海量存储

FAST2016

BTrDB：时间序列处理优化存储系统设计

姓名：李玮琛　　学号：2017282110226

摘要

高精度，高采样率遥测时间序列的增加对现有的时间序列数据库提出了新的问题。这些数据库既不能处理这些数据流的吞吐量需求，也不能提供对其进行有效分析的必要原语。本文提出了遥测时间序列数据的一种新颖的抽象概念和提供这种抽象概念的数据结构：一个时间分区版本注释的写时复制树。Go 中的一个实现被证明优于现有解决方案，在四节点集群上每秒吞吐量为 5300 万次，每秒查询值为 1.19 亿次。该系统实现了 2.9 倍的压缩比，满足了一年中的数据在 200ms 以内的统计查询，如一年一次的存储 2.1 万亿个数据点的生产部署所证明的。该数据库的原理和设计通常适用于各种时间序列类型，代表了物联网技术发展的重大进步。

1.引言

随着物联网的兴起，一类具有独特存储需求的分布式系统变得越来越重要。它涉及从大量高精度网络化传感器采集，提取和分析近乎实时和历史的时间相关遥测，具有相当高的采样率。这种情况发生在监测电网，建筑系统，工业过程，车辆，结构健康等的内部动态。它通常提供复杂基础设施的情景意识。它具有与面向用户的焦点和点击数据（在现代 Web 应用程序中普遍存在），智能计量数据（从数百万计米收集 15 分钟间隔数据）或单次专用仪器记录等几乎不同的特征。

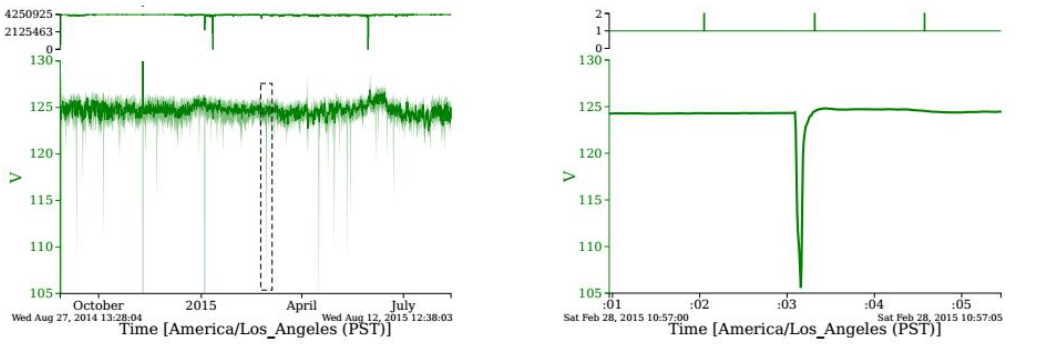
文章把重点放在这样一个遥测源 - 微观同步器或者 uPMU 上。这些新一代的小型，相对便宜和极高精度的功率计将被部署在电网的配电层中，可能以百万计。在图 2 所示的分布式系统中，每个设备产生 12 个 120 Hz 高精度值的流，时间戳精确到 100 ns（GPS 的极限）。受此类数据源成本下降的驱动，文章着手构建一个支持每个后备服务器上的这些设备超过 1000 个的系统 - 每秒超过 140 万个插入点，并且是数据分析的预期读取和写入的几倍。此外，这种遥测经常到达无序，延迟和重复。面对这些特点，存储系统不仅要保证原始数据流的一致性，还要保证所有的数据分析都是一致的。此外，快速响应时间对于从几年到几毫秒的时间范围内的查询来说非常重要。

这些需求超过了当前时间序列数据存储的能力。KairosDB [15]，OpenTSDB [20]和 Druid [7]等流行的系统设计用于低采样率的复杂多维数据，因此，这些遥测流的吞吐量和时间戳分辨

率不足，具有比较简单的基于时间范围的数据和查询。这些数据库全部宣告每台服务器的读写每秒钟远远少于 100 万个值，通常具有到达顺序和重复约束，这部分将在第 2 节详细介绍。

为了解决这个问题，文章构建了一个新颖的使用时间序列的数据库抽象概念——BTrDB，作为原始插入和查询提供更高的持续吞吐量，以及加速分析预期的高级原语 44 每个服务器每年有四十亿个数据点。

这个解决方案的核心是时间序列遥测数据的一个新的抽象和一个提供这种抽象的数据结构：一个具有时间分割，多分辨率，版本注释，写时复制的树。第 4 节介绍了使用这种数据结构的数据库设计。



(a) Statistical summary of a year of voltage data to locate voltage sags, representing 50 billion readings, with min, mean, and max shown. The data density (the plot above the main plot) is 4.2 million points per pixel column.

(b) The voltage sag outlined in (a) plotted over 5 seconds. The data density (the plot above the main plot) is roughly one underlying data point per pixel column

图 1. 使用统计摘要定位典型的 upmu 遥测数据流等感兴趣的事件

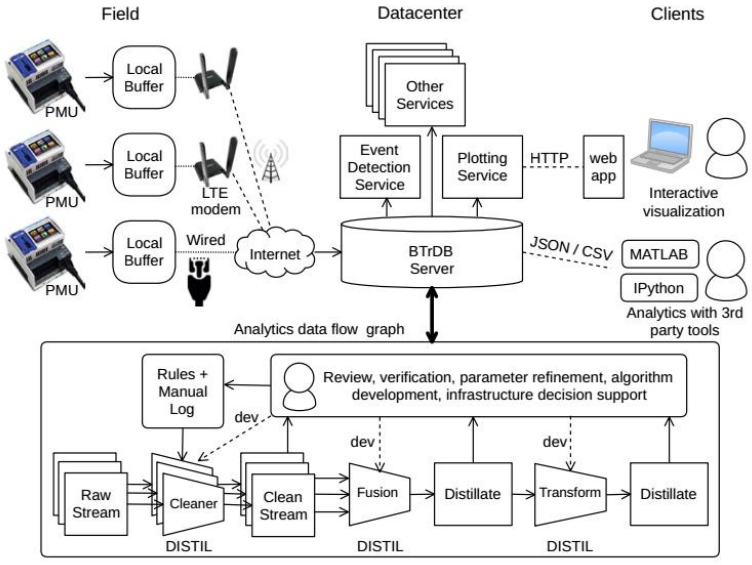


图 2. uPMU 网络存储和查询处理系统

BTrDB 的开源 4709-Go Go 实现演示了此方法的简单性和有效性，在四个节点群集上实现了每秒 5300 万个插入值和每秒 1.19 亿个查询值，并具有必要的容错和一致性保证。 此外，新颖的分析原语允许导航包含数十亿数据点的一年数据（例如图 1a），以使用在 100-200ms 内完成的统计查询序列来定位和分析亚秒事件（例如图 1b） ，目前的工具无法做到这一点。这将在第 5 节中讨论。

2. 时间序列数据提取

BTrDB 提供的基本抽象是一个一致的，一次写入的，有序的时间 - 值序列对。每个流由 UUID 标识。在典型的使用中，大量的元数据集合与每个流相关联。然而，元数据的性质在用途上差别很大，并且存在许多用于查询元数据以获得流的集合的良好解决方案。因此，文章将 lookup（或 directory）函数完全从时间序列数据存储中分离出来，只用 UUID 来标识每个流。所有访问都是在一个流版本的时间片段上执行的。所有的时间戳都是纳秒，没有样本规律性的假设。

InsertValues（UUID，[（time，value）]）创建一个插入了给定的（时间，值）对集合的流的新版本。从逻辑上讲，流按时间顺序进行维护。最常见的情况是，点被添加到流的末尾，但是这不能被假定：来自设备的读数可能不按顺序被传送到存储器，可能发生重复，空洞可能被回填，并且可能对旧数据进行校，这可能是重新校准的结果。这些情况经常发生在现实世界的实践中，但很少受时间序列数据库的支持。在 BTrDB 中，每次插入一组值都会创建一个新版本，而不修改旧版本。这允许在旧版本的数据上执行新的分析。

最基本的访问方法 GetRange（UUID，StartTime，EndTime，Version）→（Version，[（Time，Value）]）在给定版本的流中检索两次之间的所有数据。可以指示'最新'版本，从而消除在查询范围之前调用 GetLatestVersion（UUID）→Version 以获取流的最新版本。确切的版本号与数据一起返回以便将来可重复查询。BTrDB 不提供操作来重新采样特定时间表的流中的原始点，或者跨越流对齐原始样本，因为正确地执行这些操作最终取决于数据的语义模型。这样的操作得到了数学环境的很好的支持，比如 Pandas[17]，对插值方法的适当控制等等。

虽然这个操作是大多数历史学家提供的唯一一个，有数万亿分之一，但它的效用是有限的。在隔离了一个重要的窗口或执行报告，如过去一小时的干扰之后，它被用在最后一步。分析原始数据流是完全不切实际的; 例如，每个 uPMU 每年生产近 500 亿个样本。以下访问方法对于广泛的分析和增量生成计算精简的流是非常有效的。

在可视化或分析大量数据段 GetStatisticalRange（UUID，StartTime，EndTime，Version，Resolution）→（Version，[（Time，Min，Mean，Max，Count）]）中检索两次给定的统计记录 时间分辨率。每个记录涵盖 2 分辨率纳秒。开始时间和结束时间在 2 个分辨率边界上，结果记录在该时间单位内是周期性的; 因此汇总跨越流。 还可以查询未对齐的窗口，但性能略有下降。

GetNearestValue（UUID，Time，Version，Direction）→（版本，（时间，值））查找给定时间的最近点，无论是向前还是向后。它通常用于获取感兴趣流的“当前”或最近到现在的价值。

在实践中，原始数据流输入到蒸馏过程的图表中，以便清理和过滤原始数据，然后将精炼过的流合并生成有用的数据产品，如图 2 所示。这些蒸馏器反复发射，获取新数据并计算输出段。 在存在无序到达和丢失的情况下，在没有来自存储引擎的支持的情况下，确定哪些输入范围已经改变以及需要计算或重新计算哪些输出范围以保持整个蒸馏管线中的一致性可能是复杂且昂贵的。

为了支持这一点，ComputeDiff（UUID，FromVersion，ToVersion，Resolution）→[（StartTime，EndTime）]提供了包含给定版本之间差异的时间范围。返回的变更集的大小可以通过限制 FromVersion 和 ToVersion 之间版本的数量来限制，因为每个版本都有最大的大小。每个返回的时间范围将大于 2 分辨率纳秒，允许调用者优化批量大小。

作为实用程序，DeleteRange（UUID，StartTime，EndTime）：创建一个删除了给定范围并刷新（UUID）的流的新版本，确保给定的流被刷新到复制存储。

3.时间分区树

为了提供上面描述的抽象，文章使用了时间分割写时拷贝版本注释的 k-tree 树。由于原语 API 基于时间范围提供查询，因此使用划分时间的树通过允许特定时间点的快速定位来服务索引的角色。基础数据点存储在树的叶子中，树的深度由数据点之间的间隔来定义。统一采样的遥测流将具有固定的树深度，而不管树中有多少数据。所有的树在逻辑上代表了一个巨大的时间范围（从 Unix 时代开始，从-260ns 到 3 * 260ns，或者大约从 1933 年到 2079 年，精度达到纳秒），在前端和后端有大的孔，每个点之间有较小的孔。图 3 说明了 16 ns 的时间分割树。注意 8 到 12 ns 之间的孔。

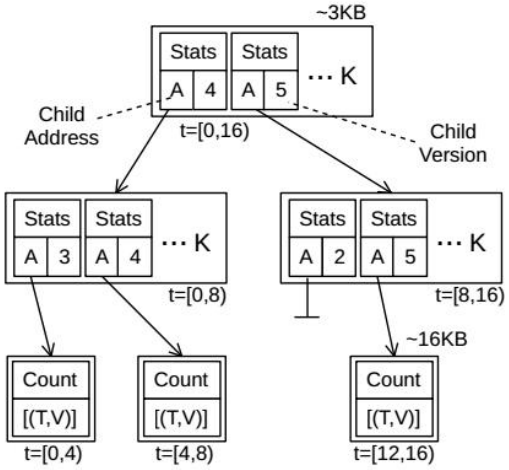


图 3. 带注释边的时间分割树的一个示例，k＝64 对应于节点大小

为了保留历史数据，该树在写入时被复制：树中的每个插入在前一个树上形成覆盖图，通过新的根节点可访问。以这种方式提供历史数据查询确保树的所有版本都需要相同的查询努力，这与日志重放机制不同，日志重放机制会导致开销与正在查询的版本有多少数据已更改或多少年份成比例。使用存储结构作为索引可确保对任何版本的流的查询都有索引使用，并减少网络往返。

树中的每个链接都使用引入该链接的树的版本进行注释，如图 3 所示。具有非零版本注释的空子指针意味着版本是删除。在两个版本的树之间修改的时间范围可以通过加载对应于更高版本的树来进行，并且降序到用开始版本或更高版本注释的所有节点中。树只需要走到所期望的差异分辨率的深度，因此 ComputeDiff（）在不读取原始数据的情况下返回其结果。这种机制允许流的消费者查询和处理新的数据，而不管在哪里进行更改，没有完整的扫描，并且只需要 8 个字节的状态维护“最后的版本处理”。

每个内部节点保存下面的子树的标量摘要，以及到子树的链接。在修改或插入叶节点之后，统计聚合被计算为更新节点。目前支持的统计信息是 min，mean，max 和 count，但是可以使用任何使用子树的中间结果而不需要迭代原始数据的操作。 任何关联操作都可以满足这个要求。

这种方法比传统的离散汇总有几个优点。IO 操作（分布式存储系统中最昂贵的部分）的概要计算是免费的。 所有用于计算的数据都已经存储在内存中，并且无论如何都需要复制内部节点，因为它包含新的子地址。总结的确增加了内部节点的大小，但即便如此，内部节点也只占总占地面积的很小一部分（K = 64 单树型的单个版本<0.3％）。可观察的统计数据保证与底层数据一致，因为在计算过程中失败会阻止根节点被写入，整个覆盖层将不可访问。

当查询流的统计记录时，树只需要遍历到与期望的分辨率相对应的深度，因此响应时间

与描述时间范围的返回记录的数量成正比，而不是范围的长度，它内的数据点。来自不同流的记录按时间排列，因此时间相关的分析可以直接进行。对于需要特定非功率窗口的查询，通过使用预先计算的统计信息来填充每个窗口的中间，操作仍然显着加速，只需要在每个窗口的侧面“下钻”，以便生成一个窗口的努力再次与它所覆盖的时间长度的对数成比例，而不是基础数据中的线性。

尽管在概念上是二叉树，但是实现可以通过使用 k-tree 树并且针对位于树的实际层之间的窗口的统计度量的刚好时间计算来交易增加的查询 - 时间计算以减少存储和 IO 操作。但是，如果 k 太大，动态计算的影响会增加统计查询延迟的可变性，正如 5.3 节所讨论的那样。

为了允许在单个 IO 操作中从树中获取节点，树中使用的所有地址都是“本地的”，因为它们可以直接由存储层解析而不需要转换步骤。如果使用间接地址，则需要在中央地图中进行昂贵的远程查找，或者需要复杂的机器来同步本地存储的地图。多个服务器一次可以在同一个流上执行读取操作，所以所有的服务器都需要这个映射的最新视图。原生地址完全消除了这个问题，但需要小心维护，如下所述。

子地址和子指针版本的内部块的基本大小为 2×8×K。此外，对于 K = 64，统计数据需要 4×8×K（最小值，平均值，最大值和计数），使得它们的大小为 3KB。叶节点需要每个（时间，值）对 16 个字节，以及 16 个字节的长度值。对于 Nlea f = 1024，他们是 16KB 大。这两个块都被压缩，如下所述。

4.系统设计

BTrDB 的整个系统设计（如图 4 所示）与上述多分辨率 COW 树数据结构紧密联系，也代表了复杂性，性能和可靠性之间的一系列折衷。这个设计优先考虑简单性，性能第二，然后是可靠性，尽管它们都做得非常好。排序是开发数据库的自然演变，可能需要频繁更改以匹配动态变化的问题域和工作负载（简单性导致易于修改的设计）。性能要求来源于文章正在部署的设备不可避免的要求，并且，由于该系统在生产中使用，可靠性需要尽可能高，而不会牺牲其他两个目标。

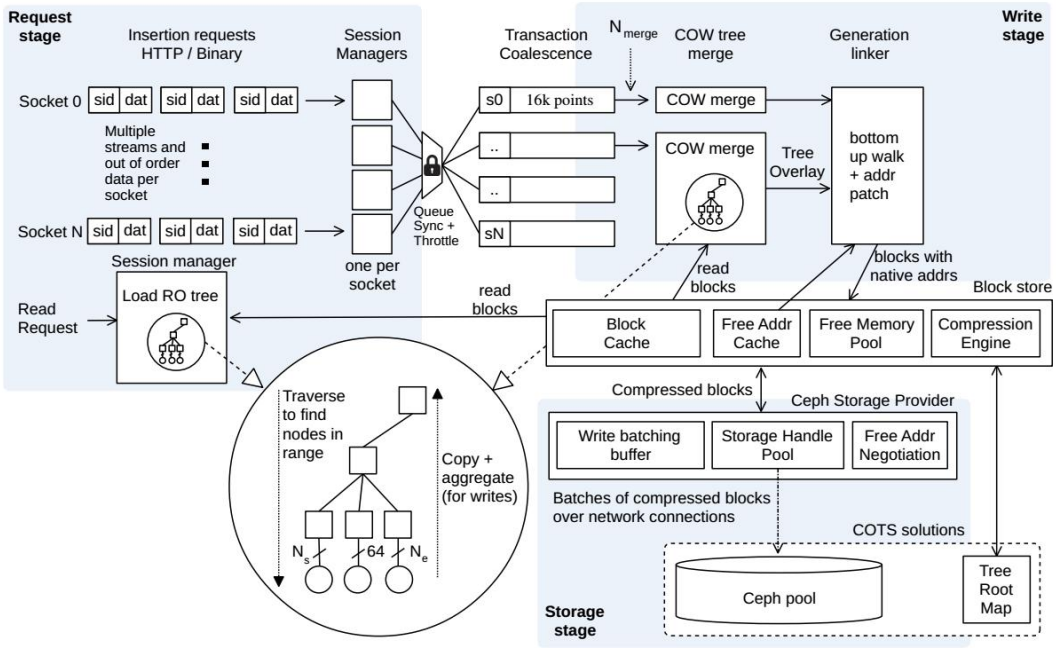


图 4. 显示三个 SEDA 型阶段的 BTrDB 概述，以及组成模块

设计包括几个模块：请求处理，事务合并，COW 树构建和合并，生成链接，块处理和块存储。该系统遵循 SEDA 模式，处理发生在三个资源控制阶段 - 请求，写入和存储 - 能够施加背压的队列将它们解耦。

4.1 请求处理阶段

在前端，通过多个套接字（二进制或 HTTP）接收插入和查询请求流。每个流都由 UUID 标识。许多流上的操作可以在单个套接字上到达，而针对特定流的操作可以分布在多个套接字上。插入是时间值对的集合，但不必按顺序排列。

插入和查询路径基本上是分开的。读取请求相对较轻，并在会话管理器的线程中处理。如上所述，这些构造和遍历 COW 树的局部视图，从块存储请求块。块存储反过来从可靠的存储提供者（在文章的实现中是 Ceph）请求块和最近使用的块的缓存。读取限制是通过存储阶段限制为会话线程提供多少存储句柄来加载块来实现的。命中缓存的请求仅受套接字输出的限制。

在插入路径上，输入数据通过 UUID 被解复用为每个流合并缓冲区。会话管理员竞争一个短暂的地图锁定，然后锁定所需的缓冲区。这个同步点提供了一个重要的写入限制机制，如 7.3 节所述。每个流都被缓冲，直到经过一定的时间间隔或者到达一定数量的点，这会触发写入阶段的提交。 这些参数可以根据目标工作负载和平台进行调整，在流更新延迟，流的数量，内存压力以及每个版本提交的树和块开销的比率方面有明显的折衷。这些缓冲区不需要很大，文章在运行中表现出了出色的存储利用率，其中缓冲区配置为最大提交 5 秒或 16k 点，平均提交大小为 14400 点。

4.2 COW 合并

写阶段中的一组线程接收等待提交的缓冲区并构建一个可写的树。除了所有遍历的节点被修改为合并的一部分之外，这个过程与读取请求完成的树相似，因此必须保留在内存中。复制现有节点或创建新节点需要从块存储中的空闲池中获取大量内存。此时，新创建的块具有临时地址，然后，链接器将解析这些地址以获得无索引本机寻址。

4.3 块存储

块存储分配空块，存储新块并获取存储块。它还提供存储块的压缩/解压缩和一个缓存。在树合并中使用的空块主要从免费池中满足以避免分配。 块被从块缓存中逐出并即将被垃圾收集后，它们被插回到这个池中。

诸如块地址，UUID，分辨率（树深度）和时间范围之类的字段对于遍历树是有用的，但是当从磁盘读取块时可以从上下文推断出，所以在块进入压缩引擎之前将其剥离。

块缓存保存所有通过块存储的块，其中最先使用的块最先被清除。它消耗了内存占用的一个重要（可调）部分。除了 COW 树中的内部节点之外，时间序列存储的高速缓存似乎不是一个明显的胜利，但对于近实时分析而言，这一点非常重要。由于文章的大部分读取工作负载都是等待消耗对一组数据流的任何更改的进程（即刚刚通过系统的数据），最近使用的块的缓存大大提高了性能。

4.4 压缩引擎

作为块存储的一部分，压缩引擎压缩内部节点中的最小值，平均值，最大值，计数，地址和版本字段，以及叶节点中的时间和值字段。它使用了一种我们称之为 deltadelta 编码的方法，然后是使用固定树的霍夫曼编码。典型的增量编码通过计算序列中每个值之间的差异，并使用可变长度符号（因为增量通常小于绝对值[18]）来存储。不幸的是，对于高精度的传感器数据，这个过程不能很好地工作，因为纳秒时间戳会产生非常大的变化量，甚至线性变化的值会产生很大但相似的变化量值序列。

在较低精度的流中，通常使用运行长度编码去除相同三角形的长流，从而去除相同三角形的序列。不幸的是，高精度传感器值的较低位中的噪声会妨碍运行长度编码成功压缩三角洲序列。但是，这种噪声只会给 delta 值增加一个小的抖动。 他们是非常相似的。Delta-delta 压缩替代了运行长度编码，并将每个增量编码为与先前增量值的窗口平均值的差异。结果是只有抖动值的序列。顺便说一句，这对于地址和版本号很合适，因为它们也是线性递增值，而且在 delta 中有一些抖动。 在系统开发过程中，文章发现这个算法比 FLAC 中的残差编码产生更好的结果，实现起来比较简单，这是最初的启发。

这种方法只有与整数一起使用才是无损的。为了克服这个问题，双浮点数值被分解为尾数和指数，而 delta-delta 压缩为独立的数据流。由于指数字段很少发生变化，因此如果 delta-delta 值为零，则会被忽略。

虽然这种压缩算法的定量和比较分析超出了本文的范围，其功能如第 5 节所示。

4.5 代连接器

代连接器从 COW 合并过程接收新的树覆盖，将新树节点从最深到最浅排序，并将它们分别发送到块存储，同时将临时地址解析为底层存储提供者本地的地址。 由于节点只引用已经写入的比自己更深的节点，遇到的任何临时地址都可以立即解析为本地地址。

这个阶段是必需的，因为大多数高效的存储提供者（例如仅追加日志）只能将任意大小的对象写入特定的地址。在简单文件的情况下，任意大小的对象只能写入尾部，否则会覆盖现有的数据。一旦对象的大小已知，例如在链接器将对象发送到块存储器并且被压缩之后，可以从前一个导出新的地址。存储的性质可能会限制从给定的初始地址可以派生出多少个地址。例如，如果达到最大文件大小并需要使用新文件。

对于某些存储提供商来说，获得第一个地址是一个昂贵的操作， 在集群操作中，这可能涉及获得分布式锁以确保生成的地址的唯一性。为此，块存储维护一个预先创建的初始地址池。

4.6 根构图

根构图在树构造之前在读取和写入中使用。它将 UUID 和版本解析为存储“地址”。当新版本的块已被存储提供程序持久保留时，该版本的新映射将插入到此根映射中。地图具有容错性是非常重要的，因为它代表了单点故障。没有这个映射，就不能访问流。如果从映射中删除最新的流版本条目，则在逻辑上相当于回滚提交。顺便提一下，由于少数孤行版本的存储成本较低，因此可以有意使用此行为来获取廉价的单流事务语义，而不需要数据库中的支持代码。

这些插入/请求的要求远远低于实际数据所要求的要求，所以许多现成的解决方案都可以提供这个设计的组件。文章使用 MongoDB，因为它具有易于使用的复制。

选择外部提供者的一个副作用是，解决这个第一查找的延迟是存在于所有查询中的，甚至是针对查询的其余部分的缓存。由于此地图的尺寸较小，因此在所有 BTrDB 节点上复制地

图并使用更简单的存储解决方案来减少此延迟是合理的。所有的记录都是相同的大小，并且版本号依次递增，所以通过偏移索引的平面文件是可接受的。

4.7 存储提供

存储提供程序组件包装基础的持久存储系统，并添加写入批处理，预取和连接句柄池。在 BTrDB 中，只要可以为提交中的所有节点生成地址，而不需要执行与底层存储的中间通信，就可以将树提交作为单个写入来执行。出于 6.3 节所述的原因，在这里实施对底层存储的限制。

由于存储系统的性能通常随着其功能的丰富性而下降，所以 BTrDB 被设计为仅需要来自底层存储的三个非常简单的条件：

1. 必须能够提供一个或多个免费的“地址”，以便随后可以写入任意大的对象。这些地址中只有少量的有限数量需要一次突出。
2. 客户端必须能够从原始地址中获得另一个空闲的“地址”，以及写入该地址的对象的大小。
3. 客户必须能够读回只有“地址”和长度的数据。

基于整个 BTrDB 部署的期望特性，可能需要额外的属性，例如分布式操作和耐久性写道。没有这些额外的要求，即使是一个简单的文件就足够作为存储提供者：（1）是文件的当前大小，（2）是加法，（3）是随机读取。

请注意，由于文章使用类似文件的 API 来追加和读取数据，因此几乎每个分布式文件系统都可以自动符合 HDFS，GlusterFS [25]，CephFS [32]，MapR-FS 等标准。

还要注意的是，如果组成任意但唯一的“地址”，则任何提供键值 API 的数据库也可以工作，例如，Cassandra，MongoDB，RADOS [33]（CephFS 下的对象存储），HBase 或 BigTable。其中大多数功能的功能远远超出 BTrDB 所要求的功能，但通常是以性能或空间成本。

虽然文章支持文件备份存储，但文章在生产中使用 Ceph RADOS。 初始地址是从存储在 RADOS 对象中的单调递增的整数中读取的。服务器在保持分布式锁（由 Ceph 提供）的同时向这个整数增加一个大增量。服务器然后有一个数字它知道是唯一的。高位用作 16MB RADOS 对象标识符，低位用作该对象内的偏移量。块存储中的地址池将此操作的延迟与写入操作解耦。

5.准生产实施

使用 Go [11]构建了 BTrDB 的实现。 选择这种语言是因为它提供了原始语言，可以在 SEDA 范式（即频道和 goroutine）中快速开发高度可伸缩的 SMPs 程序。如上所述，BTrDB 的主要原则之一是通过简单的性能：整个实现 sans 测试代码和自动生成的库只有 4709 行。

BTrDB 的各种版本已经用于一年的部署，以捕获野外部署的大约 35 个微阵列相机的数据，每个数据包含 120 个数据流。如图 2 所示，来自这些设备的数据流通过 LTE 和有线连接，导致无法预测的延迟，无序块传输和许多重复（当 GPS 导出时间与不同卫星同步时）。BTrDB 中的许多功能都是为了支持传感器数据的存储和分析而开发的。

此部署的硬件配置如图 5 所示。计算服务器运行 BTrDB（在单个节点配置中）。它还运行 DISTIL 分析框架[1]和一个 MongoDB 副本集主。MongoDB 数据库用于根映射以及各种元数据，例如流的工程单位和 DISTIL 算法的配置参数。存储服务器是包含 28 个商品 4TB 5900 RPM 旋转金属驱动器的单路服务器。对于高性能数据库而言，此服务器的 IO 容量可能看起来异

常低，但对于数据仓库应用程序来说，这是典型的。BTrDB 的 IO 模式选择了这种类型的服务器：1MB 的读取和写入，具有卓越的数据局部性，用于主分析工作负载。

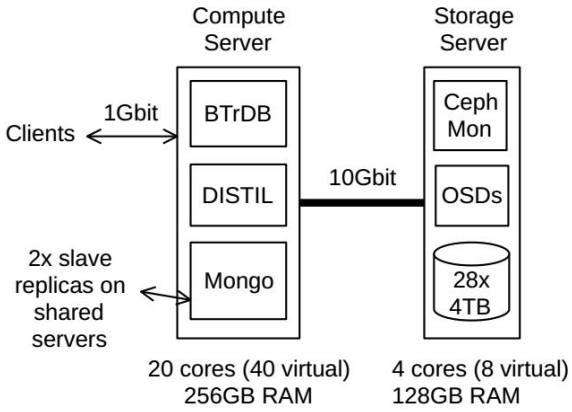


图 5. 文章提出的运行系统架构

5.1Golang - SEDA 的化身

SEDA 倡导通过分解构建可靠的，高性能的系统进入独立的阶段，这些独立的阶段由进入控制的队列隔开。虽然没有明确提到这个范例，但是 Go 鼓励复杂的分区系统转换为并发逻辑单元，通过通道连接，Go 原语大致等于一个带有原子入队和出队操作的 FIFO。另外，Goroutines--一个极其轻便的线程式原始数据库（userland scheduling） - 允许为系统的组件分配 goroutines 池来处理连接系统的通道上的事件，这与 SEDA 主张事件调度的方式非常相似。与 SEDA 的 Java 实现不同的是，Go 被主动维护，运行速度接近本地速度，并且可以优雅地操作二进制数据。

5.2 读取节流

如下所述，仔细施加背压是获得良好写入性能所必需的。相比之下，尽管读取负载高于写入负载，但文章还没有发现需要显式地节制读取。保存在内存中以满足读取的块的数量少于写入的块的数量。如果节点不在块缓存中（其中 95％是），则仅在遍历它们的子树时才需要它们，然后可以释放它们。这与写操作不同，在写操作中，所有遍历的块都将被复制，因此必须保存在内存中，直到连接器将其修补并写入存储提供者。另外，Go 通道用于在查询时直接将查询数据传输到套接字。如果套接字速度太慢，通道会向树遍历施加反压，这样在数据到达某个位置之前，节点不会被获取。由于这个原因，即使是大的查询也不会给系统带来沉重的内存压力。

5.3 实时数据定量评估

尽管在生产环境中运行的 BTrDB 版本缺乏在第 7 节中评估的版本上实现的性能优化，但它可以提供对具有大量实际数据集的数据库行为的深入了解。在撰写本文时，文章累积了超过 2 万 3 千亿个数据点，其中有超过 823 个数据流，其中有 5000 亿个数据点来自部署在该领域的仪器。其余的 1.6 万亿分是由 DISTIL 分析框架产生的。在这个分析数据中，由于算法改变，手动标记或者替换输入流中的数据，导致大约 1.1 万亿分点被无效。这个庞大的数据集使文章能够评估设计的几个方面。

压缩：使用具有“重量级”内部节点的写入时复制树数据结构的问题是存储开销可能是不可接受的。 使用真实的数据，压缩不仅可以弥补这种开销。生产 Ceph 池中仪器数据的总大小（不包括复制）为 2.757 TB。除以原始数据点的数量等于每个读数 5.514 字节，包括所有统计和历史开销。由于原始元组是 16 个字节，所以文章的压缩比是 2.9 倍，尽管时间分割树的成本。压缩是高度依赖于数据的，但是这个比值比在类似的同步相量遥测中对压缩的深入参数研究的结果好[16] [30]。

统计查询：由于这些查询发挥更大的数据集，他们是最好的评估数月的真实数据，而不是在第 6 节的受控研究。这些查询通常用于事件检测器定位感兴趣的领域 - 原始数据太大而无法轻松导航 - 并且可视化。为了模拟这种工作负载，文章查询一年的电压数据（与图 1a 所示的数据相同），找到一个电压骤降（虚线框），然后发出逐渐细化的查询，直到查询 5 秒的窗口（图 1b）。自动事件检测器通常在查询之间跳过几个级别的分辨率，但是这种模式是用户以交互方式放大事件的数据探索的典型特征。这个过程重复了 300 次，在每个序列之间暂停以获得查询响应时间的分布。结果可以在图 6 中找到。通常情况下，这些分布将更加紧密，但生产服务器负载很重。

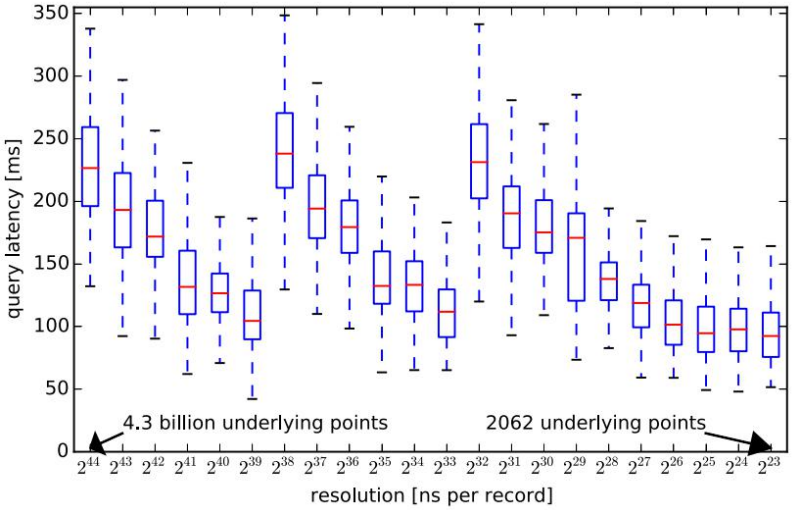


图 6.查询覆盖时间范围（1 年到 5 秒）的 2048 个统计记录的查询延迟，从单个节点查询

每个查询的统计记录数是相同的（2048），但是这些记录表示的数据点的数量随着分辨率变粗糙（图 6 中的从右到左）而呈指数增长。在一个典型的即时汇总数据库中，查询时间也会呈指数增长，但是对于 BTrDB，它在三分之一内保持大致恒定。在查询响应时间内，实现选择 K（64 = 26）非常明显。查询可以直接从内部节点满足，不需要每 6 个分辨率级别的运行计算。在这些级别之间，BTrDB 必须执行一定程度的聚合 - 在查询延迟中可见 - 返回统计摘要，大部分工作发生在树的下一层（244,238,232）之前。在 227 以下，数据密度足够低（每个像素列<16 个点），从叶子中查询得到满足。

缓存命中率：尽管缓存行为是依赖于工作负载的，但文章的大多数自动分析可能代表了大多数使用情况。超过 22 天的时间，块缓存命中率达到 95.93％，Ceph 后置预取缓存命中率达到 95.22％。

5.4 管道分析

从现场传感器获得的原始数据最终用于决策支持; 网格状态估计; 岛检测和反向功率流检测等等。为了获得有用的信息，数据必须首先通过由多个转换，融合和合成阶段组成的流水线，如图 2 所示。

流水线的所有阶段都使用相同的分析框架来实现，并且都由相同的操作序列组成：查找输入中的变化，计算哪些输出范围需要更新，获取计算所需的数据，计算 输出，并插入它们。最后，如果这个过程成功完成，蒸馏器现在“追上”的输入的版本号被写入到持久存储器（用于根映射的同一个 MongoDB 副本集）。这种架构允许没有机制的容错，因为每个计算都是幂等的：与给定输入范围相对应的输出范围被删除并替换为每个 DISTIL 阶段的运行。如果发生任何错误，只需在更新输入流的“输入→上一版本”元数据记录之前重新运行该阶段，直至成功。

这说明了 BTrDB CalculateDiff（）基元的强大功能：分析流可以被“搁置”，即不保持最新状态，当它变得必要时，可以在保证时间的情况下及时更新一致性，即使对整个流的随机时间发生依赖关系的更改。此外，消费者只需要每个数据流 8 个字节的状态即可获得。该机制允许将流中的变化传播到依赖于它的所有流，即使实现依赖流的过程不在线或不知道上游进行变更的过程。在现有系统中实现这一级别的一致性保证通常需要一个未完成的操作日志，当下游消费者再次出现时，这些操作必须重播。

6.可扩展性评估

为了以可重复的方式评估 BTrDB 的设计原则和实现，文章使用了七个 Amazon EC2 实例的配置。有四个主服务器，一个元数据服务器和两个负载生成器。这些机器都是 c4.8xlarge 的实例。选择这些是因为它们是具有 10GbE 网络功能和 EBS 优化的唯一可用实例类型。这种组合允许在网络和磁盘带宽是限制因素的多节点配置中建立 BTrDB 的可扩展性。

使用 Ceph 0.94.3 版本来提供超过 16 个对象存储守护进程（OSD）的存储池。它配置了两个大小（复制因子）。表 2 中显示了池的带宽特性。重要的是要注意 Ceph 操作的延迟，因为这会在冷查询延迟上建立一个下限，并与交易合并背压机制相互作用。给定 BTrDB 节点上的某个 OSD 卷的磁盘带宽大约为 175MB / s。这与 ceph tell osd.N bench 报告的 OSD 的性能相匹配。

表 2. 底层 Ceph 池在最大带宽下的性能

Metric	Mean	Std. dev.
Write bandwidth [MB/s]	833	151
Write latency [ms]	34.3	40.0
Read bandwidth [MB/s]	1174	3.8
Read latency [ms]	22.0	18.7

为了保持这些特性大致不变，Ceph 节点的数量保持在四，不管有多少服务器运行 BTrDB 给定的实验，尽管池的带宽和延迟随时间变化。由于 Ceph CRUSH 数据放置规则与 BTrDB 放置规则是正交的，因此 RADOS 请求击中本地 OSD 的概率对于所有实验都是 0.25。

6.1 吞吐量

BTrDB 以每秒原始记录元组的吞吐量是针对插入，冷查询（在冲刷 BTrDB 缓存之后）和热查询（具有预热的 BTrDB 缓存）而测量的。 每个元组都是一个 8 字节的时间戳和一个 8 字节的值。 温度和冷藏缓存的性能是独立表征的，因为它可以在估计缓存命中率之后估算不同工作负载下的性能。

表 1. 吞吐量评估随着服务器数量和负载的大小而增加

#BTrDB	Streams	Total points	#Conn	Insert [mil/s]	Cold Query [mil/s]	Warm Query [mil/s]
1	50	500 mil	30	16.77	9.79	33.54
2	100	1000 mil	60	28.13	17.23	61.44
3	150	1500 mil	90	36.68	22.05	78.47
4	200	2000 mil	120	53.35	33.67	119.87

插入和查询是以 10 千字块完成的，但是如果将其减少到 2 千字，性能没有显着变化。

图 7 显示了插入吞吐量线性扩展到四个节点每秒约 5300 万条记录。水平虚线计算为最大测量池带宽（823MB / s）除以原始记录大小（16 字节）。这是通过简单地将记录附加到 Ceph 中的日志而无需任何处理即可实现的带宽。这表明，尽管 BTrDB 提供的功能以及必须存储的附加统计值，但 BTrDB 却能与理想的数据记录器相媲美。

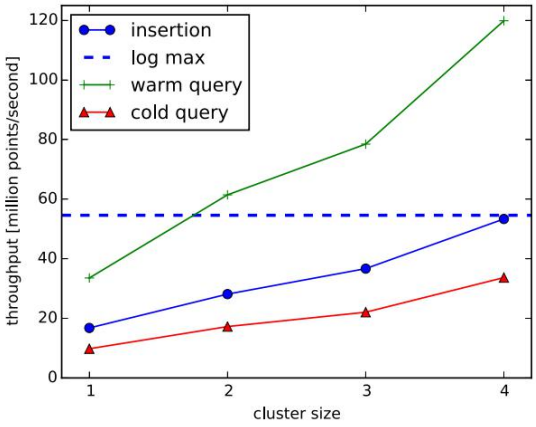


图 7. 吞吐量随着 BTrDB 节点数量的增加而增加，水平虚线表示底层存储系统的独立基准写入带宽

对于蒸馏器最近处理数据变化的拖尾分析工作负载，热量查询吞吐量为每秒 1.19 亿个读数。这个吞吐量相当于大约 1815 MB / s 的网络流量，或每个负载生成器 907MB / s。

6.2 数据和操作顺序

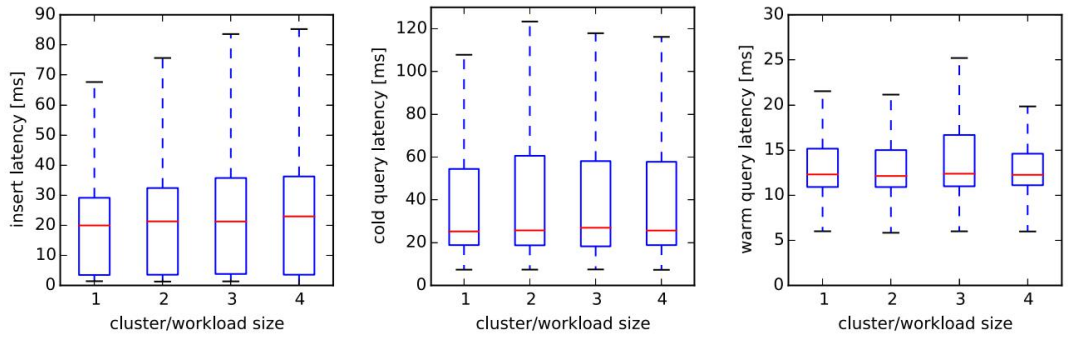
BTrDB 允许以任意顺序插入数据，并以任意顺序查询。为了表征插入和查询顺序对吞吐量的影响，使用随机操作进行测量。工作量包括每个 10k 点（总共 20 亿点）的二十万个插入/查询。构建两个数据集，一个是按时间顺序插入数据，另一个是随机插入数据。在这之后，在两个数据集上测试按时间顺序和随机顺序的冷和热查询的性能。对于随机插入的情况，也以与插入相同（非按时间顺序）的顺序进行查询。请注意，操作是按请求的粒度随机化的；在每个请求中，10k 点仍然是有序的。结果列于表 3 中。吞吐量的差异完全在实验噪声范围内，并且在很大程度上是微不足道的。这种无序的性能对于提供接近于按顺序附加日志的插入速度的数据库是重要的结果。

表 3. 查询/插入顺序对吞吐量的影响

	When insertion was	
Throughput [million pt/s] for	Chrono.	Random
Insert	28.12	27.73
Cold query in chrono. order	31.41	31.67
Cold query in same order	-	32.61
Cold query in random order	29.67	28.26
Warm query in chrono. order	114.1	116.2
Warm query in same order	-	119.0
Warm query in random order	113.7	117.2

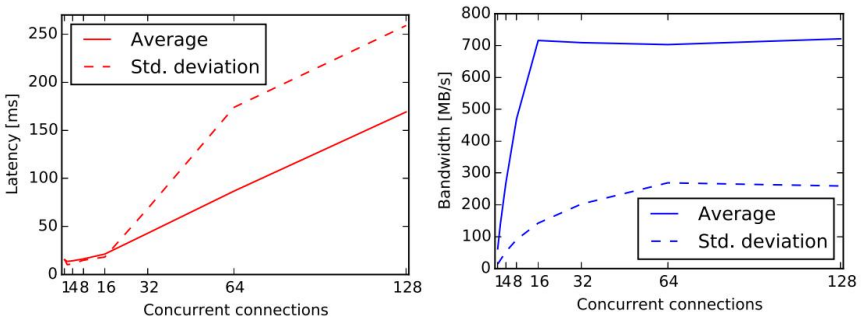
6.3 延迟

尽管 BTrDB 被设计为在吞吐量大幅增加的情况下交易延迟的小幅增加，但延迟仍然是评估负载下性能的重要指标。负载生成器记录每个插入或查询操作所用的时间。图 8 给出了随着服务器工作负载和数量的增长，操作延迟的概况。理想情况下，所有四个点将相等，表示完美的缩放。插入操作的延迟范围随着集群接近 Ceph 的最大带宽而增加。这完全是 Ceph 作为背压呈现给客户的延迟。当事务合并缓冲区已满或正在提交时，不会将指定给该流的数据导入数据库。此外，一次允许固定数量的树合并，所以一些缓冲区可能会保持一段时间。虽然这看起来是违反直觉的，实际上它增加了系统性能。尽早应用这种背压可防止 Ceph 达到病理性潜伏期。由图 9a 可以明显看出，不仅 Ceph 操作延迟随着并发写入操作的数量而增加，而且形成了长长的尾巴，标准偏差超过了平均值。此外，这个等待时间什么也没有买，如图 9b 所示，在并发操作数之后的总带宽达到了 16，即 OSD 的数量。



(a) Insert latencies (b) Cold query latencies (c) Warm query latencies

图 8. 作为服务器计数和工作负载的操作延迟线性增加



(a) Latency (b) Aggregate bandwidth

图 9. 随着并发连接数量的增加，Ceph 池的性能特征

考虑到 Ceph 的延迟特性，BTrDB 在最大负载下的写入延迟非常显着。插入超过 5300 万个点/秒的四节点群集表现出 35 毫秒的三分之一延迟：小于比原始池延迟更高的一个标准偏差。

6.4 存在的限制和未来工作

EC2 上的测试表明，系统的吞吐量和延迟特性主要由底层存储系统来定义。这是理想的地方，因为它使得时间序列数据库层中的最优化不再相关。

这个的例外是优化，减少 IO 操作的数量。文章已经将写入路径优化为每次提交一次写入操作。但是，通过优化读取路径将有显着的性能提高。一个途径是改进块缓存策略，减少读操作。目前，缓存清除了最近最少使用的块。更复杂的策略可能会提高缓存利用率：例如，

如果客户端只查询流的最新版本，则在树合并操作期间复制的所有块的原始数据都可能从缓存中被逐出。如果大多数客户端正在执行统计查询，那么叶节点（比内部节点大 5 倍）可以优先进行驱逐。而且，由于块是不变的，所以分布式缓存并不难实现，因为不需要一致性算法。从内存中查询对等 BTrDB 服务器会比通过 Ceph 访问磁盘更快。

7.结论

BTrDB 提供了一套新颖的基元，特别是快速差异计算和快速，低开销的统计查询，使分析算法能够在几十亿个数据点的数据中定位亚秒级瞬态事件，所有这些都在几分之一秒内完成。 这些原语通过时间分区版本注释的写时复制树来高效地提供，这被显示为容易实现的。Go 实现的性能优于现有时间序列数据库，每秒运行 5300 万个插入值，每四个节点集群每秒查询值为 1.19 亿个。这个数据库的基本原理可能适用于广泛的遥测时间序列，并且稍作修改，适用于存在统计聚合函数并按时间索引的所有时间序列。

参考文献

[1] ANDERSEN, M. P., KUMAR, S., BROOKS, C., VON MEIER, A., AND CULLER, D. E. DISTIL: Design and Implementation of a Scalable Synchrophasor Data Processing System. Smart Grid, IEEE Transactions on (2015).

[2] APACHE SOFTWARE FOUNDATION. Apache Cassandra home page. http://cassandra.apache.org/, 9 2015.

[3] APACHE SOFTWARE FOUNDATION. Apache HBase home page. http://hbase.apache.org/, 9 2015.

[4] BUEVICH, M., WRIGHT, A., SARGENT, R., AND ROWE, A. Respawn: A distributed multiresolution time-series datastore. In Real-Time Systems Symposium (RTSS), 2013 IEEE 34th (2013), IEEE, pp. 288 – 297.

[5] COUCHBASE. CouchBase home page. http: //www.couchbase.com/, 9 2015.

[6] DATASTAX. DataStax home page. http://www.datastax.com/, 9 2015.

[7] DRUID. Druid.io. http://druid.io/, 9 2015.

[8] DUNNING, TED. MapR: High Performance Time Series Databases. http: //www.slideshare.net/NoSQLmatters/teddunning-very-high-bandwidth-timeseries database-implementation-nosqlmatters-barcelona-2014, 11 2014.

[9] ENDPOINT. Benchmarking Top NoSQL Databases. Tech. rep., Endpoint, apr 2015. http://www.datastax.com/wpcontent/themes/datastax-2014-08/files/ NoSQL_Benchmarks_EndPoint.pdf.

[10] GOLDSCHMIDT, T., JANSEN, A., KOZIOLEK, H., DOPPELHAMER, J., AND BREIVOLD, H. P. Scalability and robustness of time-series databases for cloud-native monitoring of industrial processes. In Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on (2014), IEEE, pp. 602 – 609.

[11] GOOGLE. The Go Programming Language. https://golang.org/, 9 2015.

[12] INFLUXDB. InfluxDB home page. https://influxdb.com/, 9 2015.

[13] JOSH COALSON. Free Lossless Audio Codec. https://xiph.org/flac/, 9 2015.

[14] KAIROSDB. KairosDB import/export documentation. https://kairosdb.github.io/

kairosdocs/ImportExport.html, 9 2014.
[15] KAIROSDB. KairosDB home page. http://github.com/kairosdb/kairosdb, 9 2015.
[16] KLUMP, R., AGARWAL, P., TATE, J. E., AND KHURANA, H. Lossless compression of synchronized phasor measurements. In Power and Energy Society General Meeting, 2010 IEEE (2010), IEEE, pp. 1–7.
[17] LAMBDA FOUNDRY. Pandas home page. http://pandas.pydata.org/, 9 2015.
[18] LELEWER, D. A., AND HIRSCHBERG, D. S. Data compression. ACM Computing Surveys (CSUR) 19, 3 (1987), 261–296.
[19] MONGODB INC. MongoDB home page. https://www.mongodb.org/, 9 2015.
[20] OPENTSDB. OpenTSDB home page. http://opentsdb.net/, 9 2015.
[21] ORACLE CORPORATION. MySQL home page.https://www.mysql.com/, 9 2015.
[22] PELKONEN, T., FRANKLIN, S., TELLER, J., CAVALLARO, P., HUANG, Q., MEZA, J., AND VEERARAGHAVAN, K. Gorilla: a fast, scalable, inmemory time series database. Proceedings of the VLDB Endowment 8, 12 (2015), 1816–1827.
[23] PROJECT VOLDEMORT. Project Voldemort home page. http://www.project-voldemort.com/, 9 2015.
[24] RABL, T., GOMEZ´ -VILLAMOR, S., SADOGHI, M., MUNTES´ -MULERO, V., JACOBSEN, H.-A., AND MANKOVSKII, S. Solving big data challenges for enterprise application performance management. Proceedings of the VLDB Endowment 5, 12 (2012), 1724–1735.
[25] REDHAT. Gluster home page. http://www.gluster.org/, 9 2015.
[26] REDIS LABS. Redis home page. http://redis.io/, 9 2015.
[27] SCOTT, JIM. MapR-DB OpenTSDB bulk inserter code. https://github.
com/mapr-demos/opentsdb/commit/c732f817498db317b8078fa5b53441a9ec0766ce,
9 2014.
[28] SCOTT, JIM. MapR: Loading a time series database at 100 million points per second.https://www.mapr.com/blog/loadingtime-series-database-100-millionpoints second, 9 2014.
[29] STONEBRAKER, M., AND WEISBERG, A. The voltdb main memory dbms. IEEE Data Eng. Bull. 36, 2 (2013), 21–27.
[30] TOP, P., AND BRENEMAN, J. Compressing Phasor Measurement data. In North American Power Symposium (NAPS), 2013 (Sept 2013), pp. 1–4.
[31] VOLTDB INC. VoltDB home page. https://voltdb.com/, 9 2015.
[32] WEIL, S. A., BRANDT, S. A., MILLER, E. L.,LONG, D. D., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In Proceedings of the 7th symposium on Operating systems design and implementation (2006), USENIX Association, pp. 307–320.
[33] WEIL, S. A., LEUNG, A. W., BRANDT, S. A., AND MALTZAHN, C. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing’07 (2007), ACM, pp. 35–44.
[34] WELSH, M., CULLER, D., AND BREWER, E. Seda: an architecture for well-conditioned, scalable internet services. ACM SIGOPS Operating Systems Review 35, 5 (2001), 230–243.