内存中大数据管理和处理:调查

Hao Zhang、Gang Chen,IEEE 会员,Beng Chin Ooi,IEEE 院士,
IEEE 会员 Kian-Lee Tan 和 IEEE 会员张美惠

摘要 不断增长的主内存容量推动了内存大数据管理和处理的发展。通过消除磁盘 I/O 瓶颈,现在可以支持交互式数据分析。然而,内存系统对其他开销来源更加敏感,而这些开销在传统的 I/O 限制的基于磁盘的系统中并不重要。在内存环境中处理容错和一致性等一些问题也更具挑战性。我们正在见证一场利用主存作为数据存储层的数据库系统设计革命。其中许多研究都集中在几个方面:现代 CPU 和内存层次结构利用率、时间/空间效率、并行性和并发控制。在本次调查中,我们旨在对各种内存数据管理和处理建议和系统进行全面审查,包括数据存储系统和数据处理框架。我们还全面介绍了内存管理中的重要技术,以及为实现高效的内存数据管理和处理而需要考虑的一些关键因素。

索引词主内存、DRAM、关系数据库、分布式数据库、查询处理

什么

1简介

大数据的爆炸式增长引发了大量研究

并发支持超低延迟服务和实时数据分析的系统。由于对硬盘的高访问延迟,现有的基于磁盘的系统不再能够提供及时的响应。不可接受的性能最初是亚马逊、谷歌、Facebook 和 Twitter 等互联网公司遇到的,但现在也成为其他希望提供有意义的实时服务(例如,实时竞价,广告、社交游戏)。例如,贸易公司需要检测交易价格的突然变化并立即(在几毫秒内)做出反应,这是使用传统的基于磁盘的处理/存储系统无法实现的。为了满足在毫秒内分析大量数据和服务请求的严格实时要求,需要一个始终将数据保存在随机存取存储器 (RAM) 中的内存系统/数据库。

在过去的十年中,多核处理器和大量主内存的可用性正在以直线下降的成本创造新的突破,使得构建内存系统成为可能,其中大部分(如果不是全部)数据库都适合在记忆中。例如,内存存储容量和带宽大约每三年翻一番,而其价格每五年下降 10 倍。同样,SSD 等非易失性存储器 (NVM) 也取得了重大进展,相变存储器 (PCM) 等各种 NVM 也即将推出。此类设备每秒的 I/O 操作数远远大于硬盘。现代高端服务器通常有多个插槽,每个插槽可以有数十或数百GB的DRAM,以及数十个内核,总计一台服务器可能有数TB的DRAM和数百个内核。此外,在分布式环境中,可以将来自大量服务器节点的内存聚合到聚合内存能够保留各种大型应用程序的所有数据的程度(例如,Facebook [2]).

Jim Gray 的“内存是新磁盘,磁盘是新磁带”的见解在今天正在成为现实 [1] 我们正在见证一种趋势,内存最终将取代磁盘,而磁盘的作用必然变得更具存档性。在

数据库系统在过去几十年里一直在发展,主要受硬件进步、大量数据的可用性、以前所未有的速度收集数据、新兴应用程序等驱动。

基于应用程序域(即依赖于关系数据、基于图形的数据、流数据的应用程序),数据管理系统的格局越来越分散。

图 1 显示了用于基于磁盘和内存操作的最先进的商业和学术系统。在本次调查中,我们关注内存系统;读者可以参考 [3] 对基于磁盘的系统的调查。

在业务运营中,速度不是可选项,而是必须的。因此,各种途径都被用来进一步提高性能,包括减少对硬盘的依赖,添加更多内存以使更多数据驻留在

H. Zhang、BC Ooi 和 K.-L。 Tan 与新加坡国立大学计算机学院,新加坡 117417。

E-mail: {zhangh, ooibc, tankl}@comp.nus.edu.sg.
G. Chen (浙江大学计算机学院, 杭州 310027)邮箱:cg@cs.zju.edu.cn。

M. Zhang 任职于新加坡科技设计大学信息系统技术与设计系,新加坡 487372。

电邮:meihui_zhang@sutd.edu.sg。

手稿于 2015 年 1 月 16 日收到; 2015 年 4 月 22 日修订; 4月25日接受 2015. 出版日期 2015 年 4 月 28 日;当前版本日期 2015 年 6 月 1 日。
推荐由 J. Pei 接受。
有关获得本文转载的信息,请发送电子邮件至:reprints@ieee.org,并参考下面的数字对象标识符。
数字对象标识符号10.1109/TKDE.2015.2427795

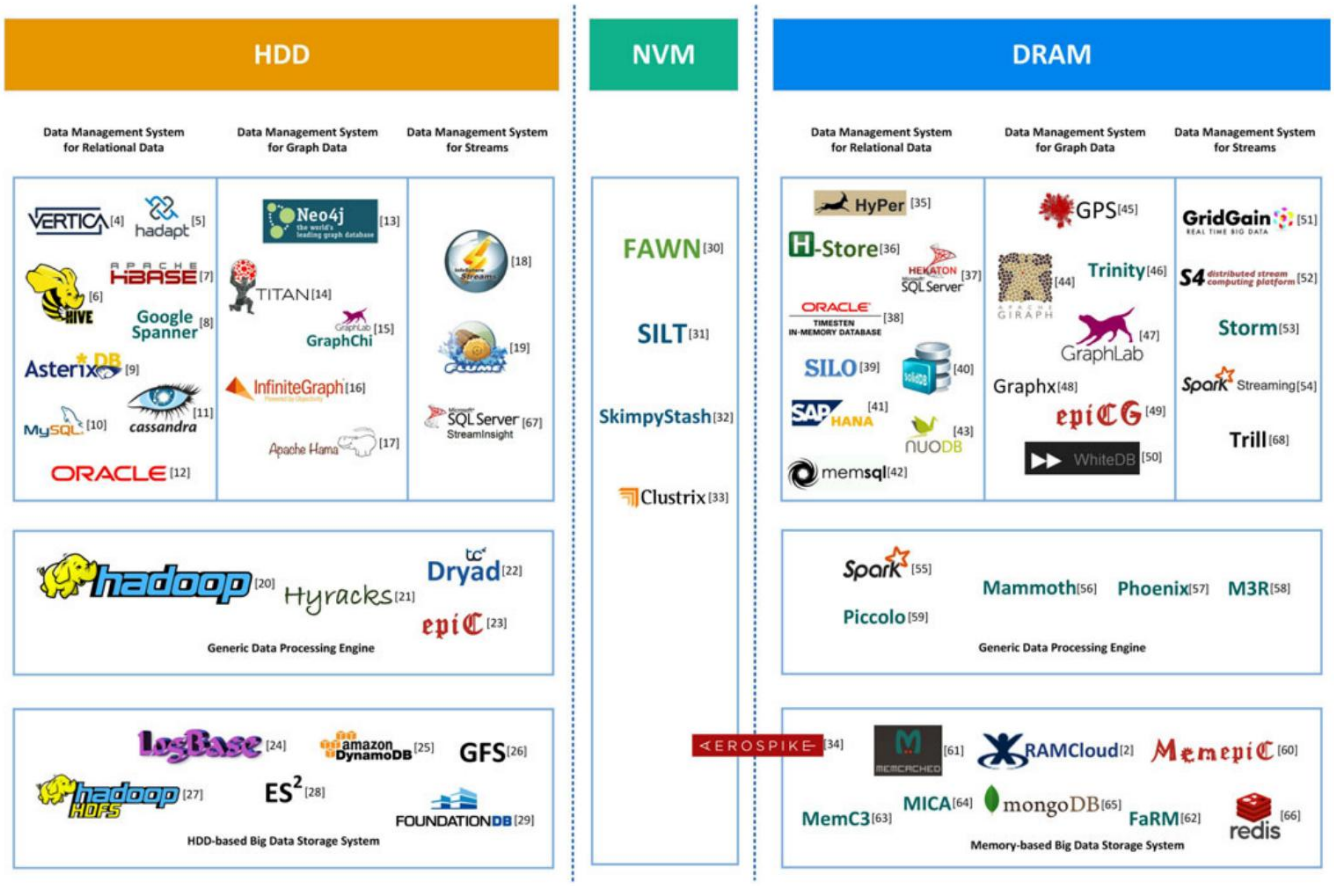

图 1. 基于磁盘和内存数据管理系统的（部分)景观。

内存,甚至部署内存系统,所有数据都可以保存在内存中。早在 1980 年代 [69]、[70]、[71]、[72]、[73] 就已经对内存数据库系统进行了研究。然而,最近硬件技术的进步已经使许多早期的工作失效,并重新产生了将整个数据库托管在内存中的兴趣,以提供更快的访问和实时分析 [35]、[36]、[55]、 [74]、[75]、[76]。大多数商业数据库供应商最近都引入了内存数据库处理,以支持完全在内存中的大型应用程序 [37]、[38]、[40]、[77]。高效的内存数据管理是各种应用程序所必需的 [78]、[79]。

尽管如此,内存中数据管理仍处于起步阶段,并可能在未来几年内得到发展。

一般来说,如表 1 所总结的,内存数据管理和处理的研究集中在以下方面以提高效率或增强 ACID 属性:

索引。尽管与磁盘访问相比,内存数据访问速度非常快,但仍然需要一个高效的索引来支持点查询,以避免内存密集型扫描。为内存数据库设计的索引与为B+树等基于磁盘的数据库设计的传统索引有很大不同,因为传统索引主要关心I/O效率而不是内存和缓存利用率。基于哈希的索引通常用于键值存储,例如 Memcached [61]、Redis [66]、RAMCloud [75],并且可以进一步优化以提高缓存利用率

减少指针追逐[63]。然而,基于哈希的索引不支持范围查询,而范围查询对于数据分析至关重要,因此,也提出了基于树的索引,例如 T-Tree [80]、缓存敏感搜索树（CSS-Trees)[81]]、缓存敏感B+树（CSB+-Trees)[82]、D-Tree[83]、BD-Tree[84]、快速架构敏感树（FAST)[85]、Bw-tree[86]和自适应Radix Tree (ART) [87],其中一些也考虑指针归约。

数据布局。内存数据布局对内存使用和缓存利用率有重大影响。关系表的列式布局有助于类似扫描的查询/分析,因为它可以实现良好的缓存局部性 [41]、[88],并且可以实现更好的数据压缩 [89],但对于需要在行级别 [74]、[90]。也可以混合使用行和列布局,例如仅在一个页面内按列组织数据的 PAX [91],以及具有由多个增量行/列存储和一个主列组成的多层存储的 SAP HANA 商店,定期合并[74]。此外,还有处理内存碎片问题的建议,例如 Memcached中的基于slab的分配器[61],RAMCloud中定期清理的日志结构数据组织[75],以及更好地利用一些硬件特性（例如,位级并行性,SIMD） ,例如 BitWeaving [92] 和 ByteSlice [93]。

并行性。一般来说,并行性分为三个层次,即数据级并行性（例如,位级

表格1

内存数据管理和处理的优化方面

Aspects	Concerns	Related Work
Index	cache consciousness, time/space efficiency	T-Tree [80], CSS-Trees [81], CSB$^+$-Trees [82], Δ-Tree [83], BD-Tree [84], FAST [85], ART [87]
Data Layout	cache consciousness, space efficiency	PAX [91], columnar layout [41], [88], HANA Hybrid Store [74], slab allocator [61], log-structure [75]
Parallelism	linear scaling, partitioning	BitWeaving [92], bit-parallel aggregation [94], SIMD sorting [95], SIMD scanning [96], [97], multi-core join [98], distributed computing [2], [55], [99], [100]
Concurrency Control/ Transaction Management	overhead, correctness	virtual snapshot [35], lock-eliding [101],transactional memory [102], [103], PALM [104], LIL [105], VLL [106], OCC [39], [107], MVCC [108], [109], DGCC [110]
Query Processing	code locality, register temporal locality, time efficiency	stored procedure [111], JIT compilation [112], [113], join [98], [114], [115], [116], [117], [118], [119], [120], [121], [122], sort [95], [123], [124]
Fault Tolerance	durability, correlated failures, availability	Copyset [125], fast recovery [126], group commit and log coalescing [37], [127], NVM [128], [129], [130], command logging [131], adaptive logging [132], remote logging [2], [40]
Data Overflow	locality, paging strategy, hot/cold classification	Anti-caching [133], Hekaton Siberia [134], data compression [74], [89], [135], virtual memory management [136], pointer swizzling [137], UVMM [138]

并行性,SIMD），1共享内存扩展并行性（线程/进程），2和无共享扩展并行性（分布式计算）。可以同时利用所有三个级别的并行性,如图 2 所示。位并行算法通过将多个数据值打包到一个 CPU 字中,充分释放现代 CPU 的周期内并行性,可以处理在一个循环中 [92]、[94]。周期内并行性能可以与打包率成正比,因为它不需要任何并发控制 (CC) 协议。

SIMD 指令可以极大地改进向量式计算,广泛用于高性能计算,也用于数据库系统 [95]、[96]、[97]。纵向扩展并行性可以利用超级计算机甚至商用计算机的多核架构[36]、[98],而横向扩展并行性在云/分布式计算中得到了高度利用[2]、[55]、[99]。纵向扩展和横向扩展并行性都需要良好的数据分区策略,以实现负载平衡并最大限度地减少跨分区协调 [100]、[139]、[140]、[141]。

并发控制/事务管理。并行控制/事务管理成为多核系统内存数据管理中一个极其重要的性能问题。基于锁/信号量的重量级机制大大降低了性能,这是由于其阻塞式方案以及集中式锁管理器和死锁检测[142]、[143]造成的开销。轻量级意向锁 (LIL) [105] 被提议在全局锁表中维护一组轻量级计数器,而不是意向锁的锁队列。 Very Lightweight Locking (VLL) [106] 通过将一个记录的所有锁定状态压缩为一对用于分区数据库的整数,进一步简化了数据结构。另一类并发控制是基于时间戳的,其中一个预定义的顺序用于保证事务的可串行性[144],例如乐观并发控制（OCC)[39]、[107]和多版本并发控制（MVCC）

[108]、[109]。此外,H-Store [36]、[101] 试图通过根据先验工作负载预先对数据库进行分区并为每个分区提供一个线程来消除单分区事务中的并发控制。 HyPer [35] 通过为基于硬件辅助虚拟快照的 OLAP 作业分叉子进程（通过 fork() 系统调用)来隔离 OLTP 和 OLAP,该虚拟快照永远不会被修改。

DGCC [110] 被提议通过基于依赖图将并发控制与执行分开来减少并发控制的开销。

Hekaton [104]、[107] 利用乐观的 MVCC 和无锁数据结构来有效地实现高并发。此外,硬件事务内存 (HTM) [102]、[103] 越来越多地用于 OLTP 的并发控制。

查询处理。查询处理正在经历内存数据库的演变。虽然传统的 Iterator-/Volcano-style 模型 [145] 有助于轻松组合任意运算符,但它会生成大量函数调用（例如,next()) ,从而导致逐出寄存器内容。

较差的代码局部性和频繁的指令错误预测进一步增加了开销 [112]、[113]。粗粒度存储过程（例如事务级)可以用来缓解这个问题[111],动态编译（Just-in-Time)是另一种实现更好的代码和数据局部性的方法[112],[113]].还可以通过优化特定查询操作来实现性能提升

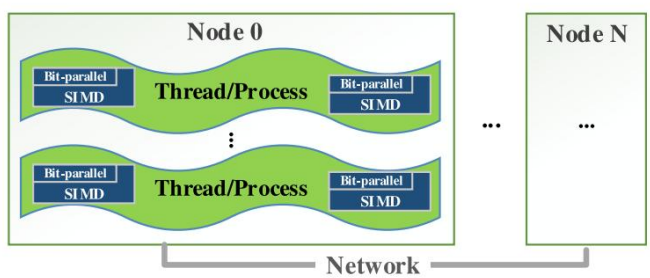

图 2. 三个级别的并行性。

1. 这里的数据级并行既包括数据打包实现的位级并行,也包括SIMD实现的字级并行。

2. GPGPU 和 Xeon Phi 等加速器也被视为共享内存纵向扩展并行性。

例如连接 [98]、[114]、[115]、[116]、[117]、[118]、[119]、[120]、[121]、[122] 和排序 [95]、[123], [124]。
容错。 DRAM 是易失性的,因此容错机制对于保证数据持久性以避免数据丢失以及在出现故障(例如电源、软件或硬件故障)时确保事务一致性至关重要。传统的预写日志记录 (WAL) 也是内存数据库系统中使用的事实上的方法 [35]、[36]、[37]。但是内存存储的数据易变性使得没有必要应用任何持久的撤消日志记录 [37],[131] 或在某些情况下完全禁用它 [111]。为了消除由日志记录引起的潜在 I/O 瓶颈,采用组提交和日志合并 [37]、[127] 和远程日志记录 [2]、[40] 来优化日志记录效率。 SSD 和 PCM 等新硬件技术被用来提高 I/O 性能 [128]、[129]、[130]。最近的研究建议使用命令日志记录 [131],它只记录操作而不是更新的数据,这在传统的 ARIES 日志记录 [146] 中使用。 [132]研究如何自适应地在这两种策略之间切换。为了加快恢复过程,必须定期检查一致的快照 [37]、[147],并且副本应该分散以预测相关故障 [125]。高可用性通常通过维护多个副本和备用服务器 [37]、[148]、[149]、[150] 或依赖故障时的快速恢复 [49]、[126] 来实现。数据可以进一步备份到更稳定的存储上,如 GPFS [151]、HDFS [27] 和 NAS [152],以进一步保护数据。

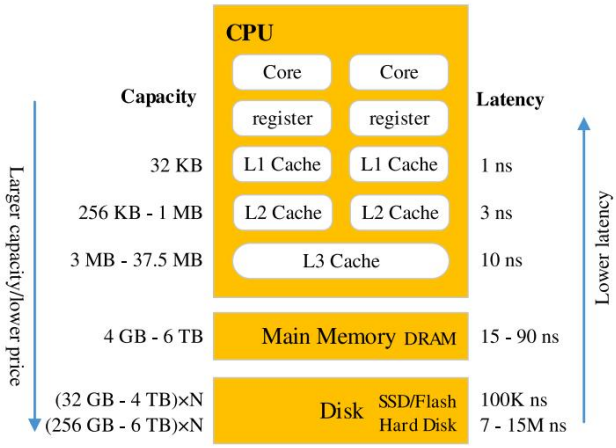

图 3. 内存层次结构。

大致分为两类,即内存数据存储系统和内存数据处理系统。

因此,其余部分组织如下。第 2 节介绍了内存数据管理的一些背景知识。我们详细阐述了内存数据存储系统,包括第 3 节中的关系数据库和 NoSQL 数据库,以及内存数据处理系统,包括第 4 节中的内存批处理和实时流处理。作为总结,我们提出在第 5 节中对本次调查中涵盖的内存数据管理系统进行定性比较。最后,我们在第 6 节中讨论了一些研究机会,并在第 7 节中得出结论。

数据溢出。尽管内存容量大幅增加,价格大幅下降,但仍然跟不上大数据时代数据的快速增长,这使得处理数据大小超过容量的数据溢出变得必不可少。主存的大小。随着硬件的进步,包含非易失性存储器 (NVM)(例如 SCM、PCM、SSD、闪存)的混合系统 [30]、[31]、[32]、[118]、[127]、[153]], [154], [155], [156] 成为实现速度的自然解决方案。或者,与传统数据库系统一样,当主内存不足时,可以采用有效的驱逐机制来替换内存中的数据。 [133]、[134]、[157]的作者提出将冷数据移动到磁盘,[136]重新组织内存中的数据并依赖操作系统进行分页,而[137]引入指针调配数据库缓冲池管理,以减轻传统数据库造成的开销,以便与完全重新设计的内存数据库竞争。

2内存中的核心技术
系统

在本节中,我们将介绍一些对于有效的内存数据管理很重要的概念和技术,包括内存层次结构、非统一形式内存访问 (NUMA)、事务内存和非易失性随机访问内存 (NVRAM)。

这些是内存数据管理系统性能严重依赖的基础知识。

2.1 内存层次结构内存层次结
构是根据访问延迟和到 CPU 的逻辑距离来定义的。一般来说,它由寄存器、缓存(通常包含L1缓存、L2缓存和L3缓存)、主内存(即RAM)和磁盘(如硬盘、闪存、SSD)从最高性能到最低性能组成.图 3 描述了内存层次结构,以及各个组件的容量和访问延迟 [158]、[159]、[160]、[161]、[162]、[163]、[164]、[165]、[166] .它表明访问较高层的数据比访问较低层要快得多,本节将分别介绍这些层。

UVMM [138] 利用硬件辅助和语义感知访问跟踪以及非阻塞内核 I/O 调度程序的混合体,以促进高效的内存管理。数据压缩也被用于减轻内存使用压力 [74]、[89]、[135]。

在现代体系结构中,除非将数据放入寄存器,否则 CPU 无法处理数据。因此,将要处理的数据必须通过每个存储层传输,直到到达寄存器。因此,每个上层都充当底层下层的缓存,以减少重复数据访问的延迟。数据密集型程序的性能在很大程度上取决于

调查的重点是大规模内存数据管理和处理策略,可以是

内存层次结构的使用。如何实现良好的空间和时间局部性通常是效率优化中最重要的。特别地,空间局部性假设相邻数据更有可能被一起访问,而时间局部性是指观察到一个项目很可能在不久的将来再次被访问。我们将分别介绍不同内存层的一些重要的效率相关属性。

2.1.1 寄存器处理器

寄存器是CPU 中的少量存储空间,机器指令可以直接对其进行操作。在普通指令中,数据首先从较低的内存层加载到用于算术或测试操作的寄存器中,然后将结果放回另一个寄存器,然后通常将其存储回主内存,或者通过相同的方式指令或后续指令。一个寄存器的长度通常等于一个CPU的字长,但也有双字,甚至更宽的寄存器(如Intel Sandy Bridge CPU微架构中256位宽的YMMX寄存器),可用于单指令多数据 (SIMD) 操作。

虽然寄存器的数量取决于架构,但寄存器的总容量远小于缓存或内存等较低层的容量。但是,从寄存器访问数据要快得多。

2.1.2 缓存

寄存器充当CPU用来执行指令的存储容器,而由于寄存器和主存之间的传输延迟高,缓存充当寄存器和主存之间的桥梁。

缓存由高速静态 RAM (SRAM) 制成,而不是通常构成主存储器的较慢且较便宜的动态 RAM (DRAM)。通常,缓存分为三级,即L1缓存、L2缓存和L3缓存(也称为末级缓存LLC),其延迟和容量不断增加。一级缓存进一步分为数据缓存(即L1-dcache)和指令缓存(即L1-icache),以避免数据访问和指令访问之间的任何干扰。如果请求的数据在缓存中,我们称之为缓存命中;否则称为高速缓存未命中。

缓存通常被细分为固定大小的逻辑缓存行,它们是不同级别缓存之间以及最后一级缓存与主存之间传输数据的原子单元。在现代架构中,缓存行通常为 64 字节长。通过按缓存行填充缓存,可以利用空间局部性来提高性能。主存和高速缓存之间的映射由几种策略决定,即直接映射ping、N路集合关联和完全关联。使用直接映射,内存中的每个条目(一个缓存行)只能放在缓存中的一个位置,这使得寻址速度更快。在完全关联策略下,每个条目可以放在任何地方,从而减少缓存未命中。 N 路关联策略是直接映射和完全关联之间的折衷 它允许内存中的每个条目位于缓存中的 N 个位置中的任意一个,这称为缓存集。实践中经常使用N-way associative,映射在缓存集方面是确定性的。

此外,大多数架构通常采用最近最少使用(LRU)替换策略来在没有足够空间时驱逐缓存行。这种方案本质上是利用时间局部性来提高性能。

如图 3 所示,访问缓存的延迟比访问主存的延迟要短得多。为了获得良好的 CPU 性能,我们必须保证高缓存命中率,从而减少高延迟内存访问。在设计内存管理系统时,重要的是要利用缓存的空间和时间局部性的特性。例如,顺序访问内存会比随机访问更快,而且将经常访问的对象驻留在缓存中也会更好。现代 CPU 的预取策略强化了顺序内存访问的优势。

2.1.3 主存和磁盘主存也称为内存储器,

与磁盘等外部设备不同,它可以直接寻址并可能被CPU 访问。主存储器通常由易失性 DRAM 制成,在没有高速缓存的情况下会产生与随机访问等效的延迟,但在电源关闭时会丢失数据。最近,DRAM 变得便宜且大到足以使内存数据库可行。

尽管内存成为新的磁盘 [1],但 DRAM 的易失性使得仍然需要磁盘 3 来备份数据的情况很常见。主存和磁盘之间的数据传输以页为单位进行,一方面利用了数据的空间局部性,另一方面最大限度地减少了磁盘寻道的高延迟导致的性能下降。页通常是磁盘扇区的倍数4 ,是硬盘的最小传输单位。在现代架构中,操作系统通常保留一个缓冲区,它是主内存的一部分,以使内存和磁盘之间的通信更快。5缓冲区主要用于弥合 CPU 和磁盘之间的性能差距。它通过缓冲写入来提高磁盘 I/O 性能,以消除每次写入操作的代价高昂的磁盘寻道时间,并缓冲读取以快速响应后续对相同数据的读取。从某种意义上说,缓冲区之于磁盘就像缓存之于内存。

它还公开了空间和时间局部性,这是有效处理磁盘 I/O 的一个重要因素。

2.2 Memory Hierarchy Utilization本节从
三个角度回顾了相关工作 register-conscious optimization, cache-conscious optimization and disk I/O optimization。

2.2.1 寄存器意识优化寄存器相关的优化通常在编

译器和汇编语言编程中很重要,这需要

3.这里指的是硬盘。

4. 现代文件系统中的一页通常为 4 KB。硬盘的每个磁盘扇区传统上是 512 字节。

5. 内核缓冲区还用于缓冲来自其他块 I/O 的数据
以固定大小的块传输数据的设备。

有效地利用有限数量的寄存器。最近对内存数据库的传统迭代器式查询处理机制提出了一些批评,因为它通常会导致代码和数据局部性较差 [36]、[112]、[167]。 HyPer 使用低级虚拟机 (LLVM) 编译器框架 [167] 将查询动态转换为机器代码,通过尽可能避免递归函数调用并尝试将数据尽可能长时间地保存在寄存器中来实现良好的代码和数据局部性尽可能 [112]。

SIMD 在超标量处理器中可用,它在宽寄存器(例如 256 位)的帮助下利用数据级并行性。 SIMD 可以显着提高性能,特别是对于矢量式计算,这在大数据分析工作中非常常见 [95]、[96]、[97]、[112]、[168]。

2.2.2 缓存意识优化缓存利用率在现代架构中变得越来越重要。几项工作负载特征研究提供了 DBMS 在现代处理器上执行的时间分解的详细分析,并报告说 DBMS 在现代架构上运行时会遇到与高内存相关的处理器停顿。这是由大量数据缓存未命中引起的 [169],占 OLTP 工作负载 [91]、[170] 的 50-70% 到 DSS 工作负载 [91]、[171] 的 90%,占总内存-相关摊位。在分布式数据库中,由于大量 TCP 网络 I/O [172],指令缓存未命中是性能下降的另一个主要来源。

为了更有效地利用缓存,一些工作着重于通过将 N 元存储模型 (NSM) 页面 [91] 中每个属性的所有值组合在一起或使用分解存储模型 (DSM) [173] 来重新组织数据布局] 或完全组织列存储中的记录 [41]、[90]、[174]、[175]。这种优化有利于通常只需要几列的 OLAP 工作负载,但对元组内缓存局部性有负面影响 [176]。

还有其他技术可以优化主要数据结构的缓存利用率,例如压缩 [177] 和着色 [178]。

此外,对于内存驻留数据结构,已经提出了各种缓存敏感索引,例如缓存敏感搜索树[81]、缓存敏感B+树[82]、快速架构敏感树[85]和自适应基数树[87]。缓存意识算法也被提议用于基本操作,例如排序(例如,突发排序)[179] 和连接 [98]、[114]。

总之,为了优化缓存利用率,以下应考虑以下重要因素:

- 缓存行长度。此特征暴露了空间局部性,这意味着访问相邻数据会更有效。
- 缓存大小。将常用数据至少保留在 L3 缓存大小内也会更有效。
- 缓存替换策略。最流行的替换策略之一是 LRU,它会在缓存未命中时替换最近最少使用的缓存行。为了获得高性能,应该利用时间局部性。

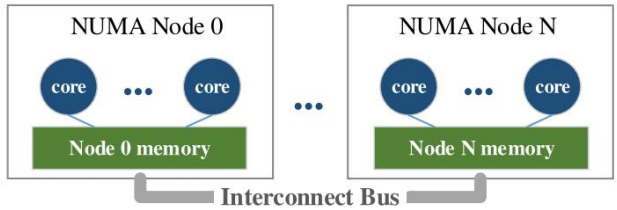

图 4. NUMA 拓扑。

2.3 非统一内存访问非统一内存访问是主内存子系统的一种架构,其中内存操作的延迟取决于执行内存操作的处理器的相对位置。从广义上讲,NUMA 系统中的每个处理器都有一个本地内存,可以以最小的延迟访问,但也可以访问至少一个具有更长延迟的远程内存,如图 4 所示。

采用 NUMA 架构的主要原因是为了提高可部署在服务器节点中的主内存带宽和总内存大小。 NUMA 允许将多个内存控制器集群到一个服务器节点中,从而创建多个内存域。

尽管早在 1980 年代就在专用系统中部署了 NUMA 系统 [185],但自 2008 年以来,所有英特尔和 AMD 处理器都集成了一个内存控制器。

因此,大多数现代多处理器系统都是 NUMA;因此,NUMA 意识正在成为主流挑战。

在数据管理系统的背景下,当前关于 NUMA 感知的研究方向可大致分为三类:

- 对数据进行分区,使对远程 NUMA 域的内存访问最小化 [115]、[186]、[187]、[188]、[189];管理 NUMA 对延迟敏感的工作负载的影响,例如 OLTP 事务 [190]、[191];跨 NUMA 域的高效数据改组 [192]。

2.3.1 数据分区数据库的工作集分区一直被用于最小化跨计算节点内和跨计算节点的不同数据域的数据传输。 Bubba [186] 是早期并行数据库系统的一个示例,它使用无共享架构扩展到数百个计算节点。它使用基于散列或范围的方法对数据进行分区,并始终仅在包含相关分区的节点中执行分析操作。 Gamma [187] 是另一个例子,它设计用于在具有 32 个处理器和 32 个磁盘驱动器的 Intel iPSC/2 超立方体的复杂架构上运行。与 Bubba 一样,Gamma 跨多个磁盘驱动器对数据进行分区,并使用基于散列的方法在分区数据之上实施连接和聚合操作。分区数据和执行提供分区并行性[193]。随着 NUMA 系统成为主流,许多研究工作已经开始明确解决 NUMA 问题,而不仅仅是依赖数据分区。此外,现代系统

表 2
STM与HTM的比较

	Performance Penalty	Hardware Support	Transaction Size	Implementations
STM	Much	Atomic Operation	Large	TinySTM [180], Clojure [181], Haskell [182]
HTM	No or Little	Cache, Bus Protocol	Small	Intel TSX [183], AMD ASF [184]

拥有越来越多的核心。最近内存拓扑和处理能力的变化确实引起了人们在 NUMA 的背景下重新审视传统处理方法的兴趣。

[115] 中提出了一种新的排序合并技术来划分连接操作,以利用具有高内存带宽和多核的 NUMA 系统。与散列连接和经典的排序合并连接相比,并行排序合并策略还并行化了最终的合并步骤,并且自然地在本地内存分区上运行。这是为了利用大多数 NUMA 系统所具有的多核架构和较大的本地内存带宽。

为 Buzzard 系统 [188] 提出了对数据库索引进行分区的建议。索引操作通常会在遍历基于树的索引期间引起频繁的指针追逐。在 NUMA 系统中,这些指针操作最终可能会从一个内存域切换到另一个内存域。

为了解决这个问题,Buzzard 提出了一种 NUMA 感知索引,该索引将基于前缀树的索引的不同部分跨不同的 NUMA 域进行分区。此外,Buzzard 使用一组专用工作线程来访问每个内存域。这保证了线程在索引遍历期间仅访问其本地内存,并通过仅使用本地比较和交换操作而不是昂贵的锁定进一步提高了性能。

在[189]中提出了对输入数据和查询执行进行分区。与计划驱动的查询执行相比,提出了一种细粒度的运行时任务调度,称为“少量查询执行”。少量驱动的查询处理策略动态地组合小块输入数据,并通过将它们分配给工作线程池以流水线方式执行它们。由于对并行性和输入数据的这种细粒度控制,morsel-driven execution 知道每个运算符的数据局部性,并且可以在本地内存域上安排它们的执行。

2.3.2 OLTP 延迟由于

NUMA 系统具有异构访问延迟,它们对通常对延迟非常敏感的 OLTP 事务提出了具有挑战性的问题。 [190] 中介绍了 NUMA 系统上不感知 NUMA 的 OLTP 部署的性能,其中许多系统被认为已实现次优和不可预测的性能。为了解决对 NUMA 感知的 OLTP 系统的需求,该论文提出了“硬件岛”,其中它将每个内存域视为一个逻辑节点,并使用 UNIX 域套接字在物理节点的 NUMA 内存域之间进行通信。

最近提出的 ATraPos [191] 是一种基于此原理构建的自适应交易处理系统。

2.3.3 数据混洗 NUMA 系

统中的数据混洗旨在通过饱和 NUMA 互连网络的传输带宽,尽可能高效地跨 NUMA 域传输数据。 [192] 中提出了一种 NUMA 感知协调环洗牌方法。为了尽可能高效地跨 NUMA 域混洗数据,所提出的方法强制跨 NUMA 域的线程以协调的方式进行通信。它将螺纹分为内圈和外圈,并在一系列循环中进行通信,在此期间内圈保持固定,而外圈旋转。这种轮换保证了内存域中的所有线程都将根据可预测的模式配对,因此 NUMA 互连的所有内存通道始终处于繁忙状态。与在域周围混洗数据的朴素方法相比,当使用具有四个域的 NUMA 系统时,此方法将传输带宽提高了四倍。

2.4 事务内存事务内存是一种共享

内存访问的并发控制机制,类似于原子数据库事务。两种类型的事务内存,即软件事务内存（STM)和硬件事务内存（HTM）,在表 2 中进行了比较。STM 在执行过程中会导致显着减慢,因此实际应用受到限制 [194],而 HTM 具有自从英特尔在其主流 Haswell 微架构 CPU [102]、[103] 中引入它以来,它因其高效的硬件辅助原子操作/事务而引起了新的关注。 Haswell HTM是基于缓存一致性协议实现的。特别是,L1 缓存用作本地缓冲区,以在缓存行的粒度上标记所有事务性读/写。对其他缓存或主内存的更改传播被推迟,直到事务提交,并且使用缓存一致性协议 [195] 检测写/写和读/写冲突。这种 HTM 设计几乎没有交易执行的开销,但有以下缺点,这使得 HTM 仅适用于小而短的交易。

事务大小受限于 L1 数据缓存的大小,通常为 32 KB。因此,不可能将数据库事务简单地作为一个单一的 HTM 事务来执行。

缓存关联性使其更容易出现错误冲突,因为一些缓存行可能会进入同一个缓存集,并且缓存行的逐出会导致事务中止,这无法通过重新启动事务来解决,因为缓存映射策略的确定性（参见第 2.1.2 节）。

HTM 事务可能由于中断事件而中止,这限制了 HTM 事务的最大持续时间。

Haswell HTM在Trans中有两个指令集

操作同步扩展 (TSX),6即硬件锁埃里森 (HLE) 和受限事务内存 (RTM)。 HLE 允许通过消除锁来乐观地执行事务,以便其他线程可以释放锁,如果事务由于数据竞争而失败则重新启动它,这主要不会产生锁定开销,并且还提供与没有 TSX 的处理器的向后兼容性. RTM 是一个新的指令集,它提供了在事务中止后指定回退代码路径的灵活性。作者 [102] 利用基于 HLE 的 HTM,通过将数据库事务划分为一组具有时间戳排序 (TSO) 并发控制的相对较小的 HTM 事务,并通过数据/索引分段将错误中止概率降至最低。 RTM 在 [103] 中得到利用,它使用三阶段乐观并发控制来协调整个数据库事务,并使用 RTM 事务保护单个数据读取（以保证序列号的一致性)和验证/写入阶段。

2.5 非显存

新兴的非易失性存储器为大容量持久高速存储器带来了前景。 NVM 的示例包括具有块粒度寻址能力的 NAND/NOR 闪存和具有字节粒度寻址能力的非易失性随机存取存储器。7闪存/SSD 已在实践中得到广泛应用,并在学术界和工业界[32]、[33],但其块粒度接口和昂贵的“擦除”操作使其仅适合充当较低级别的存储,例如更换硬盘[30]、[32]],或磁盘缓存 [198]。因此,在本次调查中,我们只关注具有字节寻址能力和与DRAM相当的性能,并且可以带到主存层甚至CPU缓存层的NVRAM。

先进的 NVRAM 技术,例如相变存储器 [199]、自旋转移力矩磁 RAM (STT MRAM) [200] 和忆阻器 [201],可以提供比传统硬盘或闪存好几个数量级的性能,并且提供与 DRAM 相同数量级的出色性能,但具有持久写入 [202]。 PCM 的读取延迟仅比 DRAM 慢 2-5 倍,STT-MRAM 和忆阻器甚至可以实现比 DRAM 更低的访问延迟 [118]、[128]、[129]、[203]。通过适当的缓存,精心设计的 PCM 也可以匹配 DRAM 的性能 [159]。此外,据推测 NVRAM 的存储密度比 DRAM 高得多,并且功耗低得多 [204]。虽然NVRAM目前只有小尺寸,每比特的成本远高于硬盘或闪存甚至DRAM,但估计到下一个十年,我们可能会拥有一个PCM与1TB 和 Memristor 100 TB,价格接近入门级硬盘[128],[129]。 NVRAM 的出现为革新数据管理和重新思考系统设计提供了一个有趣的机会。

已经表明,由于繁琐的文件系统接口（例如,文件系统缓存和昂贵的系统调用）、块粒度访问和高经济成本等带来的高开销,简单地用 NVRAM 替换磁盘并不是最优的。 [127]、[130]、[205]。相反,NVRAM 被提议与内存总线上的 DRAM 并排放置,可供普通 CPU 加载和存储,这样物理地址空间就可以在易失性和非易失性内存之间划分 [205],或者完全由非易失性存储器 [155]、[206]、[207] 构成,配备微调操作系统支持 [208]、[209]。与 DRAM 相比,NVRAM 表现出其独特的特征,例如有限的耐久性、写/读不对称性、排序的不确定性和原子性 [128]、[205]。例如,PCM 的写入延迟比其读取延迟慢一个数量级以上 [118]。

此外,没有标准机制/协议来保证 NVRAM 写入的顺序和原子性 [128],[205]。耐久性问题可以通过硬件或中间件级别的磨损均衡技术来解决[206]、[207]、[210],这可以很容易地隐藏在软件设计中,而读/写不对称、顺序和原子性写,必须在系统/算法设计中考虑[204]。

有希望的是,NVRAM 可以被设计为通用系统中的主内存,具有精心设计的架构 [155]、[206]、[207]。特别是,PCM 较长的写入延迟可以通过数据比较写入 [211]、部分写入 [207] 或以写入换取读取的专门算法/结构来解决 [118]、[204]、[212],这可以也缓解了续航问题。当前写入顺序和原子性问题的解决方案要么依赖于一些新提出的硬件原语,例如原子 8 字节写入和纪元屏障 [129]、[205]、[212],要么利用现有的硬件原语,例如缓存模式（例如,回写、写组合）、内存屏障（例如,mfence）、缓存行刷新（例如,clflush)[128]、[130]、[213],然而,这可能招致不平凡的开销。通用库和编程接口被提议将 NVRAM 公开为持久堆,以易于使用的方式启用 NVRAM,例如 NV-heaps [214]、Mnemosyne [215]、NVMalloc [216],以及恢复和持久结构 [213]、[217]。此外,文件系统支持可以透明地利用 NVRAM 作为持久存储,例如英特尔的 PMFS [218]、BPFS [205]、FRASH [219]、ConquestFS [220]、SCMFS [221],它们也利用了NVRAM 的字节寻址能力。

此外,数据库中广泛使用的特定数据结构,如 B-Tree [212]、[217],以及一些常见的查询处理运算符,如 sort 和 join [118],都开始适应和利用 NVRAM特性。

实际上,NVRAM 给数据库带来的最喜欢的好东西是它的非易失性属性,这有助于更有效的日志记录和容错机制 [127]、[128]、[129]、[130]。但是写入原子性和确定性排序应该通过精心设计的算法来保证和有效地实现,比如组

6. 2014 年 8 月,由于一个错误,英特尔在 Haswell、Haswell-E、Haswell-EP 和早期的 Broadwell CPU 上禁用了其 TSX 功能。目前 Intel 仅在具有 Broadwell 架构的 Intel Core M CPU 和新发布的具有 Haswell-EX 架构的 Xeon E7 v3 CPU 上提供 TSX [196],[197]。

7. NVM和NVRAM通常可以互换使用,没有太大区别。 NVRAM 也称为存储级内存 (SCM)、持久内存 (PM) 或非易失性字节可寻址内存 (NVBM)。

commit [127],被动组提交 [128],两步日志记录（即,首先在 DRAM 中填充日志条目,然后将其刷新到 NVRAM)[130]。还应该消除集中式日志记录瓶颈,例如,通过分布式日志记录 [128]、分散式日志记录 [130]。否则,NVRAM 带来的高性能将因遗留软件开销（例如,中央日志的中心)而降低。

3内存数据存储系统

在本节中,我们将介绍一些内存数据库,包括关系数据库和 NoSQL 数据库。我们还介绍了一类特殊的内存存储系统,即缓存系统,它用作应用服务器和底层数据库之间的缓存。在大多数关系数据库中,固有地支持 OLTP 和 OLAP 工作负载。 NoSQL 数据库中缺乏数据分析操作导致数据分析作业不可避免地需要数据传输成本 [172]。

3.1 In-Memory Relational Databases关系数据

库自1970 年代以来得到了发展和增强,并且自1990 年代初以来,关系模型在几乎所有的大规模数据处理应用程序中占据主导地位。一些广泛使用的关系数据库包括 Oracle、IBM DB2、MySQL 和 PostgreSQL。在关系数据库中,数据被组织成表/关系,并且保证了 ACID 属性。最近,出现了一种称为 NewSQL 的新型关系数据库（例如,Google Spanner [8]、H-Store [36]）。这些系统寻求为 OLTP 提供与 NoSQL 数据库相同的可扩展性,同时仍然保持传统关系数据库系统的 ACID 保证。

在本节中,我们将重点关注自 1980 年代以来一直在研究的内存关系数据库 [73]。然而,近年来兴趣激增[222]。

商业内存关系数据库的示例包括 SAP HANA [77]、VoltDB [150]、Oracle TimesTen [38]、SolidDB [40]、IBM DB2 with BLU Acceleration [223]、[224]、Microsoft Hekaton [37]、 NuoDB [43]、eXtremeDB [225]、Pivotal SQLFire [226] 和 MemSQL [42]。还有一些著名的研究/开源项目,例如 H Store [36]、HyPer [35]、Silo [39]、Crescando [227]、HYRISE [176] 和 MySQL Cluster NDB [228]。

3.1.1 H-Store / VoltDB H-

Store [36],[229] 或其商业版本 VoltDB [150] 是一种分布式的基于行的内存关系数据库,用于高性能 OLTP 处理。它的动机是两个观察结果：首先,传统基于磁盘的数据库中的某些操作,例如日志记录、闩锁、锁定、B 树和缓冲区管理操作,会产生大量的处理时间（超过 90%)[222]当移植到内存数据库时;其次,可以重新设计内存数据库处理,从而不再需要这些组件。

在H-Store中,大部分这些“重”组件被移除或优化,以实现高性能的交易处理。

H-Store 中的事务执行基于以下假设:所有（至少大部分)事务模板都是预先已知的,这些模板表示为数据库中一组已编译的存储过程。这减少了运行时事务解析的开销,还可以对数据库设计和轻量级日志策略进行预优化[131]。特别是,数据库可以更容易地分区以避免多分区事务[140],每个分区由一个站点维护,这是单线程守护进程,串行和独立地处理事务,而不需要重量级大多数情况下并发控制（例如,锁定)[101]。接下来,我们将详细阐述它的事务处理、数据溢出和容错策略。

事务处理。 H-Store 中的事务处理是在分区/站点的基础上进行的。站点是一个独立的事务处理单元,它按顺序执行事务,这使得它只有在大多数事务是单一站点的情况下才可行。这是因为如果一个交易涉及多个分区,那么所有这些站点都是顺序协作处理这个分布式交易的（通常是2PC）,因此不能独立地并行处理交易。 H-Store 设计了一种倾斜感知分区模型 园艺[140] 根据数据库模式、存储过程和样本事务工作负载自动对数据库进行分区,以最大限度地减少多分区事务的数量,同时减轻工作量中时间偏差的影响。 Horticulture 采用大邻域搜索 (LNS) 方法以引导方式探索潜在分区,其中还考虑只读表复制以减少频繁远程访问的传输成本,二级索引复制以避免广播,以及存储过程路由属性允许高效的请求路由机制。

园艺分区模型可以大大减少多分区事务的数量,但不能完全减少。因此,并发控制方案必须能够区分单分区事务和多分区事务,这样它就不会在不需要的地方（即只有单分区事务时)产生高开销。 H-Store 设计了两种低开销并发控制方案,即轻量级锁定和推测并发控制[101]。轻量级锁定方案通过在没有活动的多分区事务时允许单分区事务在没有锁的情况下执行来减少获取锁和检测死锁的开销。并且推测性并发控制方案可以在等待 2PC 完成时（恰好在多分区事务的最后一个片段执行完之后)继续推测性地执行排队的事务,只要中止次数很少或很少,它就优于锁定方案涉及多轮通信的多分区事务。

此外,基于分区和并发控制策略,H-Store在事务处理上进行了一系列优化,特别是针对单事务和多事务交错的工作负载。特别是,为了处理传入事务（具有具体参数值的存储过程）,H-Store 使用马尔可夫模型

基于方法 [111] 通过预测最可能的执行路径和它可能访问的分区集来确定必要的优化。基于这些预测,它相应地应用了四大优化,即 (1)在其将访问最多的分区的节点上执行事务; (2) 只锁定事务访问的分区; (3) 禁用非中止事务的撤消日志记录; (4) 推测性地在它不再需要访问的分区上提交事务。

数据溢出。虽然 H-Store 是一个内存数据库,但它还利用一种称为反缓存 [133] 的技术,允许大于内存大小的数据存储在数据库中,而不会牺牲太多性能,通过移动冷以事务安全的方式在元组级别将数据写入磁盘,这与操作系统虚拟内存管理的页面级别形成对比。特别是,为了将冷数据驱逐到磁盘,它将最近最少使用的元组从数据库中弹出到一组将写出到磁盘的块缓冲区,更新跟踪驱逐的元组和所有索引的驱逐表,通过特殊的驱逐交易。此外,非阻塞获取是通过简单地中止访问被逐出数据的事务然后在稍后从磁盘检索数据后重新启动它来实现的,通过在中止之前执行预传递阶段来进一步优化以确定所有逐出事务所需的数据,以便可以一次检索它而无需多次中止。

容错。 H-Store 使用混合容错策略,即它利用副本集来实现高可用性 [36]、[150],并在所有副本丢失的情况下使用检查点和日志记录进行恢复 [131] .特别是,每个分区都被复制到 k 个站点,以保证 k-safety,即在 k 个站点同时发生故障的情况下它仍然提供可用性。此外,H-Store 通过分布式事务将所有提交的数据库状态定期检查点到磁盘,分布式事务将所有站点置于写入时复制模式,其中更新/删除导致行被复制到影子表。在两个检查点的间隔之间,命令记录方案[131]用于通过记录命令(即事务/存储过程标识符和参数值)来保证持久性,而不是记录每个操作(插入/删除/更新)) 由交易执行,就像传统的 ARIES 生理记录一样 [146]。此外,内存驻留的undo log 可用于支持某些可中止事务的回滚。很明显,命令日志记录的运行时开销比生理日志记录要低得多,因为它在运行时做的工作更少,写入磁盘的数据也更少,但代价是恢复时间增加。

因此,命令日志记录方案更适用于节点故障不频繁的短事务。

3.1.2 Hekaton

Hekaton [37] 是一个内存优化的 OLTP 引擎,完全集成到 Microsoft SQL 服务器中,可以同时访问Hekaton表 8 和常规 SQL 服务器表,从而为用户提供了很大的灵活性。它专为高并发OLTP而设计,利用无锁或无闩数据结构(例如,无闩哈希和范围索引)[86]、[230]、[231] 和乐观的 MVCC 技术 [107]。它还包含一个名为 Siberia [134]、[232]、[233] 的框架,以不同方式管理冷热数据,使其具备经济高效地处理大数据的能力。此外,为了减轻传统数据库中基于解释器的查询处理机制带来的开销,Hekaton 采用一次编译多次执行的策略,将 SQL 语句和存储过程先编译成 C 代码,然后将被转换成本地机器代码 [37]。

具体来说,整个查询计划被折叠成一个函数,使用标签和 gotos 进行代码共享,从而避免在函数之间传递昂贵的参数和昂贵的函数调用,最终编译的二进制文件中的指令数量最少。此外,Hekaton 通过使用增量检查点和具有日志合并和组提交优化的事务日志来确保持久性,并且通过维护高可用性副本来实现可用性 [37]。接下来我们将详细介绍它的并发控制、索引和冷热数据管理。

多版本并发控制。 Hekaton 采用优化 MVCC 来提供无锁定和阻塞的事务隔离 [107]。基本上,事务分为两个阶段,即事务从不阻塞以避免昂贵的上下文切换的正常处理阶段,以及检查读取集和幻像的可见性9的验证阶段,然后解决未完成的提交依赖性并强制执行日志记录。具体来说,更新将创建一个新版本的记录而不是更新现有记录,并且只有有效时间(即由开始和结束时间戳表示的时间范围)与事务的逻辑读取时间重叠的记录是可见的.如果那些记录已经到达验证阶段,则允许推测性地读取/忽略/更新未提交的记录,以便推进处理,而不是在正常处理阶段阻塞。但是推测处理强制执行提交依赖性,这可能会导致级联中止并且必须在提交之前解决。它利用原子操作来更新记录的有效时间、可见性检查和冲突检测,而不是锁定。最后,如果某个记录版本对任何活动事务不再可见,则以协作和并行方式对其进行垃圾回收 (GC)。即运行事务工作负载的工作线程在遇到垃圾时可以将其清除,这也自然提供了并行GC机制。从未访问过的区域中的垃圾将由专门的 GC 进程收集。

无闩锁 Bw 树。 Hekaton 提出了一种无闩锁的 B 树索引,称为 Bw 树 [86]、[230],它使用增量更新来进行状态更改,基于原子比较和交换 (CAS) 指令和弹性虚拟页面 10管理换子系统 LLAMA [231]。 LLAMA 提供了一个

8. Hekaton表在SQL Server中声明为“内存优化”,以区别于普通表。

9. 某些验证检查不是必需的,具体取决于隔离级别。例如,read commit ted和snapshot isolation不需要验证,repeatable read只需要read set visibility check。只有可序列化隔离才需要进行这两项检查。

10、这里的虚拟页面并不是指OS使用的。页面大小没有硬性限制,页面通过将“增量页面”添加到基页来增长。

虚拟页面接口,Bw-tree使用逻辑页面ID（PID)代替指针,可以根据映射表将其转换为物理地址。这允许 Bw 树节点的物理地址在每次更新时发生变化,而不需要将地址变化传播到树的根。

特别是,增量更新是通过将更新增量页面添加到先前页面并自动更新映射表来执行的,从而避免了就地更新,这可能导致代价高昂的缓存失效,尤其是在多套接字环境中,并防止了 in-使用同时更新的数据,实现无闩锁访问。增量更新策略适用于通过简单地将增量页面添加到包含先前叶节点的页面来实现的叶节点更新,以及通过一系列非阻塞协作的结构修改操作（SMO）（例如,节点拆分和合并)和原子增量更新,任何遇到未完成的 SMO [86] 的工作线程都会参与。 Delta页面和Base页面合并在一个later pointer中,以缓解delta页面链长带来的搜索效率下降。替换页面由纪元机制 [234] 回收,以保护可能被其他线程使用的数据不被过早释放。

赫卡顿的西伯利亚。西伯利亚项目 [134]、[232]、[233] 旨在使 Hekaton 能够自动透明地在更便宜的二级存储上维护冷数据,从而允许 Hekaton 中容纳比可用内存更多的数据。 Siberia 不是像 H-Store Anti-Caching [133] 那样维护 LRU 列表,而是通过先记录元组访问来对冷热数据进行离线分类,然后对其进行离线分析以预测具有最高估计访问频率的前 K 个热元组,使用基于指数平滑的高效并行分类算法[232]。

就内存和 CPU 使用而言,记录访问日志记录方法比 LRU 列表产生更少的开销。此外,为了减轻逐出元组造成的内存开销,除了多个可变大小的布隆过滤器 [235] 之外,Siberia 不在内存中存储任何关于逐出元组的附加信息（例如,索引中的键、逐出表）。]和自适应范围过滤器[233],用于过滤对磁盘的访问。此外,为了使其即使在事务访问冷热数据时也具有事务性,它在事务上协调冷热存储以保证一致性,通过使用持久更新备忘录临时记录指定当前状态的通知寒冷的记录[134]。

3.1.3 HyPer/ScyPer

HyPer [35], [236], [237] 或其分布式版本 ScyPer [149] 被设计为混合 OLTP 和 OLAP 高性能内存数据库,最大限度地利用现代硬件功能。 OLTP 事务以无锁方式顺序执行,这种方式首先在 [222] 中提倡,并行性是通过对数据库进行逻辑分区并并行接受多个分区约束事务来实现的。它可以产生前所未有的高交易率,高达每秒 100,000 个 [35]。卓越的性能归因于内存数据库中数据访问的低延迟、节省空间的有效性 Adaptive Radix Tree [87] 和存储事务处理程序的使用。 OLAP查询是在基于硬件支持的影子页的虚拟内存快照机制实现的一致性快照上进行的,是一种维护开销低的高效并发控制模型。此外,HyPer 采用动态查询编译方案,即 SQL 查询首先被编译成汇编代码[112],然后可以使用 LLVM 提供的优化即时（JIT)编译器直接执行[167].此查询评估遵循以数据为中心的范例,通过对数据对象应用尽可能多的操作,从而尽可能长时间地将数据保存在寄存器中以实现寄存器局部性。

HyPer 的分布式版本,即 ScyPer [149],采用主从架构,其中主节点负责所有 OLTP 请求,同时作为 OLAP 查询的入口点,而从节点仅用于执行 OLAP 查询。

为了将主节点的更新同步到辅助节点,使用实用通用多播协议 (PGM) 将逻辑重做日志多播到所有辅助节点,其中重做日志被重放以赶上主节点。此外,辅助节点可以订阅特定分区,从而允许为特定分区提供辅助节点并启用更灵活的多租户模型。在当前版本的 ScyPer 中,只有一个主节点,它将所有数据保存在内存中,从而将数据库大小或事务处理能力限制在一台服务器上。接下来,我们将详细介绍HyPer的快照机制、register-conscious编译方案和ART索引。

HyPer 中的快照。 HyPer 通过使用自己复制的虚拟内存空间 [35]、[147] 分叉子进程（通过 fork() 系统调用)来构建一致的快照,这不涉及软件并发控制机制,而是硬件辅助虚拟内存管理,维护开销很小。通过 fork 一个子进程,父进程中的所有数据实际上都被“复制”到子进程中。然而,它非常轻量级,因为写时复制机制只会在某些进程试图修改页面时触发真正的复制,这是由操作系统和内存管理单元 (MMU) 实现的。正如 [236] 中所报告的,页面复制是高效的,因为它可以在 2 毫秒内完成。因此,可以为 OLAP 查询有效地构建一致的快照,而无需大量的同步成本。

在 [147] 中,对四种快照机制进行了基准测试:基于软件的 Tuple Shadowing 在修改元组时生成新版本,基于软件的 Twin Tuple 始终保留每个元组的两个版本,HyPer 使用的基于硬件的 Page Shadowing,和 HotCold Shad,它通过集群更新密集型对象结合了 Tuple Shadowing 和硬件支持的 Page Shadowing。研究表明,Page Shadowing 在 OLTP 性能、OLAP 查询响应时间和内存消耗方面更胜一筹。 Page Shadowing 机制中创建快照最耗时的任务是复制进程的页表,可以通过使用大页面（x86 上每页 2 MB)来减少冷数据 [135]。监控热数据或冷数据并使用

通过读取/重置页面的年轻和脏标志的硬件辅助方法。对冷数据应用压缩以进一步提高 OLAP 工作负载的性能并减少内存消耗 [135]。

快照不仅用于 OLAP 查询,还用于长时间运行的事务 [238],因为这些长时间运行的事务会阻塞串行执行模型中的其他短时良好事务。在 HyPer 中,这些恶意事务被识别并在具有一致快照的子进程上暂时执行,并且这些事务所做的更改通过向主数据库进程发出确定性的“应用事务”来实现。应用事务验证临时事务的执行,方法是检查快照上执行的所有读取是否与主数据库上读取的内容相同(如果需要视图可序列化),或者检查快照上的写入是否与如果需要快照隔离,则在创建快照后由主数据库上的所有事务写入。如果验证成功,它会将写入应用到主数据库状态。否则,将向客户端报告中止。

注册意识编译。为了处理查询,HyPer 使用 LLVM 编译器框架 [112]、[167] 将其转换为紧凑高效的机器代码,而不是使用经典的基于迭代器的查询处理模型。

HyPer JIT 编译模型旨在通过将递归函数调用扩展到代码片段循环中来避免函数调用,从而产生更好的代码局部性和数据局部性(即 CPU 寄存器的时间局部性),因为每个代码片段执行所有操作一个执行流水线中的一个元组,在此期间元组保存在寄存器中,然后将结果具体化到下一个流水线的内存中。

由于经过优化的高级语言编译器(例如 C++)速度很慢,因此 HyPer 使用 LLVM 编译器框架为 SQL 查询生成可移植的汇编代码。特别地,在处理SQL查询时,首先按常规处理,即将查询解析、翻译和优化成一个代数逻辑计划。然而,代数逻辑计划并没有像传统方案那样被翻译成可执行的物理计划,而是编译成命令式程序(即LLVM汇编代码),然后可以直接使用LLVM提供的JIT编译器执行。尽管如此,查询处理的复杂部分(例如,复杂的数据结构管理、排序)仍然是用预编译的C++编写的。由于 LLVM 代码可以直接调用本机 C++ 方法而无需额外的包装器,因此 C++ 和 LLVM 可以在没有性能损失的情况下相互交互 [112]。然而,在代码整洁度、可执行文件的大小、效率等方面,在一个紧凑的代码片段中定义函数和内联代码之间存在权衡。

艺术索引。 HyPer 使用自适应基数树 [87] 进行高效索引。基数树的性质保证了键是按位字典顺序排列的,使得范围扫描、前缀查找等成为可能。基数树的跨度越大,树高线性降低,从而加快搜索过程,但是以指数方式增加空间消耗。 ART 通过自适应地使用具有相同、相对较大跨度但不同扇出的不同内部节点大小来实现空间和时间效率。

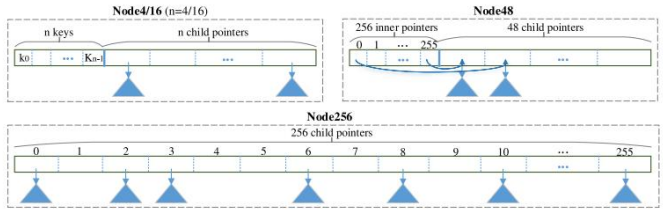

图 5. ART 内部节点结构。

具体来说,有四种跨度为8位但容量不同的内部节点:Node4、Node16、Node48和Node256,根据其最大存储子节点指针的容量来命名。特别是,Node4/Node16 最多可以存储 4/16 个子指针,并使用一个长度为 4/16 的数组用于排序的键,另一个长度相同的数组用于子指针。 Node48 使用一个 256 元素的数组将关键位直接索引到容量为 48 的指针数组,而 Node256 只是一个包含 256 个指针的数组作为普通基数树节点,用于存储 49 到 256 个条目。图 5 说明了 Node4、Node16、Node48 和 Node256 的结构。采用延迟扩展和路径压缩技术进一步降低内存消耗。

3.1.4 SAP HANA

SAP HANA [77]、[239]、[240] 是一个分布式内存数据库,用于集成 OLTP 和 OLAP [41],以及结构化(即关系表)[74]、半结构化的统一(即图形)[241]和非结构化数据(即文本)处理。只要有足够的可用空间,所有数据都会保存在内存中,否则整个数据对象(例如,表或分区)将从内存中卸载并在再次需要时重新加载到内存中。 HANA具有以下特点:

它支持关系数据的面向行和面向列的存储,以优化不同的查询工作负载。此外,它通过添加两级增量数据结构来为高效的 OLAP 和 OLTP 利用列式数据布局,以减轻列式数据结构中插入和删除操作的低效率 [74]。

它通过提供多种查询语言接口(例如,标准 SQL、SQLScript、MDX、WIPE、FOX 和 R)来提供丰富的数据分析功能,这使得将更多应用语义下推到数据管理层变得容易,从而避免繁重的数据传输成本。

它自然地支持基于时间线索引 [242] 的时间查询,因为数据在 HANA 中是版本化的。
它提供基于多版本并发控制的快照隔离、基于优化的两阶段提交协议 (2PC) [243] 的事务语义,以及通过日志记录和定期检查点到 GPFS 文件系统 [148] 的容错。

我们将仅详细说明前三个特征,因为另一个特征是文献中使用的相当普遍的技术。

关系商店。 SAP HANA 支持关系表的面向行和面向列的物理表示。

行存储有利于大量更新和插入,因为

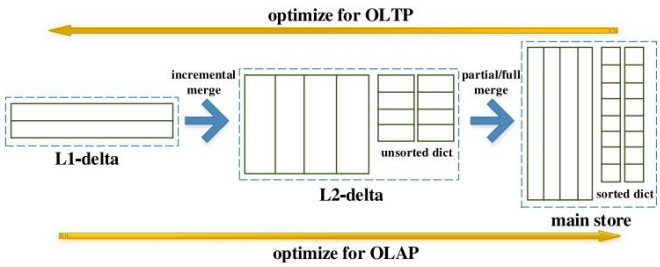

图 6. HANA 混合商店。

以及在 OLTP 中常见的点查询,而列存储是 OLAP 应用程序的理想选择,因为它们通常一起访问列的所有值,并且一次访问几个列。面向列的表示的另一个好处是它可以更有效和高效地利用压缩技术。在 HANA 中,可以将表/分区配置为行存储或列存储,也可以将其从一个存储重组为另一个存储。

HANA 还提供了一个存储顾问 [244],通过同时考虑查询成本和压缩率来推荐基于数据和查询特征的最佳表示。

由于一个表/分区只存在于行存和列存中,两者各有弱点,HANA设计了一个三级的面向列的统一表结构,由L1-delta、L2-delta和main store组成,如图 6 所示,为 OLTP 和 OLAP 工作负载提供有效支持,这表明列存储也可以为 OLTP 高效部署 [41],[74]。

通常,一个元组首先以行格式存储在 L1-delta 中,然后以列格式传播到 L2-delta,最后以更重的压缩与主存储合并。这三个阶段的整个过程在HANA术语中称为元组的生命周期。

丰富的数据分析支持。 HANA 支持用于数据分析（即 OLAP)的各种编程接口,包括用于通用数据管理功能的标准 SQL,以及更专业的语言,例如 SQL 脚本、MDX、FOX、WIPE [74]、[240] 和 R [245] .虽然 SQL 查询的执行方式与传统数据库中的相同,但必须转换其他专门的查询。

这些查询首先被解析为称为“计算图模型”的中间抽象数据流模型,其中源节点表示持久化或中间表,内部节点反映这些查询执行的逻辑运算符,然后转换为类似于 SQL 的执行计划询问。与其他系统不同,HANA 支持将 R 脚本作为系统的一部分,以更好地优化临时数据分析作业。具体来说,R 脚本可以嵌入到计算图中的自定义运算符中 [245]。当要执行 R 运算符时,将使用 Rserve 包 [246] 调用单独的 R 运行时。

由于 HANA 面向列的表的列格式类似于 R 的面向向量的数据框,因此从表到数据框的转换开销很小。数据传输是通过共享内存实现的,这是一种高效的进程间通信（IPC)机制。在 RICE 包[245]的帮助下,它只需要复制一次就可以使数据可用于 R 进程,即它只是将数据从数据库复制到共享内存部分,R 运行时可以直接从共享内存部分访问数据。

时间查询。 HANA 基于称为时间轴索引 [88]、[242]、[247] 的统一索引结构支持时间查询,例如时间聚合、时间旅行和时间连接。对于每个逻辑表,HANA 将表的当前版本保存在当前表中,将以前版本的整个历史记录保存在时态表中,并附有时间线索引以方便时态查询。时态表的每个元组都带有一个有效间隔,从提交时间到最后一个有效时间,在该时间某些事务使该值无效。 HANA 中的事务时间由离散的、单调递增的版本表示。基本上,时间线索引将每个版本映射到在该版本之前或在该版本时提交的所有写入事件（即临时表中的记录）。时间线索引由事件列表和版本映射组成,其中事件列表跟踪每个失效或验证事件,版本映射跟踪每个版本的数据库可以看到的事件序列。

因此,由于时态表在每个时间点的所有可见行都被跟踪,因此可以通过同时扫描事件列表和版本映射来实现时态查询。

为了降低构建时态视图的完整扫描成本,HANA 在历史记录的特定时间用许多完整的视图表示（称为检查点)增强了基于差异的时间线索引。具体来说,checkpoint是一个长度等于Temporal Table行数的位向量,表示在某个时间点（即某个版本)Temporal Table可见的行。借助检查点,可以通过从该时间之前的最新检查点开始扫描来获得特定时间的时间视图,而不是每次都从事件列表的开头扫描。

3.2 In-Memory NoSQL Databases NoSQL

是Not Only SQL的缩写,NoSQL数据库提供了一种不同于关系数据库的数据存储和检索机制。 NoSQL 数据库中的数据通常结构为树、图或键值,而不是表格关系,查询语言通常也不是 SQL。 NoSQL 数据库的动机是其简单性、水平扩展和对可用性的更好控制,并且它通常会牺牲一致性以支持可用性和分区容错性 [25]、[248]。

随着“内存就是新磁盘”的趋势,内存NoSQL数据库近年来蓬勃发展。有键值存储,例如 Redis [66]、RAMCloud [2]、Mem epiC [60]、[138]、Masstree [249]、MICA [64]、Mercury [250]、Citrusleaf/Aerospike [34]、 Kyoto/Tokyo Cabinet [251]、Pilaf [252]、MongoDB [65]、Couchbase [253] 等文档存储、Trinity [46]、Bitsy [254] 等图形数据库、OWLIM [255] 等 RDF 数据库、 WhiteDB [50] 等。有一些系统部分在内存中,例如 MongoDB [65]、MonetDB [256]、MDB [257],因为它们使用内存映射文件来存储数据,以便数据可以就像在内存中一样被访问。

在本节中,我们将介绍一些具有代表性的内存 NoSQL 数据库,包括 MemepiC [60]、[138]、

MongoDB [65]、RAMCloud [2]、[75]、[126]、[258]、[259]、Redis [66] 和一些图形数据库。

3.2.1 MemepiC

MemepiC [60] 是 epiC [23] 的内存版本,是一个基于 Actor 并发编程模型 [260] 的可扩展和可伸缩系统,专为处理大数据而设计。它不仅作为分布式键值存储提供低延迟存储服务,还集成了内存数据分析功能以支持在线分析。凭借高效的数据回收和获取机制,MemepiC 旨在维护比可用内存大得多的数据,而不会严重降低性能。我们将从三个方面来阐述 MemepiC:减少系统调用、集成存储服务和分析操作、以及虚拟内存管理。

更少的系统调用设计。依赖系统调用与硬件通信或同步的传统数据库设计不再适合实现内存系统所需的良好性能,因为系统调用产生的开销不利于整体性能。因此,MemepiC 遵循较少系统调用的设计原则,并尝试在存储访问(通过内存映射文件代替)、网络通信(通过 RDMA 或基于库的方式)中尽可能减少系统调用的使用。网络)、同步(通过事务内存或原子原语)和容错(通过远程日志记录)[60]。

存储服务和分析操作的集成。为了满足在线数据分析的要求,MemepiC 还集成了数据分析功能,允许在存储数据的地方分析数据[60]。通过数据存储和分析的集成,它显着消除了数据移动成本,这通常在传统数据分析场景中占主导地位,其中数据首先从数据库层获取到应用程序层,然后才能对其进行分析[172]。数据分析和存储服务之间的同步是基于原子原语和基于分叉的虚拟快照实现的。

用户空间虚拟内存管理 (UVMM)。 MemepiC 通过有效的用户空间虚拟内存管理机制缓解了主内存相对较小的问题,当总数据大小超过内存大小时,基于可配置的分页,允许数据自由地逐出到磁盘策略 [138]。通过利用混合存储,数据存储的适应性可以实现从基于磁盘的数据库到基于内存的数据库的平稳过渡。它不仅利用了语义感知驱逐策略,还利用了硬件辅助 I/O 和 CPU 效率,作为一种更通用的“反缓存”[138] 方法展现出巨大潜力。具体而言,它采用了以下策略。

混合访问跟踪策略,包括用户支持的元组级访问日志记录、MMU 辅助页面级访问跟踪、基于虚拟内存区域 (VMA) 保护的方法和 malloc 注入,实现轻量级和语义感知访问跟踪。

基于上述访问跟踪方法收集到的细粒度访问痕迹定制的 WSCLOCK分页策略,以及其他替代策略包括LRU、基于老化的 LRU和FIFO,使在线驱逐策略更加准确和灵活。

基于VMA保护的记账方式,记账数据位置内存开销小,一次跟踪数据访问。

具有快速压缩技术(即 LZ4 [261])和内核支持的异步 I/O 的更大数据交换单元,可以利用内核 I/O 调度程序和块 I/O 设备,减少 I/O交通显着。

3.2.2 MongoDB

MongoDB [65] 是一个面向文档的NoSQL 数据库,对文档模式(即 BSON 风格)的限制很少。具体来说,一个 MongoDB 托管许多数据库,每个数据库都包含一组文档集合。 MongoDB 在文档级别提供原子性,索引和数据分析只能在单个集合中进行。因此不支持“跨集合”查询(例如在传统数据库中加入)。它使用主/从复制机制来保证高可用性,并使用分片来实现可扩展性。从某种意义上说,MongoDB 还可以充当文档(例如 HTML 文件)的缓存,因为它通过为文档设置 TTL(Time-to-Live)来提供数据过期机制。

我们将在以下部分详细讨论 MongoDB 的两个方面,即存储和数据分析功能。

内存映射文件。 MongoDB 利用内存映射文件来管理所有数据并与之交互。如果总数据可以放入内存,它可以充当完全内存中的数据库。否则,它取决于虚拟内存管理器 (VMM),它将决定何时以及在哪个页面进行页面调入或页面调出。内存映射文件提供了一种访问磁盘上文件的方式,就像我们访问动态内存一样 通过指针。因此,我们可以通过提供其指针(即虚拟地址)直接获取磁盘上的数据,这是通过优化的 VMM 实现的,以使分页过程尽可能快。访问内存映射文件通常比直接文件操作更快,因为它不需要系统调用来进行正常访问操作,并且在大多数操作系统中不需要将内存从内核空间复制到用户空间。另一方面,VMM 无法适应 MongoDB 自身特定的内存访问模式,尤其是当多个租户驻留在同一台机器上时。通过考虑特定的使用场景,更智能的临时方案将能够更有效地管理内存。

数据分析。 MongoDB 支持两种类型的数据分析操作:聚合(即 MongoDB 术语中的聚合管道和单一目的聚合操作)和 MapReduce 函数,应使用 Java Script 语言编写。需要集中组装的分片集群上的数据分析分两步进行:

查询路由器将作业划分为一组任务,并将任务分配给适当的分片实例,分片实例在完成专用计算后将部分结果返回给查询路由器。

然后查询路由器将组装部分结果并将最终结果返回给客户端。

3.2.3 内存云

RAMCloud [2]、[75]、[126]、[258]、[259] 是一种分布式内存键值存储,具有低延迟、高可用性和高内存利用率的特点。特别是,它可以通过利用低延迟网络(例如 Infiniband 和 Myrinet)的优势实现数十微秒的延迟,并通过利用大规模从系统故障中恢复 1-2 秒来提供“持续可用性”。此外,它采用日志结构的数据组织,采用两级清洗策略,将数据结构化在内存和磁盘上。这导致了高内存利用率和单一的统一数据管理策略。 RAMCloud 的架构由一个协调器组成,协调器负责维护集群中的元数据,例如集群成员关系、数据分布和一些存储服务器,每个存储服务器包含两个组件,一个管理内存数据和处理内存的主模块来自客户端的读/写请求,以及使用本地磁盘或闪存备份其他服务器拥有的数据副本的备份模块。

数据组织。 RAMCloud 中的键值对象被分组到一组表中,每个表都根据键的哈希码单独进行范围分区。 RAMCloud 依靠散列函数的一致性,将对象按存储服务器覆盖的散列空间量(即范围)的比例均匀分布在表中。存储服务器使用单个日志来存储数据,并使用哈希表进行索引。通过哈希表访问数据,该哈希表指示对对象当前版本的访问。

RAMCloud 采用日志结构的内存管理方法,而不是传统的内存分配机制(例如C 库的malloc),通过消除内存碎片使内存利用率达到 80-90%。

特别地,日志被分成一组片段。由于日志结构是仅附加的,因此不允许就地删除或更新对象。因此,应安排定期清理作业来清理已删除/过时的对象以回收可用空间。 RAMCloud 设计了一个高效的两级清理策略。

当可用内存低于 10% 时,它会安排段压缩作业首先清理内存中的日志段,方法是将其实时数据复制到较小的段并释放原始段。

当磁盘上的数据比内存中的数据大一个阈值时,将启动联合清理作业,将内存中的日志和磁盘上的日志一起清理。

二级清理策略可以通过更频繁地清理内存中的日志来获得较高的内存利用率,同时通过尝试降低磁盘利用率(即增加百分比)来降低磁盘带宽需求已删除/陈旧数据),因为这可以避免在清理期间在磁盘上复制大量实时数据。

快速崩溃恢复。内存存储的一大挑战是容错,因为数据驻留在易失性 DRAM 中。 RAMCloud 使用复制通过在远程磁盘中复制数据来保证持久性,并利用大规模资源(例如 CPU、磁盘带宽)来加快恢复过程[126]、[259]。具体来说,当主服务器收到来自客户端的更新请求时,主服务器将新对象附加到内存日志中,然后将对象转发给R(通常R = 3)个远程备份服务器,这些服务器首先将对象缓存在内存中,将缓冲区批量刷新到磁盘上(即以段为单位)。一旦对象被复制到缓冲区中,备份服务器就会立即响应,因此响应时间主要由网络延迟而不是磁盘 I/O 决定。

为了使恢复更快,数据的副本以段为单位分散在集群中的所有备份服务器中,从而使更多的备份服务器协作进行恢复过程。每个主服务器通过随机化和细化相结合的方式独立决定在何处放置段副本,这不仅消除了病态行为,而且实现了近乎最优的解决方案。此外,当一台服务器发生故障后,除了所有相关的备份服务器外,还涉及多个主服务器共同分担恢复工作(即重新构建内存日志和哈希表),并负责确保均匀分区恢复的数据。 recovery job的分配由master在crash前立下的遗嘱决定。遗嘱是根据平板电脑配置文件计算的,每个平板电脑配置文件都维护一个直方图,以跟踪单个表格/平板电脑中资源使用的分布(例如,记录数量和空间消耗)。该遗嘱旨在平衡恢复作业的分区,以便它们需要大致相等的时间来恢复。

随机复制策略产生几乎均匀的副本分配并利用大规模,从而防止数据丢失并最大限度地减少恢复时间。

然而,这种策略可能会导致在同时发生节点故障的情况下数据丢失[125]。虽然由于段副本的高度分散性,丢失的数据量可能很小,但有可能某部分数据的所有副本都变得不可用。因此,RAMCloud 还支持另一种基于 Copyset [125]、[258]、[259] 的复制模式,以降低在大型协调故障(如断电)后数据丢失的可能性。 Copyset 通过限制可以将主服务器中的所有段复制到的备份服务器集来权衡丢失数据量以降低数据丢失频率。但是,这可能会导致更长的恢复时间,因为用于从磁盘读取副本的备份服务器更少。权衡可以通过分散宽度来控制,分散宽度是每个服务器的数据被允许复制到的备份服务器的数量。例如,如果分散宽度等于集群中所有其他服务器(除了要复制的服务器)的数量,Copyset 则转向随机复制。

3.2.4 Redis

Redis [66] 是一个用 C 实现的内存中键值存储,支持一组复杂的数据结构,包括散列、列表、集合、排序集合和一些高级的

发布/订阅消息、脚本和交易等功能。它还嵌入了两种持久性机制 快照和仅附加日志记录。快照会定期将内存中的所有当前数据备份到磁盘上，这有助于恢复过程,而仅附加日志记录会记录每个更新操作,从而保证更高的可用性。 Redis 是单线程的,但它通过使用事件通知策略来重叠网络 I/O 通信和数据存储/检索计算来异步处理请求。

Redis 还维护了一个哈希表来构造所有键值对象,但它使用简单的内存分配（例如,malloc/free）,而不是基于 slab 的内存分配策略（即,Memcached 的）,因此它不是很适合作为 LRU 缓存,因为它可能会产生严重的内存碎片。在以后的版本中采用 jemalloc [262] 内存分配器可以部分缓解这个问题。

脚本。 Redis 具有服务器端脚本功能（即 Lua 脚本）,它允许应用程序在服务器内部执行用户定义的功能,从而避免一系列相关操作的多次往返。然而,在脚本引擎和主要存储组件之间的通信中不可避免地存在昂贵的开销。此外,长时间运行的脚本会降低服务器的整体性能,因为 Redis 是单线程的,并且长时间运行的脚本会阻塞所有其他请求。

分布式Redis。分布式 Redis 的第一个版本是通过客户端的数据分片实现的。最近,Redis 组推出了一个名为 Redis Cluster 的分布式 Redis 的新版本,它是一个自治的分布式数据存储,支持自动数据分片、主从容错和在线集群重组（例如,添加/删除一个节点,重新分片数据）。

Redis Cluster 是完全分布式的,没有集中的 master 来监控集群和维护元数据。基本上,Redis 集群由一组 Redis 服务器组成,每个服务器都知道其他服务器。也就是说,每个 Redis 服务器保留所有元数据信息（例如,分区配置、其他节点的活动状态)并使用八卦协议传播更新。

Redis Cluster 使用哈希槽分区策略将总哈希槽的一个子集分配给每个服务器节点。
因此,每个节点负责其哈希码在其分配的插槽子集中的键值对象。客户端可以自由地向任何服务器节点发送请求,但是当特定节点无法在本地响应请求时,它将获得包含适当服务器地址的重定向响应。在这种情况下,单个请求需要两次往返。如果客户端可以缓存哈希槽和服务器节点之间的映射,则可以避免这种情况。

当前版本的 Redis Cluster 需要手动重新分片数据并将插槽分配给新添加的节点。可用性是通过一个Redis主服务器和多个复制主服务器所有数据的从服务器来保证的,它使用异步复制以获得良好的性能,但是这可能会引入主副本和副本之间的不一致。

3.2.5 内存图形数据库Bitsy。 Bitsy [254] 是
一个可嵌入的内存中图形数据库,它实现了 Blueprints API,ACID 保证基于乐观并发模型的事务。 Bitsy 在内存中维护整个图形的副本,但在提交操作期间将每个更改记录到磁盘,从而实现从故障中恢复。
Bitsy 设计用于在多线程 OLTP 环境中工作。

具体来说,它使用多级双缓冲区/日志来提高写入事务性能并便于日志清理,并使用顺序锁的无锁读取来改善读取性能。基本上,它具有三个主要设计原则：

不求。 Bitsy 将所有更改附加到无序事务日志中,并依靠重组过程来清理过时的顶点和边。

没有插座。 Bitsy 充当嵌入式数据库,将集成到应用程序（基于 Java)中。
因此应用程序可以直接访问数据,而不需要通过基于套接字的接口进行传输,这会导致大量的系统调用和序列化/反序列化开销。

没有 SQL。 Bitsy 实现了面向属性图模型的 Blueprints API，而不是使用 SQL 的关系模型。

三位一体。 Trinity 是用于图形分析的内存分布式图形数据库和计算平台 [46]、[263],其图形模型是基于内存键值存储构建的。具体来说，每个图节点对应一个Trinity cell,它是一个(id, blob)对,其中id表示图中的一个节点,blob是序列化二进制格式的节点的相邻列表,而不是运行时对象格式,以最小化内存开销并促进检查点过程。但是,这会在分析计算中引入序列化/反序列化开销。大单元,即具有大量邻居的图形节点，表示为一组小单元,每个小单元仅包含局部边缘信息,以及包含分散单元的单元 ID 的中心单元。此外,边缘可以用标签（例如,谓词)标记,这样它就可以扩展到 RDF 存储[263],同时支持本地谓词索引和全局谓词索引。在局部谓词索引中,每个节点的邻接表首先按谓词排序,然后按相邻节点id排序,例如传统RDF存储中的SPO11或OPS索引,而全局谓词索引可以像传统那样定位具有特定谓词的单元格PSO 或 POS 索引。

3.3 内存缓存系统缓存在增强系统性能方
面起着重要作用,尤其是在Web 应用程序中。 Facebook、Twitter、维基百科、LiveJournal 等。都在广泛地利用缓存来提供良好的服务。缓存可以为应用程序提供两种优化：通过允许从内存访问数据来优化磁盘 I/O,以及通过保留结果而不需要重新计算来优化 CPU 工作负载。许多缓存系统已被开发用于

11. S代表主语,P代表谓语,O代表宾语。

各种目标。有通用的缓存系统,例如 Memcached [61] 和 BigTable Cache [248],旨在加速分析作业的系统,例如 PACMan [264] 和 Grid Gain [51],以及更多为支持特定框架而设计的特定用途系统例如 .NET 的 NCache [265] 和 Windows 服务器的 Velocity/AppFabric [266],支持严格事务语义的系统,例如 TxCache [267],以及网络缓存,例如 HashC ache [268]。

尽管如此,缓存系统主要是为 Web 应用程序设计的,因为 Web 2.0 增加了计算的复杂性和服务级别协议 (SLA) 的严格性。全页缓存 [269]、[270]、[271] 在早期被采用,而使用细粒度对象级数据缓存 [61]、[272] 以提高灵活性变得很有吸引力。在本节中,我们将介绍一些具有代表性的缓存系统/库及其在内存数据管理方面的主要技术。

3.3.1 内存缓存

Memcached [61] 是一种轻量级内存中键值对象缓存系统,具有严格的 LRU 驱逐。它的分布式版本是通过客户端库实现的。因此,客户端库管理数据分区(通常是基于散列的分区)和请求路由。 memc ached针对C/C++、PHP、Java、Python等多种语言都有不同版本的客户端库,另外还提供了文本协议和二进制协议两种主要协议,同时支持UDP和TCP连接。

数据组织。 Memcached 使用一个大的哈希表来索引所有键值对象,其中键是一个文本字符串,值是一个不透明的字节块。特别是,内存空间被分成 1 MB 的 slab,每个 slab 都分配给一个 slab 类。并且 slab 不会被重新分配给另一个类。平板被进一步切割成特定尺寸的块。每个 slab 类都有自己的块大小规范和驱逐机制(即 LRU)。 Key-value 对象根据其大小存储在相应的 slab 中。基于slab的内存分配如图7所示,其中增长因子表示相邻slab类之间的块大小差异比率。 slab设计有助于防止内存碎片和优化内存使用,但也会导致slab钙化问题。

例如,在 Memcached 用完 512 KB slab 但有很多 2 MB slab 时尝试插入 500 KB 对象的场景中,它可能会导致不必要的逐出。在这种情况下,一个 512 KB 的对象将被逐出,尽管仍有大量可用空间。这个问题在 Facebook 和 Twitter [273]、[274] 的优化版本中得到缓解。

并发。 Memcached 使用 libevent 库来实现异步请求处理。另外,Memcached是一个多线程程序,具有细粒度的pthread互斥锁机制。静态项锁哈希表用于控制内存访问的并发性。锁哈希表的大小根据配置的线程数确定。并且在锁哈希表的内存使用和平行等位性程度之间存在权衡。尽管 Memcached 提供了如此细粒度的锁定机制,但大多数操作(例如

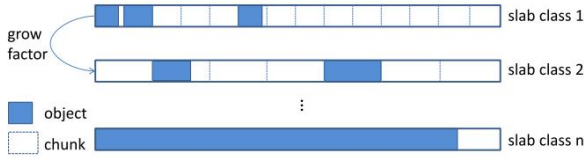

图 7. 基于 Slab 的分配。

索引查找/更新和缓存逐出/更新仍然需要全局锁 [63],这会阻止当前的 Memcached 在多核 CPU 上扩展 [275]。

生产中的 Memcached Facebook 的 Memcache [273] 和 Twitter 的 Twemcache [274]。 Facebook 从工程的角度在三个不同的部署级别(即集群、区域和跨区域)扩展 Memcached,通过关注其特定的工作负载(即读取密集型)并在性能、可用性和一致性之间进行权衡 [273]]. Memcache 通过设计细粒度锁定机制、自适应 slab 分配器以及惰性和主动驱逐方案的混合来提高 Memcached 的性能。此外,Memcache 更侧重于部署级别的优化。为了减少请求的延迟,它采用并行请求/批处理,使用无连接 UDP 获取请求,并结合流量控制机制来限制 incast 拥塞。它还利用租约 [276] 和陈旧读取 [277] 等技术来实现高命中率,并使用配置池来平衡负载和处理故障。

Twitter 在其名为 Twemcache 的分布式 Memcached 版本上采用了类似的优化。当没有足够的空间时,它通过随机驱逐整个 slab 并重新分配所需的 slab 类来缓解 Memcached 的 slab 分配问题(即 slab 钙化问题)。

它还通过 updater 聚合器模型启用无锁统计信息收集,Facebook 的 Memcache 也采用了该模型。另外,Twitter 还提供了 Memcached 协议的代理,可以用来减少大量部署 Memcached 时的 TCP 连接数

服务器。

3.3.2 MemC3

MemC3 [63] 通过使用乐观的并发布谷鸟哈希算法和基于 CLOCK [279] 的 LRU 近似驱逐算法在性能和内存效率方面优化 Memcached,假设小和只读请求占主导地位在现实世界的工作负载中。 MemC3 主要促进读取密集型工作负载,因为写入操作仍然在 MemC3 中序列化,并且布谷鸟哈希有利于读取而不是写入操作。此外,涉及大量小对象的应用程序应该在内存效率方面从 MemC3 中获益更多,因为 MemC3 消除了嵌入在键值对象中的大量指针开销。基于 CLOCK 的驱逐算法比基于列表的严格 LRU 占用更少的内存,并且由于不需要全局同步来更新 LRU,因此可以实现高并发。

乐观并发布谷鸟哈希。布谷鸟哈希 [278] 的基本思想是使用两个哈希函数为哈希表中的每个键提供两个可能的桶。当插入一个新密钥时,它被插入到它的两个可能的桶之一。如果两个桶都被占用,它会随机

替换已经驻留在这两个桶之一中的密钥。然后将置换后的键插入到它的替代桶中,这可能会进一步触发置换,直到找到空槽或直到达到置换的最大数量(此时,使用新的哈希函数重建哈希表)。这个位移序列形成布谷鸟位移路径。布谷鸟哈希的冲突解决策略可以实现高负载因子。此外,它消除了 Memcached 使用的基于链接的哈希中每个键值对象中嵌入的指针字段,这进一步提高了 MemC3 的内存效率,特别是对于小对象。

MemC3 通过允许每个桶具有四个标记槽(即四向集合关联)并将有效布谷鸟位移路径的发现与路径的执行分开以实现高并发,从而优化了传统的布谷鸟哈希。槽中的标签用于过滤不匹配的请求,帮助计算置换过程中的备选桶。这是在不需要访问确切键的情况下完成的(因此没有额外的指针取消引用),这使得查找和插入操作缓存友好。通过先搜索布谷鸟位移路径,然后将需要位移的键沿布谷鸟位移路径向后移动,实现了细粒度的乐观锁机制。

MemC3 使用锁条带化技术来平衡锁的粒度,使用乐观锁来实现多读/单写并发。

3.3.3 发送缓存

TxCache [267] 是一种基于快照的事务缓存,用于管理对事务数据库查询的缓存结果。 TxCache 确保事务只看到来自缓存和数据库的一致快照,它还提供了一个简单的编程模型,其中应用程序只需将函数/查询指定为可缓存,而 TxCache 库处理结果的缓存/失效。

TxCache 使用版本控制来保证一致性。特别是,缓存和数据库中的每个对象都标记有一个版本,由其有效间隔描述,该时间戳是对象有效的时间戳范围。

事务可以有一个陈旧条件来指示事务可以容忍过去陈旧秒内的一致快照。因此,只有与事务的容忍范围重叠的记录(即,其时间戳减去陈旧性与其时间戳之间的范围)才应在事务执行中被考虑。为了提高缓存命中率,通过维护一组令人满意的时间戳并在查询缓存时修改它来延迟选择事务的时间戳。这样,从缓存中获取更多请求记录的概率就会增加。

而且,它仍然同时保持了多版本的一致性。缓存的结果在其相关记录更新时自动失效。这是通过将缓存中的每个对象与一个无效标记相关联来实现的,该标记描述了它所依赖的数据库的哪些部分。当数据库中的某些记录被修改时,数据库会识别受影响的失效标签集并将这些标签传递给缓存节点。

4内存数据处理系统

内存数据处理/分析在大数据时代变得越来越重要,因为需要在短时间内分析大量数据。通常,有两种类型的内存处理系统:专注于批处理的数据分析系统,例如 Spark [55]、Piccolo [59]、SINGA [280]、Pregel [281]、GraphLab [47] ,Mammoth [56],Phoenix [57],Grid Gain [51],以及实时数据处理系统(即流处理),如 Storm [53],Yahoo! S4 [52]、Spark Streaming [54]、MapReduce Online [282]。在本节中,我们将回顾两种类型的内存数据处理系统,但主要关注那些为支持数据分析而设计的系统。

4.1 内存大数据分析系统 4.1.1 主内存 MapReduce (M3R)

M3R [58] 是 MapReduce 框架的主要内存实现。它设计用于交互式分析万亿字节的数据,这些数据可以保存在一个小型节点集群的内存中,具有很高的平均故障时间。它提供了与传统 Map Reduce [283] 的向后兼容接口,并且性能显着提高。然而,它不能保证弹性,因为它在 map/reduce 阶段后将结果缓存在内存中,而不是刷新到本地磁盘或 HDFS,这使得 M3R 不适合长时间运行的作业。具体来说,M3R 对传统的 MapReduce 设计进行了以下两方面的优化:

它将输入/输出数据缓存在内存中的键值存储中,以便后续作业可以直接从缓存中获取数据,并且消除了输出结果的物质化。基本上,键值存储使用路径作为键,并将路径映射到包含数据块位置的元数据位置。

它通过指定一个分区器来控制键如何映射到缩减器之间的分区,从而保证分区稳定性以实现局部性,从而允许迭代作业重用缓存数据。

4.1.2 小

Piccolo [59] 是一个内存中以数据为中心的编程框架,用于跨多个节点运行数据分析计算,支持数据局部性规范和面向数据的积累。基本上,分析程序由一个在主节点上执行的控制函数和一个内核函数组成,该函数作为多个实例启动,同时在许多工作节点上执行并共享分布式可变键值表,可以随时更新-粒度键值对象级别。具体而言,Piccolo 支持以下功能:

用户定义的累积函数(例如,最大值、求和)可以与每个表相关联,并且 Piccolo 在运行时执行累积函数以组合对同一键的并发更新。

为了在分布式计算过程中实现数据局部性,允许用户定义一个分区

一个表的函数,并将内核执行与某个表分区放在一起,或者将来自不同表的分区放在一起。

Piccolo 通过全局用户辅助检查点/恢复机制处理机器故障,方法是在控制功能中明确指定检查点的时间和内容。

计算期间的负载平衡通过工作窃取进行优化,即指示已完成所有分配任务的工作人员从剩余任务最多的工作人员那里窃取尚未开始的任务。

4.1.3 Spark/RDD

Spark 系统 [55]、[284] 提出了一种用于大数据分析的数据抽象,称为弹性分布式数据集（RDD）,它是一种粗粒度的确定性不可变数据结构,具有基于沿袭的容错性 [285]、[286]。在 Spark 之上,Spark SQL、Spark Streaming、MLlib 和 GraphX 分别为基于 SQL 的操作、流处理、机器学习和图形处理而构建。它有两个主要特点：

它使用弹性持久模型来提供将数据集持久保存在内存、磁盘或两者中的灵活性。通过将数据集保存在内存中,它有利于需要多次读取数据集的应用程序（例如,迭代算法）,并支持交互式查询。

它结合了轻量级容错机制（即沿袭）,无需检查点。 RDD的沿袭包含足够的信息,因此可以根据其沿袭和依赖的RDD重新计算它,这些RDD在最坏的情况下是HDFS中的输入数据文件。 Tachyon [287] 也采用了这个想法,它是一个分布式文件系统,可以通过内存实现可靠的文件共享。

以容错方式将数据持久保存在内存中的能力使 RDD 适用于许多数据分析应用程序,尤其是那些迭代作业,因为它消除了像 Hadoop 那样在每个阶段将数据洗牌到磁盘的沉重成本。我们详细阐述RDD的以下两个方面:数据模型和作业调度。

数据模型。 RDD 为只读分布式数据集提供了抽象。数据修改是通过粗粒度的 RDD 转换实现的,该转换将相同的操作应用于 RDD 中的所有数据项,从而生成一个新的 RDD。这种抽象为高一致性和轻量级容错方案提供了机会。具体来说,RDD 记录它依赖的转换（即它的沿袭）,没有数据复制或容错检查点。当 RDD 的一个分区丢失时,它会根据其沿袭从其他 RDD 重新计算。由于 RDD 通过粗粒度转换进行更新,与传统的数据复制或检查点方案相比,它通常需要更少的空间和精力来备份线龄信息,但代价是计算密集型的重新计算成本更高工作,当出现故障时。因此,对于涉及大量重新计算成本的长沿袭图的 RDD,使用检查点,这更有利。

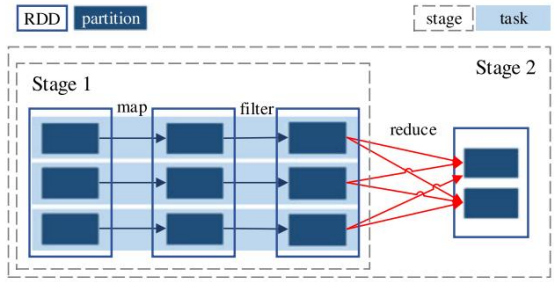

图 8. Spark 作业调度程序。

RDD 模型在计算过程中为“工作集”提供了很好的缓存策略,但它不够通用,无法支持传统的数据存储功能,原因有二：

RDD 容错方案基于粗粒度数据操作假设,无需就地修改,因为它必须保证程序大小远小于数据大小。

因此,该模型不支持细粒度的数据操作,例如更新单个键值对象。

它假设在一个稳定的存储上存在一个持久化的原始数据集,这保证了容错模型的正确性和基于块的组织模型的适用性。

然而,在传统的数据存储中,数据是动态到达的,无法预先确定数据的分配。结果,数据对象分散在内存中,导致内存吞吐量下降。

作业调度。 Spark 中的作业被组织成一个 DAG,它捕获作业依赖项。 RDD 使用惰性物化,即 RDD 不被计算,除非它被用于一个动作（例如,count()）。当在 RDD 上执行操作时,调度程序会检查 RDD 的沿袭以构建要执行的作业的 DAG。 Spark 使用两阶段作业调度,如图 8 [55] 所示：

它首先将作业组织成阶段的 DAG,每个阶段可能包含一系列作业,这些作业仅对分区级别具有一对一的依赖性。例如,在图 8 中,Stage 1 由两个作业组成,即 map 和 filter,它们都只有一对一的依赖关系。阶段的边界是带有洗牌的操作（例如,图 8 中的减少操作）,它们具有多对多的依赖关系。

在每个阶段,任务由分区上的一系列作业组成,例如图 8 中阴影分区上的映射和过滤作业。任务是系统中的调度单元,它消除了物化中间状态（例如,图 8 中阶段 1 的中间 RDD）,并启用细粒度调度策略。

4.2 内存实时处理系统 4.2.1 Spark Streaming Spark Streaming [54] 是一种基于Spark [55] 构建的容错流处理系统。它构建了一个流

计算作为一系列无状态的、确定性的、小时间间隔（比如 1 秒)的批量计算,而不是保持连续的、有状态的运算符。因此,它针对的是可以容忍几秒延迟的应用程序。 Spark Streaming充分利用了RDD的不变性和Spark基于线龄的容错机制,并进行了一些扩展和优化。具体来说,传入的流根据时间间隔分为一系列不可变的 RDD,称为 D-streams,它们是确定性转换可以作用的基本单元,不仅包括许多在正常情况下可用的转换Spark RDD（例如,map、reduce 和 groupBy）,以及专用于 Spark Streaming 的窗口计算（例如,reduceByWindow 和 countBy Window）。当新流到达时,来自历史间隔的 RDD 可以自动与新生成的 RDD 合并。流数据在两个工作节点之间进行复制,以保证基于沿袭的恢复所依赖的原始数据的持久性,并且定期进行检查点操作以减少由于长沿袭图而导致的恢复时间。 D-streams 的确定性和分区级沿袭使得在节点失败后执行并行恢复成为可能,并通过推测执行减轻掉队问题。

4.2.2 雅虎！ S4

S4（简单可扩展流系统)[52] 是一个完全分散的分布式流处理引擎,其灵感来自 MapReduce [283] 和 Actors 模型 [99]。

基本上,计算是由分布在集群中的处理元素（PE)执行的,消息以数据事件的形式在它们之间传输,这些消息根据它们的身份被路由到相应的PE。特别是,事件由其类型和键标识,而 PE 由其功能和它打算使用的事件定义。传入的流数据首先被转换为事件流,然后由用户为特定应用程序定义的一系列 PE 进行处理。然而,S4 并没有设计提供数据容错,因为即使支持自动 PE 故障转移到备用节点,如果没有用户定义的状态/消息备份功能,故障 PE 的状态和消息也会在切换过程中丢失PE里面。

5定性比较

在本节中,我们从数据模型、支持的工作负载、索引、并发控制、容错、内存溢出控制和查询处理策略等方面总结了本文阐述的一些代表性内存数据管理系统,如表 3 所示。

一般来说,内存数据管理系统也可以根据其存储和数据分析等功能分为三类,即存储系统、分析系统和同时具备这两种功能的成熟系统：

内存存储系统纯粹是为高效的存储服务而设计的,例如仅用于 OLTP 的内存关系数据库（例如,H-Store12 [36]、Silo [39]、Microsoft Hekaton [37]）,没有分析支持的 NoSQL 数据库（例如,RAMCloud [75]、Masstree [249]、MICA [64]、Mercury [250]、Kyoto/Tokyo Cabinet [251], Bitsy [254]), cache systems (eg, Memcached [61], MemC3 [63], TxCache [267], HashCache [268]), etc. 存储服务更关注低延迟和高吞吐量短期查询作业,并配备了用于在线查询的轻量级框架。它通常充当上层应用程序（例如,Web 服务器、ERP)的底层,其中快速响应是服务级别协议的一部分。

内存分析系统专为大规模数据处理和分析而设计,例如内存大数据分析系统（例如,Spark/RDD [55]、Piccolo [59]、Pregel [281]、GraphLab [47]、Mammoth [56]、Phoenix [57])和实时内存处理系统（例如,Storm [53]、Yahoo! S4 [52]、Spark Streaming [54]、MapReduce Online [282]）。这些系统的主要优化目标是通过实现高并行性（例如,多核、分布式、SIMD 和流水线)和批处理,最大限度地减少分析作业的运行时间。

成熟的内存系统不仅包括支持 OLTP 和 OLAP 的内存关系数据库（例如,HyPer [35]、Crescando [227]、HYRISE [176]）,还包括具有通用查询功能的数据存储语言支持（例如,SAP HANA [77]、Redis [66]、MemepiC [60]、[138]、Cit rusleaf/Aerospike [34]、GridGain [51]、MongoDB [65]、Couchbase [253]、MonetDB [256],三位一体 [46]）。这类系统的一个主要挑战是通过使用适当的数据结构和组织、资源争用等,在两种不同的工作负载之间做出合理的权衡;并发控制也非常重要,因为它处理并发的混合工作负载。

6研究机会

在本节中,我们简要讨论内存数据管理的研究挑战和机遇,包括以下优化方面,这些方面已在前面的表 1 中介绍：

索引。现有的内存数据库索引工作试图优化时间和空间效率。Hash-based索引简单易实现,访问时间复杂度为O(1),而tree-based索引天然支持范围查询,通常具有良好的空间效率。基于 Trie 的索引具有有限的 O(k) 时间复杂度,其中 k 是键的长度。还有其他类型的索引,例如位图和跳跃列表,它们适用于高效的内存中和分布式处理。例如,跳过列表,它

12、在H-Store网站的基础上,加入了一个新的基于JVM快照的实验性OLAP引擎。基于它的主要关注点,我们将其归入存储类别。

表3
内存数据管理系统的比较

	Systems	Data Model	Workloads	Indexes	Concurrency Control (CC)	Fault Tolerance	Memory Overflow	Query Processing
Relational Databases	H-Store	relational (row)	OLTP	hashing, B^+-tree, binary tree	partition, serial execution, light-weight locking, speculative CC	command logging, checkpoint, replica	anti-caching	stored procedure
	Hekaton	relational (row)	OLTP	latch-free hashing, Bw-tree	optimistic MVCC	logging, checkpoint, replica	Project Siberia	complied stored procedure
	HyPer/ ScyPer	relational	OLTP, OLAP	hashing, balanced search tree, ART	virtual snapshot, strict timestamp ordering (STO), partition, serial execution for OLTP	logging, checkpoint, replica	compression	JIT, stored procedure
	SAP HANA	relational, graph, text	OLTP, OLAP	timeline index, CSB^+-tree, inverted index	MVCC, 2PC	logging, checkpoint, standby server, GPFS	table/partition-level swapping, compression	"calculation graph model"
NoSQL Databases	MemepiC	key-value	object operations, analytics	hashing, skip-list	atomic primitives, virtual snapshot	logging, replica	user-space VMM	JIT
	MongoDB	document (bson)	object operations, analytics	B-tree	database-level locking	memory-mapped file	N/A	N/A
	RAMCloud	key-value	object operations	hashing	fine-grained locking	logging, replica	N/A	N/A
	Redis	key-value	object operations	hashing	single-threaded	logging, checkpoint	compression	scripting
Graph Databases	Bitsy	graph	OLTP	N/A	optimistic concurrency control (version)	logging, backup	N/A	stored procedure
	Trinity	graph	graph operations	N/A	fine-grained spin-lock	replica, Trinity File System (TFS)	N/A	stored procedure
Cache Systems	Memcached	key-value	object operations	hashing	fine-grained locking	N/A	N/A	N/A
	MemC3	key-value	object operations	hashing	lock striping, optimistic locking	N/A	N/A	N/A
	TxCache	key-value	OLTP	hashing	MVCC	N/A	N/A	N/A
Big Data Analytics Systems	M3R	key-value	analytics	N/A	partition, locking	N/A	N/A	offline
	Piccolo	key-value	analytics	hashing	locking	checkpoint	N/A	offline
	Spark/ RDD	RDD	analytics	N/A	partition, read/write locking	lineage, checkpoint	block-level swapping	offline
Real-time Processing Systems	Spark Streaming	RDD	streaming	N/A	partition, read/write locking	lineage, replica, checkpoint	block-level swapping	N/A
	Yahoo! S4	Event	streaming	hashing	message passing	standby server	N/A	N/A

允许对具有 O(log n) 复杂度的有序元素序列进行快速点查询和范围查询,正在成为内存数据库中 B 树的理想替代方案,因为它可以轻松实现无闩锁,因为它的分层结构。

内存数据库的索引与基于磁盘的数据库的索引不同,后者侧重于 I/O 效率,而不是内存和缓存利用率。

对于基于散列索引和特里索引实现的点访问,设计一个具有恒定时间复杂度的索引,基于树和基于特里索引实现的范围访问的有效支持,以及基于散列和特里索引实现的良好空间效率,设计一个索引将非常有用。基于树的索引(如 ART 索引 [87])。

无锁或无锁索引结构对于实现高并行性而没有与闩锁相关的瓶颈至关重要,无索引设计或有损索引也很有趣,因为它具有 DRAM 的高吞吐量和低延迟 [41],[64] .

数据布局。数据布局或数据组织对于内存系统的整体性能至关重要。缓存意识设计,如柱状结构、缓存行对齐和空间利用优化,如压缩、数据碎片整理是内存数据组织的主要焦点。在主存系统中引入了连续数据分配作为日志结构的思想,以消除数据碎片问题并简化并发控制[2]。但是,为内存系统设计一个独立于应用程序的数据分配器可能会更好,该分配器具有内存系统的通用内置功能,例如容错、应用程序辅助数据压缩和碎片整理。

并行性。为了加快处理速度,应该利用三个级别的并行性,这在第 1 节中有详细说明。在现代体系结构中提供的指令级(例如,位级并行性,SIMD)增加并行性通常是有益的,它可以实现近乎最佳的加速,没有并发问题和其他开销,但对允许的最大并行度和要操作的数据结构有限制。指令级并行性可以产生良好的性能提升,因此在设计高效的内存数据管理系统时应该考虑它,特别是在数据结构的设计中。随着

许多集成核心 (MIC) 协处理器（例如,英特尔至强融核）,它为并行计算提供了一个有前途的替代方案,具有更广泛的 SIMD 指令、许多低频有序核心和硬件环境 [288]。

并发控制/事务管理。对于内存系统,并发控制的开销会显着影响整体性能,因此如果完全没有并发控制就完美了。因此,值得使串行执行策略对跨分区事务更有效,对倾斜的工作负载更稳健。

无锁或无锁并发控制机制在内存数据管理中很有前途,因为重量级的基于锁的机制可以大大抵消内存环境提高的性能。大多数主流编程语言中提供的原子原语是有效的替代方案,可用于设计无锁并发控制机制。此外,HTM 为高效并发控制协议提供了一种硬件辅助方法,尤其是在数据库中的事务语义下。在对延迟敏感的内存环境中,硬件辅助方法是不错的选择,因为软件解决方案通常会产生大量开销,从而抵消了并行性和快速数据访问带来的好处。

但是我们应该注意它在某些情况下的意外中止。这些数据保护机制（即 HTM、锁、时间戳、原子原语)的组合应该能够实现更有效的并发控制模型。此外,该协议应该是数据局部敏感和缓存感知的,这对现代机器来说更重要 [192]。

查询处理。即使在传统的基于磁盘的数据库中,查询处理也是一个广泛研究的研究课题。然而,基于 Iterator-/Volcano-style 模型的传统查询处理框架虽然灵活,但由于其代码/数据局部性较差,不再适用于内存数据库。

现代 CPU 的高计算能力和易于使用的编译器基础设施（如 LLVM [167])支持高效的动态编译 [112],由于更好的代码和数据局部性,可以显着提高查询处理性能。 SIMD 或多核增强处理可用于加速复杂的数据库操作,如连接和排序,NUMA 架构将在未来几年发挥更大的作用。

容错。为了保证持久性,容错是内存数据库的必要条件;但是,它也是 I/O 造成的主要性能瓶颈。因此,容错的一种设计理念是通过尽可能减少关键路径中的 I/O 成本,使其对正常操作几乎不可见。命令记录 [131] 可以减少需要记录的数据,而 RAMCloud [2] 和 2-solidDB [40] 的安全可见策略使用的远程记录可以通过在远程节点记录数据并回复来减少响应时间一旦数据被写入缓冲区。快速恢复可以在发生故障时提供高可用性,这可以以更多和组织良好的备份文件（日志/检查点)为代价来实现。应进一步检查对正常性能的干扰与恢复效率之间的权衡[132]。硬件/操作系统辅助方法很有前途,例如 NVRAM、内存映射文件,在此之上需要优化算法和数据结构才能发挥其性能潜力。

数据溢出。一般来说,解决数据溢出问题的方法可以分为三类:用户空间（例如,H-Store Anti-caching [133],Hekaton Siberia [134]）,内核空间（例如,OS Swap,MongoDB内存映射文件 [65])和混合（例如,Efficient OS Paging [136] 和 UVMM [138]）。语义感知的用户空间方法可以对分页策略做出更有效的决策,而硬件意识和开发良好的内核空间方法能够利用操作系统在交换期间带来的 I/O 效率。

潜在地,可以利用语义感知的分页策略和硬件感知的 I/O 管理来提高性能 [136]、[138]。

除了上述之外,硬件解决方案正越来越多地被用于提高性能。特别是,新的硬件/架构解决方案,如 HTM、NVM、RDMA、NUMA 和 SIMD,已被证明能够显着提高内存数据库系统的性能。由于 DRAM 在总功耗中所占比例相对较大 [289]、[290],并且分布式计算进一步加剧了该问题,因此在内存系统中,能效也变得越来越有吸引力。

在基于磁盘的系统中被认为可以忽略不计的每一个操作开销,都可能成为基于内存的系统的新瓶颈。因此,消除这些遗留瓶颈（例如系统调用、网络堆栈和跨缓存行数据布局)将有助于显着提升内存系统的性能。此外,如 [117] 中的示例,在对开销敏感的内存环境中,即使实现也很重要。

7结论

随着内存成为新的磁盘,内存中的数据管理和处理变得越来越受到学术界和工业界的关注。将数据存储层从磁盘转移到主内存可以在响应时间和吞吐量方面带来超过 100 的理论上的改进。当数据访问变得更快时,在传统的基于磁盘的系统中无关紧要的每一个开销来源都可能会显着降低整体性能。这种转变促使人们重新思考传统系统的设计,尤其是数据库,在数据布局、索引、并行性、并发控制、查询处理、容错等方面。现代 CPU 利用率和内存层次结构意识优化在内存系统的设计中发挥着重要作用,HTM 和 RDMA 等新硬件技术为解决软件解决方案遇到的问题提供了一个有希望的机会。

在本次调查中,我们重点介绍了内存数据管理和处理的设计原则,以及设计和实现高效、高性能内存系统的实用技术。我们回顾了内存层次结构和一些高级技术,例如 NUMA 和事务内存,它们为内存中数据管理和处理提供了基础。此外,我们还讨论了一些开创性的内存中 NewSQL 和 NoSQL 数据库,包括缓存系统、批处理和在线/连续处理系统。我们详细地强调了一些有前途的设计技术,从中我们可以学到实用和具体的系统设计原则。

本综述对内存管理中的重要技术进行了全面回顾,并对迄今为止的相关工作进行了分析,希望能为进一步的面向内存的系统研究提供有用的资源。

致谢

这项工作得到了 A*STAR 项目 1321202073 的支持。
作者要感谢匿名审稿人,以及 Bingsheng He、Eric Lo 和 Bogdan Marius Tudor 富有洞察力的评论和建议。

参考

[1] S. Robbins,“RAM is the new disk”,InfoQ 新闻,2008 年 6 月。
[2] J. Ousterhout、P. Agrawal、D. Erickson、C. Kozyrakis、J. Leverich、D. Mazieres、S. Mitra、A. Narayanan、G. Parulkar、M. Rosenblum、SM Rumble、E. Stratmann 和 R. Stutsman,“RAMClouds 案例:完全在 dram 中的可扩展高性能存储”,ACM SIGOPS Operating Syst。牧师,卷。 43,第 92-105 页,
2010.
[3] F. Li、BC Ooi、MT Ozsu 和 S. Wu,“使用 MapReduce 进行分布式数据管理”,ACM Comput。幸存者,卷。 46,第 31:1–31:42,2014 年。

[4] 惠普。 (2011)。 Vertica 系统 [在线]。可用:http://www.vertica.com
[5] Hadapt 公司 (2011)。 Hadapt:Hadoop 上的 SQL [在线]。可用:http://hadapt.com/ A. Thusoo、JS Sarma、N. Jain、Z. Shao、P. Chakka、S. Anthony、H. Liu、P. Wyckoff 和
[6] R. Murthy,“Hive:A基于 map-reduce 框架的仓储解决方案”,Proc。 VLDB 捐赠基金,第一卷。 2,第 1626–1629 页,2009 年。

[7]阿帕奇。 (2008)。 Apache hbase [在线]。可用:http://hbase。 apache.org/ JC Corbett、J. Dean、M. Epstein、A. Fikes、C. Frost、JJ Furman、S. Ghemawat、A. Gubarev、C.
[8] Heiser、P. Hochschild、W. Hsieh、S.
Kanthak、E. Kogan、H. Lee、A. Lloyd、S. Melnik、D. Mwaura
Nagle、S. Quinlan、R. Rao、L. Rolig、Y. Saito、M. Szymaniak、C.S.
Taylor、R. Wang 和 D. Woodford,“Spanner:Google 的全球分布式数据库”,
Proc。 USENIX 症状。操作系统
的。实施,2012,p. 251–264。
[9] S. Alsubaiee、Y. Altowim、H. Altwaijry、A. Behm、VR Bor kar、Y. Bu、MJ Carey、I. Cetindil、M. Cheelangi、K. Faraaz、E.
Gabrielova、R. Grover、Z. Heilbron、Y. Kim、C. Li、G. Li、JM
Ok、N. Onose、P. Pirzadeh、VJ Tsotras、R. Vernica、J. Wen 和 T. Westmann,
“Asterixdb:一种可扩展的开源 BDMS”,Proc。超大型数据库,第 1905–1916 页,
2014 年。
[10] MySQL AB。 (1995)。 Mysql:世界上最流行的开源数据库[在线]。可用:http://www.mysql.com/ [11] Apache。 (2008)。 Apache cassandra [在线]。可用:http://cassandra.apache.org/ [12] 甲骨文。 (2013)。 Oracle 数据库 12c [在线]。可用:https://www.oracle.com/database/index.html [13] Neo Technology,“Neo4j - 世界领先的图形数据库”
2007. [在线]。可用:http://www.neo4j.org/ [14] Aurelius。
(2012)。 Titan 分布式图形数据库 [在线]。
可用:http://thinkaurelius.github.io/titan/ [15] A. Kyrola、G.
Blelloch 和 C. Guestrin,“Graphchi:仅在一台 pc 上进行大规模图形计算”,Proc。第 10 届 USENIX 会议。
操作系统德斯。实施,2012 年,第 31-46 页。

[16] 客观性公司 (2010)。无限图 [在线]。可用:http://www.objectivity.com/infinitegraph
[17]阿帕奇。 (2010)。 Apache Hama [在线]。可用:https:// hama.apache.org [18] A. Biem、E. Bouillet、H. Feng、A. Ranganathan、A. Riabov、O. A. 和 J. A. Biem。
Verscheure、H. Koutsopoulos 和 C. Moran,“用于可扩展、实时、智能交通服务的 IBM 信息领域流”,Proc。 ACM SIGMOD 诠释。会议。管理。数据,2010 年,第 1093–1104 页。

[19] S. Hoffman,Apache Flume:Hadoop 的分布式日志收集。
伯明翰,英国 Packt 出版社,2013 年。
[20]阿帕奇。 (2005)。 Apache hadoop [在线]。可用:http://hadoop.apache.org/ [21] V. Borkar、M. Carey、R. Grover、N. Onose 和 R. Vernica,“Hyracks:数据密集型计算的灵活且可扩展的基础,”在过程中。 IEEE 第 27 届国际会议。会议。数据工程,2011 年,第 1151–1162 页。

[22] M. Isard、M. Budiu、Y. Yu、A. Birrell 和 D. Fetterly,“Dryad:来自顺序构建块的分布式数据并行程序”,Proc。第二届 ACM SIGOPS/EuroSys Eur。会议。电脑。
系统,2007 年,第 59-72 页。
[23] D. Jiang, G. Chen, BC Ooi, K.-L. Tan 和 S. Wu,“epiC:用于处理大数据的可扩展和可扩展系统”,Proc。
VLDB 捐赠基金,第一卷。 7,第 541–552 页,2014 年。
[24] HT Vo、S. Wang、D. Agrawal、G. Chen 和 BC Ooi,“LogBase:云中可扩展的日志结构数据库系统”,Proc。
VLDB 捐赠基金,第一卷。 5,第 1004–1015 页,2012 年。
[25] G. DeCandia、D. Hastorun、M. Jampani、G. Kakulapati、A. Lakshman、A. Pilchin、S. Sivasubramanian、P. Vosshall 和 W. Vogels,“Dynamo:Amazon 的高可用性键值存储”,
ACM SIGOPS 操作系统牧师,卷。 41,第 205–220 页,2007 年。
[26] S. Ghemawat、H. Gobioff 和 S.-T。 Leung,“Google 文件系统”,载于 Proc。第 19 届 ACM 症状。操作系统原则,2003 年,第 29-43 页。

[27] K. Shvachko、H. Kuang、S. Radia 和 R. Chansler,“hadoop 分布式文件系统”,Proc。 IEEE 第 26 届 Symp。海量存储系统技术,2010 年,第 1-10 页。

[28] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, BC Ooi, HT Vo, S. Wu 和 Q. Xu,“ES2:支持 OLTP 和 OLAP 的云数据存储系统”,Proc。 IEEE 第 27 届国际会议。会议。数据工程,2011 年,第 291–302 页。

[29] 基金会数据库。 (2013)。 Foundationdb [在线]。可用的:https://foundationdb.com
[30] DG Andersen、J. Franklin、M. Kaminsky、A. Phanishayee、L. Tan 和 V. Vasudevan,“Fawn:一组快速的弱节点”,Proc。 ACM SIGOPS 第 22 届症状。操作系统原则,2009 年,第 1-14 页。

[31] H. Lim、B. Fan、DG Andersen 和 M. Kaminsky,“Silt:内存高效、高性能的键值存储”,Proc。第 23 届 ACM 症状。操作系统原则,2011 年,第 1-13 页。

[32] B. Debnath、S. Sengupta 和 J. Li,“Skimpystash:Ram space skimpy key-value store on flash-based storage”,Proc。 ACM SIGMOD 诠释。会议。管理。数据,2011 年,第 25-36 页。
[33] Clustrix 公司(2006 年)。 Clustrix [在线]。可用:http://www. clustrix.com/ [34] V. Srinivasan 和 B. Bulkowski,“Citrusleaf:一种保留酸的实时 NoSQL 数据库”,Proc。诠释。会议。超大型数据库,2011,卷。 4,第 1340–1350 页。

[35] A. Kemper 和 T. Neumann,“HyPer:基于虚拟内存快照的混合 OLTP 和 OLAP 主内存数据库系统”,IEEE 第 27 届国际会议。会议。数据工程,2011 年,第 195–206 页。

[36] R. Kallman、H. Kimura、J. Natkins、A. Pavlo、A. Rasin、S. Zdonik、EPC Jones、S. Madden、M. Stonebraker、Y. Zhang、J. Hugg 和 DJ Abadi, “H-store:一种高性能的分布式主内存事务处理系统,”Proc。 VLDB 捐赠基金,第一卷。 1,第 1496–1499 页,2008 年。

[37] C. Diaconu、C. Freedman、E. Ismert、P.-A。拉尔森、P. 米塔尔、R. Stonecipher、N. Verma 和 M. Zwilling,“Hekaton:SQL 服务器的内存优化 OLTP 引擎”,Proc。 ACM SIGMOD 诠释。
会议。管理。数据,2013 年,第 1243–1254 页。
[38] T. Lahiri, M.-A. Neimat 和 S. Folkman,“Oracle timesten:用于企业应用程序的内存数据库”,IEEE 数据工程。
公牛,卷。 36,没有。 2,页。 6–13,2013 年 6 月。
[39] S. Tu、W. Zheng、E. Kohler、B. Liskov 和 S. Madden,“多核内存数据库中的快速交易”,Proc。 ACM 症状。操作系统原则,2013 年,第 18-32 页。