基于内存的分布式隐私流查询系统

张鹏[1,2]，刘庆云[1,2*]，熊翠文[1,2,4]，李保珲[1,2]，郑超[1,2]，易立[3]
[1]中国科学院 信息工程研究所，北京 中国 100091
[2]信息内容安全技术国家工程实验室，北京 中国 100091
[3]国家计算机网络应急技术处理协调中心，北京 中国 100029
[4]中国科学院大学，北京 中国 100049

MDPSS: Memory-based Distributed Private Stream Searching System

ZHANG Peng[1,2], LIU Qingyun[1,2*], XIONG Cuiwen[1,2,4], LI Baohui[1,2], ZHENG Chao[4], YI Li[3]
[1](Institute of Information Engineering CAS, Beijing 100091, China)
[2](National Engineering Laboratory for Information Security Technologies, Beijing 100091, China)
[3](National Computer Network Emergency Response and Coordination Center, Beijing 100029, China)
[4](University of Chinese Academy of Sciences, Beijing, 100049, China)

Abstract The era of Big Data has begun, people are more concerned about data privacy. On the one hand, the users are more eager for fresh and low-latency searching results than ever. On the other hand, they do not want to open the searching information. To this end, this paper proposes a memory-based distributed private stream searching system, in which queries are encrypted by Paillier cryptosystem. The system adopts shared-nothing architecture to support the horizontal scalability, and partitions the stream into fragments to achieve parallel query and bitmap index-based storage. Experimental results show the effective and efficient of the system.

Key words private stream searching; in-memory computing; paillier cryptosystem; bitmap index

摘要 随着大数据时代的到来，隐私问题备受关注，用户一方面希望获得新鲜和低延迟的查询结果，另一方面又希望对查询信息进行隐私保护，为此本文提出了一种基于内存的分布式隐私流查询系统，其中，查询可以通过Paillier 密码系统进行加密。该系统在shared-nothing 架构下支持水平扩展，实现了在内存中对流数据进行分片的并行查询以及基于位图索引的压缩存储。实验证明了该系统的有效性。

关键词 隐私流查询；内存计算；Paillier 密码系统；位图索引

中图法分类号

1 引言

大数据时代已经到来，其中典型的 4 个特点是：规模性，多样性，高速性和价值性[1]，同时，随着物联网的兴起，很多应用开始通过大量的传感器来采集数据，这些数据的规模往往是 TB 级或者更多，随着数据规模的增多，对已有数据的利用率的高低直接影响企业的经济效益，因此业界也逐渐开始意识到高效的数据查询在帮助企业进行市场决策，提高企业生产力的重要

*通讯作者 liuqingyun@iie.ac.cn

本课题得到国家自然科学基金(No.61402464),中国博士后基金(No.2013M541076)资助.张鹏(1984-),男,博士,计算机学会(CCF)会员（E200029316G），主要研究领域为分布式系统和数据挖掘,E-mail: pengzhang@iie.ac.cn;刘庆云,男,博士,高级工程师,主要研究领域为信息安全;熊翠文(1991－),女,硕士研究生,主要研究领域为数据挖掘;李保珲(1986－),男,博士研究生,主要研究领域为云计算和信息安全;郑超(1984－),男,博士研究生,主要研究领域为数据挖掘;易立(1984－),男,博士,工程师,主要研究领域为数据挖掘

性，为此，大数据的处理模式也从批处理向流处理发生转变[2]。然而，数据访问延迟使得传统的硬件体系和软件架构限制了流处理的性能。在过去四十年里，计算机系统的主要存储方式是硬盘，文件系统和数据库都是针对硬盘发展起来的。虽然硬盘的容量提高很快（自 1980 年代中期以来提高超过 1000 倍），但是相比之下性能却一直难如人意，传输速率仅提高 50 倍，访问延迟只减少了 50%。如果按容量/带宽来衡量（Jim Gray 规则[3]），硬盘的访问延迟实际上急剧恶化了。内存计算作为低延迟和高吞吐的代表，能够使得流处理的性能呈几何级增长，是实现高效流处理的优秀解决方案，受到业界青睐[4]。由于内存计算需要把大量的数据装载在内存中，所以内存开销就会比传统的解决方案要大很多，然而，内存价格的大幅下降为内存计算的可行性扫清了障碍，MapUpdate[5]，D-Streams[6]，RAMCloud[7]和 Spark[8]是内存计算中与查询相关的几个代表性产品。然而，对于一些应用，出于个人隐私或者商业利益的保护等多种原因，用户想要隐藏他们的查询条件，因此如何在内存查询基础上实现隐私流查询成为新的挑战。隐私流查询不同于加密数据查询，在隐私流查询中，数据不被加密而查询被加密[9]。隐私流查询事实上与数据库的隐私信息检索（Private Information Retrieval，PIR）相关[10]。不同之处是隐私信息检索的计算开销取决于整个数据库的大小，而隐私流查询的计算开销与整个数据库或者流的大小无关，而只与被检索的数据大小有关。隐私信息检索和隐私流查询的另一个不同点是大多数隐私信息检索把数据库看作一个可被检索的很长的比特串，查询则是可被检索的比特串上的索引，而隐私流查询则支持文本中关键词的检索，目前的研究主要集中在对查询的加密构造上[11][12]，目的是提高查询条件的隐私保护能力，但是缺少对查询处理的可扩展能力的考虑。为此，本文提出了一种基于内存的分布式隐私流查询系统—MDPSS，其中，查询可以通过 Paillier 密码[13]系统进行加密。该系统在 shared-nothing 架构下支持水平扩展，实现了在内存中对流数据进行分片的并行查询以及基于位图索引的压缩存储，具体包含以下三个创新点：

1) 基于内存的分布式查询处理，该技术通过在内存中对流数据进行分片存储和并行查询，提高了查询处理的可扩展能力。

2) 基于位图索引的压缩存储，该技术不仅节省了存储开销，而且通过在索引压缩形式上的布尔操作进一步提高了查询性能。

3) 支持对查询进行 Paillier 加密，该技术保护了查询条件不被泄漏。

文章的组织如下：第二节重点介绍 MDPSS 的分布式处理架构以及对加密查询的支持；第三节通过实验详细分析 MDPSS 的可扩展能力；第四节是总结和下一步工作的展望。

2 分布式隐私流查询模型

首先介绍 MDPSS 的分布式处理架构，其中包括：

2.1 数据模型

MDPSS 的数据模型是分片。在 MDPSS 中，每个表都会被划分成一些分片的集合，每个分片大约有 1 万行，如表 1 所示。MDPSS 利用一个时间戳列作为数据分布、数据存储和数据查询的优化手段。MDPSS 把数据源按照定义好的时间间隔分片，一般情况下是按一小时或一天，也可以进一步根据从其他列的值进行划分，已达到期望的分片大小。分片的标识符由数据源标识符、数据的时间间隔、分片创建时递增的版本号以及分块号组成。分片的元数据被系统用来进行并发控制，读操作总是访问指定时间范围内最新版本的分片中的数据。

在 MDPSS 中，大多数分片是不变的历史分片。这些分片永久存储在分布式文件系统中，例如 Hadoop File System[14]。所有的历史分片都有一个元数据，元数据描述了分片的属性，例如字节大小、压缩格式和存储位置。通过创建新的历史分片可以更新已有历史分片所包含的时间间隔的数据。实时分片表示最近时间间隔的分片，它会随着新的数据注入增量更新，并且在增量更新的过程中就可以支持查询。当指定的时间过后，实时分片会转变成历史分片。MDPSS 非常适合聚集查询，其中历史分片和实时分片都是通过增量索引来构建的，而增量索引只作用在数据流经过的行中同一个属性上计算的聚集指标的数值（例如表 1 中的 impression 和 revenue 的总和）。这通常会带来一个数量级的压缩，而不会影响聚集数值的精确度。

表1：分片的数据示例

Timestamp	Publisher	Advertiser	Gender	Country	Impressions	Clicks	Revenue
2014-01-01 T01:00:00Z	bieberfever.com	google.com	Male	USA	1800	25	15.70
2014-01-01 T01:00:00Z	bieberfever.com	google.com	Male	USA	2912	42	29.18
2014-01-01 T01:00:00Z	ultratrimfast.com	google.com	Male	USA	1953	17	17.31
2014-01-01 T01:00:00Z	ultratrimfast.com	google.com	Male	USA	3914	170	34.01

2.2 分布式处理

MDPSS 的查询处理架构由以下几种类型的节点组成，如图 1 所示。

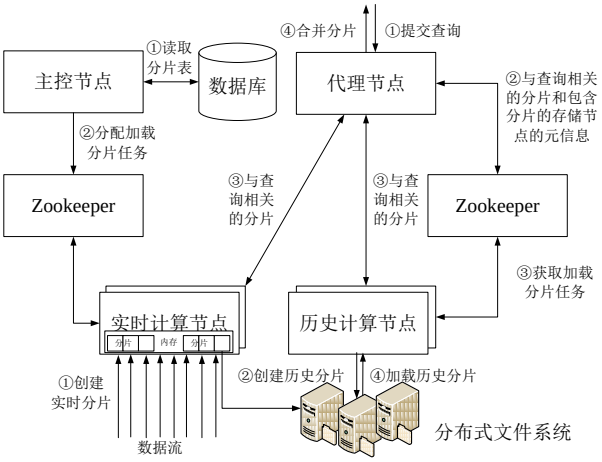

图 1 MDPSS 的分布式处理架构

实时计算节点负责最新数据的注入、存储以及查询。同样，历史计算节点负责对历史数据的加载和查询。MDPSS 把数据被存放到存储节点，存储节点可以是历史计算节点或者是实时计算节点。一个查询首先会访问代理节点，该节点负责把查询路由到各个包含相关数据的存储节点，然后存储节点并行执行所属它们查询的部分并且将结果返回给代理节点，代理节点接收到这些结果后进行合并，然后把合并后的结果返回给查询的请求者。代理节点、历史计算节点和实时计算节点都被认为是可查询节点。MDPSS 通过一组主控结点来管理分片的分布和复制。主控结点是不可查询节点，它主要是用来协调集群的稳定性。主控结点需要依赖外部的 MySQL 数据库。MDPSS 通过 Apache Zookeeper[15]来进行集群协作。虽然查询是通过 HTTP 转发，但是集群内部通信是通过 Zookeeper。

2.2.1 历史计算节点

历史计算节点是 MDPSS 中重要的工作者并且不需要依赖外部组件。历史计算节点会从持久化的分布式文件系统中加载历史分片并且让它们可以被查询。由于历史计算节点之间不知道对方，所以它们不存在单点竞争问题。历史计算节点操作起来非常简单，它们仅需知道如何完成分配给它们的任务。为了帮助其他节点发现历史计算节点和它们所提供的分片，每个历史计算节点都会维护一个与 Zookeeper 的连接。历史计算节点通过在指定配置的 Zookeeper 路径下创建临时节点来发布它们的在线状态和所提供的分片。一个历史计算节点加载新的分片或者丢弃已存在的分片是通过在该节点指定的 load queue 路径下创建临时 znodes 来实现的。

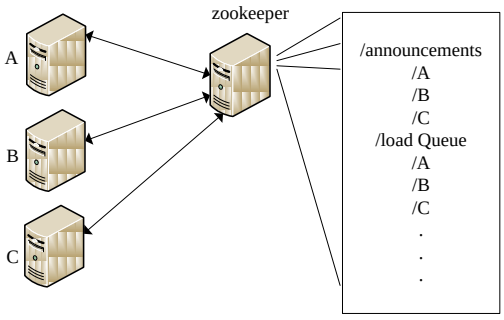

图 2 历史计算节点在指定路径下创建临时的 znodes

图 2 给出了一个简单的历史计算节点和 Zookeeper 的交互过程。每个历史计算节点都有一个与其关联的 load queue 路径，当它们在线时，会在 announcements 路径下发布它们所提供的分片。为了使一个分片可以被查询，历史计算节点必须首先使用这个分片的本地副本。在一个历史计算节点开始从文件系统中下载一个分片之前，它首先检查本地存储的目录（也称为缓存）来判断这个分片是否已经在本地存储。如果这个分片的缓存信息不存在，那么这个历史计算节点

将会从 Zookeeper 中下载这个分片的元信息。这个元信息包含这个分片在文件系统的位置以及如何解压和处理这个分片的信息。一旦历史计算节点完成这个分片的处理，这个节点就在 Zookeeper 上发布它可以提供这个分片。此时，这个分片就是可以被查询了。

2.2.2 实时计算节点

实时计算节点提供了实时数据的注入和查询的功能。通过实时计算节点建立索引的数据可以立即被查询。

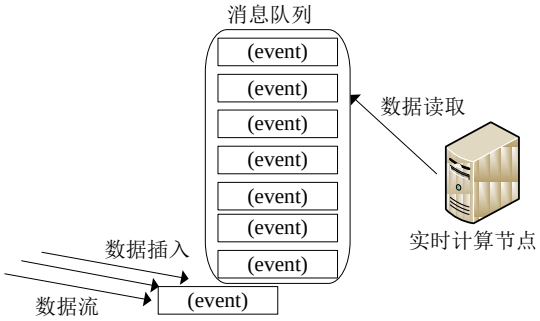

图3 实时计算节点的数据读取

实时计算节点是数据的使用者，因此它需要一个对应的生产者来提供数据。典型地，为了数据持久化的目的，一个消息队列，例如 Kafka[16] 放在生产者和实时计算节点之间，如图 3 所示。其中的消息队列可以看作实时计算节点读取数据流的中间缓存。这个消息队列能够维护一个实时计算节点已经读取的数据位置的偏移量，实时计算节点周期性地更新这个偏移量。这个消息队列也可以看作是实时计算节点最近读取的数据的副本存储器。从数据产生，到消息队列存储，再到提供给实时计算节点使用，总共的时间大约在几百毫秒。实时计算节点对所有注入数据在内存中维护一个索引缓存，随着新的数据出现在消息队列中，这些索引被增量地添加，并且可以被直接查询。实时计算节点周期性（或者当达到最大行数时）把这些索引持久化到文件系统，在每次持久化后，一个实时计算节点利用最近被持久化索引的最后一个数据位置的偏移量来更新消息队列。每个被持久化的索引都是不可变更的。如果一个实时计算节点出现故障，那么当它开始恢复时，它只需要重新加载已经被持久化到文件系统的索引并且继续从最后一个偏移量被提交的点开始读取消息队列。周期性地提交偏移量可以减少一个实时计算节点在出现故障后重新扫描的数据量。实时计算节点维护一个当前正在更新的索引和所有持久化到文件系统的索引的视图。这个视图使得一个节点中的所有索引都可以被查询。实时计算节点定期地会安排一个后台任务来检索一个数据源的所有持久化的索引。这个任务会把这些索引全部合并后构建一个历史分片。然后，实时计算节点再把这个分片上传到文件系统中，同时向历史计算节点发出可以使用这个分片的信号。实时分片转换成历史分片的过程不会出现数据丢失，图 4 显示了这个过程。与历史计算节点类似，实时计算节点也在 Zookeeper 上发布分片。与历史分片不同，实时分片可以表示延长到未来的一段时间。例如，一个实时计算节点可以发布它正在使用的一个分片，这个分片包含了当前这个小时的数据。在这个小时结束之前，这个实时计算节点都会一直收集数据。例如，每隔 10 分钟（这个时间是可配置的），实时计算节点都会刷新并且把收集的数据在内存中的索引持久化到文件系统中。当这个小时结束时，实时计算节点会通过创建一个新的索引并且发布下一个小时的分片来提供下一个小时的查询。在指定的窗口时间还没有结束之前，实时计算节点不会把实时分片转换成历史分片。通过利用窗口可以分散进入的数据点以降低数据丢失的风险。在这个窗口时间结束时，实时计算节点会合并所有持久化的索引，并把它转成一个历史分片，同时把这个历史分片交给历史计算节点来使用。一旦这个分片在历史计算节点上是可以被查询的，那么实时计算节点就会刷新所有关于这个分片的信息并且发布它不再提供这个分片。

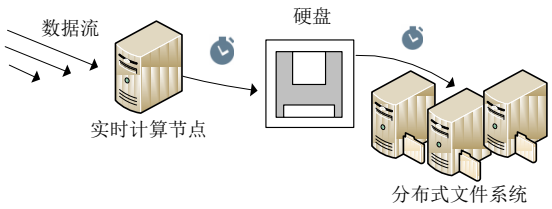

图4 实时计算节点的分片持久化

实时计算节点是可扩展的。如果一个给定数据流的注入速率超过一个实时计算节点的最大容量，那么其他的实时计算节点则会加入。多个实时计算节点同时使用来自同一个数据流的数据，并且每个实时计算节点只负责其中的一部分，这自然地在节点之间产生了划分。每个实时计算节点发布它所提供的实时分片，并且每个实时分片都有一个分块号。来自每个实时计算节点的数据

将在代理层进行合并。

2.2.3 代理节点

代理节点类似查询路由器，它能把查询路由到可查询的节点上，例如历史计算节点和实时计算节点。代理节点知道发布在 Zookeeper 上关于哪些分片存在以及这些分片被存放到哪些节点的元信息。代理节点也负责在返回给请求者最终结果前，合并来自各个节点的查询结果。此外，代理节点还通过维护一个最近结果缓存来提供数据持久化层。当出现多个节点发生故障并且一个分片的所有副本都丢失的情况，如果缓存中保存了这些信息，那么查询结果仍然可以被返回。

2.2.4 主控节点

主控结点主要负责分片的管理和分布，其中包括加载新的分片，丢弃过时的分片，管理分片副本以及进行分片负载均衡。主控结点周期性地检查集群状态。在运行时，主控结点通过比较集群的预期状态和真实状态来做任务分配的决定。主控结点维护一个 Zookeeper 连接来获取集群中所有节点的信息。同时，主控结点也维护一个 MySQL 数据库连接来获取运行时的参数和配置的信息。MySQL 数据库中包含了分片表，它记录了目前应该提供的历史分片。这个表可以被任意一个创建历史分片的节点所更新。MySQL 数据库也包含一个规则表，该表包含了分片在集群中是如何创建，销毁以及复制的信息。当主控结点给一个计算节点分配任务时，它不会直接和这个计算节点进行通信，而是在 Zookeeper 中创建一个临时 znode 节点，该节点包含了这个计算节点应该做什么的信息。每个计算节点维护一个类似和 Zookeeper 的和主控结点的连接以监听分配给它的新任务。

2.2.5 位图索引的压缩存储

MDPSS 采用以列为单元进行数据存储以支持在大规模数据上进行聚集查询。这是因为列存储只需要加载和扫描需要列上的数据，使得CPU的使用效率更高[17]。相反，在一个以行为单元的数据存储中，与行相关联的所有列作为聚集的一部分被扫描，冗余的扫描时间可以导致性能下降250%[18]。

MDPSS 支持不同的列类型。根据这些类型，MDPSS 通过使用不同的压缩方式来减少在内存和文件系统上存储一个列的代价。在例子中表 2 所示，publisher、advertiser、gender 和 country 列只包含字符串。字符串列可以是词典编码的。词典编码是一个压缩数据的普通方法，例如，在表 2 中，我们把每一个 publisher 映射成一个唯一的整型标识符。

```
bieberfever.com -> 0
ultratrimfast.com -> 1
```

该映射把 publisher 列表示成一个整型数组，数组下标个数等于数据集的行数。对于 publisher 列，我们可以如下表示 publishers：[0, 0, 1, 1]。这个结果的整型数组使其非常适合压缩。压缩算法在列存储中使用非常普遍。MDPSS 目前使用位图索引[19]进行压缩。类似的压缩方法也可以应用于数值型的列。例如，表中的 clicks 列和 revenue 列也可以分别表示成一个数组。

```
Clicks -> [25, 42, 17, 170]
Revenue -> [15.70, 29.18, 17.31, 34.01]
```

在这种情况下，我们只压缩原始数值而不压缩词典表示的数值。为了支持任意过滤集，MDPSS 为字符串列创建了额外的查询索引。这些查询索引被压缩，MDPSS 操作它们的压缩形式。过滤集可以通过多个查询索引的布尔表达式来表示。在索引压缩形式上的布尔操作能够提高性能，节省空间。考虑表 2 中的 publisher 列，对于每一个唯一的 publisher，我们可以得到这个 publisher 出现在这个表的哪些行的表示。我们把这些信息存储在二进制数组里，其中数组下标的索引代表出现的行。如果这个 publisher 在某一行出现，那么这个数组下标被标记为 1，例如：

```
bieberfever.com-> rows[0, 1]->[1][1][0][0]
ultratrimfast.com-> rows[2, 3]->[0][0][1][1]
```

bieberfever.com 出现在行 0 和 1 中。列值到行下标的映射形成一个倒排索引。为了知道哪些行包含了 bieberfever.com 或 ultratrimfast.com，我们可以把两个数组用 OR 连接起来。

```
[0][1][0][1] OR [1][0][1][0] = [1][1][1][1]
```

2.3 加密查询模型

通过对代理节点进行扩展，MDPSS 实现了加密查询功能，其中包含以下几个步骤：

第一步是客户端的查询构建。假设关键词字典表示为 $D=\{w_1,w_2,\ldots,w_{|D|}\}$。通过一些关键字 $K\subseteq D$ 的析取构建加密查询。客户端生成一个密钥对，对于每个 $i\in 1,\ldots,|D|$，如果 $w_i\in K$，则 $q_i=1$，如果 $w_i\notin K$，则 $q_i=0$。然后，$q_1,q_2,\ldots,q_{|D|}$被加密，放入一个表示加密查询的数组 $Q=(E(q_1),E(q_2),\ldots,$

$E(q_{|D|})$)。然后，客户端发送加密查询 Q 和公钥 n 给代理节点。

第二步是代理节点的流查询程序。当代理节点处理流查询时，必须维护三个缓存。它们是数据缓存，系数缓存和匹配索引缓存，分别被表示为 F，C 和 I。这些缓存都是来自密文空间 $Z_{n^2}^*$ 的元素数组，其中 F 和 C 的长度都为 l_F，I 的长度为 l_I。为了简化解释，假设每个分片最多 n 个比特位，使之适合 Z_n 中的一个明文。对于 Z_n 包含 s 个元素的长文档，我们令 F 是 $l_F \times s$ 的数组，随后 F 更新的操作是按块进行的。

数据缓存存储匹配的加密分片，这些分片随后被客户端用于重构匹配的原始分片。特别地，数据缓存包含匹配的加密分片内容的线性方程组。这个线性方程组随后由客户端求解以获得匹配的原始分片。

系数缓存按加密格式存储每个匹配分片中匹配的关键词个数。这里把一个分片匹配的关键词的个数记为分片的 c-value。系数缓存将用于数据缓存获得的匹配的加密分片的重构。和数据缓存一样，系数缓存以线性方程组形式存储它的信息。客户端随后求解线性方程组以重构 c-values 值。

匹配索引缓存是一个加密的布隆过滤器，能够跟踪加密的匹配分片的索引。更准确地来说，匹配索引缓存是一些满足 $\{\alpha_1,\alpha_2,...,\alpha_r\}\subseteq\{1,...,t\}$ 的索引集合 $\{\alpha_1,\alpha_2,...,\alpha_r\}$ 的加密表示，其中 r 是满足查询条件的分片个数。

每一个缓存开始时，所有的元素初始化为 0 的加密表示形式。现在给出每个分片被处理时的缓存更新细节。假设代理节点处理第 i 个分片 f_i。

步骤 1：计算加密的 c-value。首先，代理节点在分片中查找每个 w_j 对应的查询数组项 $Q[j]$，然后计算这些项的乘积。因为 Paillier 密码系统的同态性，这个乘积是分片的 c-value 的加密结果，也即是分片中找到的 K 中不同元素的个数。也就是说，$\prod_{w_j\in W_i} Q[j]=E(\sum_{w_j\in W_i} q_j)=E(\mathrm{c}_i)$，其中 W_i 是第 i 个分片 f_i 中的不同关键词的集合，c_i 定义成 $|K\cap W_i|$。特别地，当且仅当分片与查询匹配时才有 $c_i\neq 0$。

步骤 2：更新数据缓存。代理节点使用 Paillier 密码系统的同态性来计算 $E(\mathrm{c}_i\ f_i)$。

$$E(c_i)^{f_i}=E(c_i f_i)=\begin{cases}E(c_i f_i) & \text{如果} f_i \text{与查询匹配}\\ E(0) & \text{其他}\end{cases}$$

代理节点按照下面的步骤把 $E(\mathrm{c}_i\ f_i)$ 乘到数据缓存中的位置的子集中。令 G 是将 $Z\times Z$ 映射到 {0,1}的伪随机函数族。随机地选择 $g\xleftarrow{R}G$ (在初始化过程中执行一次，并且所有的分片使用相同的 g)。算法将 $E(\mathrm{c}_i\ f_i)$ 乘到数据缓存的每个满足 $g(i,j)=1$ 的位置 j。假设我们用第二个分片更新数据缓存中的第三个位置，同时第一个分片也乘这个位置，那么 $g(1,3)=g(2,3)=1$。这两个分片可能都与查询匹配或都不与查询匹配。假设 f_1 与查询匹配，而 f_2 与查询不匹配。在处理 f_2 之前，$D(F[3])=c_1f_1$。然后乘以 $E(\mathrm{c}_2\ f_2)$，$D(F[3])=c_1f_1+c_2f_2$。由于 f_2 与查询不匹配，则 $c_2=0$，因此 $D(F[3])=c_1f_1$，数据缓存在效果上没有变化。$D(c)$表示密文 c 解密结果。这一机制允许数据缓存累积匹配分片的线性组合同时丢弃所有不匹配的分片。

步骤 3：更新系数缓存。$E(\mathrm{c}_i)$ 以 $E(\mathrm{c}_i\ f_i)$ 更新数据缓存相似的方式乘以系数缓存的每个位置。特别地，当 $g(i,j)=1$ 时，代理节点将 $E(\mathrm{c}_i)$ 乘以系数缓存的每个位置 j。

步骤 4：更新匹配索引缓存。代理节点进一步将 $E(\mathrm{c}_i)$ 乘以匹配索引缓存中固定数目的位置。这里使用 k 个哈希函数 $h_1,h_2,...,h_k$ 来选择 $E(\mathrm{c}_i)$被添加到的 k 个位置。为了优化性能，客户端应该如下选择参数 $k=\left\lfloor\frac{l_I log 2}{m}\right\rfloor$，其中 m 是与查询进行匹配的分片个数。匹配分片 i 乘以匹配索引缓存的位置表示为 $h_1(\mathrm{i}),h_2(\mathrm{i}),...,h_k(\mathrm{i})$。这里如果 f_i 不匹配，则 $c_i=0$，那么匹配索引缓存在效果上不会变化。在对流中 t 个分片完成上述步骤后，代理节点向客户端返回三个缓存，同时返回函数 g。

第三步是客户端重构加密的匹配分片，其中包括三个步骤。

步骤 1：解密缓存。首先，客户端用它的私钥 K_{priv} 使用 Paillier 解密算法解密三个缓存中的值，得到解密的缓存 F'，C' 和 I'。

步骤 2：重构匹配索引。对于每个索引 $i\in\{1,2,\ldots,t\}$，客户端计算 $h_1(\mathrm{i}),h_2(\mathrm{i}),...,h_k(\mathrm{i})$，在解密的匹配索引缓存中校验对应的位置，如果所有的位置非 0，那么 i 被添加到可能匹配索引列表 $\alpha_1,\alpha_2,...,\alpha_\beta$ 中。注意，如果 $c_i\neq 0$，那么 i 将被添加到这个列表中。然而，由于布隆过滤器的假阳性，这里也许会得到一些多余的索引。因此当假阳值的个数加上真正匹配数 r 超过 l_F 时，那么需要检查是否出现溢出。这时，如果 $\beta<l_F$，那么将继续向列表中添加索引直到长度为 l_F。注意，由于 $t\geq l_F$，这里不会用完索引。

步骤 3：重构匹配分片的 c-values。假设匹配索引的超集是 $\{\alpha_1,\alpha_2,...,\alpha_{l_F}\}$，客户端求解

$\{c_{\alpha_1}, c_{\alpha_2}, ..., c_{\alpha_{l_F}}\}$的值。这一步是通过求解线性方程组 $A\cdot\vec{c}=C'$ 得到的，其中 A 是第 i, j 项设置成 $g(\alpha_i, j)$ 的矩阵，C' 是存储在解密的 c-buffer 的数值的向量，$\vec{c}$ 是列向量 $(c_{\alpha_i})_{i=1,..l_F}$。现在匹配索引的精确集合 $\{\alpha_1', \alpha_2', ..., \alpha_r'\}$ 可以通过检查每个 $i\in\{1,\ldots,l_F\}$ 满足 $c_{\alpha_i}=0$ 来得到。在执行这步之前，需要用 1 代替向量 $\vec{c}$ 中的所有 0。假设在解密的 c-buffer 中有 4 个点，例如 l_F=4，7 个分片被处理，可能的匹配索引列表 $\{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}=\{1,3,5,7\}$。那么可以得到：

$$\mathrm{A}=\begin{pmatrix}1&0&1&0\\1&1&0&1\\1&0&0&1\\0&1&1&0\end{pmatrix},\ C'=\begin{pmatrix}2\\3\\1\\3\end{pmatrix}$$

计算：

$$c_{\alpha_1}=c_1=1$$
$$c_{\alpha_2}=c_3=2$$
$$c_{\alpha_3}=c_5=1$$
$$c_{\alpha_4}=c_7=0$$

然后可以发现有三个匹配分片（r=3），它们是 f_1，f_3，f_5。

步骤 4：重构匹配分片。最后，匹配分片 $f_{\alpha_1'}, f_{\alpha_2'}, \ldots, f_{\alpha_r'}$ 的原始内容可以通过求解线性方程组 $A\cdot diag(\vec{c})\cdot\vec{f}=F'$ 来得到，其中：

$$diag(\vec{c})=\begin{pmatrix}c_1&0&\cdots\\0&c_2&\\\vdots&&\ddots\end{pmatrix}$$

然后计算 $\vec{f}=diag(\vec{c})^{-1}\cdot A^{-1}\cdot F'$。注意，$diag(\vec{c})$ 是非奇异的，因为之前确保了在 $\vec{c}$ 中不会出现 0。匹配分片的内容表示为 $f_{\alpha_1'}, f_{\alpha_2'}, \ldots, f_{\alpha_r'}$；$\vec{f}$ 中的其他项表示为 0。继续上面的例子（构造 F'的值），对应求解下面的方程组：

$$f_1+f_5=32$$
$$f_1+2f_3+f_7=32$$
$$f_1+f_7=10$$
$$2f_3+f_5=44$$

解得 f_1=10，f_3=11 和 f_5=22。（f_7=0 这个值被忽略），也即是编号为 10，11 和 22 的分片是与查询匹配的结果。

3 实验评价

下面是我们的实验评价。为了测试 MDPSS 的扩展性，我们创建了一个 80GB 数据集。这个数据集合包括十几个维度，取值范围从双位到几千万。我们为每行（总个数、求和、平均值）计算三个聚集指标。数据首先在时间戳上被划分，然后根据维度的值被划分，创建几百个分片，每个分片大约 1 万行。

测试基准的集群包含 6 个历史计算节点，每个节点有 16 个内核，16GB 的 RAM，10GigEFA 的以太网和 1TB 的磁盘空间。总体上说，这个集群包含了 96 个内核，96GB 的 RAM，以及足够快的以太网和足够的磁盘空间。表 2 描述了每个查询的目的，其中：

1、查询的时间戳范围覆盖了所有的数据

2、每一台机器都有 16GB 的 RAM 和 1TB 的磁盘以及 16 个内核。机器被设置只使用 15 个线程来处理查询

3、使用 memory-mapped 存储引擎（机器使用内存映射数据而不是把数据加载到 Java 堆中）

图 5 显示集群的扫描速率，图 6 显示内核的扫描速率。在图 5 中，我们给出了基于 5 个内核节点的集群的预计线性扩展的结果。特别地，我们观察性能的边际收益随着集群规模增长而递减，在预计的线性扩展上，SQL 查询 1 在 55 个内核节点的集群上实现了每秒 370 万行的速率。事实上，速率是每秒 260 万行。然而，查询 2-6 则保持线性的加速比直到 30 个内核节点，同时这些查询在图 6 中内核扫描速率几乎保持稳定。

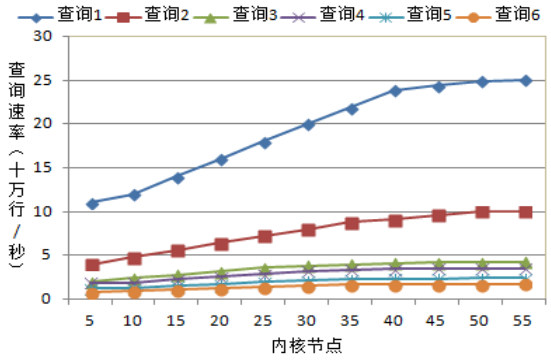

图 5 MDPSS 集群扫描率

根据 Amdahl 定律[20]，并行计算系统速率的提高通常受限于系统串行操作的时间需求。表 2 中列举的第一个查询是简单的计数，如图 6 所显示，实现了每个内核节点每秒 33,000 行的扫描率。事实上，对于给定的数据集，我们认为 55 个内核节点的集群是 over-provisioned，这也解释了为什么比 30 个内核节点集群增长不多的原因。MDPSS 的并发模型是基于分片的：一个线

程扫描一个分片。如果一个计算节点的分片个数对内核节点的个数取模的值很小（例如 17 个分片和 15 个内核节点），在计算的最后一轮，一些内核节点将会被闲置。当增加更多的聚集指标时，我们发现性能下降。这是因为 MDPSS 利用了以列为主的存储格式。对于计数（*）查询，MDPSS 需要检查时间戳列来判断是否满足“where”子句。当增加聚集指标时，MDPSS 需要加载聚集指标的数值以及扫描它们，占用了被扫描的内存。

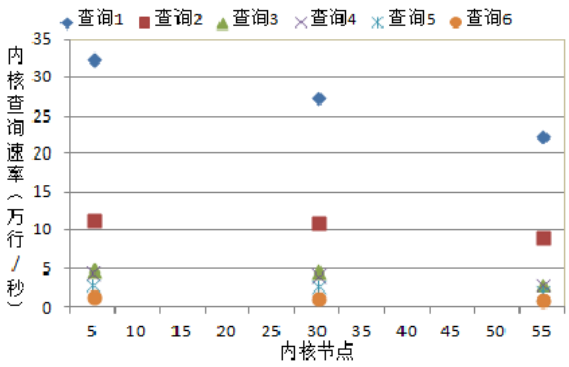

图 6 MDPSS 内核扫描率

表 2：查询示例

Query#	Query
1	SELECT count(*) FROM _table_ WHERE timestamp ≥ ? AND timestamp < ?
2	SELECT count(*), sum(metric1) FROM _table_ WHERE timestamp≥? AND timestamp < ?
3	SELECT count(*), average(metric1) FROM _table_ WHERE timestamp ≥? AND timestamp <?
4	SELECT high_card_dimension, count(*) AS cnt FROM _table_ WHERE timestamp ≥? AND timestamp < ? GROUP BY high_card_dimension ORDER BY cnt limit 100
5	SELECT high_card_dimension, count(*) AS cnt, sum(metric1) FROM _table_ WHERE timestamp≥? AND timestamp < ? GROUP BY high_card_dimension ORDER BY cnt limit 100
6	SELECT high_card_dimension, count(*) AS cnt, sum(metric1), sum(metric2), sum(metric3), sum(metric4) FROM _table_ WHERE timestamp ≥? AND timestamp < ? GROUP BY high_card_dimension ORDER BY cnt limit 100

4 结论

本文提出了一种基于内存的分布式隐私流查询系统。该系统在 shared-nothing 架构下支持水平扩展，实现了在内存中对流数据进行分片的并行查询以及基于位图索引的压缩存储，并且通过 Paillier 密码系统支持隐私流查询。然而，在一些应用中，预先指定关键词集合 W 也许是不可接受的。为此，下一步准备不再使用 W，允许 K 是 Σ^* 的任意有限子集，其中 Σ 是一个字母表。在隐私流查询的构建过程中，设定 Q 的长度为 l_Q，初始化每个元素为 $E(0)$。然后，对于每个 $w_i \in K$，使用哈希函数 $h:\Sigma^* \to \{1,K,l_Q\}$ 在 Q 中选择位置 $h(w)$，且设置 Q[h(w)]=E(1)。在流查询中处理第 i 个分片时，代理节点计算 $E(c_i)=\prod_{w\in W_i} Q[h(w)]$，其余部分不修改。使用这一扩展，如果有些字 $w' \in W_i$，那么对于 $w \in K$ 有 $h(w')=h(w)$，分片 f_i 很有可能是伪匹配查询。这些假阳的可能性是这种方法的主要缺点。然而，这种方法的优点是可以扩展查询范围，支持关键词合取。

参 考 文 献

[1] 孟小峰, 慈祥. 大数据管理:概念,技术与挑战. 计算机研究与发展, 2013, 50(1):146-169
X. F. Meng, X. Ci. Big Data Management: Concepts, Techniques and Challenges[J]. Journal of Computer Research and Development, 2013, 50(1):146-169

[2] 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, 软件学报, 2014, 25(4):839-862
D. W. SUN, G. Y. ZHANG, W.M. ZHENG. Big Data Stream Computing: Technologies and Instances[J]. Journal of Software, 2014, (4): 839-862

[3] Gray Jim, Graefe Goetz. The five-minute rule ten years later, and other computer storage rules of thumb[J], ACM Sigmod Record, 1997, 26(4):63-68.

[4] Krutov Ilya, Vey Gereon, Bachmaier Martin. In-memory Computing with SAP HANA on IBM eX5 Systems, IBM Redbooks, 2014

[5] Wang Lam, Lu Liu, STS Prasad et al..Muppet: MapReduce Style Processing of Fast Data[C], Proceedings of the VLDB Endowment, 2012, 5(12): 1814-1825.

[6] Matei Zaharia, Tathagata Das, Haoyuan Li et al.. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters[C], Proceedings of the 4th USENIX conference on Hot Topics

in Cloud Computing, 2012, 10.

[7] Ousterhout John, Agrawal Parag, Erickson David et al.. The case for RAMClouds: scalable high-performance storage entirely in DRAM[J], ACM SIGOPS Operating Systems Review, 2010, 43(4):92-105.

[8] Zaharia Matei, Chowdhury Mosharaf, Franklin Michael J, Shenker Scott, Stoica Ion. Spark: cluster computing with working sets[C]. Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, 2010.

[9] Bethencourt J, Song D, Waters B. New techniques for private stream searching[J]. ACM Transactions on Information and System Security (TISSEC), 2009, 12(3): 16.

[10] Jarecki S, Jutla C, Krawczyk H, et al. Outsourced symmetric private information retrieval[C], Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013: 875-888.

[11] Y Sergey, Private information retrieval[J]. Communications of the ACM, 2010, 53(4):68-73.

[12] B Christoph, H Pieter, J Willem, P Andreas. A Survey of Provably Secure Searchable Encryption[J], ACM Computing Surveys, 2014, 47(2):18.

[13] P. Paillier, Public-Key Cryptosystems Based on Composite Degree Residuosity Classes[C], Proc. 17th Int',l Conf. Theory and Application of Cryptographic Techniques (EUROCRYPT), pp. 1999, 223-238.

[14] Bhandarkar Milind. Hadoop: a view from the trenches[C], Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining, 2013, 1138-1138.

[15] Hunt Patrick, Konar Mahadev, Junqueira Flavio P, Reed Benjamin. ZooKeeper: wait-free coordination for internet-scale systems[C]. Proceedings of the 2010 USENIX conference on USENIX annual technical conference, 2010, 8:11-11.

[16] Kreps Jay, Narkhede Neha, Rao Jun. Kafka: A distributed messaging system for log processing[C]. Proceedings of the NetDB, 2010.

[17] R. Cattell. Scalable sql and nosql data stores[J]. ACM SIGMOD Record, 2011, 39(4):12-27.

[18] C Bear, A Lamb, N. Tran. The ertica database: Sql rdbms for managing big data[C]. In Proceedings of the 2012 workshop on Management of big data systems, 2012, 37-38.

[19] Su Yu, Agrawal Gagan, Woodring, Jonathan, et al.. Taming massive distributed datasets: data sampling using bitmap indices[C]. Proceedings of the 22nd international symposium on High-performance parallel and distributed computing, 2013, 13-24.

[20] Ma Sen, Andrews David.. On energy efficiency and amdahl's law in FPGA based chip heterogeneous multiprocessor systems[C], Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays, 2014, 253.