiu Mingming, et al. Time-series data aggregation index[J]. Journal of Tsinghua University: Science and Technology, 2016, 56(3): 229−236, 245 (in Chinese)
(黄向东, 郑亮帆, 邱明明, 等. 支持时序数据聚合函数的索引[J]. 清华大学学报: 自然科学版, 2016, 56(3): 229−236, 245)

[2] Oetiker T. RRD tool: Round-robin database tool [EB/OL]. [2023-04-10]. http://oss.oetiker.ch/rrdtool/

[3] Gelbmann M. Time series DBMS are the database category with the fastest increase in popularity[EB/OL]. [2023-04-10].https://db-engines.com/de/blog_post/62

[4] solid IT. DB-Engines ranking of time series DBMS[EB/OL]. [2023-04-10].https://db-engines.com/en/ranking/time+series+dbms

[5] Jensen S K, Pedersen T B, Thomsen C. Time series management systems: A survey[J]. IEEE Transactions on Knowledge and Data Engineering, 2017, 29(11): 2581−2600

[6] Huang Jian, Badam A, Chandra R, et al. WearDrive: Fast and energy-efficient storage for wearables[C] //Proc of the 2015 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2015: 613-625

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

[8] Khalefa M E, Fischer U, Pedersen T B, et al. Model-based integration of past & future in TimeTravel[J]. Proceedings of the VLDB Endowment, 2012, 5(12): 1974−1977

[9] Bader A, Kopp O, Falkenthal M. Survey and comparison of open source time series databases[C] //Proc of the 17th Conf on Database Systems for Business, Technology, and Web. Bonn, Germany: German Informatics Society, 2017: 249−268

[10] InfluxData. InfluxDB time series database[EB/OL]. [2023-04-10].https://www.influxdata.com

[11] The OpenTSDB Team. OpenTSDB-A distributed, scalable monitoring system [EB/OL]. [2023-04-10]. http://opentsdb.net/

[12] Timescale. TimescaleDB time series database[EB/OL]. [2023-04-10].https://www.timescale.com/

[13] The Apache Software Foundation (ASF). Apache Cassandra is an open source NoSQL distributed database [EB/OL]. [2023-04-10]. https://cassandra.apache.org/

[14] Sanaboyina T P. Performance evaluation of time series databases based on energy consumption [D]. Karlskrona, Sweden: Blekinge Institute of Technology, 2016

[15] Fadhel M, Sekerinski E, Yao Shucai. A comparison of time series databases for storing water quality data[C] //Proc of the 12th Int Conf on Interactive Mobile Communication Technologies and Learning. Berlin: Springer, 2019: 302−313

[16] The Cloud Native Computing Foundation (CNCF). Prometheus is a free software application used for event monitoring and alerting[EB/OL]. [2023-04-10].https://prometheus.io/

[17] Grzesik P, Mrozek D. Comparative analysis of time series databases in the context of edge computing for low power sensor networks[C] // Proc of the 20th Int Conf on Computational Science. Berlin: Springer, 2020: 371−383

[18] Riak. Riak TS is a distributed NoSQL key/value store optimized for time series data[EB/OL]. [2023-04-10].https://riak.com/products/riak-ts/index.html

[19] The PostgreSQL Global Development Group. PostgreSQL is a free and open-source relational database management system [EB/OL]. [2023-04-10].https://www.postgresql.org/

[20] SQLite Consortium. SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine [EB/OL]. [2023-04-10]. https://sqlite.org/index.html

[21] Brillinger D R. Time Series: Data Analysis and Theory[M]. Philadelphia, PA: SIAM, 2001

[22] International Electrotechnical Commission (IEC). IEC 61400-25-6: 2016: Wind energy generation systems-part 25-6: Communications for monitoring and control of wind power plants-Logical node classes and data classes for condition monitoring [S]. Geneva, Switzerland: International Electrotechnical Commission, 2016

[23] Dotis-Georgiou A. When you want holt-winters instead of machine learning[EB/OL]. [2023-04-10]. https://www.influxdata.com/blog/when-you-want-holt-winters-instead-of-machine-learning/

[24] DolphinDB. DolphinDB database[EB/OL]. [2023-04-10].https://dolphindb.com/

[25] Lampson B, Sturgis H E. Crash recovery in a distributed data storage system[R]. Palo Alto, CA: Xerox Palo Alto Research Center, 1979

[26] Gray J N. Notes on data base operating systems[M] //Operating Systems: An Advanced Course. Berlin: Springer, 2005: 393-481

[27] Bernstein P A, Hadzilacos V, Goodman N. Concurrency Control and Recovery in Database Systems[M]. Reading, MA: Addison-Wesley, 1987

[28] InfluxData. InfluxDB edge data replication[EB/OL]. [2023-04-10]. https://www.influxdata.com/products/influxdb-edge-data-replication/

[29] Yang Yang, Cao Qiang, Jiang Hong. EdgeDB: An efficient time-series database for edge computing[J]. IEEE Access, 2019, 7: 142295−142307

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

[31] Lu Lanyue, Pillai T S, Gopalakrishnan H, et al. WiscKey: Separating keys from values in SSD-conscious storage[J]. ACM Transactions on Storage, 2017, 13(1): 1−28

[32] Raju P, Kadekodi R, Chidambaram V, et al. Pebblesdb: Building key-value stores using fragmented log-structured merge trees[C] //Proc of the 26th Symp on Operating Systems Principles (SOSP). New York, ACM, 2017: 497−514

[33] Doekemeijer K, Trivedi A. Key-Value stores on flash storage devices: A survey[J]. arXiv preprint, arXiv: 2205. 07975, 2022

[34] Daim T U, Ploykitikoon P, Kennedy E, et al. Forecasting the future of data storage: Case of hard disk drive and flash memory[J]. Foresight, 2008, 10(5): 34−49

[35] Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data[J]. ACM Transactions on Computer Systems, 2008, 26(2): 1−26

[36] Ghemawat S, Dean J. LevelDB database[EB/OL]. [2023-04-10].https://github.com/google/leveldb

[37] Agrawal N, Prabhakaran V, Wobber T, et al. Design tradeoffs for SSD performance[C] //Proc of the 2008 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2008: 57–70

[38] Yang Mingchang, Chang Yuming, Tsao C W, et al. Garbage collection and wear leveling for flash memory: Past and future[C] //Proc of the 2014 Int Conf on Smart Computing. Piscataway, NJ: IEEE, 2014: 66−73

[39] Caulfield A M, De A, Coburn J, et al. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories[C] //Proc of the 43rd Annual IEEE/ACM Int Symp on Microarchitecture (MICRO). Piscataway, NJ: IEEE, 2010: 385−395

[40] Condit J, Nightingale E B, Frost C, et al. Better I/O through byte-addressable, persistent memory[C] //Proc of the 22nd ACM Symp on Operating Systems Principles (SOSP). New York: ACM, 2009: 133−146

[41] Bender M A, Farach-Colton M, Johnson R, et al. Don't thrash: How to cache your Hash on flash[J]. Proceedings of the VLDB Endowment, 2012, 5(11): 1627−1637

[42] Rottenstreich O, Keslassy I. The Bloom paradox: When not to use a Bloom filter[J]. IEEE/ACM Transactions on Networking, 2014, 23(3): 703−716

[43] Fan Bin, Andersen D G, Kaminsky M, et al. Cuckoo filter: Practically better than Bloom[C] //Proc of the 10th ACM Int Conf on Emerging Networking Experiments and Technologies (CoNEXT). New York: ACM, 2014: 75−88

[44] Graf T M, Lemire D. Xor filters: Faster and smaller than Bloom and cuckoo filters[J]. Journal of Experimental Algorithmics, 2020, 25: 1−16

[45] Pugh W. Skip lists: A probabilistic alternative to balanced trees[J]. Communications of the ACM, 1990, 33(6): 668−676

[46] Rothermel K, Mohan C. ARIES/NT: A recovery method based on write-ahead logging for nested transactions[C] //Proc of the 15th Int Conf on Very Large Data Bases (VLDB). San Francisco, CA: Morgan Kaufmann, 1989: 337–346

[47] Balmau O, Didona D, Guerraoui R, et al. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores[C] //Proc of the 2017 USENIX Annual Technical Conf (USENIX ATC 17). Berkeley, CA: USENIX Association, 2017: 363−375

[48] Meta. RocksDB: A persistent key-value store for flash and RAM storage [EB/OL]. [2023-04-10].https://rocksdb.org

[49] Wikimedia Foundation. Locality of reference[EB/OL]. [2023-04-10].https://en.wikipedia.org/wiki/Locality_of_reference

[50] InfluxData. InfluxDB storage engine[EB/OL]. [2023-04-10].https://archive.docs.influxdata.com/influxdb/v0.11/concepts/storage_engine/

[51] The Apache Software Foundation (ASF). Apache HBase is the Hadoop database, a distributed, scalable, big data store [EB/OL]. [2023-04-10].https://hbase.apache.org/

[52] Hawkins B. KairosDB: Fast time series database on Cassandra [EB/OL]. [2023-04-10].https://kairosdb.github.io/

[53] The H2 Database Team. H2 is an embeddable RDBMS written in Java. [EB/OL]. [2023-04-10].https://github.com/h2database/h2database

[54] The Apache Software Foundation (ASF). Apache IoTDB [EB/OL]. [2023-04-10].https://iotdb.apache.org/

[55] InfluxData. Time series index (TSI) overview[EB/OL]. [2023-04-10].https://docs.influxdata.com/influxdb/v1.8/concepts/time-series-index/

[56] TAOS Data. TDengine is an open source, high-performance, cloud native time-series database[EB/OL]. [2023-04-10].https://tdengine.com/

[57] The QuestDB Team. QuestDB is an open-source time-series database for high throughput ingestion and fast SQL queries with operational simplicity [EB/OL]. [2023-04-10].https://questdb.io/

[58] The VictoriaMetrics Team. VictoriaMetrics: The high-performance, open source time series database & monitoring solution[EB/OL]. [2023-04-10].https://victoriametrics.com/

[59] Shi Xuanhua, Feng Zezhao, Li Kaixi, et al. ByteSeries: An in-memory time series database for large-scale monitoring systems[C] //Proc of the 11th ACM Symp on Cloud Computing (SoCC). New York: ACM, 2020: 60−73

[60] DataStax. How is data maintained[EB/OL]. [2023-04-10].https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/dml/dmlHowDataMaintain.html#dmlHowDataMaintain__twcs-compaction

[61] TDengine. High cardinality in time series data[EB/OL]. [2023-04-10].https://tdengine.com/tsdb/high-cardinality-in-time-series-data/

[62] Ilyushchenko V. How databases handle 10 million devices in high-cardinality benchmarks[EB/OL]. [2023-04-10].https://questdb.io/blog/2021/06/16/high-cardinality-time-series-data-performance/

[63] O'neil E J, O'neil P E, Weikum G. The LRU-K page replacement algorithm for database disk buffering[J]. ACM SIGMOD Record, 1993, 22(2): 297−306

[64] Dix P. Announcing InfluxDB IOx - The future core of InfluxDB built with rust and arrow[EB/OL]. [2023-04-10].https://www.influxdata.com/blog/announcing-influxdb-iox/

[65] Comer D. Ubiquitous B-tree[J]. ACM Computing Surveys, 1979,

11(2): 121−137

[66] Flynn M J. Some computer organizations and their effectiveness[J]. IEEE Transactions on Computers, 1972, 100(9): 948−960

[67] VictoriaMetrics. VictoriaMetrics cardinality explorer[EB/OL]. [2023-04-10].https://victoriametrics.com/blog/cardinality-explorer/

[68] De La Briandais R. File searching using variable length keys[C] //Proc of the 1959 Western Joint Computer Conf. New York: ACM, 1959: 295−298

[69] Samulowitz H, Reddy C, Sabharwal A, et al. Snappy: A simple algorithm portfolio[C] //Proc of the 16th Int Conf on Theory and Applications of Satisfiability Testing (SAT 2013). Berlin: Springer, 2013: 422−428

[70] The Apache Software Foundation (ASF). Aligned timeseries [EB/OL]. [2023-04-10].https://iotdb.apache.org/UserGuide/Master/Data-Concept/Data-Model-and-Terminology.html#aligned-timeseries

[71] Yao Ting, Zhang Yiwen, Wan Jiguang, et al. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with a matrix container in NVM[C] // Proc of the 2020 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2020: 17−31

[72] Narayanan D, Thereska E, Donnelly A, et al. Migrating server storage to SSDs: Analysis of tradeoffs[C] //Proc of the 4th ACM European Conf on Computer Systems (EuroSys). New York: ACM, 2009: 145-158

[73] RocksDB. Strategies to reduce write amplification[EB/OL]. [2023-04-10].https://github.com/facebook/rocksdb/issues/19

[74] Luo Chen, Carey M J. LSM-based storage techniques: A survey[J]. The VLDB Journal, 2020, 29(1): 393−418

[75] Kerrisk M. Fsync: Standard C library[EB/OL]. [2023-04-10]. https://man7.org/linux/man-pages/man2/fdatasync.2.html

[76] Valialkin A. WAL usage looks broken in modern time series databases [EB/OL]. [2023-04-10].https://valyala.medium.com/wal-usage-looks-broken-in-modern-time-series-databases-b62a627ab704

[77] Dias L B, Silva D S, de Sousa Junior R T, et al. C* DynaConf: An Apache Cassandra auto-tuning tool for Internet of things data[C] //Proc of the 6th Int Conf on Internet of Things, Big Data and Security. Setúbal, Portugal: SciTePress, 2021: 92−102

[78] Tangwongsan K, Hirzel M, Schneider S. Optimal and general out-of-order sliding-window aggregation[J]. Proceedings of the VLDB Endowment, 2019, 12(10): 1167−1180

[79] Grulich P M, Traub J, Breß S, et al. Generating reproducible out-of-order data streams[C] //Proc of the 13th ACM Int Conf on Distributed and Event-based Systems. New York: ACM, 2019: 256−257

[80] Weiss W, Jimenez V J E, Zeiner H. Dynamic buffer sizing for out-of-order event compensation for time-sensitive applications[J]. ACM Transactions on Sensor Networks, 2020, 17(1): 1−23

[81] Zhang Lingzhe, Huang Xiangdong, Qiao Jialin, et al. Two-stage file compaction framework by log-structured merge-tree for time series data[J]. Journal of Computer Application, 2021, 41(3): 618−622 (in Chinese)
(张凌哲, 黄向东, 乔嘉林, 等. 面向时序数据的两阶段日志结构合并树文件合并框架 [J]. 计算机应用, 2021, 41(3): 618−622)

[82] Kang Yuyuan, Huang Xiangdong, Song Shaoxu, et al. Separation or not: On handing out-of-order time-series data in leveled LSM-tree[C] //Proc of the 38th Int Conf on Data Engineering (ICDE). Piscataway, NJ: IEEE, 2022: 3340−3352

[83] Huang Xiangdong, Wang Jianmin, Wong R, et al. Pisa: An index for aggregating big time series data[C] //Proc of the 25th ACM Int on Conf on Information and Knowledge Management (CIKM). New York: ACM, 2016: 979−988

[84] Qiao Jialin, Huang Xiangdong, Wang Jianmin, et al. Dual-PISA: An index for aggregation operations on time series data[J]. Information Systems, 2020, 87(C): 101427

[85] Zhao Dongming, Qiu Yuanhui, Kang Rui, et al. Physical metadata management in Apache IoTDB for aggregate queries[J]. Journal of Software, 2022, 34(3): 1027−1048 (in Chinese)
(赵东明, 邱圆辉, 康瑞, 等. 面向聚合查询的 Apache IoTDB 物理元数据管理 [J]. 软件学报, 2022, 34(3): 1027−1048)

[86] InfluxData. Data retention in InfluxDB[EB/OL]. [2023-04-10].https://docs.influxdata.com/influxdb/v2.7/reference/internals/data-retention/

[87] Xiao Jinzhao, Huang Yuxiang, Hu Changyu, et al. Time series data encoding for efficient storage: A comparative analysis in Apache IoTDB[J]. Proceedings of the VLDB Endowment, 2022, 15(10): 2148−2160

[88] The Apache Software Foundation (ASF). Encoding methods [EB/OL]. [2023-04-10].https://iotdb.apache.org/UserGuide/Master/Data-Concept/Encoding.html

[89] Golomb S. Run-length encodings[J]. IEEE Transactions on Information Theory, 1966, 12(3): 399−401

[90] Campobello G, Segreto A, Zanafi S, et al. RAKE: A simple and efficient lossless compression algorithm for the Internet of things[C] //Proc of the 25th European Signal Processing Conf. Piscataway, NJ: IEEE, 2017: 2581−2585

[91] Spiegel J, Wira P, Hermann G. A comparative experimental study of lossless compression algorithms for enhancing energy efficiency in smart meters[C] //Proc of the 16th IEEE Int Conf on Industrial Informatics. Piscataway, NJ: IEEE, 2018: 447−452

[92] Blalock D, Madden S, Guttag J. Sprintz: Time series compression for the Internet of things[J]. Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, 2018, 2(3): 1−23

[93] Welch T A. A technique for high-performance data compression[J]. Computer, 1984, 17(6): 8−19

[94] Howard P G, Vitter J S. Parallel lossless image compression using Huffman and arithmetic coding[C] //Proc of the Data Compression Conf. Piscataway, NJ: IEEE, 1992: 299−308

[95] Anh V N, Moffat A. Index compression using 64-bit words[J]. Software: Practice and Experience, 2010, 40(2): 131−147

[96] AVEVA. AVEVA PI server[EB/OL]. [2023-04-10].https://www.aveva.com/en/products/aveva-pi-server/

[97] Bristol E H. Swinging door trending: Adaptive trend recording[C] //Proc of the ISA National Conf. 1990: 749−754. [2023-04-10].

https://cir.nii.ac.jp/crid/1574231875546173824

[98] Feng Xiaodong, Cheng Changling, Liu Changling, et al. An improved process data compression algorithm[C] //Proc of the 4th World Congress on Intelligent Control and Automation. Piscataway, NJ: IEEE, 2002: 2190−2193

[99] Gailly J L. GNU Gzip[EB/OL]. [2023-04-10].https://www.gnu.org/software/gzip/

[100] Bartík M, Ubik S, Kubalik P. LZ4 compression algorithm on FPGA[C] //Proc of the IEEE Int Conf on Electronics, Circuits, and Systems. Piscataway, NJ: IEEE, 2015: 179−182

[101] Cheng Hongze. Compressing time series data[EB/OL]. [2023-04-10].https://tdengine.com/compressing-time-series-data/

[102] Jensen S K, Pedersen T B, Thomsen C. ModelarDB: Modular model-based time series management with Spark and Cassandra[J]. Proceedings of the VLDB Endowment, 2018, 11(11): 1688−1701

[103] Eichinger F, Efros P, Karnouskos S, et al. A time-series compression technique and its application to the smart grid[J]. The VLDB Journal, 2015, 24(2): 193−218

[104] Chiarot G, Silvestri C. Time series compression survey[J]. ACM Computing Surveys, 2023, 55(10): 1−32

[105] Yu Xinyang, Peng Yanqing, Li Feifei, et al. Two-level data compression using machine learning in time series database[C] //Proc of the 36th Int Conf on Data Engineering (ICDE). Piscataway, NJ: IEEE, 2020: 1333−1344

[106] Schizofreny. Middle-out compression for time-series data[EB/OL]. [2023-04-10].https://github.com/schizofreny/middle-out

[107] TAOS Data. Tiered storage[EB/OL]. [2023-04-10].https://docs.tdengine.com/tdinternal/arch/#tiered-storage

[108] Thulab. iot-benchmark[EB/OL]. [2023-04-10].https://github.com/thulab/iot-benchmark

[109] Timescale. TimescaleDB vs. InfluxDB: Purpose built differently for time-series data[EB/OL]. [2023-04-10].https://www.timescale.com/blog/timescaledb-vs-influxdb-for-time-series-data-timescale-influx-sql-nosql-36489299877/

[110] Timescale. Time series benchmark suite[EB/OL]. [2023-04-10].https://github.com/timescale/tsbs

[111] Valialkin A. High-cardinality TSDB benchmarks: VictoriaMetrics vs TimescaleDB vs InfluxDB[EB/OL]. [2023-04-10].https://valyala.medium.com/high-cardinality-tsdb-benchmarks-victoriametrics-vs-timescaledb-vs-influxdb-13e6ee64dd6b

[112] TAOS Data. DevOps performance comparison: InfluxDB and TimescaleDB vs TDengine[EB/OL]. [2023-04-10]. https://tdengine.com/devops-performance-comparison-influxdb-and-timescaledb-vs-tdengine/

[113] The QuestDB Team. Comparing InfluxDB, TimescaleDB, and QuestDB Time-Series Databases[EB/OL]. [2023-04-10]. https://questdb.io/blog/comparing-influxdb-timescaledb-questdb-time-series-databases/

[114] Liu Rui, Yuan Juan. Benchmarking time series databases with IoTDB-benchmark for IoT scenarios[J]. arXiv preprint, arXiv: 1901.08304, 2019

[115] The Transaction Processing Performance Council (TPC). TPCx-IoT[EB/OL].[2023-04-10].https://www.tpc.org/tpcx-iot/default5.asp

[116] Poess M, Nambiar R, Kulkarni K, et al. Analysis of TPCx-IoT: The first industry standard benchmark for IoT gateway systems[C] //Proc of the 34th Int Conf on Data Engineering (ICDE). Piscataway, NJ: IEEE, 2018: 1519−1530

[117] InfluxData. influxdb-comparisons[EB/OL]. [2023-04-10].https://github.com/influxdata/influxdb-comparisons

[118] Hao Yuanzhe, Qin Xiongpai, Chen Yueguo, et al. TS-Benchmark: A benchmark for time series databases[C] //Proc of the 37th Int Conf on Data Engineering (ICDE). Piscataway, NJ: IEEE, 2021: 588−599

[119] Radford A, Metz L, Chintala S. Unsupervised representation learning with deep convolutional generative adversarial networks[J]. arXiv preprint, arXiv: 1511. 06434, 2016

[120] Shah B, Jat P, Sashidhar K. Performance study of time series databases[J]. arXiv preprint, arXiv: 2208. 13982, 2022

[121] The Apache Software Foundation (ASF). Druid is a high performance, real-time analytics database[EB/OL]. [2023-04-10].https://druid.apache.org/

[122] Mei Fei, Cao Qiang, Jiang Hong, et al. SifrDB: A unified solution for write-optimized key-value stores in large datacenter[C]//Proc of the Symp on Cloud Computing (SoCC). New York: ACM, 2018: 477−489

[123] Chen Feng, Hou Binbing, Lee R. Internal parallelism of flash memory-based solid-state drives[J]. ACM Transactions on Storage, 2016, 12(3): 1−39

[124] Chen Feng, Lee R, Zhang Xiaodong. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing[C] //Proc of the 17th Int Symp on High Performance Computer Architecture (HPCA). Piscataway, NJ: IEEE, 2011: 266−277

[125] Wang Peng, Sun Guangyu, Jiang Song, et al. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD[C/OL] //Proc of the 9th European Conf on Computer Systems (EuroSys). New York: ACM, 2014[2023-04-10].https://dl.acm.org/doi/10.1145/2592798.2592804

[126] Kannan S, Bhat N, Gavrilovska A, et al. Redesigning LSMs for nonvolatile memory with NoveLSM[C] //Proc of the 2018 USENIX Annual Technical Conf (USENIX ATC 18). Berkeley, CA: USENIX Association, 2018: 993−1005

[127] You Litong, Wang Zhenjie, Huang Linpeng. A log-structured key-value store based on non-volatile memory[J]. Journal of Computer Research and Development, 2018, 55(9): 2038−2049 (in Chinese)
(游理通, 王振杰, 黄林鹏. 一个基于日志结构的非易失性内存键值存储系统 [J]. 计算机研究与发展, 2018, 55(9): 2038−2049)

[128] Dong Haowen, Zhang Chao, Li Guoliang, et al. A survey of cloud-native databases[J/OL]. Journal of Software, 2023[2023-04-10]. http://www.jos.org.cn/jos/article/abstract/6952 (in Chinese)
(董昊文, 张超, 李国良, 等. 云原生数据库综述 [J/OL]. 软件学报, 2023[2023-04-10]. http://www.jos.org.cn/jos/article/abstract/6952)

[129] Meng Xiaofeng, Ma Chaohong, Yang Chen. Survey on machine learning for database systems[J]. Journal of Computer Research and Development, 2019, 56(9): 1803−1820 (in Chinese)
(孟小峰, 马超红, 杨晨. 机器学习化数据库系统研究综述 [J]. 计算机研究与发展, 2019, 56(9): 1803−1820)

Liu Shuai, born in 1992. PhD candidate. His main research interest includes database systems and technologies.

刘　帅, 1992 年生. 博士研究生. 主要研究方向为数据库系统与技术.

Qiao Ying, born in 1973. PhD, professor. Her main research interests include real-time intelligence and real-time scheduling.

乔　颖, 1973 年生. 博士, 研究员. 主要研究方向为实时智能、实时调度.

Luo Xiongfei, born in 1977. PhD, senior engineer. His main research interest includes real-time data management and analysis technology.

罗雄飞, 1977 年生. 博士, 高级工程师. 主要研究方向为实时数据管理与分析技术.

Zhao Yijing, born in 1994. PhD candidate. Her main research interests include intelligent database technology and data mining.

赵怡婧, 1994 年生. 博士研究生. 主要研究方向为智能数据库技术、数据挖掘.

Wang Hong'an, born in 1963. PhD, professor, PhD supervisor. Senior member of CCF. His main research interests include natural human-computer interaction and real-time intelligence.

王宏安, 1963 年生. 博士, 研究员, 博士生导师, CCF 高级会员. 主要研究方向为自然人机交互、实时智能.