Index Provider技术解读

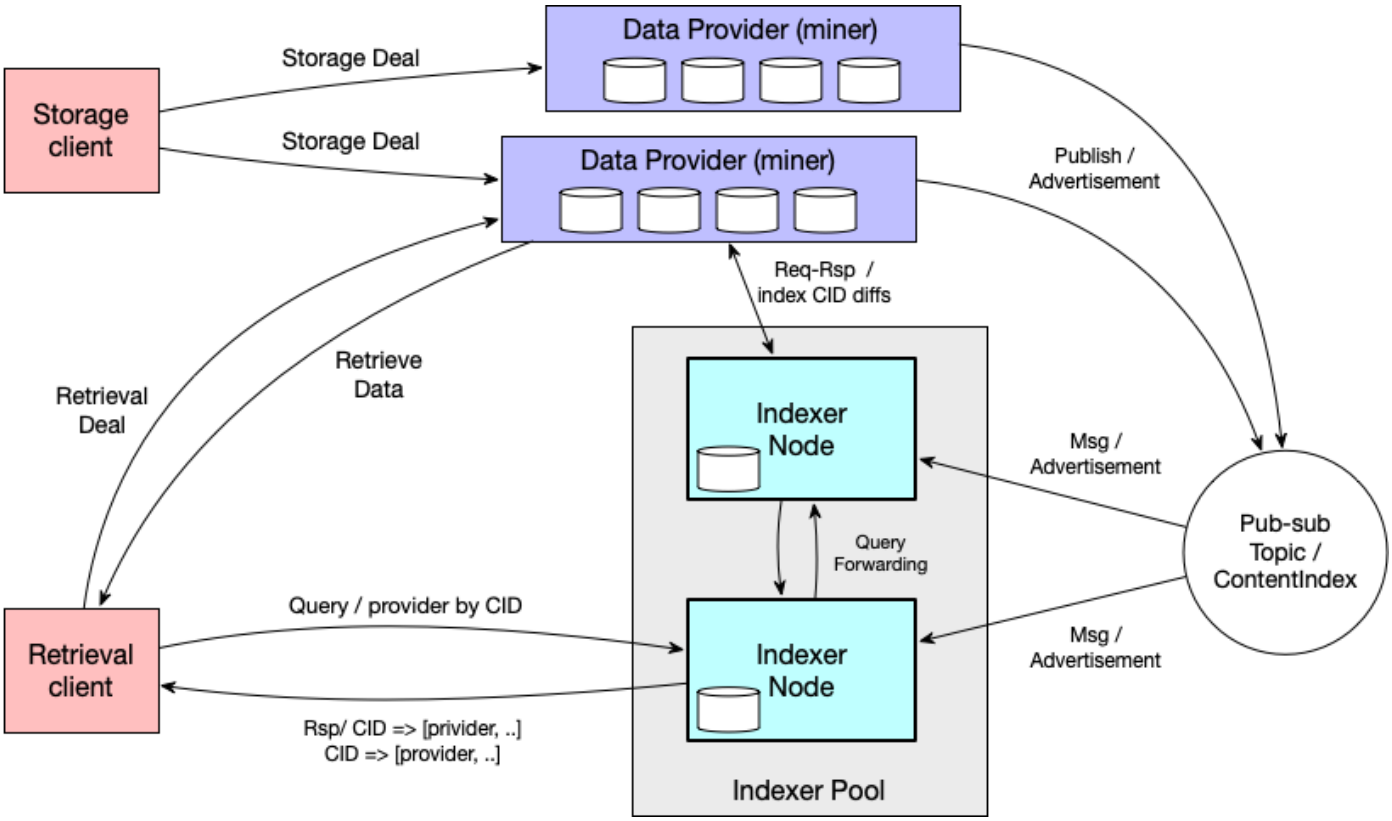
本文是Indexer-Node-Design的翻译,并加入了一些译者自己的理解.

概述

data provider 也就是SP(storage provider, miner), 在本地保存数据的索引. 它们需要发布持有数据的公告,以便潜在客户使用;当数据失效时还能够撤回这些公告.
indexer nodes 发现data provider并且追踪其的数据的变化.它们需要处理对数据的请求,解析后, 发送给对应的的data provider. 在无法处理某些请求内容时, 也可能将其转发到其它indexer.
indexer clients 向indexer发布类似'findProvider'的请求, 接收包含'provider'集合的响应. 响应中应该包含任何关于如何选择provider的信息, 以及provider被要求提供,以验证请求的信息(例如: deal ID)

indexer发现新的data provider后, 通过监听它们发布到gossip pub-sub相关主题上的推送, 接收其关于数据更新的通知. indexer还可以通过监控旷工在链上的活动来发现数据的变化. 这些链上的信息让indexers发现新的数据,并且数据的索引入口能够从某个已经存在或新的provider上获取. indexer可以自由的决定 是否, 何时, 从哪个provider拉取哪些类型的索引, 完全基于indexer对于此provider的策略.

为了处理来自任意数量provider的任何规模的索引数据, indexer节点需要能够水平扩展,以便在indexers集群中进行负载均衡. indexer集群的大小需要可以动态变化.



关键设计决策

- indexer 使用Mult-hash而非`CIDs`来指向索引内容
  CID的编/解码是独立于被索引的内容,以及从provider检索内容的方式的.

> 译者注: multihash是一种自描述的数据格式, 顾名思义, 支持不同算法的hash
>
> 0-4字节: hash算法, 4-8字节:hash值的长度, 8-末尾:hash值
>
> 此外,提取car文件的indics并转换为multihash, 具体细节可以参考: DAGStore.initializeShard函数的具体实现细节.

- 除非配置为仅在内存中操作, indexer会保留客户端请求过的multihas的内存缓存.
  由于客户通常只会请求某个对象的顶层multihash,故而索引中的大多数multihash并不会被请求. 因此只有被真正请求过的部分multihash才会被保留在缓存中.剩下的部分会被保存在较为节省空间的二级存储中.一种内存缓存的分代缓存回收机制防止了占用的内存无止境的增长.
- StoreTheHash作为持久化存储服务
  旨在高效的存储hash数据,只需2次磁盘读取就能获取任何`mulithash`. 由于非常可信的假设为能够节省显著的空间, 其被评价为更加成熟的系统. 然而这种节省需要非常显著才能在意义上超过新存储设施的开发成本和风险.indexer允许不同的实现, 以为不同的部署方案提供合适的选择. 拥有一种嵌入式的实现也许不但能减少管理工作, 而且能更加容易将indexer作为库来使用.
- indexer追踪`multihash`的分布和使用
  index会保持统计`multihash,`用于预测数据存储,和缓存的增长率,以及`multihash`在provider上的分布情况. 也会跟踪`multihash`的使用,展示客户对于各种不同`multihash`的需求情况.
- indexer不会索引`IDENTITY` 多重哈希
  indexer不会索引`IDENTITY`多重hash并从收到的广告中将此类内容过滤掉.(参考:Appendix C, Handling of IDENTITY Multihashes)

数据供应商接口

data provider在本地保存了其数据的索引,我们可以抽象的把provider暴露出来的资源想象成类似下面的树:

```
Catalog (List of Indexes)
Catalog/<id> (Individual Index)
Catalog/<id>/multihashes (list of multihashes in the Index)
Catalog/<id>/TBD (semantic for selection)
```

> **译者注:**
> 针对**Individual Index**和**list of multihashes in the Index**, 有人评论:
> 问: i don't understand the difference between these two
> 答: cids is the hamt of all cids. the full index.
> 大致可以理解为provider暴露3中类型的资源:
> `cids[hamt]`, `multihashes`, 以及`/TBD semantic for selection`(译者注:这是待确认的项)

每个provider都有一个用于存放1个或多个index的目录, 其中的当前索引是provider所知的所有`multihash`的集合, 其余的index是它的历史版本,每个都代表之前某个时间点的multihashes. 语意上来说,对provider对indexes目录的修改被视为发生在全局有序的添加/撤销日志中,每个都链接之前的操作. indexers通过追踪这些变化记录,来保持它们对provider索引的视图最新.

> 译者注:
> 理解像按时间排序的链表一样, 每个节点上都记录了对index的操作. 如果遍历重放整个链表, 就得到了最新的index集合.

索引更新公告(原文为:Advertisments)

> 译者注:这里的`Advertisement`应该理解为公告,provider在libp2b相关topic上发布的其存储数据的索引的相关更新.

provider在`content index`的gossipsub pub-sub的主题中公布其新索引, indexer订阅此主题. 公告消息的格式如下:

```
{ Index: <multihash of current index manifest>,
  Previous: <multihash of previous index manifest>,
  Provider: <libp2p.PeerID>,
  Signature: <signature over index, previous>
}
```

> 译者注:
> 大神们在这里对于到底应该放`multi-addr`,`miner-id`还是`peer-id`到清单中发生了激烈的讨论,
> 因为不管`multi-addr`,`miner-id`还是`peer-id`,
> 都最终都可以转换为`endpoint`以连接到`provider`.最终结果是放`peer-id`
> 基于几下几点原因:
>
> - miner的peerid是不变的, 但是`multiaddr`是变化的.(实际上为peerid也是可以变化的,只是multiaddr可能会随着外部的变化而变化, 比如IP地址变动)
> - 如果放`miner-id`就只能适用于filecoin上的存储提供者(miner).

当收到一个公告时,indexer会检查是否已经有了最新的`index ID`. 如果已经有了则忽略; 否则向provider发送请求,获取公告中,从`Previous`到`Index`的变化集合. indexer在内部处理`peer-id`到`endpoint`的映射.
除了监听pub-sub管道来获取公告的通知之外, indexer也可以直接从provider直接请求最近的公告. 这在发现新的provider时非常有用.

> 注:`Signature`字段是基于每个条目中'当前'和'Previous' ID计算的. 它保证根据签名的数据提供者, 这个顺序是正确的.

一个`Index ID`总是一个从CID提取的mutihash(因为索引化的内容并没有追踪内容的编码方式), 这应该是来源于由provider托管内容的默克尔树的CID.

数据供应商对Indexer请求的响应

基于`Provider-Indexer`的协议,indexer请求如下:

```
{ "prev_index_id": <get changes starting at this index>,
  "index_id": <get changes up to this index>,
  "start": n, // optional - start at this record (for pagination)
  "count": n, // optional - maximum number of records to fetch
}
```

之后, indexer收到可能分页的response, 其中包含了`changelog`的子集和相关的公告消息:

```
{ "totalEntries": n, // size of provider's catalog
  "error": "",       // optional. indicate the request could not be served
  "advertisement": {
    Index: n3 <ID of index manifest>,
    Previous: n2 <ID of previous index manifest>,
    Provider: <libp2p.AddrInfo>, Signature: <signature(index, previous)>
  },
  "start": n,         // starting entry number
  "entries": [
    {"add_multihashes": [<multihash>, ...], "metadata": "???",
     "del_multihashes": [<multihash>, ...]},
    {"add_multihashes": [<multihash>, ...], "metadata": "???"},
}
```

这是从请求中`prev_multihash`开始的, 应用到index数据的修改的集合. 每个译者注:response中`entries`字段中的entry包含, 一组provider新增的multihash;1组(可能有)被移除的不在有效的multihash;被限制(<=100字节)的`metadata`字段,包含特定于提供者与添加的multihash相关的数据.译者注:`metadata`到底是什么没说清楚.应该举个例子. `metadata`以protocal ID 作为前缀,后跟根据协议编码的数据.metadata的内容有provider提供,如果需要的大小超过的限制,元数据中应该包含provider存储的ID,标识模式复杂数据.

值得注意的是,同一个multihash可能重复出现在同一个provider的多个不同的公告条目中.例如, 假设某个filecoin中的miner收到一个某部分数据曾经被索引过的新的deal, 其中包含一个CID, CID中包含一个multihash. 那么,可能会在下一次的公告中为这个multihash包含一个带有新`metadata`(例如:deal的新过期时间)的条目,以便将此更新通知到indexer节点.

当indexer发现provider中的某个multihash已经存在,则只更新`metadata`就就可以了.最终,如果包含此multihash的所有deal都过期了,provider将此multhash添加到下一次公告的`del_multihashes中, 以便通知indexer节点.

响应分页

请求的响应可能由于请求指定了最大条目数量或者传输的条目数量达到了配置的最大值而分页.为了获取其余的条目,需要在后续的请求中设置start值为之前获取过的所有条目的数量. 所:如果`totalEntries`是`100`, 且response中包含了50条, 接下来的请求就应该将`start`这是为`50`, 直到indexer获取完整的条目.

链式响应

indexer保留了公告过的索引的多重哈希的记录, indexer一直往后回退, 直到某个response中包含的`Previous`multihash等于indexer的最新值, 或者到达链的头部(没有previous了)为止.当所有的链上所有缺失的索引都被按顺序的被应用.以更新indexer的multihash记录和数据提供者.

> 译者注: 这里描述的方法是不是和filecoin链的同步逻辑极其相似?

无响应的数据供应商

indexer可以配置为在它们没有通过gossip pub-sub收到任何内容持续一定时间的情况下,定期轮询provider来更新公告.此外可以将indexer配置为,在provider既不响应轮询请求,也不主动发送任何更新通知一定次数的情况下, 直接清除provider的数据.

清单 (post MVP - TBD if required)

随着provider的index变化,每次修改都导致新增一组multihash到index中. 索引multihash与其对应的multihash集合被记录在通过IPFS提供的的Manifest记录中.这意味着provider公告的每次更改都有对应的manifest.

每个provider同样应该支持最新Manifest有效的IPNS(译者注:IPNS: Inter-Planetary Naming Service, 用于将url映射为IPFS系统中的一串hash, PieceMount.Deserialize有关?) 地址一个独立index条目的manifest应该是下面的结构:

```
{
"Version": "1",
"Full": "<multihash of full index>",
"Selected": "<multihash of selective index>",
"Timestamp": "<time stamp index was provided>",
"Supplants": "<multihash of previous manifest this entry replaces>", // optional
}
```

如果indexer中没有某个provider的前一个记录,indexer可以用manifest(假设piece CIDs在这里可用)来为此provider初始化其multihash. indexer可以用这种方法来更新其某个provider的记录,但是处理manifest引用的multihash全集比只请求获取provider的修改日志工作量大得多.indexer需要计算每两个前后清单之间的变化来确认哪些部分需要更新. 或者, indexer需要移除provider的所有记录,然后重新保存manifest中的所有multihash.

如果provider为了最新manifest提供了`IPNS`记录, indexer就能够定期检查最新的manifest是否变化. 如果变化, 则触发indexer向provider请求更新.这是接收公告或者直接寻轮provider的一种可能的替代方案.indexer能在多大程度上利用这一点尚待确认.

发现数据提供商

当pub-sub通道上出现了新的provider的公告时, 新的provider就被发现了. priovider也能由独立的代理发现.indexer并不清楚agent是如何发现provider的. 只有当代理将新发现的provider的`libp2b-peerID`发送到indexer的`discovery`接口上时,indexer才知道.

这样设计好处是,允许任何类型的代理被独立于indexer构建出来, 并且能够将新发现的provider通知给任意的indexer.代理也许拥有区块扫描的能力,或者它仅仅是简单的读取一个地址文件, 然后将其报告给indexer.

公告过滤

可以使用黑/白名单过滤公告. 这些名单定义在indexer的配置中.名单中包含provider的ID.

- 如果使用白名单,则provider没有在白名单中的公告会被忽略.
- 如果使用黑名单,则来自任何名单上的provider的公告都会被阻止.
- 如果黑白名单都被配置,则优先使用白名单,忽略黑名单.

初始indexer部署时,使用了白名单,选则较小的provider的集合来开始.

轮询更新

Indexer节点可能会轮询已知的provider更新index.这在indexer或者provider离线或错过了某些公告, 或者provider选择不发布更新公告的情况下是有非常有用.indexer向provider发送最新公告的请求.provider回复最新的公告. indexer处理这个公告的方式与其处理通过gossip pub-sub收到的公告完全一样.

轮询计划

轮询计划是在indexer配置文件中的`UpdateSchedule`节中指定的,这里有一个默认的`"Poll"`项指定了`定时任务`的值.如果没有设置, 则默认不会有轮询任务.对于不同的provider也有同样的配置项,用于覆盖默认全局配置.

```
"UpdateSchedule": {
  "Poll": "0 0 * * *",
  <provider-id>: {
    "Poll": "0 /12 * * *"
  },
```

供应商验证

索引的增加和删除应该与其它需要验证身份的资源一样,一视同仁.provier不允许伪造或篡改其它provider的索引. 每个索引更新的公告都应该包含provider的签名. 如果indexer无法验证签名, 则公告被丢弃.indexer用provider在公告中提供的provider ID来查找签名证书.
旷工的公钥应该在链上并且能够通过lotus节点来查找.其它类型的provider也许需要不同的方式来获取它们的公钥(证书).

Indexer的数据存储

Response缓存(在内存中)

压缩字典树(又名:基数树), 或其它能够提供时间/空间性能的数据结构, 将被用来在内存中缓存请求的响应. 要求O(n)的搜索时间复杂度,且比hash映射的存储更加高效.检索时间必须在10ms以内. 存储的数据是multihash到proivder及其指定的metadata的列表的映射.

数据存储

- Key: multihash(二进制)
- Value: [{provider_id:<libp2p_peer_id>, metadata:"bytes"}, ..]

提供multihash来进行查找,mutilhash解码后的原始hash字节被用于在缓存中进行查找.
存储的值是`provider_id`-`piece_multihash`对的列表.实际的存储并不是json格式, 只是在输出到客户端时编码为json.不同的multhash 键映射到相同的值时,不应该存储重复的拷贝, 而应该使用引用.

缓存淘汰策略

当缓存到达配置容量的50%时, 就会创建一个新的缓存实例与之前的并存.当收到客户端请求, 优先从新的缓存查找, 然后才查找旧的.当缓存大小到达配置的限制,老的缓存就会被移除,再次重新创建一个新的空缓存出来,循环往复.这种策略是为了避免在缓存中保存时间戳或者LRU计数器译者注:为了更节省空间?
当一个indexer被引导并获取其初始数据时,其缓存可以使用这种策略填充和管理,以便一开始就加载部分或者所有的multihash. 随着时间推移,只有客户需要的数据才会被保留.

> 译者注:这里讲了一些缓存上的细节, 但引出了许多疑问:
>
> 1. 文中说查找数据时, 先从新的缓存上查询然后再查老的缓存, 很明显当新缓存才创建的一段时间, 并没有什么数据,此时优先从其中开始查询很明显命中率是非常低的.为什么不优先从老的开始查询呢?
> 2. 如果优先从新的开始查询, 并没有命中, 然后再从老的查询, 那么会把数据写入到新的缓存中吗?
> 3. 文中说当缓存大小达到限制时, 会移除老的缓存, 带来的问题是: 突然移除一半缓存, 是否会导致缓存的命中突然出现猛烈的下降?

> 这是留下来的疑问, 也是需要我们思考和弄清楚的地方.

负缓存(译者注:可以快速返回一个`not exists`)

查找没有命中缓存时就会去二级存储中查找,这会花费更多的时间和更大的资源消耗.negative cache用于快速响应Indexer上不存在的multihash的请求. 如果此缓存具有不同的大小或者其中的item具有不同的生命周期,它可以独立于主缓存.

缓存的指标

- 缓存命中,丢失
- 负缓存命中和丢失
- multihash条目数量,内存用量
- 命中耗时,丢失耗时
- Rotations译者注:不太理解这个指标..
- Provider请求计数

值存储(持久化)

已有解决方案:

- Postgres, CassandraDB, FoundationDB, etc.
- Pogreb(imbedded), RocksDB(high scale, battle-tested and Facebool)

Indexer和miner之间的规模差异妨碍它们使用同样的解决方案.

data client不会请求大量的index数据,因此不需要将这些数据保存在主内存中.这些数据应该以空间消耗最小化的方式来存储,但当人工检索时,任然需要做出亚秒级别延迟的响应.数据库`storethehash`是为高效存储hash数据而专门设计的,如果要支持删除和更新,需要对其进行一些修改.
二级存储中的数据与主缓存中的数据一样.译者注:这里的'数据一样',是说的种类, 而不是说的内容,但取决于不同的存储情况,可能有不同的形式(待定中).二级存储的配置是可选的,如果不配置,indexer只会操作内存数据,替代在二级存储中写入更新, 且不会循环它的缓存(译者注: 这里的'循环''是前面讲的缓存淘汰机制).

存储指标

- Multihashes found, multihashes not found
- Number of multihash entries, storage used
- Found latency, not found latency
- Adds, deletes.
- Compaction time, storage reclaimed
- Multihash count by provider
- Multihash count + and - delta per day, week, month

客户端接口

> 译者注:client就是向indexer发送multihash, 查询multihash对应的proivder的`libp2p_peer_id`和`metadata`, 所以略过,
> 感兴趣的可以去阅读原文.

管理相关接口

管理相关的接口包含两大类:

- 健康检查 - 输出正常运行时间和错误
- 指标 - 输出当前的指标译者注:之前的缓存和存储两个章节提及一些指标

健康检查接口主要用于自动检查indexer是否运行, 以及是否出现一些错误情况.
指标接口主要用于报告indexer节点当前的运行水平.

大小/规模预估

持有验证订单的旷工: 135
https://filplus.d.interplanetary.one/miners
存储客户在交易中使用改的数据量:~215 TiB (为客户分配的总量为~780TiB,但大部分并没有使用.译者注:cc sector?细节不清楚)

Dumbo drop(7月): 订单总容量约1600个驱动器(每个驱动器约7000个deal.) 译者注:这里的驱动器是说的sector吗?

网络应该允许数据供应商声明数十亿级别的内容地址.

Indexer配置项

> 译者注:略

Indexer扩展策略

- indexer集群:附录 A
- Indexer分层: 附录 B

参考文献

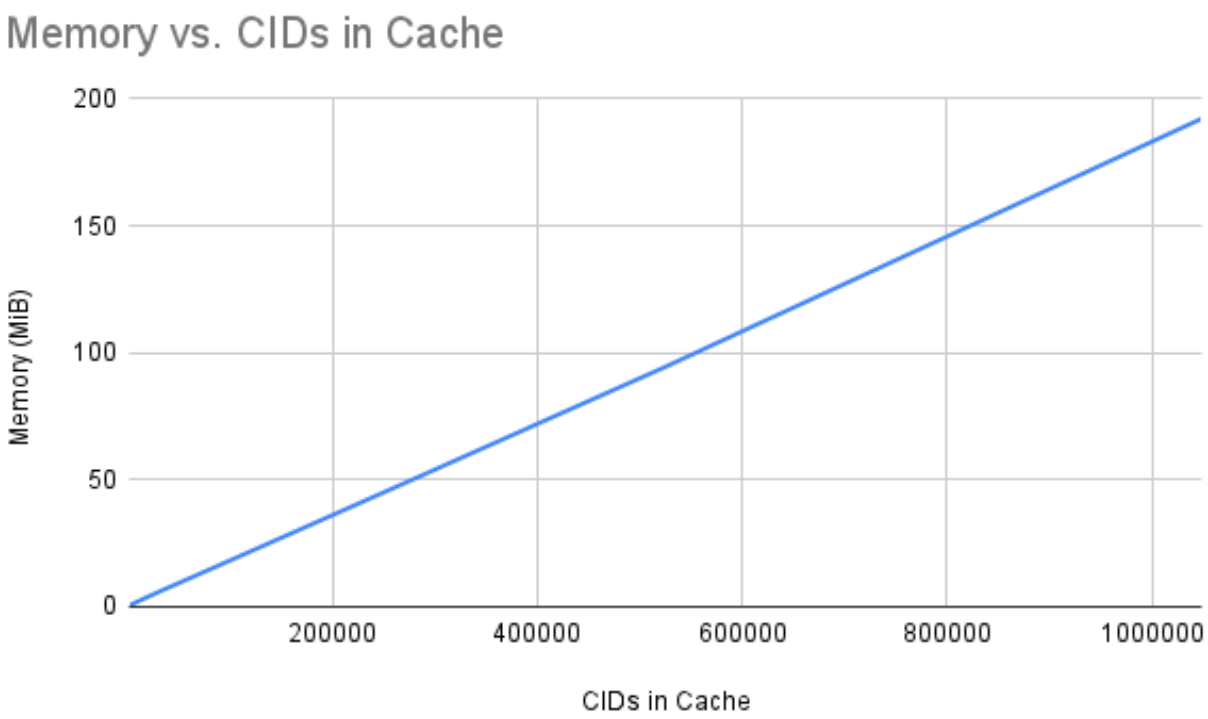
1. Miner(or dataset producer)<>Indexer Design
2. IPFS Collections spec
3. Composable Routing spec design
4. Composable Routing spec language v0
5. nitro wishlist for "large provider dht"

安全性和滥用问题

- 当被评定为`Bad`的miner被加入黑名单后,修改其'PeerId'继续重新连接.
- 不应该允许miner伪装成其它旷工来破坏别人的评级.
- 旷工对indexer发起泛洪攻击,不停的修改`PeerId`假装为不同的旷工.
- 大量随机multihash对indexer发起Dos攻击, 导致由于`负缓存`无法命被中, 增加二级存储访问.
- 防止DDOS攻击的旷工利用indexer的协议 - 我们需要确保将旷工运行的软件用于服务市场过程中的索引请求(not the critial miner process), 注意不要接收来自单个indexer的大量的庞大的数据内容的请求.
- 旷工运行它们自己的indexer偏向自己.
- 对hash数据加盐以掩盖译者注:不是很理解这个问题的危害?

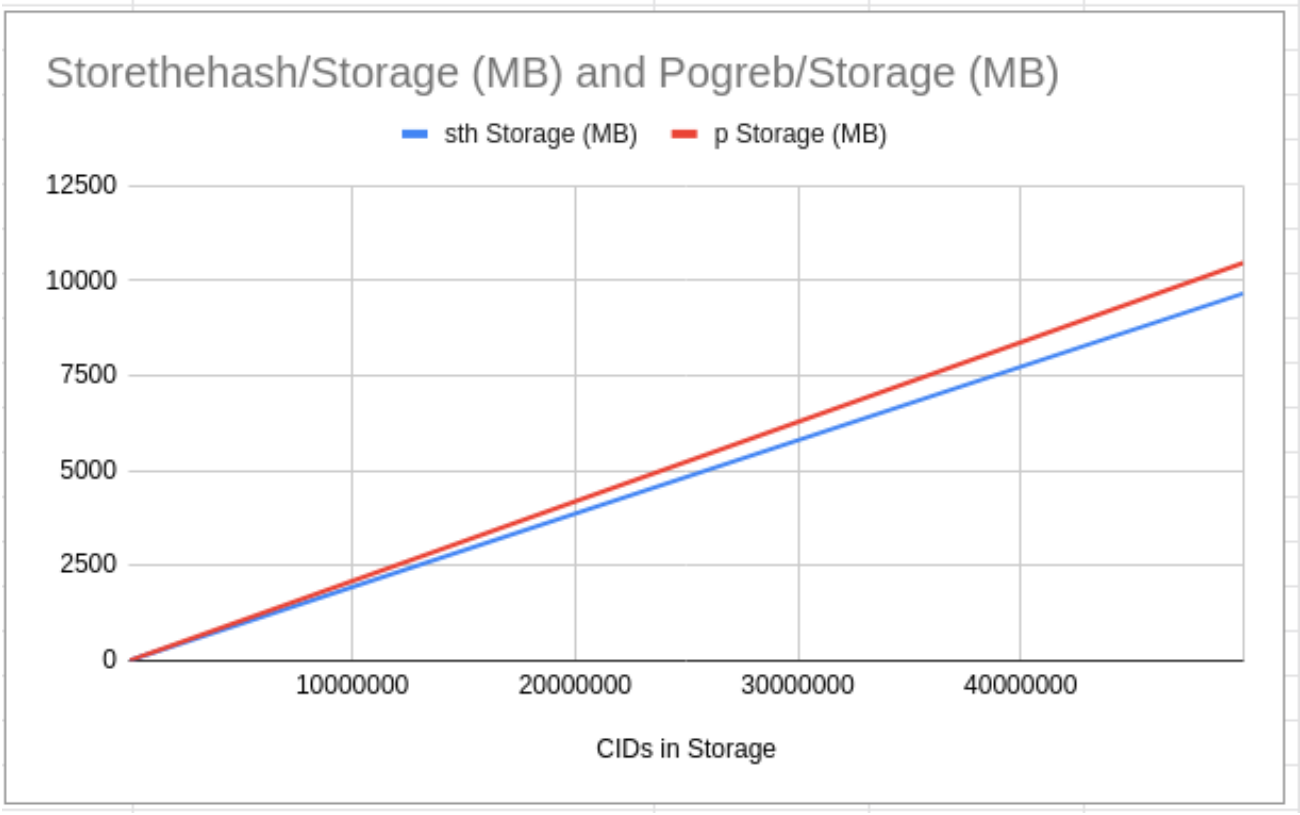
基准数据

缓存内存用量



存储100万条mutlhash大概使用200M内存.

持久化存储



图中展示的是`StoreTheHash`和`Pogreb`数据库的数量/用量图.

查询耗时

Aa	Storethehash数据库	Pogreb数据库	对比差异
平均耗时 获取单个(ms)	0.003	0.012	75%
平均耗时 并发-20个协程(ms)	0.008	0.047	82.98%

负载测试

测试环境为:AWS的t2.xlarge, st1存储使用`storetheindex`.实例对外暴露API监听请求.从我本机环境发送GET请求到API来产生负载. 请求中的multihash是从数据集中随机抽取, 遵守:(i) Zipf分布; (ii) uniform分布.

> 译者注:
> Zipf分布:20%的资源占了总请求量的80%. 及我们常说的2/8原则.
> uniform分布:随机分布, 均匀分布
>
> 下面的图标中, 分别展示了请求基于这两种分布的时候, 不同的性能表现.

	Aasth	sth+cache	cache
Zipf分布	中位数: 160ms	中位数: 160ms	中位数: 140ms
	95%耗时: 210ms	95%耗时: 200ms	95%耗时: 170ms
	~5100 rps requests per seconds	~5100 rps	~5100 rps
随机multihahs	中位数: 170ms	中位数:170ms	中位数:170ms
	95%耗时: 370ms	95%耗时:330ms	95%耗时:300ms
	~4200 rps	~4200 rps	~4200 rps

- 当增加大量并发用户时, 系统达到的限制是最大打开文件数. 我们确定已经将ulimits设置为最大值.系统支持为:当1000-1100左右的并发用户时u, 不会出现open file超过限制的问题; 并发超过1100的并发后, API就会由于打开的文件数太多产生一些等待, 但不会有请求被丢失,请求会被处理, 不会发生错误; 但对于zipf分布和uniformly分布, 我们似乎能达到的最大吞吐量分别为5100 rps(req/s) 和 4200 rps.
- 在检测中, 没有达到磁盘io的最大限制.
- 在为cache和storethehash运行基准测试时, 我们看到了get操作巨大的数量级差异.但在这些差异在网络中运行时却并不明显(这是有道理的).可以看出使用zipf分布来测试cache时,缓存大小对测试结果的影响很明显.为了充分利用缓存,我们也许需要额外的基准来确定最佳大小.

附录 A

通过在indexer集群的节点之间共享index数据实现扩展

本节将尝试定义一个最小化indexer集群的实现.目的是调查是否用这样一个简单的解决方案就可行且足够了, 或者它是否需要解决很多现有方案中已经解决的问题.
如果indexers需要支持的不仅仅是简单的分工,那么可以在现有分布式方案上构建接口来提供功能.有许多分布式数据方案, 例如: Cassanrda, CockroachDB, yugabytedb, 等. 也有像ke Zookeeper, etcd 这样的分布式共识的系统,

如果它们解决了indexer的必要问题,则应该考虑它们.

构建indexer集群

可以构建一个协作indexer集群,每个indexer值负责处理multihash键空间的不同区间.indexer配置指定集群中的所有成员并且所有的成员必须共享相同的配置.划分的区间由手工配置.

由于所有的成员都订阅了公告的pub-sub通道,消息公告会被所有的成员收到.当indexer收到消息时,检查当前索引multihash确定是否由它来负责处理.

处理Index更新

> 译者注:又把前面的链式更新讲了一遍,所以略.

添加索引节点到集群

支持动态添加节点到集群.修改每个indexer的配置重新指定了其需要处理的multihash的区间.当新的indexer上线,在每个indexer上执行一条重新平衡的命令, 集群中的indexer就自动重新平衡它们负责的区间. 这个操作将会让index重新加载其配置文件, 并将其不再负责的multihash发送给其它对应的indexer.当所有multihash被重新分配完成, 就可以将它们从存储中移除.在重新平衡过程中, 正好遇到一个查询其负责范围内的multihash请求, 如果索引器没有数据, 则请求将转发到之前处理multihash的indexer上(如果它还在线).

从集群移除索引节点

支持动态移除集群中的节点,方法与动态添加一样.只是被删除的节点, 必须最后命令平衡命令.

索引节点丢失

当索引节点丢失,其负责的multihash就不在可用了,请求会收到一个503(服务不可达)的错误. 除非此时有另外的indexer集群来支持multihash(请看:"主从复制"章节).如果没有备用的indexer集群,这些丢失的multihash会在每个provider中重新构建.

主从复制

通过配置第二个集群可以启动indexer集群的备用集群.当集群中的indexer丢失后,此indexer负责的multihash会直接路由到其它的集群上, 直到丢失的indexer重新上线并完成加载必要的multihash.节点重新上线会,通过自动从备用池中的索引节点同步来追上丢失的index.
备用集群与主集群之间的唯一区别是他们不与provider通信,而是依赖主集群来发送更新.备用集群可用于负载均衡.

> 译者注:
> 前面说当节点丢失,会将丢失节点负责的multihash请求路由给备用节点.
> 但又说备用集群依赖主集群来更新索引.
> **疑惑的地方就是:**
> 如果主节点丢失了, 而从节点又依赖主集群来更新, 那从节点岂不是没办法更新? 此时把multihash的请求路由到备节点岂不是白费心机?
> 所以,这里面应该还有一些细节没有说清楚..

客户请求路由

客户必须从正确的索引节点请求响应.如果client对此并不清楚,可以先向任意一个indexer发送一个请求. indexer会在返回值中包含的所有multihash; 对于其不包含的, 则会返回对应的indexer地址.

附录 B

通过索引器分层的方式来纵向扩展

索引节点可以分层, 顶层root索引节点知道一组其它的节点,并将请求委托过去.委托的节点可以把被委托的节点的响应缓存起来. 这样,没有一个单独的节点需要保存所有的数据供应商的索引.如果一个索引节点保存数据的量很大, 我们应该将其拆分成多个.
数据供应商可能有它们自己的索引节点,并或许填充多个索引节点来实现冗余.高等级的委托节点需要过滤从多个委托节点返回的响应,并在将唯一值缓存起来.
待定: 索引节点是否应该能够存储值,并委托给其它节点?

附录 C

处理 `IDENTITY` multihashes

Indexer node does not index `IDENTITY` multihashes. It is the responsibility of the provider to make sure any links within such multihashes are explicitly indexed instead. For example, if an `IDENTITY` multihash is root of a DAG that within its data block has Links to other non-identity CIDs, then the provider needs to publish advertisements that include the non-identity CIDs. Simply publishing the root `IDENTITY` CID is not sufficient and will result in the DAG being unindexed.

The rationale for this design decision is to discourage the indexing of root identities alone and instead encourage providers to index all their available content and their corresponding multihashes. This will facilitate partial DAG retrieval and open opportunities for caching when DAGs have overlapping content.