LinkedIn 是如何将 Hadoop YARN 集群扩展到超过一万个节点

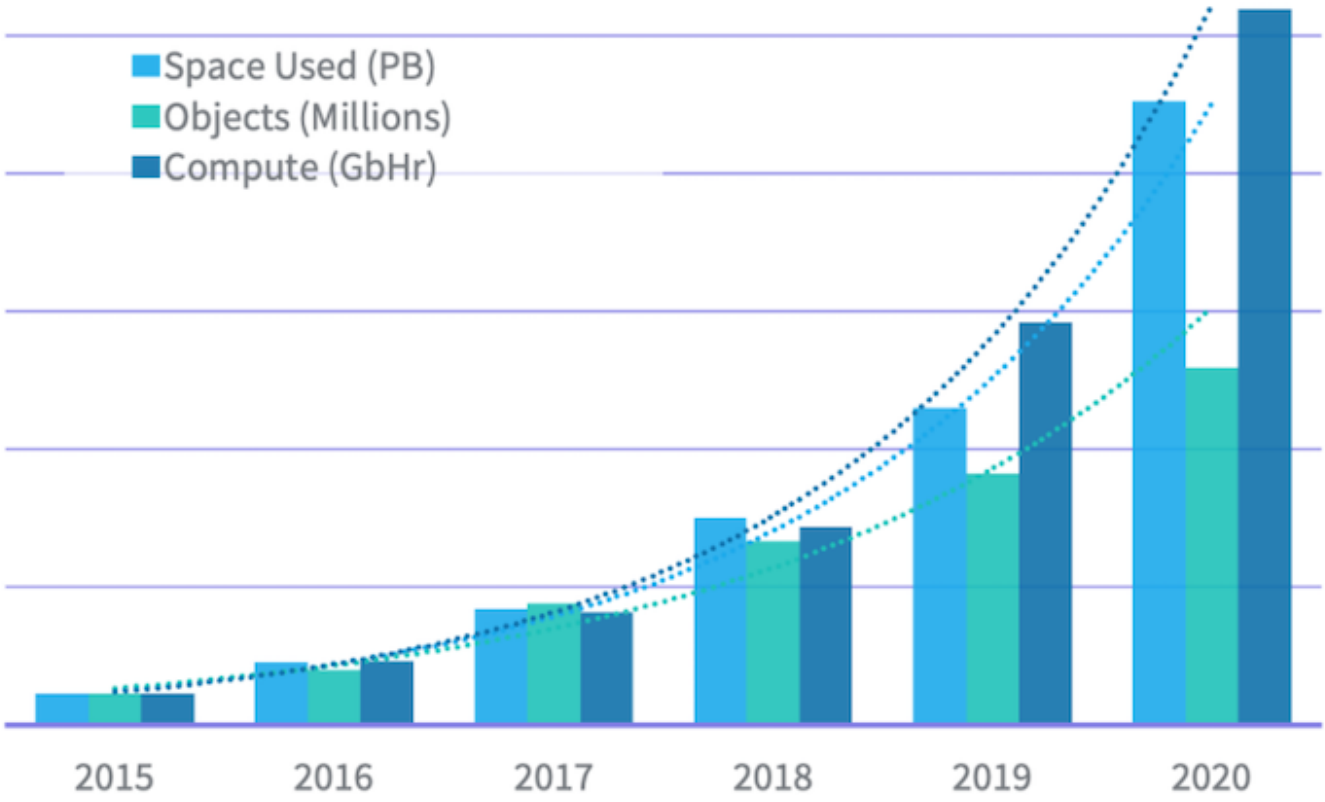
在 LinkedIn，我们使用 Hadoop 作为大数据分析和机器学习的基础组件。随着数据量呈指数级增长，并且公司在机器学习和数据科学方面进行了大量投资，我们的集群规模每年都在翻倍，以匹配计算工作负载的增长。我们最大的集群现在有大约 10,000 个节点，是全球最大（如果不是最大的）Hadoop 集群之一。多年来，扩展 Hadoop YARN 已成为我们基础设施最具挑战性的任务之一。

在这篇博文中，我们将首先讨论在集群接近 10,000 个节点时观察到的 YARN 集群性能很慢以及我们为这个问题开发的修复程序。然后，我们将分享我们主动监控未来性能下降的方法，包括我们编写的一个名为 DynoYARN 的开源工具，它可以可靠地预测任意大小的 YARN 集群性能。最后，我们将介绍内部称为 Robin 的服务，使我们能够将集群水平扩展到 10,000 个以上的节点。

当 YARN 集群开始出现问题

与 Hadoop 分布式文件系统 (HDFS) 的 NameNode 相比，所有文件系统元数据都存储在一台机器上，YARN 资源管理器则是相当轻量级的，它只维护少量元数据。因此，我们早先时候遇到了 HDFS 的可扩展性问题，并且我们在 2016 年开始解决 HDFS 扩展行问题。相比之下，Hadoop 中的计算组件 YARN 一直是基础设施中一个非常稳定的组件。

我们的集群规模每年都在翻倍，我们知道总有有一天 YARN 的可扩展性将成为一个问题，因为资源管理器的单线程调度机制无法维持集群的无限增长。尽管如此，我们从未真正投入了解 YARN 的扩展限制，并假设它可能会在下一个新技术出现之前起作用，直到 2019 年初我们的 YARN 集群开始出现了扩展性问题。



如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：过往记忆大数据

过去，我们在一个数据中心构建了两个 Hadoop 集群：主集群服务于主流量业务，存储和计算都是绑定在一起的，而为其他业务构建的辅助集群存储是绑定的，而计算资源是利用闲置的。为了提高资源利用率，我们将辅助 Hadoop 集群的计算节点合并到主 Hadoop
集群，并作为一个单独的分区。

不幸的是，大约两个月后，集群开始出现问题。

现象

计算节点合并后，集群有两个分区，分别有约 4,000 和约 2,000
个节点（我们称它们为“主要”和“次要”）。很快，Hadoop
用户在提交作业之前遇到了长达数小时的延迟；然而，集群中有丰富的可用资源。

在寻找延迟的原因时，我们最初认为 Hadoop YARN 中处理软件分区（software partitioning）的逻辑有问题，但经过调试和调查后，我们没有发现那段代码有任何问题。我们还怀疑将集群的大小增加 50% 会使资源管理器过载，导致调度程序无法跟上。

我们仔细查看了队列的
AggregatedContainerAllocation，它表示容器分配速度。合并前，主集群的平均吞吐量为每秒 500 个 containers，辅助集群的平均吞吐量为每秒 250 个
containers；合并后，AggregatedContainerAllocation 约为每秒 600 个

containers，但分配速度也经常在较长时间（数小时）内下降至每秒 50 个 containers。

我们做了几轮分析，发现一些代价高昂的操作，比如 DNS 操作，其标记了 @synchronized 注释，这限制了并行性。将这些操作移出同步块后，我们观察到吞吐量提高了约 10%，但延迟对用户来说仍然很明显。

通过重新定义公平来减轻压力

在解析 resource manager 的审计日志后，我们注意到调度器经常在切换到其他队列之前将 containers 调度在一个队列中很长一段时间。即使在性能合理的时期（每秒 600 个 containers），一些队列中的用户也经历了数小时的延迟，而其他队列中的用户几乎没有经历过延迟。一些队列的 containers 分配速度是正常的，但对于其他队列已经下降到几乎为零。这一观察使我们重新审视了调度程序如何决定优先调度哪个队列的 containers。在 LinkedIn，我们使用 Capacity Scheduler，它根据利用率对队列进行排序，并首先将 containers 分配给利用率最低的队列。

假设我们有两个队列 A 和 B，如果 A 的利用率为 10%，B 的利用率为 20%，那么调度器将首先为队列 A 调度 containers，然后再移动到 B 队列，并为提供其服务。这在大多数情况下是有效；但是，在容器流失率高的环境中可能会出现短暂的死锁。假设队列 B 中的大多数正在运行的作业都是相对较长的作业，而队列 A 中运行的作业是非常短命的作业。由于 A 的利用率仅为 10%，因此将在队列 A 中调度 containers 而不是队列 B。由于队列 A 中的 containers 流失率远高于队列 B，当调度程序完成队列 A 中的一次调度工作负载迭代时，队列 A 的利用率可能保持不变甚至下降，但仍远低于队列 B，例如 9.5%，而队列 B 的利用率略有下降至 19%。在队列利用率收敛并且队列 A 的利用率超过队列 B 之前，调度程序不会接收提交到队列 B 的工作负载，但由于队列工作负载的不同特征，这可能需要几个小时。从观察者的角度来看，调度程序似乎在调度队列 A 中的工作负载时卡住了，而队列 B 中的工作负载却缺乏资源。

我们问自己为什么这只在合并两个集群后才成为问题，并意识到主分区队列中的工作负载主要由 AI 实验和数据分析组成，这些工作实现为更长时间运行的 Spark 作业，而辅助分区队列中的工作负载主要是快速运行的 MapReduce 作业。如果 resource manager 可以任意快速地调度 containers，这将不是问题；然而，由于集群合并显着降低了调度速度，分配公平性问题浮出水面。

我们提出的缓解方法是，当调度程序分配 containers 时，以相等的概率挑选队列；换句话说，我们随机选择队列而不是基于利用率。瞧！我们的问题暂时得到缓解。我们后来将补丁贡献给了 Apache Hadoop。

效率低下的根本原因

尽管缓解了队列公平性，但仍然没有解决调度缓慢的根本原因。 我们知道我们的 YARN 集群中仍然存在迫在眉睫的扩展性问题。我们不得不深入挖掘！

回顾合并前后的总调度吞吐量，在最好的情况下，我们达到了 80% 的性能（每秒约 600 个 containers 与每秒 750 个 containers）；在最坏的情况下，我们的性能仅为 7%（每秒约 50 个容器与每秒 750 个容器）。 这种差距直观地引导我们重新审视分区的调度逻辑，在那里我们发

现了引起我们注意的不规则性。

默认情况下，YARN
资源管理器使用同步调度，即节点心跳到资源管理器，这会触发调度器在该节点上调度未完成的 container。 未完成的 container 被提交到主分区或从分区，如果 container
的分区和节点的分区不匹配，则不会在该节点上分配 container。

在队列中调度应用程序时，调度程序以先进先出 (FIFO)
的顺序遍历它们。现在假设主分区中的一个节点向资源管理器发送心跳；调度程序选择队列 A 进行调度，队列 A 中的前 100
个未完成的应用程序正在从辅助分区请求资源。我们发现调度程序仍然试图将这些应用程序中的 container 匹配到该节点，尽管匹配总是失败。由于两个分区都处理了大量的工作负载，这就在每个心跳上产生了巨大的开销，从而导致了速度的下降。

为了解决这个问题，我们优化了逻辑，如果主（或次）分区的节点向资源管理器心跳，调度器在调度时只考虑提交到主（或次）分区的应用程序。更改后，我们观察到合并前后总平均吞吐量的大致相同，并且在两个分区都在最坏的情况下也提高了 9 倍！我们将这个修复也贡献给社区了。

可衡量才可以被修复

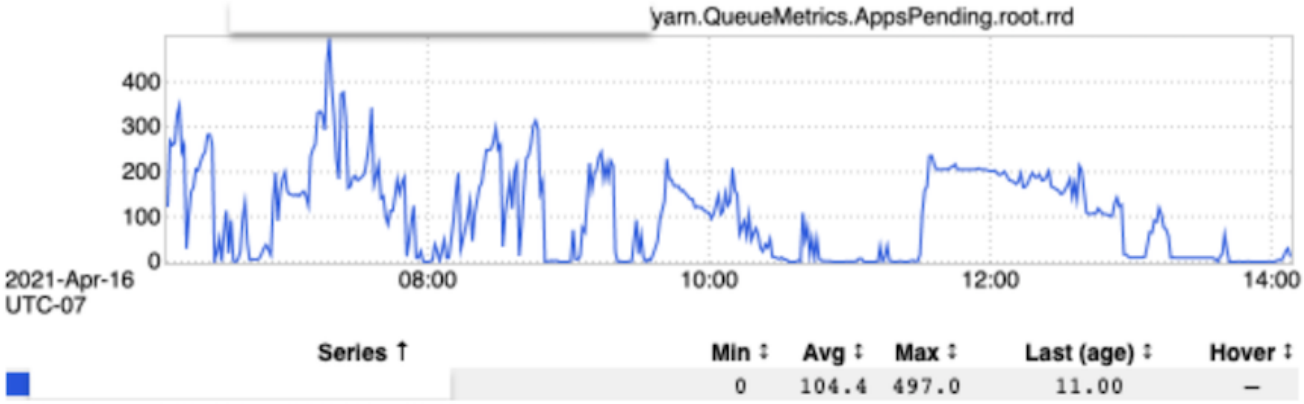
为了应对 YARN 可伸缩性的挑战，我们遵循了我们以前工程主管 David Henke
的智慧，“可衡量才可以被修复（What gets measured gets fixed）。”
紧接着的下一步是构建度量和工具来度量和监控可扩展性。

在我们的集群达到今天的规模之前，用户遇到的任何慢作业的问题都可以用用户队列中的资源不足来解释——这个问题只会影响在该队列中运行的团队。为了找到任何影响性能的根本原因，我们可以简单地找出哪些工作负载消耗了该队列中不成比例的资源，并要求他们调整工作流程。

然而，我们的集群最终达到了资源管理器级可伸缩性问题导致用户作业速度变慢的地步。因此，我们需要 (1) 一种衡量和反映资源管理器变慢的方法，以及（2）随着我们继续扩大集群规模和工作负载，一种预测未来资源管理器性能的工具。

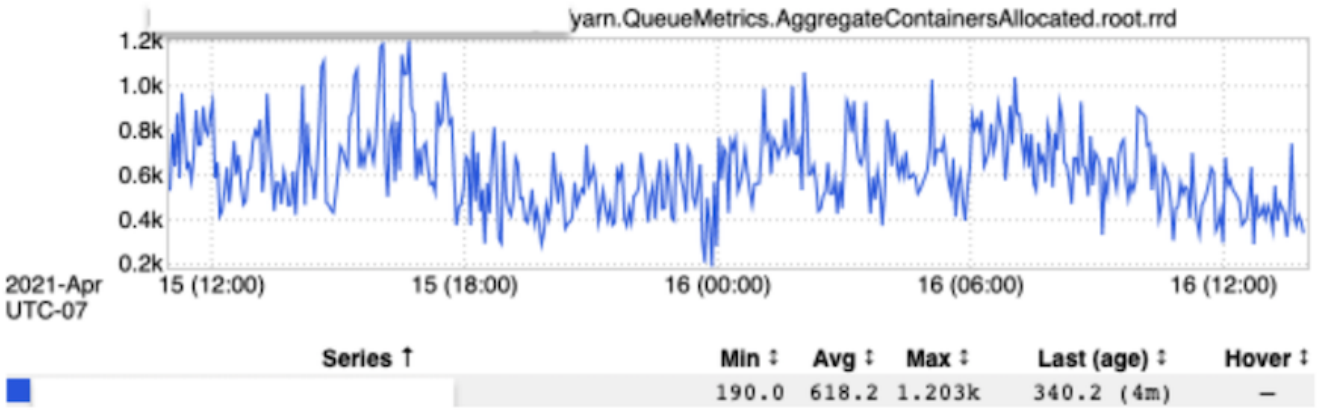
设置可扩展性指标和警报

我们能够利用现有的资源管理器指标来衡量性能问题。最相关的是：
1) 待处理的应用程序：



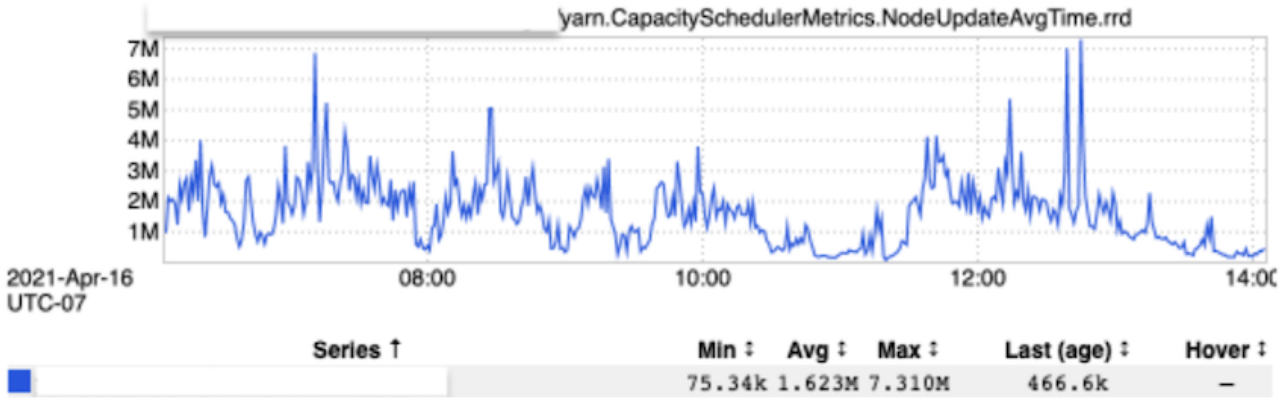
如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：过往记忆大数据

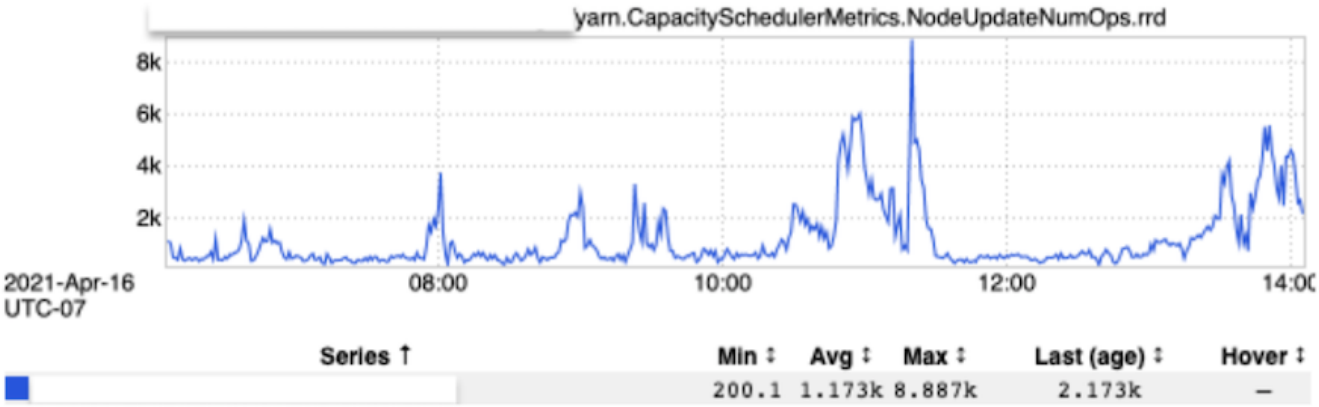
2）容器分配吞吐量（AggregateContainersAllocated ）：



如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：过往记忆大数据

3）NodeManager 心跳处理速率:





如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：过往记忆大数据

待处理的应用程序指标让全面了解用户所看到的性能。有许多应用程序待处理意味着队列已满，许多用户的应用程序还没有运行。

在资源管理器方面，容器分配吞吐量指标告诉我们资源管理器是否不能足够快地调度容器；长时间（例如 30 分钟）持续的低吞吐量表明可能出现问题。然而，单独的低容器分配吞吐量并不表示资源管理器性能问题。例如，如果集群被充分利用并且容器流失率低，我们可能会看到低吞吐量，但这是因为集群资源不够。

在 capacity scheduler 中，我们在 NodeManager 发送心跳的时候分配 container，也就是 yarn.scheduler.capacity.per-node-heartbeat.multiple-assignments-enabled 为 true。NodeManager 心跳处理速率指标告诉我们这个关键代码路径是否有任何缓慢。例如，在一次事件中，我们注意到资源管理器在上线新功能后花费了更多的 CPU 周期。使用此指标帮助我们确定该功能对节点心跳处理逻辑进行了更改，优化此代码路径后，节点更新吞吐量大幅增加，资源管理器 CPU 使用率恢复到以前的水平。

DynoYARN 用于解决 YARN 可扩展性问题

评估 YARN 可扩展性的另一个重要目标是能够预测未来 resource manager 的性能。虽然我们从历史容量分析中知道我们的工作负载和集群规模同比增长了 2 倍，但我们不知道 resource manager 将如何应对这种增长，也不知道在什么时候 resource manager 的性能将不再能够维持这些增长。与我们编写的用于评估未来 HDFS NameNode 性能的规模测试工具 Dynamometer 类似，我们编写了 DynoYARN，一种用于启动任意大小的模拟 YARN 集群，然后在这些集群上运行任意工作负载的工具。

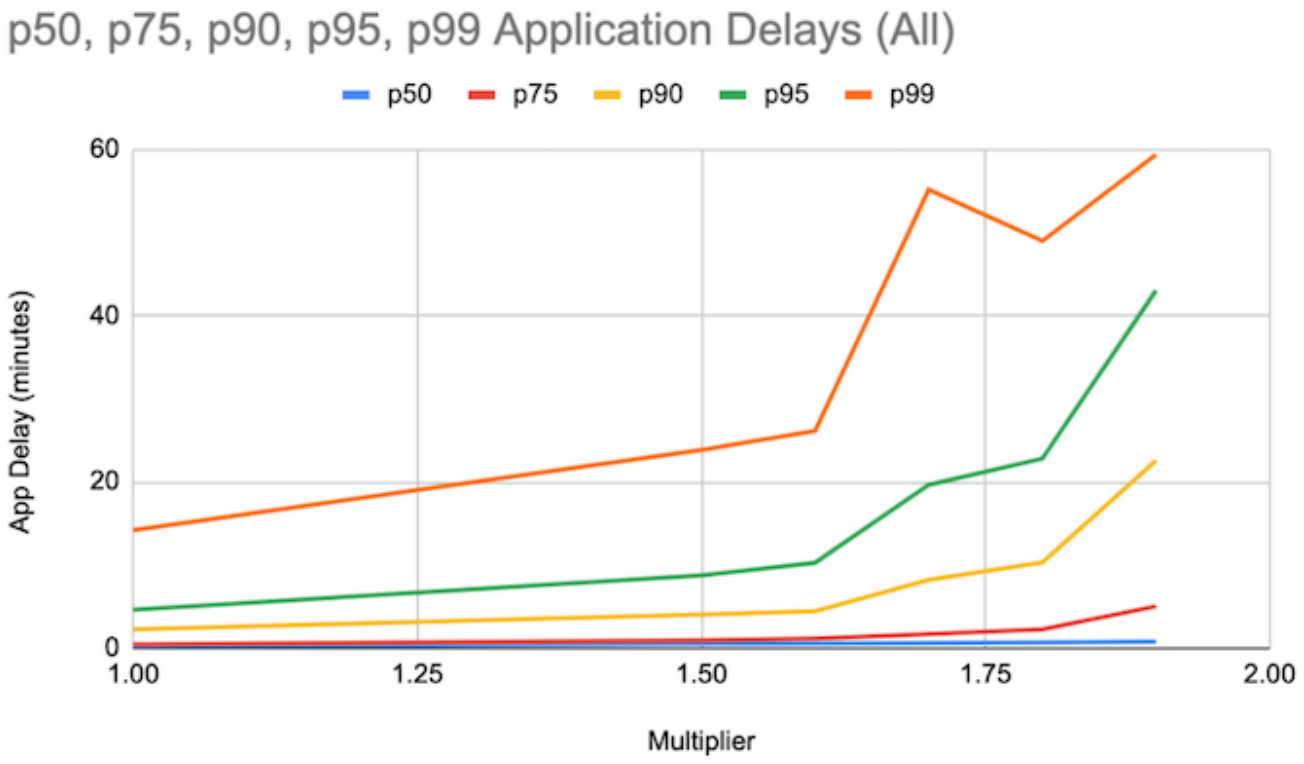
DynoYARN 由两个组件组成：一个用于启动模拟 YARN 集群的“driver”，以及一个用于在该集群上运行模拟工作负载的“workload”。两者都是作为 YARN 应用程序实现的；我们实际上是在一个 YARN 集群中运行一个 YARN 集群，但是资源约束要低得多。例如，要启动一个1200个节点模拟集群，驱动程序将分配一个 container 来运行模拟的 resource manager，并分配 container 来运行模拟的 node

managers。后一种 container 可以运行多个模拟的 node managers
。在我们的设置中，我们可以在单个50GB container 中运行30个模拟 node
managers。因此，在256GB主机上，我们可以运行5个容器，每个容器带有30个模拟 node
managers，或者在每个真实的256GB主机上运行150个模拟 node
managers。因此，模拟的1200节点集群只需要1200/150 = 8台真实主机。

为了评估 resource manager 的性能，工作负载（workload）应用程序解析来自生产集群的审计日志，并将它们提供给驱动程序（driver）应用程序的模拟集群。在模拟的 resource manager 上如实地重跑生产工作负载。从审计日志中，我们提取每个应用程序请求的容器数量、每个容器的内存/vcore需求、应用程序提交的时间，以及其他元数据，如哪个用户提交了应用程序（以模拟用户限制约束）和应用程序提交到哪个队列。结果模拟的性能非常接近我们在生产中看到的性能，因为工作负载几乎是完全相同的。

为了预测未来的可扩展性，我们在工作负载应用程序中实施了一项功能，允许我们修改解析的审计日志以模拟预计的工作负载。
例如，我们经常模拟的一个用例是将生产工作负载"乘以"一个固定数字，例如 1.5 倍或 2 倍。 在 1.5x 的情况下，每个申请有 50% 的机会被提交两次； 在 2x 的情况下，每个申请都有 100% 的机会被提交两次。使用这种策略，我们保留了生产中的优先级高的工作负载模式（例如，Spark 应用程序的比例、长时间运行与短期运行的应用程序的比例等），同时将它们扩展以预测未来的性能。

通过在许多细粒度乘数（例如，1.5x、1.6x、1.7x、1.8x、1.9x、2x）上重新运行模拟，我们可以得到 resource manager
的性能如何随着我们逐步扩大生产集群而变化的准确趋势。以下是这些模拟的结果:



如果想及时了

乘数	Node Manager 的数量	每天 Applications 的数量	p95 application delay (minutes)
1	7152	237472	4.633
1.5	10728	354600	8.769
1.6	11443	377962	10.278
1.7	12158	401440	19.653
1.8	12873	424540	22.815
1.9	13588	441090	43.029

可扩展性结果

我们的目标是将 p95 应用程序延迟保持在 10 分钟或以下。根据我们的模拟，我们发现 11,000 个节点的集群可以将应用程序延迟大致保持在 10 分钟的时间窗口内（11,443 个节点的集群为我们提供 10.278 分钟的延迟，仅略高于我们的 10 分钟目标），但 12,000 个节点的集群应用程序延迟为 19.653 分钟，远远超出我们的 SLA。

根据这一预测，我们推断（基于我们 2 倍的同比增长）何时达到这一里程碑，因此我们有多少时间来处理因扩展而开始遇到严重的资源管理器性能问题。

DynoYARN 开源

除了确定未来的扩展性能外，在 LinkedIn，我们还使用 DynoYARN 来评估大型功能在将它们推向生产之前的影响，并确保将我们的集群升级到更高的社区版本时的性能情况。例如，当我们将 Hadoop 集群从 Hadoop 2.7 升级到 2.10 时，我们使用 DynoYARN 来比较资源管理器的性能。我们还使用此工具对前面讨论的资源管理器优化进行了 A/B 测试。它是我们确定 YARN 性能路线图以及自信地推出大型资源管理器功能和升级的有用工具。我们认为 YARN 社区也可以从中受益，因此我们很高兴地宣布，我们正在 GitHub 上开源 DynoYARN。 该 repo 可在 https://github.com/linkedin/dynoyarn 上获得。 欢迎评论和贡献！

使用 Robin 实现水平扩展

虽然我们能够快速推出多项优化以缓解我们在资源管理器中发现的瓶颈，但很明显单个 YARN 集群很快将不再能够维持 LinkedIn 当前的计算增长（主要是由于单线程调度程序）。因此，我们踏上了寻找未来几年可以依赖的长期解决方案的旅程。

我们首先评估了 Hadoop 开源社区的两个潜在解决方案，即 Global Scheduling 和 YARN Federation。

Global Scheduling 的主要目标是解决默认心跳驱动调度程序（heartbeat-driven scheduler）无法满足的复杂资源放置要求。 它还引入了多线程调度，结合乐观并发控制，提高集群整体调度吞吐量。但是，我们在生产跟踪中没有观察到 DynoYARN 模拟中默认单线程调度程序的显着改进（可能是由于调度线程之间的过度锁争用或最终提交步骤中容器分配的高拒绝率）。 鉴于我们只能通过使用 YARN 进行调度优化来实现有限的（相对于我们的增长率）改进，所以我们没有进一步朝这个方向发展。

另一方面，专门为解决单个 YARN 集群的可扩展性限制而设计的 YARN Federation 对我们来说似乎是一个更有前途的长期计划。它允许应用程序跨越数万个节点的多个集群，同时呈现单个 YARN 集群的视图，这非常适合在我们添加更多集群以适应未来计算增长，而且这个可以做到对用户透明。但是，出于一些原因，我们决定不在 LinkedIn 上使用它。

- 控制平面（Global Policy Generator）的当前实现是一个基于 CLI 的手动过程，它不能处理 Hadoop 集群中的动态变化。
- 它引入了新的依赖项（用于策略存储的 MySQL，用于注册 YARN 的 Zookeeper）并需要许多我们从未测试或使用过的功能，例如 YARN Reservation, unmanaged AM 以及 AMRMProxy。这些复杂性对于我们这样规模的团队来说意义重大。

请注意，设计的大部分复杂性来自允许 YARN 应用程序跨越多个集群，如果应用程序可以保持在单个 YARN 集群的边界内，我们可以避免大部分复杂性并为 YARN 构建特定于域（domain-specific）的负载均衡器，非常像规范的 L7 负载平衡器。

Robin

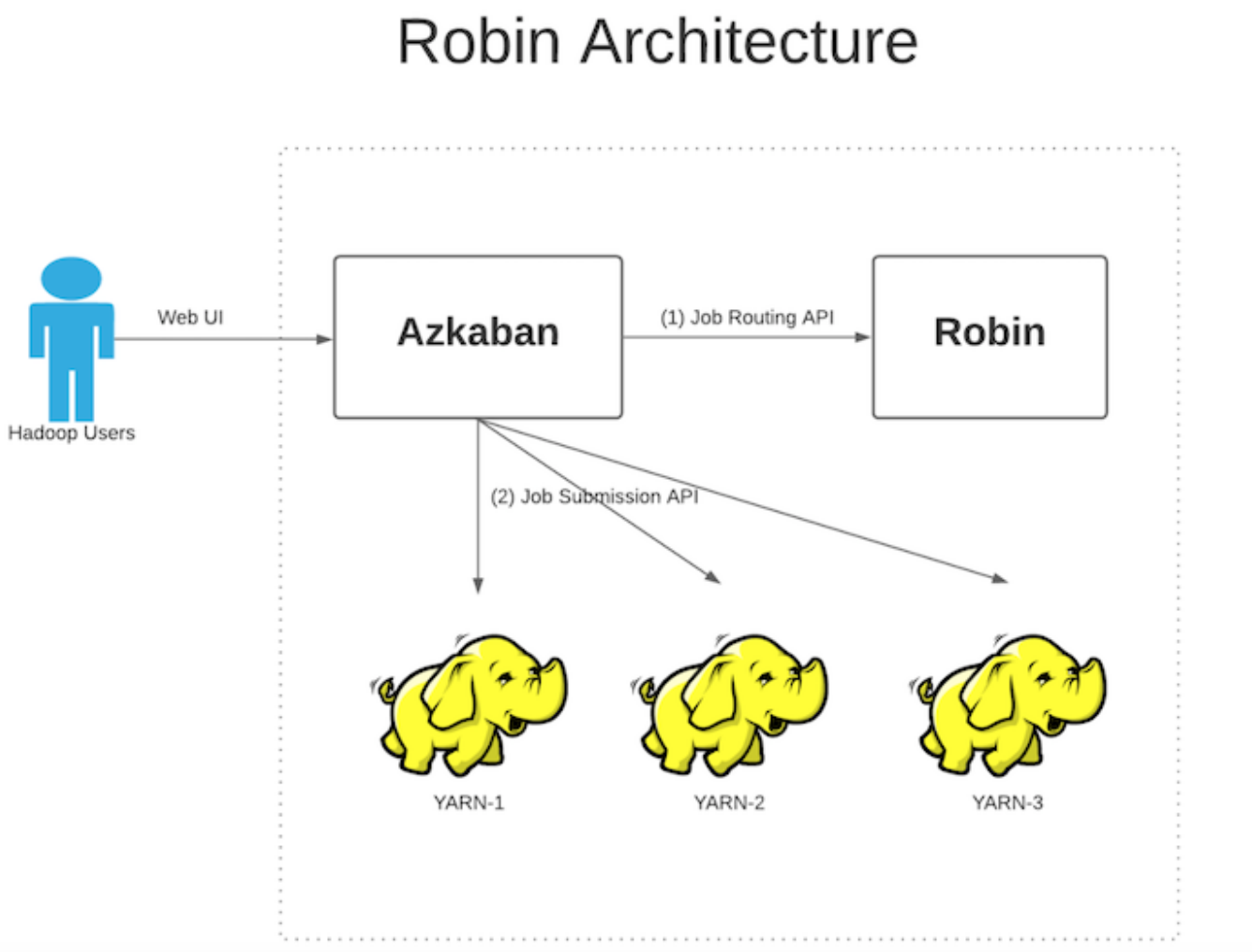
我们设想我们的 Hadoop 集群由每个包含约 5,000 个节点的子集群组成，因此所有应用程序都可以保持在一个子集群的边界内。有了这个假设，我们可以构建一个集群协调器来协调底层 YARN 集群之间的工作。这就是 Robin：一个负载均衡器，用于将 YARN 应用程序动态分发到多个 Hadoop 集群，我们在 LinkedIn 开发它以扩展我们的 YARN 集群。

如果想及时了

解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：过往记忆大数据

在较高的层次上，Robin 提供了一个简单的 REST API，它为给定的作业返回一个 YARN 集群。在提交作业之前，YARN 客户端会与 Robin 核对以确定应将作业路由到哪个集群，然后将作业发送到正确的集群。一旦作业提交后，作业就一直保留在那个集群中。
虽然应用程序最多只能使用单个集群的容量，但我们没有发现这对我们的工作负载构成限制。

在 LinkedIn，大部分工作负载由 Azkaban 管理，Azkaban
是我们的工作流编排引擎，代表最终用户充当 YARN 客户端。 它原生版本只支持单个物理 Hadoop 集群； 因此，我们必须改造 Azkaban 以支持动态作业提交并添加 Robin
集成，以便在我们扩展逻辑集群并在其下添加物理集群时向最终用户呈现单个逻辑集群的视图。因此，大多数最终用户对 Robin 是无感知的。



如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：过往记忆大数据

虽然 Robin 的整体理念及其设计很简单，但我们必须解决一些其他值得一提的问题。

高可用性：Azkaban 是我们 Hadoop 最终用户的核心接口。 Azkaban 中的每一项工作执行都依赖于 Robin，因此 Robin 始终保持高可用是至关重要的。

- Robin 会定期在后台检查每个 YARN 集群的可用资源，并仅根据最新快照做出路由决策，因此即使 Robin 到 YARN 连接间歇性失败，作业也可以路由；
- Robin 被设计为无状态的，因此我们可以通过添加或删除副本来扩大和缩小规模。它部署在 Kubernetes (K8s) 上，以提供动态扩缩容。

负载均衡策略：选择正确的负载均衡策略对于保持每个集群的工作负载平衡以及减少资源争用和作业延迟至关重要。 我们已经尝试了一些策略，例如：

- 使用绝对占优资源公平分享算法（Dominant Resource Fairness，简称 DRF），即，将作业路由到具有最多可用资源的集群。 例如，集群 A 总内存为 100 TB，其中的 20 TB 可用；而集群 B 总内存为 200 TB ，其中的 30 TB 可用，则将作业路由到集群 B。
- 使用相对占优资源公平分享算法，即，将作业路由到可用资源百分比最高的集群。例如，如果集群 A 总内存为 100 TB，其中 20 TB 可用（20% 净空）；而集群 B 总内存为 200 TB ，其中的 30 TB 可用（15% 净空），则将作业路由到集群 A。
- 队列级别的绝对可用资源。即将作业路由到该作业将运行的队列中可用资源最多的集群，（我们所有的集群具有相同的队列结构）。

我们在 DynoYARN 中模拟了带有生产工作负载跟踪的每个策略，并发现第一个策略最小化了应用程序延迟，使其成为我们的最佳策略。

数据本地性：如今，LinkedIn 的工作负载性能在很大程度上依赖于数据本地性。作为将 Robin 部署到我们最大的生产集群的一部分，我们将现有的 YARN 集群拆分为两个相同大小的 YARN 子集群，这两个 YARN 子集群和 HDFS 集群共享相同的节点。因为现在每个作业只能访问 50% 的 HDFS 数据，与拆分前可以访问 100% 的数据相比，数据局部性有所损失。 为了缓解这种情况，我们必须在两个子集群之间平均分配每个机架上的机器，以便无论作业在哪个集群上运行，仍然可以从同一机架访问数据。 事实证明，这个方法是很有效的。

下一步

LinkedIn 正在积极迁移到 Azure。 下一步，我们正在研究在云上管理和扩展 YARN 集群的最佳方式。将我们的本地 Hadoop YARN 集群的 10,000 多个节点提升到云端有很多令人兴奋的挑战；与此同时，Azure 中也有令人兴奋的探索机会，比如 Spot 实例、自动缩放等。我们还计划扩展 Robin 以支持跨本地和云集群的路由，以及后期会开源 Robin。

本文翻译自 Scaling LinkedIn's Hadoop YARN cluster beyond 10,000 nodes