Dgraph：同步复制、事务和分布式图数据库

# 抽象的

Manish Jain  [manish@dgraph.i](mailto:manish@dgraph.io)o Dgraph Labs, Inc.

版本：0.8 最后更新：2021 年 3 月 1 日

Dgraph 用独特的分片解决了连接深度问题

Dgraph 是一个分布式图数据库，它提供水平可扩展性、分布式集群范围的 ACID 事务、低延迟任意深度连接、同步复制、高可用性和崩溃弹性。针对实时事务性工作负载，Dgraph 以优化连接和遍历的方式对数据进行分片和存储，同时仍提供数据检索和聚合。 Dgraph 的独特之处在于在执行单个连接所需的恒定数量的网络调用（通常只有一个网络调用）中提供低延迟的任意深度连接，而不管集群的大小或大小结果集的。

# 介绍

分布式系统或数据库往往会遇到连接深度问题。也就是说，随着查询中关系遍历次数的增加，所需的网络调用次数（在足够分片的数据集中）也会增加。这通常是由于基于实体的数据分片，其中实体随机（有时带有启发式）分布在包含所有关系和属性的服务器上。这种方法在图查询的中间步骤中受到高扇出结果集的影响，导致它们在集群中进行广播以对实体执行连接。因此，单个图查询会导致网络广播，从而导致查询延迟随着集群的增长而跳跃。

Dgraph 是一个具有原生图形后端的分布式数据库。它是唯一可水平扩展并支持完全符合 ACID 的集群范围分布式事务的原生图形数据库。事实上，Dgraph 是第一个经过 Jepsen [?] 事务一致性测试的图数据库。

随着数据量或服务器数量的变化，Dgraph 自动将数据分片到机器中，并自动重新分片以跨服务器移动数据以平衡负载。它还支持由 Raft [?] 协议支持的同步复制，允许查询无缝故障转移以提供高可用性。

机制。 Dgraph 不是像大多数系统那样按实体进行分片，而是按关系进行分片。 Dgraph 独特的数据分片方式受到 Google [?] 研究的启发，该研究表明查询的整体延迟大于最慢组件的延迟。查询涉及执行的服务器越多，查询延迟就越慢。通过进行基于关系的分片，Dgraph 可以在单个网络调用中执行连接或遍历（如果第一个网络调用较慢，则备份网**络**调用），而不管集群的大小或实体的输入集。 Dgraph 执行任意深度的连接，无需网络广播或在中心位置收集数据。这允许查询快速且延迟低且可预测。

# 图形架构

Dgraph 由 Zeros 和 Alphas 组成，每个代表他们所服务的一个组。零为第 0 组服务，阿尔法为第 1 组、第 2 组及以后的组服务。每个组形成一个由 1、3 或 5 个成员组成的 Raft 集群，可由人工操作员（以下简称操作员）进行配置。对组所做的所有更新都通过 Raft 共识算法进行序列化，并按该顺序应用于领导者和追随者。

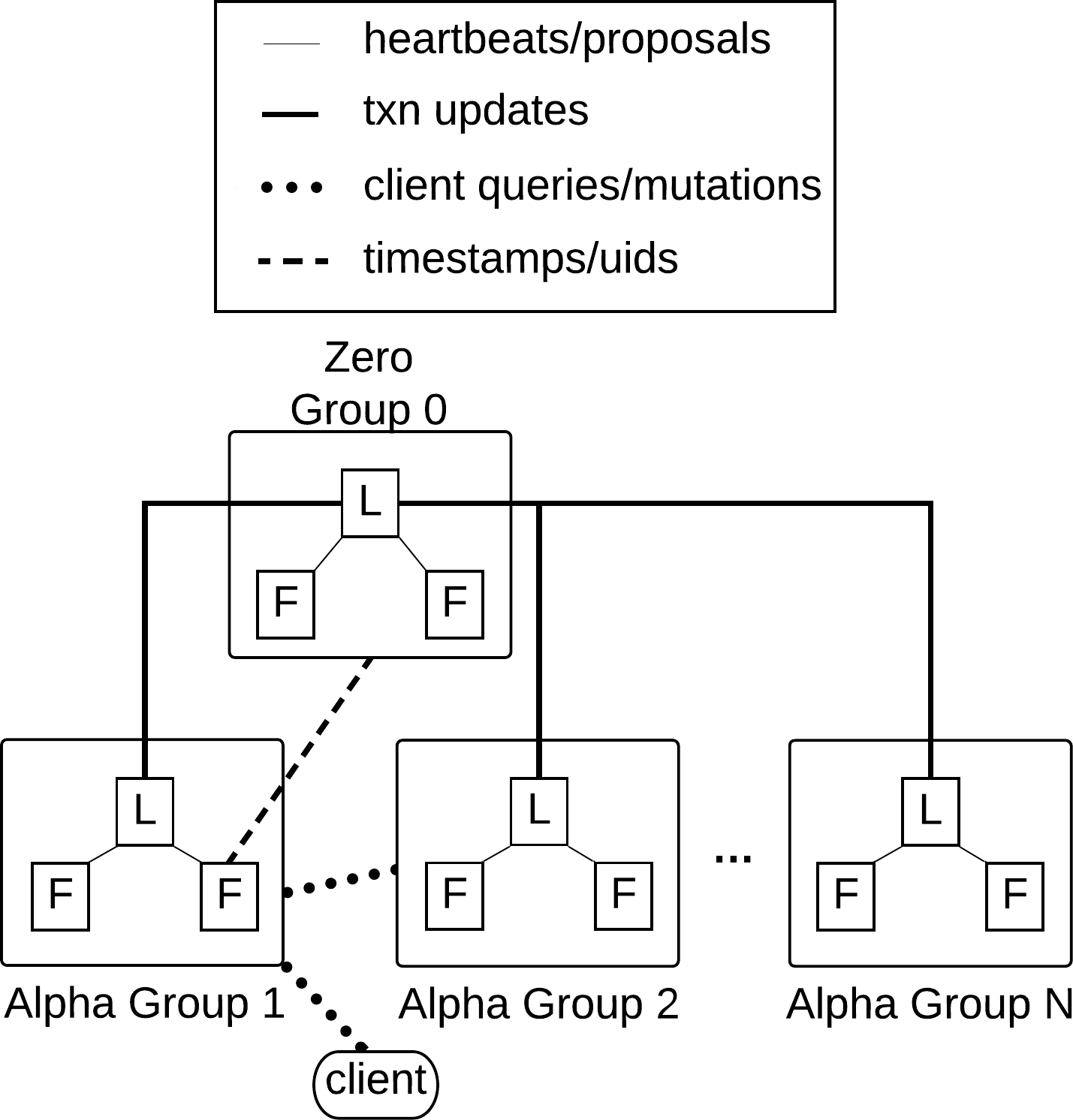
Zeros 存储和传播有关集群的元数据，而 Alphas 存储用户数据。特别是，Zeros 负责成员信息，它跟踪每个 Alpha 服务器所服务的组、它在集群内通信的内部 IP 地址、它所服务的分片等。 Zeros 不跟踪健康状况Alpha 并对其采取行动——这被认为是操作员的工作。使用此信息，零可以告诉新 Alpha 加入并为现有组提供服务，或组成一个新组。

会员信息从零流向所有Alpha。 Alpha 可以使用此成员资格信息来路由命中集群的查询（或突变）。集群中的每个实例都与其他每个实例形成一个连接（从而形成 2 (N) 个开放连接，其中 N

2

*×*

= 集群中 Dgraph 实例的数量），但是，

protocol buffer  **[**?] 数据格式和两者之间不能互换。

{

"uid": "0xab",

“类型”：“宇航员”，“姓名”：“马克沃特尼”，“出生”：“2005/01/02”，

“跟随者”：{“uid”：“0xbc”，…}，

}

<0xab> <type> "宇航员" 。

<0xab> <名称> “马克·沃特尼”。

<0xab> <出生> "2005/01/02" 。

<0xab> <追随者> <0xbc> 。

图 1：Dgraph 架构：有一个零组和多个 Alpha 组。每个组都是一个 Raft 组，由一个或多个成员组成。

这种连接的用法取决于它们的关系。例如，Raft leader-follower 关系会有心跳（每 100 毫秒）和数据流动，而 Alpha 只会在需要处理查询或突变时与另一组中的 Alpha 交谈。每个打开的连接都会进行轻量级的健康检查，以避免在无响应（死机、分区等）的目标服务器上停顿。 Alphas 和 Zeros 都公开了一个端口用于通过 Grpc [?] 进行集群内通信，另一个用于通过 HTTP 与客户端进行外部通信。 Alphas 还公开了一个外部 Grpc 端口，用于与基于 Grpc 的客户端进行通信——所有官方客户端都运行在 Grpc 上。

Zero 还运行一个预言机，它为集群中的事务分发单调递增的逻辑时间戳（与系统时间无关）。零领导者通常会通过 Raft 提议预先租用时间戳带宽，然后严格从内存中服务时间戳请求，无需任何进一步的协调。零 oracle 跟踪有助于事务提交的其他内容，这将在 ?? 部分详细说明。

零从 Alpha 领导者那里获取有关每个组中数据大小的信息，用于做出有关分片移动的决策，这将在 ?? 部分详细说明。

# 数据格式

Dgraph 可以输入 JSON 格式或（稍加修改的）RDF NQuad 格式的数据。 Dgraph 会将 JSON 映射分解为更小的块，每个 JSON 键值形成一个记录，相当于单个 RDF 三重记录。解析 RDF Triple 或 JSON 时，数据直接转换为内部

三元组通常表示为主语-谓语-宾语或主语-谓语-值。主体是节点，谓词是关系，客体可以是另一个节点或原始数据类型。一个从一个节点指向另一个节点，另一个从一个节点指向一个值。在上面的例子中，带有名称的三元组是一种主谓值（通常称为属性），而带有跟随者的三元组是一种主谓宾。 Dgraph 在处理这两种类型的记录方面没有区别（为了避免混淆这两种类型，我们将它们称为对象值）。 Dgraph 将其视为记录单元，典型的 JSON 映射会被分解为多个这样的记录。

可以使用 GraphQL [?] 和 GraphQL 的修改版本（称为 GraphQL+-[?]）**从**  Dgraph 中检索数据。 GraphQL+- 具有与 GraphQL 相同的大部分属性。但是，添加了对数据**库**很重要的各种属性，如查询变量、函数和块。有关查询语言如何形成以及 GraphQL 和 GraphQL+- 之间差异的更多信息，可以在这篇博文 [?] 中找到。

正如第 ?? 节所述，Dgraph 中的**所有**内部和外部通信都通过 Grpc 和 Protocol Buffers 运行。 Dgraph 还公开了 HTTP 端点，以允许使用这两种不支持的语言构建客户端库。 HTTP 端点和通过 Grpc 公开的 API 之间存在功能对等性。

根据 GraphQL 规范，来自 Dgraph 的查询响应采用 JSON 格式，通过 HTTP 和 Grpc。

# 数据存储

Dgraph 数据存储在称为 Badger [?] 的可嵌入键值数据库中，用于磁盘上的数据输入输出。 Badger 是一种基于 LSM 树的设计**，**但与其他设计不同的是，它可以选择性地将值与键分开存储，以生成更小的 LSM 树，从而降低写入和读取放大率。团队运行的各种基准测试

显示 Badger 提供与其他基于 LSM 的 DB 相同或更快的写入，同时提供与基于 B+-tree 的 DB（其读取速度往往比 LSM 树快得多）相同的读取延迟。

如上所述，具有相同谓词的所有记录形成一个分片。在一个分片中，共享相同主谓词的记录在 Badger 中被分组并压缩为一个单独的键值对。此值称为发布列表，这是搜索引擎中常用的术语，用于指代包含搜索词的文档 ID 排序列表。发布列表作为值存储在 Badger 中，键来自主语和谓词。

<0x01> <追随者> <0xab> 。

<0x01> <追随者> <0xbc> 。

<0x01> <追随者> <0xcd> 。

...

键 = <追随者，0x01>

值 = <0xab, 0xbc, 0xcd, ...>

Dgraph 中的所有主题都分配了一个全局唯一的 id，称为 uid 。 uid 存储为 64 位无符号整数 (uint64)，以允*许 Go*  语言在*代码库中*进行高效的本地处理。零负责根据 Alpha 的需要分发 uid，并以与时间戳相同的单调递增方式进行处理（第 ?? 部分）。一旦分配了 uid，就永远不会重新分配或重新分配。因此，图中的每个节点都可以由一个唯一的整数引用。

对象值存储在过帐中。每个帖子都有一个整数 ID。当发布包含一个对象时，id 是分配给该对象的 uid。当发布持有一个值时，值的整数 id 是根据谓词的模式确定的。如果谓词允许多个值，则该值的整数 id 将是该值的指纹。如果谓词使用语*言存储值*，则整数 id 将是语言标签的指纹。否则，整数 id 将设置为最大可能的 uint64 (264 - 1)。 uid 和整数 id 都不会设置为零。

值可以是许多受支持的数据类型之一：int、float、string、datetime、geo 等。数据被转换成二进制格式并与原始类型的信息一起存储在发布中。发布也可以包含方面。分面是边缘上的键值标签，被视为附件。

在谓词只有对象（没有像 follower edge 这样的值）的常见情况下，发布列表将主要由排序的 uids 组成。这些是通过进行整数压缩来优化的。 uid 被分组为 256 个整数（可配置）的块，其中每个块都有一个基本 uid 和一个二进制 blob。 blob  *是通过*获取当前 uid 与最后一个 uid 的差异并存储使用 group varint 编码的字节差异来生*成的。这*产生了 10 的数据压缩率。在做交集时，我们可以使用这些块进行二分查找或块跳转，以避免解码所有块。有序整数编码是一个热门研究话题，这里有*很大的*优化空间

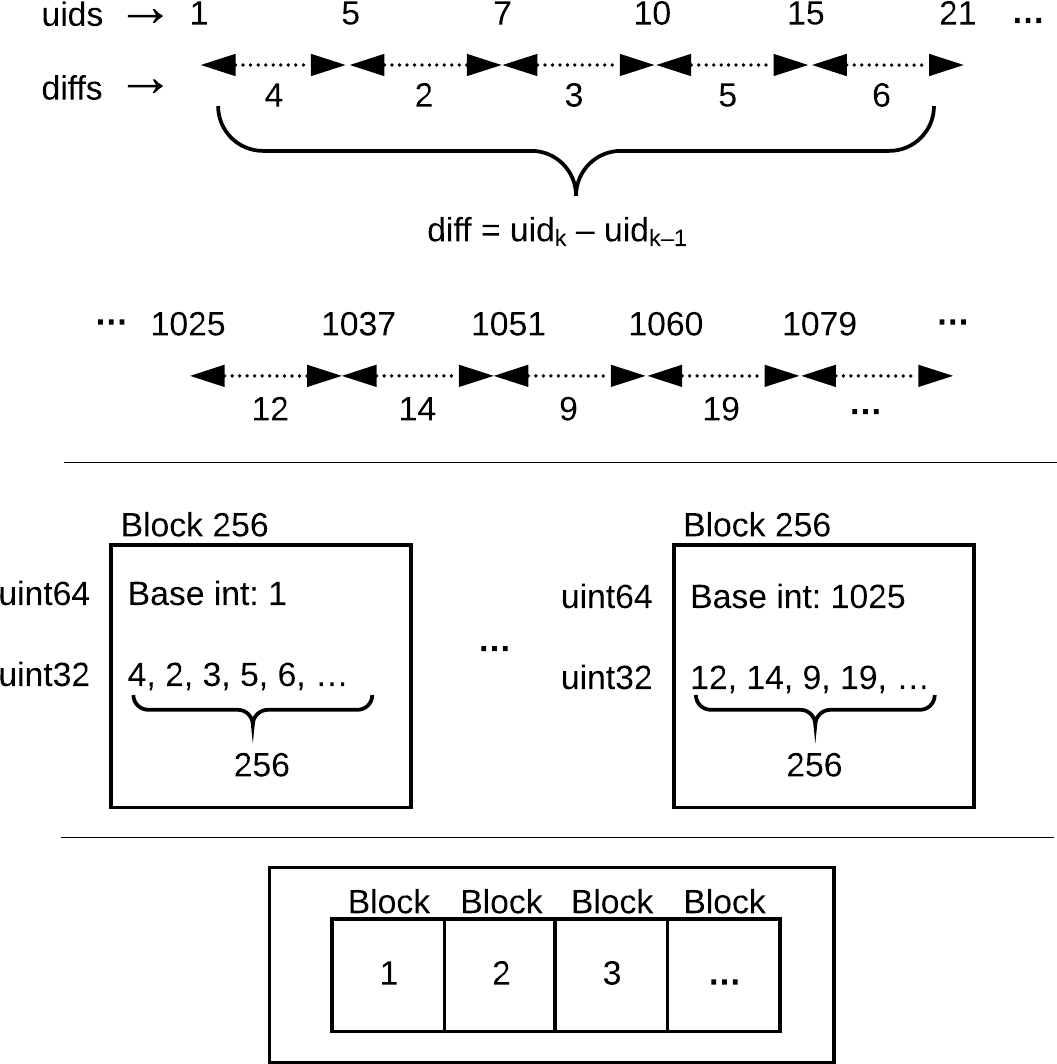


图 2：存储在组 varint 编码块中的发布列表结构

的性能。目前正在使用 Roaring Bitmaps [?] 来表示这些数据。

多亏了这些技术，单边遍历只对应于一个獾查找。例如，查找 X 的所有关注者的列表将涉及对 <follower, X> 键进行查找，这将给出一个包含所有关注者 uid 的发布列表。可以进一步查找以获取关注者发布的帖子列表。 X 和 Y 之间的公共跟随者可以通过进行两次查找，然后将 <follower, X> 和 <follower, Y> 的已排序整数列表相交来找到。请注意，分布式连接和（基于对象的）遍历仅需要通过网络传输 uid，这也非常有效。所有这些都使 Dgraph 在这些操作上非常有效*，而不会影*响典型的 select \* from table 其中 X=Y 样式的记录查找。

这种类型的数据存储在连接和遍历方面有好处，但也带来了高扇出的额外问题。如果具有相同 <subject, predicate> 的记录太多，则整个发布列表可能会增长到无法维持的大小。这通常只是对象的问题（不是值的问题）。我们通过在磁盘大小达到某个阈值后立即对发布列表进行二进制拆分来解决此问题。拆分发布列表将作为多个键存储在 Badger 中，并进行了优化以避免在操作需要它们之前检索拆分。尽管存储存在差异，但发布列表继续通过 API 提供与未拆分列表相同的排序迭代。

# 数据分片

尽管 Dgraph 具有 NoSQL 和分布式 SQL 数据库的许多功能，但它处理记录的方式却大不相同。在其他数据库中，行或文档将是最小的存储单元（保证位于一起），而分片可以像生成由许多这些记录组成的相同大小的块一样简单。

Dgraph 的最小记录单元是一个三元组（主语-谓语-宾语，如下所述），每个谓词整体形成一个分片。换句话说，Dgraph 在逻辑上将所有具有相同谓词的三元组分组，并将它们视为一个分片。然后每个分片被分配一个组 (1..N)，然后可以由服务于该组的所有 Alpha 提供服务，如第 ?? 节所述。

这种数据分片模型允许 Dgraph 在单个网络调用中执行完整的连接，而无需调用者跨服务器获取任何数据。这与磁盘上以独特方式对记录进行分组相结合，将通常由昂贵的磁盘迭代执行的操作转换为更少、更便宜的磁盘查找，这使得 Dgraph 内部工作非常高效。

为了进一步阐述这一点，请考虑一个数据集，其中包含有关人们住在哪里（谓词：“lives-in”）和他们吃什么（谓词：“eats”）的信息。数据可能看起来像这样：

<person-a> <lives-in> <sf> 。

<person-a> <eats> <sushi> .

<person-a><eats><indian>。

...

<person-b> <lives-in> <nyc> 。

<person-b> <eats> <thai> 。

在这种情况下，我们将有两个分片：live-in 和eats。假设最坏的情况*是集群太大以至于每*个分片都*位于单独*的服务器上。对于询问 [住在 SF 并吃寿司的人] 的查询，Dgraph 将对包含 live-in 的服务器执行一次网络调用，并对所有住在 SF 的人进行一次查找 (\* <lives-in> < SF>）。第二步，它将获取这些结果并将它们发送到包含吃的服务器，进行一次查找以获取所有吃寿司的人 (\*

<eats> <sushi>)，并与上一步的结果集相交，生成最终的 SF 吃 Sushi 人名单。以类似的方式，然后可以进一步过滤/加入这个结果集，每个加入在一个网络调用中执行。

正如我们在 ?? 节中了解到的，结果集是一个排序**的 6**4 位无符号整数列表，这使得检索和交集操作非常有效。

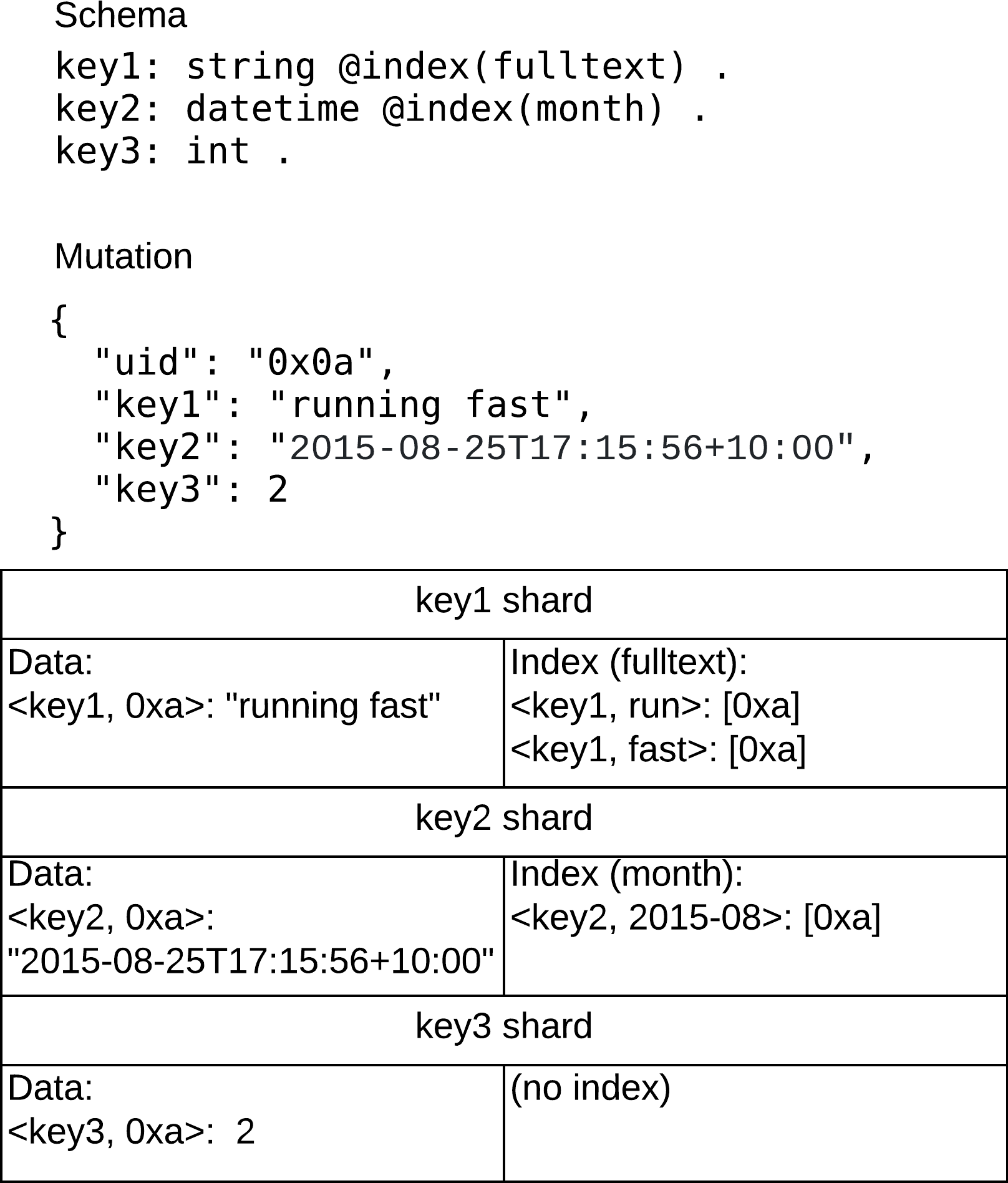


图 3：数据分片

# 数据再平衡

如上所述，每个分片都包含一个完整的谓词，这意味着 Dgraph 分片的大小可能不均匀。分片不仅包含原始数据，还包含它们的所有索引。 Dgraph 组包含许多分片，因此组的大小也可能不均匀。组和分片大小会定期传达给零。零使用此信息尝试使用启发式方法在组之间实现平衡。当前使用的只是数据大小，其想法是相同大小的组将允许在为这些组提供服务的服务器之间使用类似的资源。其他启发式方法，尤其是查询流量方面的启发式方法，可以稍后添加。

为了达到平衡，零会将碎片从一组移动到另一组。它通过将分片标记为只读，然后要求源组同时迭代底层键值并将它们流式传输到目标组的领导者来实现。目的地组长通过 Raft 提出这些键值，获得随之而来的所有正确性。一旦目标组成功应用了所有建议，零会将分片标记为由目标组提供服务。然后零会告诉源组从其存储中删除分片，从而完成该过程。

虽然这个过程听起来很简单，但这里有许多竞争和边缘条件，可能会导致违反事务正确性，如 Jepsen 测试 [?] 所示。我们将在此处展示其中一些违规行为：

* + 1. 稍稍落后于 Alpha 时可能会发生违规

服务器会认为它仍在为分片提供服务（尽管分片已移动到另一个组）并允许在其自身上运行突变。为了避免这种情况，所有事务状态都会保留写入的分片和组信息（以及它们的冲突键，我们将在第 ?? 节中看到）。然后由 Zero 检查分片组信息，以确保交易观察到的（通过它与之交谈的 Alpha）和 Zero 拥有的相同——不匹配会导致交易中止。

* + 1. 在将分片置于只读模式后提交事务时，会发生另一种违规情况——这将导致该提交在分片传输期间被忽略。零通过为移动操作分配时间戳来捕获这一点。任何在更高时间戳的提交（在这个分片上）都将被中止，直到分片移动完成并且分片被带回读写模式。
    2. 当目标组收到低于移动时间戳的读取时，或者源组在删除分片后收到读取时，可能会发生另一种违规。在这两种情况下，都不存在可能导致读取错误返回 nil 值的数据。 Dgraph 通过将移动时间戳通知给目标组来避免这种情况，它可以使用它来拒绝对其下方分片的任何读取。类似地，零包括一个成员标记，在该组删除分片之前，源 Alpha 必须到达该标记，因此，组中的每个 Alpha 成员都会知道它在删除数据之前不再为数据提供服务。

总的来说，事实证明，在事务正确性方面，分片移动期间成员信息同步的机制最难做到正确。

# 索引

Dgraph 旨在成为应用程序的主要数据库。因此，它支持大多数常用索引。特别是对于字符串，它支持正则表达式、全文搜索、术语匹配、精确和哈希匹配索引。对于日期时间，它支持年、月、日和小时级别的索引。对于geo，它支持附近、内部等操作，等等...

所有这些索引都由 Dgraph 使用与上述相同的发布列表格式存储。索引和数据之间的区别是关键。数据键通常是 <predicate, uid>，而索引键是 <predicate, token>。使用索引标记器从数据值派生出标记。

每个索引标记器都支持这个接口：

type Tokenizer interface { Name() string

// 类型返回的字符串表示

// 我们关心的 typeID。类型（）字符串

// 令牌返回给定值的令牌。这

// 令牌不应与字节一起编码

// 标识符。

令牌（接口{}）（[]字符串，错误）

// 标识符为此返回前缀字节

// 令牌类型。这应该是独一无二的。范围

// 0x80 到 0xff（含）为保留

// 用户提供的自定义标记器。标识符（）字节

// 如果分词器可以，则 IsSortable 返回 true

// 用于排序/排序。 IsSortable() 布尔值

// IsLossy() 如果我们不存储

// 值直接作为索引键

// 标记化。如果谓词被标记化

// 使用有损分词器，我们需要获取

// 实际值并进行比较。 IsLossy() 布尔值

}

每个分词器都有一个全局唯一标识符（Identifier() 字节），包括运营商提供的自定义分词器。生成的令牌以令牌化器标识符为前缀，以便能够遍历仅属于该令牌化器的所有令牌。这在对不等式查询（大于、小于等）进行迭代时很有用。请注意，只有在标记器可排序 (IsSortable() bool) 时才能进行不等式查询。例如，在字符串中，精确索引是可排序的，但哈希索引则不是。

根据谓词在模式中设置的索引，谓词中的每个变化都会调用一个或多个这些标记器来生成标记。请注意，索引仅对值进行操作，而不对对象进行操作。将使用突变前值生成一组标记，并使用突变后值生成另一组标记。将添加突变以从前令牌的发布列表中删除主题 uid 并将主题 uid 添加到后令牌。

请注意，所有索引都有对象值，因此它们主要只处理 uid。索引尤其会受到高扇出问题的影响，并且可以使用 ?? 部分中描述的发布列表拆分来解决。

# 多版本并发控制

如第 ?? 节所述，数据以发布列表格式存储，该格**式由**按整数 id 排序的发布组成。使用提交时间戳，所有发布列表写入在提交时存储为 Badger 的增量。请注意，时间戳在整个数据库中全局单调增加，因此任何未来的提交都保证具有更高的时间戳。

由于多种原因，无法就地更新此列表。一是獾（和大多数 LSM 树）写入的是

不可变的，它与文件系统和 rsync 配合得很好。其次是在排序列表中添加条目需要移动后续条目，这取决于条目的位置可能是昂贵的。第三，随着发布列表的增长，我们希望避免每次发生变化时都重写大值（对于索引，它可能会经常发生）。

Dgraph 将发布列表视为一种状态。然后，每个未来的写入都存储为具有更高时间戳的增量。增量通常由带有操作（设置或删除）的过帐组成。为了生成发布列表，Badger 将按降序迭代版本，从读取时间戳开始，选择所有增量直到找到最新状态。要运行发布列表迭代，将选择交易的正确发布，按整数 id 排序，然后在这些增量发布和底层发布列表状态之间运行合并排序操作。

这种机制的早期迭代旨在保持 delta 层也按整数 id 排序，将其覆盖在状态之上以避免在读取期间进行排序——任何添加或删除都将根据已经存在的内容进行合并。 delta 层和状态。事实证明，这些迭代太复杂，无法为团队维护，并且难以找到错误。最终，这个概念被放弃了，取而代之的是一个简单易懂的解决方案，即选择正确的帖子进行阅读并在迭代之前对其进行排序。此外，早期的 API 实现了向前和向后迭代，增加了复杂性。随着时间的推移，很明显只需要向前迭代，从而简化了设计。

避免在每次写入时重新生成发布列表状态有很多好处。同时，随着增量的积累，列表重新生成的工作被委托给读者，这会减慢读取速度。为了找到平衡并避免无限期地获得增量，我们添加了一个汇总机制。

**汇总：当键被读取**时，Dgraph 将有选择地重新生成具有最少增量的发布列表，或者有一段时间没有重新生成。重新生成是通过从最新状态开始，然后按顺序迭代增量并将它们与状态合并来完成的。然后将最终状态写回最新的 delta 时间戳，替换 delta 并形成新状态。然后可以丢弃该键的所有先前增量和状态以回收空间。

该系统允许 Dgraph 提供 MVCC。每次读取都在数据库的不可变版本上运行。更新的 deltas 是在较高的时间戳生成的，并且在读取较低的时间戳时会被跳过。

# 交易

Dgraph 的设计目标是操作简单。因此，目标之一是不依赖任何第三方系统。事实证明，这在为数据和事务提供高可用性的同时很难实现。

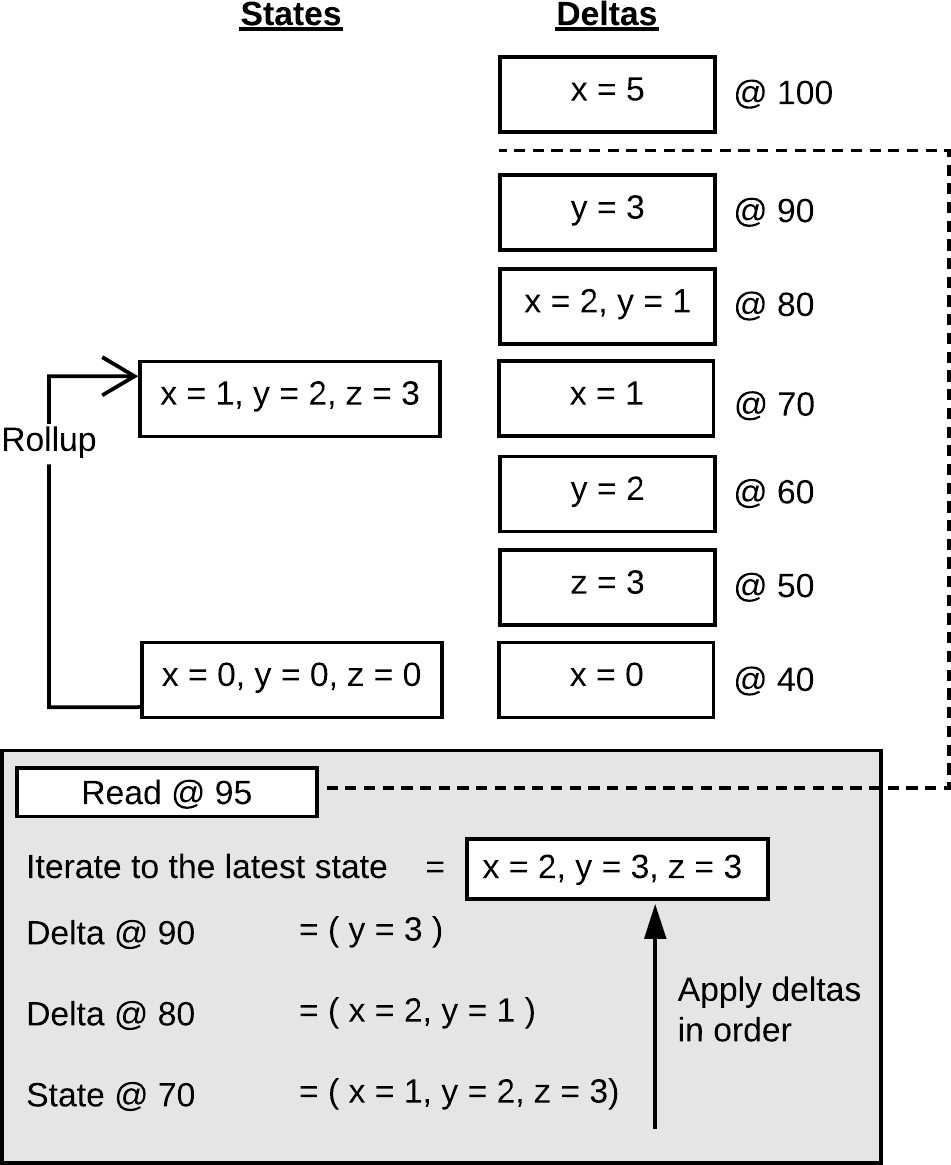


图 4：MVCC

在 Dgraph 中设计事务时，我们查看了 Spanner [?]、HBase [?]、Percolator [?] 和其他人的论文。 Spanner  **最**著名的是使用原子钟为**事**务分配时间戳。这是以在没有基于  **G**PS 的时钟同步机制的商品服务器上降低写入吞吐量为代价的。因此，我们拒绝了这个想法，转而拥有一个零服务器，它可以以更快的速度分发逻辑时间戳。

为了避免零成为单点故障，我们运行多个零实例形成一个 Raft 组。但是，这带来了一个独特的挑战，即在领导者改选的情况下如何进行交接。 Omid, Reloaded [?]（简称Omid2）论文利用外部系统解决了这个问题。在 Omid2 中，他们运行备用时间戳服务器以在领导者失败时接管。这个备用服务器不需要获取最新的事务状态信息，因为 Omid2 使用 Zookeeper [?]，一个用于维护事务日志的集中服务。**同**样，TiDB 构建了 TiKV，它使用基于 Raft 的键值复制模型。这使得 TiDB 的每一次写入都自动被认为是高可用的。类*似地，Bigtab*le [?] 使用 Google 文件系统 [?] 进行分布式存储。因此，不需要在形成法定人数的多个服务器之间发生直接的信息传输。

虽然这个概念在数据库中实现了简单性，但由于两个原因，我们并不完全对这个想法感到兴奋。第一，我们有一个明确的目标，即不依赖任何第三方系统，以便在操作上更容易地运行 Dgraph，并且认为应该可以在不推动的情况下找到解决方案

Badger（存储）内的同步复制。其次，除非必要，我们希望避免接触磁盘。通过让 Raft 成为 Dgraph 过程的一部分，我们可以在将事情写入状态时进行调优，以实现更高的效率。事实上，我们的事务实现在提交之前不会写入磁盘上的 DB 状态（仍然写入 Raft WAL）。

我们仔细查看了 HBase 论文（[?]、[?]）以寻找其他想法，但它们并**不**能直接满**足**我们的需求。例如，HBase 将大量事务信息推送回客户端，为他们提供关于应该或不应该阅读哪些内容以维护事务保证的关键信息。然而，这使得客户端库更难构建和维护，这是我们不喜欢的。最重要的是，一个图表

**算法 1 提交（Ts、**Keys）

1：**对于每**个键kKeys*做*

*∈*

2： **如果**  *lastCommi*t*(*k*) >  T***s 那么**

3: *Propose*(*Ts abort*)

*←*

4： **返回**

5： **结束如果**

6：**结束**

7：*Tc GetTimestamp*s(1) 8**：对于每**个键 k 键执行 9*：lastCom***mi**t(*k) Tc*

*←*

*∈*

*←*

10：**结束**

11:  *提议(Ts ←*  T*c)*

查询可以在中间步骤中触及数百万个键，它是

跟踪所有这些信息并将其传播给客户端的成本很高。

Dgraph 客户端库的目标是保持尽可能少的状态，以允许不熟悉 Dgraph 内部结构的开源用户使用我们不熟悉的语言（例如 Elixir）构建和维护库。

// TODO：我是否描述了第一次迭代？

我们当时根本找不到一篇论文来描述如何构建一个简单易懂、高度可用的事务系统，该系统可以在不假设存储层高度可用的情况下运行。所以，我们不得不想出一个新的解决方案。正如 Jepsen 测试所证明的那样，我们的第二次迭代仍然面临许多问题。因此，我们将第二次迭代简化为第三次迭代，如下所示。

# 无锁高可用性事务处理

Dgraph 遵循无锁事务模型。每个事务并发地进行其进程，从不阻塞其他事务，同时在其开始时间戳或低于其开始时间戳时读取已提交的数据。如前所述，零领导者维护一个 Oracle，它将逻辑事务时间戳分发给 Alpha。 Oracle 还跟踪提交映射，存储冲突键最新提交时间戳。如算法 ?? 所示，每个事务都向 Oracle 提供冲突键列表，以及事务的开始时间戳。冲突密钥源自修改后的密钥，但并不相同。对于每次写入，根据架构计算冲突键。当事务请求提交时，零将检查这些键中是否有任何键的提交时间戳高于事务的开始时间戳。如果满足条件，则中止事务。否则，Oracle 会租用新的时间戳，设置为更新映射中的提交时间戳和冲突键。

*→*

然后，零领导者以开始提交 ts（其中提交 ts = 0 表示中止）的形式向追随者提议此状态更新（提交或中止）并达到法定人数。一旦达到法定人数，零领导者就会将此更新流式传输给订阅者，即 Alpha 领导者。为了保持设计

*→*

**算法 2 水印：计算**  DoneUntil (T , isPending)

1：**如果**  *T*/*MinHeap*  **那么**

*∈*

2： *最小堆*

*←*

3：**结束如果**

4：待定*(T)isPe*n*d*i *ng*

*←*

5：*curDoneTs DoneUntil*

*←*

6：**对于每**个 mi*nTsMi nHeap.P*e*ek()*  做

*∈*

7： **如果***挂起（minT*s*）那么*

8： *休息*

9： **结束如果**

10: *MinHeap*.*Pop*()

11: *curDoneTs minTs*

*←*

12：**结束**

13：Do*neUntil ← curDoneTs*

简单，零不会推送给任何 Alpha 领导者。最新的 Alpha 领导者（无论是谁）的工作是从零建立开放流以接收交易状态更新。

随着交易状态的更新，零领导者还发送了一个 MaxAssigned 时间戳。 MaxAssigned 是使用 Watermark 算法 ?? 计算的，该算法维护所有已分配时间戳的最小堆，包括开始和提交时间戳。当达成共识时，时间戳被标记为完成，MaxAssigned 被推进到最大时间戳，直到一**切都**根据需要达成共识。请注意，开始时间戳通常不需要共识（除非需要更新租约）并立即标记为完成。提交时间戳始终需要达成共识，以确保零组在交易状态上达到法定人数。这允许零追随者成为领导者并完全了解交易状态。这种排序对于实现交易保证至关重要，我们将在下面看到。

一旦 Alpha 领导者收到此更新，他们就会将其提交给他们的追随者，以相同的顺序应用更新。 Alphas 中的所有 Raft 提案申请都是串行完成的。 Alphas 还有一个 Oracle，它跟踪待处理的事务。他们维护开始时间戳，以及一个事务缓存，将所有更新的发布列表保存在

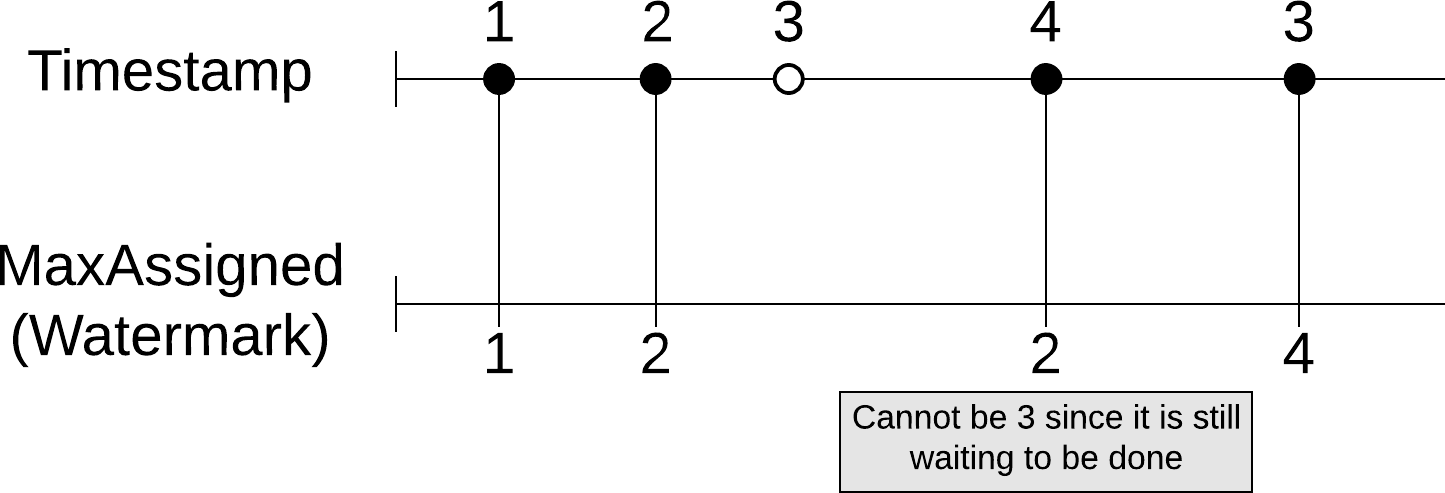


图 5：MaxAssigned 水印。空心圆代表，实心圆代表完成。开始时间戳 1、2 和 4 会立即标记为完成。提交时间戳 3 开始，并且必须在完成之前达成共识。水印跟踪最高时间戳，在该时间戳下一切都完成了。

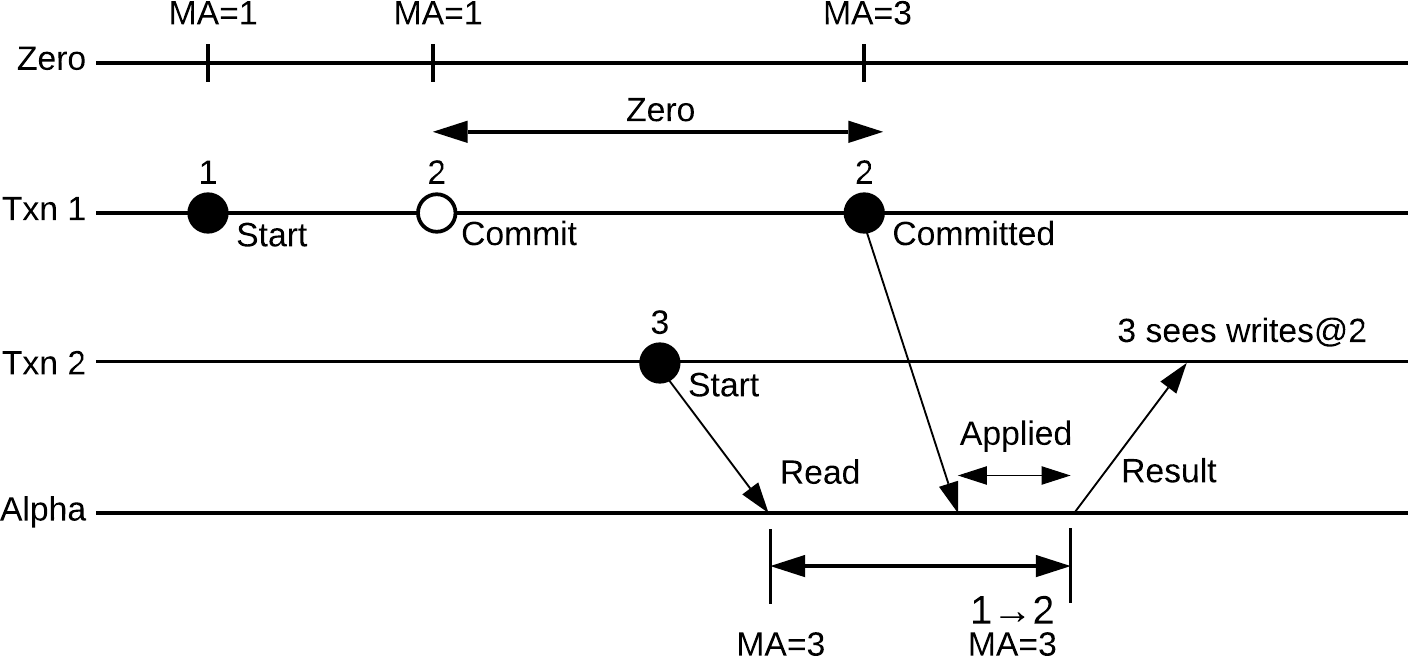


图 6：MaxAssigned 系统确保可线性读取。在时间戳高于当前 MaxAssigned (MA) 的读取必须阻塞以确保写入直到应用读取时间戳。 Txn 2 接收开始 ts 3，并且在 ts 3 处的读取必须确认直到 ts 2 的任何写入。

记忆。在事务中止时，缓存会被简单地删除。在事务提交时，使用提交时间戳将发布列表写入 Badger。最后，更新 MaxAssigned 时间戳。

每个读或写操作都必须有一个开始时间戳。当一个新的查询或突变遇到一个 Alpha 时，它会

事务和线性化读取。

为了正确起见，只允许零领导者分配时间戳、uid 等。在极端情况下，零追随者会错误地认为他们是领导者并提供陈旧数据——Dgraph 做了很多事情来避免这些情况。

* + 1. 如果零领导发生变化，新领导将出租比前任领导所看到的时间戳范围更高的时间戳。然而，一个被老领导卡住的老提交提议可以被转发给新的。这可能允许在较旧的时间戳发生提交，从而导致事务保证失败。我们通过禁止零追随者向领导者转发请求并拒绝这些提议来避免这种情况。

// TODO：我们应该有一个成员资格部分，它解释了成员资格的运作方式并传送到 Alphas。

* + 1. 从零流式传输的每个成员状态更新都需要读取法定人数（与零对等方核对以查找该组看到的最新 Raft 索引更新）。例如，如果零在分区后面，它将无法达到此法定人数并发送成员资格更新。 Alpha 期望定期更新，如果在几个周期后他们没有收到零领导者的消息，他们会认为零领导者已不复存在，取消连接并重试与（可能不同的）健康领导者建立连接。

# 一致性模型

Dgraph 支持 MVCC、读取快照和分布式 ACID 事务。事务跨通用数据集在集群范围内进行——不受任何密钥级别或服务器级别限制的限制。交易也是无锁的。他们不会阻止/等待未提交的事务看到待处理的写入。它们都可以同时进行，零会根据冲突选择提交或中止它们。

考虑到跟踪单个图形查询读取的所有数据（可能是数百万个键）的开销，Dgraph 不提供 Serializable Snapshot Isolation。相反，Dgraph 提供快照隔离，跟踪写入比读取包含更多的集合。

Dgraph 为交易（由 Tx 表示）分发单调递增的时间戳（由 T 表示）。因此，如果任何事务 Txi 在 Txj 开始之前提交，则

*开始*

要求零分配时间戳。这个操作通常是

*锡*

*犯罪*

*T*

*< TTxj*  。在 Tcommit 的*任何提交都保证*是

每个 Alpha 只允许一个挂起的分配调用给零领导。如果新收到的查询的开始时间戳高于该 Alpha 注册的 MaxAssigned，它将阻塞查询，直到其 MaxAssigned 达到或超过开始 ts。该解决方案很好地解决了各种边缘情况，包括 Alpha 回退或落后于其对等方的网络分区，或者只是在崩溃后重新启动等。在所有这些情况下，查询将被阻塞，直到 Alpha已经看到了直到查询时间戳的所有更新，从而保持了

由任何客户端在时间戳 Tread 处读取，如果  *Tread*  >

*提交。因此，D*graph 读取是可线性化的。此外，所有读取都是整个集群的快照，可以完整查看之前提交的所有事务。

如前所述，Dgraph 读取是可线性化的。虽然这对正确性很有好处，但当大量读取和写入同时进行时，它可能会导致性能问题。所有读取都应该阻塞，直到 Alpha 看到所有写入，直到读取时间戳。在许多情况下，运营商会选择性能而不是实现线性化。图形

提供两种加速读取的选项：

1. 典型的读写事务会为客户端分配一个新的时间戳。这将更新 MaxAssigned，然后它会通过零领导者流向 Alpha 领导者，然后被提议。在此之前，读取无法继续。只读事务仍然需要从零开始的读取时间戳，但零会随机将相同的读取时间戳分发给多个调用者，从而允许 Alpha 分摊在多个查询中达到 MaxAssigned 的成本。
2. 尽力而为交易是只读交易的一种变体，它将使用 Alpha 观察到的 MaxAssigned 时间戳作为读取时间戳。因此，接收方Alpha根本不必阻塞并且可以继续处理查询。这相当于其他数据库中典型的最终一致性模型。最终，每次 Dgraph 读取都是整个分布式数据库的快照，并且没有任何读取会违反快照保证。 1

# 复制

Dgraph 的大多数更新都是通过 Raft 完成的。让我们从 Alphas 开始，它可以通过系统推送大量数据。所有的变更和交易更新都是通过 Raft 提出的，并成为 Raft 预写日志的一部分。在崩溃和重启时，Raft 日志会从最后一个快照开始重放，以使状态机恢复到正确的最新状态。另一方面，日志越长，Alpha 在重新启动时重放它们所需的时间就越长，从而导致启动延迟。因此，必须通过拍摄快照来修剪日志，该快照表明该点之前的状态已被持久化，并且不需要在重新启动时重播。

如上所述，Alphas 将变更写入 Raft WAL，但将它们保存在内存中的事务缓存中。提交事务时，更改将写入提交时间戳的状态。这意味着在重新启动时，必须通过 Raft WAL 将所有待处理的事务带回内存。这需要一个计算来选择正确的 Raft 索引来修剪日志，这会将所有待处理的事务完整地保存在日志中。

我们在修复 Jepsen 问题时学到的教训之一是，为了提高复杂分布式系统的可调试性，系统应该像时钟一样运行。换句话说，一旦一个系统中的事件发生了，其他系统中的事件应该几乎是可以预测的。这一指导原则决定了我们如何拍摄快照。

Raft paper 允许领导者和追随者彼此独立地拍摄快照。 Dgraph 曾经这样做过，但这给系统带来了不可预测性并进行了调试

1但是请注意，典型的 Dgraph 查询可能会命中不同组中的多个 Alpha——其中一些 Alpha 可能尚未达到读取时间戳（初始 Alpha 的 MaxAssigned 时间戳）。在这些情况下，查询仍然会阻塞，直到那些 Alpha 赶上。

难多了。因此，根据可预测性原则的惨痛教训，我们对其进行了更改，使领导者计算快照索引并提出此结果。这允许领导者和追随者都在同一索引处拍摄快照，完全相同的时间（如果他们通常被赶上）。此外，然后将此组级别快照事件传达给零，以允许它通过删除快照时间戳以下的所有条目来修剪冲突图。跟踪日志中的这一事件链极大地提高了系统的可调试性。

Dgraph 只在 Raft 快照中保存元数据，实际数据单独存储。 Dgraph 在快照期间不会复制该数据。当跟随者落后并需要快照时，它会向领导者索要快照，领导者将从其状态流式传输快照（Badger，就像 Dgraph 一样，支持 MVCC，并且在特定时间戳执行读取时，正在对数据库的逻辑快照）。在以前的版本中，follower 在接受来自 leader 的更新之前会清除其当前状态。在较新的版本中，leader 可以选择只向 follower 发送 delta 状态更新，这可以大大减少传输的数据。

# 高可用性和可扩展性

Dgraph 的架构围绕 Raft 组进行更新日志序列化和复制。在 CAP throrem 中，这遵循 CP，即在网络分区中，Dgraph 将选择一致性而不是可用性。但是，CAP 定理的概念不应与高可用性混淆，高可用性取决于在不影响服务的情况下可以丢失多少实例。

在三节点组中，Dgraph 可以为每个组松散一个实例，而不会对数据库的功能造成任何可衡量的影响。但是，考虑到所有更新都通过 Raft，从同一组中丢失两个实例会导致 Dgraph 阻塞。在一个五节点组中，可以在不影响功能的情况下丢失的实例数量是两个。我们不建议每个组运行超过五个副本。

鉴于 Dgraph Zero 的核心管理角色，人们可能会认为 Zero 将是单点故障。然而，事实并非如此。在零追随者死亡的情况下，没有什么真正改变。如果零领导者死亡，零追随者之一将成为领导者，更新其时间戳和 uid 分配租约，获取交易状态日志（通过 Raft 存储）并开始接受来自 Alphas 的请求。在此转换期间，唯一可能丢失的是尝试使用丢失的零提交的事务。他们可能会出错，但可以重试。阿尔法也一样。所有 Alpha 追随者都拥有与 Alpha 领导者相同的信息，并且该组的任何成员都可以在不丢失任何状态的情况下丢失。

Dgraph 可以支持尽可能多的组

由 32 位整数（即使这是一个人为的限制）。每个组可以有一个、三个、五个（可能更多，但不推荐）副本。系统中可以存在的 uid（图形节点）的数量受 64 位无符号整数的限制，交易时间戳也是如此。所有这些都是非常慷慨的限制，而不是可扩展性的问题。

# 查询

一个典型的 Dgraph 查询可以命中许多 Alpha，具体取决于谓词所在的位置。每个查询又细分为多个任务，每个任务负责一个谓词。

# 遍历

Dgraph 查询任务（以下简称任务）一般围绕遍历时将 uid 列表转换为矩阵的机制构建。查询可以有一个要遍历的 uid 列表，执行引擎将在 Badger 中并发查找以获取每个 Uid 的发布列表（注意谓词始终是任务的一部分），将每个 uid 转换为列表。因此，任务查询将返回一个 Uid 列表列表，也就是 UidMatrix。如果谓词包含一个值（例如，谓词名称），则 UidList 返回一个值列表，也就是 ValueMatrix。一个谓词可以只允许一个 uid/value，或者允许多个 uids/value。这种机制在这两种情况下都能正常工作。如果发布列表只有一个 uid/value，则结果列表将只有一个元素。在这种情况下，矩阵将有一个列表列表，每个列表有零个或一个元素。请注意，列表中 Uid 的索引与 UidMatrix 中列表的索引之间存在奇偶校验。因此，Dgraph 可以准确地维护这些关系。

ValueMatrix 通常是任务树中的叶子。一旦我们有了值，我们只需要在结果中对它们进行编码。但是，具有 UidMatrix 结果的任务通常会有子任务。这些子任务需要一个查询 UidList 进行处理。 Dgraph 会将 UidMatrix 合并排序为一个已排序的 Uid 列表，该列表将被复制到子任务中。每个子任务可以类似地在相同或其他谓词上运行扩展。

# 职能

Dgraph 也支持函数。当需要将全局 uid 空间限制为一个小的集合（甚至单个 uid）时，这些函数提供了一种查询 Dgraph 的简单方法。函数还提供高级功能，如正则表达式、全文搜索、可排序数据类型的相等和不等式、地理空间搜索等。这些函数也被编码到任务查询中，只是这次它们不以一个 UidList。任务查询反而包含从对应于这些函数使用的索引的分词器派生的标记（如上所述）。大多数函数需要某种索引来操作，例如，正则表达式查询

使用三元组索引，地理空间查询使用基于 S2 单元的地理索引等等……如上一节所述，索引键编码谓词和标记，而不是谓词和 uid。因此，填充矩阵的机制与任何其他任务查询中的机制相同。只是这一次，我们使用令牌列表而不是 Uid 列表作为查询集。

# 过滤器

上述技术适用于遍历。但是，过滤器（交叉点）是用户查询的重要组成部分。每个任务都包含一个 UidList 作为查询和一个矩阵作为结果。 Task 还存储了一个生成的 uid 列表，它可以存储来自生成的 UidMatrix 的 uid 集。根据是否应用过滤器，此 uid 集可以与合并排序的 UidMatrix 或其子集相同。

过滤器本身就是一棵树。 Dgraph 支持 AND、OR 和 NOT 过滤器，它们可以进一步组合以创建复杂的过滤器树。过滤器通常由可以请求更多信息并表示为任务的函数组成。这些任务以与上述相同的机制执行，但做一件额外的事情。这些任务还包含 Uid 的源列表（过滤器应用到的父任务的结果集）。此 uid 列表作为过滤器任务的一部分发送。任务使用这些 uid 在目标服务器上执行任何交集，只返回结果的一个子集，而不是检索任务的所有结果。这可以显着减少结果负载大小，同时还允许在过滤器任务执行期间进行优化以加快速度。一旦返回结果，协调器服务器将使用 AND、OR 或 NOT 运算符拼接结果。

# 交叉路口

uid 交集本身使用整数交集的三种模式，根据结果大小与源 UidList 大小的比例在线性扫描、块跳转或二进制搜索之间进行选择，以提供最佳性能。当两个列表大小相同时，Dgraph 对两个列表使用线性扫描。当一个列表比另一个长得多时，Dgraph 将迭代较短的列表并对较长的列表进行二元查找。对于介于两者之间的某个范围，Dgraph 将在较短的列表上进行迭代，并在较长的列表上进行前向搜索块跳转。 Dgraph 基于块的整数编码机制使这一切变得非常高效。

TODO：谈谈酸。

# 未来的工作

由于严重的读写争用，我们从 Dgraph 中删除了数据缓存，并构建了一个新的、无争用的 Go 缓存库来帮助我们的读取。将其与 Dgraph 集成的工作正在进行中。 Dgraph 没有任何查询或响应

缓存——这样的缓存在 MVCC 环境中很难维护，在这种环境中，每次读取都会根据其时间戳产生不同的结果。

有序整数编码和交集是一个热门研究话题，在性能方面有很大的优化空间。如前所述，正在试验切换到 Roaring Bitmaps 的工作正在进行中。

我们还计划研究查询优化器，它可以更好地确定执行查询的正确顺序。到目前为止，GraphQL 的简单特性已经让操作员手动优化他们的查询——但肯定 Dgraph 可以在了解数据状态时做得更好。

未来的工作是允许在分片移动期间写入，这取决于分片的大小可能需要一些时间。

TODO：添加一个结论。

# 致谢

如果没有核心开发团队和扩展社区的不懈贡献，Dgraph 就不可能实现。如果没有我们投资者的资助，这项工作也是不可能完成的。完整的贡献者列表在这里：

github.com/dgraph-io/dgraph/graphs/contributors

Dgraph 是一个开源软件，可在

https://github.com/dgraph-io/dgraph

有关 Dgraph 的更多信息，请访问

https://dgraph.io