内容由 福昕翻译 生成（Content is generated by foxit）

时间（time）: 2021-12-25

Dgraph: Synchronously Replicated, Transactional and Distributed Graph Database

Dgraph：同步复制、事务和分布式图数据库

Abstract

抽象的

Manish Jain manish@dgraph.io Dgraph Labs, Inc.

Manish Jain manish@dgraph.io Dgraph Labs, Inc.

Version: 0.8 Last Updated: March 1, 2021

版本：0.8 最后更新：2021 年 3 月 1 日

Dgraph solves the join depth problem with a unique shard-

Dgraph 用独特的分片解决了连接深度问题

Dgraph is a distributed graph database which provides hori- zontal scalability, distributed cluster-wide ACID transactions, low-latency arbitrary-depth joins, synchronous replication, high availability and crash resilience. Aimed at real-time trans- actional workloads, Dgraph shards and stores data in a way to optimize joins and traversals, while still providing data retrieval and aggregation. Dgraph’s unique take is to provide low-latency arbitrary-depth joins in a constant number of net- work calls (typically, just one network call) that would be required to execute a single join, irrespective of the size of the cluster or the size of the result set.

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。

Introduction

介绍

Distributed systems or databases tend to suffer from join depth problem. That is, as the number of traversals of relationships increase within a query, the number of network calls required (in a sufficiently sharded dataset) increase. This is typically due to entity-based data sharding, where entities are randomly (sometimes with a heuristic) distributed across servers con- taining all the relationships and attributes along with them. This approach suffers from high-fanout result set in interme- diate steps of a graph query causing them to do a broadcast across the cluster to perform joins on the entities. Thus, a sin- gle graph query results in network broadcasts, hence causing a jump in the query latency as the cluster grows.

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 is a distributed database with a native graph back- end. It is the only native graph database to be horizontally scalable and support full ACID-compliant cluster-wide dis- tributed transactions. In fact, Dgraph is the first graph database to have been Jepsen [?] tested for transactional consistency.

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。

Dgraph automatically shards data into machines, as the amount of data or the number of servers change, and auto- matically reshards data to move it across servers to balance the load. It also supports synchronous replication backed by Raft [?] protocol, which allows the queries to seamlessly failover to provide high availability.

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。

ing mechanism. Instead of sharding by entities, as most sys- tems do, Dgraph shards by relationships. Dgraph’s unique way of sharding data is inspired by research at Google [?], which shows that the overall latency of a query is greater than the latency of the slowest component. The more servers a query touches to execute, the slower the query latency would be. By doing relationship based sharding, Dgraph can execute a join or traversal in a single network call (with a backup network call to replica if the first is slow), irrespective of the size of the cluster or the input set of entities. Dgraph executes arbitrary-depth joins without network broadcasts or collecting data in a central place. This allows the queries to be fast and latencies to be low and predictable.

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。

Dgraph Architecture

图形架构

Dgraph consists of Zeros and Alphas, each representing a group that they are serving. Zeros serve group zero and Alphas serve group one, group two and onwards. Each group forms a Raft cluster of 1, 3 or 5 members configurable by a human operator (henceforth, referred to as the operator). All updates made to the group are serialized via Raft consensus algorithm and applied in that order to the leader and followers.

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 store and propagate metadata about the cluster while Alphas store user data. In particular, Zeros are responsible for membership information, which keeps track of the group each Alpha server is serving, its internal IP address for communi- cation within the cluster, the shards it is serving, etc. Zeros do not keep track of the health of the Alphas and take actions on them – that is considered the job of the operator. Using this information, Zero can tell the new Alpha to either join and serve an existing group, or form a new group.

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。

The membership information is streamed out from Zero to all the Alphas. Alphas can use this membership information to route queries (or mutations) which hit the cluster. Every instance in the cluster forms a connection with every other instance (thus forming 2 (N) open connections, where N

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

2

×

×

= number of Dgraph instances in the cluster), however, the

= 集群中 Dgraph 实例的数量），但是，

protocol buffer [?] data format and not interchanged among the two.

protocol buffer [?] 数据格式和两者之间不能互换。

{

{

"uid" : "0xab",

"uid": "0xab",

"type" : "Astronaut", "name" : "Mark Watney", "birth" : "2005/01/02",

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

"follower": { "uid": "0xbc", ... },

“跟随者”：{“uid”：“0xbc”，…}，

}

}

<0xab> <type> "Astronaut" .

<0xab> <type> "宇航员" 。

<0xab> <name> "Mark Watney" .

<0xab> <名称> “马克·沃特尼”。

<0xab> <birth> "2005/01/02" .

<0xab> <出生> "2005/01/02" 。

<0xab> <follower> <0xbc> .

<0xab> <追随者> <0xbc> 。

Figure 1: Dgraph Architecture: There is one Zero group and multiple Alpha groups. Each group is a Raft group consisting of one or more members.

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。

usage of this connection depends on their relationship. For example, a Raft leader-follower relationship would have heart- beats (every 100 ms) and data flowing, while an Alpha would only talk to Alpha in another group when it needs to do so for processing queries or mutations. Every open connection does have light-weight health checks to avoid stalling on a tar- get server which has become unresponsive (died, partitioned, etc.). Both Alphas and Zeros expose one port for intra-cluster communication over Grpc [?] and one for external commu- nication with clients over HTTP. Alphas additionally expose an external Grpc port for communication with Grpc based clients – all official clients run over Grpc.

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 also runs an oracle which hands out monotonically- increasing logical timestamps for transactions in the cluster (no relation to system time). A Zero leader would typically lease out a bandwidth of timestamps upfront via Raft proposal and then service timestamp requests strictly from memory without any further coordination. Zero oracle tracks additional things for aiding with transaction commits, which would be elaborated in section ??.

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。

Zero gets information about the size of data in each group from the Alpha leaders, which it uses to make decisions about shard movement, which would be elaborated in section ??.

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。

Data Format

数据格式

Dgraph can input data in a JSON format or (slightly modified) RDF NQuad format. Dgraph would break down a JSON map into smaller chunks, with each JSON key-value forming one record equivalent of a single RDF triple record. When parsing RDF Triple or JSON, data is directly converted into an internal

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

A triple is typically expressed as a subject-predicate-object or a subject-predicate-value. Subject is a node, predicate is a relationship, and object can be another node or a primitive data type. One points from a node to another node, the other points from a node to a value. In the above example, the triple with name is a type of subject-predicate-value (typically referred to as an attribute), while the triple with follower is a type of subject-predicate-object. Dgraph makes no difference in how it handles these two types of records (to avoid confusion over these two types, we’ll refer to them as object-values). Dgraph considers this as the unit of record and a typical JSON map would be broken into multiple such records.

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。

Data can be retrieved from Dgraph using GraphQL [?] and a modified version of GraphQL, called GraphQL+- [?]. GraphQL+- has most of the same properties as GraphQL. But, adds various properties which are important for a database, like query variables, functions and blocks. More information about how the query language came to be and the differences between GraphQL and GraphQL+- can be found in this blog post [?].

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。

As mentioned in section ??, all internal and external com- munication in Dgraph runs via Grpc and Protocol Buffers. Dgraph also exposes HTTP endpoints to allow building client libraries in languages which are not supported by these two. There is a functionality parity between HTTP endpoints and APIs exposed via Grpc.

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。

In accordance with the GraphQL spec, query responses from Dgraph are in JSON format, both over HTTP and Grpc.

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

Data Storage

数据存储

Dgraph data is stored in an embeddable key-value database called Badger [?] for data input-output on disk. Badger is an LSM-tree based design, but differs from others in how it can optionally store values separately from keys to generate a much smaller LSM tree, which results in both lower write and read amplification. Various benchmarks run by the team

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

show Badger to provide equivalent or faster writes than other LSM based DBs, while providing equivalent read latencies compared to B+-tree based DBs (which tend to provide much faster reads than LSM trees).

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。

As mentioned above, all records with the same predicate form one shard. Within a shard, records sharing the same subject-predicate are grouped and condensed into one single key-value pair in Badger. This value is referred to as a posting list, a terminology commonly used in search engines to refer to a sorted list of doc ids containing a search term. A posting list is stored as a value in Badger, with the key being derived from subject and predicate.

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> <follower> <0xab> .

<0x01> <追随者> <0xab> 。

<0x01> <follower> <0xbc> .

<0x01> <追随者> <0xbc> 。

<0x01> <follower> <0xcd> .

<0x01> <追随者> <0xcd> 。

...

...

key = <follower, 0x01>

键 = <追随者，0x01>

value = <0xab, 0xbc, 0xcd, ...>

值 = <0xab, 0xbc, 0xcd, ...>

All subjects in Dgraph are assigned a globally unique id, called a uid . A uid is stored as a 64-bit unsigned integer (uint64) to allow efficient, native treatment by Go language in the code base. Zero is responsible for handing out uids as needed by the Alphas and does it in the same monotonically increasing fashion as timestamps (section ??). A uid once allocated is never reallocated or reassigned. Thus, every node in the graph can be referenced by a unique integer.

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。

Object-values are stored in postings. Each posting has an integer id. When the posting holds an object, the id is the uid assigned to that object. When posting holds a value, the integer id for value is determined based upon the schema of the predicate. If the predicate allows multiple values, the integer id for the value would be a fingerprint of the value. If the predicate stores values with language, the integer id would be a fingerprint of the language tag. Otherwise, the integer id would be set to maximum possible uint64 (264 - 1). Both uid and integer id is never set to zero.

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。

Value could be one of the many supported data types: int, float, string, datetime, geo, etc. The data is converted into bi- nary format and stored in a posting along with the information about the original type. A posting can also hold facets. Facets are key-value labels on an edge, treated like attachments.

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。

In a common case where the predicate only has objects (and no values like follower edge), a posting list would consist largely of sorted uids . These are optimized by doing integer compression. The uids are grouped in blocks of 256 integers (configurable), where each block has a base uid and a binary blob. The blob is generated by taking a difference of current uid with the last and storing the difference in bytes encoded using group varint. This generates a data compression ratio of 10. When doing intersections, we can use these blocks to do binary searches or block jumps to avoid decoding all the blocks. Sorted integer encoding is a hotly researched topic and there is a lot of room for optimization here in terms

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

Figure 2: Posting list structure stored in group varint-encoded blocks

图 2：存储在组 varint 编码块中的发布列表结构

of performance. Work is going on currently to use Roaring Bitmaps [?] instead to represent this data.

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

Thanks to these techniques, a single edge traversal corre- sponds to only a single Badger lookup. For example, finding a list of all of X’s followers would involve doing a lookup on <follower, X> key which would give a posting list con- taining all of their followers’ uids . Further lookups can be made to get a list of posts made by followers . Common fol- lowers between X and Y an be found by doing two lookups followed by intersecting the sorted int lists of <follower, X> and <follower, Y>. Note that distributed joins and (ob- ject based) traversals only require uids to be transmitted over network, which is also very efficient. All this allows Dgraph to be very efficient on these operations, without compromis- ing on the typical select \* from table where X=Y style record lookups.

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。

This type of data storage has benefits in joins and traversals, but comes with an additional problem of high fan-out. If there are too many records with the same <subject, predicate>, the overall posting list could grow to an untenable size. This is typically only a problem for objects (not so much for val- ues). We solve this by binary splitting a posting list as soon as its on-disk size hits a certain threshold. A split posting list would be stored as multiple keys in Badger, with optimiza- tions made to avoid retrieving the splits until the operation needs them. Despite storage differences, the posting list con- tinues to provide the same sorted iteration via APIs as an unsplit list.

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。

Data Sharding

数据分片

While Dgraph shares a lot of features of NoSQL and dis- tributed SQL databases, it is quite different in how it handles its records. In other databases, a row or document would be the smallest unit of storage (guaranteed to be located together), while sharding could be as simple as generating equal sized chunks consisting of many of these records.

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’s smallest unit of record is a triple (subject- predicate-object, described below), with each predicate in its entirety forming a shard. In other words, Dgraph logically groups all the triples with the same predicate and considers them one shard. Each shard is then assigned a group (1..N) which can then be served by all the Alphas serving that group, as explained in section ??.

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。

This data sharding model allows Dgraph to execute a com- plete join in a single network call and without any data fetch- ing across servers by the caller. This combined with grouping of records in a unique way on disk to convert operations which would typically be executed by expensive disk iterations, into fewer, cheaper disk seeks makes Dgraph internal working quite efficient.

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。

To elaborate this further, consider a dataset which contains information about where people live (predicate: "lives-in") and what they eat (predicate: "eats"). Data might look some- thing like this:

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> <lives-in> <sf> 。

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

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

<person-a> <eats> <indian> .

<person-a><eats><indian>。

...

...

<person-b> <lives-in> <nyc> .

<person-b> <lives-in> <nyc> 。

<person-b> <eats> <thai> .

<person-b> <eats> <thai> 。

In this case, we’ll have two shards: lives-in and eats. As- suming the worst case scenario where the cluster is so big that each shard lives on a separate server. For a query which asks for [people who live in SF and eat Sushi], Dgraph would execute one network call to server containing lives- in and do a single lookup for all the people who live in SF (\* <lives-in> <sf>). In the second step, it would take those results and send them over to server containing eats, do a single lookup to get all the people who eat Sushi (\*

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>), and intersect with the previous step’s re- sultset to generate the final list of people from SF who eat Sushi. In a similar fashion, this result set can then be further filtered/joined, each join executing in one network call.

<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。

As we learnt in section ??, the result set is a list of sorted 64-bit unsigned integers, which make the retrieval and inter- section operations very efficient.

正如我们在 ?? 节中了解到的，结果集是一个排序的 64 位无符号整数列表，这使得检索和交集操作非常有效。

Figure 3: Data sharding

图 3：数据分片

Data Rebalancing

数据再平衡

As explained above, each shard contains a whole predicate in its entirety which means Dgraph shards can be of uneven size. The shards not only contain the original data, but also all of their indices. Dgraph groups contain many shards, so the groups can also be of uneven size. The group and shard sizes are periodically communicated to Zero. Zero uses this information to try to achieve a balance among groups, using heuristics. Current one being used is just data size, with the idea that equal sized groups would allow similar resource usage across servers serving those groups. Other heuristics, particularly around query traffic, could be added later.

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。

To achieve balance, Zero would move shards from one group to another. It does so by marking the shard read-only, then asking the source group to iterate over the underlying key- values concurrently and streaming them over to the leader of the destination group. The destination group leader proposes these key-values via Raft, gaining all the correctness that comes with it. Once all the proposals have been successfully applied by the destination group, Zero would mark the shard as being served by the destination group. Zero would then tell source group to delete the shard from its storage, thus finalizing the process.

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。

While this process sounds pretty straighforward, there are many race and edge conditions here which can cause transac- tional correctness to be violated as shown by Jepsen tests [?]. We’ll showcase some of these violations here:

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：

A violation can occur when a slightly behind Alpha

稍稍落后于 Alpha 时可能会发生违规

server would think that it is still serving the shard (despite the shard having moved to another group) and allow mutations to be run on itself. To avoid this, all transactions states keep the shard and the group info for the writes (along with their conflict keys as we’ll see in section ??). The shard-group information is then checked by Zero to ensure that what the transaction observes (via Alpha it talked to) and what Zero has is the same – a mismatch would cause a transaction abort.

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。

Another violation happens when a transaction commits after the shard was put into read-only mode – this would cause that commit to be ignored during the shard transfer. Zero catches this by assigning a timestamp to the move operation. Any commits (on this shard) at a higher timestamp would be aborted, until the shard move has completed and the shard is brought back to the read-write mode.

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。

Yet another violation can occur when the destination group receives a read below the move timestamp, or a source group receives a read after it has deleted the shard. In both cases, no data exists which can cause the reads to incorrectly return back nil values. Dgraph avoids this by informing the destination group of the move timestamp, which it can use to reject any reads for that shard below it. Similarly, Zero includes a membership mark at which the source Alpha must reach before the group can delete the shard, thus, every Alpha member of the group would know that it is no longer servig the data before deleting it.

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。

Overall, the mechanism of membership information syn- chronization during a shard move proved the hardest to get right with respect to transactional correctness.

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。

Indexing

索引

Dgraph is designed to be a primary database for applications. As such, it supports most of the commonly needed indices. In particular, for strings, it supports regular expressions, full-text search, term matching, exact and hash matching index. For datetime, it supports year, month, day and hour level indices. For geo, it supports nearby, within, etc. operations, and so on...

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...

All these indices are stored by Dgraph using the same post- ing list format described above. The difference between an index and data is the key. A data key is typically <predicate, uid>, while an index key is <predicate, token>. A token is derived from the value of the data, using an index tokenizer.

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。

Each index tokenizer supports this interface:

每个索引标记器都支持这个接口：

type Tokenizer interface { Name() string

type Tokenizer interface { Name() string

// Type returns the string representation of

// 类型返回的字符串表示

// the typeID that we care about. Type() string

// 我们关心的 typeID。类型（）字符串

// Tokens return tokens for a given value. The

// 令牌返回给定值的令牌。这

// tokens shouldn’t be encoded with the byte

// 令牌不应与字节一起编码

// identifier.

// 标识符。

Tokens(interface{}) ([]string, error)

令牌（接口{}）（[]字符串，错误）

// Identifier returns the prefix byte for this

// 标识符为此返回前缀字节

// token type. This should be unique. The range

// 令牌类型。这应该是独一无二的。范围

// 0x80 to 0xff (inclusive) is reserved for

// 0x80 到 0xff（含）为保留

// user-provided custom tokenizers. Identifier() byte

// 用户提供的自定义标记器。标识符（）字节

// IsSortable returns true if the tokenizer can

// 如果分词器可以，则 IsSortable 返回 true

// be used for sorting/ordering. IsSortable() bool

// 用于排序/排序。 IsSortable() 布尔值

// IsLossy() returns true if we don’t store the

// IsLossy() 如果我们不存储

// values directly as index keys during

// 值直接作为索引键

// tokenization. If a predicate is tokenized

// 标记化。如果谓词被标记化

// using a lossy tokenizer, we need to fetch

// 使用有损分词器，我们需要获取

// the actual value and compare. IsLossy() bool

// 实际值并进行比较。 IsLossy() 布尔值

}

}

Every tokenizer has a globally unique identifier (Identifier() byte), including custom tokenizers pro- vided by operators. The tokens generated are prefixed with a tokenizer identifier to be able to traverse through all tokens belonging to only that tokenizer. This is useful when doing iteration for inequality queries (greater than, less than, etc.). Note that inequality queries can only be done if a tokenizer is sortable (IsSortable() bool). For example, in strings, an exact index is sortable, but a hash index is not.

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。

Depending upon which index a predicate has set in the schema, every mutation in that predicate would invoke one or more of these tokenizers to generate the tokens. Note that indices only operate on values, not objects. A set of tokens would be generated with the before mutation value and an- other set with the after mutation value. Mutations would be added to delete the subject uid from the posting lists of before tokens and to add the subject uid to the after tokens.

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。

Note that all indices have object values, so they largely deal only in uids. Indices in particular can suffer from high fan-out problem and are solved using posting list splits described in the section ??.

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。

Multiple Version Concurrency Control

多版本并发控制

As described in section ??, data is stored in posting list format, which consists of postings sorted by integer ids. All posting list writes are stored as deltas to Badger on commit, using the commit timestamp. Note that timestamps are monotonically increasing globally across the DB, so any future commits are guaranteed to have a higher timestamp.

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。

It is not possible to update this list in-place, for multiple reasons. One is that Badger (and most LSM trees) writes are

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

immutable, which plays very well with filesystems and rsync. Second is that adding an entry within a sorted list requires moving following entries, which depending upon the position of the entry can be expensive. Third, as the posting list grows, we want to avoid rewriting a large value every time a mutation happens (for indices, it can happen quite frequently).

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 considers a posting list as a state. Every future write is then stored as a delta with a higher timestamp. A delta would typically consist of postings with an operation (set or delete). To generate a posting list, Badger would iterate the versions in descending order, starting from the read timestamp, picking all deltas until it finds the latest state. To run a posting list iteration, the right postings for a transaction would be picked, sorted by integer ids, and then merge-sort operation is run between these delta postings and the underlying posting list state.

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。

Earlier iterations of this mechanism were aimed at keep- ing the delta layer sorted by integer ids as well, overlaying it on top of the state to avoid doing sorting during the reads — any addition or deletion made would be consolidated based on what was already in the delta layer and the state. These iterations proved too complex to maintain for the team and suffered from hard to find bugs. Ultimately, that concept was dropped in favor of a simple understandable solution of pick- ing the right postings for a read and sorting them before itera- tion. Additionally, earlier APIs implemented both forward and backward iteration adding complexity. Over time, it became clear that only forward iteration was required, simplifying the design.

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。

There are many benefits in avoiding having to regenerate the posting list state on every write. At the same time, as deltas accumulate, the work of list regeneration gets delegated to the readers, which can slow down the reads. To find a balance and avoid gaining deltas indefinitely, we added a rollup mechanism.

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。

Rollups: As keys get read, Dgraph would selectively re- generate the posting lists which have a minimum number of deltas, or haven’t been regenerated for a while. The regener- ation is done by starting from the latest state, then iterating over the deltas in order and merging them with the state. The final state is then written back at the latest delta timestamp, re- placing the delta and forming a new state. All previous deltas and states for that key can then be discarded to reclaim space.

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。

This system allows Dgraph to provide MVCC. Each read is operating upon an immutable version of the DB. Newer deltas are being generated at higher timestamps and would be skipped during a read at a lower timestamp.

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。

Transactions

交易

Dgraph has a design goal of being simple to operate. As such, one of the goals is to not depend upon any third party system. This proved quite hard to achieve while providing high availability for not only data but also transactions.

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。

Figure 4: MVCC

图 4：MVCC

While designing transactions in Dgraph, we looked at pa- pers from Spanner [?], HBase [?], Percolator [?] and others. Spanner most famously uses atomic clocks to assign times- tamps to transactions. This comes at the cost of lower write throughput on commodity servers which don’t have GPS based clock sync mechanism. So, we rejected that idea in fa- vor of having a single Zero server, which can hand out logical timestamps at a much faster pace.

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 GPS 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。

To avoid Zero becoming a single point of failure, we run multiple Zero instances forming a Raft group. But, this comes with a unique challenge of how to do handover in case of leader relection. Omid, Reloaded [?] (referenced as Omid2) paper handles this problem by utilizing external system. In Omid2, they run a standby timestamp server to take over in case the leader fails. This standby server doesn’t need to get the latest transaction state information, because Omid2 uses Zookeeper [?], a centralized service for maintaining transac- tion logs. Similarly, TiDB built TiKV, which uses a Raft-based replication model for the key-values. This allows every write by TiDB to automatically be considered highly-available. Sim- ilarly, Bigtable [?], uses Google Filesystem [?] for distributed storage. Thus, no direct information transfer needs to happen among the multiple servers forming the quorum.

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，Bigtable [?] 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。

While this concept achieves simplicity in the database, we were not entirely thrilled with this idea due to two reasons. One, we had an explicit goal of non-reliance on any third- party system to make running Dgraph operationally easier, and felt that a solution should be possible without pushing

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

synchronous replication within Badger (storage). Second, we wanted to avoid touching disk unless necessary. By having Raft be part of the Dgraph process, we can find-tune when things get written to state to achieve better efficiency. In fact, our implementation of transactions don’t write to DB state on disk until they are committed (still written to Raft WAL).

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）。

We closely looked at HBase papers ( [?], [?]) for other ideas, but they didn’t directly fit our needs. For example, HBase pushed a lot of transaction information back to the client, giving them critical information about what they should or should not read to maintain the transactional guarantees. This however, makes the client libraries harder to build and maintain, something we did not like. On top of that, a graph

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

Algorithm 1 Commit (Ts, Keys)

算法 1 提交（Ts、Keys）

1: for each key kKeys do

1：对于每个键kKeys做

∈

∈

2:if lastCommit(k) > Ts then

2：如果 lastCommit(k) > Ts 那么

3:Propose(Tsabort)

3:Propose(Tsabort)

←

←

4:return

4：返回

5:end if

5：结束如果

6: end for

6：结束

7: Tc GetTimestamps(1) 8: for each key k Keys do 9: lastCommit(k) Tc

7：Tc GetTimestamps(1) 8：对于每个键 k 键执行 9：lastCommit(k) Tc

←

←

∈

∈

←

←

10: end for

10：结束

11: Propose(Ts ← Tc)

11: 提议(Ts ← Tc)

query can touch millions of keys in the intermediate steps, it’s

查询可以在中间步骤中触及数百万个键，它是

expensive to keep track of all that information and propagate that to the client.

跟踪所有这些信息并将其传播给客户端的成本很高。

Aim for Dgraph client libraries was to keep as minimal state as possible to allow open-source users unfamiliar with the internals of Dgraph to build and maintain libraries in languages unfamiliar to us (for example, Elixir).

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: Do I describe the first iteration?

// TODO：我是否描述了第一次迭代？

We simply could not find a paper at the time which de- scribed how to build a simple to understand, highly-available transactional system which could be run without assuming that the storage layer is highly available. So, we had to come up with a new solution. Our second iteration still faced many issues as proven by Jepsen tests. So, we simplified our second iteration to a third one, which is as follows.

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。

Lock-Free High Availability Transaction Processing

无锁高可用性事务处理

Dgraph follows a lock-free transaction model. Each transac- tion pursues its course concurrently, never blocking on other transactions, while reading the committed data at or below its start timestamp. As mentioned before, Zero leader maintains an Oracle which hands out logical transaction timestamps to Alphas. Oracle also keeps track of a commit map, storing a conflict key latest commit timestamp. As shown in al- gorithm ??, every transaction provides the Oracle the list of conflict keys, along with the start timestamp of the transac- tion. Conflict keys are derived from the modified keys, but are not the same. For each write, a conflict key is calculated depending upon the schema. When a transaction requests a commit, Zero would check if any of those keys has a commit timestamp higher than the start timestamp of the transaction. If the condition is met, the transaction is aborted. Otherwise, a new timestamp is leased by the Oracle, set as the commit timestamp and conflict keys in the map are updated.

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。

→

→

The Zero leader then proposes this status update (commit or abort) in the form of a start commit ts (where commit ts = 0 for abort) to the followers and achieves quorum. Once quorum is achieved, Zero leader streams out this update to the subscribers, which are Alpha leaders. To keep the design

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

→

→

Algorithm 2 Watermark: Calculate DoneUntil (T , isPending)

算法 2 水印：计算 DoneUntil (T , isPending)

1: if T / MinHeap then

1：如果 T/MinHeap 那么

∈

∈

2:MinHeapT

2：最小堆

←

←

3: end if

3：结束如果

4: pending(T )isPending

4：待定(T)isPending

←

←

5: curDoneTsDoneUntil

5：curDoneTsDoneUntil

←

←

6: for each minTsMinHeap.Peek() do

6：对于每个 minTsMinHeap.Peek() 做

∈

∈

7:if pending(minTs) then

7：如果挂起（minTs）那么

8:break

8：休息

9:end if

9：结束如果

10:MinHeap.Pop()

10:MinHeap.Pop()

11:curDoneTsminTs

11:curDoneTsminTs

←

←

12: end for

12：结束

13: DoneUntil ← curDoneTs

13：DoneUntil ← curDoneTs

simple, Zero does not push to any Alpha leader. It is the job of (whoever is) the latest Alpha leader to establish an open stream from Zero to receive transaction status updates.

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。

Along with the transaction status update, Zero leader also sends out a MaxAssigned timestamp. MaxAssigned is cal- culated using a Watermark algorithm ??, which maintains a min-heap of all allocated timestamps, both start and com- mit timestamps. As consensus is achieved, the timestamps are marked as done and MaxAssigned gets advanced to the maximum timestamp up until which everything has achieved consensus as needed. Note that start timestamps don’t typi- cally need a consensus (unless lease needs to be updated) and get marked as done immediately. Commit timestamps always need a consensus to ensure that Zero group achieves quorum on the status of the transaction. This allows a Zero follower to become a leader and have full knowledge of transaction statuses. This ordering is crucial to achieve the transactional guarantees as we will see below.

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。

Once Alpha leaders receive this update, they would propose it to their followers, applying the updates in the same order. All Raft proposal applications in Alphas are done serially. Alphas also have an Oracle, which keeps track of the pending transactions. They maintain the start timestamp, along with a transaction cache which keeps all the updated posting lists in

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

Figure 5: MaxAssigned watermark. Open circles represent and filled circles represent done. Start timestamps 1, 2, and 4 are immediately marked as done. Commit timestamp 3 begins and must have consensus before it is done. Watermark keeps track of the highest timestamp at and below which everything is done.

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。

Figure 6: The MaxAssigned system ensures that linearizable reads. Reads at timestamps higher than the current MaxAs- signed (MA) must block to ensure the writes up until the read timestamp are applied. Txn 2 receives start ts 3, and a read at ts 3 must acknowledge any writes up to ts 2.

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。

memory. On a transaction abort, the cache is simply dropped. On a transaction commit, the posting lists are written to Bad- ger using the commit timestamp. Finally, the MaxAssigned timestamp is updated.

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。

Every read or write operation must have a start times- tamp. When a new query or mutation hits an Alpha, it would

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

transactions and linearizable reads.

事务和线性化读取。

For correctness, only Zero leader is allowed to assign times- tamps, uids, etc. There are edge cases where Zero followers would mistakenly think they’re the leaders and serve stale data — Dgraph does multiple things to avoid these scenarios.

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。

If a Zero leadership changes, the new leader would lease out a range of timestamps higher than the previous leader has seen. However, an older commit proposal stuck with the older leader can get forwarded to the new one. This can allow a commit to happen at an older timestamp, causing failure of transactional guarantees. We avoid this by disallowing Zero followers forwarding requests to the leader and rejecting those proposals.

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: We should have a membership section, which explains how membership works and is transmitted to Alphas.

// 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。

Every membership state update streamed from Zero re- quires a read-quorum (check with Zero peers to find the latest Raft index update seen by the group). If the Zero is behind a partition, for example, it wouldn’t be able to achieve this quorum and send out a membership update. Alphas expect an update periodically and if they don’t hear from the Zero leader after a few cycles, they’d consider the Zero leader defunct, abolish connection and retry to establish connection with a (potentially different) healthy leader.

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。

Consistency Model

一致性模型

Dgraph supports MVCC, Read Snapshots and Distributed ACID transactions. The transactions are cluster-wide across universal dataset – not limited by any key level or server level restrictions. Transactions are also lockless. They don’t block/wait on seeing pending writes by uncommitted trans- actions. They can all proceed concurrently and Zero would choose to commit or abort them depending on conflicts.

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。

Considering the expense of tracking all the data read by a single graph query (could be millions of keys), Dgraph does not provide Serializable Snapshot Isolation. Instead, Dgraph provides Snapshot Isolation, tracking writes which is a much more contained set than reads.

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 hands out monotonically increasing timestamps (represented by T ) for transactions (represented by Tx). Ergo, if any transaction Txi commits before Txj starts, then

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

start

开始

ask Zero to assign a timestamp. This operation is typically

要求零分配时间戳。这个操作通常是

Txi

锡

commit

犯罪

T

T

< TTxj . Any commit at Tcommit is guaranteed to be

< TTxj 。在 Tcommit 的任何提交都保证是

batched to only allow one pending assignment call to Zero leader per Alpha. If the start timestamp of a newly received query is higher than the MaxAssigned registered by that Al- pha, it would block the query until its MaxAssigned reaches or exceeds the start ts. This solution nicely tackles a wide- array of edge case scenarios, including Alpha falling back or going behind a network partition from its peers or just restart- ing after a crash, etc. In all those cases, the queries would be blocked until the Alpha has seen all updates up until the timestamp of the query, thus maintaining the guarantee of

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

seen by a read at timestamp Tread by any client, if Tread >

由任何客户端在时间戳 Tread 处读取，如果 Tread >

Tcommit. Thus, Dgraph reads are linearizable. Also, all reads are snapshots across the entire cluster, seeing all previously committed transactions in full.

提交。因此，Dgraph 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。

As mentioned, Dgraph reads are linearizable. While this is great for correctness, it can cause performance issues when a lot of reads and writes are going on simultaneously. All reads are supposed to block until the Alpha has seen all the writes up until the read timestamp. In many cases, operators would opt for performance over achieving linearizablity. Dgraph

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

provides two options for speeding up reads:

提供两种加速读取的选项：

A typical read-write transaction would allocate a new timestamp to the client. This would update MaxAssigned which would then flow via Zero leader to Alpha leaders and then get proposed. Until that happens, a read can’t proceed. Read-only transactions would still require a read timestamp from Zero, but Zero would opportunistically hand out the same read timestamp to multiple callers, allowing Alpha to amortize the cost of reaching MaxAssigned across multiple queries.

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。

Best-effort transactions are a variant of read-only trans- actions, which would use an Alpha’s observed MaxAssigned timestamp as the read timestamp. Thus, the receiver Alpha does not have to block at all and can continue to process the query. This is the equivalent of eventual consistency model typical in other databases. Ultimately, every Dgraph read is a snapshot over the entire distributed database and none of the reads would violate the snapshot guarantee. 1

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

Replication

复制

Most updates to Dgraph are done via Raft. Let’s start with Alphas which can push a lot of data through the system. All mutations and transaction updates are proposed via Raft and are made part of the Raft write-ahead logs. On a crash and restart, the Raft logs are replayed from the last snapshot to bring the state machine back up to the correct latest state. On the flip side, the longer the logs, the longer it takes for Alpha to replay them on a restart, causing a start delay. So, the logs must be trimmed by taking a snapshot which indicates that the state up until that point has been persisted and does not need to be replayed on a restart.

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。

As mentioned above, Alphas write mutations to the Raft WAL, but keep them in memory in a transaction cache. When a transaction is committed, the mutations are written to the state at the commit timestamp. This means that on a restart, all the pending transactions must be brought back to memory via the Raft WAL. This requires a calculation to pick the right Raft index to trim the logs at, which would keep all the pending transactions in their entirety in the logs.

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。

One of the lessons we learnt while fixing Jepsen issues was that, to improve debuggability of a complex distributed system, the system should run like clock work. In other words, once an event in one system has happened, events in other systems should almost be predictable. This guiding principle determined how we take snapshots.

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 allows leaders and followers to take snapshots independently of each other. Dgraph used to do that but that brought unpredictability to the system and made debugging

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

1Note however that a typical Dgraph query could hit multiple Alphas in various groups — some of these Alphas might not have reached the read timestamp (initial Alpha’s MaxAssigned timestamp) yet. In those cases, the query could still block until those Alphas catch up.

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。

much harder. So, keeping with the hard learnt lesson of pre- dictability principle, we changed it to make the leader calcu- late the snapshot index and propose this result. This allowed leader and followers to all take snapshot at the same index, exactly the same time (if they’re generally caught up). Further more, this group level snapshot event is then communicated to Zero to allow it to trim the conflict map by removing all entries below the snapshot timestamp. Following this chain of events in logs has improved debuggability of the system dramatically.

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 only keeps metadata in Raft snapshots, the actual data is stored separately. Dgraph does not make a copy of that data during snapshot. When a follower falls behind and needs a snapshot, it asks the leader for it and leader would stream the snapshot from its state (Badger, just like Dgraph, supports MVCC and when doing a read at a certain times- tamp, is operating upon a logical snapshot of the DB). In the previous versions, follower would wipe out its current state before accepting the updates from the leader. In the newer versions, leader can choose to send only the delta state up- date to the follower, which can decrease the data transmitted considerably.

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。

High Availability and Scalability

高可用性和可扩展性

Dgraph’s architecture revolves around Raft groups for update log serialization and replication. In the CAP throrem, this follows CP, i.e. in a network partition, Dgraph would choose consistency over availability. However, the concepts of CAP theorem should not be confused with high availability, which is determined by how many instances can be lost without the service getting affected.

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。

In a three-node group, Dgraph can loose one instance per group without causing any measurable impact on the function- ality of the database. However, loosing two instances from the same group would cause Dgraph to block, considering all updates go through Raft. In a five-node group, the number of instances that can be lost without affecting functionality is two. We do not recommend running more than five replicas per group.

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。

Given the central managerial role of Dgraph Zero, one might assume that Zero would be the single point of failure. However, that’s not the case. In the scenario where Zero follower dies, nothing changes really. If the Zero leader dies, one of the Zero followers would become the leader, renew its timestamp and uid assignment lease, pick up the transaction status logs (stored via Raft) and start accepting requests from Alphas. The only thing that could be lost during this transition are transactions which were trying to commit with the lost Zero. They might error out, but could be retried. Same goes for Alphas. All Alpha followers have the same information as the Alpha leader and any of the members of the group can be lost without losing any state.

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 can support as many groups as can be represented

Dgraph 可以支持尽可能多的组

by 32-bit integer (even that is an artificial limit). Each group can have one, three, five (potentially more, but not recom- mended) replicas. The number of uids (graph nodes) that can be present in the system are limited by 64-bit unsigned integer, same goes for transaction timestamps. All of these are very generous limits and not a cause of concern for scalability.

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。

Queries

查询

A typical Dgraph query can hit many Alphas, depending upon where the predicates lie. Each query is sub-divided into tasks, each task responsible for one predicate.

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。

Traversals

遍历

Dgraph query tasks (henceforth referred to as tasks) are gen- erally built around the mechanism of converting uid list to matrix during traversal. The query can have a list of uids to traverse, the execution engine would do lookups in Badger concurrently to get the posting lists for each Uid (note that predicate is always part of the task), converting each uid to a list. Thus, a task query would return a list of Uid lists, aka UidMatrix. If the predicate holds a value (example, predicate name), the UidList returns a list of values, aka ValueMatrix. A predicate could allow only one uid/value, or allow mul- tiple uids/value. This mechanism works correctly in either of those scenarios. If the posting list only has one uid/value, the resulting list would only have one element. A matrix in this case would have a list of lists, each list with zero or one element. Note that there’s parity between the index of the Uid in list and the index of the list in UidMatrix. So, Dgraph can accurately maintain the relationships.

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。

A ValueMatrix is typically the leaf in the task tree. Once we have values, we just need to encode them in the results. However, a task with UidMatrix result would typically have sub-tasks. Those sub-tasks would need a query UidList for processing. Dgraph would merge-sort the UidMatrix into a single, sorted list of Uids, which would be copied over to the sub-tasks. Each sub-task could similarly run expand on the same or other predicates.

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。

Functions

职能

Dgraph also supports functions. These functions provide an easy way to query Dgraph when the global uid space needs to be restricted to a small set (or even a single uid). Functions also provide advanced functionality like regular expressions, full-text search, equality and inequality over sortable data types, geo-spatial searches, etc. These functions are also en- coded into a task query, except this time they don’t start with a UidList. The task query instead contains tokens, derived from the tokenizers corresponding to the index these functions are using (as explained above). Most functions require some sort of index to operate, for example, regular expression queries

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

use trigram indexing, geo-spatial queries uses S2-cell based geo indexing and so on... As described in section above, in- dexing keys encode predicate and token, instead of a predicate and uid. So, the mechanism to fill up the matrix is the same as in any other task query. Only this time, we use list of tokens instead of a list of Uids as the query set.

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。

Filters

过滤器

The technique described above works for traversals. But, fil- ters (intersections) are a big part of user queries. Each task contains a UidList as a query and a matrix as a result. Task also stores a resulting uid list, which can store a uid set from the resulting UidMatrix. Depending upon whether filters are applied or not, this uid set can be the same as merge-sorted UidMatrix or a subset of it.

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。

Filters are a tree in their own right. Dgraph supports AND, OR and NOT filters, which can be further combined to create a complex filter tree. Filters typically consist of functions which can ask for more information and are represented as tasks. These tasks execute in the same mechanism described above, but do one additional thing. The tasks also contain the source list of Uids (the resulting set from the parent task to which the filter is being applied to). This list of uids is sent as part of the filter task. The task uses these uids to perform any intersections at the destination server, returning only a subset of the results, instead of retrieving all results for the task. This can significantly cut down the result payload size while also allowing optimizations during filter task execution to speed things up. Once the results are returned, the co-ordinator server would stitch up the results using the AND, OR or NOT operators.

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。

Intersections

交叉路口

The uid intersection itself uses three modes of integer inter- section, choosing between linear scan, block jump or binary search depending upon the ratio of the size of the results and the size of the source UidList to provide the best performance. When the two lists are of the same size, Dgraph uses linear scan over both the lists. When one list is much longer than other, Dgraph would iterate over the shorter list and do bi- nary lookups over the longer. For some range in between, Dgraph would iterate over the shorter and do forward seeking block jumps over the longer list. Dgraph’s block based integer encoding mechanism makes all this quite efficient.

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: Talk about ACID.

TODO：谈谈酸。

Future Work

未来的工作

We had removed data caching from Dgraph due to heavy read- write contention, and built a new, contention-free Go cache library to aid our reads. Work is underway in integrating that with Dgraph. Dgraph does not have any query or response

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

caching — such a cache would be difficult to maintain in an MVCC environment where each read can have different results, based on its timestamp.

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。

Sorted integer encoding and intersection is a hotly re- searched topic and there is a lot of room for optimization here in terms of performance. As mentioned earlier, work is underway in experimenting a switch to Roaring Bitmaps.

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。

We also plan to work on a query optimizer, which can better determine the right sequence in which to execute query. So far, the simple nature of GraphQL has let the operators manually optimize their queries — but surely Dgraph can do a better job knowing the state of data.

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。

Future work here is to allow writes during the shard move, which depending upon the size of the shard can take some time.

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: Add a conclusion.

TODO：添加一个结论。

Acknowledgments

致谢

Dgraph wouldn’t have been possible without the tireless con- tributions of its core dev team and extended community. This work also wouldn’t have been possible without funding from our investors. A full list of contributors is present here:

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

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

Dgraph is an open source software, available on

Dgraph 是一个开源软件，可在

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

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

More information about Dgraph is available on

有关 Dgraph 的更多信息，请访问

https://dgraph.io

https://dgraph.io