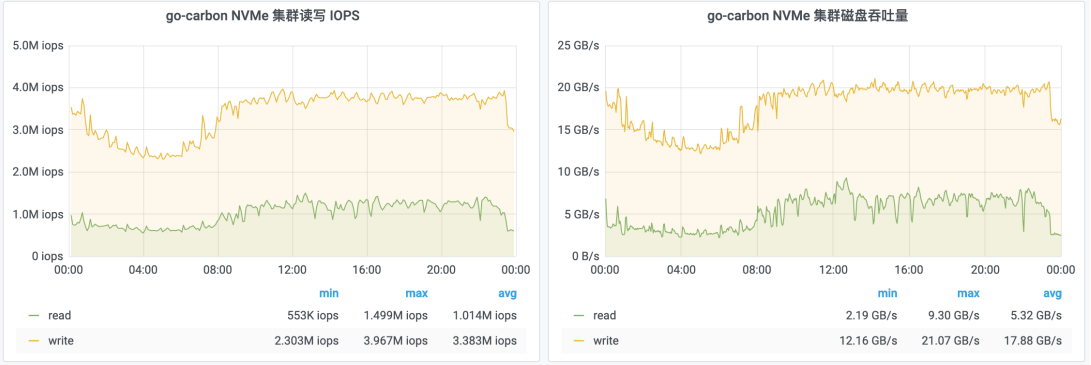
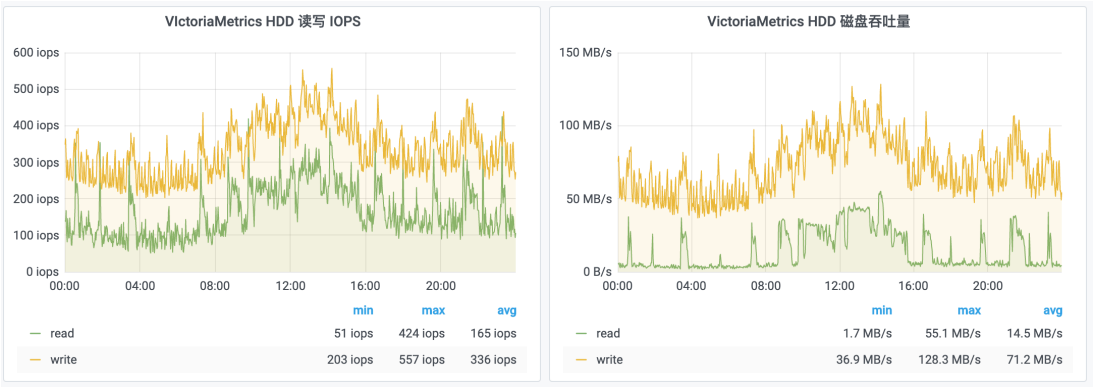
单机 20 亿指标，知乎 Graphite 极致优化！ · zhihu/promate Wiki

1. 1. 平均响应时间从 400ms 降低到 40ms，P95 从 800ms 降低到 100ms
   2. 部分复杂、长时间跨度的查询时间从数十秒降低到数十毫秒，提升 60~100 倍
   3. 在查询压力不变的情况下，内存和 CPU 开销减少 80%
2. 对磁盘性能要求降低 1w 倍
   1. 集群读写总 IOPS 从约 500w 降低到约 500
   2. 指标存储空间从原 100T 降低到约 15T，节省 85% 存储空间
   3. 磁盘从约 60 块 NVMe 固态缩减为 1 个 HDD 盘阵列
3. 分析能力极大增强
   1. 支持与 PromQL 结合增强查询能力，十万序列亿级时序点的运算秒级返回
   2. 历史指标不降低精度，返回的数据精度根据查询的时间范围动态调整

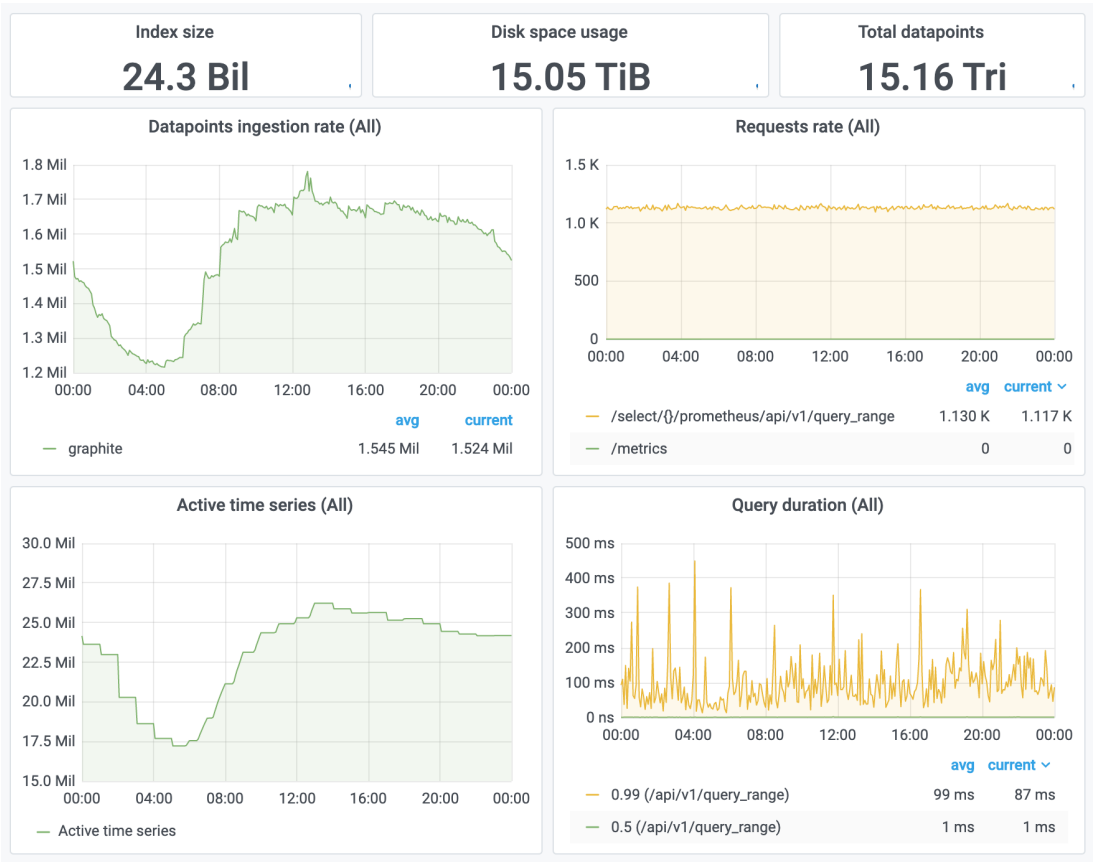
性能的提升是全方位的，这里重点展示一下磁盘 IOPS 的显著变化：



切换到新方案后降低 1w 倍：



在这套 Graphite On VictoriaMetrics 方案的加持下，我们甚至可以将知乎全量时序指标存储在单台 HDD 盘物理机节点。当然，实际部署时为了高可用和支持不同分析场景我们也做了多副本。



当前我们的一个节点上已经存储了约 22 亿个指标序列的 15 万亿个数据点，占用磁盘空间约 15T，承载约 1k 的报警检查 QPS。

深入理解 Whisper

Graphite 指标形式是以点分割的多维度标签组合而成，例如：

```
host.machine_name.memory.percent_used.value
```

在存储设计上，官方的 Whisper 引擎依赖文件系统目录层级，例如上述指标会被存储为：

```
host/machine_name/memory/percent_used/value.wsp
```

这样的存储架构无法存储指标的 Tag，所以现在 Graphite 只能通过外挂 SQL 数据库或 Redis 来支持 Tag。这样的设计也比较蹩脚，但我们先暂时不讨论 Tag 的问题。

文件内的时序数值存储时采用了多个预分配空间的环形结构。新的数据写入时会进入最高精度的例如 10s 的环形上，环写满了就覆盖旧数据。Whisper 也会在后台触发聚合高精度的例如 6 个 10s 的数值写到低精度例如 60s 的环上，来支撑更长时间跨度的查询，有一定写放大问题。

Whisper 查询支持 Unix Shell 风格的通配符，即类似于：

```
host.machine_[a-zA-Z0-9]*.memory?.
{percent_used,percent_free}.value
```

查询的实现也基本是一个带缓存的 glob 模块，查找到文件后读取文件内容进行函数运算后返回结果。

Whisper 数据库的设计相对简单，在大数据量下的问题也很明显：

1. 查询指标层级变多或量大时需要操作文件多，读取性能差
2. 每个活跃指标对应打开一个本地文件，IOPS 随读写线性递增
3. 更新指标数据是随机写，对磁盘性能要求高
4. 指标数据压缩能力差，读取和写入占用内存高，磁盘存储空间占用大
5. 临时和无效的指标难区分和清理

原 Prometheus V2 的 TSDB 的也有类似的问题，到 V3 时也选择了推倒重来 (参考 Writing a Time Series Database from Scratch)。

Graphite 存储开源方案一览

Graphite 项目自 2006 年开始，发展到如今对性能一直都不太关注。主要组件 Graphite-web、Carbon、Whisper 都是 Python 实现，也没有对存储引擎设计做出根本性改进。

针对 Whsiper 的性能问题，开源社区也给出了一些解决方案，主要分为两大类：

1. 基于 Whisper 改良，代表是 go-carbon 和 Kenshin
2. 迁移到通用列式存储，代表是 graphite-clickhouse 和 metrictank

对于 Whisper 的改良比较典型的有：

1. go-carbon 使用 Golang 对 Whisper 引擎的重写，获得了一部分语言性能的提升，但在存储访问模型上没有太大改进，没有解决根本问题

2. Kenshin 豆瓣出品，通过合并指标文件降低了数倍的 IOPS，单机支持指标序列增加到 300w，但在其他方面收益不明显

我们认为 Whisper 改良方案解决问题的范围有限，无法满足我们日益增长的性能需求。

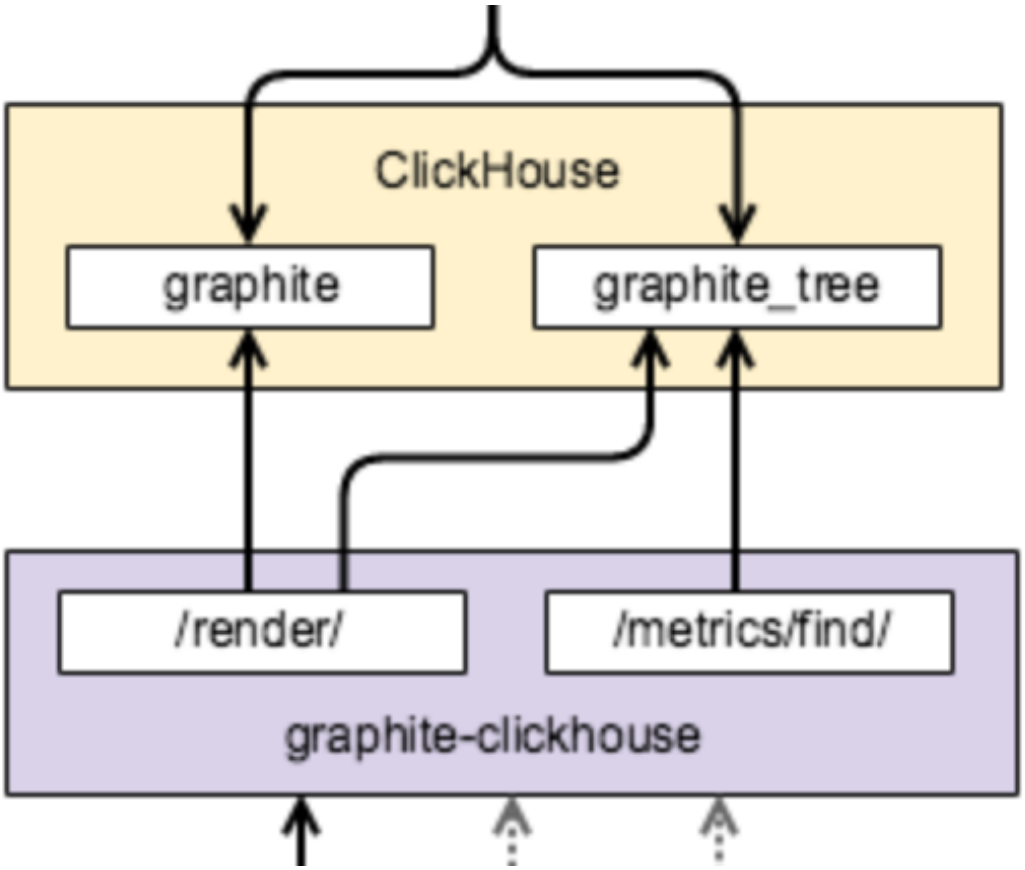
我们将目光聚焦到第二类方案上，时序指标一般按照固定的间隔写入，查询往往是指定一定时间范围的一个或多个时间序列，这种使用场景其实与列式存储非常契合。列式存储一般通过 LSM Tree 或类似方案提升写入性能，避免随机写。在存储时序数据时又能将单个序列的数据存储到同一列，可以更容易加持 Gorilla 或其他压缩算法以提升单列内的数据压缩率。

换言之，设计优良的通用列式存储其实可以解决大部分 Whsiper 存储设计带来的问题，包括graphite-clickhouse 和 metrictank 其实在解决存储性能的问题上已经做的挺棒了。但它们没有解决的问题是查询性能的问题，而高效的索引恰是提升查询性能的关键！

索引！索引！

我们暂时抛开存储引擎本身写入和读取的性能差异，着重介绍 graphite-clickhouse 和 metrictank 是怎么为 Graphite 指标构建查询索

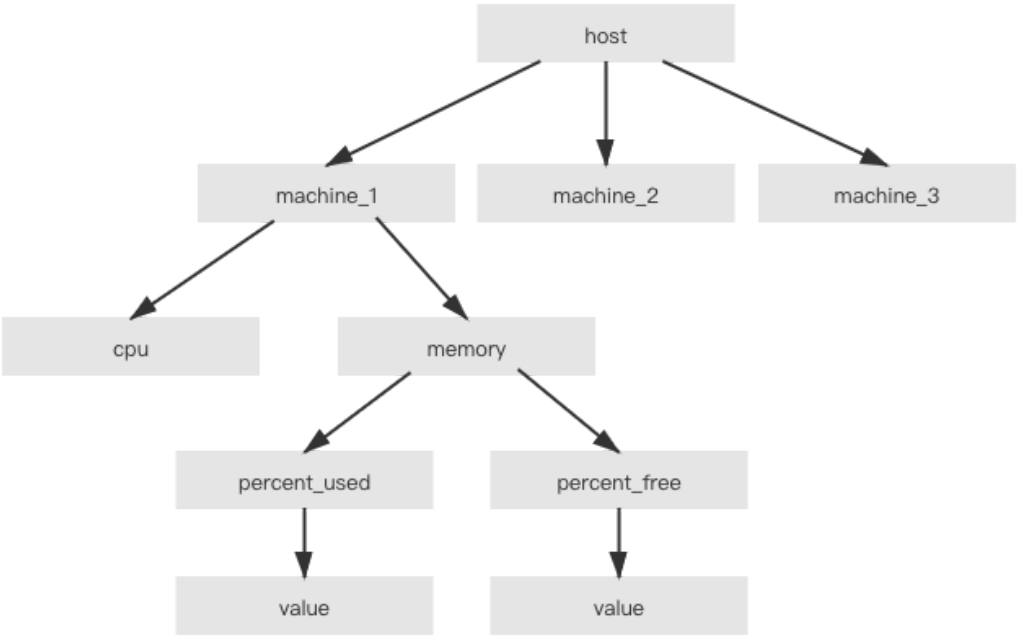
引的。



从 graphite-clickhouse 的架构图和代码看，它将指标名和数据点拆分成了两张表，每次查询都会在内部对 clickhouse 进行两次查询，一次根据 Target 表达式查找匹配的指标名，第二次根据指标名获取数据。

graphite-clickhouse 的问题是它在每次查找匹配的指标时都需要对 clickhouse 中的索引表进行完整正则匹配，导致其查询性能较差，这与我们的测试结果也基本相符。我们实际测试 graphite-clickhouse 对比 Whisper 写入 IOPS 能降低约 40 倍，查询性能较 Whisper 略提升 2-4 倍，对高并发查询支持较差。

metrictank 是 Grafana Labs 的开源方案，我们通过阅读代码知道它是在内存中构建索引，每次指标查询是先根据索引查询完整指标名，再根据指标名从缓存或存储读取数据。



metrictank 的内存索引是构建了一个前缀树，将 Graphite 指标以 . 分隔为多段插入到树的节点上。在执行查询时，按顺序从前往后在前缀树上查找能匹配的全部子节点并返回。

这种方式相对 graphite-clickhouse 暴力全表正则扫描看似更先进一些，但 ... 是这样吗?

metrictank 的前缀树保存了全量的指标名，会随着指标的写入量无限增长。如果拆按月或者天拆分索引子树，内存占用可能会翻多倍，同时子树间大部分节点相同，非常不经济。如果不拆，那可能有很多临时性的指标无法从索引树中清理掉，直到程序 OOM。同时这个索引树全局只有一个，当大量指标新增时也会有严重的锁冲突问题。

目前 metrictank 的准备了两个方案解决这个问题：

1. 定期删除不活跃的指标索引，效果类似于删历史数据了，会导致历史指标无法再被查询
2. 前缀树垂直拆分到集群各个节点，可以同时缓解容量和锁冲突的问题，但这些临时指标还是会拖慢查询的速度

在写本篇文章时，我们在 Graphite 中查询了部分在年初就已经回收的物理机的指标。这类不再持续的指标在 metrictank 中就只能被 GC 掉，否则留在前缀树中既会增加内存压力，也会影响查询速度。

我们认为更合理的索引行为是：对于出现过的历史指标，在查询到其出现的时间区间的时候能返回正确数据；查询范围不在其出现的时间区间时尽量不拖慢整体查询性能。

总体来说，我们认为 graphite-clickhouse 和 metrictank 在索引方案上有诸多妥协，效果不佳。

创新者 Uber M3

我们反复理解过 Graphite 的指标格式，是以点分割的多层级的标签。而不管是文件目录树还是内存中的前缀树，大家都倾向于使用树这种结构来构建 Graphite 指标的查询索引。

我们已知的真正在 Graphite 存储设计上首先打破这个观念的是 Uber 开源的 M3。

M3 是个非常有意思的项目，它把对 Graphite 和 Prometheus 的支持做到了一个项目中。它会尝试给 Graphite 指标的每一层生成一个 Label，例如将：

```
host.machine_name.memory.percent_used.value
```

转换为类似 Prometheus 指标格式：

```
{__g0__="host", __g1__="machine_name", __g2__="memory",
__g3__="percent_used", __g4__="value"}
```

所以？这个转换只是不就简单转了下存储格式吗？

不，这个设计更重要的意义是打破了 Graphite 指标的层级关系。将 Graphite 每段拆分为可以独立匹配的 Label，可以在每个 Label 上建倒排索引以加速查询。

例如我们有这样一组数据：

```
1: host.machine_1.memory.percent_used.value
2: host.machine_2.cpu.percent_used.value
```

```
3: host.machine_3.memory.percent_used.value
4: host.machine_4.memory.percent_free.value
```

倒排索引可以建成这样：

```
__g0__:host          -> 1,2,3,4
__g1__:machine_1     -> 1
__g1__:machine_2     -> 2
__g1__:machine_3     -> 3
__g1__:machine_4     -> 4
__g2__:memory        -> 1,3,4
__g2__:cpu           -> 2
__g3__:percent_used -> 1,2,3
__g3__:percent_free -> 4
__g4__:value         -> 1,2,3,4
```

我们在查询 `host.machine_*.memory.percent_free.value` 时可以按 Label 值的个数从小到大开始匹配：

```
查找 __g0__:host 得到                              -> [1,2,3,4]
在 [1,2,3,4] 中查找 __g4__:value 得到              -> [1,2,3,4]
在 [1,2,3,4] 中查找 __g2__:memory 得到             -> [1,3,4]
在 [1,3,4] 中查找 __g3__:percent_free 得到         -> [4]
在 [4] 中进行正则匹配 __g1__:machine_* 得到         -> [4]
```

上面例子中大部分的 Label 查询都能直接命中索引，同时进行了一次正则匹配就查找到了正确的序列，整体查询开销非常低。如果是前缀树则在每一步都需要进行字符串比较，同时需要在第二级查找时进行四次正则匹配。复杂的正则匹配的性能在指标量大将对查询性能造成的损害是巨大的，而这在前缀树中几乎是不可优化的。

设计优良的倒排索引可以大幅度降低匹配的指标量，从而降低整体的查询开销。更更重要的是倒排索引无须一次全量加载到内存中使用，每个索引项可以被独立存储和查询，也可以根据时间来进行切片实现按需加载索引。

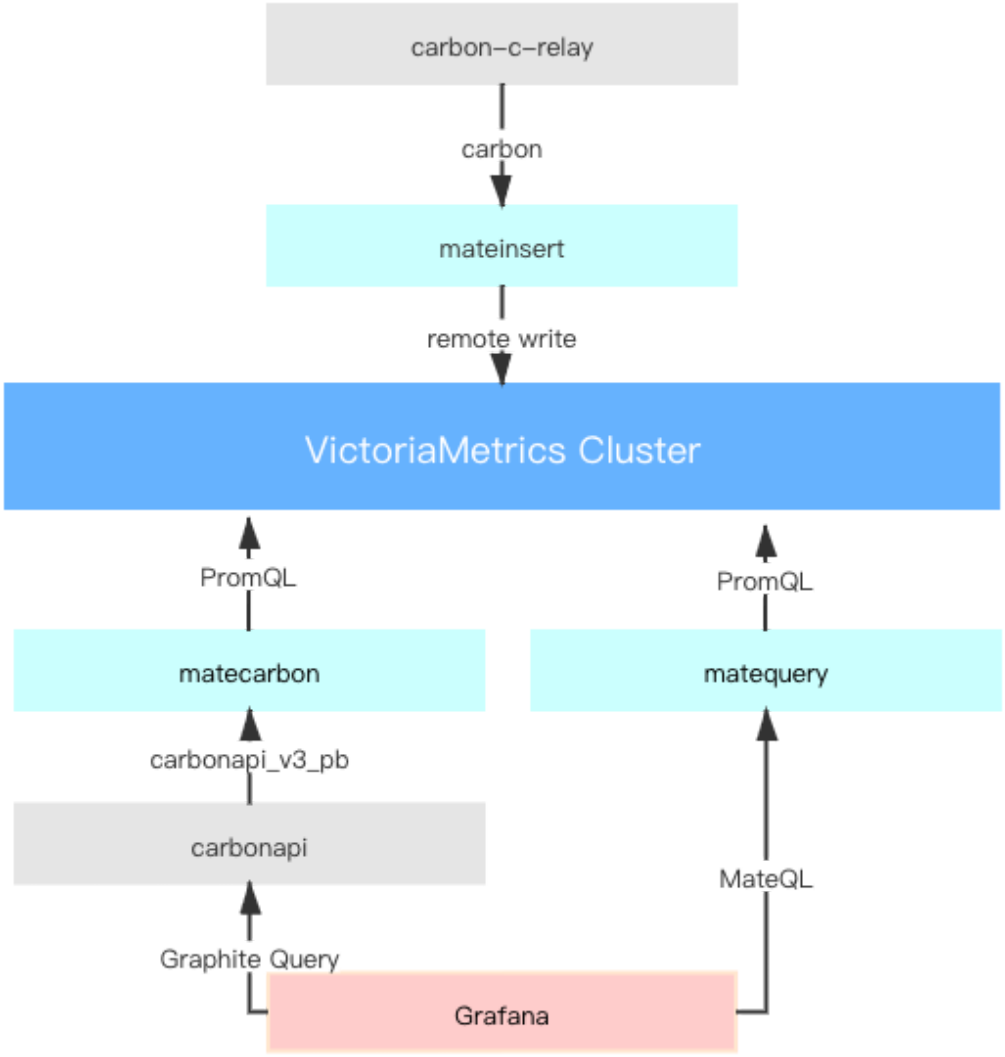
比如我们当前指标总索引大小在压缩后已经达到 500G 了，在其他方案下的存储和使用就会非常困难。而倒排索引可以将历史和临时指标给系统带来的性能负面影响降到最低，这对我们来说意义重大。M3 成功地将倒排索引引入 Graphite 指标体系，在各方面都取得了比较好的均衡，我们非常喜欢这个设计！

遗憾的是 M3 的架构里存储引擎与 Graphite 查询引擎强绑定，而他们自己实现的 Graphite 查询函数较少，无法满足我们全量业务透明迁移的需求。同时我们也期望方案内各组件的可替换性更强一点，未来可以在选型上给我们提供更多的可能性。

所以我们选择带着 M3 的倒排索引思路继续出发，去支持更完整的 Graphite 查询函数，同时拥抱更广阔的 Prometheus 社区。

Graphite On VictoriaMetrics

是我们最终给出的完整方案：



指标在经过 carbon-c-relay 组件后就是纯粹 carbon 格式的时序数据了，格式是：

```
host.machine_name.memory.percent_used.value 80 1600969976
```

这里，我们聚焦后面时序数据写入和查询的部分，主要包含这几个组件：

1. 计算引擎 carbonapi
2. 存储引擎 VictoriaMetrics
3. 自研组件 mateinsert、matecarbon、matequery

carbonapi 是开源社区用 Golang 重新实现的 Graphite 函数计算引擎，目标是完全兼容和替换 Graphite-web，当前支持了绝大部分的

Graphite 查询函数，同时与 Graphite-web 相比也有接近 10 倍的性能提升。没有经过更多复杂的抉择，我们选择了 carbonapi 来作为我们新方案的计算引擎。

VictoriaMetrics

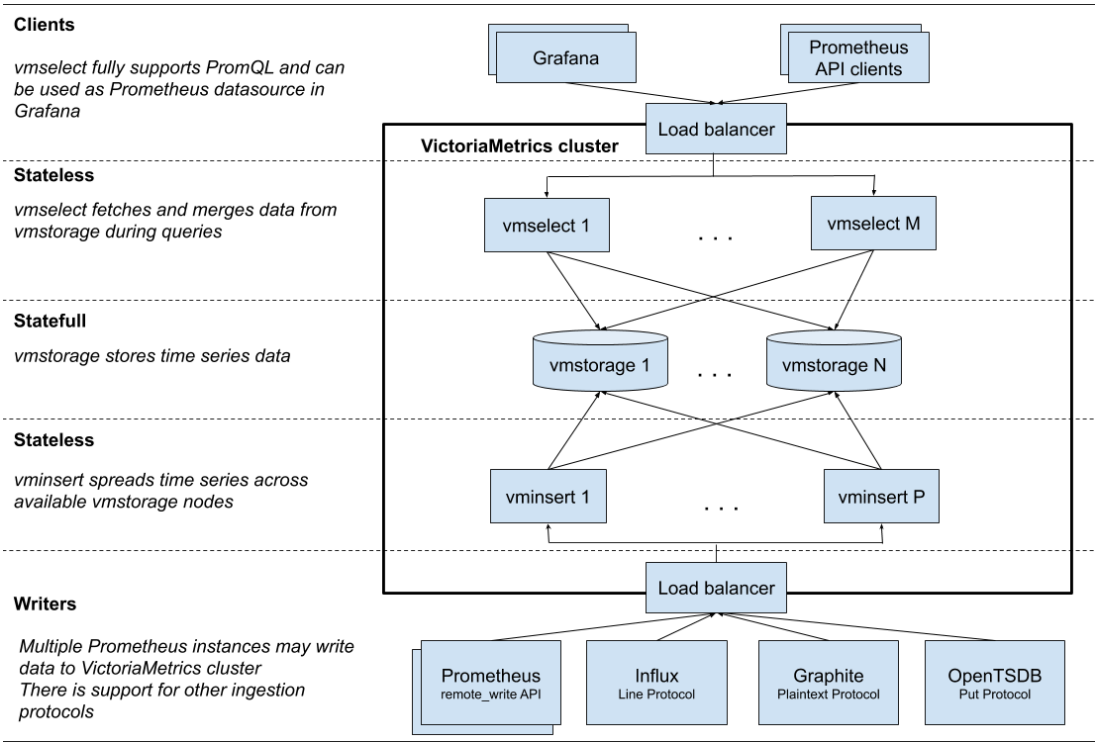
VictoriaMetrics 是 Prometheus 社区中一个活跃而且优秀的远端存储，在各项性能测试中都名列前茅。VictoriaMetrics 的主要贡献者是 @valyala，他也是 fasthttp 的作者，他非常擅长写出高性能的 Golang 程序。他也有丰富的 Clickhouse 项目经验，Clickhouse 的 chproxy 负载均衡器也是他的作品。

VictoriaMetrics 从 Clickhouse 汲取了一些设计灵感，设计了 MergeSet 结构来存储指标数据和索引，也在其他方面针对 TSDB 做了很多的优化，例如比官方 RoaringBitmap 更高效的 Set 设计、引入 zstd 算法提升 10 倍数据压缩率等等。

MergeSet 的思路跟 LSM Tree 非常类似，先在内存中排序写入的数据并定期刷入磁盘中，再后台合并磁盘上的文件以减少文件数量。较大的差异在于 VictoriaMetrics 会将 Index、Timestamp、Value 分列分文件存储，这样可以在扫描索引时加快速度，也因为数据类型一致可以提升单列数据的压缩率，而且没有 WAL 节约了一部分磁盘开销。

VictoriaMetrics 的性能出众的原因还在于所有可复用的对象都做了池化，非常多的函数调用都分别有针对性地优化了 fast path 和 slow path，大量运用缓存的同时也严格限制内存上限等等，更多细节可以观看 Go optimisations in VictoriaMetrics，我们也非常喜欢 VictoriaMetrics 代码中简洁优雅的实现。

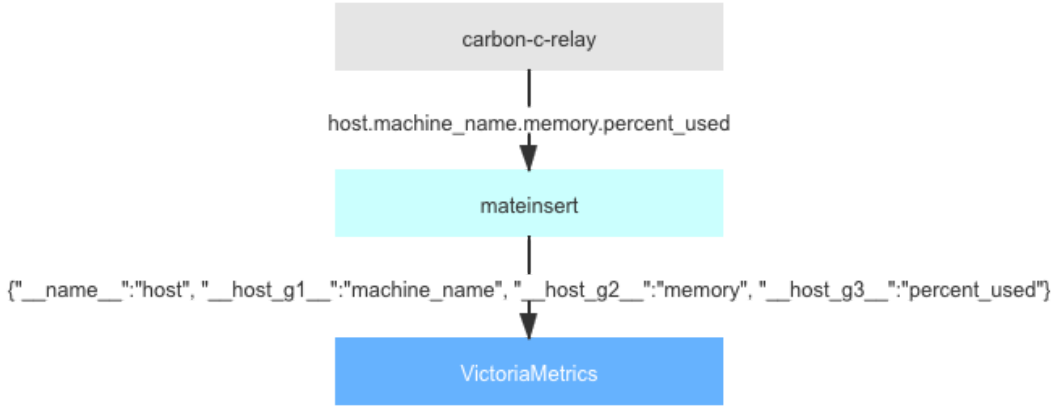
同时 VictoriaMetrics 的集群架构也非常简单，写入 vminsert、存储 vmstorage 、查询 vmselect 组件间无需共享任何信息，集群的维护、扩缩容都相对简单。



这些都是我们选择 VictoriaMetrics 的理由，让我们给 carbonapi 与 VictoriaMetrics 继续注入灵魂，让它们能协同工作起来。

mateinsert

我们通过 mateinsert 组件在写入路径上作出如下转换：



我们在大致思路上与 M3 保持一致，但同时多做了一点微小的改进：

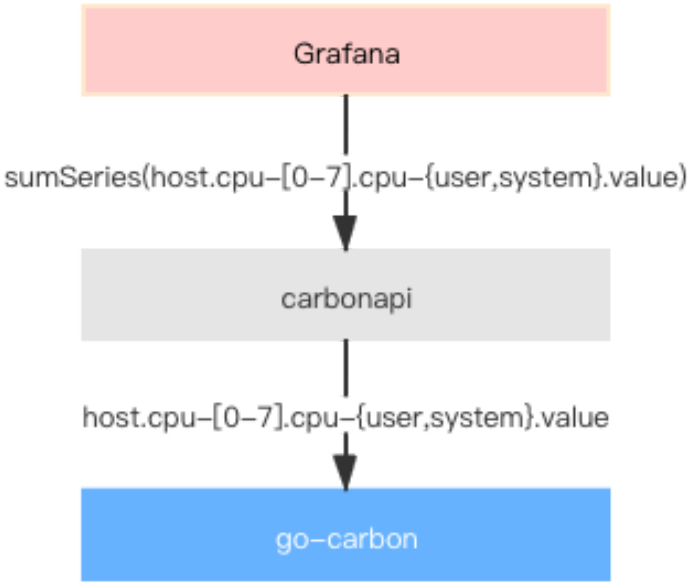
1. 将 Graphite 指标第一段记为 Prometheus 指标名，即 **name** Label
2. 将 Graphite 指标第一段插入到其他 Label 名称中

我们会要求每个业务使用自己的 App 名作为指标名前缀，所以上述这两个改动有这些好处：

1. 可以通过 TSDB Stats API 直接观测每个 App 的指标量，方便治理
2. 每个业务的 Label 都有命名空间前缀，当业务指标单 Label 维度过高时只会影响自己指标的查询，隔离了对其他业务的影响

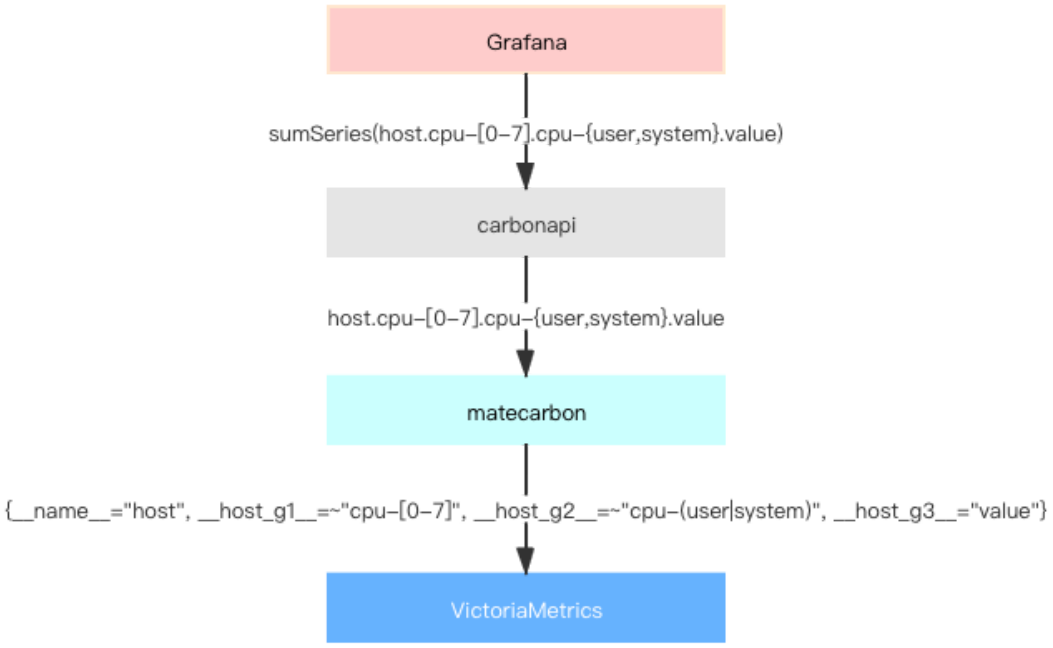
matecarbon

在我们之前的 Whisper 架构下，一次完整的查询如下图：



原 go-carbon 是支持 Graphite Target 查询表达式的。但我们已经将 Graphite 指标存储到 VictoriaMetrics 中了，查询时 VictoriaMetrics 不认识 Graphite Target 表达式。

所以我们新增了一层 matecabon 组件，在这一层可以解析 Graphite Target 的每一段并生成对应的 PromQL 表达式，并将查询提交给 VictoriaMetrics。VictoriaMetrics 返回查询数据后，再处理格式并返回给 carbonapi。



到这里其实就已经完成 Graphite 数据到 VictoriaMetrics 数据的写入和查询，carbonapi 和业务方都无需感知转换，可以透明的完成迁移。

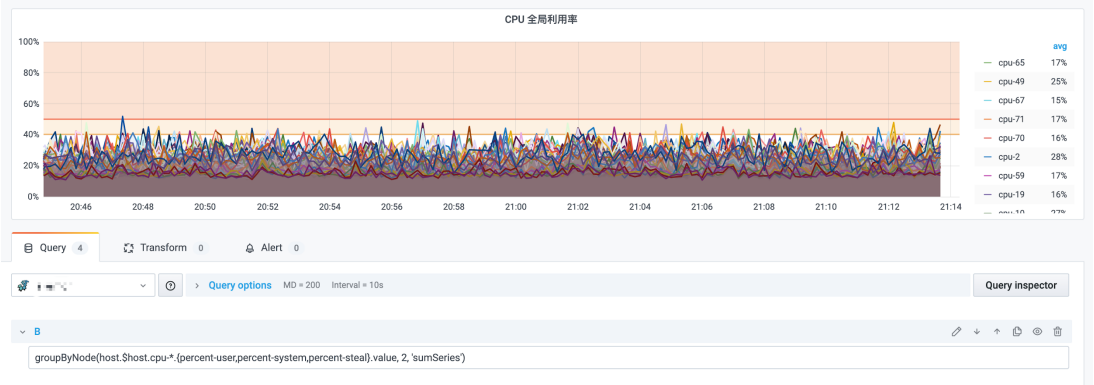
matequery

MateQL 是我们对 Graphite 与 Prometheus 融合的一次尝试，我们通过修改 PromQL 的 Parser 额外支持了 Graphite Target 的查询。

同一份 Grapite 指标，我们给业务方同时提供 Graphite 以及 MateQL 两种查询方式，可以让业务方无需任何成本地使用 PromQL 语法来处理自己的 Graphite 指标。

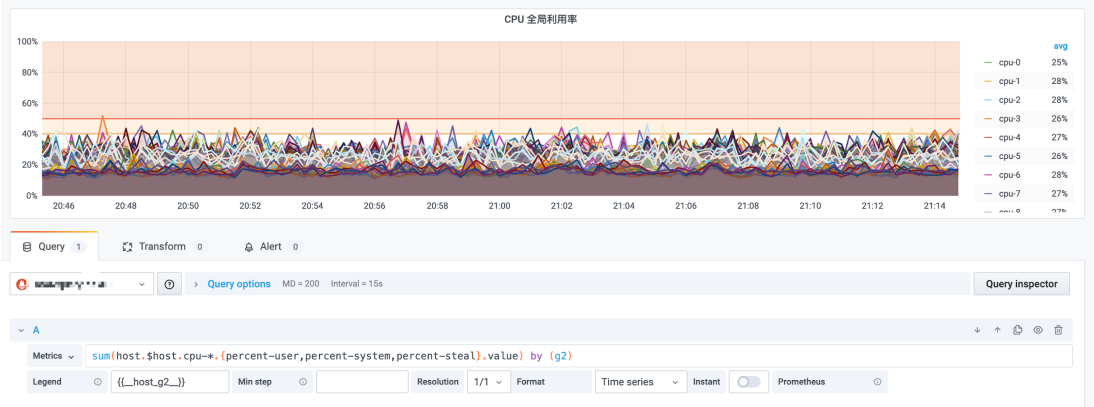
以查询物理机 CPU 利用率指标为例，Gaphite 查询语句是：

```
groupByNode(host.machine_name.cpu-*.{percent-user,percent-system,percent-steal}.value, 2, 'sumSeries')
```

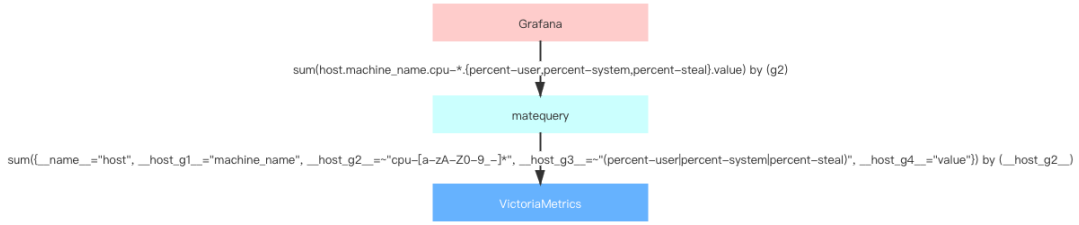


等价替换到 MateQL：

```
sum(host.machine_name.cpu-*.{percent-user,percent-system,percent-steal}.value) by (g2)
```



MateQL 查询会经过 matequery 组件转换处理，转换逻辑可以跟上述的 matecarbon 复用。



我们测试查询半个月内指标序列超过 10w、包含数据点超过 2 亿个时，MateQL 计算时序累计值可以在 1s 左右返回，速度大约是经过 carbonapi 速度的 60 倍。

对比上面的 Graphite 兼容查询，MateQL 的查询主要都在 VictoriaMetrics 集群内完成，少了到 carbonapi 的指标序列化、传输、反序列化和再计算的过程，在处理大规模指标时性能优势会比较明显，同时也可以在 PromQL 计算引擎的持续优化中受益。

我们推荐指标量大或复杂分析场景的业务开始尝鲜 MateQL，更多的玩法我们也还在探索中。

Graphite 未来

我们在设计出 MateQL 的时候眼中的也一阵恍惚，什么是 Graphite 什么是 Prometheus，他们的边界在哪里？Graphite 的灵魂是 Whisper 是 Gaphite-web 还是 Statsd ？

首先排除 Whisper，再剩下的选项中 Gaphite-web 和 Statsd 似乎都有道理。

Graphite-web 代表的查询引擎是可以被平滑替换的，我们可以自由拓展更多的查询语法，或者实现 dogstatsd functions，或者支持 PromQL 就像我们在 MateQL 中实现的那样。只要查询语言的易学易用、表达能力充分，我们的观念是可以保持开放、博采众长。

Statsd 代表的是指标采集模块的预聚合和推模型。推拉模型各有优劣，总体来说推模型对使用方要求更低，拉模型可以提升数据可靠性，在云原生设施和业务场景中的有些取舍也不一样。Google Monarch 监控系统已经抛弃了 BorgMon 的拉模型改为推模型，Google 在论文中说：

> Push-based data collection improves system robustness while simplifying system architecture. Early versions of Monarch discovered monitored entities and “pulled” monitoring data by querying the monitored entity. This required setting up discovery services and proxies, complicating system architecture and negatively impacting overall scalability. Push-based collection, where entities simply send their data to Monarch, eliminates these dependencies.

整个指标系统中的每个组件都可以权衡和抉择，我们在排列组合中寻找最优解，而最后组合的结果是不是符合谁定义的 Graphite 其实已经不重要了。我们在时刻关注着业界的发展，也在不断反思关于指标系统的最佳实践，未来有机会我们也会持续跟大家分享我们最新的进展和想法，

非常感谢 Uber M3、VictoriaMetrics、carbonapi 项目对我们的帮助，我们也积极参与了开源社区建设。我们给 carbonapi 和 VictoriaMetrics 贡献了十余个 PR，自己实现的 mateinsert、matecarbon、matequery 组件也已开源在 zhihu/promate，欢迎大家关注~