分布式数据流计算系统的数据缓存技术综述

袁旭初，付国，毕继泽，张岩峰，聂铁铮，谷峪，鲍玉斌，于戈
东北大学计算机科学与工程学院，辽宁 沈阳 110169

摘要
数据流编程模型以其高度并行计算、支持流水线处理、支持函数式编程等优点被许多主流的计算系统采用。在分布式数据流系统和异构数据流系统中，算子之间数据生产和数据消化的速度不一致可能会导致数据堆积或者算子闲置等问题。为支持高效的数据流系统，需要设计缓存系统，以保证数据流的高效缓存和移动。选取了几个典型的分布式数据流系统与分布式消息队列系统进行系统分析，并总结了目前消息队列系统对数据流缓存系统的支持程度。最后对数据缓存技术进行了阐述，并分析了未来的数据流缓存系统的需求和研究方向。

关键词
数据流；缓存；分布式系统；消息队列

中图分类号：TP391 文献标识码：A doi: 10.11959/j.issn.2096-0271.2020027

Survey on data caching technology of distributed dataflow system

YUAN Xuchu, FU Guo, BI Jize, ZHANG Yanfeng, NIE Tiezheng, GU Yu, BAO Yubin, YU Ge
College of Computer Science and Engineering, Northeastern University, Shenyang 110169, China

Abstract
Dataflow model is adopted by several dataflow systems for its advantages of high parallel computing, pipeline processing and functional programming. In distributed dataflow systems and heterogeneous dataflow systems, due to the speed mismatch between the data production of data source operators and the data consumption of data sink operators, data could be delayed and operators could be idle. In order to support an efficient dataflow system, a dataflow cache system was desired to ensure efficient caching and movement of dataflow. Several distributed dataflow systems and distributed message queuing systems were analyzed, and the support degree of current message queuing system to data flow caching system was summarized. Finally, the cache technique was introduced, and the demands and research directions of future dataflow caching systems were analyzed.

Key words
dataflow, cache, distributed system, message queue

1 引言

计算机的计算模型可以分为控制流[1-2]和数据流[3-6]两大类。控制流计算模型按指令的顺序驱动操作，计算机内的数据是否参加运算依赖于当时执行的指令。图灵机理论是控制流计算机的基础，控制流计算机也被称为冯·诺伊曼型计算机，它是主流计算机一直采用的基本体系结构。控制流天然擅长描述控制逻辑，但其使用变量缓存中间结果，不利于并行或异构计算抽象。数据流计算模型采用数据驱动方式，只有当一条或一组指令需要的操作数全部准备好时，才能激发相应指令的一次执行，执行结果又流向等待这一数据的下一条或一组指令，以驱动该条或该组指令的执行。指令之间天然的依赖关系决定了指令的执行顺序，指令按照数据流图执行。

数据流计算模型在许多方面优于控制流计算模型，其优点主要体现在以下3个方面。

（1）高度并行计算

在数据流方法中，由于没有指令执行顺序的限制，从理论上来说，指令执行更加灵活，通过系统优化可以获得最大的并行性。相似地，其灵活性同样适用于高度异构计算。

（2）支持流水线处理

由于在指令中直接使用数值本身，而不是使用存放数值的地址，因此可以在过程级及指令级充分开发异步并行性，把串行执行算子实际处理的数据变成一条异步处理流水线，即前一个算子处理完部分结果后就让后一个算子开始处理。

（3）函数式编程

面向数据流的编程模型对丰富的算子进行了抽象，通过用户定义函数为算子指定用户处理逻辑，用户无须使用变量维护中间状态，实现优化空间巨大且灵活的函数式编程。

目前许多主流的计算系统（如Spark[7-9]、Flink[10-11]、TensorFlow[12-14]、Google Dataflow[15]等）采用了数据流编程模型。一个数据流程序中一般包含算子和中间结果数据两大类元素，算子还包含数据源算子（source）、数据池算子（sink）、转换算子（transformer）。数据源是数据的生产者，如文件或者视频采集设备；数据池指定程序输出的位置，如文件或者数据库；转换算子是系统提供或者用户自定义的数据操作集合，描述对一个或多个输入数据的处理过程，同时输出一个或多个中间结果数据。中间结果数据位于各算子之间，是由算子产生或供算子消费的数据。数据流处理程序采用算子连接数据流的模式，当一个数据流程序被执行的时候，它会被映射为一个有向无环图（directed acyclic graph，DAG），数据流图的顶点为算子，数据流图的边为中间结果数据。程序启动时从一个或多个数据源算子开始，结束于一个或多个数据池算子。

数据流模型不仅被应用于内存计算中，也被应用到分布式集群（如Spark）或者异构计算环境（如TensorFlow）中，算子可能被设计为跨多台机器的分布式算子，有些算子在CPU执行，有些算子在GPU执行，甚至是跨CPU-GPU执行。而算子之间数据的流动需要考虑跨网络或者跨CPU-GPU的情况，数据流的维护和管理也不仅在内存中完成。在这种分布式数据流系统和异构数据流系统中，算子和数据不再统一存在于单机内存中。算子之间数据生产和数据消化的速度不一致可能会导致数据堆积或者算子闲置等问题，造成严重的空间开销，影响数据流系统的效率。为了支持高效的数据流系统，需要为分布

式数据流系统和异构数据流系统设计数据流的缓存系统，以保证数据流在分布式计算节点之间或者异构CPU-GPU之间的高效缓存和移动。然而，目前并没有针对分布式或异构数据流系统的通用数据流缓存系统。

现有的消息队列（message queue，MQ）系统（如Kafka[16-19]、Pulsar、RabbitMQ[20-21]等）常被用作数据源算子的数据缓存系统，特别是为视频采集设备这种主动推送数据源提供数据缓冲支持。这些系统利用优化的分布式存储将数据消息存到保持数据有序性的消息队列中，可以在一定程度上满足缓存需求。

本文选取Kafka、RabbitMQ、ActiveMQ[22-23]、RocketMQ[24-25]、Pulsar 5个典型的分布式消息队列系统进行系统分析，并分析未来的数据流缓存系统的需求和研究方向。

2 分布式数据流计算系统概述

2.1 数据流计算模型

计算模型是对计算任务完成过程的一种抽象描述，主要由3个部分组成：计算任务的描述方法、计算任务的执行机构以及计算任务在执行机构上的运行方法。根据计算任务的描述形式，可以将计算模型分为控制流计算模型和数据流计算模型。在控制流计算模型中，采用控制流的方式描述计算任务。控制流即以控制驱动程序。以下面包含控制流概念的代码1为例，由于控制条件的存在，无论输入是多少，总是执行被控制的部分，而不执行另一部分。在控制流计算模型中，进行数据传递的关键是借助变量保存中间状态。通过中间变量，可以根据任务的执行逻辑将其划分为不同的阶段，这样一来，每个阶段只需要完成一部分逻辑子功能即可。

代码1：控制流代码样例

```
Input: arg
Output: result
if arg > MAX then
    result = arg
else
    result = -arg
return result
```

将代码1的控制逻辑用数据流的方式表示，如代码2所示，代码的执行逻辑以流水线的方式按顺序执行。不论是否满足条件，均执行相应代码，只不过数据总是只满足一种情况，最后将两部分的结果做交集。如果上游输入数据不断到来，这段代码便可以不断地执行下去，并且总是同时执行真（ture）和假（false）的分支逻辑，但是无论何时，总有一个分支上的流水线的数据集为空。

代码2：数据流代码样例

```
Input: arg
Output: result
turearg = arg.filter(>max)
resultA = turearg
falsearg = arg.filter(≤max)
resultB = -falsearg
result = resultA.union(resultB)
return result
```

在数据流计算模型中，用数据流图的形式表示计算任务。根据任务中不同子任务的依赖关系将其转化为数据流图，复杂的程序逻辑便可以容易地以流水线的方式执行，同时提高执行效率。数据流编程模型是以数据驱动程序的，一个处理逻辑的输出作为下一个处理逻辑的输入，无须维护数据的中间状态，将这种处理逻辑抽象为算子，通常不同算子之间的任务相互独立，可以在不同的线程上执行。在分布

式或异构的环境下，算子也可以在不同的机器或容器内执行。只要数据到达，算子即可开始处理，从而使得各个算子形成流水线的结构，数据则在流水线中被并行处理，这种处理方式在处理具有复杂依赖关系的程序逻辑时有天然的优势。

在数据流图中，用节点和边描述程序逻辑。其中，节点表示操作，即数据流的逻辑计算单元，有向边表示数据依赖关系。数据流计算模型的核心思想是用数据控制计算。当一个操作所需的数据全部准备完毕之后，便可以启动运算。当只有部分数据到达时，则需要等待。当一个操作执行完成并将结果传递给下一个操作后，无论下一个操作是否能正常执行，这个操作都可以立刻对新数据进行计算。如此，整个程序便可以以流水线的方式并行执行。**图1**显示了在数据流系统Spark中分别对2组数据进行映射（map）和过滤（filter）之后再进行连接（join）的执行过程，弹性分布式数据集（RDD）表示Spark中的基本数据集。首先，数据集1（RDD1）和数据集2（RDD2）准备完毕，并被输入计算节点中，分别执行映射和过滤操作，这两步没有相互依赖的关系，也没有执行先后之分。然后，当连接算子的2个操作数都准备完毕后，即数据集3（RDD3）和数据集4（RDD4）已经计算得出时，执行连接操作。最后，计算出结果，数据向下一个计算单元传输。在连接操作进行的同时，如果有新的映射或过滤操作数到达，映射操作或过滤操作可以同时执行。如此，数据流图中多个计算节点便可以以流水线的方式并行执行。当一个程序有多个这样的计算过程时，它们之间也可以以流水线的方式并行执行。

传统的计算机采用控制流作为计算机的核心，即冯·诺伊曼体系结构，它通过一个中央处理器执行计算任务，用程序计数器根据程序控制逻辑控制指令依次执行。数据流的体系结构不同于传统的冯·诺伊曼体系结构，它以数据为驱动，数据在程序运行过程中起主导作用，这对于计算机发展来说是一个突破。针对数据流计算机的具体设计方案有很多，学术界和工业界也相继成功研制出一些专用机。以全新的体系结构设计出的数据流计算机不再需要CPU[26]，而是把功能分散到各个部件中，取消了程序计数器，以数据是否到达异步控制每一条指令的执行，这样更容易实现数据的并行。但这种新型的体系结构仅适

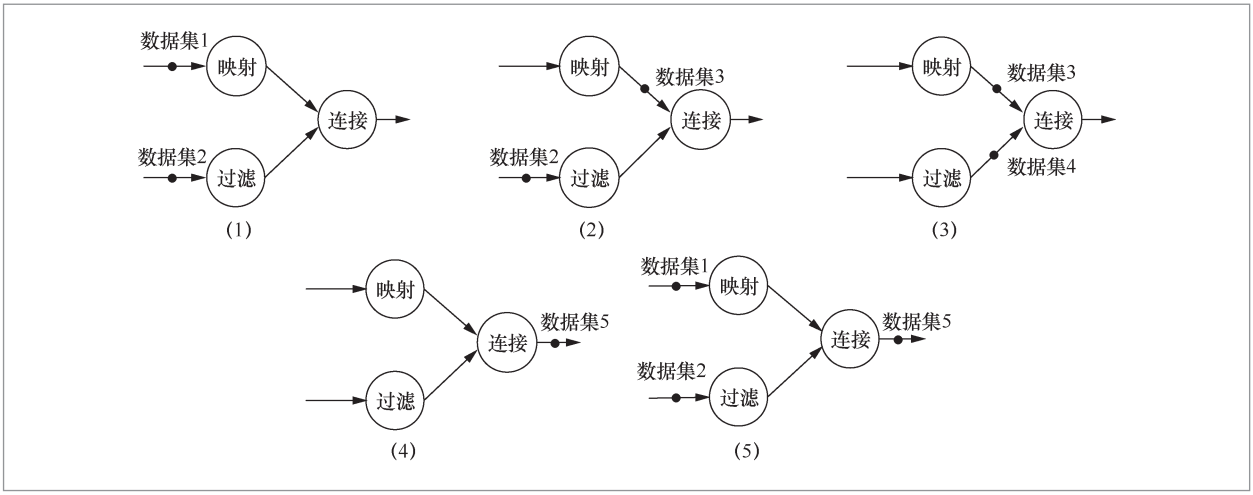

图1 数据流模型示例

用于某些特殊应用场景，还不能代替传统的控制流计算机。进入20世纪80年代后，随着多线程概念的发展，学术界和工业界的研究者更多地将研究重点放在更高层次的线程级并行上，结合传统控制流与数据流的优势研究数据流计算模型。

虽然在硬件上已经能够做到支持数据流并行，但在软件生态上仍发展落后，难以在实际生产中应用。数据流计算模型在算法程序设计中的优势引起了广大研究者的关注，与传统并行计算模型算法相比，基于数据流的并行计算模型具有支持度高、可拓展性好、性能功耗比高等优点，许多数据流执行模型相继被提出[27-29]。而由于大数据的快速发展，对大数据的处理需要更加高效的平台，因此在现有硬件的基础上，以数据流为核心的大数据处理平台应运而生。

2.2 主流分布式数据流系统

依赖数据流的概念，工业界发展出许多支持大数据处理、机器学习等任务的系统，这些系统在大数据、人工智能时代发挥着举足轻重的作用。结合控制流和数据流的优势，Suettlerlein等人[30]针对多核系统，基于硬件体系结构提出了一种新的数据流程序执行模型——Codelet，这是一种细粒度的、由事件驱动的、混合控制流/数据流的并行执行模型，其中基本执行单元负责存储执行指令，线程程序作为容器负责输入、输出以及保存共享数据。Codelet强调使用更细粒度的并行，更好地利用可用的硬件。基于Codelet计算模型，又设计实现了实时系统（如DARTS[31]，SWARM[32]等），实现具体的任务调度、负载均衡、功耗管理和内存管理等任务，同时，这种计算模型也可应用于许多领域，但这种依赖于新硬件体系结构的数据流执行模型难以得到大规模应用。在现有的硬件系统之上，利用数据流模型的思想，研究者设计了更高层次的应用数据流系统，如MapReduce[33]、Spark、Storm[34-35]、Flink、TensorFlow、Google Dataflow等。这些系统在多次的版本迭代中不断适应变化的需求，发挥着越来越重要的作用，展现出越来越强大的性能，逐步实现对异构环境的支持、对新硬件的支持以及在云环境下的应用等。当然，还有一些系统由于一些限制并没有得到大规模的应用，但在数据流系统的应用探索中也扮演着重要的角色，例如HAMR[36]基于Codelet执行模型，并拓展到集群系统中，实现了更好的资源利用和任务同步，同时支持批处理和实时流处理。Naiad[37]引入时间戳（timestamp）的概念描述任意复杂的流式计算，同时也解决了一般分布式系统难以处理的增量计算问题。Yita由中兴飞流信息科技有限公司研发，是基于数据流的运行时系统，采用特有的、动态的、细粒度的任务调度及资源管理，在计算性能、资源消耗等方面表现优异。几个目前比较流行的数据流系统如下。

（1）Spark

Spark是由美国加州大学伯克利分校的AMP实验室于2009年开发的基于内存计算的大数据并行处理框架。作为大数据处理平台的后起之秀，Spark在2014年打破了由Hadoop[38]保持的基准排序记录，对于同样的数据集，Spark仅用Hadoop十分之一的计算资源便将计算速度提高3倍。Spark以其运行速度快、易使用、通用性好以及运行模式多样的优势得到了众多开发者的青睐。Spark最大的特点就是基于内存，数据和中间结果都存储在内存中，避免了频繁的磁盘I/O开销。除此之外，Spark采用数据流计算模型，将一个应用划分为不同的任务，然后根据其依赖关系转化为DAG，在DAG中，各个任务以数据流的模式执行，极大地开发了程序中潜在的并行

性，大大加快了执行效率。

（2）Google Dataflow

Google Dataflow是由谷歌公司研究开发的一个数据处理模型，其目的在于提供一种统一批处理和流处理的系统。Dataflow模型基于事件时间（event-time）实现对流式数据的顺序处理，支持非对齐的窗口聚合，在正确性、时延和成本之间能做到较好的平衡，并实现数据处理中的逻辑概念和底层物理之间的解耦。目前已经在Google Cloud（谷歌云）中使用，其针对批数据和流数据提供统一的应用程序接口（application programming interface，API），开发者能够更加聚焦于数据逻辑本身定义数据处理流水线，然后由Google Cloud执行。

（3）Flink

Flink起源于一个叫作Stratosphere的研究项目，旨在建立下一代大数据分析引擎，于2014年4月成为Apache的孵化项目。Flink的基本模型也是数据流模型。它同时支持批处理和流处理，将计算任务转化为DAG，以数据流的模式执行。相对于Spark框架而言，Flink支持更高吞吐率、低时延、高性能的流处理，更适合对实时性要求高的场景。

（4）TensorFlow

TensorFlow是谷歌公司在2015年开源的通用高性能计算库，用于机器学习和深度神经网络方面的研究，它的通用性使其也可以应用于多种计算领域。TensorFlow也采用数据流的形式进行计算。数据流图中的节点表示数学操作，边表示节点之间相互联系的数据数组，即张量。一旦输入端的所有张量准备好，节点将被分配到各种计算设备中异步并行地执行运算。

数据流打破了并行度的限制，更容易实现超大规模的并行，基于数据流的系统在许多方面有发展的空间。首先，在大数据和高性能计算的应用需求下，基于数据流的系统应当与体系结构结合起来，协同发展，使得应用可以拓展到更大规模的平台上；其次是实现更细粒度的资源管理与任务管理，提升并行性，保证系统的兼容性，使其可在不同硬件平台进行移植；最后，数据流系统在系统可靠性以及能耗方面都有深入研究的空间。

3 典型分布式数据流计算系统中的缓存技术分析

如前所述，数据流系统常使用消息队列系统对数据源进行缓存处理。在数据流系统中，并没有一个统一的缓存管理机制进行算子之间的数据缓存。一般系统会通过多种机制处理算子之间速度不匹配的问题，包括系统底层实现和手动参数设置。下面对这些技术进行介绍。

（1）设置合理的并行度

数据流系统在处理大数据问题时很重要的一点在于可以实现流水线并行，这大大提高了传统串行处理的效率。但并行处理的能力跟硬件资源密切相关，因此在进行数据流作业时，要合理设置作业的并行度，充分利用硬件资源的性能，提升作业的处理速度。以Spark为例，设置并行度是Spark应用程序性能调优的重要方法之一，通过合理设置并行度，充分利用集群资源，避免资源的闲置，减少每个任务（task）要处理的数据量，提升整个Spark作业的处理速度。

（2）流量控制

在网络通信中，经常会出现这样的情况，生产者（producer）端产生数据的速度比消费者（consumer）端处理数据的速度快或者慢，如果仅在发送端和接收端设置一个缓存区，明显是不够的。如果缓存区空间是有限的，那么很快缓存区就会被耗

尽，新到达的数据只能被丢弃；如果缓存区空间是无限的，那么缓存区会不断增长，直到内存耗尽。为了解决缓存问题，需要通过流量控制解决上下游的速度差。流量控制通常有2种解决方案：静态限速和动态反压。静态限速通过限制发送端的发送速率实现，但这种方式有2点限制：第一是无法预估接收端能承受多大的速率，第二是其承受能力通常也会动态地波动。一般以动态反压的方式进行流量控制，接收端根据自己的处理情况及时地给予发送端反馈，告知发送端自己能承受的传输速率，使得发送端能实时地调整自己的发送速率以匹配接收端的处理能力。

（3）数据本地化

在分布式系统中，数据分布存储在各个节点中。为尽可能地减少不必要的网络传输，任务在执行前都会根据数据的分区信息进行分配，优先将其分配到要计算的数据所在的节点上。当本地计算资源不足时，任务会暂停，以等待空闲的资源释放出来；当等待一段时间还没有空闲的计算资源时，便会降低数据本地化级别，将任务转移到其他进程、节点甚至机架上运行。这似乎是与最快地处理任务的目标相矛盾的，但这一措施尽量地避免了网络传输通信带来的性能开销，同时也比较好地处理了各个节点计算资源和计算任务之间不匹配的问题，令作业的处理效率达到极致。

除了上述机制，各个数据流系统中还有许多用于解决算子之间可能存在的速度不匹配问题的措施，如代码优化、通信优化等。

4 典型分布式消息队列系统

4.1 分布式消息队列系统

分布式消息队列系统又称为消息中间件，是企业IT系统的核心组件，具有可靠投递、低耦合、流量控制、广播、最终一致性等功能，是异步远程过程调用（remote procedure call，RPC）的主要手段之一。由于数据源的差异性，尤其针对实时数据，数据流系统往往需要利用消息队列对数据进行缓存，再从中读取及处理数据。利用消息队列的特性，数据流系统可以更方便地处理数据。因此消息队列系统作为数据流系统数据源的缓存系统得到了广泛的应用。目前有很多流行的分布式消息系统，如顶级项目Kafka[16-19]，老牌的RabbitMQ[20-21]、ActiveMQ[22-23]，阿里巴巴集团自主开发的RocketMQ[24-25]以及炙手可热的新星Pulsar等。下面对其中几个进行简单介绍。

（1）Kafka

Apache Kafka[16-19]是比较著名的分布式消息系统之一，其最初由领英创建并开源，后来在Apache孵化器中“毕业”。Kafka得到许多公司的青睐。简单地说，Kafka是一个分布式发布-订阅消息系统。发布者向系统中写入数据，订阅者从系统中读取数据。消息被归到不同的主题中，每个主题可以划分为多个分区，分布存储在不同的节点上。如**图2**所示，在一个正常的Kafka集群中，有若干个生产者（即发

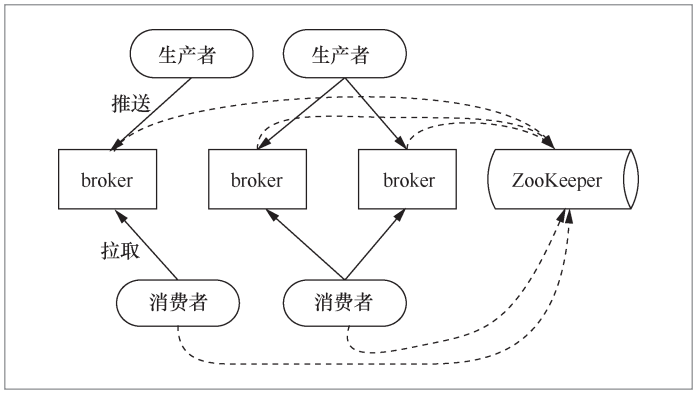

图 2 Kafka 系统架构

送端，消息的源头）、若干个broker（集群的主要组成部分，承担消息存储与计算任务）、若干个消费者（即订阅者，消息使用者）以及一个ZooKeeper[39-40]集群（对集群进行管理）。生产者产生消息后，将其推送（push）到broker中，当消费者需要消费消息时从broker中拉取（pull）。

Kafka主要支持简单的消息队列功能，一般适用于在大数据类的系统中执行实时数据计算、日志采集等任务，总体来说功能比较单一，但这换来的是超高的吞吐量和毫秒级的时延，在单机下吞吐量可以达到十万级，而且还提供了优秀的可用性及可靠性，方便拓展，活跃的社区也保证了使用的便易性。但为了保证良好的性能，Kafka一般要求支撑较少的主题（topic）数量。除此之外，Kafka存在消息被重复消费的问题，这对数据准确性会有一些影响。

（2）RabbitMQ

RabbitMQ[20-21]是一个开源的消息缓存系统，最初创建于2007年。RabbitMQ使用一个交换器接收broker的消息，并将它们推送给消费者。RabbitMQ不是面向磁盘的，即大多数消息传递操作是在内存中进行的，只有在内存不足或者指定存储消息时，才会将消息持久化到磁盘中。RabbitMQ最初起源于金融系统，用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。RabbitMQ是高级消息队列协议（advanced message queuing protocol，AMQP）的一个开源实现，其系统架构如**图3**所示。RabbitMQ没有使用第三方分布式集群管理服务（如Kafka中的ZooKeeper）对集群进行管理，Erlang语言可以很好地实现分布式管理。RabbitMQ比较特别的一点是具有灵活的路由功能，在消息进入队列之前，首先通过交换器（exchange）对消息进行路由。RabbitMQ可以利用内置的exchange实现典型的路由功能，也可以将多个exchange绑定在一起，实现更复杂的路由功能。

RabbitMQ基于Erlang语言开发，并发能力和性能都很强。它最大的特点是时延低，可以达到微秒级，但吞吐量比较低，逊色于Kafka。它基于主从架构实现高可用性，社区也相对活跃，适用于对实时性、可靠性要求比较高的任务。由于其采用Erlang语言开发，对于很多Java开发者来说不是很友好，难以进行深入的研究和维护。

（3）ActiveMQ

ActiveMQ[22-23]是一个比较流行的开源、多协议、基于Java的消息服务器，旨在为应用程序提供高效、可扩展、稳定、安全的企业级消息通信。ActiveMQ实现了JMS1.1（Java消息服务），并提供了很多附加的特性，比如Java管理拓展（Java management extensions，JMX）、主从管理、消息组通信、消息优先级、延迟接收消息、虚拟接收者、消息持久化、消息队列监控等。ActiveMQ的结构与Kafka类似，通过ZooKeeper对集群进行管理。ActiveMQ同时支持对消息的持久化和非持久化，可以将消息存储在内存、文件或数据库中。其中，可以通过Java数据库连接（Java database connectivity，JDBC）将消息存储在数据库中，这是其不同于其他消息系统的一点。

相比于Kafka，ActiveMQ的吞吐量要低一个数量级，在单机环境下可以达到万级，时延则为毫秒级。ActiveMQ利用主从架构实现高可用性，MQ功能极其完备，整体比较成熟，适用于处理解耦和异步问题。然而其社区活跃度日趋愈下，而且也缺乏大规模吞吐量场景的验证。

（4）RocketMQ

RocketMQ[24-25]是阿里巴巴集团在

2012年开源的分布式消息系统，在2016年成为Apache的孵化项目，并于2017年9月25日成为Apache的顶级项目。作为经历过多次阿里巴巴“双11”这种“超级工程”的洗礼并有稳定出色表现的国产中间件，近年来RocketMQ以其高性能、低时延和高可靠等特性被越来越多的国内企业使用。如**图4**所示，RocketMQ包含四大组件：生产者、消费者、名称服务器（nameserver）、broker，每个组件都可以部署成集群模式进行水平拓展。同样，broker是消息存储中心，接收来自消费者的消息并进行存储，消费者从这里拉取消息。broker有主节点（master）和从节点（slave）2种类型，其中master可读可写，而slave是只读的。从物理结构上看，broker有单master、多master、多master多slave等多种集群部署方式。另外，nameserver用来保存与broker相关的元信息，其功能与第三方集群管理服务ZooKeeper类似。

RocketMQ也提供优秀的吞吐量，时延为毫秒级，在性能上与Kafka很接近。与Kafka不同的是，RocketMQ在同等机器资源下，可以支撑大规模的主题，这是其很大的优势。同时，RocketMQ的MQ功能也较为完善，基于分布式的结构，便于拓展，模型简单，接口易用。基于Java开发的RocketMQ也方便开发者进行深度定制开发。

（5）Pulsar

Apache Pulsar是2016年由雅虎开源的下一代大规模分布式消息系统。Pulsar在实时计算系统的消息、存储、计算3个方面进行了很好的协调和统一。Pulsar遵循发布者-订阅者模型，使用内置的多数据中心副本，引入了多租户的概念。另外，Pulsar有一个Kafka API兼容接口，这使得将现有的Kafka程序移植到其中更加容易。如**图5**所示，Pulsar在架构上最明显的特征就是采用了消息服务和消息存储分层的策略。大多数消息系统将数据存储在broker中，而Pulsar依赖BookKeeper这种可拓展度高、强容灾和低时延的存储服务，将存储与计算分离，既充分地保证了数据的可用性，又可以在不移动实际数据的前提下，实现broker的动态扩展。

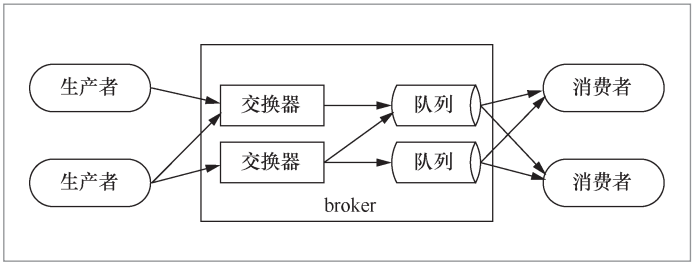

图 3 RabbitMQ 系统架构

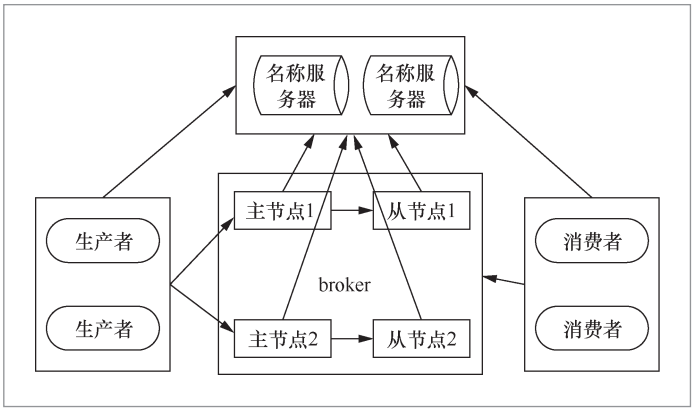

图 4 RocketMQ 系统架构

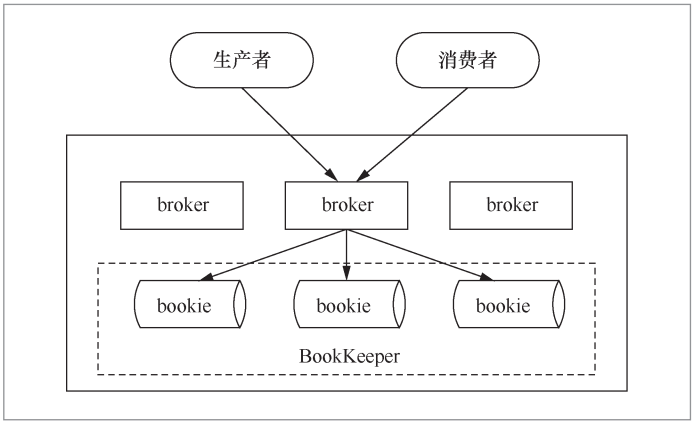

图 5 Pulsar 系统架构

Pulsar支持多租户的概念，可以在主题命名空间级别实现数据隔离，并且支持细粒度的访问控制，这使得Pulsar的应用程序更加安全、可靠。同时，Pulsar在性能上的表现也极为出色，相比于Kafka，在一个小型集群中针对一个分区一个主题、消息为100 byte的测试，Pulsar的吞吐量提升了2.5倍，时延降低了40%。整体来说，Pulsar能够作为消息系统领域的有力竞争者，但是由于其发展较晚，社区相对不够活跃，还需进一步接受市场的检验。

4.2 系统特性对比

本节选取一些重要的特征，对各个系统进行对比[41]，见**表1**。对于其中一些特征，解释如下。

实现语言：由于Java简单易用、功能强大，而且有平台独立、分布式等诸多特性，目前大多数消息系统是使用Java语言开发的，由于Java语言使用广泛，因此这些消息系统便于开发者们进行二次开发。而RabbitMQ则使用Erlang语言，尽管不利于二次开发，但是由于Erlang语言本身的高可用、高并发特点，并且其消息机制与AMQP极度吻合，使得RabbitMQ拥有诸多特性，这也是其被阿里巴巴集团青睐的一个重要原因。

客户端软件开发工具包（software development kit，SDK）：每个消息系统都提供了包括原生和第三方的多种语言版本的SDK，故消息队列系统在许多领域、许多应用中得到了广泛的使用。

通信协议：在实现消息队列功能时需要应用消息协议，各个消息系统使用的协议不同，根据使用的消息协议是否向行业开放消息规范文档，可以将其分为开放协议和私有协议。常见的开放协议有AMQP、简单（流）文本定向消息协议（simple（or streaming）text orientated messaging protocol，STOMP）、可拓展通信和表示协议（extensible messaging and presence protocol，XMPP）等，有些

表1 消息队列系统特性对比

对比项	Kafka	RabbitMQ	ActiveMQ	RocketMQ	Pulsar
实现语言	Scala/Java	Erlang	Java	Java	Java
客户端软件开发工具包	Java、.NET、C++等	Java、Scala等	Java、C、C++等	Java、C++、Go等	Java、C++、Go等
通信协议	TCP	AMQP	OpenWire、STOMP、REST、XMPP、AMQP	自定义	TCP
消息顺序性保证	顺序（一个分区内）	顺序（一个用户）	顺序（独有消费者和排他性队列模式）	全局顺序	全局顺序
消息延时与定时投递	不支持	支持	支持	支持	不支持
批量传输	支持	不支持	支持	支持	支持
广播	不支持	支持	支持	支持	支持
持久化	磁盘文件	内存、文件	内存、文件、数据库	磁盘文件	磁盘文件
消息优先级	不支持	支持	支持	不支持	支持
事务消息	支持	支持	支持	支持	不支持
集群管理服务	ZooKeeper		ZooKeeper	nameserver（自带）	BookKeeper、ZooKeeper

系统会根据自身情况对一些基本协议进行封装，如Kafka基于TCP/IP自行封装了一套协议，而RocketMQ则完全使用了一套自定义的消息协议。

消息顺序性保证：在消息系统中，能否保证消息的发送和消费的顺序一致是一个很重要的问题。某些应用场景（如银行业务）对消息的顺序要求很严格，而另一些应用场景则对消息顺序的要求较为宽松。Kafka的分布式单位是一个分区，在一个分区内部是保证有序的，但多个分区之间并不保证有序。RabbitMQ和ActiveMQ也是在某些特殊情况或模式下才能保证顺序。因此这些系统更适用于那些对消息顺序要求宽松的应用场景。而RocketMQ和Pulsar则能够保证消息的全局有序。某些业务场景（如短信定时发送需求）会要求消息延时或定时发送，在上述几种系统中，除了Kafka和Pulsar不能实现这样的需求，其他系统都能通过直接或间接的方式实现。

批量传输：消息批量处理表示在消息系统中，可以一次传输多条消息，以减少通信消耗，提高消息处理能力。在上述几种系统中，除RabbitMQ外，其他系统均支持批量传输。

持久化：不同的消息系统支持不同的持久化模式，即将消息以多种方式存储在磁盘、内存或文件中，根据应用需求选择合适的持久化模式。

消息优先级：在某些应用场景下，某些消息可能需要被优先消费，这时就可以对消息设置优先级，不同的系统以不同的方式支持消息优先级设置，但Kafka和RocketMQ不支持这一特性。

事务消息：为了保证消息传输的可靠性，即确保消息在传输过程中不丢失，需要利用消息的事务机制，而在上述消息系统中，Pulsar不支持事务消息，但可以通过其他方式在一定程度上保证消息的可靠性，官方也通报在接下来即将发布的新版本中加入对事务的支持。

集群管理服务：作为一个分布式系统，各个消息系统一般需要配套其他集群管理服务实现集群下的环境，如Kafka、ActiveMQ和Pulsar都使用ZooKeeper作为集群管理工具，RocketMQ则使用自带的nameserver实现，RabbitMQ则完全依靠Erlang语言的分布式特性来构建集群。除此之外，各个系统还拥有许多共同或不同的特性。

4.3 对数据流系统的支持与未来的展望

目前，大多数消息队列系统提供了一些针对数据流系统的编程接口，消息系统可以作为数据源与数据流系统之间的缓存系统缓存源数据，并为数据流系统提供稳定的数据输入。但对于数据流系统算子计算的中间数据，还无法使用现有的数据流系统进行缓存。尽管数据流系统本身有一些机制可以用来平衡算子之间可能存在的速度差异，但这些机制耦合在系统的各个模块之中，没有一个统一的缓存管理模块解决这个问题。而且随着数据量的增多，应用对计算效率的要求越来越高，未来很有可能出现系统自身基本的缓存机制无法支撑巨量数据和高速计算的情况，故需要一个统一且强大的缓存管理机制。消息队列系统可以作为一个独立的缓存管理模块中的存储子模块，充分发挥其数据缓存管理性能，从而解决数据流系统中的缓存问题。

5 数据缓存技术

在传统意义上，缓存（cache）是用一个硬件或软件组件存储数据，此数据可能是前序计算的结果，也可能来自其他存储位

置。当后续进程需要用到该数据时，可以直接从缓存中读取，而读取速度要比重新计算或者从数据原始位置读取快得多。这一过程要求请求的数据位于缓存中，称为缓存命中。如今缓存的概念更加广泛，不仅在CPU和内存之间有缓存，内存和磁盘之间也有缓存，甚至在网络应用中也存在缓存的概念。凡是位于2个速度相差较大的数据读写或处理单元之间，用于平衡两者数据传输速度差异的结构，都可以被称为缓存。

缓存技术中包括几个重要概念，分别是命中率、缓存容量、缓存更新策略。

（1）命中率

在缓存系统中，若可以直接通过缓存获取需要的数据，则称为命中，否则称为没有命中。其中，命中率=命中数/（命中数+没有命中数）。显然，命中率越高，缓存系统的效率就越高，对系统性能的提升就越明显。

（2）缓存容量

缓存容量就是缓存中最多能容纳的数据量。通常各种缓存机制会对缓存容量大小进行一定的限制。当实际缓存的数据超出缓存容量时，就会触发缓存更新策略。

（3）缓存更新策略

缓存更新策略一般有3种，分别是先进先出（first in first out，FIFO）、最近最少使用（less frequently used，LFU）和最近最久未使用（the least recently used，LRU）。FIFO即最先进入缓存的数据，在缓存空间不够的情况下，会先被清除出去。LFU即使用次数最少的数据会先被淘汰，这里需要记录数据的使用次数。LRU即如果一个数据在最近一段时间没有被访问，那么可以认为将来它被访问的可能性也很小，这时便可以优先将其淘汰。同时，基于这3种基本缓存更新策略已经衍生出许多改进算法，使缓存效率更高，并且适用于不同的场景。为了充分发挥作用，缓存不仅暂存刚刚访问过的数据，还可以根据上下文对应用进行预测，以实现数据的预取，把未用过但将要用到的数据存入缓存中。

数据缓存可以应用在很多方面。其中比较传统的是应用在CPU和内存之间的缓存，作为临时数据暂存器，缓存在一定程度上解决了CPU运行处理速度和内存读写速度不匹配的问题。CPU的缓存也可以使用多级缓存，每一级缓存的读写速度和容量均不相同，从而在控制成本的情况下，可以尽可能地发挥CPU处理性能的潜力。另外是网络中的缓存，随着互联网的发展，每时每刻都有大量的数据产生，面对爆炸式增长的数据，为保证数据的高效传输，缓存成为一种很有效的手段。互联网流量的主要来源是流视频内容。随着视频质量的不断提高，流媒体给网络基础设施带来的压力也在增加。使用内容分发网络将内容缓存到离用户更近的地方，是减少网络负载的常见解决方案。

在数据流系统中，算子之间的速度差异也可能导致数据堆积或算子闲置的问题，而随着数据量的增加，应用对系统处理能力的要求日益提高，这一问题可能越发明显。因此对算子之间的中间数据进行缓存管理也是很重要的一个研究方向。但目前的数据流系统中还没有一个系统性的缓存管理机制，仅使用一些分布式系统中常用的技术来优化这一问题。数据流系统中以数据为驱动的特点使得数据的流畅运转变得十分重要，根据数据流计算模型的特点设计相应的缓存管理也将有很大的发展空间。

6 结束语

数据流系统以其良好的性能在业界得到了广泛的应用，已经成为新一代大数据解决方案的重要组成部分。数据流算子之间的紧密配合使得数据流系统的性能发挥到

极致，因此数据流算子之间的数据缓存管理和存储抽象也将成为一个重要的研究方向。而传统的消息队列系统可以以其优秀的数据缓存管理能力在数据流系统中发挥作用，成为数据流系统的重要组成部分。

参考文献:

[1] YAZDANPANAH F, ALVAREZ-MARTINEZ C, JIMENEZ-GONZALEZ D, et al. Hybrid dataflow/von-neumann architectures[J]. IEEE Transactions on Parallel and Distributed Systems, 2014, 25(6): 1489-1509.

[2] NOWATZKI A, GANGADHAR V, SANKARALINGAM K. Computer with hybrid von-neumann/dataflow execution architecture: U.S. Patent 10,216,693[P]. 2019-2-26.

[3] DENNIS J B. First version of a data flow procedure language[C]// Programming Symposium. Heidelberg: Springer, 1974: 362-376.

[4] DENNIS J B, FOSSEEN J B, LINDMAN J P. Data flow schemas[C]// International Sympoisum on Theoretical Programming. Heidelberg: Springer, 1972: 187-216.

[5] DENNIS J B. Fresh breeze: a multiprocessor chip architecture guided by modular programming principles[J]. ACM SIGARCH Computer Architecture News, 2003, 31(1): 7-15.

[6] NAJJAR W A, LEE E A, GAO G R. Advances in the dataflow computational model[J]. Parallel Computing, 1999, 25(13-14): 1907-1929.

[7] ZAHARIA M, CHOWDHURY M, FRANKLIN M J, et al. Spark: cluster computing with working sets[C]// The 2nd USENIX Conference on Hot Topics in Cloud Computing. Berkeley: USENIX Association, 2010: 10.

[8] MENG X, BRADLEY J, YAVUZ B, et al. Mllib: machine learning in Apache Spark[J]. The Journal of Machine Learning Research, 2016, 17(1): 1235-1241.

[9] KARAU H, KONWINSKI A, WENDELL P, et al. Learning Spark: lightning-fast big data analysis[M]. Sebastopol: O'Reilly Media, Inc., 2015.

[10] CARBONE P, KATSIFODIMOS A, EWEN S, et al. Apache Flink: stream and batch processing in a single engine[J]. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2015, 36(4): 28-38.

[11] FRIEDMAN E, TZOUMAS K. Introduction to Apache Flink: stream processing for real time and beyond[M]. Sebastopol: O'Reilly Media, Inc., 2016.

[12] ABADI M, BARHAM P, CHEN J, et al. TensorFlow: a system for large-scale machine learning[C]// The 12th USENIX Symposium on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2016: 265-283.

[13] ABADI M, AGARWAL A, BARHAM P, et al. TensorFlow: large-scale machine learning on heterogeneous distributed systems[J]. Computer Science, 2016, arXiv: 1603.04467.

[14] WONGSUPHASAWAT K, SMILKOV D, WEXLEX J, et al. Visualizing dataflow graphs of deep learning models in TensorFlow[J]. IEEE Transactions on Visualization and Computer Graphics, 2017, 24(1): 1-12.

[15] AKIDAU T, BRADSHAW R, CHAMBERS C, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing[J]. Proceedings of VLDB Endowment, 2015, 8(12): 1792-1803.

[16] 倪炜. 分布式消息中间件实践[M]. 北京: 电子工业出版社, 2018.
NI W. Practice of distributed message middleware[M]. Beijing: Publishing House of Electronics Industry, 2018.

[17] NARKHEDE N, SHAPIRA G, PALINO T. Kafka: the definitive guide: real-time data and stream processing at scale[M]. Sebastopol: O'Reilly Media, Inc., 2017.

[18] 郑奇煌. Kafka技术内幕[M]. 北京: 人民邮电

出版社, 2017.
ZHENG Q H. Inside of Kafka technology[M]. Beijing: The People's Posts and Telecommunications Press, 2017.
[19] DOBBELAERE P, ESMAILI K S. Kafka versus RabbitMQ: a comparative study of two industry reference publish/subscribe implementations: industry paper[C]// The 11th ACM International Conference. New York: ACM Press, 2017: 227-238.
[20] AYANOGLU E, AYTAS Y, NAHUM D. Mastering RabbitMQ[M]. Birmingham: Packt Publishing Ltd, 2016.
[21] ROSTANSKI M, GROCHLA K, SEMAN A. Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ[C]// The 2014 Federated Conference on Computer Science and Information Systems. Piscataway: IEEE Press, 2014: 879-884.
[22] IONESCU V M. The analysis of the performance of RabbitMQ and ActiveMQ[C]// Roedunet International Conference-Networking in Education & Research. Piscataway: IEEE Press, 2015: 132-137.
[23] HENJES R, SCHLOSSER D, MENTH M, et al. Throughput performance of the ActiveMQ JMS server[M]. Heidelberg: Springer, 2007.
[24] 丁威, 周继锋. RocketMQ技术内幕[M]. 北京: 机械工业出版社, 2018.
DING W, ZHOU J F. Inside of RocketMQ technology[M]. Beijing: China Machine Press, 2018.
[25] 郭嘉凯. RocketMQ: 从阿里巴巴走向世界[J]. 软件和集成电路, 2018(11): 13.
GUO J K. RocketMQ: from Alibaba to the world[J]. Software and Integrated Circuit, 2018(11): 13.
[26] 李国杰. 一种新的体系结构——数据流计算机[J]. 电子计算机动态, 1981(11): 3-10.
LI G J. A new architecture: dataflow computer[J]. Computer Review, 1981(11): 3-10.
[27] ZUCKERMAN S, SUETTLERLEIN J, KANUERHASE R, et al. Using a Codelet program execution model for exascale machines: position paper[C]// The 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era. New York: ACM Press, 2011: 64-69.
[28] 高光荣. 大数据的流动之美——数据流与大数据: 挑战与机遇[J]. 中国计算机学会通讯, 2013, 9(12): 16-19.
GAO G R. The flowing beauty of big data, dataflow and big data: challenge and opportunity[J]. Communication of the CCF, 2013, 9(12): 16-19.
[29] NEUMEYER L, ROBBINS B, NAIR A, et al. S4: distributed stream computing platform[C]// The 2010 IEEE International Conference on Data Mining Workshops. Piscataway: IEEE Press, 2010: 170-177.
[30] SUETTLERLEIN J, ZUCKERMAN S, GAO G R. An implementation of the Codelet model[M]. Heidelberg: Springer, 2013: 633-644.
[31] SUETTERLEIN J. Darts: a runtime based on the Codelet execution model[D]. Newark: University of Delaware, 2014.
[32] LAUDERDALE C, KHAN R. Towards a Codelet-based runtime for exascale computing: position paper[C]// The International Workshop on Adaptive Self-tuning Computing Systems for the Exaflop Era. New York: ACM Press, 2012: 21-26.
[33] DEAN J, CHEMAWAT S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.
[34] TOSHNIWAL A, TANEJA S, SHUKLA A, et al. Storm@twitter[C]// The 2014 ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2014: 147-156.
[35] 李明, 王晓鹏. Storm源码分析[M]. 北京: 人民邮电出版社, 2014.
LI M, WANG X P. Source code analysis of Storm[M]. Beijing: The People's Posts and Telecommunications Press, 2014.
[36] WU Y, ZHENG L, HEILIG B, et al. HAMR: a dataflow-based real-time in-memory

cluster computing engine[J]. The International Journal of High Performance Computing Applications, 2017, 31(5): 361-374.
[37] MURRAY D G, MCSHERRY F, ISAACS R, et al. Naiad: a timely dataflow system[C]// The 24th ACM Symposium on Operating Systems Principles. New York: ACM Press, 2013: 439-455.
[38] SHVACHKO K, KUANG H, RADIA S, et al. The Hadoop distributed file system[C]// The 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies. Piscataway: IEEE Press, 2010: 1-10.
[39] HUNT P, KONAR M, JUNQUEIRA F P, et al. ZooKeeper: wait-free coordination for internet-scale systems[C]// The 2010 USENIX Annual Technical Conference. Berkeley: USENIX Association, 2010: 11.
[40] JUNQUEIRA F, REED B. Zookeeper: distributed process coordination[M]. Sebastopol: O' Reilly Media, Inc., 2013.
[41] 吴璨, 王小宁, 肖海力, 等. 分布式消息系统研究综述[J]. 计算机科学, 2019(B06): 1-5.
WU C, WANG X N, XIAO H L, et al. Survey on distributed message system[J]. Computer Science, 2019(B06): 1-5.

作者简介

袁旭初(1995-),男,东北大学计算机科学与工程学院硕士生,主要研究方向为分布式系统、并行计算等。

付国(1996-),男,东北大学计算机科学与工程学院硕士生,主要研究方向为分布式系统、并行计算等。

毕继泽(1998-),男,东北大学计算机科学与工程学院本科生,主要研究方向为大数据处理、并行与分布式计算等。

张岩峰(1982-),男,博士,东北大学计算机科学与工程学院教授,主要研究方向为大数据处理与挖掘、深度学习、并行与分布式计算等。

聂铁铮(1980-),男,博士,东北大学计算机科学与工程学院副教授,主要研究方向为大数据管理、数据集成与融合、区块链等。

谷峪(1981-),男,博士,东北大学计算机科学与工程学院教授,主要研究方向为大数据分析、分布式计算、时空和图数据管理等。

鲍玉斌(1968-),男,博士,东北大学计算机科学与工程学院教授,主要研究方向为商务智能、数据挖掘、大数据分析等。

于戈(1962-),男,博士,东北大学计算机学院教授、博士生导师,中国计算机学会会士。现任中国计算机学会信息系统专业委员会主任、数据库专业委员会委员、系统软件专业委员会委员,《计算机学报》《软件学报》《计算机研究与发展》等期刊编委。曾获得“教育部跨世纪人才基金”和“中国高校青年教师奖”。主要研究方向为分布式数据库系统、数据科学与大数据管理、区块链技术与应用等。

收稿日期: 2020-01-19

基金项目: 国家重点研发计划基金资助项目(No.2018YFB1003404);国家自然科学基金资助项目(No.61672141);中央高校基本科研业务费专项资金资助项目(No.N181605017, No.N181604016)

Foundation Items: The National Key Research and Development Program of China(No.2018YFB1003404), The National Natural Science Foundation of China(No.61672141), Fundamental Research Funds for the Central Universities(No.N181605017, No.N181604016)