浙江工业大学

硕士学位论文

论文题目： 面向大数据流的分布式 B+树索引构建

作者姓名　　卢晨曦

指导教师　　杨良怀　教　授

学科专业　　计算机科学与技术

学位类型　　工学硕士

培养类别　　全日制学术型硕士

所在学院　　计算机科学与技术学院

提交日期：2019 年 06 月

万方数据

A Distributed B+ Tree for Big Data Stream

Dissertation Submitted to

Zhejiang University of Technology

in partial fulfillment of the requirement

for the degree of

Master of Engineering

by

Chen-xi LU

Dissertation Supervisor: Prof. Liang-huai YANG

Jun., 2019

万方数据

浙江工业大学学位论文原创性声明

本人郑重声明：所提交的学位论文是本人在导师的指导下，独立进行研究工作所取得的研究成果。除文中已经加以标注引用的内容外，本论文不包含其他个人或集体已经发表或撰写过的研究成果，也不含为获得浙江工业大学或其它教育机构的学位证书而使用过的材料。对本文的研究作出重要贡献的个人和集体，均已在文中以明确方式标明。本人承担本声明的法律责任。

作者签名：[signature]　　日期：2019年 6 月

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权浙江工业大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于 1、保密□，在一年解密后适用本授权书。

2、保密□，在二年解密后适用本授权书。

3、保密□，在三年解密后适用本授权书。

4、不保密☑。

（请在以上相应方框内打“√”）

作者签名：[signature]　　日期：2019年 6 月

导师签名：[signature]　　日期：2019年 6 月

万方数据

中图分类号 TP31　　学校代码 10337
UDC 004　　密级 公开
研究生类别 全日制学术型硕士研究生

浙江工业大学

硕士学位论文

面向大数据流的分布式 B+树索引构建

A Distributed B+ Tree for Big Data Stream

作　　者 卢晨曦　　第一导师 杨良怀 教　授
申请学位 工学硕士
学科专业 计算机科学与技术　　培养单位 计算机科学与技术学院
研究方向 数据流存储　　答辩委员会主席 黄德才

答辩日期： 2019 年 05 月 21 日

万方数据

面向大数据流的分布式 B+树索引构建

摘　要

随着大数据时代的到来，数据的产生及其应用方式更加多元化。数据流是一种特殊的大数据形态，具有实时性、无限性、突发性等特点，在众多领域有着广泛的应用，具有很高的价值。然而，数据流的流速快且数据量巨大，其在实时处理、存储和查询等方面都存在很大的挑战。对此，本文提出了一种适用于数据流场景的分布式索引结构，其能支持数据流的高效存储与查询。本文贡献如下：

1. 提出了一种适用于大数据流场景的分布式 B+树索引结构：WB-Index。WB-Index 是一种双层的主从索引结构，其利用时间窗口机制切分数据流。在每个时间窗口内，根据流元组内容构建 B+树索引作为底层索引，针对各连续时间窗口，以时间窗口起始时间戳作为“Key”值，时间窗口对应底层索引元信息作为“Value”值构建顶层 B+树索引。WB-Index 将底层索引分发到多个节点来减轻索引维护压力。WB-Index 系统架构中，通过多种节点类型将流元组存储、索引构建和查询请求分离，从而满足数据流的高效存储与查询。
2. 针对 WB-Index 索引结构，提出了高效的索引构建方法。由于数据流流速快，索引的构建效率至关重要。针对底层索引，提出基于并行排序的预装载 B+树批量构建法，提高底层索引构建效率；针对顶层索引，提出预分配节点空间的不分裂 B+树更新方法，保证更新效率，提高空间利用率。所提索引构建方法构建速度快、时延低，能够支撑高速数据流的索引构建。
3. 针对 WB-Index 索引结构，提出了高效的持久化方法。由于数据流具有无限性，本文利用分布式文件系统存储数据流以及 WB-Index 索引结构，设计了紧凑的存储格式来减少存储开销。为了提高索引持久化后的查询效率，底层索引中添加辅助索引结构来过滤不必要的查询请求，并通过缓存新数据和热点数据来进一步提高查询效率。

WB-Index 能支持数据流的高效存储和查询，通过理论评估和实验分析证明了 WB-Index 的有效性。

关键词：数据流，B+树，分布式索引，分布式存储

万方数据

A Distributed B+ Tree for Big Data Stream

ABSTRACT

With the coming of the era of big data, the generation and application of data have become more diversified. As a special form of big data, data stream is characterized by timeliness, infinity and suddenness, etc. With high value, it's used in many fields widely. However, result from the high speed and large amount of data stream, there are great challenges of real-time processing, storage and query. In this regard, a distributed index structure is presented, which can support efficient storage and query of data stream. Contributions are as follows:

1. Proposed a distributed B+ tree index structure: WB-Index, which can be applied to data stream. WB-Index adopts master-slave index structure, splits data stream with the use of time window. In each time window, according to the content of tuple, B+ tree index is built as the bottom index. For each continuous time window, the timestamp of the window and the corresponding bottom index meta compose <key, value> to build top index. WB-Index distributes the bottom index to multiple nodes to ease index maintenance. In the architecture of WB-Index system, tuple storage, index construction and query are separated by multiple nodes, so as to satisfy the efficient storage and query of data stream.
2. An efficient index construction method is proposed for WB-Index. The efficiency of index construction is of great importance. For the bottom index, a batch construction method is proposed to accelerate the bottom index construction. For the top index, a non-splitting B+ tree updating method of pre-allocated node space is proposed to ensure updating efficiency and improve space utilization.
3. An efficient persistence method is proposed for WB-Index. Because of the infiniteness of data stream, this paper uses the distributed file system to store the data stream and distributed index structure, and has designed a compact storage format to reduce the storage overhead. In order to improve the query efficiency after index persistence, secondary index structure is added in the

万方数据

bottom index to filter unnecessary queries, and new data and hot data are cached to further improve the query efficiency.

WB-Index can support efficient storage and query of data stream, the validity of the WB-Index is proved by calculation and experiment.

KEY WORDS: data stream, B+ tree, distributed index, distributed storage

万方数据

目 录

万方数据

万方数据

万方数据

插图清单

万方数据

万方数据

附表清单

万方数据

第一章 绪 论

1.1 背景及意义

随着计算机产业的高速发展，计算机应用更加多元化，人们每天产生的数据量呈几何级增长，能够达到 PB 级，当今时代被称为大数据时代。大数据具有多样性(variety)、价值性(value)、规模性(volume)、高速性(velocity)等特点，大数据的存储、分析、应用已成为当今的研究热点。数据流是大数据中的一种特殊数据形态，其表现形式为：$\{\dots,a_{t-1},a_t,a_{t+1},\dots\}$[1]，其中 a_t 表示在 t 时刻到达的数据，称为流元组。

数据流出现在电子商务、物联网、智慧交通等众多领域，对数据流的分析处理能用于实时预测、推荐、监控等应用[2,3]，具有极高的价值。例如在城市交通体系中，所有的监控卡口会提取经过的车辆信息，实时的产生海量的交通数据流。交通数据流可用于实时的车流监控、车流预测、车辆追踪等应用，能有效的提高交通的管制效率。在电子商务领域，系统会实时采集用户的行为数据，根据这些行为数据流，可以动态的调节销售策略、展示方式，从而使得利益最大化。相比静态数据集，数据流有以下几个显著特点[4]：

1. 实时性：数据流具有很强的时效性，最有利用价值的往往是最新的数据。
2. 高速性：数据流的流速快，且有较大的波动性。
3. 无限性：数据流时时刻刻都会产生，无法评估其数据规模。
4. 易失性：数据流到达后一旦没能及时保存和处理，就无法重新获取。

数据流具备的这些特点，使其在存储、计算、查询等方面都面临着很大的挑战。存储方面，由于数据流的无限性和高速性，传统的数据库无法支撑高速的更新操作，而且传统的单机数据库也无法存储海量的数据流；计算方面，由于数据流的易失性和高速性，传统的单机处理方式计算效率低，计算速度无法匹配数据流到达的速度，会造成数据的丢失；查询方面，数据流具有很强的时效性，关于数据流的查询要求能够快速响应连续的实时查询。对于海量的历史流数据，也需要能够快速检索，以支持对数据流的深度分析。

合理的索引结构能有效地提升数据的获取效率，然而索引的构建过程会产生一定的时间开销。传统的数据库无法支撑高速数据流频繁的索引更新，而且在海量数据规模下，索引规模也变得巨大，集中式架构难以维护相应的索引结构。对此，本文旨在设计适用于数据流场景的分布式索引结构，来解决由数据

万方数据

流的实时性、高速性、无限性所带来的存储和查询问题。本文所提分布式索引能够支持高速数据流的实时存储，支持数据流中新数据和热点数据的实时查询，确保海量历史数据流的高效查询，从而最大化地发挥数据流的价值。

1.2 本文研究内容及贡献

1.2.1 研究内容

随着计算机软硬件和通信网络的高速发展，数据流在众多领域有着广泛的应用，数据流的存储、查询等相关工作也成为了研究热点。

数据流应用场景中，需要能够快速响应对实时数据的连续查询，也需要高效查询历史数据以支持深度分析。传统集中式架构的计算和存储能力有限，难以处理高速数据流，也无法存储海量数据。因此本文采用分布式架构，通过多种节点的有机结合，来支持数据流的高效存储与查询。

构建索引是提高查询效率的重要手段。海量数据下，索引的规模巨大，索引的更新、存储同样也面临巨大的挑战。对此，研究人员提出了多种分布式索引结构，但都没能很好地结合数据流的特性，无法适用于数据流场景。本文旨在结合数据流的特点，提出一种适用于数据流的分布式索引结构。数据流的流速快，分布式索引的构建效率尤为重要。本文采用 B+树作为基本索引结构，然而传统的 B+树索引构建方法构建效率低，无法满足高速数据流的实时更新，本文旨在根据分布式索引结构提出高效的索引构建方法。

索引和数据的存储方式对查询性能也有很大的影响。传统数据库一般采用行式存储格式。在分布式存储场景下，出现了大量的列式存储格式，并结合一些简单的内部索引结构，使其能够在数据处理过程中减少无用数据的读取开销。本文旨在为数据流和分布式索引设计合理的持久化方法，在保证存储效率的同时提高数据流的查询效率。

当前，内存存储介质的容量越来越大，价格也越来越低，越来越多的系统利用内存来提高数据处理和查询的性能。结合这一趋势，本文在所提方案中，更多利用内存来提高索引的构建效率、数据流的实时查询效率以及热点数据的查询效率。

综上所述，本文针对海量高速数据流，高效地构建分布式索引，从而支持数据流的实时存储和高效查询。

1.2.2 本文贡献

本文深入研究了数据流在存储和索引等方面的问题。根据目前存在的问题，提出了一种适用于数据流场景的分布式索引结构，来支持大数据流的实时存储和高效查询。本文贡献如下：

万方数据

1. 提出了一种适用于大数据流场景的分布式索引结构：WB-Index(Window Based Index)。WB-Index 采用双层的主从索引结构，利用时间窗口机制来切分数据流。每个时间窗口内，根据流元组内容构建 B+树索引作为底层索引。针对各个连续的时间窗口，以时间窗口起始时间戳作为“Key”值，对应窗口底层索引元信息作为“Value”值构建顶层 B+树索引。WB-Index 将底层索引分发到多个节点来减轻索引维护压力。为了更好地缓存底层索引，底层索引采用索引结构和流元组分开存储的策略。WB-Index 系统架构中，由索引构建节点、查询节点、存储节点、协调节点和主控节点这 5 类节点构成，将流元组存储、索引构建和查询请求分离，从而满足数据流的高效存储与查询。
2. 针对 WB-Index 索引结构，提出了高效的索引构建方法。数据流流速快，索引构建效率至关重要。本文采用 B+树作为基本索引结构，对于底层 B+树索引，本文提出基于并行排序的预装载 B+树批量构建法，提高底层索引构建效率；对于顶层 B+树索引，本文提出预分配节点空间的不分裂 B+树更新方法，保证更新效率，提高空间利用率。所提构建方法构建速度快、时延低，能够支撑高速数据流的索引构建。
3. 针对 WB-Index 索引结构，提出了高效的持久化方法。数据流具有无限性，本文利用分布式文件系统存储数据流以及分布式索引结构，并设计紧凑的存储格式来减少存储开销。为了提高索引持久化后的查询效率，底层索引中添加辅助索引结构来过滤不必要的查询开销，并通过缓存新数据和热点数据来进一步提高查询效率。
4. 通过理论评估和实验分析证明了 WB-Index 的有效性。

1.3 论文组织

本文的结构组织如下：

第一章，绪论。详细介绍了本文选题的背景及意义，并列出了本文的主要工作内容及贡献。

第二章，相关工作。介绍了课题涉及到的数据流处理技术、索引技术和分布式存储技术等在国内外的研究成果与现状。

第三章，面向大数据流的分布式索引架构。详细阐述了 WB-Index 索引的基本结构、WB-Index 系统架构以及索引的构建和查询流程。

第四章，分布式索引构建方法。详细介绍了 WB-Index 索引的构建方法，包括底层索引构建、底层索引发布、顶层索引更新三大模块，并通过实验验证了构建方法的有效性。

万方数据

第五章，分布式索引持久化方法。详细介绍了流元组和索引结构的持久化方法，并通过实验验证持久化方法的有效性。

第六章，结论与展望。总结了本文的研究工作，展望数据流存储和索引等相关主题的后续研究与发展。

1.4 本章总结

本章主要阐述了本课题的研究背景和意义，根据目前数据流存储和索引等相关工作存在的不足之处，提出了相应的研究内容及方法，最后阐述了本文的内容安排和框架结构。

万方数据

第二章 相关工作

本章首先介绍了集中式和分布式数据流处理系统，总结了数据流处理的几种方式；随后阐述和分析了常用的索引结构以及加速索引构建的方法；最后，阐述了多种分布式存储系统以及多种分布式索引方法，综述了合理利用内存、优化存储格式等提高海量数据处理、存储和查询效率的相关工作。

2.1 数据流处理

数据流在金融、交通、监控等领域有着广泛的应用。由于数据流具有无限性和实时性等特点，使其在存储、计算、查询等方面都面临着巨大的挑战。不同应用场景下，数据流的处理方式有所差异，主要可以归纳为以下 4 种处理方式[5]：

1. 实时处理。每来一条流元组就会触发处理操作，此模式实时性高，但吞吐率较低。
2. 微批处理。将一定量的数据流缓存后再处理，此方式将流式计算转换成批量计算，虽然损失了一定的实时性，但可以大幅度提升吞吐率。
3. 抽样处理。若数据流流速过快，远远超过处理能力，可对数据流抽样，只处理部分数据流。这种方式类似于近似计算，适用于对精度要求不高的场景。
4. 梗概处理。将数据流元组中的一些无用字段去除，保持元组骨架，减少在系统中的存储和通讯开销，从而提高数据流处理效率。

早期的研究人员根据不同的应用场景，提出了多种数据流管理系统，以支持数据流的存储、计算和查询。Aurora[6]是针对监控应用提出的数据流管理系统，其预定义了 7 种不同的操作符。针对每个监控数据流，通过组合多个操作符生成一个 DAG 图，在此基础上处理数据流。Aurora 支持持续查询、即席查询和视图查询三种查询方式。加州大学伯克利分校提出的 TelegraphCQ[7,8]是面向数据流的连续查询系统，支持数据流的实时监控。斯坦福大学提出的 Stream[9]系统，支持数据流的连续查询，处理过程中，使用滑动窗口切分数据流，将无限流转化为有界流。对于高速数据流，Stream 通过合理的资源调度，对数据流进行抽样处理，提供较为准确的近似结果。以上这些数据流管理系统都是集中式架构，由于系统资源有限，能最大支撑的数据流流速小，且每个数据流管理系统都有明确的应用场景，缺乏通用性。

万方数据

近年来，随着数据流的广泛应用，涌现出了一批分布式流处理框架[10]。Storm[11]是由 Twitter 推出的分布式数据流处理框架，Storm 内部将一个任务抽象为一个 Topology，其中由 Spout 和 Bolt 两种节点构成，Spout 负责数据流的摄取，Bolt 负责数据流的处理，通过在不同节点定义处理逻辑来高效处理数据流。节点之间通过 ACK 机制保证数据流处理的可靠性。另外，Storm 通过 Nimbus 节点和 Zookeeper 协调服务进行任务的分发、调度及管理。S4[12]是由雅虎推出的通用开放形分布式流处理系统，与 Storm 较为相似，通过定义一些数据流算子来处理数据流，另外也支持类 SQL 的方式定义数据流处理任务。S4 利用基于状态校验点的故障容错协议，保证系统的可靠性。Zaharia M[13]等人提出了一种离散型的数据流处理模型 D-streams，其核心思想是把数据流处理看成一个个连续时间区域内的批处理组合，该思想能够很好地将流处理和批处理进行结合，将流处理转换成微批量处理。这不仅可以大幅度提高数据流的处理效率，也能更好地进行容错处理，但会影响一定的实时性。Spark-streaming 即是基于 D-streams 思想设计出的基于 Spark[14]的分布式数据流处理框架。Flink 是另一种数据流处理框架，其核心思想为：将批处理和流处理统一，在流处理模式下支持完备的 SQL 查询。Flink 在数据流处理过程中，采用轻量级的分布式异步快照机制，实现高效的流状态维护[15]，也使得流处理过程更加的精确。以上所提的分布式数据流处理系统侧重于数据流的实时计算，没有涉及数据流的实时存储。

数据流存储方面，Druid[16]是一款适用于数据流的实时 OLAP 系统，Druid 根据预先设置的聚合方式，从时间维度对数据流进行预聚合，这种方式能大幅的减少数据流的存储压力，而且能结合位图索引支持高效的聚合查询。Druid 也充分考虑了数据流的时效性，通过对新数据缓存以提高新数据的查询效率，但对数据的预聚合会使得查询存在局限性。OpenTSDB[17]是一款分布式时序数据库，其底层基于分布式 Key-Value 存储系统 HBase。OpenTSDB 利用 HBase 写入性能好的特点，将时序数据的时间戳与时序数据的一些属性值组合作为“Key”值，并将完整的时序数据作为“Value”值存储在分布式 Key-Value 存储系统中，但 HBase 中缓存的数量有限，查询过程会涉及到多个持久化索引结构的顺序查找，查询性能不高。另外，由于“Key”值是由多个属性组合而成，查询时只能通过前缀属性查找，若查询条件不存在前缀属性时，查找过程会退化为全序遍历。Facebook 提出的 Gorilla[18]是一种基于内存的时序数据库，Gorilla 将较新的数据全部存储在内存中，为了节约内存存储空间，分别对数据流中的时间戳和数据值压缩存储。历史数据则会保存在 HBase 中，保存之前会通过预聚合来减少海量数据的存储开销，但由于内存的存储空间终究有限，在大数据流场景下，Gorilla 最多在内存中缓存一天的数据。

数据流的查询一直是研究的重点。为了提高数据流的查询效率，研究者结合数据流特点提出了适用于数据流的索引结构，主要分为以下 4 大类：

万方数据

1. 基于位图的索引结构。Pu 等人[19]提出了一种动态的位图索引结构，根据数据流概要结构中的某个字段将数据流存储在不同的文件中，查询时可以通过位图索引快速定位到对应的文件。
2. 基于滑动窗口的索引结构。其核心思想是用最近窗口中的数据代替整个数据流，这能有效减少数据流的处理和存储压力，但是基于此方法的查询只能提供近似结果，只适用于关心最新数据的应用场景。
3. 小波索引。小波索引的核心思想是：将数据流进行划分，并对每一部分数据流添加索引，这样在数据更新时，只需修改小部分的索引结构，无需全局修改。
4. 基于时间的索引结构。利用数据流的时间维度，通过预定义查询的方式，周期性地发送请求，并利用 B+树索引对查询结果构建索引。此方法通过预计算的方式提高查询效率。

上述 4 种数据流索引比起传统的索引方式具有很强的特殊性，缺乏通用性。

2.2 索引技术

海量数据下，索引技术可以有效地提高数据的查询效率，基本的索引类型主要有以下 3 类。

（1） Hash 索引

Hash[20,21]索引广泛地应用于 Key-Value 存储系统，其原理是根据“Key”值的 Hash 值直接定位插入或查找的位置，在理想情况下，查找和插入的时间复杂度为 O(1)。Hash 索引也存在一些明显的缺点，其无法根据“Key”值前缀进行查询，且无法支持范围查询。布隆过滤器[22]借鉴了 Hash 索引的思想，将一条记录通过多个 Hash 值映射到位数组中的不同位置上，通过位数组，可以快速判断一条记录是否存在。布隆过滤器具有构建速度快且占用内存小的优点，但是无法适用于具有数据删除操作的场景。对此，Fan B 等人提出 Cuckoo Filter[23]，对布隆过滤器进行优化，使其能够应用于具有数据删除操作的场景。

（2） 树形索引

树形索引结构包括 AVL 树[24]、红黑树[25]、B 树[26]、B+树、R 树[27]等。AVL 树即平衡二叉搜索树，其树节点最多包含两个子节点。为了达到 O(logn)的查询效率，需要保证树的平衡，因此在插入数据时，会有多种节点调整策略来保证 AVL 树的平衡性，这损失了一定的更新性能。红黑树是二叉树的衍进，其包含红黑两种颜色的树节点，通过定义根节点到叶节点链路上的红黑节点的数量和顺序规则，来保证树的相对平衡。比起 AVL 树，红黑树减少了更新时的节点调整频率，权衡了更新和查询过程的时间开销。AVL 树和红黑树都属于二叉树结构，节点最多包含两个子节点，故在大数据量情况下，树高较高。B 树也是一种平衡

万方数据

树，B 树节点中包含多个子节点，可以看成是二叉树的衍进，在大数据量的情况下可以保证较低的树高，在磁盘存储中，此特性可以减少搜索过程中随机 IO 的次数，因此 B 树广泛应用于磁盘存储。B+树是 B 树的扩展，B+树在叶子节点中保存全部的“Key”值，内部节点的“Key”值由各子节点“Key”值的最小值构成。另外，B+树将所有的叶子节点用指针相连，这使得 B+树能很好地支持范围查询。B*树是 B+树的扩展，B*树同层的内节点也通指针相连，这使得节点分裂时，能更好地利用兄弟节点的空间。T 树也是一种树形索引结构，其结合了 B 树和 AVL 树的特点，广泛地应用于内存数据库。Lu H 等人对比了 B+树和 T 树[28]，指出在高并发条件下，B+树的查询性能要优与 T 树，原因是 T 树在更新时，会涉及到大量的节点更新，需要大量的加锁操作。Yu C 等人提出 FB+树[29,30]，对 B+树进行了改造，使其更适用于内存场景。FB+树的每个节点都明确标识了子节点“Key”值范围，这可以在查询过程中减少无用的搜索开销。CSB+ tree[31] 也是一种更适用于内存场景的 B+树变种，其通过减少节点中的指针数量来提高每个节点中包含的“Key”值数量，使其能更好地利用缓存行来提高查询效率。Rao J 等人提出一种适用于 GPU 场景的 B+树结构：Harmonia，其利用 GPU 的计算能力强的特点，能够提供高吞吐的查询服务[32]。R 树是用于空间检索的平衡树，将 B+树的思想拓展到多维空间中，利用 MBR 技术切分空间范围，将各个区域组合，形成 R 树索引结构。

（3） 文档索引

倒排索引是一种基于词的索引结构，广泛应用于文档检索领域。针对一个词，倒排索引将包含该词的所有文档连接在一起。查询时，可以针对单词快速检索到包含该词的所有文档。

大数据量下，索引的构建效率至关重要。对此，研究者提出了加快索引构建的方法。研究者[33,34]提出了基于排序的 B+树批量构建法，该方法首先对已有数据根据“Key”值排序，然后遍历数据，自底向上的逐层构建 B+树。为了保证后续的更新效率，在构建过程中为每个节点内部预留空间，以减少更新时的节点分裂开销。Lo M [35]等人提出了基于抽样的 B+树批量构建法，该方法根据数据样本将数据划分为多个范围区间，根据区间生成对应的索引结构，再将数据插入到对应的区间。由于抽样具有一定的随机性，该方法会导致 B+树各节点不均匀。Ngu HCV 等人[36]提出了一种基于 Hadoop 的高效 B+树构建方法，该方法首先切分原始数据，通过一个 MapReduce 任务对每个数据分片构建一棵 B+树，当所有分片的 B+树构建完成后，再利用一个 MapReduce 任务合并所有的分片 B+树，从而生成一棵完整的 B+树。该方法通过并行构建的方式有效地提高海量数据下的 B+树构建性能，但不适用于小数据量或实时性要求较高的场景。Cai r 等人提出了一种基于模版的高性能 B+树构建法[37,38]，该方法对数据进行分段，第一段的数据采用普通的方法构建 B+树索引，后面分段的数据均使用前面分段 B+树的

万方数据

骨架作为模版，直接插入到对应的叶节点。该方法只适用于数据中“Key”值分布较为均匀的场景，不具备通用性。

2.3 分布式存储

大数据场景下，单台机器无法存储海量数据。近年来，不断涌现出多种分布式存储系统来解决海量数据的存储问题。GFS[39]是由谷歌提出的分布式文件系统，其采用主从架构，从节点以块的形式对数据文件进行切分，每个文件默认以三副本的方式存储在多台从节点中，以保证其高可用性。数据写入时，通过版本信息和校验和信息保证写入数据在多副本间的一致性。GFS 的主节点维护全局元信息，包括块信息、文件目录信息等，主节点对所有的文件进行访问控制，以保证在并发情况下保持文件的一致性。GFS 在写入时需要写入多个副本，写入的效率低，适用于离线存储，不适用于实时性要求高的存储场景。为此，谷歌基于 GFS 设计出分布式 Key-Value 存储系统 Bigtable[40]，其底层借鉴了 LSM-tree[41] （Log-Structured Merge-Tree）思想，能够支持高吞吐的实时写入。LSM-tree 思想最早由 Rosenblu m 等人[42]在研究日志结构文件系统时提出。他们将整个磁盘看作一个日志文件，往日志中存放永久性数据时，每次都添加到日志的末尾，这使得文件系统的大多数写操作都是顺序性的。由于磁盘的顺序写入效率高，该方法可以有效地提高磁盘带宽利用率。O’Neil P 等人受到这种思想的启发，结合 B+树结构，提出了一种延迟更新、批量写入硬盘的数据结构 LSM-tree。LSM-tree 将近期的数据更新保存在内存索引结构中，当内存中的索引结构达到设定的大小限制后，便会将其批量的写入磁盘。查找过程中，首先查找内存索引结构，若没找到，则按照顺序依次查找磁盘中的索引结构。LSM-tree 大幅度提升了数据写的入速度，但牺牲了一定的读取性能，其通过定期合并磁盘中的索引结构优化读取性能。LSM-tree 有效地结合了内存和磁盘中的索引结构，在读和写两方面寻找到一个平衡点，适用于具有大量高速写操作的应用场景。

大数据场景下，结合分布式存储系统，涌现出了多种分布式处理框架。MapReduce[43] 是由谷歌提出的分布式处理框架，由 Map 和 Reduce 两大环节构成。Map 阶段是相互独立的，可以分发到各个节点以实现高度的并行化，Reduce 阶段则依赖于 Map 阶段，对 Map 阶段的结果做汇总处理。Map 端的处理结果需要存储在本地磁盘供 Reduce 端使用，存在大量的磁盘读写，故 MapReduce 的整体处理性能有限，仅适用于大规模数据的批量离线处理。谷歌提出的 Dremel[44]是一款交互式的分布式大数据分析系统，采用 MPP 架构，数据处理效率和 MapReduce 相比有了显著地提升，但在超大数据量的查询下，其稳定性和扩展能力不如 MapReduce 计算框架。

万方数据

随着计算机硬件的高速发展，内存的存储容量有了很大的提升，内存的读写速度远快于磁盘等存储介质。分布式场景下，有效地利用内存能够提高数据的处理、存储和查询效率[45,46]。Redis[47]是一款高性能的 Key-Value 内存数据库，其提供了丰富的数据结构以支持不同应用场景下的数据存储，并能支持多节点的分布式存储。由于内存具有易失性，Redis 提出了 AOF 和 RDB 两种数据持久化方法，保障数据可靠性。Teahyon[48]是一款具有高容错性的分布式内存文件系统，通过在内存中缓存文件，来提供低延迟的数据处理能力。VoltDB[49] 是一款分布式内存数据库，在并发情况下，其大量使用乐观锁来提高并发控制性能。Spark 是一个开源的分布式计算框架，其在内存中抽象出分布式弹性数据集 RDD[50]，更多的利用内存进行大规模数据处理，并利用 RDD 的不可变性和 RDD 间的依赖关系高效容错。和 MapReduce 处理框架相比，Spark 减少了磁盘读写开销，从而大幅度提升了处理性能。

海量的数据规模下，通过设计合理的索引结构，可以提高数据的获取效率。海量数据规模下，索引的规模同样巨大，索引的更新、存储和查询都面临着巨大的挑战。为此，研究者对索引按照一定的组织方式分片，将索引分布在不同的存储节点中，形成分布式索引结构。针对不同的应用场景，分布式索引的组织形式也有很大的不同，大致可分为以下几类。

（1） 基于 P2P 架构的分布式索引

分布式 Key-Value 数据库 Cassandra 是典型的 P2P 架构，其采用一致性 Hash 算法[51]对数据进行分片与索引，并使用 gossip 协议[52]来保证数据的一致性。海量数据下，B 树难以维护在单节点中，有研究者提出了分布式 B 树[53,54]，将 B 树的各个节点分布在不同的机器节点上。为了提高查询效率，所有的 B 树节点都缓存在内存中。然而，分布式 B 树存在一些明显的问题。首先，分布式 B 树的并发能力差，一次更新操作会对很多机器上的 B 树节点加锁。此外，分布式 B 树的更新代价高，B 树的每个节点大小固定，更新时为了保持平衡，会涉及节点的分裂和移动，导致多个节点进行跨网络的协调更新，开销巨大。研究者提出了一些方法来降低分布式 B 树更新开销，有研究者提出了延迟更新策略，将多次更新进行合并来减少更新的次数，但其会损失一定的一致性。Huang B[55] 通过记录分裂日志和动态调整节点大小的方法来减少分裂频率，提高分布式 B 树的性能。

（2） 基于主从架构的分布式索引

CG-Index[56]是基于亚马逊云计算平台的一个二级索引结构，其基本思想是在集群中的每个节点上维护一个局部 B+树，通过发布部分局部 B+树节点来构建全局的 CG-Index。查询时，可通过 CG-Index 定位到对应范围的局部 B+树节点。Li X 等人[57]提出了一种基于一致性 Hash 和 B+树的分布式索引结构，其将每一个数据块根据一致性 Hash 算法分发到各个节点上，再对每个数据块构建 B+树索

万方数据

引，从而实现双层的分布式索引。此方法在大数据块的情况下，一致性 Hash 不能做到精确的负载均衡，更没有考虑数据热点问题。在依据范围切分的情况下，一致性Hash 也无法根据某个范围定位到对应的B+树索引，存在着局限性。Melnik S 等人[58]提出了一种面向 WEB 站点的分布式全文索引，其使用倒排索引作为基本的索引类型。索引架构中由分发服务器、索引构建服务器和查询服务器构成，分发服务器负责 WEB 站点的分发，索引构建服务器负责每一部分倒排索引分片的构建，查询服务器负责每一部分倒排索引的查询请求，查询时，会将请求路由到对应的查询服务器。此外，一个查询节点维护全局索引的统计信息，以便于更精确的定位到相应的分片索引。Nguyen TT 等人[59]提出了一种基于 Key-Value 存储的 B+树索引，用于解决大集合的存储。该方法将大集合分成多个小集合，上层利用 B+树索引维护每个小集合的元信息。当小集合的数据量超过阈值后，将分裂成两个小集合，并更新上层 B+树索引。查找时需根据“Key”值和上层 B+树索引定位到对应的小集合。然而，在每个小集合内部只是一个有序的数组结构，在更新时，需要读取完整的小集合数据，更新性能较差。

（3） 基于 Hadoop 的索引结构

当前有很多基于 Hadoop 生态的数据应用，研究人员基于 Hadoop 提出了多种索引结构以加快数据处理效率。何龙等人[60]提出了一种基于 HDFS 的多层索引，实现了 HDFS 中 Split 层和 Split 内部的双层索引。Split 内部索引用于快速过滤无效数据，减少磁盘的 IO 开销。Split 层索引则根据一些 Split 层面的统计信息，可过滤掉整个 Split，从而减少 MapReduce 运行阶段的无效任务数，提高任务处理性能。Richter S 等人提出一种基于 Hadoop 的高效索引方法 HAIL[61]，HAIL 中包括静态索引和动态索引。静态索引在数据上传时对数据建立索引，并通过逻辑复制的方法，对数据构建多种聚集索引。动态索引则可以根据查询等应用需求，动态地添加索引。通过 HAIL 索引方式能有效提高 Hadoop 任务执行效率。这类索引旨在提高离线处理效率。

以上所提的分布式索引方案中，没有结合数据流流速大等特点对索引组织及构建方式进行合理优化，无法有效地应用于数据流场景。除此之外，上文提到的 LSM-tree 结构，若将 B+树持久化在分布式文件系统中，也可以看成一种分布式索引结构。数据流应用中，数据流写入后往往不存在后续的更新操作，依赖 LSM-tree 结构存储数据流，其缓存的数量非常有效，且查询时查询节点需顺序遍历磁盘中的索引结构，性能较低。

海量数据规模下，数据存储格式也直接影响着存储和查询性能。传统的关系型数据库中，往往采用行式存储，将一行记录的所有属性存储在一起。查询时，通过读取完整行进行条件过滤和部分结果的返回。列式存储则将同一属性的一列数据存储在一起[62]，相同列的数据字段类型统一，可以更好的对其压缩存储，在大数据场景下优势明显。查询时，只需读取对应列数据，能减少大量

万方数据

无用的数据读取。行式存储通过对某些字段添加索引以提高查询效率，列式存储则会对连续的数据进行分段，索引的最小粒度是一个分段的数据，分段内部数据基于关键字有序，可以通过二分查找等方法快速查找[63]。RCFile[64]是基于Hadoop 的列式存储格式，其结合了 MapReduce 任务的特性，首先对数据进行水平划分，划分后的每块数据内部采用列式存储格式。这既保证了同一行的数据在同一个节点处理，也可以利用列式存储的特性压缩数据，查找时也能过滤不必要的数据读取。ORCFile[65]在 RCFile 的基础上，对数据文件添加简单的索引结构和统计信息，使得查找时进一步过滤不必要的读取操作。Floratou A[66]等人提出 CIF 列式存储格式，在分块文件内部维护一个跳表索引结构，结合惰性加载的思想，避免无用数据的读取开销。Parquet 列式存储是一种支持嵌套结构的存储格式[44]，应用于多种查询引擎，其内部维护特定的索引结构以过滤大量无效的数据读取操作。大数据场景下，数据压缩可以大幅度减少存储开销，但同时也会影响查找和读取的效率，因此，研究者往往在两者之间选取一个折中点。

2.4 本章总结

本章首先综述了数据流在处理、存储和查询等方面的相关工作；接着比较了一些基本索引结构以及加速索引构建的方法；随后着重介绍了分布式场景下数据存储的发展现状，比较了当前不同场景下的分布式索引结构；最后介绍了分布式场景下数据存储格式的衍进。

万方数据

第三章 面向大数据流的分布式索引架构

本文提出了一种适用于大数据流场景的分布式索引：WB-Index。数据流具有实时性、无限性等特点，为了能更好的对其构建索引，本文利用时间窗口机制从时间维度切分数据流。对于每个时间窗口内的流元组，构建基于内容的底层索引。对于各个连续的时间窗口，根据时间窗口的起始时间戳以及对应的底层索引元信息构建基于时间维度的顶层索引，从而形成双层的 WB-Index 索引结构。本文选用 B+树作为基本索引结构，B+树可以同时支持点查询和范围查询，且在大数据量下能保持较小的树高，在内存和硬盘存储介质中都能保证良好的查询性能。集中式的系统架构无法支撑海量数据流的索引构建、存储和查询，对此，WB-Index 采用分布式的系统架构，将流元组存储、索引构建和查询请求分离，从而支持海量数据流的高效存储与查询。本章首先阐述了 WB-Index 索引结构以及 WB-Index 系统架构，然后介绍了 WB-Index 的构建过程和查询过程，详细阐述了中间涉及到的负载均衡和数据缓存等细节。

3.1 WB-Index 索引结构

数据流根据其应用场景的不同往往包含多个数据源，系统中会生成全局唯一的数据源标识对其进行区分。对于不同数据源对应的数据流，其索引构建和存储都是相互独立的，为了便于理解，本文以一个数据源为例进行阐述。

数据流具有实时性、无限性等特点，为了能更好的对其构建索引，本文利用时间窗口从时间维度切分数据流。一个时间窗口指定了当前处理数据流的起始和结束位置。对于第 i 个时间窗口 W_i，其起始时间戳为 t_i，对应的时间窗口可以用$<t_i,W_i>$表示。一个时间窗口中的数据流可以表示为：$W_i=\{<t_{im}, d_{im}> \mid m=1, 2, ..., |n|\}$，其中，$d_{im}$ 表示一条完整的数据流记录，称之为流元组，n 表示时间窗口 W_i 中的流元组数量。d_{im} 可以抽象的表示成<K,V>，K 表示流元组的“Key”值，V 表示流元组的“Value”值。

针对每个时间窗口 W_i，利用流元组 d_{im} 的“Key”值，构建对应的索引结构 WI_i，称为底层索引。底层索引选用 B+树索引，B+树索引能支持窗口内流元组的等值查询和范围查询。底层索引中还包括根据流元组“Key”值构建的布隆过滤器，以及根据窗口内流元组“Key”值的最大值和最小值构建的统计信息。这些结构可以看成底层索引的辅助索引，其只占用少量的存储空间，在底层索引持

万方数据

久化后，辅助索引可以高效地过滤一些无效的查询请求，从而减少无用的磁盘 IO 以提高查询效率。

针对各连续的时间窗口{W_1,W_2, W_3 …. W_i}，本文以 W_i 对应的起始时间戳 t_i 作为“Key”值，以底层索引结构 WI_i 的引用作为“Value”值，构建对应索引结构 GI，称为顶层索引。顶层索引同样选用 B+树索引结构，其能够提供时间层面的等值查询和范围查询。

将底层索引和顶层索引结合后，便形成了双层的索引结构。数据流具有无限性，海量数据下，双层索引结构难以通过集中式架构维护。故双层索引结构会被切分，由多个节点共同维护，从而形成分布式索引结构：WB-Index。WB-Index 索引结构如图 3-1 所示。

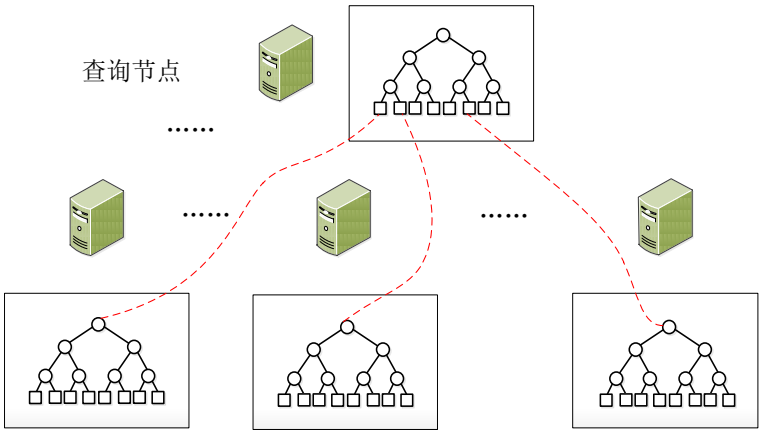

图 3-1 WB-Index 索引结构

Figure 3-1. The structure of WB-Index

WB-Index 索引结构中，顶层索引包含了各时间窗口的起始时间戳以及对应底层索引的元信息，数据量较小，故将其维护在一个主机节点中。底层索引包含了所有流元组的“Key”值，相比顶层索引，数据量较大，且不同时间窗口对应的底层索引不存在依赖关系。因此，WB-Index 以一个时间窗口对应的底层索引作为最小单位，将底层索引分散在多个主机节点上。顶层索引的“Value”值包含对应底层索引所在节点的 IP 以及对应时间窗口的起始时间戳 T，可以表示为（IP,T）。WB-Index 属于主从结构的分布式索引，有效地缓解了索引维护压力。查询时，先通过顶层索引定位到对应的底层索引，由于底层索引分布在多个主机节点上，因此可以并行地查找底层索引。

WB-Index 结构中，底层索引采用了数据和索引分离的策略。数据流应用场景中，需要实时响应对较新数据流的查询请求。WB-Index 通过在内存中维护较新时间窗口对应的索引来提高查询效率。然而，一个窗口对应的流元组数据量大，若将数据与索引结合，会显著增加底层索引大小，使得内存中只能维护少数时间窗口的底层索引。查询涉及到稍早些的数据流时，便会从磁盘中查找，查询效率

万方数据

也会下降。采取索引和数据分离策略后，索引结构中只包含了流元组的“Key”值以及纪录的偏移值，这大幅度减少了存储开销，使得大量的索引结构可以维护在内存中。虽然最后获取对应流元组的过程也要涉及到磁盘 IO，但由于底层 B+树构建完成后，时间窗口内的流元组根据“Key”值有序，即便存储在磁盘中，也可以通过偏移量直接读取对应流元组，读取过程只涉及到一次 IO，效率较高。另外，考虑到查询往往存在热点性，WB-Index 系统中设计了多类流元组缓存来提高流元组的提取效率，流元组的缓存策略将在下文详细阐述。总之，WB-Index 中的底层索引将索引和数据分离，使得系统能更好地缓存索引结构，从而提高查询效率。

3.2 WB-Index 系统架构

WB-Index 系统架构中，将流元组存储、索引构建和查询请求分离，以支持海量数据流的高效存储与查询。

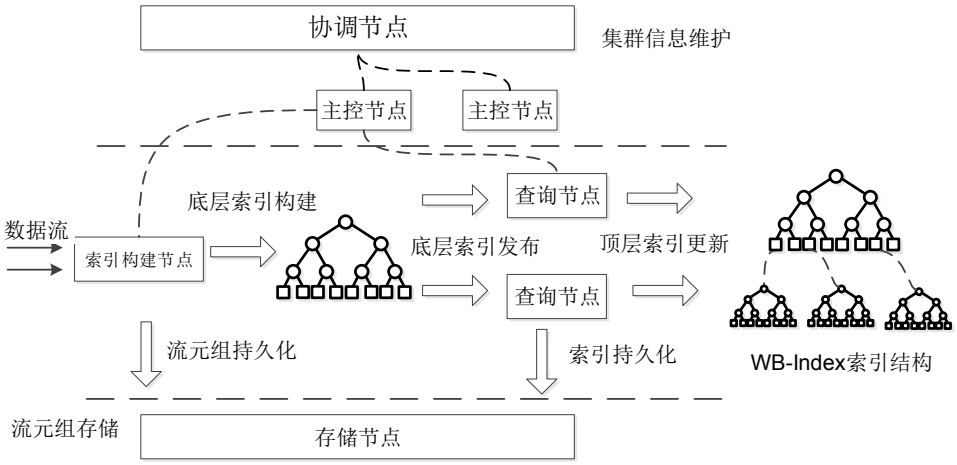

图 3-2 WB-Index 系统架构

Figure 3-2. The system architecture of WB-Index

图 3-2 为 WB-Index 系统架构图，集群由一台台普通主机节点构成。为了保证性能，所有节点部署在同一机房的内网环境，节点之间网络带宽大，数据交换时延小。节点间通信协议采用基于 TCP 的自定义协议，协议格式为：固定头部+变长消息体。传输格式的大小直接决定了传输的性能，系统实现中使用由谷歌提出的 Protocol Buffers 数据格式，其产出的二进制编码格式不仅数据体积小，而且解析速度快，适用于分布式场景下的数据传输。

WB-Index 集群中存在 5 种节点类型，分别是索引构建节点、查询节点、存储节点、协调节点和主控节点。各类节点的职责为：

1. 索引构建节点负责数据流的接收以及底层索引的构建。底层索引构建完成后，会将其发布到查询节点。

万方数据

2. 查询节点负责维护双层索引结构以及响应查询请求。
3. 存储节点负责存储流元组以及索引结构。数据流具有无限性，系统实现中使用分布式文件系统存储数据流和索引结构。
4. 协调节点负责各节点的注册，维护各节点物理信息以及索引结构的元信息。每台节点与协调节点保持心跳，通过协调节点维护的信息可以获取集群各节点的当前状态。在索引构建和流元组查询等过程中，都需要从协调节点获取所需的元信息，但每次获取都会产生一定的时间开销，影响性能。对此，集群中的各节点在本地缓存所需元信息，并监听协调节点中对应的元信息，当协调节点中对应的元信息被修改后，各节点会同步并更新本地缓存中的元信息，这种方式可以有效地避免每次都从协调节点获取元信息所带来的时间开销。系统实现中，使用 ZooKeeper[67]作为协调服务，其内部维护着类似于文件系统的树形结构，每个节点都可以存储少量的信息。协调节点维护了集群中的所有状态信息，需要保证高可用性。ZooKeeper 通过多个副本保证了高可用性，并结合 ZAB 协议保证一致性。
5. 主控节点相当于是集群的大脑，主要负责处理决策请求，如负载均衡请求等。主控节点采用无状态模式，将所有的状态信息存储在协调节点上，因此集群中可以存在多个主控节点以保持高可用性。

3.3 分布式索引构建

索引构建过程中，数据流首先流入到索引构建节点，索引构建节点根据预先定义的时间窗口长度划分数据流，根据时间窗口 W_i 内的流元组构建相应的底层索引。索引构建节点在索引构建过程中处于高负载状态，若将底层索引保留在索引构建节点并提供查询服务，会同时影响索引的构建和查询效率。因此在底层索引构建完成后，需要将其发布到查询节点。查询节点获取到 Wi 窗口对应的底层索引后，根据底层索引的元信息和时间窗口的起始时间戳更新顶层索引，从而完成索引构建。时间窗口对应的流元组首先会缓存在构建节点中，构建节点会有专门的线程负责将时间窗口 W_i 对应的流元组持久化到存储节点。底层索引中包含了窗口所有流元组的“Key”值，需要占用一定的存储空间，故在发布索引时需考虑查询节点的负载均衡。

3.3.1 查询节点负载均衡

索引构建节点在构建完底层索引后，会将底层索引发布到查询节点，查询节点维护相应的底层索引，处理对应时间窗口的查询请求。集群中存在多个查询节点，选取合适的查询节点发布索引是影响 WB-Index 构建和查询效率的关键因素。

万方数据

由于数据流流速具有阶段差异性，不同时间窗口的流元组数量相差较大，对应底层索引的大小也不尽相同，若直接采用轮询、随机等负载均衡策略会导致底层索引分布不均。其次，数据往往存在热点性，即有大量的查询请求落在某些时间窗口内的流元组上，若所有热点数据对应的底层索引都存储在同一个查询节点，会使得该节点负载过高，影响查询效率，也不利于集群的稳定。另外，不同查询节点的主机性能也存在一定的差异，负载均衡策略需要综合考虑节点的负载量和处理能力。本文根据查询节点的内存、CPU、底层索引查询热点等信息，提出了动态更新权重的加权轮询负载均衡方法，使得各查询节点负载均衡，从而提高索引的构建和查询效率。

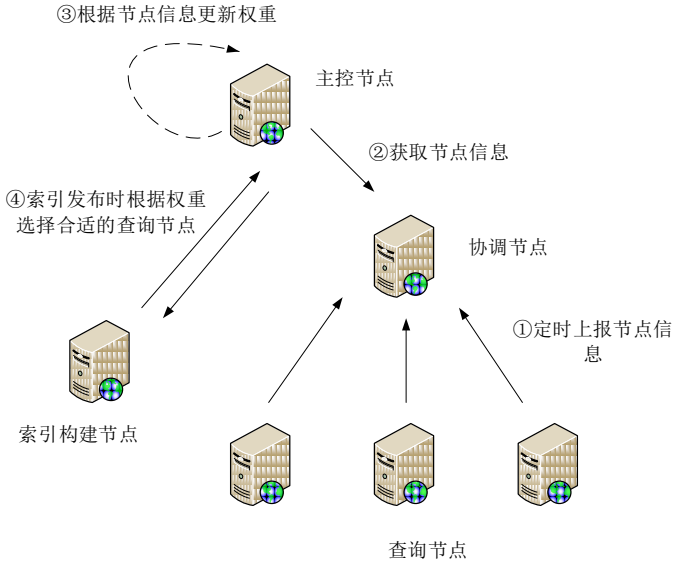

图 3-3　负载均衡流程示意图

Figure 3-3. The process of load balancing

负载均衡流程如图 3-3 所示，主要由信息采集、权重更新、节点选取 3 大步骤组成。负载均衡流程中的相关参数如表 3-1 所述。

表 3-1　负载均衡参数表

Table 3-1. The parameters of load balancing

参数	描述
N_s	查询节点维护的底层索引在过去 1 分钟内的查询次数
P_c	查询节点过去 1 分钟内的 CPU 平均负载
N_m	查询节点当前可用内存量

负载均衡过程首先要获取各查询节点的 N_s、P_c、N_m。每个查询节点周期性的向协调节点上报相应参数，由协调节点维护所有的节点参数。其中，上传周期必须适中，过小的上传周期虽然能够更加精准地获取节点当前的参数，但频繁地上传会增加额外的系统开销；过大的更新周期则会导致协调节点上维护的

万方数据

参数信息过于陈旧，影响负载均衡的准确性。系统实现中，上传周期设定为 10 秒。

主控节点会监听协调节点上维护的负载均衡参数信息，每当节点参数信息更新后，就会触发计算各查询节点权重。负载均衡中涉及的三种参数的取值范围和单位均不一致，因此计算过程中首先对各参数进行标准化处理，标准化方式为：

$$x_{new} = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{3-1}$$

其中，N_s、P_c 是负向参数，针对这些参数，标准化方式为：

$$x_{new} = \frac{|x - \max(x)|}{\max(x) - \min(x)} \tag{3-2}$$

参数标准化后，根据预先定义的各参数权重值 k_i，计算出各查询节点最终的权重值 $C(N_i)$:

$$C(N_i) = \begin{pmatrix} k_1 & k_2 & k_3 \end{pmatrix} \begin{pmatrix} N_s \\ P_c \\ N_m \end{pmatrix} \qquad (k_1 + k_2 + k_3 = 10) \tag{3-3}$$

其中参数的权重值要根据应用场景设定。对于查询节点内存限制较大的场景，可以提高 N_m 参数权重；对于查询频率高的场景，可以提高 P_c、N_s 的参数权重。计算出各查询节点的权重后，本文使用加权轮询策略选取合适的查询节点，保证查询节点负载均衡，如算法 1 所述。

算法 1. 加权轮询算法

输入：各节点权重列表 $C(N_i)$，总权重 S，当前权重列表 $C_{curr}(N_i)$

输出：选取的查询节点 i

1. 若当前的权重列表 $C_{curr}(N_i)$为空，初始化 $C_{curr}(N_i)$：$C_{curr}(N_i) \leftarrow C(N_i)$，初始化总权重 S=SUM($C(N_i)$)。
2. 选择 $C_{curr}(N_i)$中最大权重值对应的节点 i 作为返回值：t←i。
3. 更新当前权重列表 $C_{curr}(N_i)$：
   i. 针对所选的节点 t，$C_{curr}(N_t) \leftarrow C_{curr}(N_t) - S$
   ii. 针对所有的节点 i，$C_{curr}(N_i) \leftarrow C_{curr}(N_i) + C(N_i)$
4. 返回所选节点 t。

该加权轮询算法能够有效地避免大量底层索引连续的分发到高权重节点上，在保证精确度的前提下使得负载均衡过程更加平滑，有利于集群的稳定。每当协调节点中各节点权重更新后，该方法会重新初始化，以达到精确的负载均衡效果。

3.3.2 顶层索引多副本

WB-Index 是一种双层索引结构，查询时，需要根据顶层索引定位到对应的底层索引。若集群中只存在一个顶层索引，所有的查询都会首先请求顶层索引对应的查询节点，这使得查询性能存在瓶颈。顶层索引基于各时间窗口构建，具有

万方数据

量级小且更新频率低的特点，本文结合该特点，在多个查询节点上维护顶层索引的副本，并将所有顶层索引的元信息保存在协调节点中。查询节点接收到查询请求后，随机选取一个顶层索引进行查询，从而将顶层索引查询负载分摊到多个查询节点上，有效地提高了查询效率。

由于顶层索引存在多个副本，在底层索引构建完成后，需要更新所有对应的顶层索引。更新过程中，若有一个副本的顶层索引更新失败，索引构建节点会多次尝试重试，直到顶层索引更新成功。若超过重试次数后依旧更新失败，则会在协调节点中删除该顶层索引副本的元信息，使该顶层索引副本不可用，避免查询时由于顶层索引不一致而导致全局查询结果不一致。

3.4 分布式查询

数据流查询时，需要根据窗口时间（范围）、流元组“Key”值（范围）等查询条件进行查询。集群中所有的查询节点都可以响应查询请求，查询节点首先找到对应的顶层索引，根据查询条件中的窗口时间（范围）定位到底层索引，再并行的搜索对应的底层索引，最终整合并返回查询结果。分布式查询可以概括为顶层索引查找、底层索引查找、流元组获取和数据汇总这四个阶段。WB-Index 中的底层索引将索引和流元组分离，故查询完底层索引后，需要从存储节点中读取对应的流元组，存在一定的时间开销。因此，本文提出多类流元组缓存方法来提高查询效率。

3.4.1 流元组缓存方案

表 3-2　三类流元组缓存

Table 3-2. Three types of tuple cache

类别	描述
第一类缓存	基于查询结果
第二类缓存	基于查询区间内的流元组
第三类缓存	基于较新时间窗口内的流元组

WB-Index 的查询过程中会涉及到多节点间的数据传输，减少数据传输量可以大幅度提高查询效率。查询往往具有热点性，会有大量的查询请求落在相同的流元组上。数据流应用中，较新的数据流元组往往会是查询热点。本文结合数据流及其查询的特点，根据分布式查询执行到的不同阶段，为流元组分配了3类缓存，如表 3-2 所示。其中第一类是基于查询结果的缓存；第二类是基于查询区间内流元组的缓存；第三类是基于较新时间窗口流元组的缓存。通过这三类缓存的协同作用可显著提高 WB-Index 的查询效率。

万方数据

第一类缓存基于查询结果。本文利用时间窗口划分数据流，独立存储每个时间窗口内的流元组。对于一个时间窗口内的流元组，后续不存在更新操作，因此对于查询条件相同且限定时间窗口范围相同的查询语句，返回的查询结果必然是相同的，所以可以缓存查询结果，来避免不必要的重复查询。对于一个查询请求，查询条件中包含数据流标识、时间窗口范围、键值范围、查询规则等参数。第一类缓存使用哈希结构维护查询条件和查询结果的映射关系，将查询参数组合作为哈希结构的“Key”值，对应的查询结果作为“Value”值。为了减少内存开销，查询结果以字节形式存储。

第一类缓存隶属于查询节点，每个查询节点都单独维护一份缓存。每个查询请求会根据查询条件的 Hash 值落到对应的查询节点上，因此，相同查询条件的查询请求会落在相同的查询节点上，这种查询机制保证了第一类缓存的有效性。当一个查询请求到达后，根据查询条件查找第一类缓存，若命中第一类缓存，则直接响应结果；反之，则通过分布式查询获取结果，并将结果添加至缓存中。缓存受内存大小限制，采取 LRU 策略更新缓存。对于返回结果数据量较大的查询请求，其返回结果会占据大量的缓存空间，如果相同查询条件的查询频率不高，会导致大量的缓存更新，反而影响查询性能。故对于此类查询，可以在查询条件中设置不对查询结果进行缓存，从而保证缓存的效率。

第二类缓存基于查询区间内的流元组。流元组查询往往存在热点性，对于一些热点流元组，会经常被查询命中。针对热点数据的查询请求，所带的查询条件往往是不相同的，若仅利用第一类缓存，无法很好地命中缓存，每次查询时依然需要跨节点的获取对应的流元组，存在较大的时间开销。因此，本文设计出第二类缓存，第二类缓存的核心思想是缓存查询落到的流元组，下次查询落到相同的流元组将直接从缓存中读取，无需跨节点的获取流元组。然而若只缓存查询落到的流元组，会存在缓存粒度过细的问题，查询时命中缓存的机率低，也会导致频繁的缓存更新，浪费节点资源。故第二类缓存的最小缓存粒度为底层索引叶子节点子元组数量，即对于一个查询命中的流元组，会缓存其对应底层索引叶子节点中的所有流元组。

第二类缓存隶属于底层索引，底层索引对应的查询节点维护相应缓存。第二类缓存使用哈希结构维护底层索引叶节点和缓存流元组的映射关系，将数据流标识、所在时间窗口起始时间戳和叶节点编号组合作为哈希结构的“Key”值，对应叶子节点内的流元组作为“Value”值。其中流元组以数组的形式缓存，在命中二级缓存时，可直接通过偏移提取出对应的流元组，保证缓存提取效率。查询时，通过底层索引定位到数据所在的叶节点，并查找相应的二级缓存，若命中缓存，则直接提取出对应的流元组。反之，则从存储节点获取相应流元组，并添加至第二类缓存。第二类缓存受内存大小限制，采用 LRU 策略对其进行更新。

万方数据

第三类缓存基于较新时间窗口内的流元组。数据流应用场景中，新数据往往是热点数据，需要保证较好的查询性能。本文通过缓存新窗口内的所有流元组来提高新数据的查询效率。底层索引构建完成后，对应时间窗口的流元组会缓存在索引构建节点，当查询落到对应的流元组时，直接从构建节点的内存中获取流元组，相比于从存储节点的磁盘中获取流元组，可以减少时间开销。第三类缓存隶属于该数据源对应的索引构建节点，使用哈希索引结构维护窗口和流元组的映射关系，将数据源标识和时间窗口起始时间戳组合作为哈希索引的“Key”值，对应时间窗口内的流元组作为“Value”值。索引构建节点只提供数据获取服务，不涉及到数据的复杂操作，故将流元组以字节形式缓存，这可以减少内存存储开销，查询时也能节省序列化时间。其中，流元组在内存中的存储格式与持久化的格式相同，这一部分将在第五章详细阐述。由于一个时间窗口内的流元组数量较大，故第三类缓存只能缓存少数时间窗口的流元组。当内存不足时，采用 FIFO 的更新策略，移除缓存中最旧时间窗口对应的流元组。

流元组缓存方案综合了数据流和 WB-Index 的特点，在不同节点缓存不同粒度的流元组，最大限度地提升缓存命中率，从而提高查询效率。

3.4.2 查询流程

WB-Index 集群中，所有的查询节点都能提供查询服务。查询客户端首先根据查询条件计算出 Hash 值，再对集群中的查询节点数量取模，来选取相应的查询节点并发送查询请求。这种方式能使得查询请求均匀的落在各查询节点上，保证了查询节点的负载均衡。查询节点接收到查询请求后，根据顶层索引将查询拆解，并将子查询分发到对应底层索引所在的查询节点，最终由查询节点汇总结果并返回。本节将详细介绍分布式查询过程。针对数据源 S1，若给定查询的时间范围为 $T[t_l,t_r]$，流元组的“Key”范围为 $K[k_l,k_r]$，获取对应流元组的分布式查询过程如算法 2 所述。

算法 2. 分布式查询

输入：查询的时间范围 $T[t_l,t_r]$，流元组的“Key”值范围 $K[k_l,k_r]$，数据源 S1

输出：流元组结果集 tuple[]

1. 查询客户端根据查询参数 t_l、t_r、k_l、k_r、S1 计算出 Hash 值，并从协调节点获取所有查询节点信息，利用 Hash 值对查询节点数量取模，根据结果将查询请求发送到对应的查询节点。
2. 查询节点收到请求后，根据查询条件搜索第一类缓存，若命中缓存，则直接返回缓存中的结果，完成整个查询流程；否则，继续后续查询。
3. 查询节点根据节点缓存的顶层索引元信息，采用随机负载均衡策略，随机选取一个 S1 数据源对应的顶层索引，根据其所在节点的 IP，远程调用对应查询节点的 getBaseIndex($T[t_l,t_r]$)方法返回查询涉及到的底层索引元信息 base[]。
4. 查询节点根据元信息 base[]，调用 getResult (base[]，$K[k_l,k_r]$)方法返回查询对应的流元组结果集 tuple[]。

万方数据

5. 若查询请求中未禁用查询缓存，则根据查询条件以及查询结果更新本节点的第一类缓存。

过程：getBaseIndex(T[t_l,t_r]) //获取底层索引元信息

1. 处理查询边界。获取顶层索引对应的当前最小和最大时间窗口时间戳 t_{min} 和 t_{max}，将查询条件[t_l,t_r]与[t_{min}, t_{max}]取交，若交集为空，表示不存在对应的流元组，直接返回；否则，用交集更新查询条件。
2. 顶层索引中的“Key”值为时间窗口起始时间戳 t_{start}，B+树查询过程中，根据查询节点的左边界 t_l 定位到 B+树的叶子节点，依次遍历叶节点：
   2.1. 提取出叶节点的“Value”中所包含的时间窗口时长 t_{length}。
   2.2. 若[t_{start}, t_{start}+t_{length}]与查询条件[t_l,t_r]存在交集，则表明对应的底层索引符合查询条件，提取“Value”中包含的底层索引元信息，添加至底层索引元信息 base[]中，并查找下一个叶节点。
   2.3. 若[t_{start}, t_{start}+t_{length}]与查询条件[t_l,t_r]不存在交集，则表明已查找完毕，结束查询，并返回底层索引元信息 base[]。

过程：getResult (base[]，K[k_l,k_r]) //返回流元组

1. 依次遍历 base[]，对于其中的每一项元信息 $meta_{base}$，提取出 $meta_{base}$ 中的 IP 信息，将相同 IP 的元信息聚合，生成 $meta_{base}$[]，通过聚合的方式可以减少节点间的通信次数。最后根据 IP 并行地远程调用对应查询节点的 getTuple ($meta_{base}$[], K[k_l,k_r])方法获取流元组 tuple[]。
2. 等所有并行查询任务完成后，整合所有返回的流元组，返回最终的查询结果。

过程：getTuple ($meta_{base}$[], K[k_l,k_r]) //返回对应时间窗口的流元组

1. 对于 $meta_{base}$[]中的每一项目 meta，根据其中的时间窗口起始时间戳 t_{start} 查找对应的底层索引：
   1.1. 若底层索引结构在内存中，则找到 B+树根节点引用，按照 B+树的查找步骤，获取符合条件的流元组元信息 $meta_{data}$，其中包括符合查询条件的 B+树叶节点编号以及具体流元组的偏移量。
   1.2. 若底层索引不在内存中，则查找分布式文件系统中对应的索引文件。获取符合条件的流元组元信息 $meta_{data}$。索引文件中的具体查找方法将在第五章详细阐述。
2. 根据 B+树叶节点编号和窗口时间戳查找第二类缓存。若命中缓存，则直接获取对应的流元组；否则，根据流元组元信息 $meta_{data}$ 调用 getTuple($meta_{data}$)方法获取对应流元组，获取流元组的最小粒度为一个叶节点对应的完整流元组。
3. 获取到对应的流元组后，提取所需流元组并更新第二类缓存。
4. 整合并返回查询对应的流元组。

过程：getTuple ($meta_{data}$) //获取具体的流元组

1. 根据 $meta_{data}$ 中的标志位判断对应流元组是否缓存在索引构建节点：
   1.1. 若在，则根据底层索引中的偏移量从对应索引构建节点的内存中获取流元组。
   1.2. 若不在，则根据底层索引中的偏移量从分布式文件系统中获取对应的流元组。

上述分布式查询为获取对应流元组的查询过程。对于聚合、投影等查询模式，先由底层索引所在的查询节点进行预处理，等所有预处理结果返回后，再

万方数据

由接收查询请求的查询节点进行最终处理并返回查询结果。这种方式可以将数据处理过程下推，从而减少节点间的网络 IO 开销。整个分布式查询过程中，将查询任务拆解分发到各个查询节点，使得整个查询过程并行化，有效地提高了查询效率。

3.5 本章总结

本章首先描述了 WB-Index 索引结构。本文利用时间窗口切分数据流，基于时间窗口中的流元组内容构建底层索引，基于各连续时间窗口构建顶层索引。底层索引分布在查询节点中，结合顶层索引形成完整的分布式索引结构。WB-Index 系统架构中，通过多种节点类型将数据流的索引构建、存储和查询分离，从而保证数据流的高效存储与查询。随后阐述了 WB-Index 的构建过程，以及保证查询节点负载均衡的方法。最后阐述了 WB-Index 的查询过程，以及通过流元组缓存提高查询效率的方法。

万方数据

第四章 分布式索引构建方法

第三章介绍了 WB-Index 索引结构。本文利用时间窗口从时间维度切分数据流，依次处理每个时间窗口内的数据流，将实时处理转换为微批量处理，有效地提高吞吐率。针对每个时间窗口内的数据流，索引构建节点根据流元组内容构建一棵底层 B+树索引。由于数据流流速快，传统 B+树构建方法构建效率低，不适用于高速数据流的场景。因此，本文提出了一种基于并行排序的预装载 B+树批量构建法，所提方法构建效率高，能很好地适用于高速数据流场景。针对连续的时间窗口，利用底层索引的元信息和对应时间窗口的起始时间戳构建顶层 B+树索引。连续窗口的起始时间戳具有单调递增的特点，本文提出一种预分配空间的不分裂 B+树更新方法，该方法在损失一定的 B+树平衡性的前提下，有效地提高了空间利用率和更新性能。本章将详细介绍 WB-Index 的构建过程，对构建时间开销进行理论分析，最后通过实验证明构建方法的有效性。

4.1 解决方案

WB-Index 构建过程如图 4-1 所示，图中标记为 W_1、W_2、……、W_k 的矩形代表划分数据流的一个时间窗口，T_w 为窗口时长。针对每个时间窗口 W_k，P1 表示 WB-Index 索引的构建过程，其中包括了底层索引构建、底层索引发布和顶层索引更新三个阶段，图中分别用数字 1、2、3 表示。P1 也表示 WB-Index 的构建时延，“构建时延”表示每个时间窗口流元组缓存完成后，完成 WB-Index 索引构建需要的时间，为提供近实时查询，构建时延越短越好。P2 表示从当前时间窗口的索引构建完成到下一个时间窗口数据流全部到达的时间间隔，称为构建间隔。为了保证 WB-Index 构建的稳定性，构建间隔必须大于 0；否则，数据流流速大于索引构建速度，数据流将会在索引构建节点堆积，随着时间的推移，WB-Index 将不可用。以下各小节将分别介绍底层索引构建、底层索引发布和顶层索引更新这三个阶段。

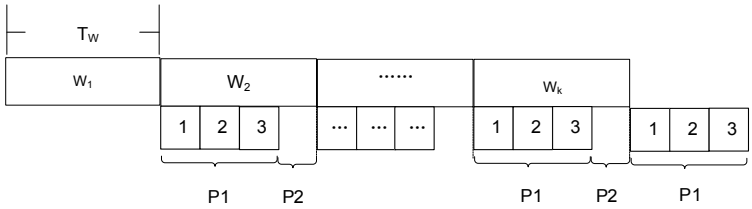

图 4-1 WB-Index 构建过程描述

Figure 4-1. The construction process of WB-Index

万方数据

4.1.1 底层索引构建

底层索引基于时间窗口内的所有流元组，其中包括 B+树索引和辅助索引结构。B+树用于流元组的快速检索，辅助索引用于在底层索引持久化后，过滤一些无效查询来提高查询效率。

辅助索引结构中包括布隆过滤器和统计信息。针对时间窗口内的每个流元组，布隆过滤器的更新只涉及到几次 Hash 计算和位运算，统计信息则只要维护一个时间窗口内流元组“Key”值的最大和最小值。辅助索引的构建过程简单，不存在构建性能问题。因此，辅助索引将在底层索引发布后，由查询节点构建。索引构建节点只负责构建 B+树，这可以有效地减轻节点的负载，也减少了与查询节点间的数据传输开销。

底层索引构建过程中最大的性能瓶颈在于 B+树构建。传统的 B+树插入构建法首先需要搜索 B+树以找到插入点，为了保持 B+树的平衡，插入的过程中会涉及到大量节点的移动、分裂，使得构建性能低下，无法满足数据流场景。传统的 B+树批量装载法没有结合数据流的特性，一次完整的全排序依旧会导致较大的构建时延。本文针对高速数据流，提出基于并行排序的预装载 B+树批量构建法，提高 B+树的构建效率。该方法沿用了批量装载 B+树的思想，首先对时间窗口内的流元组排序，然后根据流元组数量以及预设的 B+树节点大小计算出完整的 B+树结构，接着通过计算来关联父子节点和兄弟节点，最后对 B+树节点赋上“Key”值。在此基础上，本方法结合数据流的特性作出了以下优化：

1. 提高排序效率。为了提高排序效率，本方法对时间窗口作进一步地切分，形成一个个连续的分片。当接收到一个完整分片数据流后，在接收下一分片数据流的同时，并行的对该分片内的流元组进行预排序。对于每个分片内部的预排序，可以采用并行排序的方式提高排序效率。在接收数据流和对分片预排序的同时，使用一个线程并行的对已排序分片进行归并排序，等最后一个分片排序完成后，作最后一次归并使得流元组全局有序。该方法通过将数据流接收、分片排序和分片归并并行，提高全局的排序效率。
2. 将排序和索引构建并行。当时间窗口内的流元组全部到达后，排序过程还需完成最后一个分片的预排序和归并，此时可以并行的开始计算 B+树结构、创建节点以及关联节点关系，本文将此步骤称为 B+树骨架构建。
3. 预装载 B+树节点。数据流排序过程中，最后一次的归并排序会新开辟一个数组空间来存储全局有序的流元组。因此，可以预先申请该数组空间，并根据数组中的偏移预装载 B+树节点，等排序结束后，再根据偏移量赋上最终的“Key”值，从而减少了最后赋值操作的计算过程。

万方数据

优化后的 B+树构建方法如图 4-2 所示，其中编号 1 表示窗口内分片时长，编号 2 表示分片内流元组预排序时长，编号 3 表示对已排序分片归并时长，编号 4 表示 B+树结构参数计算时长，编号 5 表示 B+树节点创建和关联时长，编号 6 表示 B+树预装载时长，编号 7 表示 B+树赋值时长。

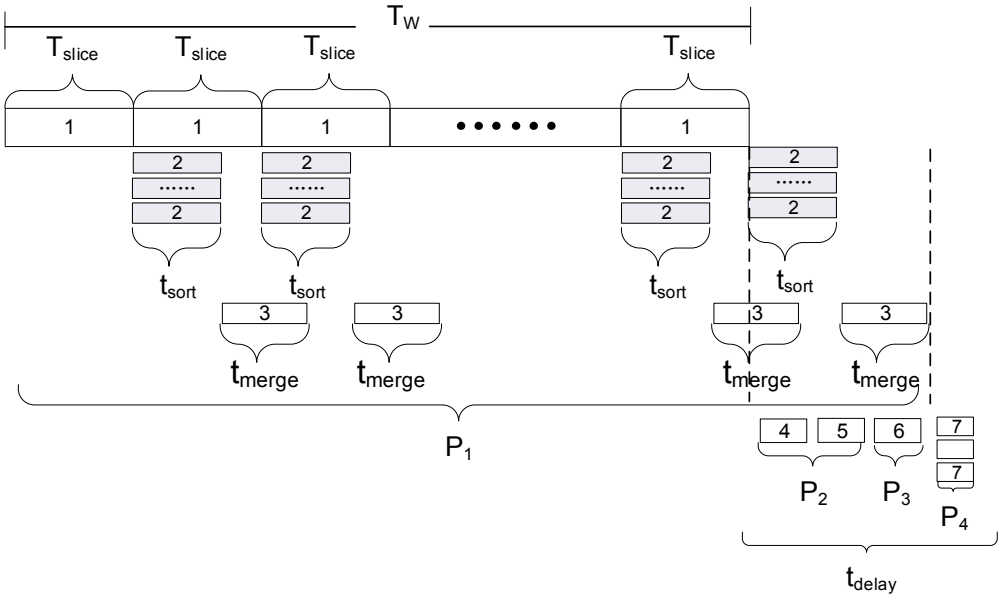

图 4-2 基于并行排序的预装载 B+树批量构建法描述

Figure 4-2. The description of B+ tree construction process

构建过程中涉及到的参数如表 4-1 所示。

表 4-1 B+树构建参数表

Table 4-1. The parameters of B+ tree construction

参数	描述
N_b	节点容量（节点可容纳的最大节点数）
N_w	窗口流元组数
N_h	B+树层数
N_{inner}	B+树内节点数
N_{leaf}	B+树叶节点数
N_m	B+树内节点的子节点数
N_d	B+树叶节点中的（Key, Value）数
N_r	B+树叶节点要增加 1 个（Key, Value）的子节点数
S_{node}	批量申请的节点数组
S_{sort}	时间窗口内排序完成的流元组

本方案的 B+树节点结构如图 4-3 所示，其中额外维护了父子节点和兄弟节点在节点数组 S_{node} 中的偏移量。底层索引构建完毕后，会发布到查询节点，构

万方数据

建节点在构建过程中，只记录父子节点和兄弟节点在 S_{node} 中的偏移量，这使得索引发布过程中能更好的对其序列化。查询节点收到 S_{node} 后，根据偏移量可重新赋于对应的节点引用。

```
Class Node{
boolean isLeaf;      //是否为叶子节点
Node parent;         //父节点引用
int parentsIdx;      //父节点在Snode数组中的偏移
Node next;           //叶子节点的下一个节点引用
int nextIdx;         //叶子节点的下一个节点在Snode数组中的偏移
object[] keys;       //节点所有的key值
int [] keysIdx;      //节点所有key值所对应流元组数组的偏移
node[] childs;       //节点所有子节点的引用
int[] childsIdx;     //节点所有子节点在Snode数组中的偏移
KV[] chKeys;         //叶子节点所有流元组引用
}
```

图 4-3 B+树节点结构

Figure 4-3. The structure of B+ tree node

如图 4-2 所示，底层索引的构建过程可以概括为排序（P1）、B+树骨架构建（P2）、预装载（P3）和赋值（P4）四个阶段：

（1） 排序

为了提高流元组的排序效率，本方法对时间窗口进行分片，在接受数据流的同时对已到达分片排序。针对分片内的流元组，可采用并行排序的方式加速排序过程。并行排序过程首先要对分片内的数据分段，针对每个分段数据，分配一个线程对其排序，等所有分段排序完成后，再分配一个线程对已排序分段进行多路归并，从而完成排序。对于并行排序，若分片内数据量为 n，使用 m 个线程进行排序的时间复杂度为:

$$O(\frac{n}{m}\log(\frac{n}{m}) + n\log(m)) \tag{4-1}$$

对于单线程排序，时间复杂度为 O(nlogn)。当构建节点的核数多，分片内的流元组数量大，采用并行的排序方式可以显著地提高排序效率；相反，若分片内元组数量小，构建节点核数少，则直接采用单线程的排序方式效率反而更高。

当一个分片数据排序完成后，会发布归并排序任务，将该分片与之前已到达数据归并。当最后一个分片的归并任务完成后，整个时间窗口内的流元组即全局有序。由于归并排序与分片排序并行，故只能采用二路归并法。若分片数量为 k，时间窗口内的数据量为 n 时，归并排序的总时间复杂度为 O(nk)。构建节点由一

万方数据

个线程负责归并排序，这使得归并排序可以和分片排序、流元组接收并行，进一步提高了排序效率。

（2） B+树骨架构建

B+树骨架构建阶段首先计算B+树参数。当时间窗口内的流元组缓存结束后，便可得到时间窗口内的流元组数量，并以此来计算 B+树结构。若时间窗口的流元组数据量为 N_W，则可计算出以下参数[68]：

i. B+树的层数 N_h：

$$\begin{aligned} N_b^{N_h-1} < N_w \le N_b^{N_h} \\ N_h = \left\lceil \log_{N_b}^{N_w} \right\rceil \end{aligned} \tag{4-2}$$

ii. B+树内节点的子节点数 N_m：

$$N_m = \sqrt[N_h-1]{N_w / N_b} \tag{4-3}$$

iii. B+树内节点数 N_{inner}：

$$N_{inner} = \sum_{i=0}^{N_h-2} N_w^{\,i} \tag{4-4}$$

iv. B+树叶节点数 N_{leaf}：

$$N_{leaf} = N_m^{\,N_h-1} \tag{4-5}$$

v. B+树叶节点中的（Key, Vaule）个数 N_d：

$$N_d = W / N_{leaf} \tag{4-6}$$

其中，由于窗口内的B+树不存在后续的更新，为了保证B+树分布均匀且完全平衡，计算所得的内节点和叶节点的子节点数量有所差异。当叶节点数量 N_{leaf} 无法被 N_w 整除时，会在前 N_r 个叶节点中多存储一个（Key, Value），其中 N_r 为：

$$N_r = N_w \% N_{leaf} \tag{4-7}$$

由上述计算结果可得：B+树节点数量为 $N_{inner}+N_{leaf}$。根据节点数量可批量申请对应大小的节点数组 S_{node}，从而减少空间申请的时间开销。当数据流全部到达后，排序阶段仍未完成，还需对最后一个分片排序和归并，因此，参数的计算可以在排序的同时并行完成。

万方数据

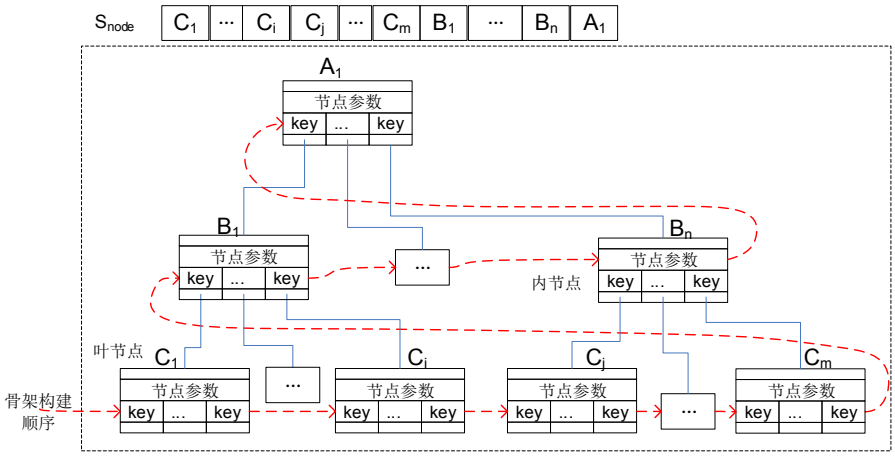

图 4-4 B+树骨架构建顺序

Figure 4-4. The order of B+ tree skeleton construction

B+树骨架构建的本质是关联节点间的父子和兄弟关系。对于每一个节点，根据其在 S_{node} 中的偏移以及计算所得的 B+树参数，即可计算出其父子节点和兄弟节点在 S_{node} 中的偏移，利用偏移值将对应节点关联即可完成 B+树骨架的构建。但若对每个节点都直接计算父子结点的偏移值，会存在计算量过大的问题，进而浪费构建节点的资源。本文采取自左向右、自底向上的构建顺序来减少骨架构建阶段的计算量，如图 4-4 所示。由于构建过程具有连续性，在构建当前节点时记录其子节点位置，称为当前构建位置。在处理下一个节点时，根据当前构建位置和子节点数量向下遍历，便可完成父子节点关联。对于兄弟结点的偏移量则只需用当前节点的偏移量加 1 即可。该方法有效地减少了计算量，从而提高了骨架构建的性能。B+树骨架构建过程完全独立于流元组，故可以在排序的同时并行构建。

（3） B+树预装载

B+树骨架构建阶段，完成了 B+树节点之间的关联。对于每个 B+树节点，还需要装载对应的“Key”值，这些“Key”值依赖于窗口内的流元组，要等到窗口流元组完成排序后才能确定。最终有序的流元组会存储在 S_{sort} 数组中，因此每个节点可根据“Key”值在 S_{sort} 数组中的偏移量预装载 B+树节点，从而降低最后赋值阶段的时间开销。预装载过程和 B+树骨架构建过程类似，装载顺序也是自左向右、自底向上。预装载完成后，各节点的“Key”值即为实际“Key”值在 S_{sort} 数组中的偏移值。

（4） B+树赋值

预装载过程中，已经得出所有“Key”值在 S_{sort} 数组中的偏移值。流元组排序结束后，根据所得偏移值，通过 S_{sort} 获取并赋于正确“Key”值即可完成 B+树的构建。由于每个 B+树节点的赋值过程都是相互独立的，可并行的对 B+树赋值，从而提高赋值阶段的效率。

万方数据

具体的底层 B+树索引构建方法如算法 3 所述。

算法 3. 基于并行排序的预装载 B+树批量构建法

输入：窗口时长 T_W，分片数量 N_{slice}，节点容量 N_b

输出：时间窗口对应的 B+树节点数组 S_{node}

1. 时间窗口开始后，调度线程根据分片时长定时从缓冲区中获取流元组，并存储在 SliceData 数组中，异步通知构建线程对分片排序。
2. 构建线程初始化值为 N_{slice} 的同步器 C_{sort} 和 C_{merge}，用于并行排序过程中的线程同步。若同步器已经初始化，则直接发布分片排序任务 ParallelSort(SliceData, $N_{sortTheread}$, C_{sort})。
3. 当时间窗口时间结束后，调度线程获取时间窗口内流元组总数 N_w。通知构建线程，构建线程发布 buildBTree(N_w) 任务构建 B+树。
4. 根据同步器 C_{sort} 和 C_{merge}，等所有排序和树结构构建任务完成后，发布 GiveKeys (S_{node}, S_{sort}, start, end)赋值任务。其中赋值操作可以并行，每个线程负责(start, end)范围的节点赋值。
5. 返回构建完成的 B+树节点数组 S_{node}。

过程：ParallelSort(SliceData, $N_{sortTheread}$, C_{sort})

1. 对分片内的流元组 SliceData 根据设定的线程数进行并行排序。
2. C_{sort} ←C_{sort}– 1. //表示一个分片排序完成
3. 发布 Merge(Slice[]，C_{merge})任务。

过程：Merge(Slice[], C_{merge})

1. 获取当前全局有序的流元组数组 dataTmp，归并排序结束后更新 dataTmp。

```
if   dataTmp = null ：
    then dataTmp← Slice[]
else
    dataTmp← Merge(dataTmp, Slice[]);     //二路归并，更新当前有序的流元组数组
    Cmerge ←Cmerge -1;                    //表示一个分片归并完成
```

过程：buildBTree(N_w) //构建 B+树 N_w：时间窗口内流元组数量

1. S_{node} = ComputeParams(N_w) //获取节点数组
2. BuildTreeStructure(S_{node}, Param) //构建 B+树骨架
3. PreLoadBTree(S_{node}, Param) //预装载 B+树

过程：ComputeParams(N_w) //计算 B+树参数,创建节点数组

1. 根据本节的计算方法计算出 B+树层数 N_h、内节点子节点数 N_m、内节点数 N_{inner}、叶节点数 N_{leaf}、叶节点子元组数 N_d 等参数。
2. 创建 S_{node} 节点数组，并根据 B+树参数为每个节点预开辟空间。
3. 返回 S_{node} 节点数组。

过程：BuildTreeStructure(S_{node}, Param) //构建 B+树骨架 Param：B+树参数

1. 初始化构建进度 currNode ← 0。
2. 对于内节点偏移 i= Param.leafSize, …, Param.nodeSize, buidNode ← S_{node}[i]:

```
while j< buidNode.childs.size
    do   buidNode.childsIdx[j] ← currNode;          //子节点关联
```


万方数据

```
            S_node[currNode].parentsIdx ← i;                    //父节点关联
                if  S_node[currNode].isleaf :                   //叶节点兄弟节点关联
                    then    S_node[currNode].nextIdx ←S_node[currNode+1]
            currNode ← currNode + 1;                            //更新构建进度
            j← j + 1;
        end
```

过程：PreLoadBTree(S_{node}, Param)　　//预装载 B+树

1. 预装载叶子节点，定义装载进度 curIdx ← 0。
2. 对于叶子节点偏移 i= 1, …, Param. leafSize，loadNode ← S_{node}[i]:

```
            while   j< buidNode.keys.size
             do   loadNode.keyIdx[i] ← curIdx          //预装载叶子节点
                  curIdx← curIdx + 1;
                  j← j + 1;
             end
```

3. 预装载内节点，定义装载进度 currNode ← 0。
4. 对于内节点偏移 i= Param.leafSize, …, Param.nodeSize, loadNode← S_{node}[i]:

```
    while   j< loadNode.keys.size：
      do   loadNode.keysIdx[j] ← currNode.keysIdx[0];
           currNode ← currNode + 1;
           j← j + 1;
      end
```

过程：GiveKeys (S_{node}, S_{sort}, start, end)　　//节点赋值

1. 对于 i=start,…,end-1，待赋值节点 Nd ←S_{node}[i]，对于 ch=0,1,…, Nd.numKeys-1：
   Nd.keys[ch]=S_{sort}[Nd. keysIdx[ch]].key;

4.1.2 底层索引发布

构建节点在构建底层索引时，涉及到大量的排序和计算过程，属于计算密集型节点，节点负载较高。底层索引构建完成后，若将底层索引维护在索引构建节点并提供查询服务，其查询性能将会很差，而且会影响后续底层索引的构建性能。因此，底层索引构建完成后，需将其发布到查询节点以处理相应的查询请求。底层索引包含了时间窗口内所有流元组的“Key”值，会占用一定的存储空间，为了保持各个查询节点的负载均衡，每次发布时，会请求主控节点获取合理的查询节点，并将索引发布到对应节点。

索引发布过程中涉及到 S_{node} 的序列化。索引构建节点在构建 B+树时，节点中只存储所有关联节点的偏移量，所有的引用连接将在发布完成后由查询节点完成，这保证了 B+树节点中的字段都是基本数据类型，避免了序列化过程中对引用进行递归序列化而导致的性能问题。节点序列化后的大小是影响传输性能的关键因素，本文使用由谷歌提出的 Protocol Buffers 格式进行数据传输，其二进制编码格式紧凑，解析速度快，适用于分布式场景下的数据传输。查询节点在接收并反序列化节点数据后，需要将 S_{node} 中每个节点关联的偏移量转换成相应的引用值，从而重建完整的 B+树索引。转换过程需要遍历 S_{node} 数组，时间复杂度为

万方数据

O(n)，而且转换过程对于每个节点都是相互独立的，故可以利用并行的方式加速B+树重建。

底层索引中，除了 B+树索引，还包含辅助索引。当底层索引存储在内存中时，内存的访问开销很小，可忽略不计，查询时无需利用辅助索引结构过滤请求。故针对维护在内存中的底层索引结构，只包含 B+树索引，底层索引持久化后，才会添加相应的辅助索引。因此，辅助索引将在持久化阶段构建。具体的底层索引发布方法如算法 4 所述。

算法 4. 底层索引发布

1. 索引构建节点根据缓存中的主控节点信息，远程调用主控节点的 selectNode()方法选取合适的查询节点。
2. 索引构建节点通过远程调用 releaseIndex(S_{node})方法发布底层索引。
3. 查询节点发布 rebuildIndex(S_{node}, start, end)任务，并行的重建底层索引。

过程：selectNode () //选取合适的索引发布节点

1. 主控节点根据当前的节点权重列表 $C_{curr}(N_i)$，选择权重最大节点 t：

$$t \leftarrow Max(C_{curr}(N_i))$$

2. 返回节点 t 信息，更新权重列表 $C_{curr}(N_i)$。

过程：releaseIndex(S_{node}) //底层索引发布 nodes: B+树节点数组

1. 利用 Protocol Buffers 格式序列化 S_{node}，生成对应的字节数组 nodeBytes。
2. 发送方在 nodeBytes 前添加字节数组长度，数据包格式为：nodeBytes.length+nodeBytes。
3. 查询节点接收到 nodeBytes 后，根据节点数据结构，将 nodeBytes 反序列化为 S_{node}。

过程：rebuildIndex(S_{node}, start, end) //底层索引重建

1. 将每个节点中的偏移量转换成引用。对于叶子节点偏移 i= start, …, end，buildNode←S_{node}[i]:

```
rebuildNode.parents←S_node[rebuildNode.parentsIdx]
if  rebuildNode.isLeaf :
      then rebuildNode.next ← node[rebuildNode.nextIdx]          //赋于引用值
else
      while  j< rebuildNode.keys.size
          do  rebuildNode. childs[j]←node[rebuildNode.childsIdx[j]]; //赋于引用值
              j← j+1;
          end
```

4.1.3 顶层索引更新

时间窗口 W_i 对应的底层索引构建并发布完成后，需要将时间窗口 W_i 对应的起始时间戳 t 作为“Key”值，对应底层索引的元信息作为“Value”值插入到顶层索引中。顶层索引每隔一个窗口时长更新一次，更新频率不高，其“Key”值是各窗口的起始时间戳，具有严格的递增性。传统 B+树更新过程中，为了保持B+树的平衡性，在插入过程中会涉及节点的分裂和移动，这会增加一定的时间开销。当一个节点分裂后，会将部分的元组移动到兄弟节点中去，原先的节点只

万方数据

保留部分元组。但在当前的应用场景中，“Key”值是单调递增的，故节点分裂后，将不会再有新的“Key”值插入到原来的节点中，原先的节点一直处于半满的状态，这浪费了存储资源。本文提出的顶层索引更新方法，结合“Key”值的特殊性以及顶层索引的更新频率，做了以下优化：

1. 顶层索引在插入过程中采用不分裂的策略，即每个节点的填充率都达到100%。若当前节点已满，则新建节点继续插入，如图 4-5 所示。这种方法虽然使得 B+树损失了一定的平衡性，但由于 B+树本身的树高非常有限，基本不会影响查询的性能。此外，该方法能大幅度提高空间利用率，也在一定程度上提高了更新效率。
2. 顶层索引的更新过程中，若待插入的叶节点已满，则需递归地查找未满父节点，然后递归地开辟新节点空间，这会增加一定的时间开销。对此，本文提出了空间预分配方法。顶层索引的更新频率为一个时间窗口的时长，若一次更新后，对应叶子节点已满，则在下一个时间窗口更新到达之前，预先开辟好节点空间。下次更新到达时，直接在新节点中插入即可。此方法可以有效地提高索引更新效率。

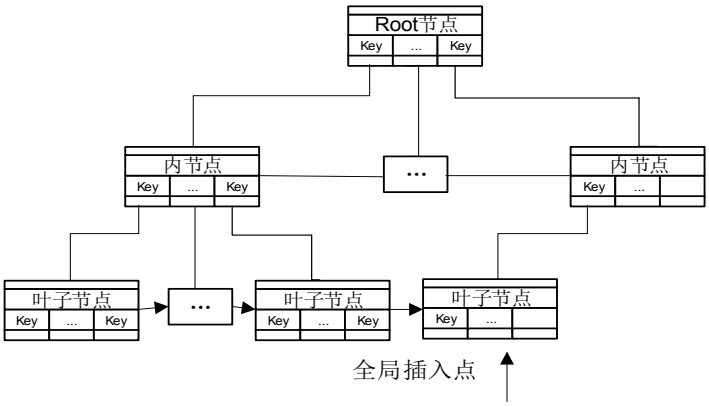

图 4-5 顶层 B+树更新示意图

Figure 4-5. The description of the top B+ tree update

表 4-2 顶层 B+树更新参数表

Table 4-2. The parameters of the top B+ tree update

参数	描述
B	节点容量（可容纳的子节点数）
P	全局的当前插入节点
P_{Node}	节点内的当前插入点（每个节点私有）
H	当前 B+树的层高
N	当前时间窗口总数
Root	B+树当前根节点

万方数据

顶层索引更新过程中涉及到的参数如表 4-2 所示，详细的更新过程如算法 5 所述。

算法 5. 顶层 B+树更新

输入：待插入时间窗口起始时间戳 K_{time}，时间窗口对应的底层索引元信息 V_{meta}

输出：更新后的顶层 B+树

1. 若当前的顶层索引为空，调用 initIndex()方法初始化顶层索引。
2. 调用 updateIndex()方法更新顶层索引。

过程：initIndex(B)　// 初始化顶层索引　B：节点容量

1. 根据参数 B，申请一个节点空间，初始化 node 节点，设置 Root 节点：Root← node。
2. 更新窗口总数：N ← 1；更新 B+树树高：H← 1。
3. 更新全局插入节点：P ← node ; 更新节点内部的插入点：P_{Node}← 0。

过程：updateIndex (K_{time}, V_{meta})　// 更新顶层索引

1. 根据插入点 P 找到对应的叶节点，直接插入并更新 P_{Node}：

   P.key[P_{Node}] ←K_{time};

   P. childKeys [P_{Node}] ←V_{meta};

   P_{Node} ← P_{Node}+1 ;
2. 若更新后 P_{Node} = 1，则表示插入的元组为节点 P 中的首个元组，则需递归地向上传播“Key”值：

```
giveKey(key,currentNode):
    if   currentNode = Root or currentNode.keyslength = currentNode.childslength
      then   return;
    else
      currentNode. Keys.add(key);
      currentNode ← currentNode.parents;
      giveKey(key, currentNode);
```

3. 若更新后的 P_{Node} = B，则表明当前叶节点已满，生成异步任务 initSpace(P)。
4. 完成顶层 B+树索引的更新。

异步任务：initSpace(P)　// 预先开辟新空间 P: 当前已满的全局插入点

1. 从 P 节点开始，递归地向上找到第一个未满的父节点，保存其引用值 P_{insert}。

```
findNotFullNode(P):
    if   P = Root and P.isFull :     //若递归到根节点，根节点也为满
      then   Root ← new Node(B) ;   // 创建新节点作为 Root 节点
          Root.keys[0]= P.keys[0];
          Root.childs[0] = P;
           return   Root;
    if   P.isNotFulll :
      then   return P;
    if   P.isFulll :
      then   return   findNotFullNode(P.parents);
```

2. 为了保持构建过程中树的相对平衡，从 P_{insert} 开始递归地向下创建子节点，直到到达叶节点。

```
createNode(P_insert):
```


万方数据

```
if  P_insert.height = 1            //创建的节点已到达叶节点
    then  return;
else
    tmp ← new Node(B);
    tmp.height ← P_insert.height-1;
    P_insert.childs.add(tmp);
    createNode(tmp);
```

本文提出的顶层索引更新方法在最优的情况下，即全局插入点所在的叶节点未满，则直接将元组插入到对应叶节点，时间复杂度为 O(1)；在最坏的情况下，即全局插入点所在的叶节点是新节点，在插入叶子节点的同时需要递归更新父节点的“Key”值，时间复杂度为 O(H)，由于 B+树的树高很小，故在最坏的情况下，也能保持良好的更新性能。传统的 B+树更新方法会随着 B+树的增大而降低，而本文提出的更新方法时间开销较为稳定，在“Key”值为窗口时间戳的场景下相比传统更新方法更具优势。

4.2 分布式索引构建性能评估

4.1 小节详细介绍了 WB-Index 的构建过程。本节对构建性能进行量化评估，其中涉及到的相关符号如表 4-3 所示。

表 4-3 索引构建符号表

Table 4-3. Some notations of index construction

符号	描述
T_W	时间窗口时长（单位：秒）
T_M	一次读 / 写内存的时长（单位：秒）
$N_{bandWidth}$	集群中节点间带宽（单位：Mbps）
N_W	时间窗口内的流元组数量（单位：个）
N_{leaf}	B+树叶子节点数量（单位：个）
N_{inner}	B+树内节点数量（单位：个）
N_{slice}	B+树构建过程中的分片数（单位：个）
$N_{sortThread}$	B+树构建过程中分片排序的线程数（单位：个）
$N_{keyThread}$	B+树构建过程中赋值的线程数（单位：个）
$N_{rebuildThread}$	查询节点重建 B+树索引的线程数（单位：个）
N_{height}	顶层索引更新时的当前树高（单位：层）
t_{base}	底层索引构建时延（单位：秒）
$t_{release}$	底层索引发布时间开销（单位：秒）
t_{top}	顶层索引更新时间开销（单位：秒）

万方数据

表 4-3 索引构建符号表（续表）

Table 4-3. Some notations of index construction (continued)

符号	描述
M	B+树节点的子节点数（单位：个）
t_{sort}	分片内排序时长（单位：秒）
$t_{sortDelay}$	分片排序带来的总时间延迟（单位：秒）
t_{merge}	归并排序时长（单位：秒）
$t_{mergeDelay}$	归并排序带来的总时间延迟（单位：秒）
t_{skltn}	B+树骨架构建时长（单位：秒）
t_{preLd}	B+树预装载时长（单位：秒）
$t_{giveKey}$	B+树节点赋值时长（单位：秒）
$t_{seialize}$	B+树序列化时长（单位：秒）
$t_{deseialize}$	B+树反序列化时长（单位：秒）
t_{send}	底层索引发布时长（单位：秒）
$t_{rebuild}$	B+树重建时长（单位：秒）
t_{delay}	WB-Index 总构建时延（单位：秒）

对于 CPU 的一次计算时间，其远远小于内存的读写时间，在理论分析过程中可以忽略不计。此外，流元组的“Key”值按照整型数据类型计算。

4.2.1 底层索引构建性能评估

针对流元组数量为 N_W，子节点数量为 m 的 B+树索引结构（为了便于计算，每个节点的子节点数量都近似看成 m），其叶子节点数量 N_{leaf} 为：

$$N_{leaf} \approx \frac{N_W}{m} \tag{4-8}$$

其内节点数量 N_{inner} 为：

$$N_{inner} \approx 1+m+m^2+\cdots+m^{N_{height}-1}=\frac{1-m^{N_{height}-1}}{1-m} \tag{4-9}$$

由于 B+树节点的子节点数 m 远大于 1，故近似计算可得：

$$N_{inner} \approx m^{N_{height}-2}=\frac{N_W}{m^2} \tag{4-10}$$

对于每个时间窗口，t_{base} 即为从窗口结束到底层 B+树索引构建完成的时延，包括了 $t_{sortDelay}$、$t_{mergeDelay}$、t_{skltn}、t_{preLd}、$t_{giveKey}$ 的时间，以下分别对各环节进行分析。

针对每个分片，若采用 $N_{sortThread}$ 线程数对分片内流元组排序，则时间开销 t_{sort} 为：

$$t_{sort}=T_M\left(\frac{4N_W}{N_{slice}\times N_{sortThread}}\times\log\left(\frac{N_W}{N_{slice}\times N_{sortThread}}\right)+\frac{2\log(N_{sortThread})N_W}{N_{slice}}\right) \tag{4-11}$$

万方数据

将式子(4-11)对 $N_{sortThread}$ 求导可得：

$$\frac{d(t_{sort})}{d(N_{sortThread})}=T_M(\frac{4N_W(-\ln(N_W/N_{slice}N_{sortThread}))-1}{N_{slice}N_{sortThread}^2})+\frac{2N_W}{N_{slice}N_{sortThread}}) \tag{4-12}$$

由式子(4-12)可得， t_{sort} 存在极小值，t_{sort} 的时间开销会随着 $N_{sortThread}$ 的增加先减少后增加。对于数量较小的时间窗口，极小值点小于 1，即直接采用单线程排序方式性能最优。式子(4-11)中的极小值点是理想状态下的值，由于排序属于 CPU 密集型任务，若极小值点大于构建节点的 CPU 核数，则线程数量将超过机器核数，进而导致大量的线程上下文切换，影响分片内排序的性能。

对于片内排序时长，若 t_{sort} 时长小于分片时长，排序总时延就是最后一个分片排序的时间开销，即 $t_{sortDelay}=t_{sort}$。数据流具有高速性，一个分片的排序时间可能会超过分片时间。为了便于评估，假设不同分片间排序任务串行执行。当一个时间分片全部到达后，前一分片的排序仍然没有完成，此分片排序任务会被积压，从而会产生额外的排序延迟，此时 $t_{sortDelay}$ 为：

$$t_{sortDelay}=t_{sort}\times(N_{slice}-1)-\frac{T_W(N_{slice}-1)}{N_{slice}}+t_{sort} \tag{4-13}$$

虽然不同分片间的排序任务可以并行执行，但在一定的 CPU 核数下，过多的并行任务也会影响分片内部排序性能，同样也会产生一定的排序延迟。

对于归并排序，所有分片的归并排序都是串行的。因此，归并排序的总时间开销为：

$$t_{merge}=2\times T_M\times N_{slice}\times N_W \tag{4-14}$$

由于前 N_{slice}-1 段归并排序与分片排序、流元组接收并行，若前 N_{slice}-1 段归并排序时间小于 $\frac{T_w(N_{slice}-2)}{N_{slice}}$，则 $t_{mergeDelay}$ 就等于最后一个分片归并的时延，即：

$$t_{mergeDelay}=2\times T_M\times 2\times N_W \tag{4-15}$$

若前 N_{slice}-1 段归并排序时间大于 $\frac{T_w(N_{slice}-2)}{N_{slice}}$，则 $t_{mergeDelay}$ 为：

$$t_{mergeDelay}=2\times T_M\times N_W\times N_{slice}-\frac{T_w(N_{slice}-2)}{N_{slice}} \tag{4-16}$$

对于排序阶段，时延为 $t_{mergeDelay}$ 与 $t_{sortDelay}$ 之和。综合式子(4-11)、(4-15)、(4-16)可得，若增加分片数量，可以减少片内排序时间 t_{sort}，但同时也会增加归并排序时长 t_{merge}。若前 N_{slice}-1 段归并排序时间和能小于 $\frac{T_w(N_{slice}-2)}{N_{slice}}$，随着分片数量的

万方数据

增加，$t_{mergeDelay}$ 能够保持不变。综上可以得出结论：存在分片数量 N_{slice}，使得底层索引构建过程中的排序总时延达到最低。

B+树骨架构建阶段，其中的 B+树参数计算只涉及到一些简单计算，可忽略这一部分的时间开销。骨架构建阶段的主要时间开销在于关联节点关系，包括关联所有节点的子节点和父节点，以及关联所有叶节点的兄弟节点，故 t_{skltn} 的时间开销为：

$$\begin{aligned} t_{skltn} &= T_M\left(\frac{N_W}{m^2}\times m+\frac{N_W}{m^2}+\frac{N_W}{m}+\frac{N_W}{m}\right) \\ &= \frac{(3m+1)\,N_W T_M}{m^2} \end{aligned} \tag{4-17}$$

预装载阶段，需要计算每个节点“Key”值在 S_{sort} 中的偏移量，t_{preLd} 时间开销为：

$$\begin{aligned} t_{preLd} &= T_M\left(\frac{N_W}{m}\times m+\frac{N_W}{m^2}\times m\right) \\ &= \frac{(m+1)N_W T_M}{m} \end{aligned} \tag{4-18}$$

赋值阶段，采用 $N_{keyThread}$ 个线程并行赋值，时间开销 $t_{giveKey}$ 为：

$$\begin{aligned} t_{giveKey} &= 2T_M\left(\frac{N_W}{m}\times m+\frac{N_W}{m^2}\times m\right)\times\frac{1}{N_{keyThead}} \\ &= \frac{2(m+1)N_W T_M}{N_{keyThread}} \end{aligned} \tag{4-19}$$

由式子(4-13)、(4-15)、(4-16)、(4-17)、(4-18)、(4-19)可得，$t_{sortDelay}+t_{mergeDelay}$ 远大于 $t_{para}+t_{skltn}+t_{preLd}$，故底层索引构建的时间开销为：

$$t_{base} \approx t_{sortDelay} + t_{mergeDelay} + t_{giveKey} \tag{4-20}$$

由于 $t_{giveKey}$ 的时间开销是固定的，且远远小于排序时延。故在窗口数据量和树结构确定的前提下，排序时延决定了底层索引的构建时延，而排序延迟又受分片数量 N_{slice} 影响。因此可得出结论：分片数 N_{slice} 是影响底层 B+树索引构建性能的关键因素。

4.2.2 底层索引发布性能评估

底层索引发布阶段的时间开销主要来自 B+树结构传输及重建开销，包括了 $t_{seialize}$、$t_{deseialize}$、t_{send}、$t_{rebuild}$ 这四个部分。

B+树传输过程首先需要序列化节点数组，序列化过程会序列化 Node 结构中的每个字段。计算时假设流元组的“Key”值为整型，并忽略序列化过程中写入的一些额外字段。反序列化是序列化的逆过程，为了简化计算，将反序列化的时间开销近似于序列化的时间开销。因此可得：

万方数据

$$t_{deseialize} \approx t_{seialize}=5T_M(\frac{N_W}{m}\times m+\frac{N_W}{m^2}\times m+\frac{N_W}{m}\times 2+\frac{N_W}{m^2}+\frac{N_W}{m^2}\times m) = \frac{5T_M N_W(m^2+4m+1)}{m^2} \tag{4-21}$$

节点间的传输开销和数据序列化后的大小密切相关。通过预实验得出，当子节点数 m 为 3000 时，每个 Node 序列化后的大小为 17KB 左右，设其为 A。故传输的时间开销 t_{send} 为：

$$t_{send}=\frac{A(\frac{N_W}{m}+\frac{N_W}{m^2})\times 10^{-6}}{8N_{bandWidth}} \tag{4-22}$$

查询节点反序列化节点数组后，并行的将每个节点关联的偏移量转换成相应的引用值，时间开销为：

$$t_{rebuild}=2T_M(\frac{N_W}{m}\times(m+2)+\frac{N_W}{m^2}\times(m+1))\times\frac{1}{N_{rebuildThread}} = \frac{2T_M N_W(m^2+3m+1)}{m^2 N_{rebuildThread}} \tag{4-23}$$

由式子(4-21)、(4-22)、(4-23)可得，底层索引发布阶段，$t_{rebuild}$ 时间开销远小于其他环节的时间开销，故可得出：

$$t_{release} \approx t_{send}+t_{deseialize}+t_{seialize} \tag{4-24}$$

由式子(4-24)可得，保持集群中节点性能不变的前提下，节点间的带宽是影响 WB-Index 构建效率的关键因素。

4.2.3 顶层索引更新性能评估

顶层索引在最坏情况下的更新时间开销 t_{top} 为：

$$t_{top}=\log(N_{height})\times T_M \tag{4-25}$$

由于 B+树的树高有限，t_{top} 时间开销可以忽略。顶层索引更新过程中，节点间传输的数据量非常小，传输开销也可以忽略。

为了保证 WB-Index 构建的稳定性，构建过程中，一个时间窗口的索引构建时延应小于窗口时长，即：

$$t_{delay} < T_W \tag{4-26}$$

若不满足该条件，数据流将会在索引构建节点中不断的堆积，影响后续窗口数据流的索引构建。其中，N_w 与 t_{delay} 直接相关，N_w 越大，构建延迟 t_{delay} 也相应越大。故针对一个 T_W 长度的时间窗口，时间窗口内的流元组数 N_W 存在着上限 N_{max}，即 WB-Index 构建方法所能支撑的最大数据流流速为 $\frac{N_{max}}{T_W}$。

万方数据

4.3 实验评价

4.3.1 实验环境

实验采用阿里云平台的 20 台 ECS 主机构建集群环境，使用 JAVA（JDK 1.8）语言实现 WB-Index 系统。每台 ECS 的配置如表 4-4 所示。

表 4-4 ECS 主机配置表

Table 4-4. The configuration of ECS

类别	配置
CPU 核数	4 核
处理器主频	2.5 GHz
处理器型号	Intel Xeon(Skylake) Platinum 8163
内存配置	32GB
内网带宽	1 Gbps
操作系统	Ubuntu 16.04(64 位)
磁盘大小	100 GB

对 20 台 ECS 主机节点用 Node1～Node20 进行编号，集群中具体的节点安排如表 4-5 所示。

表 4-5 WB-Index 集群节点分配表

Table 4-5. The nodes in WB-Index cluster

节点编号	节点类型
Node1	主控节点，查询节点，协调节点
Node2	协调节点，查询节点
Node3	协调节点，查询节点
Node4	构建节点
Node5～Node14	查询节点
Node15～Node20	存储节点

由于主控节点和协调节点的负载相对较小，为了提高集群的资源利用率，将其和查询节点部署在同一个节点中。实验中的仿真数据集采用 TPC-H 标准产生的 Line Item 关系中的数据，每条元组形如<Key,Value>，其中限制每条元组“Value”值的长度为 100 字节。实验中通过从内存读取数据来模拟数据流的流入。此外，实验模拟一个数据源对应数据流的索引构建、存储和查询，因此只安排一个索引构建节点。实验中 B+树的节点最大子节点数均设置为 3000。

万方数据

4.3.2 底层索引构建效率

底层索引构建过程中，重点关注的是构建延迟，其直接决定了 WB-Index 的构建性能。底层索引构建过程中，时间窗口分片内采用并行排序的方法加速排序过程，本实验针对不同的分片数据量，通过改变排序线程数量对比分片排序性能。

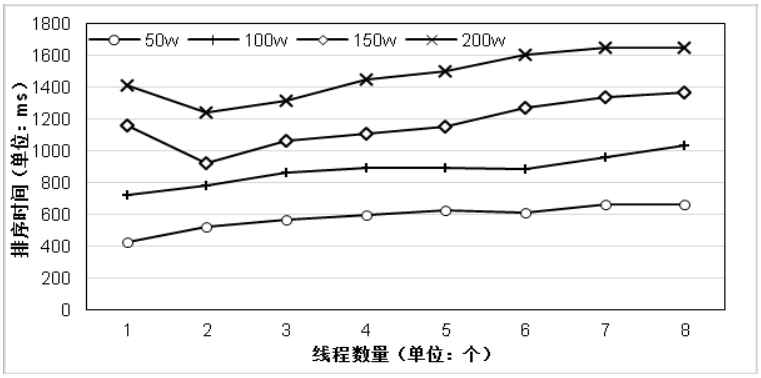

图 4-6 不同流元组数量下排序线程数量对排序性能的影响

Figure 4-6. Sort delay vs. the number of threads in different amount of tuples

实验结果如图 4-6 所示，当数据量较小时，单线程排序性能优于多线程排序性能，当数据量达到一定量级后，并行排序才会具有优势。此外，当线程数量达到一定阈值后，随着线程数量的增加，最终的排序时间反而会增加，这是因为过多的线程数量会导致最后归并排序时长增加，也会增加线程切换的时间开销，这符合 4.2 小节的理论分析结论。由实验结果可得，当数据量为 100 万和 150 万时，采用 2 个线程排序的性能最优。

由 4.2 小节分析可得，底层 B+树索引构建过程中，在窗口数量、排序线程数等条件确定的情况下，分片数 N_{Slice} 会影响构建时延。本实验评估了分片数量对底层 B+树索引的构建时延的影响。实验也对比了底层索引构建过程中流元组排序时延、B+树骨架构建+预装载时间、B+树赋值时间的占比情况。实验设置的窗口时长为 20 秒，窗口数据量为 2000 万条，即模拟的数据流平均流速为 100 万每秒。

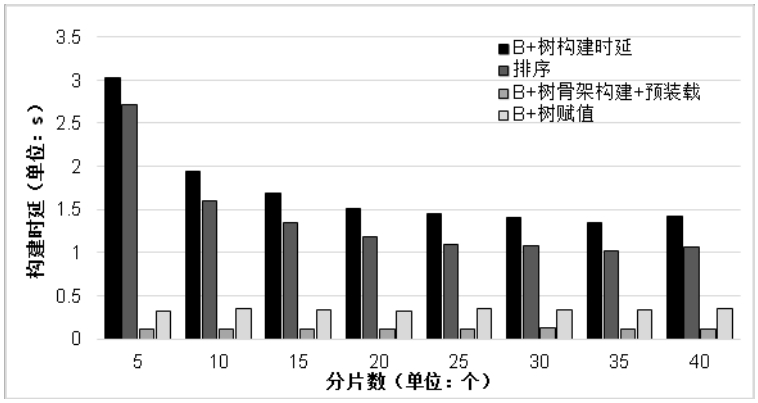

图 4-7 底层 B+树索引构建时延与分片数的关系

Figure 4-7. The construction delay of B+ tree vs. the number of slices

万方数据

实验结果如图 4-7 所示，底层索引的构建时延随着分片数的增加有着先降后增的趋势，这符合 4.2 小节中理论计算得出的结论。当分片数量超过 25 时，底层索引的构建时延基本稳定，且在分片数为 35 左右时构建时延达到最小。整个构建过程中，排序时延占构建时延的 70%～80%的左右，其他阶段的时间开销较小，且相对固定，这也证明了排序时延直接决定了构建时延。对于索引构建节点，可以采用高性能服务器，通过提高排序性能来减少构建时延。

数据流具有波动性，每个时间窗口的数据量存在较大的差异。本实验通过模拟不同流速的数据流来考察底层索引的构建性能。相比于传统的 B+树批量构建法，本文所提方法通过预排序和增加构建并行度的方法来提高构建效率，本实验也对比不同数据流流速下两种 B+树构建方法的性能。实验中设置的窗口时长为 20 秒，分片数量设置为 35。

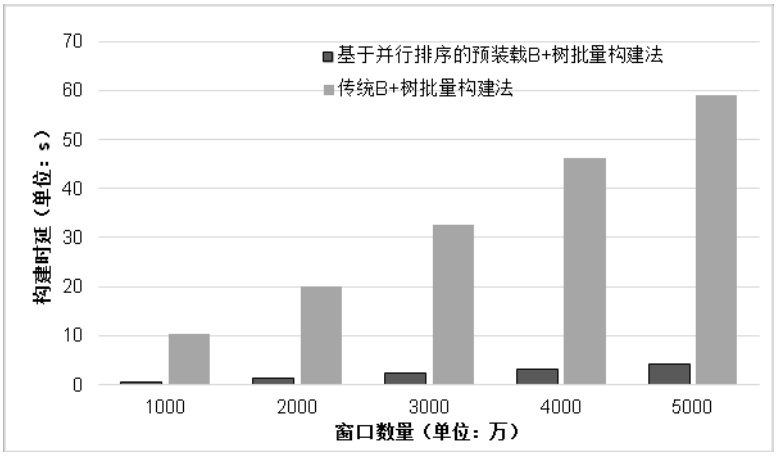

图 4-8　不同窗口流元组数量下两种 B+树构建方法的构建时延对比

Figure 4-8. The comparison of construction delays of two B+ tree construction methods under different number of stream tuples

实验结果如图 4-8 所示，底层索引的构建时延和单位窗口内的数据量基本呈线性关系，这表明底层索引的构建性能较为稳定，构建性能不会随着数据流流速的增大而下降。当单位窗口内的流元组数据量达到 5000 万时，底层索引的构建时延仅为 4.2 秒，这证明了底层索引的构建效率高，能够支撑高速数据流。另外，由实验结果可得，基于并行排序的预装载 B+树批量构建法相比于传统 B+树批量构建法具有很大的优势，且随着数据量的增大，优势更加明显，这也证明了底层索引构建方法的有效性。

4.3.3 顶层索引更新效率

对于顶层索引，本文提出了预分配空间的不分裂 B+树更新方法，来提高空间利用率以及更新效率。本实验通过模拟更新不同量级的窗口数来评估顶层索引的更新性能。

万方数据

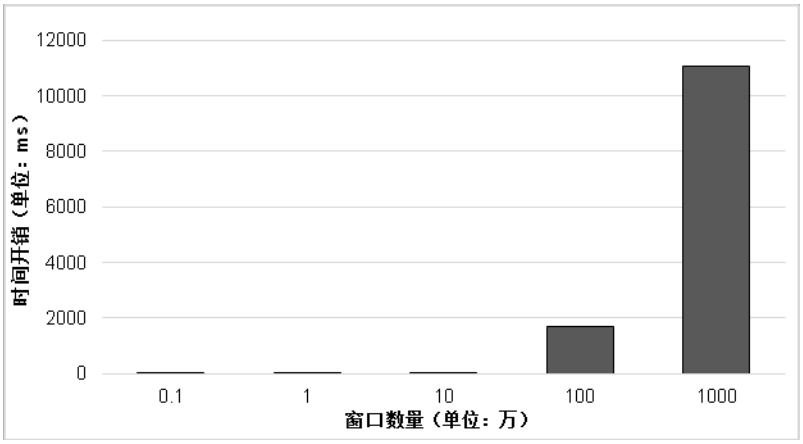

图 4-9 顶层 B+树更新窗口数量与更新时间的关系

Figure 4-9. The update time of B+ tree vs. the number of windows

实验结果如图 4-9 所示，顶层索引更新的时间开销和更新的窗口数量基本呈线性关系，顶层索引单次更新时间不会随着顶层索引的量级变大而大幅增加。从实验结果可得，1000 万次的顶层索引更新总耗时仅为 11 秒，每次的更新时间开销可以忽略不计，符合 4.2 小节理论计算得出的结论。这也证明了顶层索引更新方法的高效性，其能很好地适用于基于时间窗口的应用场景。

4.3.4 WB-Index 构建效率

WB-Index 构建过程中，由底层索引构建、底层索引发布、顶层索引更新三部分构成。本实验通过改变单位时间窗口内的流元组数量，来评估 WB-Index 的构建效率，并对比构建过程中三个环节的时间开销。实验中设置的窗口时长为 20 秒，分片数量为 35。

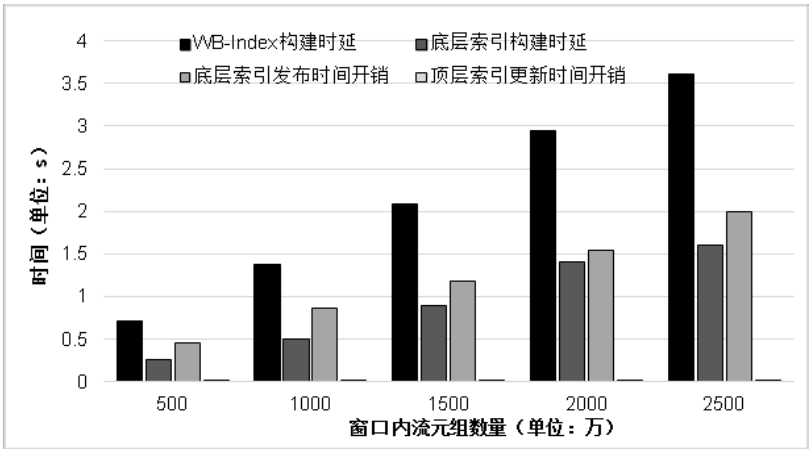

图 4-10 窗口内流元组数量与 WB-Index 构建时延的关系

Figure 4-10. The construction delay of WB-Index vs. the number of stream tuples

实验结果如图 4-10 所示，WB-Index 构建方法的性能稳定，构建时延与窗口数据量基本呈线性关系。其中，底层索引发布过程中，涉及到了网络 IO 开销、

万方数据

序列化和反序列化过程，相比起其他两个环节，时间开销较大。顶层索引的更新时间开销占比最小，可以忽略。

在相关工作中介绍到，CG-Index[56]是一种主从结构的分布式索引，LSM-tree[41]结构也可以看成一种分布式索引，广泛地应用于数据高速写入的应用场景。本实验对比 WB-Index、CG-Index 和 LSM-tree 三种分布式索引的构建性能。为了确保实验的可比性，CG-Index 中采用 10 个节点作为存储节点。LSM-tree 结构的写入磁盘阈值设置为 200 万条流元组。WB-Index 索引设置的窗口时长为 20 秒，分片数量为 35。实验在 100 秒的时间内，通过改变流元组的数量来模拟不同的数据流流速，进而比较三种索引的构建效率。

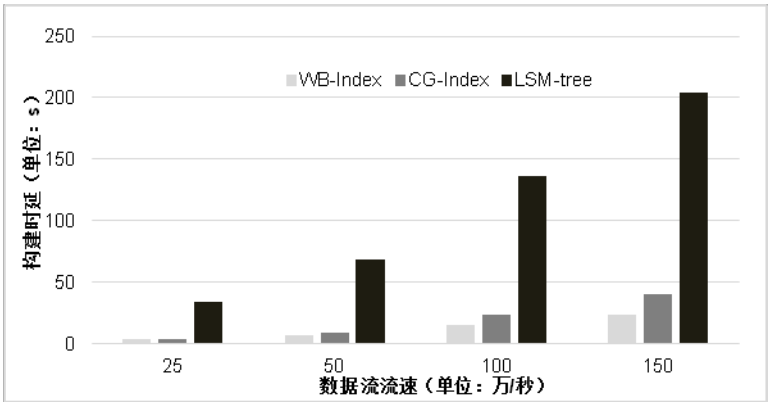

图 4-11 WB-Index、CG-Index、LSM-tree 索引构建效率对比

Figure 4-11. The comparison of construction delay between WB-Index, CG-Index and LSM-tree

实验结果如图 4-11 所示，WB-Index 在构建性能上优于 CG-Index 以及 LSM-tree，而且随着数据流流速的增大，其构建性能的优势也更加明显。CG-Index 虽然可以将流元组分发到不同节点以减少 Local Index 的大小，但随着时间的推移，Local Index 不断变大，更新开销会不断增加。相比之下，LSM-tree 结构可以定期将索引刷到磁盘，以保证索引更新效率。LSM-tree 中，流元组实时写入，有效地保障了实时性，但其构建效率远不如 WB-Index。当然，WB-Index 在构建过程中需要缓存一个时间窗口的数据，在实时性上有一定的损失，针对小流量和低流速的场景将会不具备优势。另外，CG-Index 和 LSM-tree 中，索引和数据都是结合在一起的，CG-Index 由于更常用于二级索引，其数据量较小，可以将尽可能多的索引维持在内存中以提高查询效率。对于 LSM-tree 而言，若每条纪录较大，缓存在内存中的索引很快就会达到内存限制，需要持久化到磁盘，查找时需遍历磁盘中的索引结构进行查找，查询性能较差。WB-Index 索引结构中，将索引结构和流元组分离，从而可以缓存大量的索引结构，海量数据下的分布式查询过程结合了并行查询的思想，查询效率比起 LSM-tree 也有很大的提高。

万方数据

4.3.5 WB-Index 所能承受的数据流最大流速

由 4.2 小节可知，为了保证 WB-Index 索引的平稳构建，一个窗口内的索引构建时延要小于时间窗口时长，这样才能避免数据流的堆积。本实验旨在评估 WB-Index 所能承受的最大数据流流速。假设数据流匀速到达，对于给定的时间窗口大小，时间窗口内到达的流元组数量与数据流速成正比，可以通过固定时间窗口改变窗口流元组数量的方法来模拟不同流速的数据流。本节实验设置的分片数为 35，时间窗口时长为 5 秒。

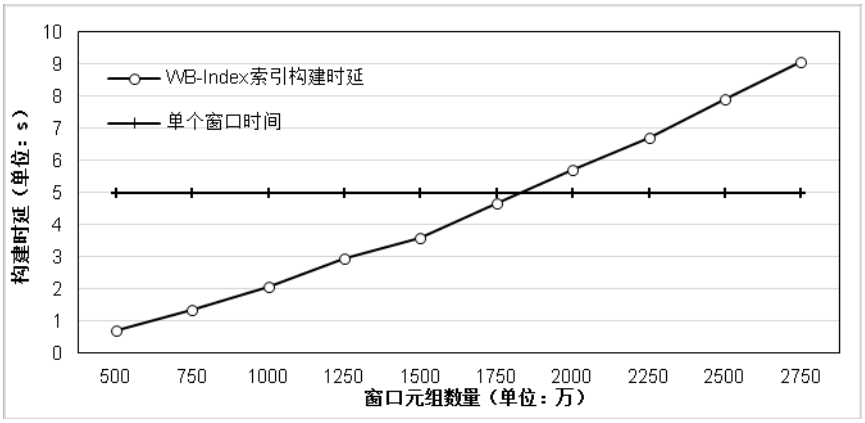

图 4-12　35 窗口数下 WB-Index 构建时延随流元组数量变化趋势

Figure 4-12. Tendency of WB-Index construction delay when change the tuple quantity

图 4-12 展示了窗口时长固定的前提下，数据量的增加对 WB-Index 构建时延的影响。从中可得，当时间窗口内的流元组数量达到 1800 万时，WB-Index 构建时延为 5 秒，与单个窗口时长（基准线）相交。故当前实验环境中，在窗口时长 5 秒和分片数为 35 的情况下，WB-Index 所能支撑的最大流速约为 360 万每秒。实验中的流元组大小约为 100B，则在理想情况下，WB-Index 能支撑百兆每秒级别的数据流速，具有很好的应用价值。

4.3.6 WB-Index 查询效率

WB-Index 分布式查询过程中，首先根据时间窗口范围定位到对应的底层索引，再将查询请求拆解到对应的查询节点，并行的查找底层索引并获取对应的流元组。本实验通过改变查询请求涉及的时间窗口数量以及查询范围涉及到的流元组比例，来评估 WB-Index 的查询性能。实验中每个时间窗口流元组数据量为 100 万，WB-Index 中的索引结构和流元组都维护在内存中，查询类型为基本的取数查询。

万方数据

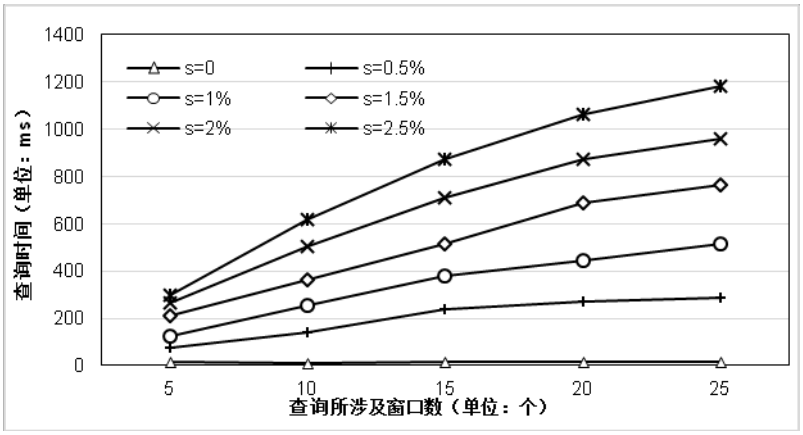

图 4-13 WB-Index 查询范围与 WB-Index 查询时间开销的关系

Figure 4-13. The query time of WB-Index vs. the query range of WB-Index

实验结果如图 4-13 所示，其中 s 为在每个时间窗口内范围查询涉及到的流元组占比。对于查询结果数据量较小的查询请求，查询时间能控制在秒级，查询性能较好。对于查询结果数据量较大的查询请求，如图中查询范围占比为 2.5%、涉及时间窗口为 25 的查询请求，其返回的查询结果有 60 多万个流元组，数据量约为 60MB，故返回查询结果时存在较大传输开销，查询效率也有所下降。

WB-Index 的整个分布式查询过程可以概括为顶层索引查找、底层索引查找、流元组获取、数据汇总这四个环节。本实验对比了分布式查询过程中各环节的时间开销。实验中固定窗口内流元组查询范围占比为 1%，通过改变查询涉及的窗口数量来对比不同数据量下的查询情况。

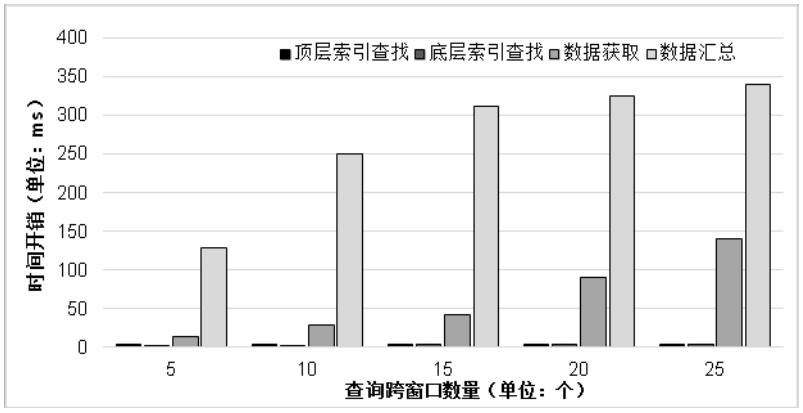

图 4-14 WB-Index 查询中各环节时间开销对比

Figure 4-14. The comparison of time cost between each step in WB-Index query

实验结果如图 4-14 所示，当底层索引和顶层索引维护在内存中时，底层索引和顶层索引的查询开销很小，均可忽略。由于底层索引的查找过程完全并行化，跨窗口数量的增加也基本不会影响底层索引的查询效率。分布式查询的时间开销主要来自于数据获取和数据汇总阶段。数据汇总阶段包含了对查询返回数据的处理、汇总和返回，实验中的查询类型为基本的取数操作，因此最终的返回数据量

万方数据

较大，从而导致返回的时间开销较大。对于一些返回数据量较小的查询请求，数据汇总阶段的时间开销将会大幅度降低。数据获取阶段是查询节点获取对应流元组的时间开销，当数据流缓存在索引构建节点或存储在分布式文件系统中时，获取流元组都存在一定的网络传输开销。因此，将流元组缓存在对应的查询节点可以有效地提高分布式查询效率，这也充分证明了合理设计流元组缓存的必要性。

4.4 本章总结

本章根据 WB-Index 索引结构，详细阐述了 WB-Index 的构建过程。WB-Index 构建过程由底层索引构建、底层索引发布和顶层索引更新三部分构成。底层索引构建阶段主要完成 B+树索引构建。根据数据流流速快的特点，本文提出基于并行排序的预装载 B+树批量构建法，该方法构建时延低，能支撑高速数据流的索引构建。索引发布阶段主要包括底层 B+树索引的传输和重建。顶层索引更新则将时间窗口的时间戳作为“Key”值，底层索引的元信息作为“Value”值插入顶层索引中。根据“Key”值单调递增的特点，本文提出了基于预分配空间的不分裂 B+树更新方法，既节省了顶层索引的存储开销，也提高了顶层索引的更新效率。最后，通过理论和实验分析证明了 WB-Index 索引构建方法的有效性。

万方数据

第五章 分布式索引持久化方法

第四章阐述了 WB-Index 索引构建方法，包括底层索引构建、底层索引发布和顶层索引更新这三个步骤。数据流具有无限性，集中式架构无法存储海量数据流。本文利用分布式文件系统来存储流元组和索引结构。本章在第四章的基础上，分别提出针对流元组、底层索引和顶层索引的持久化方法，并通过理论分析来评估相应的持久化性能，最后通过实验来证明持久化方法的可行性。

5.1 解决方案

数据流具有无限性的特点，集中式存储无法支持海量数据流的存储。本文利用分布式文件系统来持久化流元组和索引结构。HDFS 是目前主流的分布式文件系统，其采用主从架构，从节点以块为单位切分数据文件，每个块默认以三副本的方式存储在多台从节点中，以保证其高可用性。分布式文件系统虽然能够存储海量数据，但由于读写时涉及到跨节点的数据同步，导致其读写效率相对较差。对此，本文设计了紧凑的存储格式，通过减少存储量的方式来提高持久化效率。对于数据流查询，本文利用 B+树索引以及额外的辅助索引，减少查询过程中的随机读次数，从而提高查询性能。

WB-Index 索引结构中，为了更好地缓存底层索引，将流元组和底层索引分离，持久化阶段需分别持久化流元组和索引结构。对于每个时间窗口内的流元组，在底层索引构建完成后，其根据“Key”值全局有序。因此，持久化阶段只需按照顺序依次将流元组写入文件中，每个时间窗口内的流元组生成一个相应的数据文件。查找时，可以通过偏移量直接从数据文件中获取对应的流元组。底层索引中，包括了 B+树索引、布隆过滤器和统计信息三个模块，需分别持久化相应模块。B+树持久化后，不存在后续的更新操作，为了使得存储更加的紧凑，B+树以节点为单位，自底向上、自左向右的持久化相应节点，节点间的关系通过各节点在索引文件中的偏移量来维护。查找时，利用偏移量直接定位到对应节点，此方法在保证查询效率的同时也减少了存储开销。顶层索引具有量级小且更新频率低的特点，WB-Index 将顶层索引常伫内存以提高查询效率。由于内存存储具有易失性，本文利用预写日志和定时持久化的方式保障了顶层索引的可靠性。本节将详细阐述 WB-Index 的持久化方法。

万方数据

5.1.1 流元组持久化

本文以字节形式存储对应“Key”、“Value”值，并将每个数据流对应的“Key”、“Value”结构的元信息存储在协调节点中，查询时可根据元信息解析出对应的“Key”、“Value”值。流元组采用行式存储格式，具体的存储格式如图 5-1 所示。

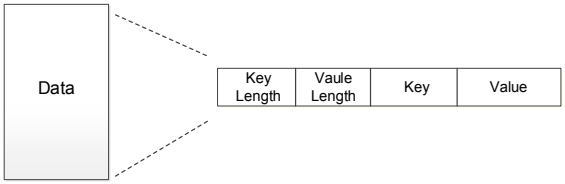

图 5-1 流元组存储格式

Figure 5-1. The storage format of stream tuple

对于每个流元组，都会带上 Key Length、Value Length 头部信息。查询时根据头部信息，可以读取到对应的“Key”、“Value”值。这种存储方式更加紧凑，并能支持“Key”、“Value”值的压缩存储。一个时间窗口内的流元组会形成一个对应的数据文件，数据文件将数据流标识和时间窗口起始时间戳组合作为文件名。查询时，首先根据数据源标识和窗口时间戳定位对应的数据文件，再根据底层索引中记录的偏移值获取对应的流元组。

为了提高新数据的查询效率，完成索引构建的数据流首先会缓存在索引构建节点，缓存期间由索引构建节点负责流元组的查询服务。当一个窗口流元组的缓存被替换后，需更新查询节点中对应底层索引的标志位，来表示后续对该窗口的查询需从分布式文件系统中获取对应的流元组。

若一个时间窗口内的流元组数量过多，可以使用并行持久化的方式，提高持久化效率。集群中的总体网络带宽会影响分布式文件系统的写入速度，在网络带宽有富余的情况下，通过并行写入的方式可以提高写入性能。针对一个时间窗口的流元组，持久化过程若采取 n 个并行度，最终会产生 n 个流元组文件。数据文件名需在原来的基础上加上分段信息，如图 5-2 所示。若一个窗口内的数据量小，使用并行的持久化方式会增加小文件数量，过多的小文件会增加分布式文件系统中主节点的压力。因此对于小数据量窗口，采用单线程的持久化方式即可。

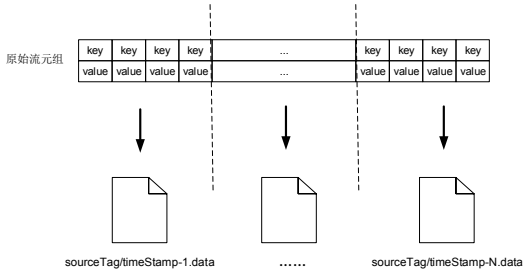

图 5-2 流元组的并行持久化方法

Figure 5-2. The parallel persistence method of stream tuple

万方数据

5.1.2 底层索引持久化

为了保障数据流的查询效率，查询节点在重建底层索引后，会将其缓存在查询节点中。海量数据下，索引无法全部缓存在查询节点，对此设置专门的线程持久化相应的底层索引。底层索引由 B+树索引、辅助索引构成，待其持久化后，辅助索引可以过滤不必要的数据读取，从而提高查询效率。

由于在索引构建过程中，没有构建相应的辅助索引。因此，持久化阶段需先构建辅助索引，构建过程如算法 6 所述。

算法 6. 构建辅助索引

输入：底层 B+树索引节点 S_{node} 数组，底层 B+树构建参数 p。

输出：构建完成的辅助索引

```
1. 初始化三个随机 Hash 函数: h1、h2、h3。
2. 初始化位数组 bitset, 位数组大小根据窗口流元组数量动态调节。
3. 构建统计信息 min、max:
        min←node[0].keys[0];
        lastNode←node[p.leafNodeSize-1];
        max←lastNode.keys[lastNode.keys.size-1];
4. 根据叶子节点中的所有“Key”值构建布隆过滤器。对于叶子节点偏移 i= 1, …,
   nodeSize，node ←S_node [i]:
        If  node.isLeaf:
            then   j←0;
                   while  j< node.keys.size
                     bitset [h1 (key)% bitset.length] ← 1;    //更新位数组
                     bitset [h2 (key)% bitset.length] ← 1;
                     bitset [h3 (key)% bitset.length] ← 1;
                     j← j + 1;
                   end
            else
                   break
```

布隆过滤器存在误判率，若时间窗口内流元组数量为n，布隆过滤器中的位数组大小为 m，Hash 函数的个数为 k，则布隆过滤器的误判率 P 为：

$$P \approx (1-e^{-\frac{kn}{m}})^k \tag{5-1}$$

通过式子(5-1)可得，当数据量一定时，m 和 k 越大，布隆过滤器的误判率越低。但是 m 越大会增加索引的存储开销，k 越大则会增加计算开销，进而增加索引的构建时长。因此，在构建布隆过滤器时，需根据时间窗口的数据量动态地调节 m 和 k 值，当 $m/n = 10$，k=3 时，布隆过滤器的误判率约为 1%。本文动态地调节布隆过滤器的参数，保证辅助索引的存储和查询效率。

持久化过程中，布隆过滤器和统计信息直接序列化成字节形式进行存储。查询时首先将对应模块读取到内存，反序列化后利用相应结构过滤无效查询。B+树的持久化以节点为单位，节点持久化的顺序为自底向上，自左向右。一个

万方数据

窗口内的 B+树不存在后续的更新操作，故将各节点连续的存储在磁盘中，通过节点在索引文件中的偏移量维护节点间的关系。另外，由于不存在后续的更新操作，父节点指针将不起作用，因此持久化阶段只维护子节点和兄弟节点关系。一个树节点的存储格式如图 5-3 所示。

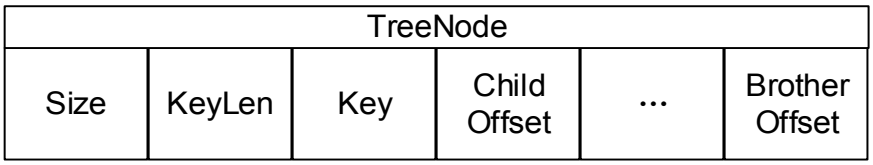

图 5-3 底层 B+树节点存储格式

Figure 5-3. The storage format of B+ tree node

其中 Size 表示当前树节点持久化后的大小，查询时可以根据 Size 值读取完整的树节点。KeyLen 表示“Key”值的大小，Child Offset 表示子节点在索引文件中的偏移，Brother Offset 表示叶节点中兄弟节点在索引文件中的偏移。由于内节点不存在兄弟节点，若一个节点的 Brother Offset 等于-1，则表示其为叶节点。查询时，读取的最小单元为一个 B+树节点，根据节点中子节点的偏移量逐层向下查找。B+树具有树高较低的特点，这使得索引查询过程中只涉及到几次随机 IO，查询效率较高。

在索引文件尾部，存在 Index Footer 模块，Index Footer 维护了底层索引结构的元信息，其中包括 B+树根节点、统计信息和布隆过滤器在索引文件中的偏移量。底层索引的存储格式如图 5-4 所示。

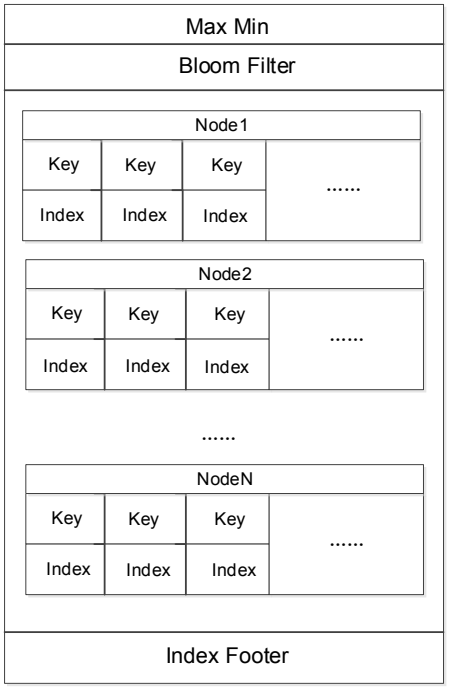

图 5-4 底层索引存储格式

Figure 5-4. The storage format of bottom index

万方数据

分布式查询过程中，若对应的底层索引没有缓存在内存中，则需要从分布式文件系统中获取对应的索引文件。查询过程中，依次读取索引文件中的各模块进行查找，具体步骤如下：

1. 读取 Index Footer 模块。Index Footer 模块大小固定且存储在文件的尾部，读取整个模块后，提取其中的统计信息、布隆过滤器和 B+树根节点的元信息。
2. 读取统计信息模块。根据获取的统计信息偏移值，读取统计信息。若查找条件中的“Key”值范围与统计信息中的“Key”值范围无交集，则停止查找流程，直接返回。
3. 读取布隆过滤器模块。若查询类型为点查询，则根据偏移量获取并反序列化出布隆过滤器，利用布隆过滤器判断查找的“Key”值是否存在，若不存在，则直接返回。
4. 读取 B+树根节点。B+树的查找以节点为单位，从根节点开始，每次从分布式文件系统中调取一个节点，根据节点中子节点的偏移量向下查找直到叶子节点。到达叶节点后，根据兄弟节点的偏移量，顺序地遍历叶子节点，获取符合查询条件的流元组元信息。

底层索引中只包含了流元组的“Key”值，持久化后的索引文件较小，且数据流的流速存在波动性，有些小数据量窗口对应的底层索引更小。在主从结构的分布式文件系统中，过多的小文件会增加主节点压力。对此，本文在底层索引存储格式的基础上，添加 Meta 模块，以支持不同窗口的底层索引合并，这能有效地减少小文件数量。添加 Meta 模块后的底层索引存储格式如图 5-5 所示。其中 Index Part 为一个窗口对应的底层索引结构。

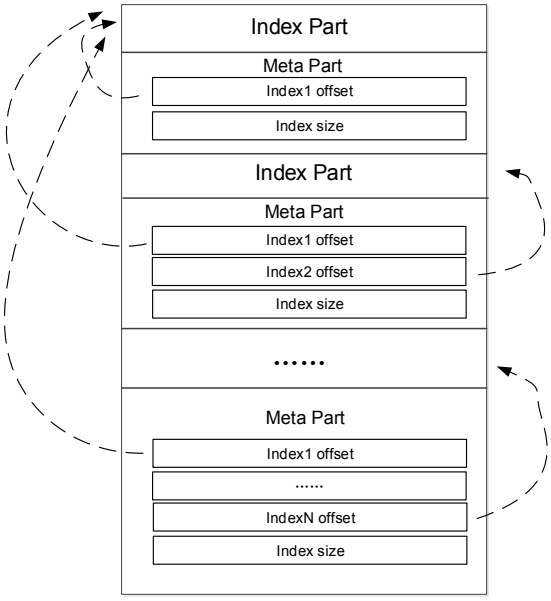

图 5-5 支持合并的底层索引存储格式

Figure 5-5. The bottom index storage format that supports merging

万方数据

Meta 模块中，记录了文件中包含的所有底层索引在索引文件中的偏移量。查找时，首先根据 Meta 结构定位到对应的底层索引结构，再进行后续的查找。索引文件名由其中包含的底层索引对应的窗口起始时间戳构成，用“-”分隔，例如：“time1-time2-time3.index”，查询时可根据时间戳定位对应得索引文件。

持久化底层索引时，首先判断前一个窗口对应的索引文件的大小是否满足最小阈值 A。若不满足，则将底层索引继续写入到该索引文件中，写入后在文件尾部添加新的 Meta 模块；若满足，则写入新的索引文件。其中，索引文件最小阈值 A 设置为分布式文件系统块大小即可。

底层索引持久化后，查询过程中会按需读取相应模块，涉及到几次随机 IO，相比底层索引缓存在内存中的情况，查询性能有所下降。查询存在热点性，当一个时间窗口对应的底层索引在单位时间 T 内的查询次数超过 N 次，则可判断其为热点数据对应的底层索引。对此，如果该底层索引没有缓存在内存中，可将其重新调入内存来提高热点数据的查询效率。其中 T、N 需根据时间窗口时长和查询频率合理调节。若 T 过大、N 过小，则会导致底层索引缓存的频繁更新；若 T 过小、N 过大，则会导致热点数据对应的底层索引难以缓存。

5.1.3 顶层索引持久化

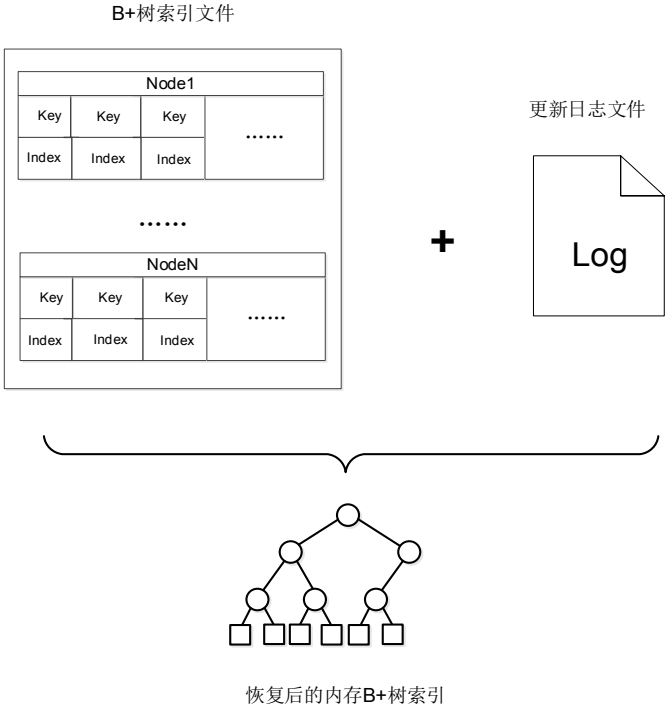

图 5-6 顶层索引恢复示意图

Figure 5-6. The diagram of top index recovery

顶层索引是一棵基于时间窗口的 B+树索引，维护了所有底层索引的元信息。相比于底层索引，顶层索引的存储开销小且更新频率低。因此，查询节点

万方数据

在内存中维护完整的顶层索引来提高查询效率。然而，内存具有易失性，对此，本文采用定期持久化和预写日志相结合的方式确保顶层索引的可靠性。

每次顶层索引更新时，会先将更新操作记录在日志文件中，预写日志过程属于顺序写入，性能较高。当内存中的顶层索引丢失时，可以通过重做日志文件中的更新操作来恢复顶层索引。随着时间的推移，日志文件的大小不断增加，通过完整的日志文件恢复顶层索引耗时过长。对此，查询节点定期对当前内存中的顶层索引持久化，相当于保存当前的快照。顶层索引持久化后，清空当前的日志文件，后面的更新日志写入新的日志文件中。恢复顶层索引时，只需在最新的顶层索引快照的基础上，重做日志文件中的更新操作即可，此方法可大幅度减少索引恢复时长。顶层索引恢复过程如图 5-6 所示。

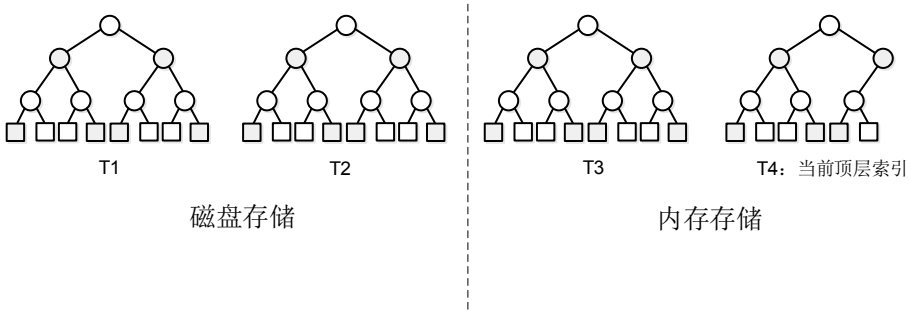

图 5-7　顶层索引分段示意图

Figure 5-7. The diagram of top index segmentation

数据流具有无限性，随着时间的推移，顶层索引会越来越大。对于旧时间窗口的数据，可以看成历史数据，查询的需求通常较少。本文根据时间跨度对顶层索引进行分段，若当前顶层索引中的时间跨度超过阈值，则下次顶层索引更新时，会增加一个新的顶层索引，后续数据更新到新的顶层索引中，如图 5-7 所示。所有时间周期和顶层索引的映射关系都在协调节点中维护，查询时根据映射关系查找对应的顶层索引。若顶层索引所占内存过多，可以将分段中最旧的顶层索引存储在磁盘中，以减轻内存压力，

5.2 分布式索引持久化性能评估

5.1 小节详细介绍了 WB-Index 中流元组以及索引结构的持久化方法。本节将对 WB-Index 持久化性能进行详细分析。其中顶层索引常驻内存，存储开销小，定时的持久化以及更新时的预写日志都不存在性能问题。因此，本节主要评估流元组和底层索引的持久化性能，相关符号如表 5-1 所示。

万方数据

表 5-1 持久化符号表

Table 5-1. Some notations in persistence

符号	描述
T_{write}	分布式文件系统平均写入速度（单位：字节/秒）
T_M	读 / 写一次内存的时间（单位：秒）
S_{key}	流元组中“Key”值序列化后的大小（单位：字节）
S_{value}	流元组中“Value”值序列化后的大小（单位：字节）
S_{tuple}	流元组序列化后的大小（单位：字节）
T_W	单个时间窗口时长（单位：秒）
N_W	时间窗口内流元组数量（单位：个）
$N_{bloomfilter}$	布隆过滤器中位数组大小（单位：个）
N_{height}	底层 B+树索引树高（单位：层）
N_m	B+树节点的子节点数（单位：个）
$t_{additional}$	辅助索引构建时间开销（单位：秒）
$t_{persistBase}$	底层索引持久化时间开销（单位：秒）
$t_{persistTuple}$	流元组持久化时间开销（单位：秒）

5.2.1 流元组持久化性能评估

流元组的持久化过程集中在数据源对应的索引构建节点上。若一个窗口流元组的持久化时长大于时间窗口时长，流元组就会在构建节点的内存中堆积，影响底层索引的构建和流元组的摄入，使得系统无法稳定运转。针对一个时间窗口，流元组序列化后的总大小为：

$$S_{tuple} = (S_{key} + S_{value} + 8) \times N_W \tag{5-2}$$

故流元组的持久化时间 $t_{persistTuple}$ 为：

$$t_{persistTuple} = \frac{(S_{key} + S_{value} + 8) \times N_W}{T_{write}} \tag{5-3}$$

为了保证系统的稳定性，避免数据流的堆积，需要保证 $t_{persistTuple} < T_W$，即：

$$N_w < \frac{T_w \times T_{write}}{S_{key} + S_{value} + 8} \tag{5-4}$$

根据式子(5-4)可得出结论：针对一个时长为 T_W 的时间窗口，持久化阶段窗口内的流元组数 N_W 存在着上限 N_{max}，即 WB-Index 持久化方法所能支撑的最大流速为 $\frac{N_{max}}{T_W}$。

万方数据

5.2.2 底层索引持久化性能评估

底层索引持久化阶段需要先构建辅助索引。其中统计信息的构建只涉及到两次赋值操作，时间开销可以忽略不计，辅助索引构建时间开销主要来自布隆过滤器的构建，其时间开销 $t_{additional}$ 为：

$$t_{additional}=3\times N_w \times T_M \tag{5-5}$$

辅助索引结构的存储开销主要来自布隆过滤器中维护的位数组，若按照 10 倍的流元组数量初始化位数组，辅助索引的存储开销 $N_{addtitonal}$ 为：

$$N_{additional}=\frac{10\times N_w}{8} \tag{5-6}$$

B+树持久化时，以 B+树节点为单位，其中叶节点和内节点数量按 4.2 小节计算的近似值为准。叶子节点的持久化开销 N_{leaf} 为：

$$N_{leaf}=(S_{key}+8)\times N_W+8\times\frac{N_W}{m} \tag{5-7}$$

内节点的持久化开销 N_{inner} 为：

$$N_{inner}=(S_{key}+8)\times\frac{N_W}{m}+8\times\frac{N_W}{m^2} \tag{5-8}$$

通过式子(5-5)、(5-6)、(5-7)、(5-8)可得，底层索引持久化过程中，流元组“key”值是主要的存储开销。流元组 S_{value} 的量级远大于 S_{key}，故底层索引持久化后的大小要远小于流元组持久化后的大小。而且 $t_{additional}$ 的时间开销也非常有限，因而可以得出结论：WB-Index 持久化的时间开销主要集中在流元组持久化过程，底层索引持久化过程中不存在性能瓶颈。

5.2.3 查询性能评估

对于每个查询节点，当索引和数据持久化后，底层索引和获取流元组阶段都需从分布式文件系统中获取相关数据，与索引和数据缓存在内存中相比，会增加一定的 IO 开销。其中，获取流元组阶段仅涉及到一次 IO，且获取的数据量与查询所涉及的范围密切相关，时间开销较为确定。本小节主要评估了每个时间窗口对应的底层索引在查询过程中需要从分布式文件系统中获取的数据量。

若查询类型为点查询，首先会调取底层索引中的布隆过滤器，其大小为：

$$N_{additional}=\frac{10\times N_w}{8} \tag{5-9}$$

底层索引中的 B+树在查找过程中，按照 B+树节点大小为单位，故查找过程中需调取的节点总数据量 N_{tree} 为：

$$N_{tree}=K\left((S_{key}+8)\times m+8\right)+(N_{height}-1)\times\left((S_{key}+8)\times m+8\right) \tag{5-10}$$

其中 K 为查询范围所包含的叶节点数量。由式子(5-10)可得，节点大小 m 直接影响查询效率。过小的节点会导致树高增加，从而增加随机 IO 的次数；过大的节

万方数据

点会导致单次获取节点的数据量过大，浪费 IO 资源。在实际应用中，B+树树高通常为 3 层左右，以此来保持较好的查询效率。

5.3 实验评价

5.3.1 实验环境

本章的实验环境以及实验数据与第四章实验相一致。在集群中的 6 台存储节点上搭建分布式文件系统用于存储数据流和索引。实验中分布式文件系统采用 HDFS，选择的版本为 2.7.3。实验过程中，底层索引中布隆过滤器位数组大小与流元组个数的比值设为 10，Hash 函数个数设置为 3 个。

5.3.2 持久化性能

WB-Index 持久化方案中包含了流元组、底层索引和顶层索引的持久化方法。其中顶层索引常驻内存，其存储开销小且更新频率低，定时的持久化以及更新时的预写日志都不存在性能问题。本实验主要评估流元组以及底层索引的持久化性能。实验中的时间窗口设为 20 秒，通过改变时间窗口内流元组数量来模拟不同流速的数据流。对比不同流速下，一个时间窗口中底层索引和流元组的持久化时间开销。

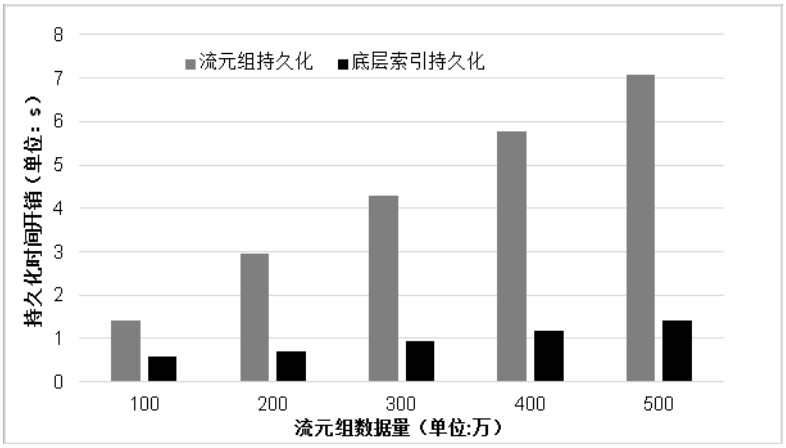

图 5-8 不同流元组数量下底层索引和流元组持久化时间对比

Figure 5-8. The comparison of persistence time between stream tuples and bottom index

实验结果如图 5-8 所示，在同一数据量下，底层索引的持久化时长远小于流元组的持久化时长，且底层索引的持久化时长随着数据量的增加变化幅度较小。持久化时间开销与写入数据量密切相关，当一个时间窗口内的数据量达到 500 万时，流元组持久化后的大小为 500MB 左右，而底层索引持久化后的大小仅为 50MB 左右。相比之下，分布式索引的持久化性能瓶颈主要在于流元组的持久化阶段，这也符合 5.2 小节得出的结论。

万方数据

5.1.2 小节提出，若一个时间窗口内的流元组数量过大，可以采用并行持久化的方法来提高持久化性能。本实验在不同数据量下，分析了持久化并行度对流元组持久化性能的影响。

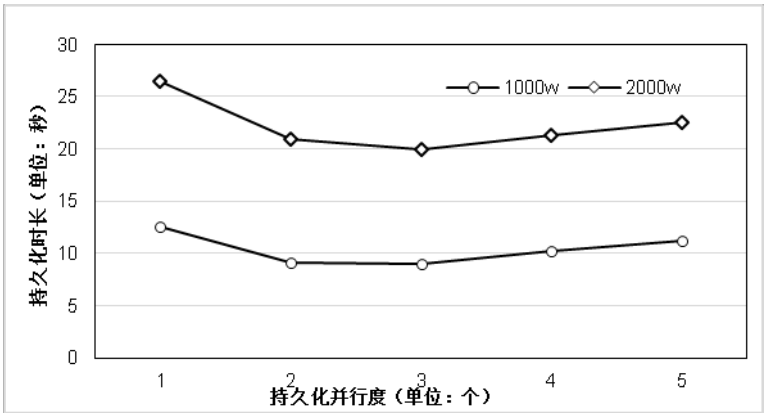

图 5-9 流元组持久化时间开销与持久化并行度的关系

Figure 5-9. The persistence time of stream tuple vs. persistence parallelism

实验结果如图 5-9 所示，随着持久化并行度的增加，流元组的持久化时间先减少后增加。当数据量为 1000 万和 2000 万时，选取 3 并行度可使得持久化性能最佳。分布式文件系统的写入性能与集群中的总带宽密切相关，随着写入并行度的提高，集群中的带宽利用率提高，整体的写入性能也会提高。但当持久化并行度提高后，写入文件的数量也会增加，出现随机写的概率也会提高，且集群中各节点间的带宽是有限的，过多的持久化并行度反而会导致持久化性能下降，故需根据数据量设置合理的持久化并行度。

5.3.3 查询性能

当流元组和索引存储在分布式文件系统中时，查询请求会增加多次 IO 开销和节点间数据传输开销。本实验模拟了针对数据流的历史数据查询场景，通过改变查询请求涉及的时间窗口以及查询范围涉及到的流元组比例，评估分布式索引在持久化后的查询性能。实验中每个时间窗口的流元组数据量为 100 万。

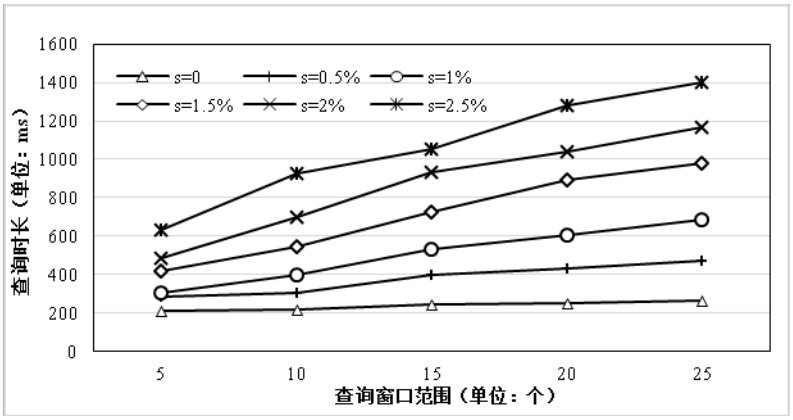

图 5-10 持久化后 WB-Index 查询范围与 WB-Index 查询时间开销的关系

Figure 5-10. The query time of WB-Index after persistence vs. the query range of WB-Index

万方数据

实验结果如图 5-10 所示，其中 s 为在每个时间窗口内范围查询所涉及的流元组占比。当底层索引和流元组持久化后，分布式查询性能比起底层索引和流元组全部缓存在内存中的情况有了一定的下降，但下降的幅度不大，这得益于 B+树的层高较低，查询过程中的随机 IO 次数较少，流元组获取过程也只涉及到一次 IO 过程。此外，针对不同时间窗口的查询完全并行化，整体的查询性能能够满足对历史数据的查询需求。

流元组和索引持久化后，分布式查询的差异主要在于底层索引查找和数据流元组获取这两个阶段。本实验针对一个时间窗口，通过改变查询范围占比，对比这两个阶段在持久化前后的查询性能差异。实验中每个时间窗口的流元组数据量为 100 万。

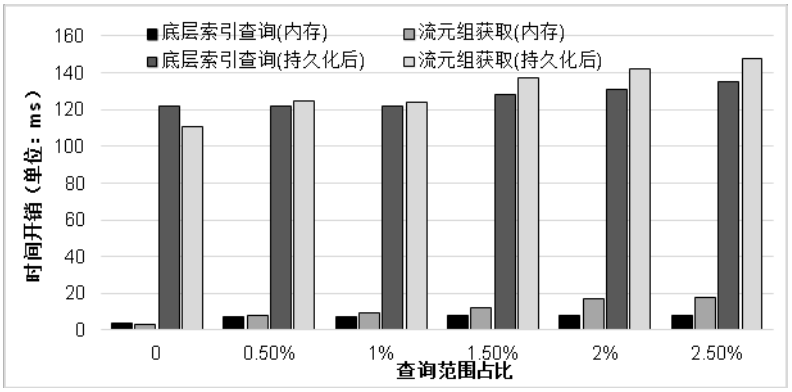

图 5-11　持久化前后底层索引查询和流元组获取时间开销对比

Figure 5-11. The comparison of time cost between bottom index query and stream tuple acquisition before and after persistence

实验结果如图 5-11 所示，持久化前后的底层索引查询性能和流元组获取性能存在明显差距。其中，底层索引查询阶段的性能差距最为明显，底层索引缓存在内存中时，查询的时间开销小，可忽略不计，且查询时间不会随着查询范围的增加而显著增加，具有良好的稳定性。而当底层索引持久化后，查询的过程会涉及到多次的随机 IO，会产生一定的时间开销。这也证明将更多的底层索引缓存能有效地提高查询效率。流元组获取阶段，平均 IO 速度是造成持久化前后查询性能差距的主要因素。当流元组缓存在索引构建节点时，获取流元组的平均 IO 速度与节点间的网络带宽相近，能够达到 150MB/s；当流元组全部存储在分布式文件系统中时，实验测试得出的平均 IO 速度为 80MB/s。另外，从分布式文件系统中获取数据时，都需要通过主节点定位对应的文件，这使得在读取少量数据时，依然存在一定的时间开销。

WB-Index 系统架构中，通过设计三类流元组缓存来提高查询效率，分别为基于查询结果的缓存、基于查询区间内的流元组缓存和基于较新时间窗口的流元组缓存。本实验通过改变查询涉及的窗口数量，对比命中不同类别缓存和未

万方数据

命中缓存时的查询时间开销，来评估流元组缓存对查询性能的影响。实验中模拟的时间窗口数据量为 100 万，窗口中查询范围占比为 1%。

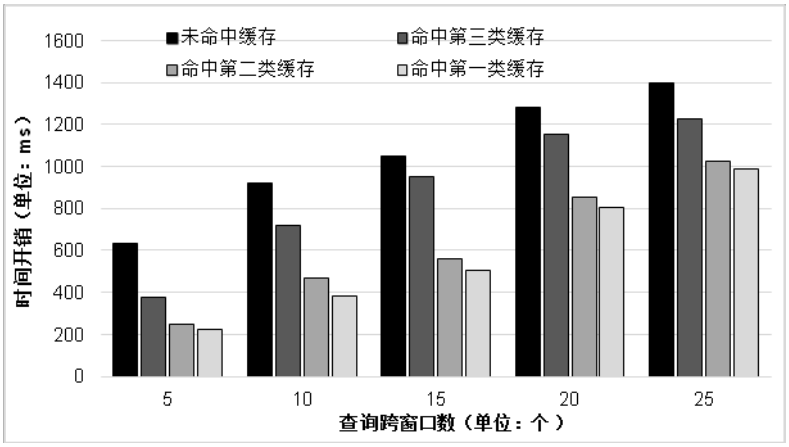

图 5-12　不同查询范围下 WB-Index 查询时间开销与命中缓存种类的关系

Figure 5-12. The query time of WB-Index vs. the type of hit cache in different query ranges

实验结果如图 5-12 所示，命中缓存能有效地提高查询效率，且命中层级越高，查询效率也越高。当命中第一类缓存时，相当于减少了整个分布式查询过程，直接返回查询结果，故查询性能达到最优。其余两类缓存旨在降低数据获取阶段的时间开销。当命中第二类缓存时，相当于将获取流元组数据的时间开销降为零；当命中第三类缓存时，相当于提高获取流元组的 IO 速度。实验结果充分证明了流元组缓存的有效性。

5.4 本章总结

本章根据 WB-Index 索引结构，分别为流元组、底层索引和顶层索引设计持久化方案。对于流元组，通过设计紧凑并支持压缩的存储格式来减少存储开销。针对数据量较大的时间窗口，提出并行持久化方法，充分利用带宽来提高持久化的性能；对于底层索引，将其中的 B+树、布隆过滤器、统计信息等分模块存储，提出支持合并的存储格式，以减少小文件对分布式文件系统的影响；对于顶层索引，采用定期持久化和预写日志相结合的方式来保证顶层索引的可靠性，并对顶层索引在时间维度进行分段，实现新旧索引的分离，从而提高了查询效率。最后评估了持久化方法的性能，通过实验证明了持久化方案的可行性。

万方数据

第六章 结论与展望

6.1 结论

本文研究了数据流存储、索引等相关问题，在面向数据流的分布式索引构建、实时存储、查询等方面进行了探索。数据流具有实时性、高速性、无限性等特点，现有的一些分布式索引无法很好地应用于数据流场景。对此，本文提出了一种适用于数据流的双层主从分布式索引结构：WB-Index。该结构采用时间窗口机制，在时间维度对数据流进行切分，将流处理转换成微批量处理，有效地提高了数据流的处理效率。底层索引是针对时间窗口内数据流元组的索引，其均匀的分散在各个节点中，相当于局部索引。顶层索引是针对各时间窗口的索引，相当于全局索引。查询时，根据查询条件，先利用顶层索引定位到相应的底层索引，再并行的查找底层索引来获取查询结果。WB-Index 系统架构中由索引构建节点、查询节点、协调节点、主控节点、存储节点这 5 类节点构成，将流元组存储、索引构建和查询请求分离，从而更高效地支撑数据流的存储和查询。

索引构建方面，对于底层索引中的 B+树索引结构，本文提出了基于并行排序的预装载批量构建法，该方法构建速度快，能够支撑高速数据流。针对顶层 B+树索引，根据其码值递增的特殊性，提出了预分配空间的不分裂 B+树更新方法。该方法在提高 B+树更新效率的同时也提高了存储效率。底层索引构建完成后，为了保证索引构建节点的构建效率，需将底层索引发布到查询节点来提供查询服务。本文根据查询节点状态，动态地调节各查询节点权重，并利用加权轮询策略来保证查询节点的负载均衡，从而提高索引构建和查询效率。

为了更好地缓存索引结构，WB-Index 索引方案将索引结构和流元组相分离。考虑到流元组的时效性和热点性，本文共提出 3 类不同层面的流元组缓存，有效地提高流元组的获取效率。

由于数据流具有无限性，本文利用分布式文件系统分别存储流元组和索引结构。对于流元组，按照其“Key”值顺序存储，并设计紧凑的存储格式来减少存储开销；底层索引采用模块化的存储结构，考虑到小文件对分布式文件系统的影响，底层索引存储格式能支持动态合并；顶层索引量级小且更新频率低，通过常驻内存的方式提高查询效率，采用定期持久化和预写日志相结合的方式来确保顶层索引的可靠性。

万方数据

总之，本文所提的 WB-Index 索引方案，不仅能够支持数据流的实时存储，还能保证新数据流的实时查询以及历史数据流的高效查询。

6.2 展望

数据流具有广泛的应用场景和极高的应用价值。通过优化数据流的处理、存储、查询和挖掘等方法，来最大化的发挥数据流价值是当前也是未来长期的研究热点。

本文提出的 WB-Index 索引结构中，底层索引是根据流元组码值构建的。在不同的查询场景中，往往需要支持多种查询请求。例如：对于包含空间属性的流元组，查询时往往涉及到空间范围的查询，故需要根据空间属性构建 R 树索引以支持空间查询。在大数据场景下，如何将不同类型的索引高效结合将是一大挑战，后续将在这一方面进一步研究。

本文提出的 WB-Index 索引中的顶层索引是根据时间维度构建的，搜索时需根据时间范围定位到对应的底层索引，对于没有时间范围条件的查询，将会进行大规模地并行查询以返回查询结果，如何提高这种场景下的查询效率将是后续的研究重点。例如可以结合应用场景，将更多时间窗口中的特征纪录在顶层索引中，也可以通过构建多种维度的顶层索引来提高不同查询场景下的查询效率。

万方数据

参考文献

[1] 金澈清, 钱卫宁, 周傲英. 流数据分析与管理综述[J]. 软件学报, 2004, 15(8): 1172-1181.

[2] Krawczyk B, Minku L L, Gama J, et al. Ensemble learning for data stream analysis: A survey[J]. Information Fusion, 2017, 37: 132-156.

[3] Garofalakis, Minos, Johannes Gehrke, et al. Data Stream Management: Processing High-Speed Data Streams[M]. Berlin: Springer, 2016.

[4] 孙大为, 张广艳, 郑纬民. 大数据流式计算: 关键技术及系统实例[J]. 软件学报, 2014, 25(4): 839-862.

[5] Babcock B, Babu S, Datar M, et al. Models and issues in data stream systems[C]. //Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. Wisconsin: ACM, 2002: 1-16.

[6] Abadi D J, Carney D, Çetintemel U, et al. Aurora: a new model and architecture for data stream management[J]. the VLDB Journal, 2003, 12(2): 120-139.

[7] Krishnamurthy S, Chandrasekaran S, Cooper O, et al. TelegraphCQ: An architectural status report[J]. IEEE Data Eng. Bull., 2003, 26(1): 11-18.

[8] Chandrasekaran S, Cooper O, Deshpande A, et al. Telegraphcq: Continuous dataflow processing for an Uncertain world[C]. //Proceedings of the 2003 ACM SIGMOD international conference on Management of data. California: ACM, 2003: 668-668.

[9] Arasu A, Babcock B, Babu S, et al. Stream: The stanford data stream management system[M]. Data Stream Management. Berlin: Springer, 2016.

[10] 张鹏, 李鹏霄, 任彦, 等. 面向大数据的分布式流处理技术综述[J]. 计算机研究与发展, 2014, 51(s2): 1-9.

[11] Toshniwal A, Taneja S, Shukla A, et al. Storm@ twitter[C]. //Proceedings of the 2014 ACM SIGMOD international conference on Management of data. Utah: ACM, 2014: 147-156.

[12] Neumeyer L, Robbins B, Nair A, et al. S4: Distributed stream computing platform[C]. //2010 IEEE International Conference on Data Mining Workshops. Sydney: IEEE, 2010: 170-177.

[13] Zaharia M, Das T, Li H, et al. Discretized streams: Fault-tolerant streaming computation at scale[C]. //Proceedings of the twenty-fourth ACM symposium on operating systems principles. Pennsylvania: ACM, 2013: 423-438.

[14] Zaharia M. An architecture for fast and general data processing on large clusters[M]. California: Morgan & Claypool, 2016.

[15] Carbone P, Fora G, Ewen S, et al. Lightweight Asynchronous Snapshots for Distributed Dataflows[J]. arXiv: Distributed, Parallel, and Cluster Computing, 2015.

[16] Yang F, Tschetter E, Léauté X, et al. Druid: A real-time analytical data store[C]. //Proceedings of the 2014 ACM SIGMOD international conference on Management of data. Utah: ACM, 2014: 157-168.

[17] Suna T. Opentsdb[EB/OL]. http://opentsdb.net/, 2017-08-23.

[18] 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.

万方数据

[19] Pu K Q, Zhu Y. Efficient indexing of heterogeneous data streams with automatic performance configurations[C]. //19th International Conference on Scientific and Statistical Database Management. DC: IEEE, 2007: 34-34.

[20] Fagin R, Nievergelt J, Pippenger N, et al. Extendible hashing—a fast access method for dynamic files [J]. ACM Transactions on Database Systems (TODS), 1979, 4(3): 315-344.

[21] Litwin W. Linear Hashing: a new tool for file and table addressing[C]. //Proceedings of the sixth international conference on Very Large Data Bases. Montreal: VLDB Endowment, 1980: 1-3.

[22] Bloom B H. Space/time trade-offs in hash coding with allowable errors[J]. Communications of The ACM, 1970, 13(7): 422-426.

[23] Fan B, Andersen D G, Kaminsky M, et al. Cuckoo filter: Practically better than bloom[C]. //Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies. Sydney: ACM, 2014: 75-88.

[24] Aho A V, Hopcroft J E, Ullman J D. The design and analysis of computer algorithms[M]. MA: Addison-Wesley, 1987.

[25] Guibas L J, Sedgewick R. A dichromatic framework for balanced trees[C]. //19th Annual Symposium on Foundations of Computer Science. Michigan: foundations of computer science, 1978: 8-21.

[26] Comer D. The Ubiquitous B-Tree [J]. ACM Computing Surveys (CSUR), 1979, 11(2): 121-137.

[27] Guttman A. R-trees: a dynamic index structure for spatial searching[C]. //Proceedings of the 1984 ACM SIGMOD international conference on Management of data[C]. NY: ACM, 1984: 47-57.

[28] Lu H, Ng Y Y, Tian Z. T-tree or b-tree: Main memory database index structure revisited[C]. //Proceedings 11th Australasian Database Conference. Canberra: IEEE, 2000: 65-73.

[29] Yu C, Boyd J. FB+-tree: indexing based on key ranges[C]. //Proceedings of the 11th IEEE International Conference on Networking, Sensing and Control. US: IEEE, 2014: 438-444.

[30] Yu C, Boyd J. FB+-tree for Big Data Management[J]. Big Data Research, 2016, 4: 25-36.

[31] Rao J, Ross K A. Making B+-trees cache conscious in main memory[C]. //Proceedings of the 2000 ACM SIGMOD international conference on Management of data. NY, ACM, 2000: 475-486.

[32] Yan Z, Lin Y, Peng L, et al. Harmonia: a high throughput B+ tree for GPUs[C]. //Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. ACM, 2019: 133-144.

[33] Bercken J, Seeger B. An evaluation of generic bulk loading techniques[C]. //Proceedings of the 27th International Conference on Very Large Data Bases. CA: Morgan Kaufmann Publishers Inc, 2001: 461-470.

[34] Kotidis Y, Roussopoulos N. An alternative storage organization for ROLAP aggregate views based on cubetrees [C]. //Proceedings of the 1998 ACM SIGMOD international conference on Management of data. Washington: ACM, 1998: 249-258.

[35] Lo M L, Ravishankar C V. The design and implementation of seeded trees: An efficient method for spatial joins[J]. IEEE Transactions on Knowledge and Data Engineering, 1998, 10(1): 136-152.

[36] Ngu H C V, Huh J H. B+-tree construction on massive data with Hadoop[J]. Cluster Computing, 2017: 1-11.

万方数据

[37] Mazumdar P, Wang L, Winslet M, et al. An index scheme for fast data stream to distributed append-only store[C]. //Proceedings of the 19th International Workshop on Web and Databases. San Francisco: ACM, 2016: 5-5.

[38] Cai R, Lu Z, Wang L, et al. Ditir: Distributed index for high throughput trajectory insertion and real-time temporal range query[J]. Proceedings of the VLDB Endowment, 2017, 10(12): 1865-1868.

[39] Ghemawat S, Gobioff H, Leung S, et al. The Google file system[J]. symposium on operating systems principles, 2003, 37(5): 29-43.

[40] 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.

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

[42] Rosenblum M, Ousterhout J K. The design and implementation of a log-structured file system[J]. Kluwer Academic Publishers, 1995, 25(1):1-15.

[43] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of The ACM, 2008, 51(1): 107-113.

[44] Melnik S, Gubarev A, Long J J, et al. Dremel: interactive analysis of web-scale datasets[J]. Proceedings of the VLDB Endowment, 2010, 3(1): 330-339.

[45] Ananthanarayanan G, Ghodsi A, Warfield A, et al. PACMan: Coordinated Memory Caching for Parallel Jobs[C]. //Proceeding NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. CA:USENIX Association, 2012: 20-20.

[46] Lehman T J, Carey M J. A study of index structures for main memory database management systems[C]. //Proceedings of the 12th International Conference on Very Large Data Bases. CA: Morgan Kaufmann Publishers Inc, 1986: 294.

[47] Carlson J L. Redis in action[M]. Manning Publications Co., 2013.

[48] Li H, Ghodsi A, Zaharia M, et al. Tachyon: Reliable, memory speed storage for cluster computing frameworks[C]. //Proceedings of the ACM Symposium on Cloud Computing. WA: ACM, 2014: 1-15.

[49] Stonebraker M, Weisberg A. The Volt DB Main Memory DBMS [J]. IEEE Data Eng. Bull., 2013, 36(2): 21-27.

[50] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing[C]. //Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. CA: USENIX Association, 2012: 2-2.

[51] Karger D, Lehman E, Leighton T, et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web[C]. //Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. Texas: ACM, 1997: 654-663.

[52] Van Renesse R, Dumitriu D, Gough V, et al. Efficient reconciliation and flow control for anti-entropy protocols[C]. //proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware. NY: ACM, 2008: 6.

[53] Aguilera M K, Golab W, Shah M A. A practical scalable distributed b-tree[J]. Proceedings of the VLDB Endowment, 2008, 1(1): 598-609.

[54] Mitchell C, Montgomery K, Nelson L, et al. Balancing CPU and network in the cell distributed B-tree store[C]. CA:usenix annual technical conference, 2016: 451-464.

万方数据

[55] Huang B, Peng Y X. An Efficient Distributed B-tree Index Method in Cloud Computing[J]. Open Cybernetics & Systemics Journal, 2014, 8: 302-308.
[56] Wu S, Jiang D, Ooi B C, et al. Efficient B-tree based indexing for cloud data processing[J]. Proceedings of the VLDB Endowment, 2010, 3(1-2): 1207-1218.
[57] Li X, Ren C, Yue M. A Distributed Real-time Database Index Algorithm Based on B+ Tree and Consistent Hashing[J]. Procedia Engineering, 2011, 24: 171-176.
[58] Melnik S, Raghavan S, Yang B, et al. Building a distributed full-text index for the Web[J]. international world wide web conferences, 2001: 396-406.
[59] Nguyen T T, Nguyen M H. Forest of Distributed B+Tree Based on Key-Value Store for Big-Set Problem[C]. //International Conference on Database Systems for Advanced Applications. TX: Springer International Publishing, 2016: 268-282.
[60] 何龙, 陈晋川, 杜小勇. 一种面向 HDFS 的多层索引技术[J]. 软件学报, 2017, 28(03): 502-513.
[61] Richter S, Quiané-Ruiz J A, Schuh S, et al. Towards zero-overhead static and adaptive indexing in Hadoop[J]. The VLDB Journal—The International Journal on Very Large Data Bases, 2014, 23(3): 469-494.
[62] 李超, 张明博, 邢春晓, 胡劲松. 列存储数据库关键技术综述[J]. 计算机科学, 2010, 37(12): 1-7+17.
[63] Abadi D J, Madden S R, Hachem N. Column-stores vs. row-stores: how different are they really?[C]. //Proceedings of the 2008 ACM SIGMOD international conference on Management of data. Vancouver: ACM, 2008: 967-980.
[64] He Y, Lee R, Huai Y, et al. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems[C]. //Proceedings of the 27th International Conference on Data Engineering. Hannover, IEEE, 2011: 1199-1208.
[65] Huai Y, Chauhan A, Gates A, et al. Major technical advancements in apache hive[C]. //Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. Utah: ACM, 2014: 1235-1246
[66] Floratou A, Patel J M, Shekita E J, et al. Column-oriented storage techniques for MapReduce[J]. Proceedings of the VLDB Endowment, 2011, 4(7): 419-429.
[67] Hunt P D, Konar M, Junqueira F, et al. ZooKeeper: wait-free coordination for internet-scale systems[C]. //usenix annual technical conference. MA: USENIX Association, 2010: 11-11.
[68] Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms[M]. MA: MIT press, 2009.

万方数据

致　谢

时光飞逝，岁月如梭，三年的研究生学习生涯即将画上句号。回首过去的三年，收获颇丰。值此论文完成之际，我要向帮助过我的师长和朋友表示感谢。

首先，衷心感谢我的导师，杨良怀教授。杨老师治学严谨、温文尔雅、为人谦和，具有高深的学术造诣。早在本科的时候，我就上过杨老师数据库相关的专业课，杨老师在专业领域知识的深度以及认真严谨的教学态度另我非常崇拜，通过这门课我收获了很多数据库相关的知识，也坚定了自己未来努力的方向。当我拿到学校保研资格时，我毫不犹豫地选择杨老师作为研究生导师，很荣幸能成为杨老师的学生。在过去的三年里，杨老师给了我们明确的目标，要求我们能够在理论深度、动手能力等多方面有综合提升。杨老师指导我们参加有挑战性的课题，在论文选题、资料收集和论文写作阶段，都耐心的指导并给予了我很大的帮助，研究生期间的完成的小论文、专利、毕业论文，杨老师都是逐字逐句的帮忙修改，提出很多中肯的指导意见。这么长时间来杨老师给予的关心和教导，我将永远铭记于心。

然后，也要衷心的感谢范玉雷老师，在课题研究过程中，范老师耐心的解答我在专业知识上的问题，给予了我很多建议，让我的研究工作能够顺利进展。每次和范老师的讨论，我都能收获很多。

另外，要感谢我的朋友、实验室的师兄师姐和师弟师妹们。我们大家一起营造了一个良好的实验室环境。在我平时的学习生活中，大家给予了我很大的帮助。当然，我还要感谢我的父母，感谢多年的养育和支持，让我能够有这样一个学习机会。

最后，我要感谢参与我论文评审和答辩的各位老师。论文的评审和答辩是对我过去三年学习时光的总结，也让我发现了其中存在的一些问题，对于我来说，这是非常珍贵的财富。我将谨记各位老师们的教诲，努力工作，回报社会。

万方数据

作者简介

1. 作者简历

1994 年 06 月出生于浙江台州。

2016 年 09 月——2019 年 06 月，在浙江工业大学计算机科学与技术学院计算机科学与技术专业学习，攻读计算机科学与技术硕士学位。

2. 攻读硕士学位期间发表的学术论文

[1] 面向大数据流的高性能内存 B+树构建. 软件学报. 2018（已录用，第二作者，导师一作）

3. 参与的科研项目及获奖情况

[1] 渔船大数据分析项目, 企业委托项目. 编号: KYY-HX-20180242

[2] 数据集设计器开发项目，企业委托项目. 编号: KYY-HX-20170269

4. 发明专利

[1] 一种面向多源大数据流的分布式索引方法，申请号：201810630231.9 （已受理，第二作者，导师一作）

万方数据

学位论文数据集

密　级*	中图分类号*	UDC*	论文资助
公开	TP31	004	
学位授予单位名称	学位授予单位代码	学位类型*	学位级别*
浙江工业大学	10337	工学硕士	硕士
论文题名*	面向大数据流的分布式 B+树索引构建		
关键词*	数据流，B+树，分布式索引，分布式存储		论文语种*
并列题名*	无		中文
作者姓名*	卢晨曦	学　号*	2111612070
培养单位名称*	培养单位代码*	培养单位地址	邮政编码
浙江工业大学计算机科学与技术学院	10337	杭州市留和路 288 号	310032
学科专业*	研究方向*	学　制*	学位授予年*
计算机科学与技术	数据流存储	3 年	2019
论文提交日期*	2019. 06		
导师姓名*	杨良怀	职　称*	教授
评阅人	答辩委员会主席*	答辩委员会成员	
	黄德才		
电子版论文提交格式：文本（　）图像（　）视频（　）音频（　）多媒体（　）其他（　）			
电子版论文出版（发布）者	电子版论文出版（发布）地	版权声明	
论文总页数*	68		
注：共 33 项，其中带*为必填数据，为 22 项。			

万方数据