阿里云

藏经阁

阿里云开发者电子书系列

“伏羲”神算

阿里巴巴经济体核心调度系统揭秘

阿里巴巴 9 位技术专家深度分析

看“伏羲”如何引领分布式调度技术的发展趋势

飞天大数据平台 APSARA BIG DATA PLATFORM × 飞天AI平台 APSARA AI PLATFORM

“伏羲”神算
阿里巴巴经济体核心调度系统揭秘

扫一扫二维码图案，关注我吧

MaxCompute 开发者社区钉钉群

SaaS 模式云数据仓库 MaxCompute

飞天大数据平台钉钉群

阿里云开发者“藏经阁”海量免费电子书下载

目录

阿里经济体核心调度系统

——伏羲（Fuxi）2.0 破界之旅

阿里经济体核心调度系统 — 伏羲（Fuxi）2.0 破界之旅

本文作者 李超 | 阿里云智能 资深技术专家

伏羲（Fuxi）是十年前最初创立飞天平台时的三大服务之一（分布式存储 Pangu，分布式计算 MaxCompute，分布式调度 Fuxi），当时的设计初衷是为了解决大规模分布式资源的调度问题（本质上是多目标的最优匹配问题）。

随阿里经济体和阿里云丰富的业务需求（尤其是双十一）和磨练，伏羲的内涵不断扩大，从单一的资源调度器（对标开源系统的 YARN）扩展成大数据的核心调度服务，覆盖数据调度（Data Placement）、资源调度（Resouce Management）、计算调度（Application Manager）、和本地微（自治）调度（即正文中的单机调度）等多个领域，并在每一个细分领域致力于打造超越业界主流的差异化能力。

过去十年来，伏羲在技术能力上每年都有一定的进展和突破（如 2013 年的 5K，2015 年的 Sortbenchmark 世界冠军，2017 年的小规模离在/在离混部能力，2019 年的 Yugong 发布并论文被 VLDB 接受等等）。本章试从面向大数据/云计算的调度挑战出发，介绍各个子领域的关键进展，并回答什么是“伏羲 2.0”。

1-1. 多元化的 4 大调度系统演进路线

过去 10 年，是云计算的 10 年，伴随云计算的爆炸式增长，大数据行业的工作方式也发生了很大的变化：从传统的自建自运维 hadoop 集群，变成更多的依赖云上的弹性低成本计算资源。海量大数据客户的信任和托付，对阿里大数据系统来说，是很大的责任，但也催生出了大规模、多场景、低成本、免运维的 MaxCompute 通用计算系统。

同样的 10 年，伴随着阿里年年双 11，MaxCompute 同样支撑了阿里内部大数据的蓬勃发展，从原来的几百台，到现在的 10 万台物理机规模。

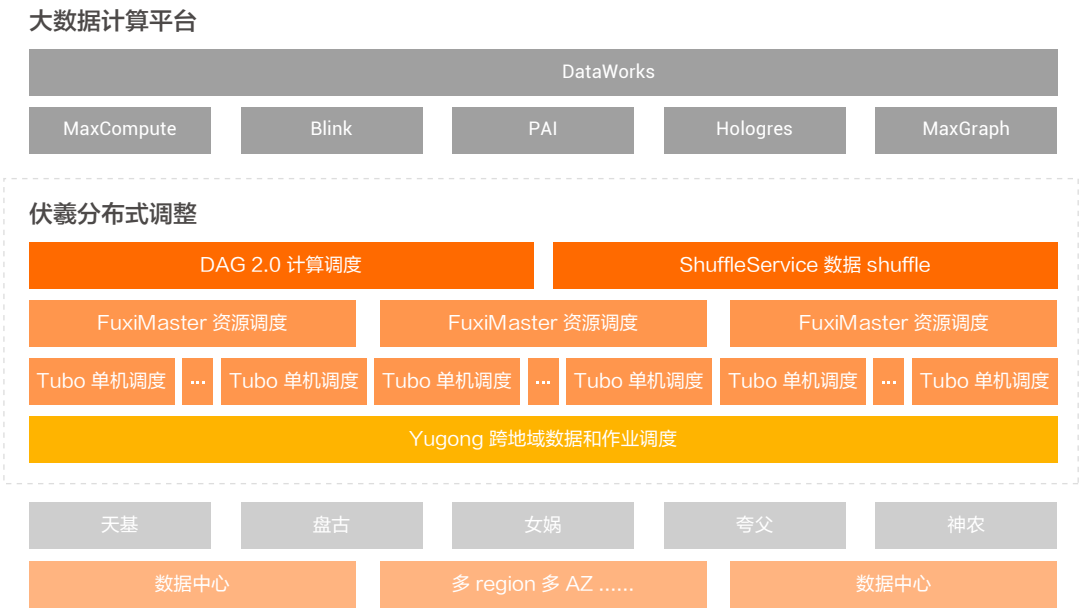
双线需求，殊途同归，海量资源池，如何自动匹配到大量不同需求的异地客户计算需求上，需要调度系统的工作。本文主要介绍阿里大数据的调度系统 Fuxi 往 2.0 的演进。先给大家介绍几个概念：

- 首先，数据从哪里来？数据往往伴随着在线业务系统产生。而在线系统，出于延迟和容灾的考虑，往往遍布北京、上海、深圳等多个地域，如果是跨国企业，还可能遍布欧美等多个大陆的机房。这也造成了我们的数据天然分散的形态。而计算，也可能发生在任意一个地域和机房。可是网络，是他们中间的瓶颈，跨地域的网络，在延迟和带宽上，远远无法满足大数据计算的需求。如何平衡计算资源、数据存储、跨域网络这几点之间的平衡，需要做好“数据调度”；

- 其次，有了数据，计算还需要 CPU，内存，甚至 GPU 等资源，当不同的公司，或者单个公司内部不同的部门，同时需要计算资源，而计算资源紧张时，如何平衡不同的用户，不同的作业？作业也可能长短不一，重要程度不尽相同，今天和明天的需求也大相径庭。除了用户和作业，计算资源本身可能面临硬件故障，但用户不想受影响。所有这些，都需要“资源调度”；

- 有了数据和计算资源，如何完成用户的计算任务，比如一个 SQL query？这需要将一个大任务，分成几个步骤，每个步骤又切分成成千上万个小任务，并行同时计算，才能体现出分布式系统的加速优势。但小任务切粗切细，在不同的机器上有快有慢，上下步骤如何交接数据，同时避开各自故障和长尾，这些都需要“计算调度”；

- 很多不同用户的不同小任务，经过层层调度，最后汇集到同一台物理机上，如何避免单机上真正运行时，对硬件资源使用的各种不公平，避免老实人吃亏。避免重要关键任务受普通任务影响，这都需要内核层面的隔离保障机制。同时还要兼顾隔离性和性能、成本的折中考虑。这都需要“单机调度”。



2013 年，伏羲在飞天 5K 项目中对系统架构进行了第一次大重构，解决了规模、性能、利用率、容错等线上问题，并取得世界排序大赛 Sortbenchmark 四项冠军，这标志着 Fuxi 1.0 的成熟。

2019 年，伏羲再次出发，从技术上对系统进行了第二次重构，发布 Fuxi 2.0 版本：阿里自研的新一代高性能、分布式的数据、资源、计算、单机调度系统。Fuxi 2.0 进行了全面的技术升级，在全区域数据排布、去中心化调度、在线离线混合部署、动态计算等方面全方位满足新业务场景下的调度需求。

伏羲 2.0 成果概览

- 业内首创跨地域多数据中心的数据调度方案 -Yugong，通过 3% 的冗余存储，节省 80% 的跨地域网络带宽；

- 业内领先的去中心化资源调度架构，单集群支持 10 万服务器* 10 万并发 job 的高频调度；

- 动态 DAG 闯入传统 SQL 优化盲区，TPC-DS 性能提升 27%，conditional join性能提升 3X；

· 创新性的数据动态 shuffle 和全局跨级优化，取代业界磁盘 shuffle；线上千万 job，整体性能提升 20%，成本下降 15%，出错率降低一个数量级；

· 在线离线规模化混合部署，在线集群利用率由 10% 提升到 40%，双十一大促节省 4200 台 F53 资源，且同时保障在线离线业务稳定。

1–2. 数据调度 – 跨地域的数据调度

阿里巴巴在全球都建有数据中心，每个地区每天会产生一份当地的交易订单信息，存在就近的数据中心。北京的数据中心，每天会运行一个定时任务来统计当天全球所有的订单信息，需要从其他数据中心读取这些交易数据。当数据的产生和消费不在一个数据中心时，我们称之为跨数据中心数据依赖（下文简称跨中心依赖）。

图. 阿里巴巴全球数据中心

MaxCompute 上每天运行着数以千万计的作业，处理 EB 级别的数据。这些计算和数据分布在全球的数据中心，复杂的业务依赖关系产生了大量的跨中心依赖。相比于数据中心内的网络，跨数据中心网络（尤其是跨域的网络）是非常昂贵的，同时具有带宽小、延迟高、稳定性低的特点。比如网络延迟，数据中心内部网络的网络延迟一般在 100 微秒以下，而跨地域的网络延迟则高达数十毫秒，相差百倍以上。因此，如何高效地将跨中心依赖转化为数据中心内部的数据依赖，减少跨数据中心网络带宽消耗，从而降低成本、提高系统效率，对 MaxCompute 这样超大规模计算平台而言，具有极其重要的意义。

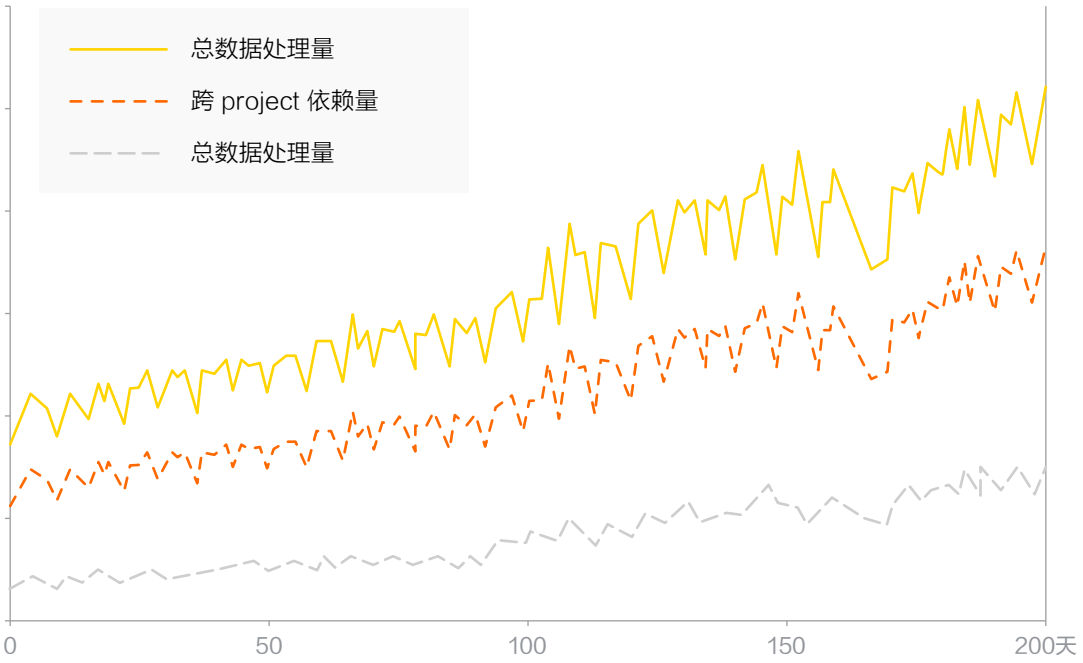


图. MaxCompute 平台数据及依赖增长趋势

为了解决这个问题，我们在数据中心上增加了一层调度层，用于在数据中心之间调度数据和计算。这层调度独立于数据中心内部的调度，目的是实现跨地域维度上存储冗余--计算均衡--长传带宽--性能最优之间的最佳平衡。这层调度层包括跨数据中心数据缓存、业务整体排布、作业粒度调度。

首先是业务的整体排布策略。数据和计算以业务为单位组织在一起（MaxCompute 中称之为 project），每个 project 被分配在一个数据中心，包括数据存储和计算作业。如果将 project 看做一个整体，可以根据作业对数据的依赖关系计算出 project 之间的相互依赖关系。如果能将有互相数据依赖的 project 放在一个数据中心，就可以减少跨中心依赖。但 project 间的依赖往往复杂且不断变化，很难有一劳永逸的排布策略，并且 project 排布需要对 project 进行整体迁移，周期较长，且需要消耗大量的带宽。

其次是对访问频次高的数据进行跨数据中心缓存，在缓存空间有限的约束下，选择合适的数据进行换入换出。不同于其他缓存系统，MaxCompute 的数据（分区）以表的形式组织在一起，每张表每天产生一个或多个分区，作业访问数据也有一些特殊规律，比如一般访问的是连续分区、生成时间越新的分区访问概率越大。

最后，当 project 之间的互相依赖集中在极少数几个作业上，并且作业的输入数据

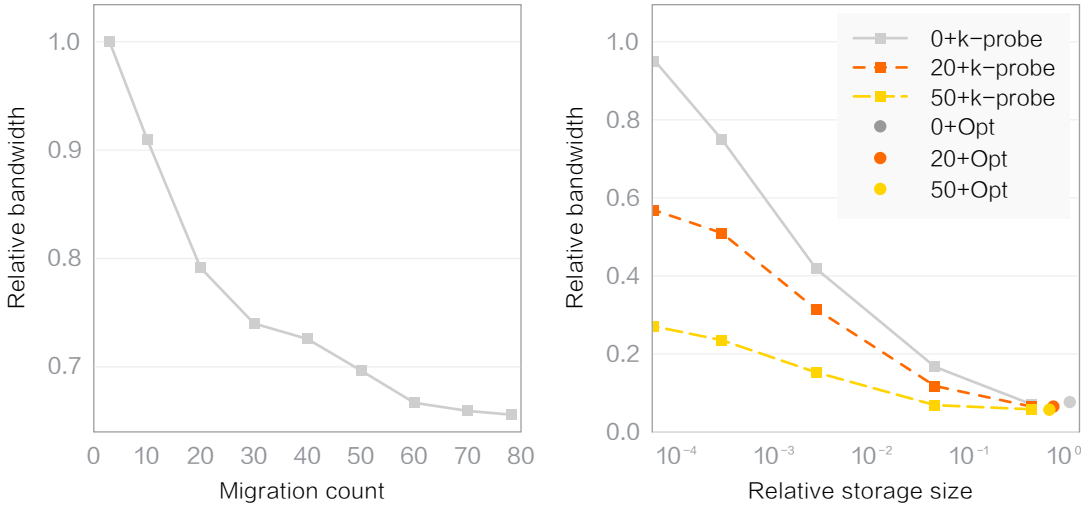
量远大于输出数据量时，比起数据缓存和 project 整体迁移，更好的办法是将这些作业调度到数据所在的数据中心，再将作业的输出远程写回原数据中心，即作业粒度调度。如何在作业运行之前就预测到作业的输入输出数据量和资源消耗，另一方面当作业调度到 remote 数据中心后，如何保证作业运行不会变慢，不影响用户体验，这都是作业粒度调度要解决的问题。

本质上，数据缓存、业务排布、作业粒度调度三者都在解同一个问题，即在跨地域多数据中心系统中减少跨中心依赖量、优化作业的 data locality、减少网络带宽消耗。

以 project 为粒度的多集群业务排布算法

随着上层业务的不断发展，业务的资源需求和数据需求也在不断变化。比如一个集群的跨中心依赖增长迅速，无法完全通过数据缓存来转化为本地读取，这就会造成大量的跨数据中心流量。因此我们需要定期对业务的排布进行分析，根据业务对计算资源、数据资源的需求情况，以及集群、机房的规划，通过业务的迁移来降低跨中心依赖以及均衡各集群压力。

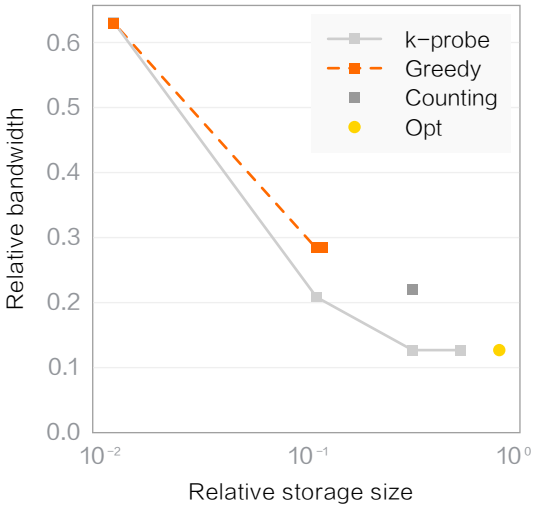
下图展示了某个时刻业务迁移的收益分析：左图横轴为迁移的 project 数量，纵轴为带宽减少比例，可以看出大约移动 60 个 project 就可以减少约 30% 的带宽消耗。右图统计了不同排布下（迁移 0 个、20 个、50 个 project）的最优带宽消耗，横轴为冗余存储，纵轴为带宽。



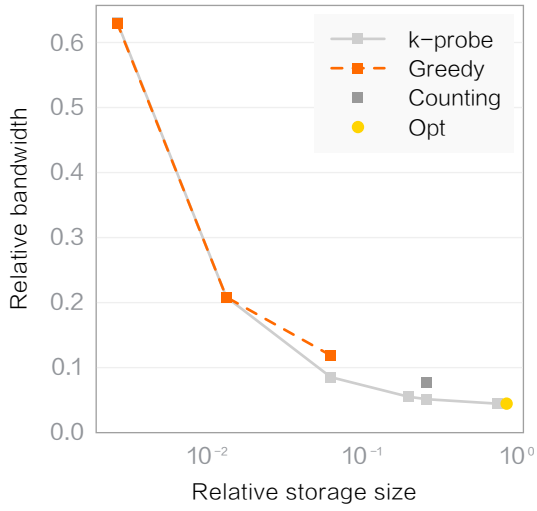
跨数据中心数据缓存策略

我们首次提出了跨地域、跨数据中心数据缓存这一概念，通过集群的存储换集群间带宽，在有限的冗余存储下，找到存储和带宽最佳的 tradeoff。通过深入的分析 MaxCompute 的作业、数据的特点，我们设计了一种高效的算法，根据作业历史的 workload、数据的大小和分布，自动进行缓存的换入换出。

我们研究了多种数据缓存算法，并对其进行了对比试验，下图展示了不同缓存策略的收益，横轴是冗余存储空间，纵轴是带宽消耗。从图中可以看出，随着冗余存储的增加，带宽成本不断下降，但收益比逐渐降低，我们最终采用的 k-probe 算法在存储和带宽间实现了很好的平衡。



(a)DC1



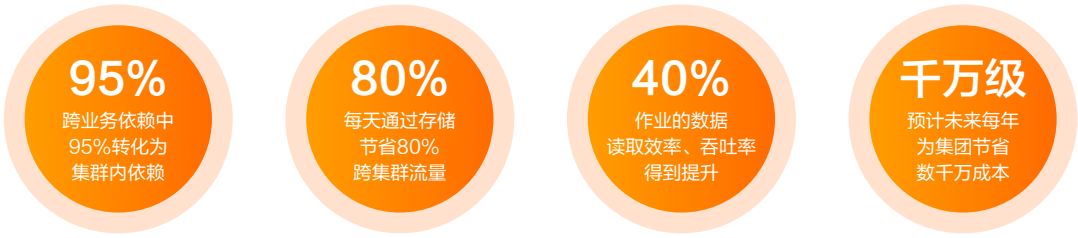
(a)DC2

跨数据中心计算调度机制

我们打破了计算资源按照数据中心进行规划的限制，理论上允许作业跑在任何一个数据中心。我们将调度粒度拆解到作业粒度，根据每个作业的数据需求、资源需求，为其找到一个最合适的数据中心。在对作业进行调度之前需要知道这个作业的输入和输出，目前我们有两种方式获得这一信息，对于周期性作业，通过对作业历史运行数据进行分析推测出作业的输入输出；对于偶发的作业，我们发现其产生较大跨域流量时，动态的将其调度到数据所在的数据中心上运行。另外，调度计算还要考虑作业对计算资源的需求，防止作业全部调度到热点数据所在的数据中心，造成任务堆积。

线上效果

线上三种策略相辅相成，数据缓存主要解决周期类型作业、热数据的依赖；作业粒度调度主要解决临时作业、历史数据的依赖；并周期性地通过业务整体排布进行全局优化，用来降低跨中心依赖。整体来看，通过三种策略的共同作用，降低了约 90% 的跨地域数据依赖，通过约 3% 的冗余存储节省了超过 80% 的跨数据中心带宽消耗，将跨中心依赖转化为本地读取的比例提高至 90%。下图以机房为单位展示了带宽的收益：



1-3. 资源调度 – 去中心化的多调度器架构

2019 年双十一，MaxCompute 平台产生的数据量已接近 EB 级别，作业规模达到了千万，有几十亿的 worker 跑在几百万核的计算单元上，在超大规模（单集群超过万台），高并发的场景下，如何快速地给不同的计算任务分配资源，实现资源的高速流转，需要一个聪明的“大脑”，而这就是集群的资源管理与调度系统（简称资源调度系统）。

资源调度系统负责连接成千上万的计算节点，将数据中心海量的异构资源抽象，并提供给上层的分布式应用，像使用一台电脑一样使用集群资源，它的核心能力包括规模、性能、稳定性、调度效果、多租户间的公平性等等。一个成熟的资源调度系统需要在以下五个方面进行权衡，做到“既要又要”，非常具有挑战性。

2013 年的 5K 项目初步证明了伏羲规模化能力，此后资源调度系统不断演进，并通过 MaxCompute 平台支撑了阿里集团的大数据计算资源需求，在核心调度指标上保持着对开源系统的领先性，比如：1、万台规模集群，调度延时控制在了 10 微秒级别，worker 启动延时控制在 30 毫秒；2、支持任意多级租户的资源动态调节能力（支持十万级别的租户）；3、极致稳定，调度服务全年 99.99% 的可靠性，并做到服务秒级故障恢复。

1-3-1. 单调度器的局限性

线上的规模与压力

大数据计算的场景与需求正在快速增长（下图是过去几年 MaxCompute 平台计算和数据的增长趋势）。单集群早已突破万台规模，急需提供十万台规模的能力。

图. MaxCompute 2015 ~ 2019 线上作业情况

但规模的增长将带来复杂度的极速上升，机器规模扩大一倍，资源请求并发度也会翻一番。在保持既有性能、稳定性、调度效果等核心能力不下降的前提下，可以通过对调度器持续性能优化来扩展集群规模（这也是伏羲资源调度 1.0 方向），但受限于单机的物理限制，这种优化总会存在天花板，因此需要从架构上优化来彻底规模和性能的可扩展性问题。

调度需求的多样性

伏羲支持了各种各样的大数据计算引擎，除了离线计算（SQL、MR），还包括实时计算、图计算，以及近几年迅速发展面向人工智能领域的机器学习引擎。

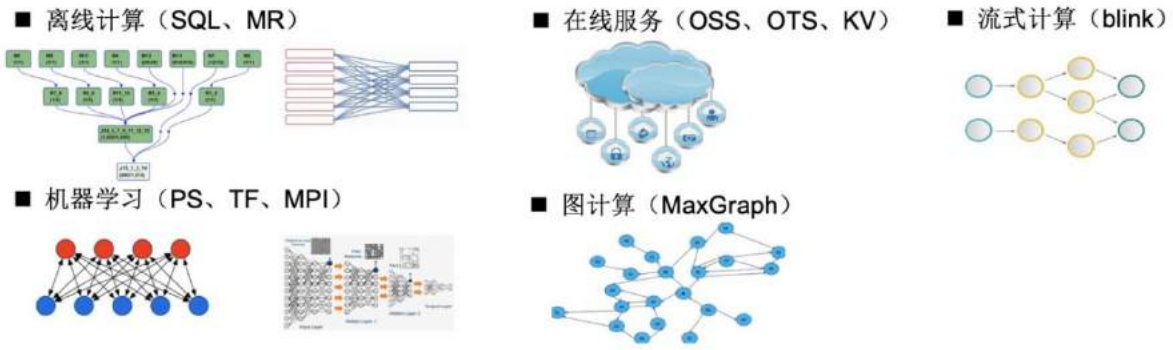


图. 资源调度器的架构类型

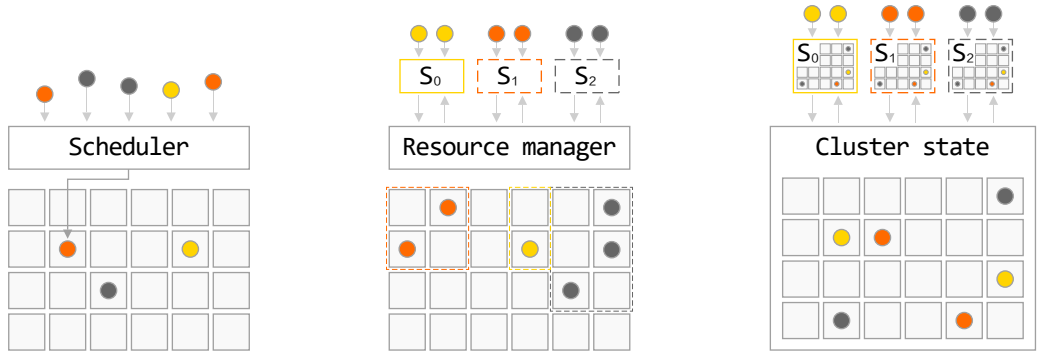
场景的不同对资源调度的需求也不相同，比如，SQL 类型的作业通常体积小、运行时间短，对资源匹配的要求低，但对调度延时要求高，而机器学习的作业一般体积大、运行时间长，调度结果的好坏可能对运行时间产生直接影响，因此也能容忍通过较长的调度延时换取更优的调度结果。资源调度需求这种多样性，决定了单一调度器很难做到“面面俱到”，需要各个场景能定制各自的调度策略，并进行独立优化。

灰度发布与工程效率

资源调度系统是分布式系统中最复杂最重要的模块之一，需要有严苛的生产发布流程来保证其线上稳定运行。单一的调度器对开发人员要求高，出问题之后影响范围大，测试发布周期长，严重影响了调度策略迭代的效率，在快速改进各种场景调度效果的过程中，这些弊端逐渐显现，因此急需从架构上改进，让资源调度具备线上的灰度能力，从而大幅提升工程效率。

1-3-2. 去中心化的多调度器架构

为了解决上述规模和扩展性问题，更好地满足多种场景的调度需求，同时从架构上支持灰度能力，伏羲资源调度 2.0 在 1.0 的基础上对调度架构做了大规模的重构，引入了去中心化的多调度器架构。



(a)Monolithic scheduler. (b)Two-level manager. (c)Shared-state scheduling.

Borg/Kubernetes **YARN** **Omega/Apollo**

图. 资源调度的架构类型

我们将系统中最核心的资源管理和资源调度逻辑进行了拆分解耦，使两者同时具备了多partition的可扩展能力（如下图所示），其中：

- 资源调度器（Scheduler）：负责核心的机器资源和作业资源需求匹配的调度逻辑，可以横向扩展；

- 资源管理和仲裁服务（ResourceManagerService，简称RMS）：负责机器资源和状态管理，对各个 Scheduler 的调度结果进行仲裁，可以横向扩展；

- 调度协调服务（Coordinator）：管理资源调度系统的配置信息，Meta 信息，以及对机器资源、Scheduler、RMS 的可用性和服务角色间的可见性做仲裁。不可横向扩展，但有秒级多机主备切换能力；

- 调度信息收集监控服务（FuxiEye）：统计集群中每台机的运行状态信息，给 Scheduler 提供调度决策支持，可以横向扩展；

- 用户接口服务（ApiServer）：为资源调度系统提供外部调用的总入口，会根据

Coordinator 提供的 Meta 信息将用户请求路由到资源调度系统具体的某一个服务上，可以横向扩展。

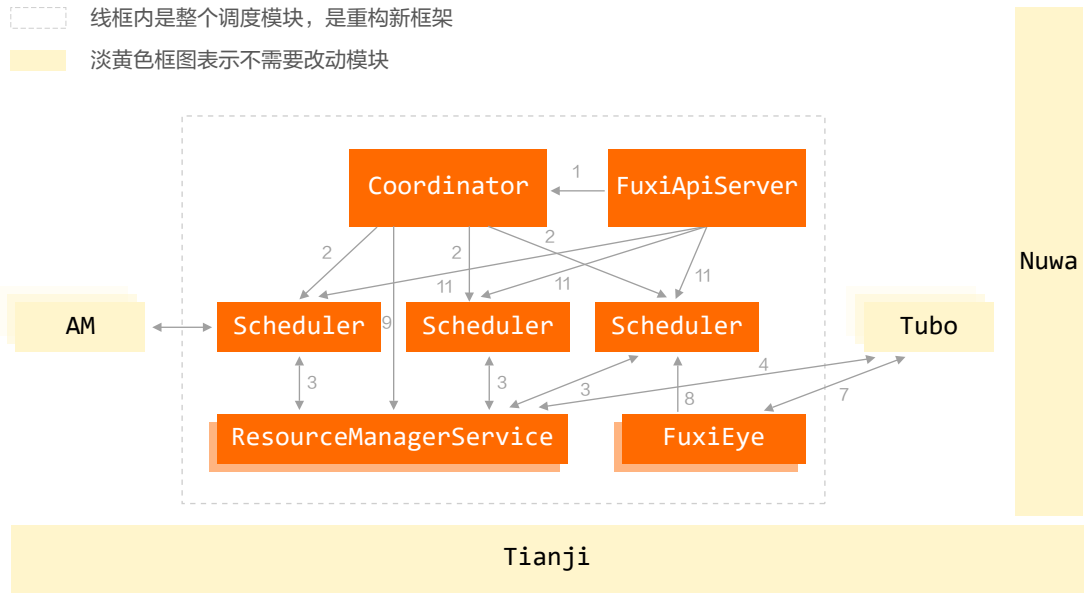


图. 伏羲多调度器新架构

1-3-3. 上线数据

以下是 10w 规模集群 /10 万作业并发场景调度器核心指标（5 个 Scheduler、5 个 RMS，单 RMS 负责 2w 台机器，单 Scheduler 并发处理 2w 个作业）。通过数据可以看到，集群 10w 台机器的调度利用率超过了 99%，关键调度指标，单 Scheduler 向 RMS commit 的 slot 的平均数目达到了 1w slot/s。

在保持原有单调度器各项核心指标稳定不变的基础上，去中心化的多调度器框架实现了机器规模和应用并发度的双向扩展，彻底解决了集群的可扩展性问题。

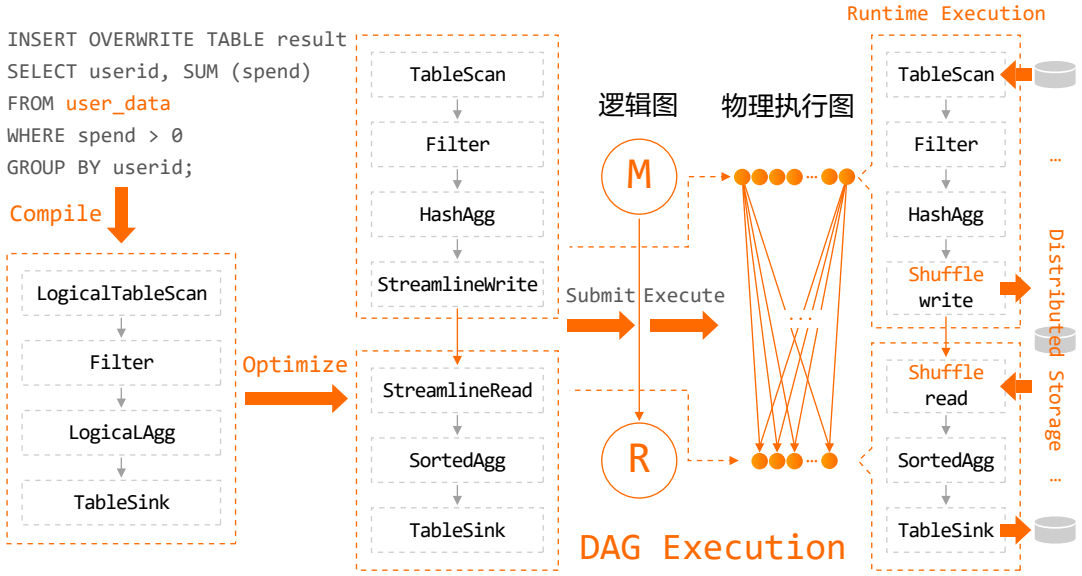
10w	10w	99%	5w
集群规模	并发 Job 数量	调度利用率	Commit qps
5 个 scheduler+5 个 rms 支撑 10w 机器规模，仍可横向扩展	5 个 scheduler 并发运行 10w running job	资源调度利用率超过 99%，资源无任何浪费	scheduler 和 rms 间维持高频 commit qps，系统各个环节丝般顺滑

目前资源调度的新架构已全面上线，各项指标持续稳定。在多调度器架构基础上，我们把机器学习场景调度策略进行了分离，通过独立的调度器来进行持续的优化。同时通过测试专用的调度器，我们也让资源调度具备了灰度能力，调度策略的开发和上线周期显著缩短。

1-4. 计算调度 – 从静态到动态

分布式作业的执行与单机作业的最大区别，在于数据的处理需要拆分到不同的计算节点上，“分而治之”的执行。这个“分”，包括数据的切分，聚合以及对应的不同逻辑运行阶段的区分，也包括在逻辑运行阶段间数据的 shuffle 传输。每个分布式作业的中心管理点，也就是 application master (AM)。这个管理节点也经常被称为 DAG (Directional Acyclic Graph，有向无环图) 组件，是因为其最重要的责任，就是负责协调分布式系统中的作业执行流程，包括计算节点的调度以及数据流（shuffle）。

对于作业的逻辑阶段和各个计算节点的管理，以及 shuffle 策略的选择/执行，是一个分布式作业能够正确完成重要前提。这一特点，无论是传统的 MR 作业，分布式 SQL 作业，还是分布式的机器学习/深度学习作业，都是一脉相承的，为了帮助更好的理解计算调度（DAG 和 Shuffle）在大数据平台中的位置，我们可以通过 Max-Compute 分布式 SQL 的执行过程作为例子来了解：

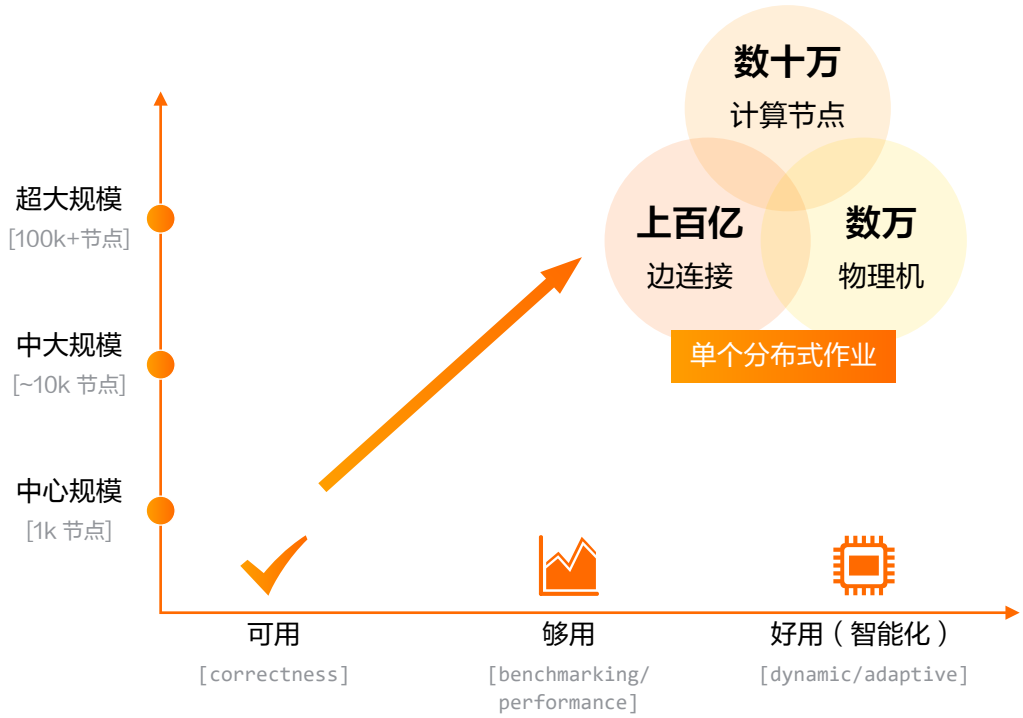


在这么一个简单的例子中，用户有一张订单表 order_data，存储了海量的交易信息，用户想所有查询花费超过 1000 的交易订单按照 userid 聚合后，每个用户的花费之和是多少。于是提交了如下 SQL query:

```
INSERT OVERWRITE TABLE result
SELECT userid, SUM(spend)
FROM  order_data
WHERE spend > 1000
GROUP BY userid;
```

这个 SQL 经过编译优化之后生成了优化执行计划，提交到 Fuxi 管理的分布式集群中执行。我们可以看到，这个简单的 SQL 经过编译优化，被转换成一个具有 M->R 两个逻辑节点的 DAG 图，也就是传统上经典的 MR 类型作业。而这个图在提交给 Fuxi 系统后，根据每个逻辑节点需要的并发度，数据传输边上的 shuffle 方式，调度时间等等信息，就被物化成右边的物理执行图。物理图上的每个节点都代表了一个具体的执行实例，实例中包含了具体处理数据的算子，特别的作为一个典型的分布式作业，其中包含了数据交换的算子 shuffle——负责依赖外部存储和网络交换节点间的数据。一个完整的计算调度，包含了上图中的 DAG 的调度执行以及数据 shuffle 的过程。

阿里计算平台的 Fuxi 计算调度，经过十年的发展和不断迭代，成为了作为阿里集团内部以及阿里云上大数据计算的重要基础设施。今天计算调度同时服务了以 MaxCompute SQL 和 PAI 为代表的多种计算引擎，在超 10 万台机器上日均运行着千万界别的分布式 DAG 作业，每天处理 EB 数量级的数据。一方面随着业务规模和需要处理的数据量的爆发，这个系统需要服务的分布式作业规模也在不断增长；另一方面，业务逻辑以及数据来源的多样性，计算调度在阿里已经很早就跨越了不同规模上的可用/够用的前中期阶段，2.0 上我们开始探索更加前沿的智能化执行阶段。



在云上和阿里集团的大数据实践中，我们发现对于计算调度需要同时具备超大规模和智能化的需求，以此为基本诉求我们开始了 Fuxi 计算调度 2.0 的研发。下面就为大家从 DAG 调度和数据 shuffle 两个方面分别介绍计算调度 2.0 的工作。

1-4-1. Fuxi DAG 2.0--动态、灵活的分布式计算生态

DAG 调度的挑战

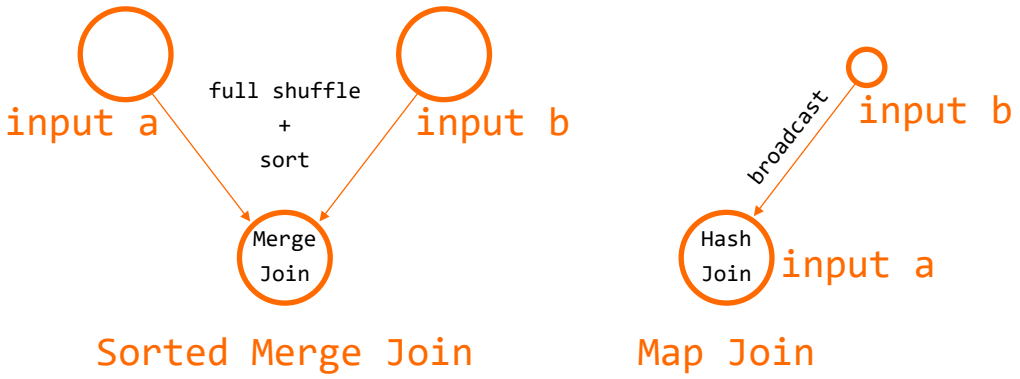
传统的分布式作业 DAG，一般是在作业提交前静态指定的，这种指定方式，使得作业的运行没有太多动态调整的空间。放在 DAG 的逻辑图与物理图的背景中来说，这要求分布式系统在运行作业前，必须事先了解作业逻辑和处理数据各种特性，并能够准确回答作业运行过程，各个节点和连接边的物理特性问题，然而在现实情况中，许多和运行过程中数据特性相关的问题，都只有个在执行过程中才能被最准确的获得。静态的 DAG 执行，可能导致选中的是非最优的执行计划，从而导致各种运行时的效率低下，甚至作业失败。这里我们可以用一个分布式 SQL 中很常见的例子来说明：

```
SELECT a.spend, a.userid, b.age
FROM   (
        SELECT  spend, userid
        FROM    order_data
        WHERE   spend > 1000
      ) a
JOIN   (
        SELECT  userid, age
        FROM    user
        WHERE   age > 60
      ) b
ON     a.userid = b.userid;
```

上面是一个简单的 join 的例子，目的是获取 60 岁以上用户花费大于 1000 的详细信息，由于年纪和花费在两张表中，所以此时需要做一次 join。一般来说 join 有两种实现方式：

一是 Sorted Merge Join（如下图左侧的所示）：也就是对于 a 和 b 两个子句执行后的数据按照 join key (userid) 进行分区，然后在下游节点按照相同的 key 进行 Merge Join 操作，实现 Merge Join 需要对两张表都要做 shuffle 操作——也就是进行一次数据交换，特别的如果有数据倾斜（例如某个 userid 对应的交易记录特别多），这时候 MergeJoin 过程就会出现长尾，影响执行效率；

二是 Map join (Hash join) 的方式（如下图右侧所示）：上述 sql 中如果 60 岁以上的用户信息较少，数据可以放到一个计算节点的内存中，那对于这个超小表可以不做 shuffle，而是直接将其全量数据 broadcast 到每个处理大表的分布式计算节点上，大表不用进行 shuffle 操作，通过在内存中直接建立 hash 表，完成 join 操作，由此可见 map join 优化能大量减少 (大表) shuffle 同时避免数据倾斜，能够提升作业性能。但是如果选择了 map join 的优化，执行过程中发现小表数据量超过了内存限制（大于 60 岁的用户很多），这个时候 query 执行就会由于 oom 而失败，只能重新执行。



Sorted Merge Join　　Map Join

但是在实际执行过程中，具体数据量的大小，需要在上游节点完成后才能被感知，因此在提交作业前很难准确的判断是否可以采用 Map join 优化，从上图可以看出在 Map Join 和 Sorted Merge Join 上 DAG 图是两种结构，因此这需要 DAG 调度在执行过程中具有足够的动态性，能够动态的修改 DAG 图来达到执行效率的最优。我们在阿里集团和云上海量业务的实践中发现，类似 map join 优化这样的例子是很普遍的，从这些例子可以看出，随着大数据平台优化的深入进行，对于 DAG 系统的动态性要求越来越高。

由于业界大部分 DAG 调度框架都在逻辑图和物理图之间没有清晰的分层，缺少执行过程中的动态性，无法满足多种计算模式的需求。例如 spark 社区很早提出了运行时调整 Join 策略的需求 (Join: Determine the join strategy (broadcast join or shuffle join) at runtime)，但是目前仍然没有解决。

除此上述用户体感明显的场景之外，随着 MaxCompute 计算引擎本身更新换代和优化器能力的增强，以及 PAI 平台的新功能演进，上层的计算引擎自身能力在不断的增强。对于 DAG 组件在作业管理，DAG 执行等方面的动态性，灵活性等方面的需求也日益强烈。在这样的一个大的背景下，为了支撑计算平台下个 10 年的发展，伏羲团队启动了 DAG 2.0 的项目，以更好的支撑上层计算需求。

DAG 2.0 动态灵活统一的执行框架

DAG 2.0 通过逻辑图和物理图的清晰分层，可扩展的状态机管理，插件式的系统管理，以及基于事件驱动的调度策略等基座设计，实现了对计算平台上多种计算模式的统一管理，并更好的提供了作业执行过程中在不同层面上的动态调整能力。作业执行的动态性和统一 DAG 执行框架是 DAG 2.0 的两个主要特色：

作业执行的动态性

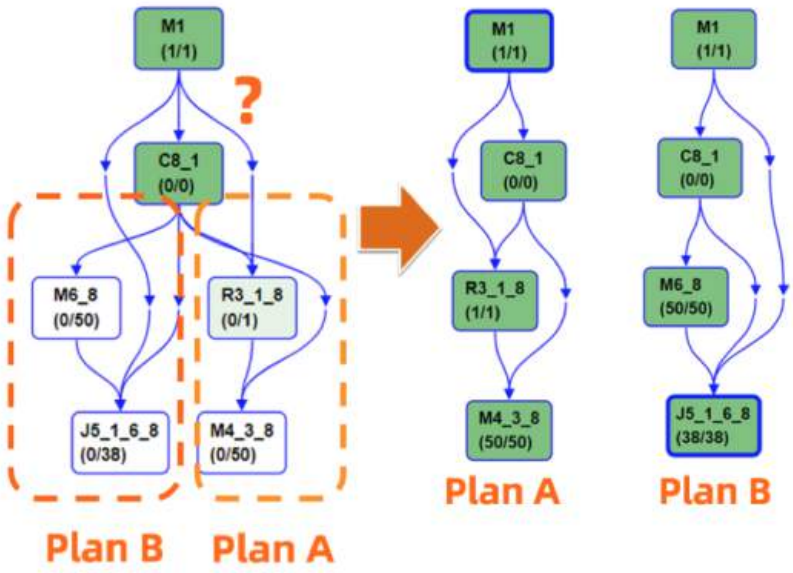
如前所述，分布式作业执行的许多物理特性相关的问题，在作业运行前是无法被感知的。例如一个分布式作业在运行前，能够获得的只有原始输入的一些基本特性(数据量等)， 对于一个较深的 DAG 执行而言，这也就意味着只有根节点的物理计划(并发度选择等) 可能相对合理，而下游的节点和边的物理特性只能通过一些特定的规则来猜测。这就带来了执行过程中的不确定性，因此，要求一个好的分布式作业执行系统，需要能够根据中间运行结果的特点，来进行执行过程中的动态调整。

而 DAG/AM 作为分布式作业唯一的中心节点和调度管控节点，是唯一有能力收集并聚合相关数据信息，并基于这些数据特性来做作业执行的动态调整。这包括简单的物理执行图调整（比如动态的并发度调整），也包括复杂一点的调整比如对 shuffle 方式和数据编排方式重组。除此以外，数据的不同特点也会带来逻辑执行图调整的需求：对于逻辑图的动态调整，在分布式作业处理中是一个全新的方向，也是我们在 DAG 2.0 里面探索的新式解决方案。

还是以 map join 优化作为例子，由于 map join 与默认 join 方式（sorted merge join）对应的其实是两种不同优化器执行计划，在 DAG 层面，对应的是两种不同的逻辑图。DAG 2.0 的动态逻辑图能力很好的支持了这种运行过程中根据中间数据特性的动态优化，而通过与上层引擎优化器的深度合作，在 2.0 上实现了业界首创的 conditional join 方案。如同下图展示，在对于join使用的算法无法被事先确定的时候，分布式调度执行框架可以允许优化提交一个 conditional DAG，这样的 DAG 同时包括使用两种不同 join 的方式对应的不同执行计划支路。在实际执行时，AM 根据上游产出数据量，动态选择一条支路执行（plan A or plan B）。这样子的动态逻辑图执行流程，能够保证每次作业运行时，根据实际产生的中间数据特性，选择最优的执行计划。在这个例子中，

- 当 M1 输出的数据量较小时，允许其输出被全量载入下游单个计算节点的内存，DAG 就会选择优化的 map join (plan A)，来避免额外的 shuffle 和排序。

- 当 M1 输出的数据量大到一定程度，已经不属于 map join 的适用范围，DAG 就可以自动选择走 merge join，来保证作业的成功执行。



除了 map join 这个典型场景外，借助 DAG2.0 的动态调度能力，MaxCompute 在解决其他用户痛点上也做了很多探索，并取得了不错的效果。例如智能动态并发度调整：在执行过程中依据分区数据统计调整，动态调整并发度；自动合并小分区，避免不必要的资源使用，节约用户资源使用；切分大分区，避免不必要的长尾出现等等。

统一的 AM/DAG 执行框架

除了动态性在 SQL 执行中带来的重大性能提升外，DAG 2.0 抽象分层的点，边，图架构上，也使其能通过对点和边上不同物理特性的描述，对接不同的计算模式。业界各种分布式数据处理引擎，包括 SPARK, FLINK, HIVE, SCOPE, TENSOR-FLOW 等等，其分布式执行框架的本源都可以归结于 Dryad 提出的 DAG 模型。我们认为对于图的抽象分层描述，将允许在同一个 DAG 系统中，对于离线/实时/流/渐进计算等多种模型都可以有一个好的描述。

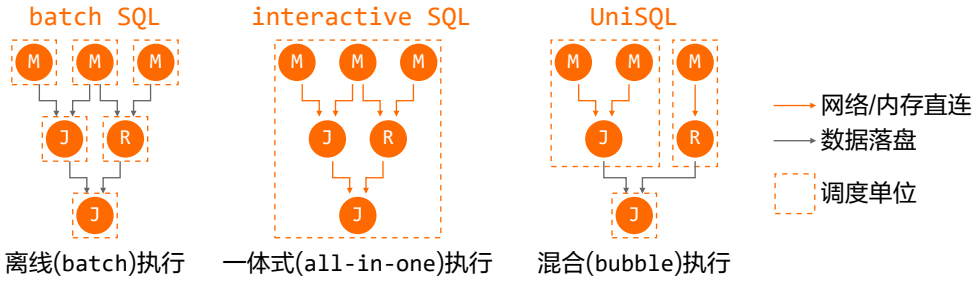
如果我们对分布式 SQL 进行细分的话，可以看见业界对于不同场景上的优化经常走在两个极端：要么优化 throughput (大规模，相对高延时)，要么优化 latency(中小数据量，迅速完成)。前者以 Hive 为典型代表，后者则以 Spark 以及各种分布式 MPP 解决方案为代表。而在阿里分布式系统的发展过程中，历史上同样出现了两种对比较为显著的执行方式：SQL 线离线（batch）作业与准实时（interactive）作业。这两种模式的资源管理和作业执行，过去是搭建在两套完全分开的代码实现上的。这除了导致两套代码和功能无法复用以外，两种计算模式的非黑即白，使得彼此

在资源利用率和执行性能之间无法 tradeoff。而在 DAG 2.0 模型上，通过对点/边物理特性的映射，实现了这两种计算模式比较自然的融合和统一。离线作业和准实时作业在逻辑节点和逻辑边上映射不同的物理特性后，都能得到准确的描述:

- 离线作业：每个节点按需去申请资源，一个逻辑节点代表一个调度单位；节点间连接边上传输的数据，通过落盘的方式来保证可靠性；

- 准实时作业：整个作业的所有节点都统一在一个调度单位内进行 gang scheduling；节点间连接边上通过网络/内存直连传输数据，并利用数据 pipeline 来追求最优的性能。

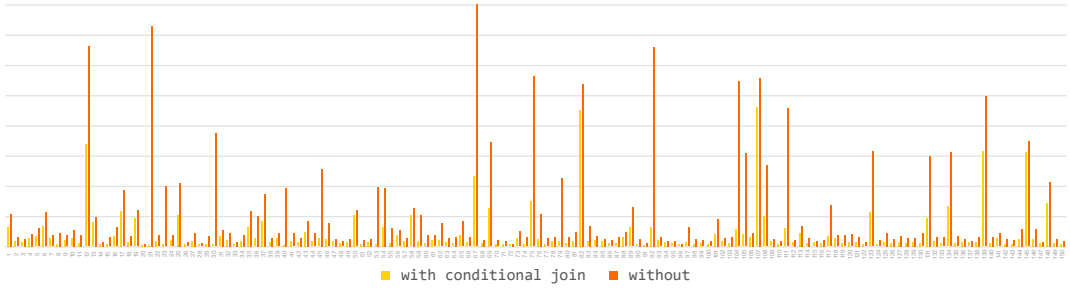
在统一离线作业与准实时作业到一套架构的基础上，这种统一的描述方式，使得探索离线作业高资源利用率，以及准实时作业的高性能之间的 tradeoff 成为可能：当调度单位可以自由调整，就可以实现一种全新的混合的计算模式，我们称之为 Bubble 执行模式。



这种混合 Bubble 模式，使得 DAG 的用户，也就是上层计算引擎的开发者（比如 MaxCompute 的优化器），能够结合执行计划的特点，以及引擎终端用户对资源使用和性能的敏感度，来灵活选择在执行计划中切出 Bubble 子图。在 Bubble 内部充分利用网络直连和计算节点预热等方式提升性能，没有切入 Bubble 的节点则依然通过传统离线作业模式运行。在统一的新模型之上，计算引擎和执行框架可以在两个极端之间，根据具体需要，选择不同的平衡点。

效果

DAG 2.0 的动态性使得很多执行优化可以运行时决定，使得实际执行的效果更优。例如，在阿里内部的作业中，动态的 conditional join 相比静态的执行计划，整体获得了将近 3X 的性能提升。



混合 Bubble 执行模式平衡了离线作业高资源利用率以及准实时作业的高性能，这在 1TB TPCH 测试集上有显著的体现，

- Bubble 相对离线作业：在多使用 20% 资源的情况下，Bubble 模式性能提升将近一倍；

- Bubble 相对准实时模式：在节省了 2.6X 资源情况下， Bubble 性能仅下降 15%。

1-4-2. Fuxi Shuffle 2.0 - 磁盘内存网络的最佳使用

背景

大数据计算作业中，节点间的数据传递称为 shuffle, 主流分布式计算系统都提供了数据 shuffle 服务的子系统。如前述 DAG 计算模型中，task 间的上下游数据传输就是典型的 shuffle 过程。

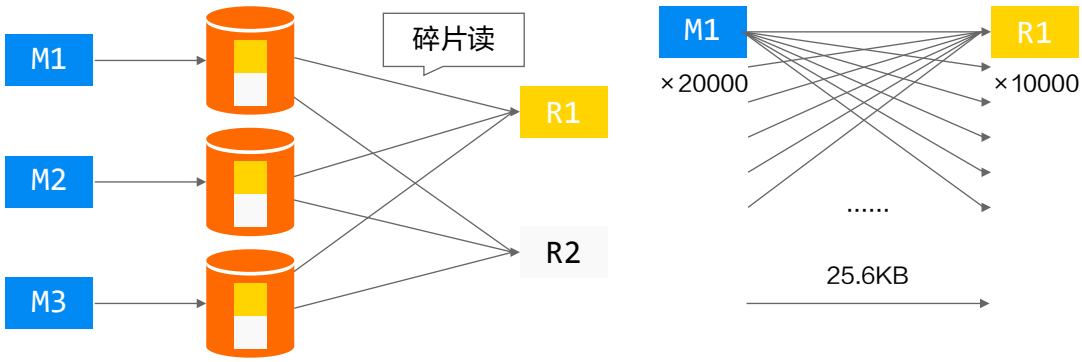
在数据密集型作业中，shuffle 阶段的时间和资源使用占比非常高，有其他大数据公司研究显示，在大数据计算平台上 Shuffle 阶段均是在所有作业的资源使用中占比超过 50%. 根据统计在 MaxCompute 生产中 shuffle 占作业运行时间和资源消耗的 30-70%，因此优化 shuffle 流程不但可以提升作业执行效率，而且可以整体上降低资源使用，节约成本，提升 MaxCompute 在云计算市场的竞争优势。

从 shuffle 介质来看，最广泛使用的 shuffle 方式是基于磁盘文件的 shuffle. 这种模式这种方式简单，直接，通常只依赖于底层的分布式文件系统，适用于所有类型作业。而在典型的常驻内存的实时/准实时计算中，通常使用网络直连 shuffle 的方式追求极致性能。Fuxi Shuffle 在 1.0 版本中将这两种 shuffle 模式进行了极致优化，保障了日常和高峰时期作业的高效稳定运行。

挑战

我们先以使用最广泛的，基于磁盘文件系统的离线作业 shuffle 为例。

通常每个 mapper 生成一个磁盘文件，包含了这个 mapper 写给下游所有 reducer 的数据。而一个 reducer 要从所有 mapper 所写的文件中，读取到属于自己的那一小块。右侧则是一个系统中典型规模的 MR 作业，当每 个mapper 处理 256MB 数据，而下游 reducer 有 10000 个时，平均每个 reducer 读取来自每个 mapper 的数据量就是 25.6KB, 在机械硬盘 HDD 为介质的存储系统中，属于典型的读碎片现象，因为假设我们的磁盘 iops 能达到 1000, 对应的 throughput 也只有 25MB/s, 严重影响性能和磁盘压力。



【基于文件系统 shuffle 的示意图 / 一个 20000*10000 的 MR 作业的碎片读】

分布式作业中并发度的提升往往是加速作业运行的最重要手段之一。但处理同样的数据量，并发度越高意味着上述碎片读现象越严重。通常情况下选择忍受一定的碎片 IO 现象而在集群规模允许的情况下提升并发度，还是更有利于作业的性能。所以碎片 IO 现象在线上普遍存在，磁盘也处于较高的压力水位。

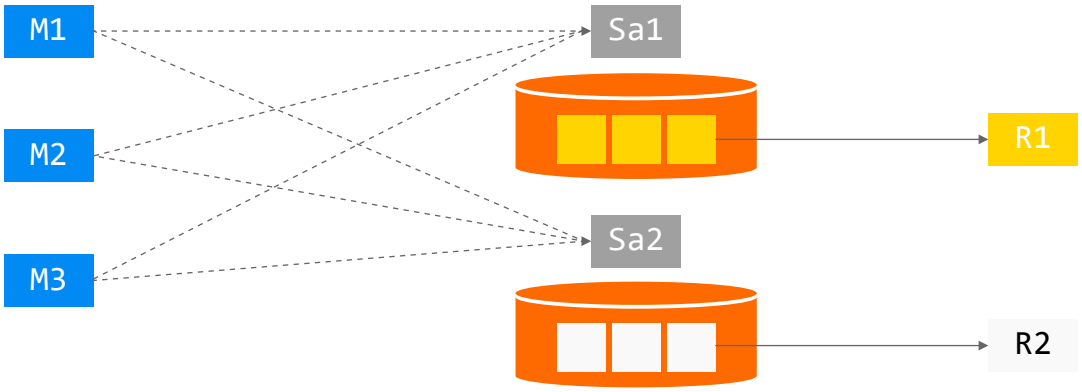
一个线上的例子是，某些主流集群单次读请求 size 为 50-100KB, Disk util 指标长期维持在 90% 的警戒线上。这些限制了对作业规模的进一步追求。

我们不禁考虑，作业并发度和磁盘效率真的不能兼得吗?

Fuxi 的答案：Fuxi Shuffle 2.0

引入 Shuffle Service - 高效管理 shuffle 资源

为了针对性地解决上述碎片读问题及其引发的一连串负面效应，我们全新打造了基于 shuffle service 的 shuffle 模式。Shuffle service 的最基本工作方式是，在集群每台机器部署一个 shuffle agent 节点，用来归集写给同一 reducer 的 shuffle 数据。
如下图



可以看到，mapper 生成 shuffle 数据的过程变为mapper将shuffle数据通过网络传输给每个 reducer 对应的 shuffle agent, 而 shuffle agent 归集一个 reducer 来自所有 mapper 的数据，并追加到 shuffle 磁盘文件中，两个过程是流水线并行化起来的。

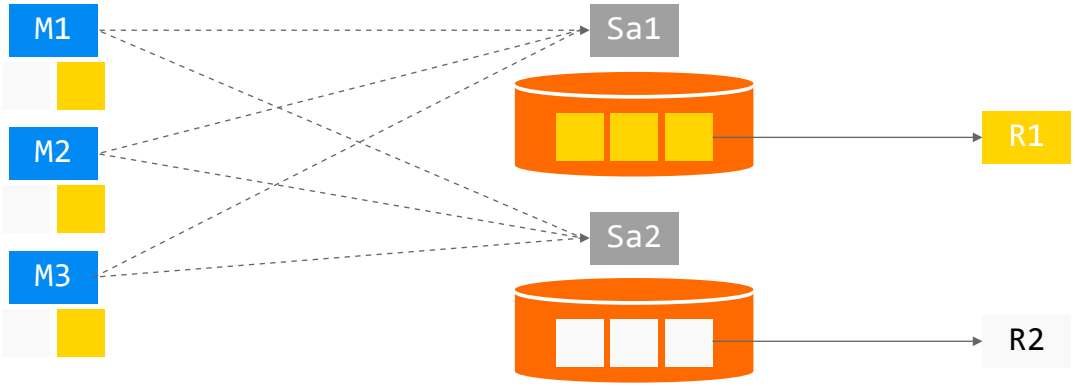
Shuffle agent 的归集功能将 reducer 的 input 数据从碎片变为了连续数据文件，对 HDD 介质相当友好。由此，整个 shuffle 过程中对磁盘的读写均为连续访问。从标准的 TPCH 等测试中可以看到不同场景下性能可取得百分之几十到几倍的提升，且大幅降低磁盘压力、提升 CPU 等资源利用率。

Shuffle Service 的容错机制

Shuffle service 的归集思想在公司内外都有不同的工作展现类似的思想，但都限于“跑分”和小范围使用。因为这种模式对于各环节的错误天生处理困难。

以 shuffle agent 文件丢失/损坏是大数据作业的常见问题为例，传统的文件系统 shuffle 可以直接定位到出错的数据文件来自哪个 mapper，只要重跑这个 mapper 即可恢复。但在前述 shuffle service 流程中，由于 shuffle agent 输出的 shuffle 这个文件包含了来自所有 mapper 的 shuffle 数据，损坏文件的重新生成需要以重跑所有 mapper 为代价。如果这种机制应用于所有线上作业，显然是不可接受的。

我们设计了数据双副本机制解决了这个问题，使得大多数通常情况下 reducer 可以读取到高效的 agent 生成的数据，而当少数 agent 数据丢失的情况，可以读取备份数据，备份数据的重新生成只依赖特定的上游 mapper。



具体来说，mapper 产生的每份 shuffle 数据除了发送给对于 shuffle agent 外，也会按照与传统文件系统 shuffle 数据类似的格式，在本地写一个备份。按前面所述，这份数据写的代价较小但读取的性能不佳，但由于仅在 shuffle agent 那个副本出错时才会读到备份数据，所以对作业整体性能影响很小，也不会引起集群级别的磁盘压力升高。

有效的容错机制使得 shuffle service 相对于文件系统 shuffle，在提供更好的作业性能的同时，因 shuffle 数据出错的 task 重试比例降低了一个数量级，给线上全面投入使用打好了稳定性基础。

线上生产环境的极致性能稳定性

在前述基础功能之上，Fuxi 线上的 shuffle 系统应用了更多功能和优化，在性能、成本、稳定性等方便取得了进一步的提升。举例如下。

（1）流控和负载均衡

前面的数据归集模型中，shuffle agent 作为新角色衔接了 mapper 的数据发送与数据落盘。分布式集群中磁盘、网络等问题可能影响这条链路上的数据传输，节点本身的压力也可能影响 shuffle agent 的工作状态。当因集群热点等原因使得 shuffle agent 负载过重时，我们提供了必要的流控措施缓解网络和磁盘的压力；和模型中一个

reducer 有一个 shuffle agent 收集数据不同，我们使用了多个 shuffle agent 承担同样的工作，当发生数据倾斜时，这个方式可以有效地将压力分散到多个节点上。从线上表现看，这些措施消除了绝大多数的 shuffle 期间拥塞流控和集群负载不均现象。

（2）故障 shuffle agent 的切换

各种软硬件故障导致 shuffle agent 对某个 reducer 的数据工作不正常时，后续数据可以实时切换到其他正常 shuffle agent. 这样，就会有更多的数据可以从 shuffle agent 侧读到，而减少低效的备份副本访问。

（3）Shuffle agent 数据的回追

很多时候发生 shuffle agent 切换时（如机器下线），原 shuffle agent 生成的数据可能已经丢失或访问不到。在后续数据发送到新的 shuffle agent 同时，Fuxi 还会将丢失的部分数据从备份副本中 load 起来并同样发送给新的 shuffle agent, 使得后续 reducer 所有的数据都可以读取自 shuffle agent 侧，极大地提升了容错情况下的作业性能。

（4）新 shuffle 模式的探索

前述数据归集模型及全面扩展优化，在线上集群中单位资源处理的数据量提升了约 20%, 而因出错重试的发生频率降至原来文件系统 shuffle 的 5% 左右。但这就是最高效的 shuffle 方式了吗?

我们在生产环境对部分作业应用了一种新的 shuffle 模型，这种模型中 mapper 的发送端和 reducer 的接收端都通过一个 agent 节点来中转 shuffle 流量。线上已经有部分作业使用此种方式并在性能上得到了进一步的提升。

内存数据 shuffle

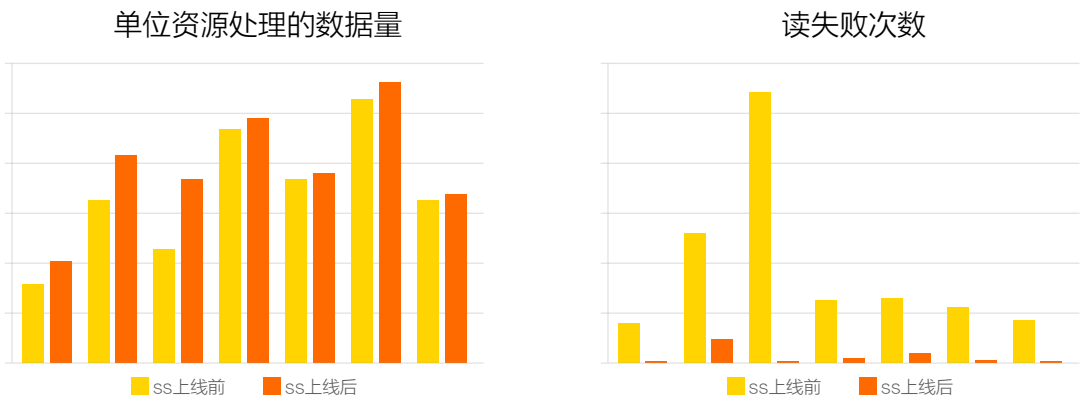
离线大数据作业可能承担了主要的计算数据量，但流行的大数据计算系统中有非常多的场景是通过实时/准实时方式运行的，作业全程的数据流动发生在网络和内存，从而在有限的作业规模下取得极致的运行性能，如大家熟悉的 Spark, Flink 等系统。

Fuxi DAG 也提供了实时/准实时作业运行环境，传统的 shuffle 方式是通过网络直连，也能收到明显优于离线 shuffle 的性能。这种方式下，要求作业中所有节点都要调度起来才能开始运行，限制了作业的规模。而实际上多数场景计算逻辑生成 shuffle 数据的速度不足以填满 shuffle 带宽，运行中的计算节点等待数据的现象明显，性能提升付出了资源浪费的代价。

我们将 shuffle service 应用到内存存储中，以替换 network 传输的 shuffle 方式。一方面，这种模式解耦了上下游调度，整个作业不再需要全部节点同时拉起；另一方面通过精确预测数据的读写速度并适时调度下游节点，可以取得与 network 传输 shuffle 相当的作业性能，而资源消耗降低 50% 以上。这种 shuffle 方式还使得 DAG 系统中多种运行时调整 DAG 的能力可以应用到实时/准实时作业中。

收益

Fuxi Shuffle 2.0 全面上线生产集群，处理同样数据量的作业资源比原来节省 15%，仅 shuffle 方式的变化就使得磁盘压力降低 23%，作业运行中发生错误重试的比例降至原来的 5%。



【线上典型集群的性能与稳定性提升示意图（不同组数据表示不同集群）】

对使用内存 shuffle 的准实时作业，我们在 TPCH 等标准测试集中与网络 shuffle 性能相当，资源使用只有原来的 30% 左右，且支持了更大的作业规模，此外也令 DAG 2.0 系统更多的动态调度功能应用至准实时作业。

1-5. 单机调度 – 内核层面的隔离机制

大量分布式作业汇集到一台机器上，如何将单机有限的各种资源合理分配给每个作业使用，从而达到作业运行质量、资源利用率、作业稳定性的多重保障，是单机调度要解决的任务。

典型的互联网公司业务一般区分为离线业务与在线业务两种类型。在阿里巴巴，我们也同样有在线业务如淘宝、天猫、钉钉、Blink 等，这类业务的特点是对响应延迟特别敏感，一旦服务抖动将会出现添加购物车失败、下单失败、浏览卡顿、钉钉消息发送失败等各种异常情况，严重影响用户体验，同时为了应对在 618、双 11 等各种大促的情况，需要提前准备大量的机器。由于以上种种原因，日常状态这些机器的资源利用率不足 10%，产生资源浪费的情况。与此同时，阿里的离线业务又是另外一幅风景，MaxCompute 计算平台承担了阿里所有大数据离线计算业务类型，各个集群资源利用率常态超负载运行，数据量和计算量每年都在保持高速增长。

一方面是在线业务资源利用率不足，另一方面是离线计算长期超负载运行，那么能否将在线业务与离线计算进行混合部署，提升资源利用率同时大幅降低成本，实现共赢?

1-5-1. 三大挑战

1. 如何保障在线服务质量

在线集群的平均 CPU 利用率只有 10% 左右，混部的目标就是将剩余的资源提供给 MaxCompute 进行离线计算使用，从而达到节约成本的目的。那么，如何能够保障资源利用率提升的同时又能够保护在线服务不受影响呢?

2. 如何保障离线稳定

当资源发生冲突时，第一反应往往是保护在线，牺牲离线。毕竟登不上淘宝天猫下不了单可是大故障。可是，离线如果无限制的牺牲下去，服务质量将会出现大幅度下降。试想，我在 DataWorks 上跑个 SQL，之前一分钟就出结果，现在十几分钟甚至一个小时都跑不出来，大数据分析的同学估计也受不了了。

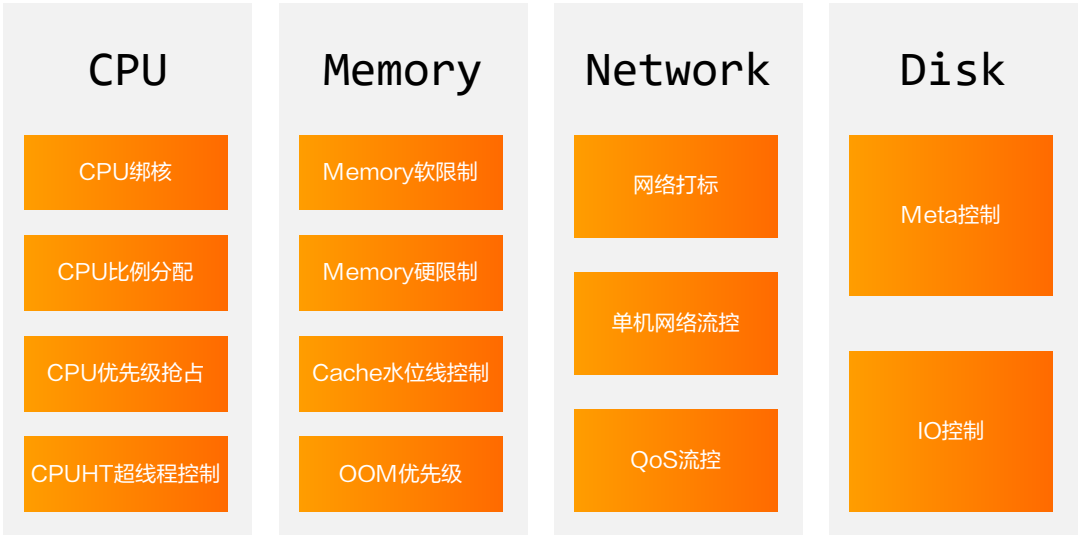
3. 如何衡量资源质量

电商业务通过富容器的方式集成多种容器粒度的分析手段，但是前文描述过离线作业的特点，如何能够精准的对离线作业资源使用进行资源画像分析，如何能够评估资源受干扰的程度，混部集群的稳定性等问题，是对我们的又一个必须要解决的挑战。

1-5-2. 资源隔离分级管理

单机的物理资源总是有限的，按照资源特性可以大体划分为可伸缩资源与不可伸缩资源两大类。CPU、Net、IO 等属于可伸缩资源，Memory 属于不可伸缩资源，不同类型的资源有不同层次的资源隔离方案。另一方面，通用集群中作业类型种类繁多，不同作业类型对资源的诉求是不同的。这里包括在线、离线两个大类的资源诉求，同时也包含了各自内部不同层次的优先级二次划分需求，十分复杂。

基于此，Fuxi2.0 提出了一套基于资源优先级的资源划分逻辑，在资源利用率、多层次资源保障复杂需求寻找到了解决方案。



下面我们将针对 CPU 分级管理进行深入描述，其他维度资源管理策略我们将在今后的文章中进行深入介绍。

CPU 分级管理

通过精细的组合多种内核策略，将 CPU 区分为高、中、低三类优先级

CPU 等级	资源隔离策略
金	金牌服务独占CPU核与银牌CPU核不重叠与铜牌CPU重叠，通过隔离机制抢占铜牌CPU
银	银牌服务不独占CPU核，通过share共享CPU与金牌CPU核不重叠，与铜牌CPU重叠，通过隔离机制抢占铜牌CPU
铜	削峰填谷使用CPU核与金牌、铜牌CPU重叠随时被隔离机制抢占

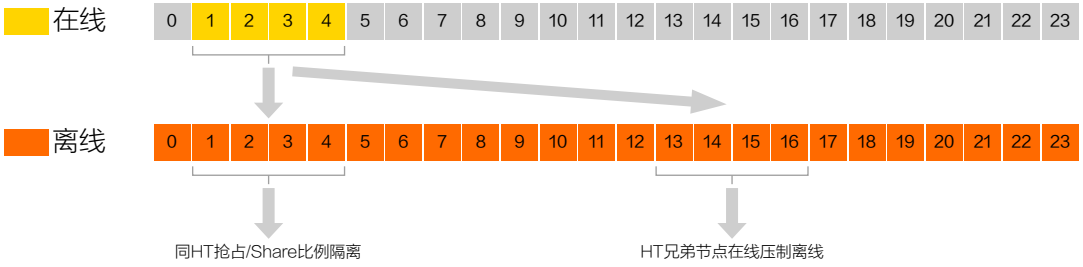
隔离策略如下图所示

- CPU 拓扑结构
  （以 24 核示例）

CPU Socket1		
物理核	超线程HT分布	
Core ID	HT ID	HT ID
0	0	12
1	1	13
2	2	14
3	3	15
4	4	16
5	5	17

CPU Socket2		
物理核	超线程 HT 分布	
Core ID	HT ID	HT ID
0	6	18
1	7	19
2	8	20
3	9	21
4	10	22
5	11	23

- CPU 隔离



基于不同类型的资源对应不同的优先级作业

调度系统	金	银	铜
在线	延迟极度敏感类型服务	大部分普通在线服务	无
离线	无	离线 CPU 敏感作业	离线普通作业

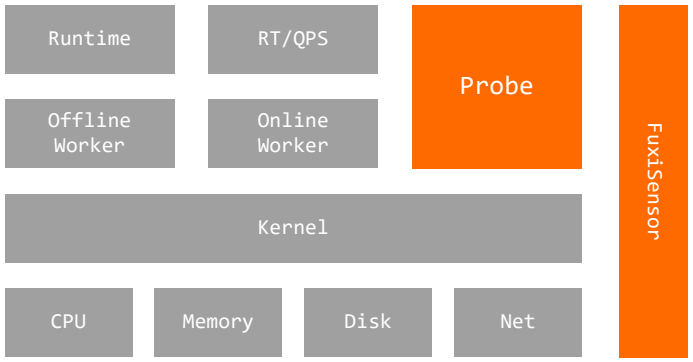
1-5-3. 资源画像

Fuxi 作为资源调度模块，对资源使用情况的精准画像是衡量资源分配，调查/分析/解决解决资源问题的关键。针对在线作业的资源情况，集团和业界都有较多的解决方案。这类通用的资源采集角色存在以下无法解决的问题无法应用于离线作业资源画像的数据采集阶段。

1. 采集时间精度过低。大部分信息是分钟级别，而 MaxCompute 作业大部分运行时间在秒级；

2. 无法定位 MaxCompute 信息。MaxCompute 是基于 Cgroup 资源隔离，因此以上工具无法针对作业进行针对性采集；

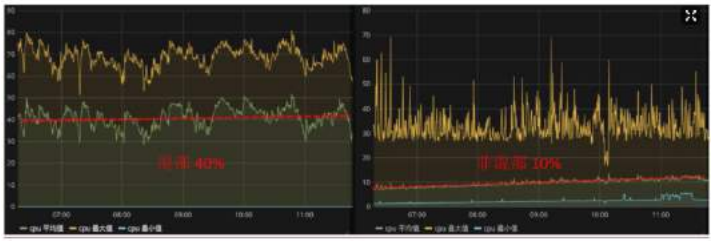
3. 采集指标不足。有大量新内核新增的微观指标需要进行收集，过去是不支持的。



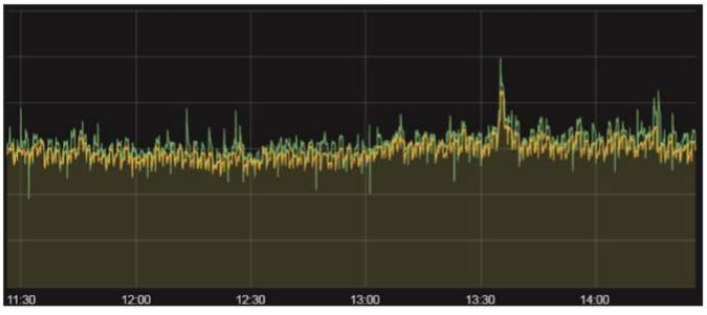
为此，我们提出了 FuxiSensor 的资源画像方案，架构如上图所示，同时利用 SLS 进行数据的收集和分析。在集群、Job 作业、机器、worker 等不同层次和粒度实现了资源信息的画像，实现了秒级的数据采集精度。在混部及 MaxCompute 的实践中，成为资源问题监控、报警、稳定性数据分析、作业异常诊断、资源监控状况的统一入口，成为混部成功的关键指标。

1-5-4. 线上效果

日常资源利用率由 10% 提升到 40% 以上



在线抖动小于 5%



1-5-5. 单机调度小结

为了解决三大挑战，通过完善的各维度优先级隔离策略，将在线提升到高优先级资源维度，我们保障了在线的服务质量稳定；通过离线内部优先级区分及各种管理策略，实现了离线质量的稳定性保障；通过细粒度资源画像信息，实现了资源使用的评估与分析，最终实现了混部在阿里的大规模推广与应用，从而大量提升了集群资源利用率，为离线计算节省了大量成本。

1-6. 面向未来的伏羲系统

从 2009 到 2019 年历经十年的锤炼，伏羲系统仍然在不断的演化，满足不断涌现的业务新需求，引领分布式调度技术的发展。接下来，我们会从以下几个方面继续创新：

- 资源调度 FuxiMaster 将基于机器学习，实现智能化调度策略和动态精细的资源管理模式，进一步提高集群资源利用率，提供更强大灵活的分布式集群资源管理服务；

- 新一代 DAG2.0 继续利用动态性精耕细作，优化各种不同类型的作业；与 SQL 深入合作，解决线上痛点，推动 SQL 引擎深度优化，提升性能的同时也让 SQL 作业运行更加智能化；探索机器学习场景的 DAG 调度，改善训练作业的效率，提升 GPU 使用率；

- 数据 Shuffle 2.0 则一方面优化 shuffle 流程，追求性能、成本、稳定性的极致，另一方面与 DAG 2.0 深入结合，提升更多场景；同时探索新的软硬件架构带来的新的想象空间；

- 智能化的精细单机资源管控，基于资源画像信息通过对历史数据分析产生未来趋势预测，通过多种资源管控手段进行精准的资源控制，实现资源利用率和不同层次服务质量的完美均衡。

“伏羲”3 大核心调度系统技术解读

“伏羲”3 大核心调度系统技术解读

2-1. EB 级计算平台调度系统伏羲 DAG 2.0

本文作者 CHEN, Yingda | 阿里云智能 高级技术专家

作为阿里巴巴核心大数据底座，伏羲调度和分布式执行系统，支撑着阿里集团内部以及阿里云上大数据平台绝大部分的大数据计算需求，在其上运行的 MaxCompute (ODPS) 以及 PAI 等多种计算引擎，每天为用户进行海量的数据运算。 在"阿里体量"的大数据生态中，伏羲系统管理着弹内外多个物理集群，超十万台物理机，以及数百万的 CPU/GPU cores。每天运行在伏羲分布式平台上的作业数已经超过千万，是业界少有的单天处理 EB 级别数据的分布式平台。其中单个作业规模已经高达数十万计算节点，管理着数百亿的边连接。在过去的十年中，阿里集团以及阿里云上这样的作业数目和规模，锤炼了伏羲分布式平台；与此同时，今天平台上作业的日益多样化，以及向前再发展的需求，对于伏羲系统架构的进一步演化，也都带来了巨大挑战与机遇。本文主要介绍一下在过去的两年多时间中，阿里巴巴伏羲团队对于整个核心调度与分布式执行系统的升级换代，code name DAG 2.0。

2-1-1. 构建更动态更灵活的分布式计算生态

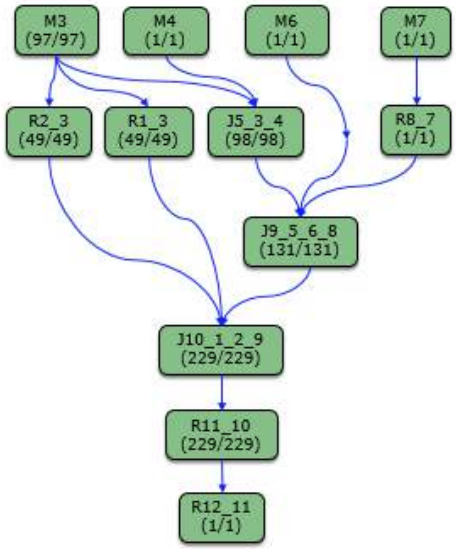
伏羲 DAG/AM 组件

从较高的层面来看整个分布式系统的体系架构，物理集群之上运行的分布式系统，大概可以分成资源管理，作业分布式调度执行，与多个计算节点的运行这三个层次，如同下图所示。通常所说的 DAG 组件，指的是每个分布式作业的中心管理点，也就是 application master (AM)。AM 之所以经常被称为 DAG (Directional Acyclic Graph，有向无环图) 组件，是因为AM最重要的责任，就是负责协调分布式作业的执行。而现代的分布式系统中的作业执行流程，通常可以通过 DAG 上面的调度以及数据流来描述。相对于传统的 Map-Reduce 执行模式，DAG 的模型能对分布式作业做更精准的描述，也是当今各种主流大数据系统（Hadoop 2.0+, SPARK, FLINK, TENSORFLOW 等）的设计架构基础，区别只在于 DAG 的语义是透露给终端用户，还是计算引擎开发者。

与此同时，从整个分布式系统 stack 来看， AM肩负着除了运行 DAG 以外更多的责任。作为作业的中心管控节点，向下其负责与 Resource Manager 之间的交互，为分布式作业申请计算资源；向上其负责与计算引擎进行交互，并将收集的信息反馈到 DAG 的执行过程中。作为唯一有能力对每一个分布式作业的执行大局有最精准的了解的组件，在全局上对 DAG 的运行做准确的管控和调整，也是 AM 的重要职责。从上图描述的分布式系统 stack 图中，我们也可以很直观的看出，AM 是系统中唯一需要和几乎所有分布式组件交互的组件，在作业的运行中起了重要的承上启下的作用。这一组件之前在伏羲系统中被称为 JobMaster (JM), 在本文中我们统一用 DAG 或者 AM 来指代。

逻辑图与物理图

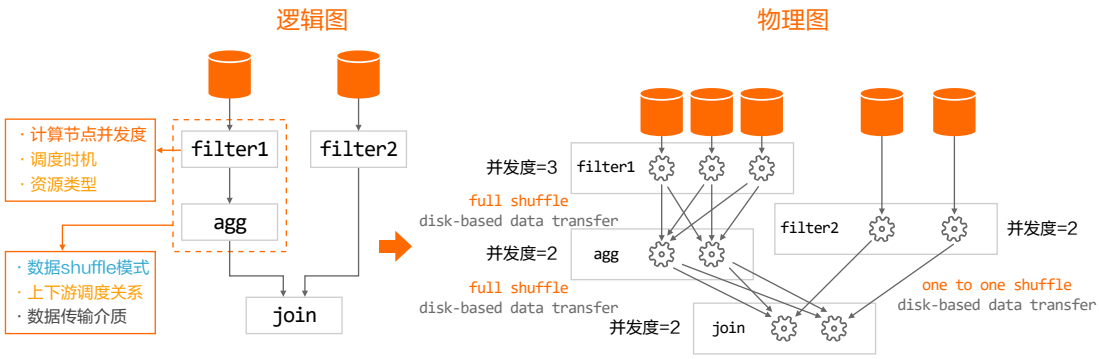
分布式作业的 DAG，有两种层面上的表述：逻辑图与物理图。简单地来说 (over-simplified)，终端用户平时理解的 DAG 拓扑，大多数情况下描述的是逻辑图范畴：比如大家平时看到的 logview 图，虽然里面包含了一些物理信息（每个逻辑节点的并发度），但整体上可以认为描述的就是作业执行流程的逻辑图。



准确一点说：

- 逻辑图描述了用户想要实现的数据处理流程，从数据库 /SQL 的角度(其他类型引擎也都有类似之处，比如 TENSORFLOW) 来看，可以大体认为 DAG 的逻辑图，是对优化器执行计划的一个延续；

- 物理图更多描述了执行计划映射到物理分布式集群的具体描述，体现的是执行计划被物化到分布式系统上，具备的一些特性：比如并发度，数据传输方式等等。

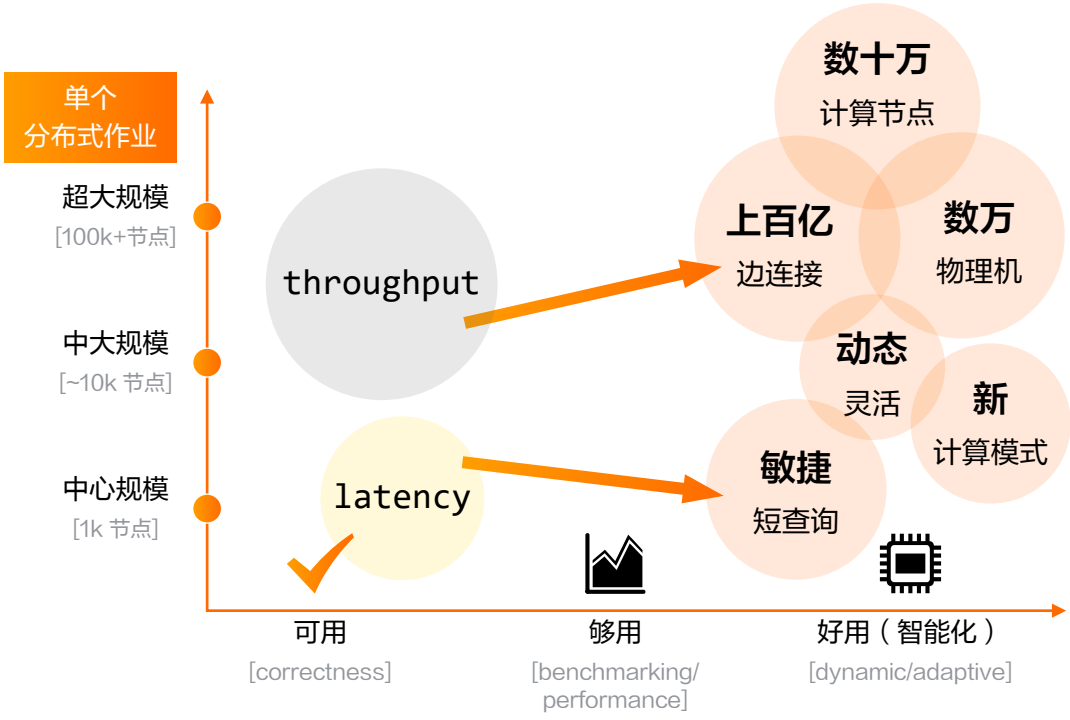


而每个逻辑图的“物理化”，可以有很多等效方式。选择合适的方式来将逻辑图变成物理化执行，并进行灵活的调整，是 DAG 组件的重要职责之一。从上图的逻辑图到物理图的映射可以看到，一个图的物理化过程，实际上就是在回答一系列图节点以及各个连接边物理特性的问题，一旦这些问题得到确认，就能得到在分布式系统上实际执行的物理图。

为什么需要 DAG 2.0 架构升级?

作为从阿里云飞天系统创建伊始就开始研发的伏羲分布式作业执行框架，DAG 1.0 在过去十年中支撑了阿里集团的大数据业务，在系统规模以及可靠性等方面都走在了业界领先。另外一方面，作为一个开发了十年的系统，虽然在这个期间不断的演进，DAG 1.0 在基本架构上秉承了比较明显的 Map-Reduce 执行框架的一些特点，逻辑图和物理图之间没有清晰的分层，这导致在这个基本架构上要继续向前走，支持更多 DAG 执行过程中的动态性，以及同时支持多种计算模式等方面，都比较困难。事实上今天在 MaxCompute SQL 线上，离线作业模式以及准实时作业模式（smode）两种执行模式，使用了两套完全分开的分布式执行框架，这也导致对于优化性能和优化系统资源使用之间的取舍，很多情况下只能走两个极端，而无法比较好的 tradeoff。

除此之外，随着 MaxCompute 以及 PAI 引擎的更新换代以及新功能演进，上层的分布式计算自身能力在不断的增强。对于AM组件在作业管理，DAG 执行等方面的动态性，灵活性等方面的需求也日益强烈。在这样的一个大的背景下，为了支撑计算平台下个 10 年的发展，伏羲团队启动了 DAG 2.0 的项目，将从代码和功能方面，完整替代 1.0 的 JobMaster 组件，实现完全的升级换代。在更好的支撑上层计算需求的同时，也同时对接伏羲团队在 shuffle 服务（shuffle service）上的升级，以及 Fuxi master (Resource Manager) 的功能升级。与此同时，站在提供企业化服务的角度来看，一个好的分布式执行框架，除了支持阿里内部极致的大规模大吞吐作业之外，我们需要支持计算平台向外走，支持云上各种规模和计算模式的需求。除了继续锤炼超大规模的系统扩展能力以外，我们需要降低大数据系统使用的门槛，通过系统本身的智能动态化能力，来提供自适应（各种数据规模以及处理模式）的大数据企业界服务，是 DAG 2.0 在设计架构中考虑的另一重要维度。

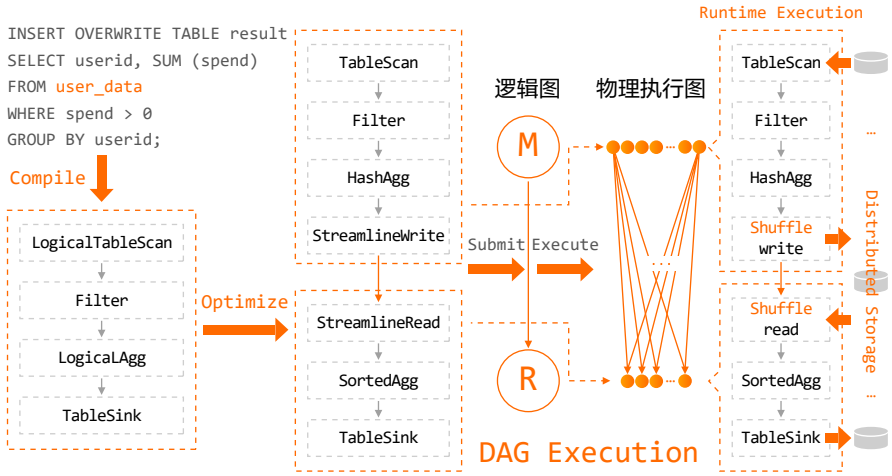


2-1-2. DAG 2.0 架构以及整体设计

DAG 2.0 项目，在调研了业界各个分布式系统（包括 SPARK/FLINK/Dryad/-Tez/Tensorlow）DAG 组件之后，参考了 Dryad/Tez 的框架。新一代的架构上，通过逻辑图和物理图的清晰分层，可扩展的状态机管理，插件式的系统管理，以及基于事件驱动的调度策略等基座设计，实现了对计算平台上多种计算模式的统一管理，并更好的提供了作业执行过程中在不同层面上的动态调整能力。

作业执行的动态性

传统的分布式作业执行流程，作业的执行计划是在提交之前确定的。以 SQL 执行为例，一个 SQL 语句，在经过编译器和优化器后产生执行图，并被转换成分布式系统（伏羲）的执行计划。



这个作业流程在大数据系统中是比较标准的操作。然而在具体实现中，如果在 DAG 的执行缺乏自适应动态调整能力的话，整个执行计划都需要事先确定，会使得作业的运行没有太多动态调整的空间。放在 DAG 的逻辑图与物理图的背景中来说，这要求框架在运行作业前，必须事先了解作业逻辑和处理数据各种特性，并能够准确回答作业运行过程，各个节点和连接边的物理特性问题，来实现逻辑图往物理图的转换。

然而在现实情况中，许多物理特性相关的问题，在作业运行前是无法被感知的。以数据特性为例，一个分布式作业在运行前，能够获得的只有原始输入的一些特性（数据量等）， 对于一个较深的 DAG 执行而言，这也就意味着只有根节点的物理计划（并发度选择等）是相对合理的，而下游的节点和边的物理特性只能通过一些特定的规则来猜测。虽然在输入数据有丰富的 statistics 的前提下，优化器有可能可以将这些 statistics，与执行 plan 中的各个 operator 特性结合起来，进行一些适度的演算：从而推断在整个执行流程中，每一步产生的中间数据可能符合什么样的特性。但这种推断在实现上，尤其在面对阿里大体量的实际生产环境中，面临着巨大的挑战，例如：

- 实际输入数据的 statistics 的缺失：即便是 SQL 作业处理的结构化数据，也无法保证其源表数据特性拥有很好的统计。事实上今天因为数据落盘方式多样化，以及

精细化统计方式的缺失，大部分的源表数据都是没有完整的 statistics 的。此外对于集群内部和外部需要处理的非结构化数据，数据的特性的统计更加困难；

- 分布式作业中存在的大量用户逻辑黑盒：作为一个通用的大数据处理系统，不可避免的需要支持用户逻辑在系统中的运行。比如 SQL 中常用的 UDF/UDTF/UDJ/Extractor/Outputer 等等，这些使用 Java/Python 实现的用户逻辑，计算引擎和分布式系统并无法理解，在整个作业流程中是类似黑盒的存在。以 MaxCompute 为例，线上有超过 20% 的 SQL 作业，尤其是重点基线作业，都包含用户代码。这些大量用户代码的存在，也造成了优化器在很多情况下无法对中间产出数据的特性进行预判；

- 优化器预判错误代价昂贵：在优化器选择执行计划时，会有一些优化方法，在数据符合一定特殊特性的时候，被合理选中能带来性能优化。但是一旦选择的前提假设错误（比如数据特性不符合预期），会适得其反，甚至带来严重的性能回退或作业失败。在这种前提下，依据静态的信息实现进行过多的预测经常得不到理想的结果。

这种种原因造成的作业运行过程中的非确定性，要求一个好的分布式作业执行系统，需要能够根据中间运行结果的特点，来进行执行过程中的动态调整。因为只有在中间数据已经在执行过程中产生后，其数据特性才能被最准确的获得，动态性的缺失，可能带来一系列的线上问题，比如：

- 物理资源的浪费：比如计算节点事先选择的资源类型的不合理，或者大量的计算被消耗用于处理后继会被丢弃的无效数据；

- 作业的严重长尾：比如中间数据分布倾斜或不合理编排，导致一个 stage 上计算节点需要处理的数据量极端化；

- 作业的不稳定：比如由于优化器静态计划的错判，导致不合理的执行计划无法完成。

而 DAG/AM 作为分布式作业唯一的中心节点和调度管控节点，是唯一有能力收集并聚合相关数据信息，并基于这些数据特性来做作业执行的动态调整的分布式组件。这包括简单的物理执行图调整（比如动态的并发度调整），也包括复杂一点的调整比如对 shuffle 方式和数据编排方式重组。除此以外，数据的不同特点也会带来逻辑执行图调整的需求：对于逻辑图的动态调整，在分布式作业处理中是一个全新的方向，也是我们在 DAG 2.0 里面探索的新式解决方案。

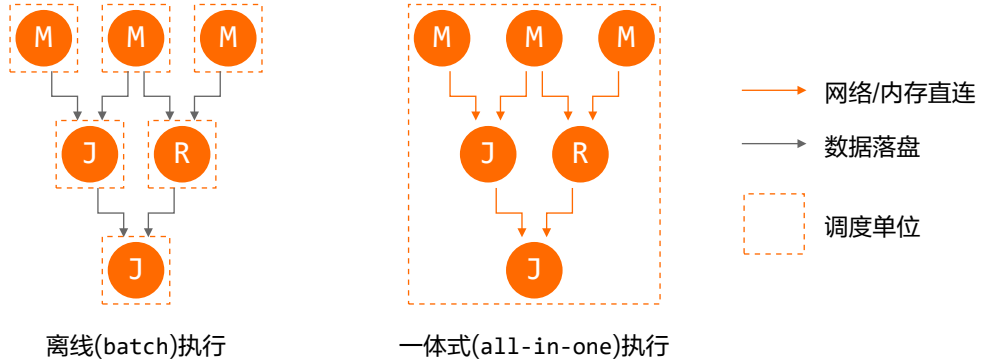
点，边，图的清晰物理逻辑分层，和基于事件的数据收集和调度管理，以及插件式的功能实现，方便了 DAG 2.0 在运行期间的数据收集，以及使用这些数据来系统性地回答，逻辑图向物理图转化过程中需要确定的问题。从而在必要的时候实现物理图和逻辑图的双重动态性，对执行计划进行合理的调整。在下文中提到几个落地场景中，我们会进一步举例说明基于 2.0 的这种强动态性能力，实现更加自适应，更加高效的分布式作业的执行。

统一的 AM/DAG 执行框架

DAG 2.0 抽象分层的点，边，图架构上，也使其能通过对点和边上不同物理特性的描述，对接不同的计算模式。业界各种分布式数据处理引擎，包括 SPARK, FLINK, HIVE，SCOPE，TENSORFLOW 等等，其分布式执行框架的本源都可以归结于 Dryad 提出的 DAG 模型。我们认为对于图的抽象分层描述，将允许在同一个 DAG 系统中，对于离线/实时/流/渐进计算等多种模型都可以有一个好的描述。在 DAG 2.0 初步落地的过程中，首要目标是在同一套代码和架构系统上，统一当前伏羲平台上运行的几种计算模式，包括 MaxCompute 的离线作业，准实时作业，以及 PAI 平台上的 Tensorflow 作业和其他的非 SQL 类作业。对更多新颖计算模式的探索，也会有计划的分步骤进行。

统一的离线作业与准实时作业执行框架

首先我们来看平台上作业数占到绝大多数的 SQL 线离线作业 (batch job) 与准实时作业 (smode)。前面提到过，由于种种历史原因，之前 MaxCompute SQL 线的这两种模式的资源管理和作业执行，是搭建在两套完全分开的代码实现上的。这除了导致两套代码和功能无法复用以外，两种计算模式的非黑即白，使得彼此在资源利用率和执行性能之间无法 tradeoff。而在 2.0 的 DAG 模型上，我们实现了这两种计算模式比较自然的融合和统一，如下图所示：

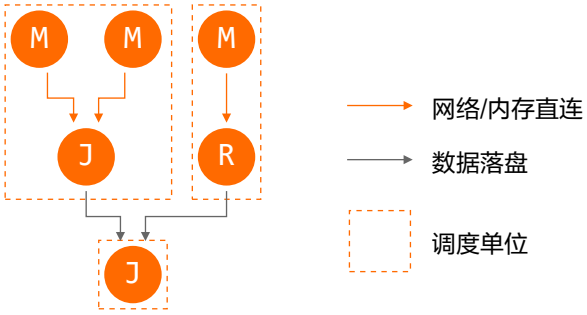


在通过对逻辑节点和逻辑边上映射不同的物理特性，离线作业和准实时作业都能得到准确的描述:

- 离线作业：每个节点按需去申请资源，一个逻辑节点代表一个调度单位；节点间连接边上传输的数据，通过落盘的方式来保证可靠性；

- 准实时作业：整个作业的所有节点都统一在一个调度单位内进行 gang scheduling；节点间连接边上通过网络/内存直连传输数据，并利用数据 pipeline 来追求最优的性能。

今天在线上，离线模式因为其 on-demand 的资源申请以及中间数据落盘等特点，作业在资源利用率，规模性和稳定性方面都有明显的优势。而准实时模式则通过常驻的计算资源池以及 gang scheduling 这种 greedy 资源申请，降低了作业运行过程中的 overhead，并使得数据的 pipelined 传输处理成为可能，达到加速作业运行的效果，但其资源使用的特点，也使其无法在广泛范围内来支持大规模作业。DAG 2.0 的升级，不仅在同一套架构上统一了这两种计算模式，更重要的是这种统一的描述方式，使得探索离线作业高资源利用率，以及准实时作业的高性能之间的 tradeoff 成为可能：当调度单位可以自由调整，就可以实现一种全新的混合的计算模式，我们称之为 Bubble 执行模式。



混合(bubble)执行

这种混合 Bubble 模式，使得 DAG 的用户，也就是上层计算引擎的开发者（比如 MaxCompute 的优化器），能够结合执行计划的特点，以及引擎终端用户对资源使用和性能的敏感度，来灵活选择在执行计划中切出 Bubble 子图。在 Bubble 内部充分利用网络直连和计算节点预热等方式提升性能，没有切入 Bubble 的节点则依然通过传统离线作业模式运行。回过头来看，现有的离线作业模式和准实时作业模式，分别可以被描述成 Bubble 执行模式的两个极端特例，而在统一的新模型之上，

计算引擎和执行框架可以在两个极端之间，根据具体需要，选择不同的平衡点，典型的几个应用场景包括：

- Greedy Bubble：在可用的资源（集群规模，quota 等）受限，一个大规模作业无法实现 gang scheduling 时，如果用户对资源利用率不敏感，唯一的目标是尽快跑完一个大规模作业。这种情况下，可以实现基于可用计算节点数目，实施 greedy 的 bubble 切割的策略, 尽量切出大的 bubble；

- Efficient Bubble：在作业的运行过程中，节点间的运算可能存在天然的 barrier (比如 sort 运算， 建 hash 表等等)。如果把两个通过 barrier 边连接的节点切到一个 bubble 中，虽然作业 e2e 性能上还是会有调度 overhead 降低等带来的提升，但是因为数据无法完全 pipeline 起来，资源的利用率达不到最高。 那么在对资源的利用率较为敏感时，可以避免 bubble 内部出现 barrier 边。这同样是计算引擎可以根据执行计划做出决定的。

这里只列举了两个简单的策略，其中还有更多可以细化以及针对性优化的地方。在不同的场景上，通过 DAG 层面提供的这种灵活按照 bubble 执行计算的能力，允许上层计算可以在不同场景上挑选合适的策略，更好的支持各种不同计算的需求。

支持新型计算模式的描述

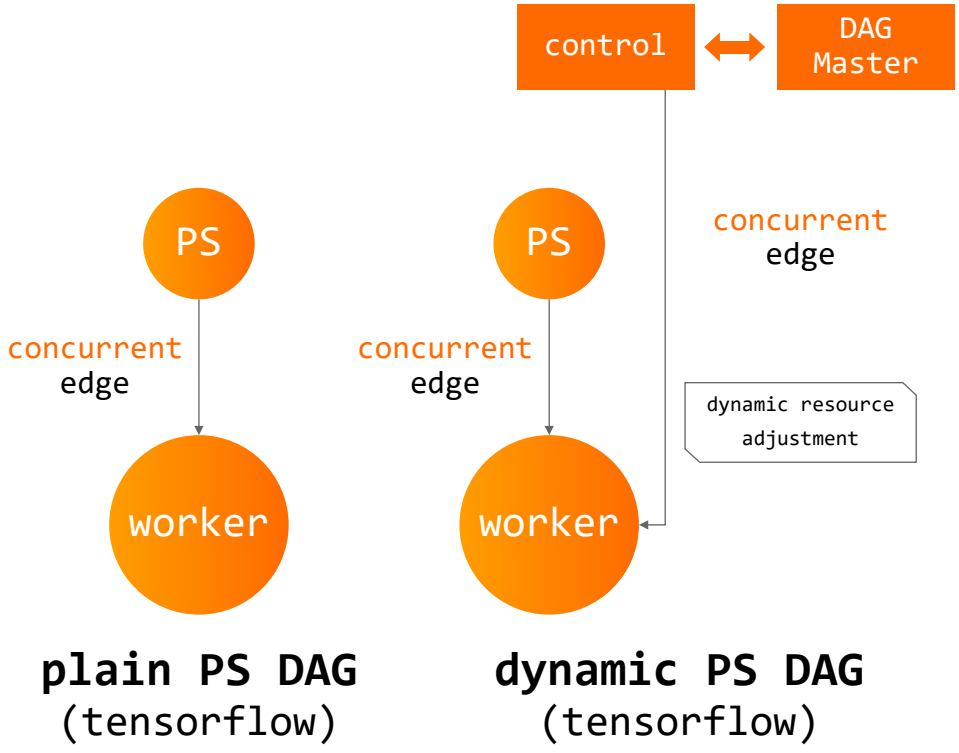
1.0 的执行框架的底层设计受 Map-Reduce 模式的影响较深，节点之间的边连接，同时混合了调度顺序，运行顺序，以及数据流动的多种语义。通过一条边连接的两个节点，下游节点必须在上游节点运行结束，退出，并产生数据后才能被调度。这种描述对于新型的一些计算模式并不适用。比如对于 Parameter Server 计算模式，Parameter Server (PS) 与 Worker 在运行过程中有如下特点：

- PS 作为 parameter 的 serving entity, 可以独立运行；

- Worker 作为 parameter 的 consumer 和 updater, 需要 PS 在运行后才能有效的运行，并且在运行过程中需要和PS持续的进行数据交互。

这种运行模式下，PS 和 worker 之间天然存在着调度上的前后依赖关系。但是因为 PS 与 worker 必须同时运行，不存在 PS 先退出 worker 才调度的逻辑。所以在 1.0 框架上， PS 与 worker 只能作为两个孤立无联系的 stage 来分开调度和运行。此外所有 PS 与 worker 之间，也只能完全通过计算节点间直连通讯，以及在外部 entity（比如 zookeeper 或 nuwa）协助来进行沟通与协调。这导致 AM/DAG 作为中心管理节点作用的缺失，作业的管理基本被下放计算引擎上，由计算节点之间自行试图协调来完成。这种无中心化的管理，对稍微复杂的情况下（failover 等）无法很好的处理。

在 DAG 2.0 的框架上，为了更准确的描述节点之间的调度和运行关系，引入并且实现了 concurrent edge 的概念：通过 concurrent edge 连接的上下游节点，在调度上存在先后，但是可以同时运行。而调度的时机也可以灵活配置：可以上下游同步调度，也可以在上游运行到一定程度后，通过事件来触发下游的调度。在这种灵活的描述能力上，PS 作业可以通过如下这种 DAG 来描述，这不仅使得作业节点间的关系描述更加准确，而且使得 AM 能够理解作业的拓扑，进行更加有效的作业管理，包括在不同计算节点发生 failover 时不同的处理策略等。



此外，DAG 2.0 新的描述模型，也允许 PAI 平台上的 Tensorflow/PS 作业实现更多的动态优化，并进行新的创新性工作。在上图的 dynamic PS DAG 中，就引进了一个额外的 control 节点，这一节点可以在作业运行过程中（包括 PS workload 运行之前和之后），对作业的资源申请，并发度等进行动态的调整，确保作业的优化执行。

事实上 concurrent edge 这个概念，描述的是上下游节点运行/调度时机的物理特性，也是我们在清晰的逻辑物理分层的架构上实现的一个重要扩展。不仅对于 PS 作业模式，在之前描述过的对于通过 bubble 来统一离线与准实时作业计算模式，这个概念也有重要的作用。

2-1-3. DAG 2.0 与上层计算引擎的集成

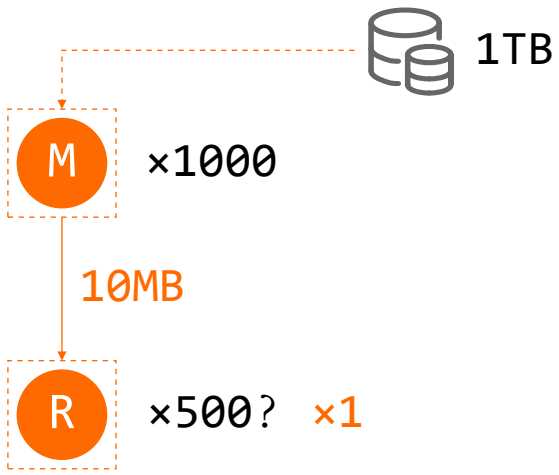
DAG 2.0 作为计算平台的分布式运行基座，它的升级换代，为上层的各种计算引擎提供了更多灵活高效的执行能力，而这些能力的落地，需要通过与具体计算场景的紧密结合来实现。接下来通过 2.0 与上层各个计算引擎（包括 MaxCompute 以及 PAI 平台等）的一些对接场景，具体举例说明 2.0 新的调度执行框架，如何赋能平台上层的计算与应用。

运行过程中的 DAG 动态调整

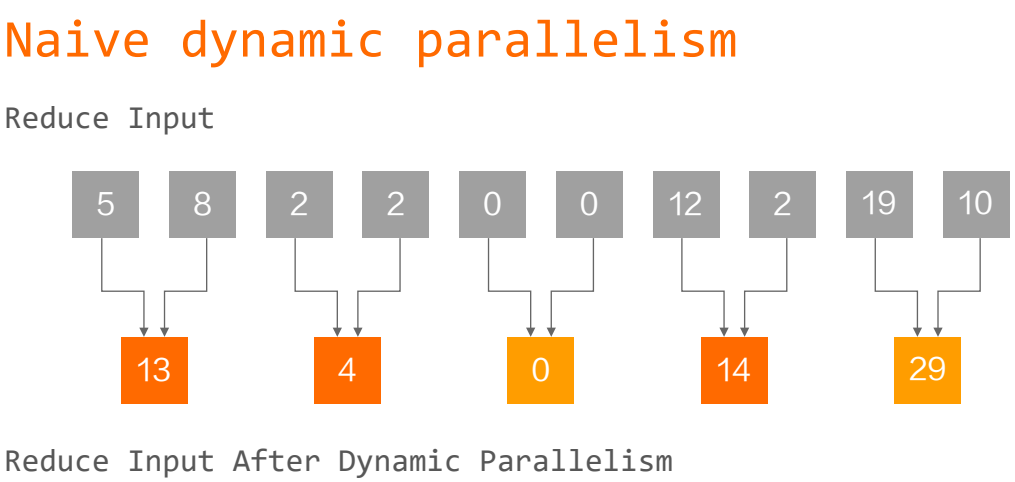
作为计算平台上的作业大户，MaxCompute 平台上多种多样的计算场景，尤其是离线作业中的各种复杂逻辑，为动态图能力的落地提供了丰富多样的场景，这里从动态物理图和逻辑图几个方面讨论几个例子。

动态并发度调整

基于作业运行期间中间数据大小进行动态并发度调整，是 DAG 动态调整中最基本的能力。以传统 MR 作业为例，对于一个静态 MR 作业而言，能根据读取数据量来比较准确判断 Mapper 的并发，但是对于 Reducer 的并发只能简单推测，比如下图中对于处理 1TB 的 MR 作业而言，提交作业时，只能根据 Mapper 1000 并发，来猜测给出 500 的 Reducer 并发度，而如果数据在 Mapper 经过大量过滤导致最终之产出 10MB 中间数据时，500 并发度 Redcuer 显然是非常浪费的，动态的 DAG 必须能够根据实际的 Mapper 产出来进行 Reducer 并发调整（500->1）。

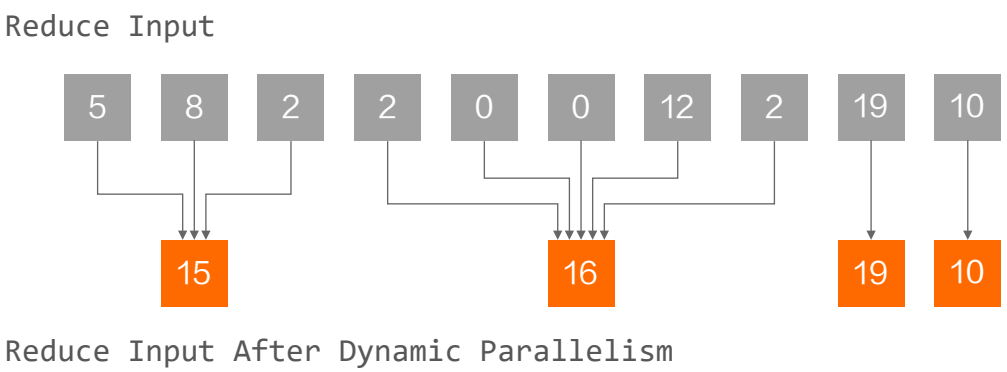


而实际实现中，最简单的动态调整，会直接按照并发度调整比例来聚合上游输出的 partition 数据，如下图这个并发度从 10 调整到 5 的例子所示，在调整的过程中，可能产生不必要的数据倾斜。

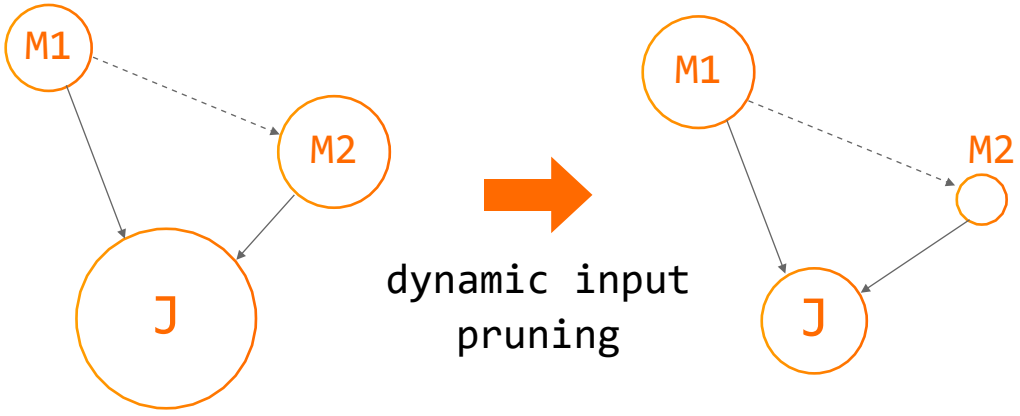


DAG 2.0 基于中间数据的动态并发调整实现，充分考虑了数据 partition 可能存在倾斜的情况，对动态调整的策略进行了优化，优化后数据的分布更加均匀，可以有效避免由于动态调整可能引入的数据倾斜。

Adaptive dynamic parallelism



这种最常见下游并发调整方式是 DAG 2.0 动态物理图能力的一个直观展示。在 DAG 2.0 项目中，结合计算引擎数据处理的特点，还探索了基于源数据的动态并发调整。例如对于最常见的两个原表数据的 join (M1 join M2 at J)， 如果用节点大小来表示 DAG 2.0 处理数据的的多少，那对于下图这么一个例子，M1处理的是中等的一个数据表（假设 M1 需要并发度为 10），M2 处理的是较大的数据表（并发度为 1000），naïve 的执行方式会将按照 10 + 1000 的并发度调度，同时因为 M2 输出需要全量 shuffle 到 J， J 需要的并发度也会较大（~1000）.



而实际上，对于这种计算 pattern 而言，M2 需要读取（并进行处理）的，应该只有能和 M1 输出 join 得上的数据，也就是说在考虑了整体执行 cost 后，在这种 M1 期望的输出数据要比 M2 小得多的情况下，可以先行调度 M1 完成计算，将 M1 输出数据的 statistics 在 AM/DAG 端进行聚合，然后只挑选出 M2 的有效数据进行处理。这里“M2 的有效数据”的选择本质上是一个 predicate push down 的过程，可以由计算引擎的优化器和运行时联合进行判断。也就是说，这种情况下 M2 的并发度调整，是和上层计算紧密结合的。

一个最直观的例子是，如果 M2 是一个 1000 个分区的分区表，并且分区的 key 和 join的 key 相同，那么可以只读取 M2 能和 M1 输出 join 上有效数据的 partition 进行读取处理。假如 M1 的输出只包含了 M2 原表数据的 3 个 partition keys, 那么在 M2 就只需要调度 3 个计算节点来处理这 3 个分区的数据。也就是说 M2 的并发度从默认的 1000，可以降低到 3，这在保证同样的逻辑计算等效性与正确性的前提下，能大大降低计算资源的消耗，并数倍加速作业的运行。这里的优化来自于几个方面：

- M2 的并发度 (1000->3) 以及处理的数据量大大降低；
- M2 需要 shuffle 到 J 的数据量以及 shuffle 需要的计算量大大降低；
- J 需要处理的数据量以及其并发度能大大降低。

从上图这个例子中我们也可以看到，为了保证 M1->M2 的调度顺序，DAG 中在 M1 和 M2 间引入了一条依赖边，而这条边上是没有数据流动的，是一条只表示执行先后的依赖边。这与传统 MR/DAG 执行框架里，边的连接与数据流动紧绑定的假设也有不同，是在 DAG 2.0 中对于边概念的一个拓展之一。

DAG 执行引擎作为底层分布式调度执行框架，其直接的对接"用户" 是上层计算引擎的开发团队，其升级对于终端用户除了性能上的提升，直接的体感可能会少一点。这里我们举一个终端用户体感较强的具体例子，来展示 DAG 更加动态的执行能力，能够给终端用户带来的直接好处。就是在 DAG 动态能力的基础上，实现的 LIMIT 的优化。

对于 SQL 用户来说，对数据进行一些基本的 ad hoc 操作，了解数据表的特性，一个非常常见的操作是 LIMIT，比如：

```
SELECT * FROM tpch_lineitem WHERE l_orderkey > 0 LIMIT 5;
```

在分布式执行框架上，这个操作对应的执行计划，是通过将源表做切分后，然后调度起所需数目的 mapper 去读取全部数据，再将mapper的输出汇总到 reducer 后去做最后的 LIMIT 截断操作。假设源表（这里的 tpch_lineitem）是一个很大的表，需要 1000 个 mapper 才能读取，那么在整个分布式执行过程中，涉及的调度

代价就是要调度 1000 mapper + 1 reducer。这个过程中会有一些上层计算引擎可以优化的地方，比如每个 mapper 可以最多输出 LIMIT 需要的 record 数目（这里的 LIMIT 5）提前退出，而不必处理完所有分配给它的数据分片等等。但是在一个静态的执行框架上，为了获取这样简单的信息，整体 1001 个计算节点的调度无法避免。这给这种 ad hoc query 执行，带来了巨大的 overhead，在集群资源紧张的时候尤其明显。

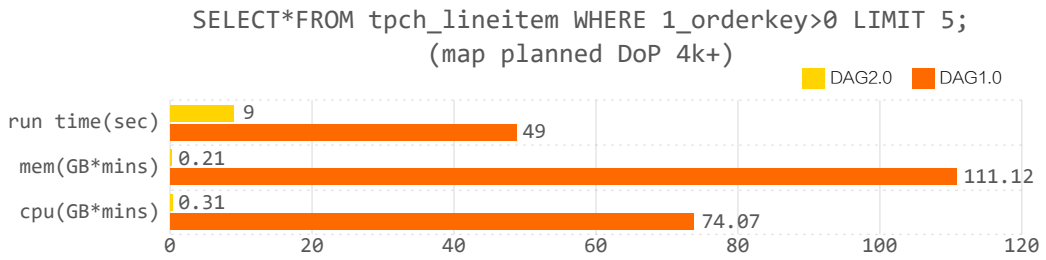
DAG 2.0 上，针对这种 LIMIT 的场景，依托新执行框架的动态能力，实现了一些优化，这主要包括几方面：

- 上游 Exponential start: 对于这种大概率下上游 mapper 计算节点不需要全部运行的情况，DAG 框架将对 mapper 进行指数型的分批调度，也就是调度按照 1，10 ... FULL 的分批执行；

- 下游的 Early scheduling: 上游产生的 record 数目作为执行过程中的统计数据上报给 AM， AM 在判断上游已经产生足够的 record 条数后，则提前调度下游 reducer 来消费上游的数据；

- 上游的 Early termination: 下游 reducer 在判断最终输出的 LIMIT 条数已经满足条件后，直接退出。这时候AM可以触发上游 mapper 整个逻辑节点的提前退出（在这种情况下，大部分 mapper 可能都还没有调度起来），整个作业也能提前完成。

这种计算引擎和 DAG 在执行过程中的灵活动态交互，能够带来大量的资源节省，以及加速作业的执行。在线下测试和实际上线效果上，基本上绝大多数作业在 mapper 执行完 1 个计算节点后就能提前退出，而无需全量调起 (1000 vs 1)。下图是在线下测试中，当 mapper 并发为 4000 时，上述 query 优化前后的区别：



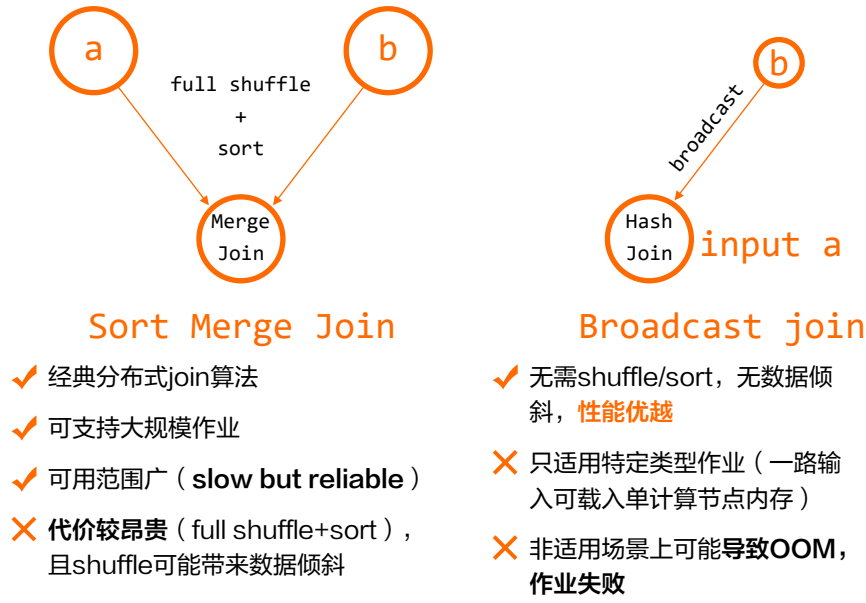
可以看到，执行时间优化后增速了 5X+， 计算资源的消耗更是减小了数百倍。这个线下测试结果作为比较典型的例子，稍微有些理想化。为了评估真实的效果，在 DAG 2.0 上线后，选取了 LIMIT 优化生效的线上作业，统计了一星期结果如下：这个优化平均为每个作业节省了 (254.5 cores x min CPU + 207.3 GB x min) 的计算资源，同时每个作业上，平均能节省 4349 个（无效）计算节点的调度。

LIMIT 执行上的改进，作为一个针对特殊场景上实现的优化，涉及了整个 DAG 执行不同策略的调整，这种细化的改进能力，能更直观的体现 DAG 2.0 架构升级诸多好处：灵活的架构使得 DAG 的执行中拥有了更多的动态调整能力，也能和计算引擎在一起进行更多有针对性的优化。

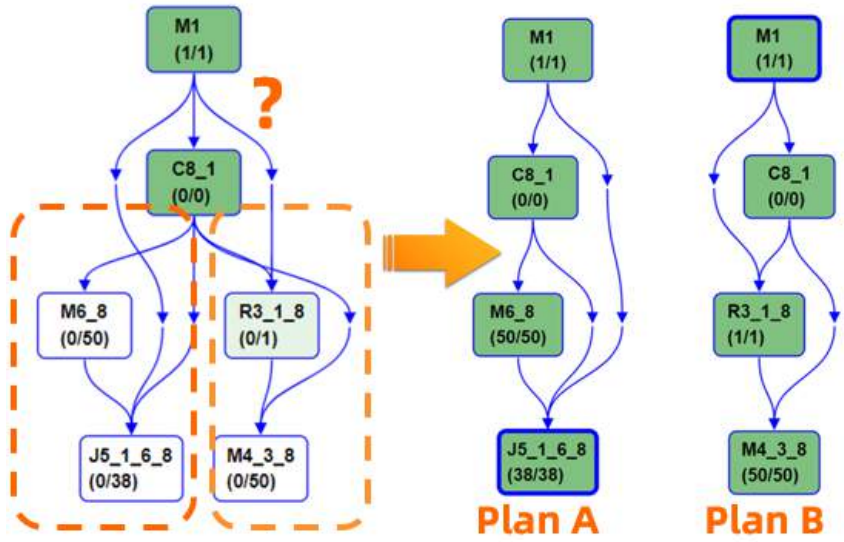
不同情况下的动态并发度调整，以及具体调度执行策略的动态调整，只是图的物理特性动态调整的几个例子。事实上对于物理特性运行时的调整，在 2.0 的框架之上有各种各样的应用，比如通过动态数据编排 /shuffle 来解决各种运行期间的 skew 问题等，这里不再做进一步的展开。接下来我们再来看看 DAG 2.0 上对于逻辑图的动态调整做的一些探索。

动态逻辑图的调整

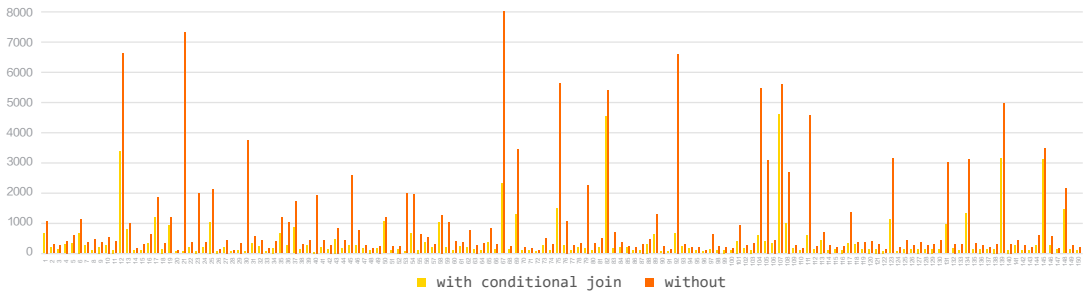
分布式 SQL 中，map join 是一个比较常见的优化，其实现原理是在 join 的两个表中，如果有一个超小的表（可以 fit 到单个计算节点的内存中），那对于这个超小表可以不做 shuffle，而是直接将其全量数据 broadcast 到每个处理大表的分布式计算节点上。通过在内存中直接建立 hash 表，完成 join 操作。map join 优化能大量减少 (大表) shuffle 和排序，非常明显的提升作业运行性能。但是其局限性也同样显著：如果"超小表"实际不小，无法fit进单机内存，那么在试图建立内存中的 hash 表时就会因为 OOM 而导致整个分布式作业的失败，而需要重跑。所以虽然 map join 在正确使用时，可以带来较大的性能提升，但实际上优化器在产生 map join 的 plan 时需要偏保守，很多情况下需要用户显式的提供 map join hint 来产生这种优化。此外不管是用户还是优化器的选择，对于非源表的输入都无法做很好的判断，因为中间数据的大小往往需要在作业运行过程中才能准确得知。



而 map join 与默认 join 方式 (sort merge join) 对应的其实是两种不同优化器执行计划，在 DAG 层面，其对应的是两种不同的逻辑图。要支持这种运行过程中根据中间数据特性的动态优化，就需要 DAG 框架具备动态逻辑图的执行能力，这也是在 DAG 2.0 上开发的 conditional join 功能。如同下图展示，在对于 join 使用的算法无法被事先确定的时候，允许优化器提供一个 conditional DAG，这样的 DAG 同时包括使用两种不同 join 的方式对应不同执行计划支路。在实际执行时，AM 根据上游产出数据量，动态选择一条支路执行 (plan A or plan B)。这样的动态逻辑图执行流程，能够保证每次作业运行时都能根据实际作业数据特性，选择最优的执行计划。

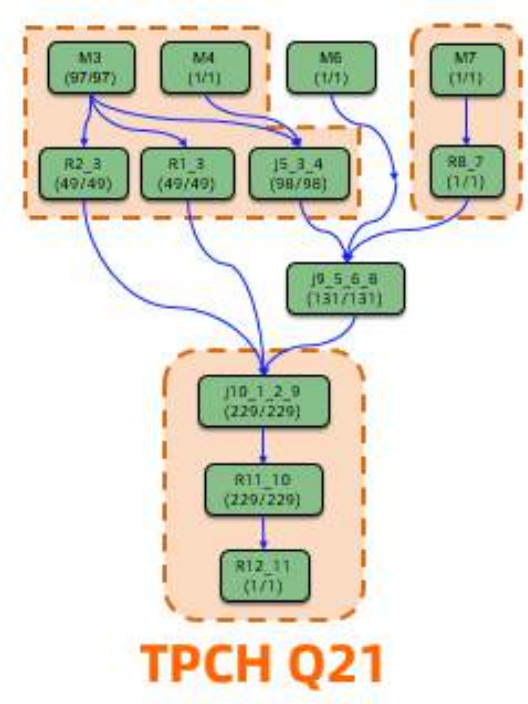


conditional join 是动态逻辑图的第一个落地场景，在线上选择一批适用性作业，动态的 conditional join 相比静态的执行计划，整体获得了将近 3X 的性能提升。



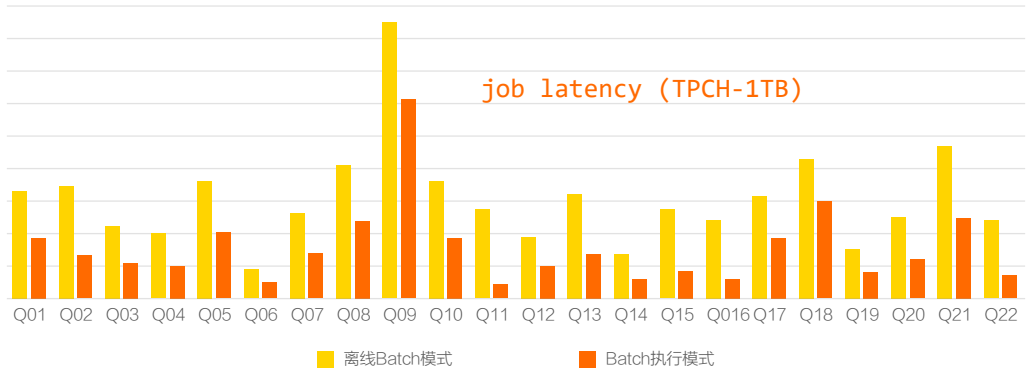
混合 Bubble 模式

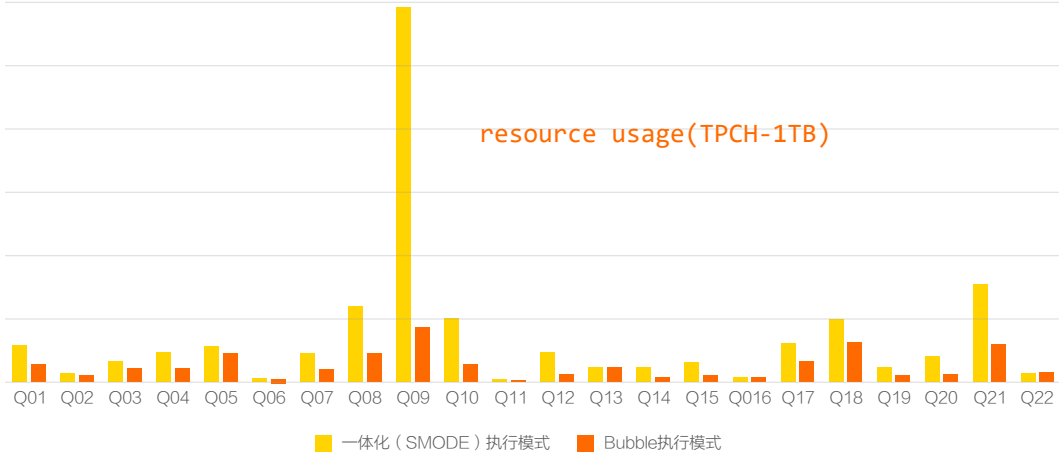
Bubble 模式是我们在 DAG 2.0 架构上探索的一种全新的作业运行方式，通过对于 bubble 大小以及位置的调整，可以获取性能和资源利用率的不同 tradeoff 点。这里通过一些更加直观的例子，来帮助大家理解 Bubble 执行在分布式作业中的实际应用。



在上图的 TPCH Q21 上。比如在 Q21 上，我们看到了通过将作业被切分为三个 "bubble"，数据能够有效的在节点之间 pipeline 起来，并且通过热点节点实现调度的加速。最终消耗的资源数 (cpu * time) 是准实时作业的 35%, 而性能则与一体化调度的准实时作业非常相近 (96%) , 比离线作业性能提升 70% 左右。

在标准 TPCH 1TB 全量测试中，混合 bubble 模式体现出了相比离线和准实时的一体化模式（gang scheduling）更好的资源/性能 tradeoff。选用 Greedy Bubble (size=500) 的策略，bubble相比离线作业性能提升了 2X（资源消耗仅增加 17%，具体数值略）。同时与一体化调度的准实时作业比较，bubble 执行在只消耗了 40% 不到的资源（cpu * time）的前提下，其性能达到了准实时作业的 85%（具体数值略）。可以看到，这种新型的 bubble 执行模式，允许我们在实际应用中获取很好的性能与资源的平衡，达到系统资源有效的利用。Bubble 执行模式目前正在阿里集团内部全量上线中，我们在实际线上的作业也看到了与 TPCH 测试非常相似的效果。





如同之前所述，混合 bubble 模式支持了不同切分策略，这里提供的只是一种切分策略上的效果。在与上层计算引擎（e.g., MaxCompute 优化器）紧密结合时，这种 DAG 分布式调度 bubble 执行的能力，能够允许我们根据可用资源和作业计算特点，来寻找性能与资源利用率的最佳平衡点。

2-1-4. 资源的动态配置和动态管理

传统分布式作业对于每个计算节点需要的资源类型 (CPU/GPU/Memory) 和大小都是预先确定下来的。然而在分布式作业，在作业运行之前，对计算节点资源类型和大小的合理选择，是比较困难的。即便对于计算引擎的开发者，也需要通过一些比较复杂的规则，才能预估出大概合理的配置。而对于需要将这些配置透明给终端用户的计算模式，终端用户要做出选择就更加困难。

在这里以 PAI 的 Tensorflow (TF) 作业为例，描述 DAG 2.0 的资源动态配置能力，怎样帮助平台的TF作业选择合理的 GPU 类型资源以及提高 GPU 资源的利用率。相比 CPU 而言，GPU 作为一种较新的计算资源，硬件的更新换代较快，同时普通终端用户对于其计算特点也相对不了解。因此终端用户在指定 GPU 资源类型时，经常存在着不合理的情况。与此同时，GPU在线上又是相对稀缺资源。今天在线上，GPU 申请量经常超过集群 GPU 总数，导致用户需要花很长时间排队等待资源。而另外一方面，集群中 GPU 的实际利用率却偏低，平均只有 20% 左右。这种申请和实际使用之间存在的 Gap，往往是由于用户作业配置中，事先指定的 GPU 资源配置不合理造成。

在 DAG2.0 的框架上，PAI TF GPU 作业（见 session 2.2.2 的 dynamic PS DAG） 引入了一个额外的“计算控制节点”，可以通过运行 PAI 平台的资源预测算法，来判断当前作业实际需要的 GPU 资源类型，并在必要的时候，通过向 AM 发送动态事件，来请求修改下游 worker 实际申请的 GPU 类型。这其中资源预测算法，可以根据算法的类型，数据的特点，以及历史作业信息来做 HBO (history based optimization)，也可以通过 dry-run 的方法来进行试运行，以此确定合理的资源类型。

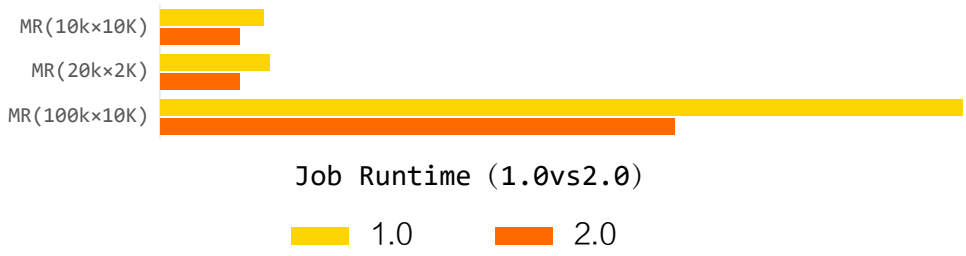
具体实现上，这个场景中 control stage 与 worker 之间通过 concurrent edge 连接，这条边上的调度触发条件是在 control stage 已经做出资源选择决定之后，通过其发出的事件来触发。这样的作业运行期间的动态资源配置，在线上功能测试中，带来了 40% 以上的集群 GPU 利用率提升。

作为物理特性一个重要的维度，对计算节点的资源特性在运行时的动态调整能力，在 PAI 以及 MaxCompute 上都能找到广泛的应用。以 MaxCompute SQL 为例，对于下游节点的 CPU/Memory 的大小，可以根据上游数据的特点进行有效的预判；

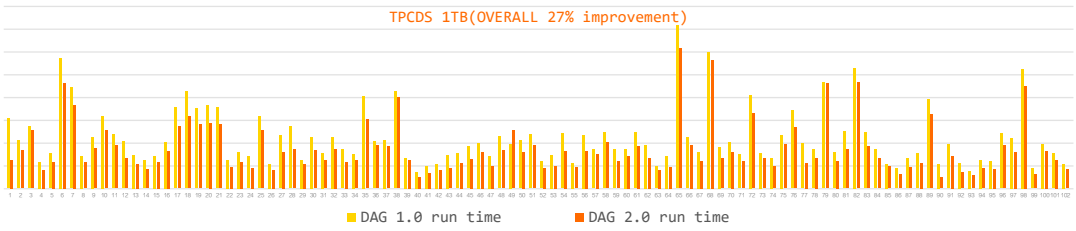
同时对于系统中发生的 OOM，可以尝试自动调高 OOM 后重试的计算节点的内存申请，避免作业的失败，等等。这些都是在 DAG 2.0 上新的架构上实现的一些新功能，在这里不做具体的展开。

2-1-5. 工程化与上线

作为分布式系统的底座，DAG 本身的动态能力以及灵活度，在与上层计算引擎结合时，能够支持上层计算实现更加高效准确的执行计划，在特定场景上实现数倍的性能提升以及对资源利用率的提高。在上文中，也集中介绍了整个 DAG 2.0 项目工作中，开发实现的一些新功能与新的计算模式。除了对接计算引擎来实现更高效的执行计划，调度本身的敏捷性，是 AM/DAG 执行性能的基本素质。DAG 2.0 的调度决策均基于事件驱动框架以及灵活的状态机设计来实现，在这里也交出 DAG 2.0 在基本工程素养和性能方面的成绩单：



这里选用了最简单的 Map-Reduce (MR) 作业为例，对于这种作业，调度执行上并无太多可以取巧的地方，考验的是调度系统本身的敏捷度和整个处理流程中的全面去阻塞能力。这个例子也凸显了 DAG 2.0 的调度性能优势，尤其作业规模越大，优势越发明显。此外，对于更接近线上的 work-load 的场景，在 TPCDS 标准 benchmark 中，当执行计划和运行逻辑完全相同时，DAG 2.0（未打开动态执行等功能）的高性能调度也给作业带来了显著的提升。



最后，对于一个从头到尾完整替代原有系统的新一代全新框架，怎样无缝对接线上场景，实现大规模的上线，是一个同样重要（甚至更重要）的话题，也是对一个实际生产系统进行升级，与小范围的新系统 POC 之间最大的区别。今天的伏羲调度系统，每天支撑着阿里集团内外大数据计算平台千万的分布式作业。DAG/AM 这一核心分布式调度执行组件的更新换代，要完整替换线上已经支撑了大数据业务 10 年的分布式生产系统，而不造成现有场景的失败，这需要的不仅仅是架构和设计上的先进性。如何在"飞行中换引擎"，保质保量的实现系统升级，其挑战完全不亚于新的系统架构本身的设计。要实现这样的升级，拥有一个稳固的工程基座，以及测试/发布框架，都是不可或缺的。没有这样子的底座，上层的动态功能与新计算模式，都无从谈起。

目前 DAG 2.0 目前已全面覆盖了阿里集团 MaxCompute 所有线上的 SQL 离线作业和所有准实时作业，以及 PAI 平台的所有 Tensorflow 作业（CPU 和 GPU）+ PyTorch 作业。每天支撑数千万分布式作业的运行，并经受了 19 年双 11 /双 12 的考验。在面对两次大促创历史记录的数据洪峰（相比 18 年增长 50%+）压力下，保障了集团重点基线在大促当天准时产出。与此同时，更多种类型的作业（例如跨集群复制作业等等）正在迁移到 DAG 2.0 的新架构，并且依托新架构升级计算作业本身的能力。DAG 2.0 的框架基座的上线，为各条计算线上依托其实现新功能打下了坚实基础。

2-1-6. DAG 2.0 规模扩张之路

伏羲 DAG 2.0 核心架构的升级，旨在夯实阿里计算平台长期发展的基础，并支持上层计算引擎与分布式调度方面结合，实现各种创新和创建新计算生态。架构的升级本身是向前迈出的重要一步，但也只是第一步。要支撑企业级的，各种规模，各种模式的全频谱计算平台，需要将新架构的能力和上层计算引擎，以及伏羲系统其他组件进行深度整合。依托阿里的应用场景，DAG 2.0 除了在作业规模等方面继续在业界保持领先之外，架构和功能上也有许多创新， 比如前面我们已经介绍过的：

- 业界首次在分布式执行框架上，实现了执行过程中逻辑图和物理图的双重动态可调；

- 通过 Bubble 机制实现了混合的计算模式，探索资源利用率和作业性能间的最佳平衡。

除此之外，DAG 2.0 更加清晰的系统封层架构带来的一个重要改变就是能有利于新功能更快速开发，提速平台和引擎向前创新。由于篇幅有限，本文只能由点及面介绍了一部分新功能与新计算模式，还有许许多多已经实现，或正在开发中的功能，在业界都是全新的探索，暂时不做进一步展开，比如：

- 准实时作业体系架构的整体升级: 资源管理与多作业管理的解耦，支持准实时作业场景上的动态图功能；

- 常驻的单 container 多 slot 执行的 cache-aware 查询加速服务（MaxCompute 短查询）；

- 基于状态机的作业节点管理以及失败下的智能重跑机制；

- 动态可定义的 shuffle 方式：通过 recursive shuffle 等方式动态解决线上大规模作业中的 in-cast 问题；

- 基于 adaptive 的中间数据动态切分与聚合，解决实际分布式作业中各种数据倾斜问题；

- 支持 PAI TF GPU 作业的多执行计划选项；

- 通过 DAG 执行过程中与优化器的交互，实现渐进式的交互式动态优化；

- 支持 Imperative 语言特性，通过 DAG 的动态自增长等能力，对接 IF/ELSE/LOOP 等语义。

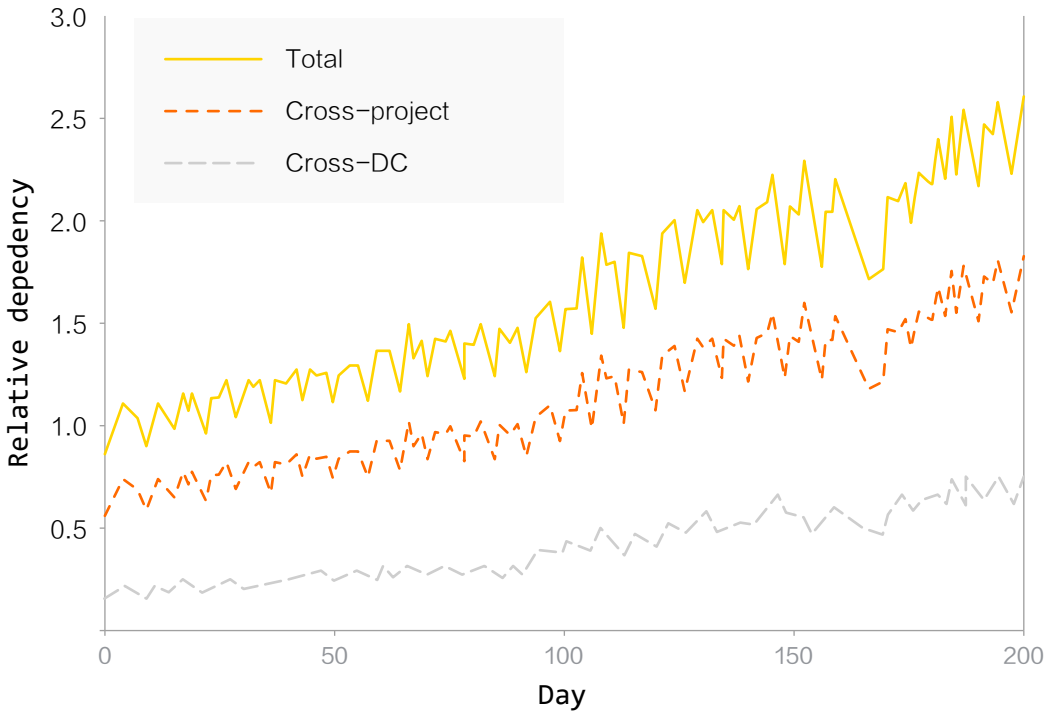
核心调度底座能力的提升，能够为上层的各种分布式计算引擎提供真正企业级的服务能力，提供必须的弹药。而这些计算调度能力提升带来的红利，最终会通过 MaxCompute 和 PAI 等引擎，透传到终端的阿里云计算服务的各个企业。在过去的十年，阿里业务由内向外的驱动，锻造了业界规模最大的云上分布式平台。而通过更好服务集团内部以及云上的企业用户，我们希望可以完成由内向外到由外至内的整个正向循环过程，推动计算系统螺旋式上升的不断创新，并通过性能/规模，以及智能化自适应能力两个维度方面的推进，降低分布式计算服务的使用门槛，真正实现大数据的普惠。

2-2. “愚公”系统：实现跨地域的数据和计算调度

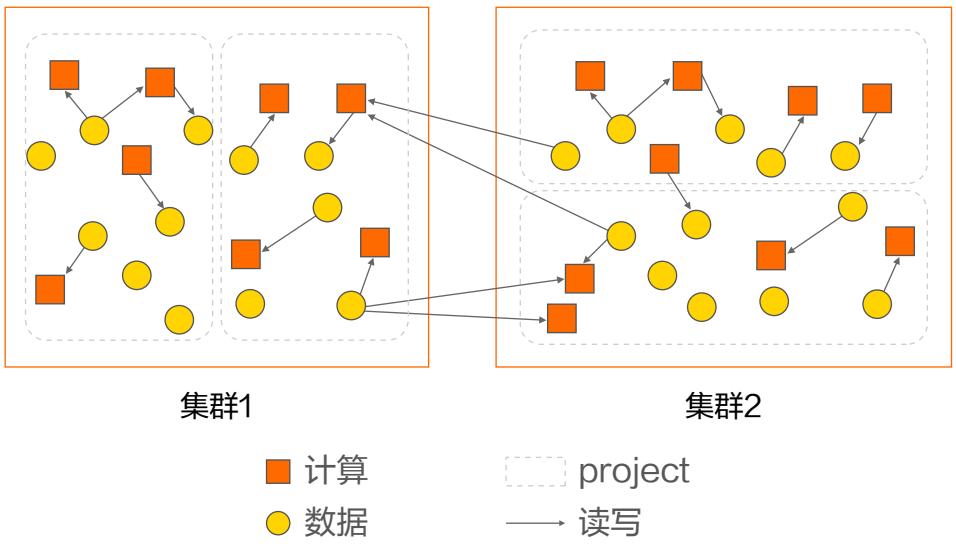
本文作者 史英杰 | 阿里云智能 技术专家

MaxCompute 作为阿里经济体的大数据计算平台，每天运行着数以千万计的作业，处理 EB 级别的数据，这些作业和数据分布在全球各个数据中心的不同集群，当作业运行和输入数据不在同一个集群中时，称之为跨集群数据依赖。随着 MaxCompute 业务的高速发展，跨集群依赖量也急速增长。复杂的业务依赖关系不可避免的会产生大量的跨数据中心的网络传输，而跨数据中心的网络具有带宽小，延迟高，稳定性低的特点，并且价格还贵。如何平衡各集群的计算和存储利用率，降低带宽成本，成为了亟待解决的一个难题。

MaxCompute 这样的跨地域、多集群的大数据系统中，作业和数据可能分布在全球各地的集群中。随着 MaxCompute 业务的高速发展，跨集群依赖量也急速增长。下图展示了一段时间内输入数据量、跨 project 依赖量、跨 idc 依赖量的增长趋势。



在逻辑上，数据和计算以 project 为单位组织。根据 project 的存储量、计算量等因素，会为每个 project 指定一个默认集群。这个 project 产生的所有数据都会存储在其默认集群上，所有计算也会运行在默认集群上。下图展示了 project、计算、数据的逻辑关系。



本文要解的主要问题，正是降低跨域大数据系统的跨集群流量。具体来说，我们通过三种方式实现这个目的。

2-2-1. project 整体排布

即将上图中的 project 在集群上进行重排列。通过将数据依赖度高的 project 聚集在相同的数据中心上，减少对跨数据中心的网络带宽使用需求。project 排布，核心思想很简单，将相互依赖较大的 project 放在一起，实现数据的产出和消费发生在集群内，实现自给自足。这个问题是一个类似 bin packing 问题，后者是 NP-hard 问题。实际上，即使找到了一个最优解，由于一些原因也无法将其落地，如存量数据的体量太大，如果不考虑现有的排布，完全通过数据迁移实现理论上的最优解，对带宽来说是个巨大的压力，对线上正在运行着的业务也是无法接受的。其次，即使找到了最优解并且落地了，由于 project 之间的依赖在不断变化，因此过段时间这个最优解不再是最优解了。因此，我们的目标并不是找到理论上的最优排布，而是在一定迁移代价的前提下，找到一个可以接受的次优解。

2-2-2. 热数据复制

通过将访问频繁的热数据进行跨集群缓存，减少频繁读取产生的直读流量。我们通过数据分析发现，有些数据被很多 job 依赖，我们称之为热数据。热数据一般数据量比较大，并且被读取次数很多。于是我们有了一个很自然的想法，将这些热数据在 remote cluster 进行缓存。由于存储是有代价的，因此这里的问题其实是一个

trade-off，在最小化带宽消耗和最小化冗余存储之间做权衡。为了降低问题的复杂度以便于求解，我们通过数据分析发现，数据的访问有一个特点：数据的访问频率与大小和数据的产出时间相关，越新的数据被访问的频率越高、被读取的越大。因此，我们将问题转化为两个问题：对哪些表进行复制、这些表的复制生命周期。问题的目标是在一定冗余存储限制的前提下，最小化带宽消耗，包括复制带宽消耗和直读带宽消耗。

2-2-3. 作业调度

将前两点未覆盖的数据依赖量大的作业调度到数据所在数据中心进行计算，对应上图中的计算节点调度到所在 project 之外的其他集群，进一步减少跨数据中心的数据访问。同数据一样，作业也是以 project 为单位进行组织，因此作业运行在所在 project 所在的集群。计算调度指将作业调度到 remote 集群，后者存储作业的大部分输入数据，以减少网络消耗。由于作业运算需要计算资源，因此计算调度需要收到集群计算资源的限制。

我们的解决方案分为离线模块和在线模块。project 排布和数据复制策略通过离线完成，作业调度需要在线实时作出决策，因为需要考虑集群负载和资源利用率，这些一直是动态变化的。

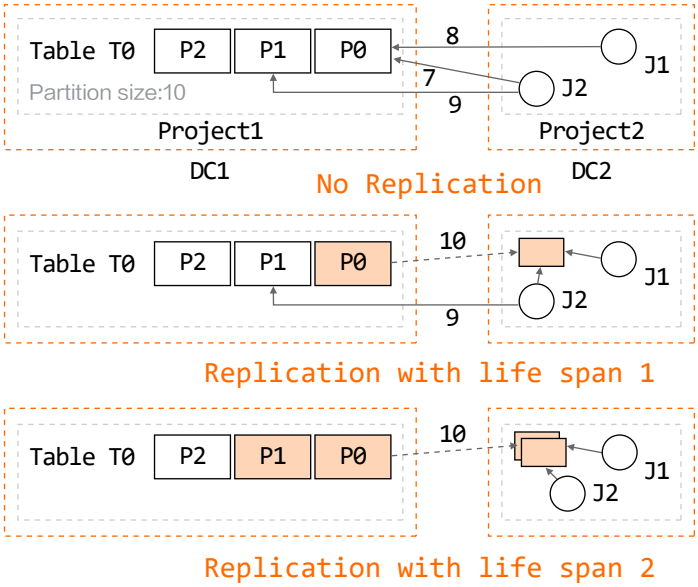
我们首先描述了离线模块使用的分析模型，被分为两个部分：project 迁移和数据复制决策。project 迁移中，我们为了降低复杂度，将副本存储限制设为无穷大；在数据复制决策中，我们提出了一种启发式算法，在有限存储限制下确定哪些表复制以及保留多久。在线模块中综合考虑静态信息（通过离线分析出作业预期的数据依赖、资源消耗）和动态的集群信息（是否有资源运行作业），来决定是否将作业调度到非默认计算集群，以节省带宽或调节集群利用率的目的。

首先我们定义数据副本生命周期的概念。考虑到距离现在越近的分区被更多的读取，我们一般只保留最近若干天的数据在远程集群上。

定义为 table i 在 DC d 上保留的分区个数，这些分区是从当前开始、连续的分区。比如 table1 在 DC1 上的生命周期为 2，那么在 Day 7 这一天，Day 6 和 Day 7 两天产生的分区会保留在 DC1 上。而在 Day 8，Day 6 的数据在 DC1 上会被删掉，取而代之的是 Day 8 这天产生的数据。那么，在任意一天，在 DC1 上读取 table1

最近两天的数据都会转化成 local read（如果本集群有数据，优先读取本集群）。维护 table1 在 DC1 上缓存数据，主要有两个开销：1. 每天通过跨集群复制将最新一天的数据复制到 DC1，所产生的跨 DC 流量（table1 分区的平均逻辑大小）；2. DC1 上保留最近 2 天分区所产生的存储开销（table1 分区的平均物理大小 * 生命周期）。如果配置更长的生命周期，会有更多的远程依赖转化为 local read，但会增加存储开销。显然，如果将生命周期设为无穷大（缓存所有分区）会最大限度减少带宽消耗，但是所消耗的冗余存储是无法接受的。

为了更好的描述生命周期的概念，以下图为例进行说明



P0，P1，P2 都是 T0 时间格式的分区，分别产生于今天，昨天和前天。T0 属于 DC1 上的 Project1。假设每个分区的大小为 10。DC2 上读取 P0 的大小为 7+8=15，读取 P1 的大小为 9。如果 DC2 上没有配置 T0 复制（T0 在 DC2 上的生命周期是 0），那么 DC2 上的作业会直读 DC1 上的 T0 的所有分区，一共会消耗 24 的跨 DC 带宽。如果 DC2 上 T0 的生命周期是 1，P0 会本地读取，而 P1 会从 DC1 直读。每天产生的直读流量为 9，每天产生的复制流量为 10，所以总的带宽消耗是 19，存储开销是 10。如果 DC2 上 T0 的生命周期是 2，P0 和 P1 都会本地读取，直读流量为 0，每天产生的复制流量仍然为 10（每天只需要复制最新产生的分区），总带宽消耗为 10，存储开销为 20。因此是否配置复制，以及保留多久的数据，是带宽和存储消耗的一个 trade-off。

以上的例子看起来简单。但我们需要为每个 table 在每个 DC 上决定是否复制（或

者说缓存），以及找到最优的生命周期，并且在每个 DC 有限存储的前提下最小化整体跨 DC 带宽，特别是 MaxCompute 有数以百万级的表，每个表有数百至数千个分区，以及百万级的作业。

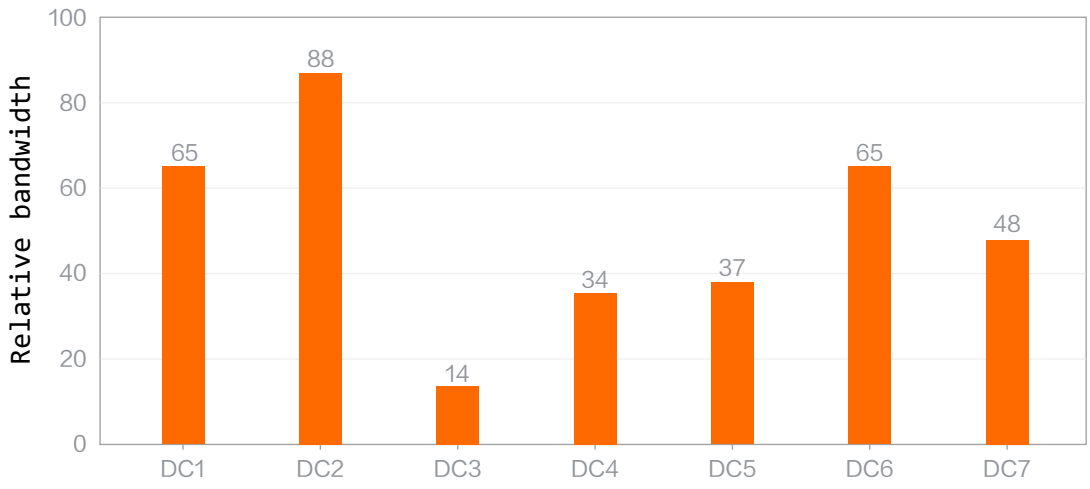
为了简化模型，我们做了一些假设：

1. 假设每两个 DC 之间的网络的单价是一样的；

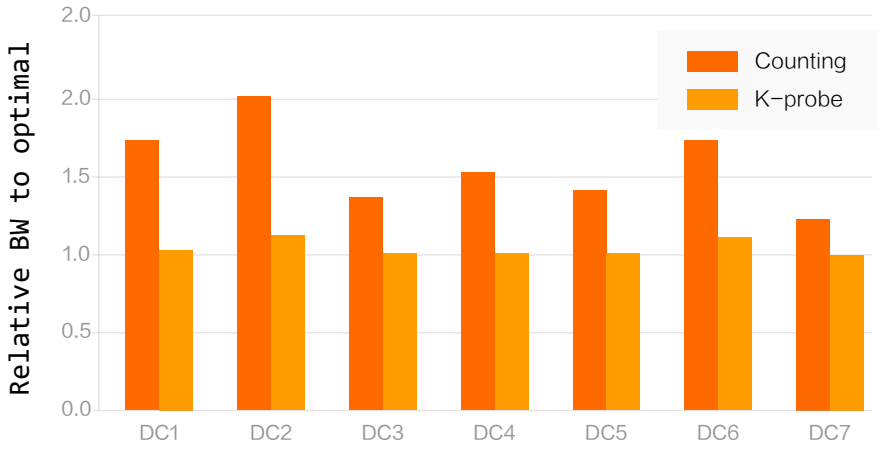
2. 假设同一张表的不同时间分区大小是接近的；

3. 每天运行的都是周期性作业，相同作业的数据依赖是相同的。

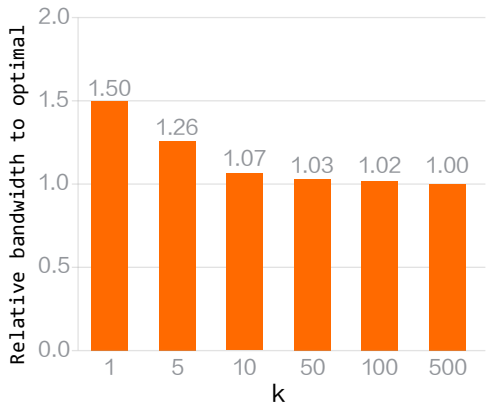
我们将论文的研究成果在阿里巴巴 MaxCompute 平台上进行了落地，下图展示了各个数据中心跨数据中心流量的减少比例，从 14% 到 88% 不等。整体上，通过引入愚公系统，跨数据中心数据流量减少了 76%。



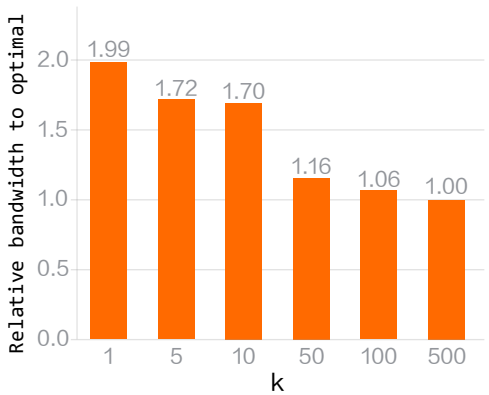
在缓存策略上，我们比较了论文中提出的 k-probe 算法和之前基于计数的方法（称之为 Counting）。Counting 的思想较为朴素，如果一个数据在过去 7 天有 5 天被读取过，那么这个数据被认为值得复制。下图对比了 Counting 和 k-probe 算法，在相同冗余存储限制的前提下 k-probe 算法分别取得了 18% 到 45% 的性能提升。另外，k-probe 算法在足够冗余存储的情况下，几乎取得了和最优解相同的性能。



另外我们对k的取值进行了实验分析。实验表明，不同 DC 对 k 值的收敛速度不同，但在超过 500 之后，对结果的影响已经很小了。

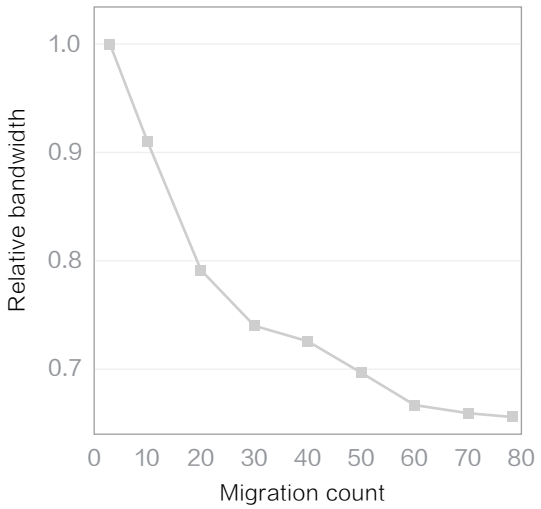


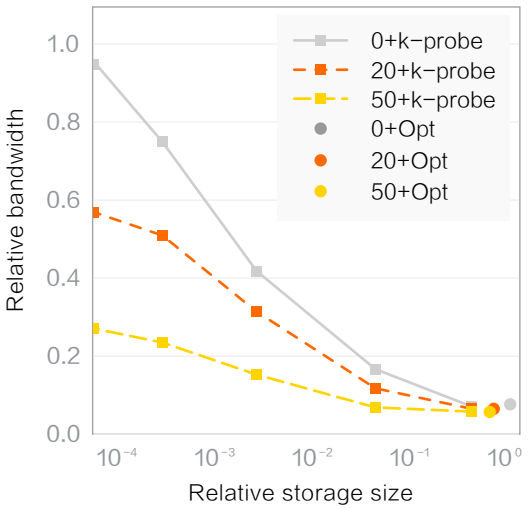
（a）DC1



（a）DC2

最后我们还对 project 迁移的收益进行了分析。左图展示了迁移 project 的数据和带宽消耗的关系，右图展示了迁移 0 个、20 个、50 个 project 在 k-probe 算法和无限存储的最优解情况下的带宽消耗。

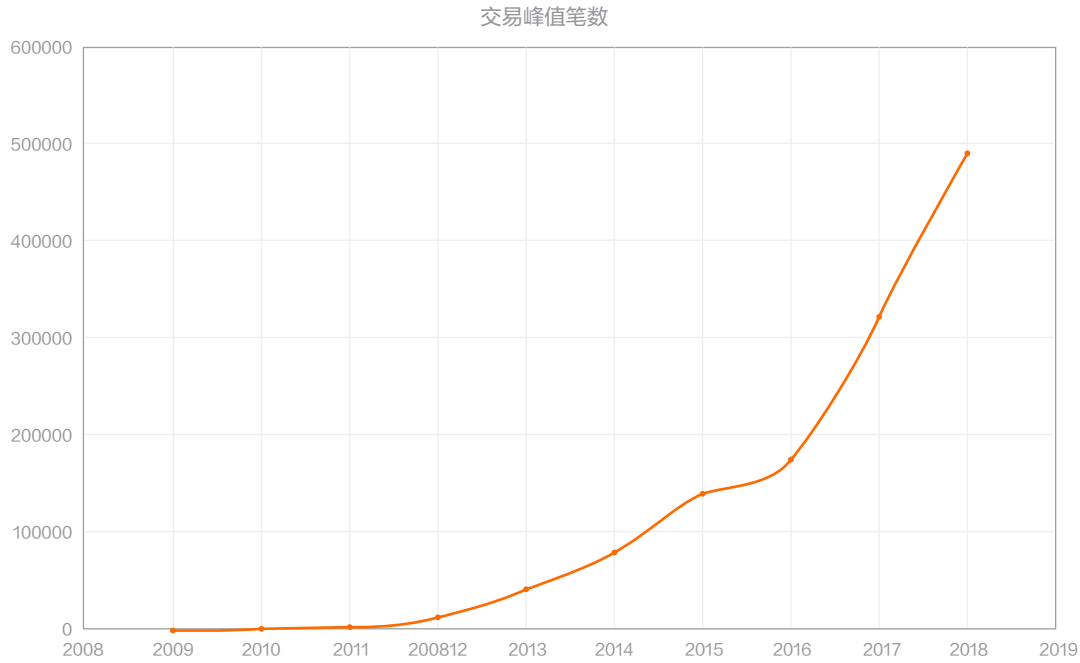




2-3. 支撑每秒十万级峰值交易——阿里规模化混部技术

作者 程正君 | 阿里云智能 技术专家

如图是阿里交易双11历年峰值笔数变化，其中 2018 年双 11 的峰值达到 49.1W 笔/秒，支撑如此大流量的背后，是巨大的机器成本。如何减少大促带来的成本增长一直是阿里巴巴调度团队重点探索解决的问题之一。本文重点介绍阿里巴巴各基础设施系统如何群力合作，为混部打下基础，共同推进混部技术的发展，并将混部应用到大促场景中，最终在不影响离线大促资源需求的情况下，通过大促当天快上快下的方式，成功支撑了在线 28% 的峰值交易，极大的减少了双 11 的机器成本。



2-3-1. 混部的历程

成本 1+1<2

从阿里在线调度来看，除了调度自身优化以外，机器资源成本降低有两个很大的提升空间

- 集团常态化整体机器的利用率很低，并且全天利用率各时间段相差较大；

- 大促峰值需要购买大量的资源来支撑压测和大促当天峰值，大促后将会有大量的资源浪费。

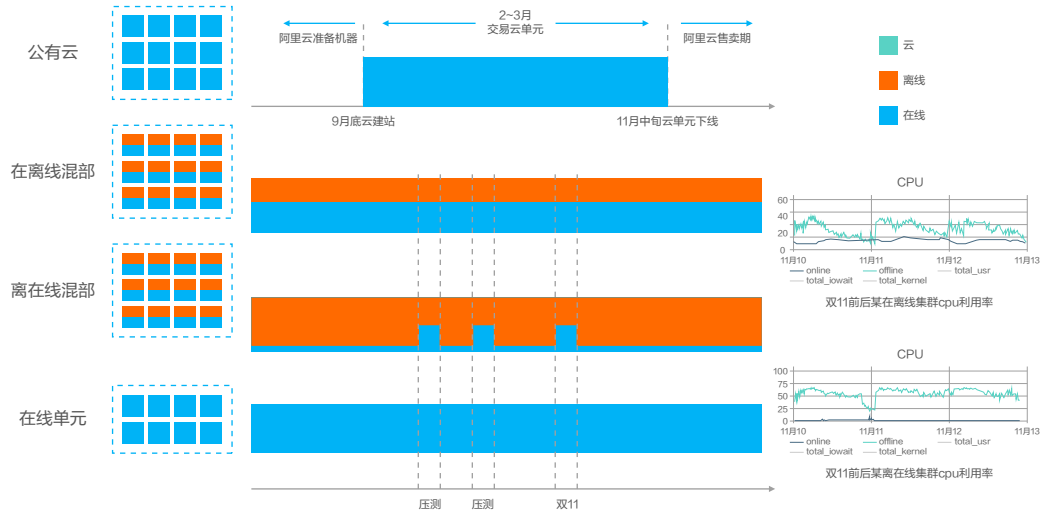
两个问题其实根因一致，都是流量在时间上分布不均，并且峰值流量很难精准预测。调度因此需要为各个业务准备足够的冗余资源来应对，资源其实并没有有效利用起来。正是看到这块问题，所以在线调度在推进资源池统一管理融合的基础上，一方面，寻求能够互补的弹性容量来覆盖大促成本，另一方面提供弹性容量，来提高常态化资源利用率，利己利他。

而纵观阿里内部，最显著的具备互补性的资源有三大块：在线，离线和阿里云。从在线侧来看，2014 年是弹性容量解成本的元年，两套方案同时开展。

1. 混合云弹性容量：让一部分流量跑在公有云资源上，与阿里云形成互补，大促后，利用阿里云公有云的售卖能力，消化掉这部分资源；

2. 混部：各基础设施共同启动混部技术研究，对离线和在线进行混合部署，在离线互为弹性容量，实现常态利用率提升（在离线混部），大促总体成本下降（离在线混部），同时减少离线夜间峰值压力以及成本增加。

归根结底：通过预算节奏上互补或者资源使用时间以及 SLO 上的互补，来实现阿里集团内不同部分资源成本上的1 +1<2。而针对双 11 场景，两个方案叠加才能满足业务的峰值资源需求，我们可以看看具体是如何实施的。



上图可解释混合云之外，为什么阿里集团重点推进混部的发展以及应用到双 11。其中公有云支撑双 11，对齐了阿里云和集团的预算，减少了阿里机器的整体成本，但同时也需要看到这也依赖阿里云的售卖能力，如果不能及时消化，反而会造成浪费，

另外整个大促备战期间长达 2 个多月，时间很长，期间大部分时间资源浪费也比较严重。而从离在线混部的资源变化看，两侧通过离线在线间快速的资源腾挪，可以实现在线以小时为单位的资源使用。另外这里补充说明下除了离线为在线大促提供了混部机器资源（离在线混部），支撑了在线大促的流量，为阿里节约了大促成本，而在线同样在常态时间里，拿出机器资源和离线进行混部（在离线混部），为离线整体节约了常态化的机器成本。

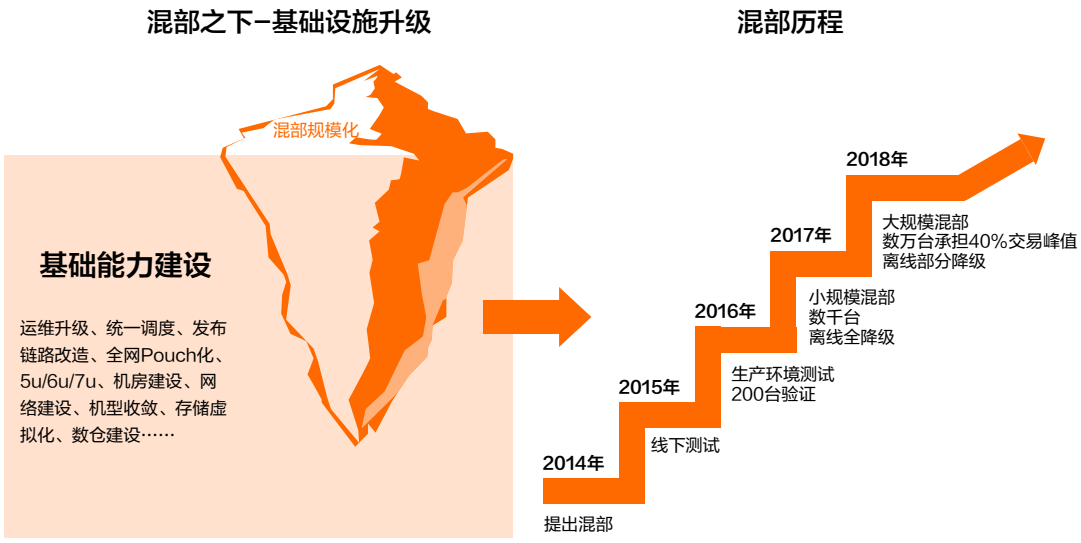
混部发展路径

清楚了混部对于阿里集团整体常态化资源利用率提升以及大促成本下降的意义，我们再看看混部技术是如何一步步发展过来的。

为什么阿里 2014 年才开始真正的做混部?

- 基础设施发展侧重点不同：早之前阿里集团基础设施也是在不断发展和完善，处于快速发展阶段，支撑业务快速增长则为第一要务；

- 基础设施能力储备不够，条件不成熟：混部是一个非常庞大的工程，需要非常强的基础设施支持，需要非常多的铺垫，如离线规模要足够大、内核的隔离能力，网络能力、存储计算分离、在线统一的资源调度、容器化、机房布局、机型收敛等等。各个基础设施团队都在默默推进向前发展，也使得混部云化能够稳步落地，最终于 2018 年达到规模化的目标。

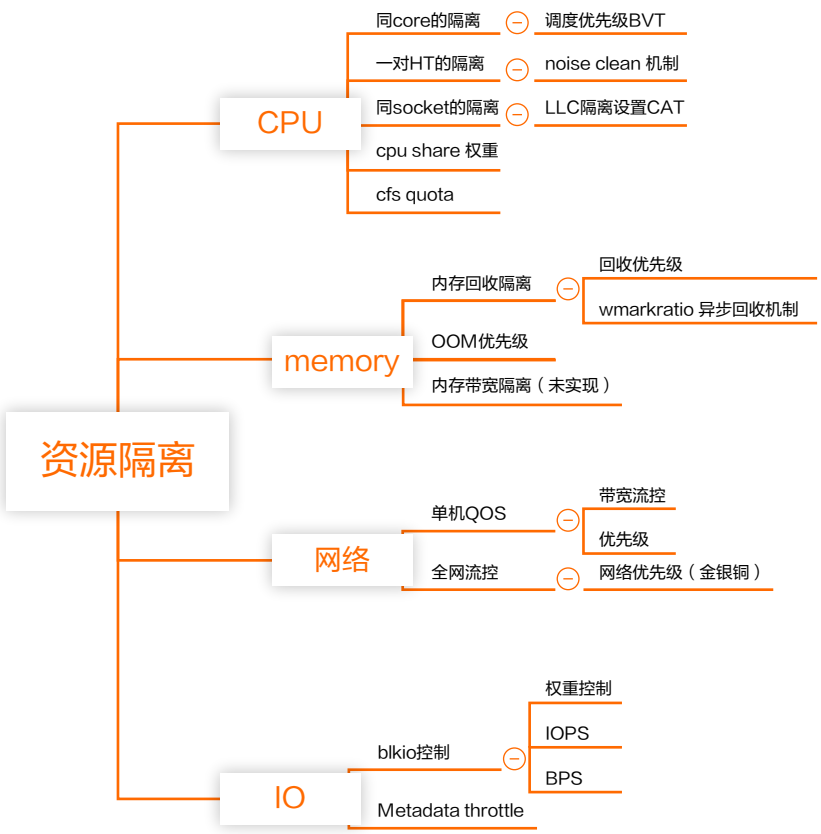


2-3-2. 混部基础能力建设

混部首先解决的是可以混，在此基础上更进一步做到混在一起，分别对应的核心问题则是资源隔离和调度。

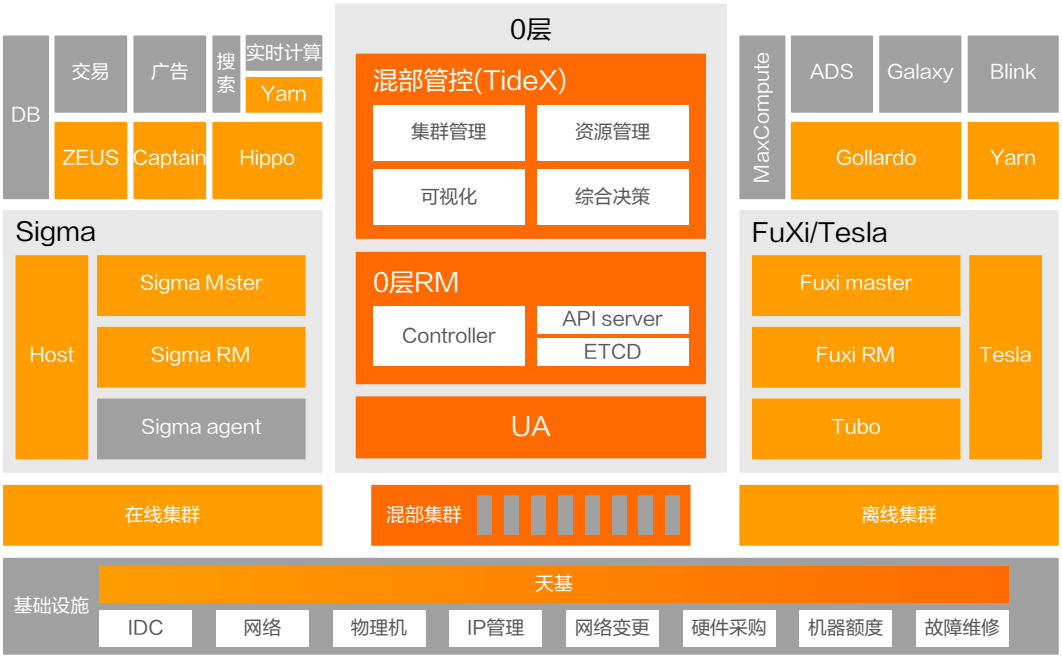
隔离能力

可以混的前提是确保两侧业务的 SLO，在线是延迟敏感型，故需要的是瞬时资源的高优先级；离线则延迟没那么敏感，但需要的是在一段时间内的资源总量的稳定保障，避免长尾问题，确保基线产出。单机隔离能力是业务能够混的前提，无论是否规模化混部，优先要解的都是资源瞬时的隔离能力和优先级能力，对应到 CPU, memory，网络，IO 上我们可以看看都做了什么?

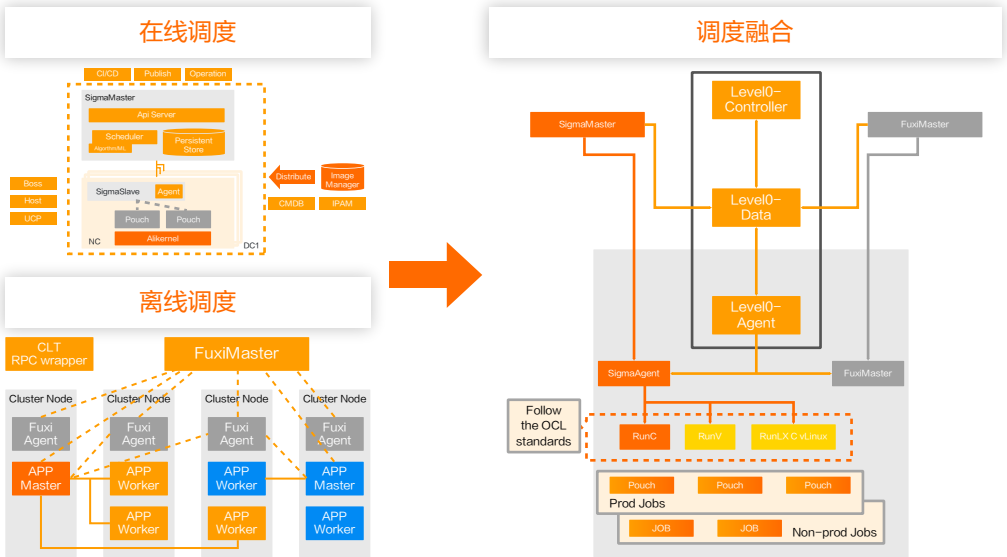


调度能力

调度能力建设更为广泛一点涉及资源池统一管理，容器化改造，调度融合，运维融合等等。这里主要介绍调度的融合，这也是小规模或者大规模的进行混部的基础。



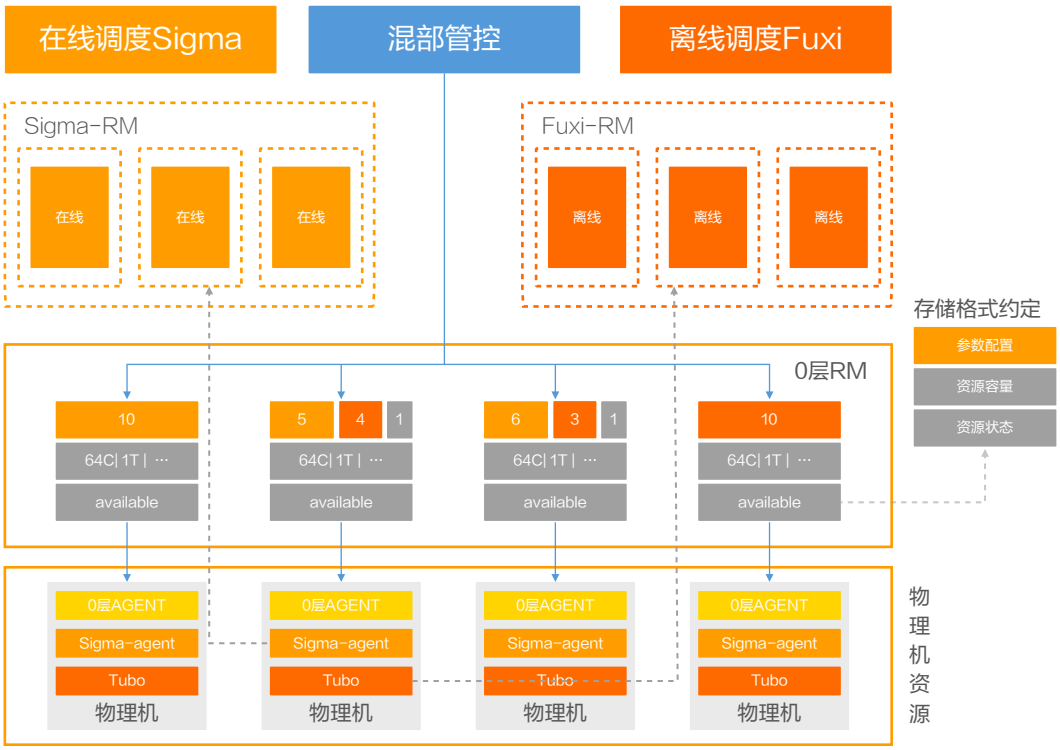
如上图是混部的整体架构，是基于阿里集团调度现状提出来的全面融合方案。由于两个调度积累非常深厚，所以是没有办法直接使用类似于 Borg 的方式统一调度，或者直接采用类似于 Mesos 这样的框架方案。而上面的方案看起来也比较清晰，共建 0 层资源层，作为两边 1 层的资源视图，实现资源层的解耦协调和统一管理。



可以看到，方案的实施保证了两个调度的独立性，从 Agent 到资源层再到运维管控等，都是抽象解耦，以协调管理为主，而未来更进一步演化，是可以最终从各层融合到一起的。

混部 V1

上面混部的基础框架指导了整体混部的实施方案，而实现上我们也以此为基础一步步演化。我们看下第一版的混部如下支撑了 2017 年小规模的混部。



我们可以看到，混部 V1 直接由混部管控 Tidex 管理资源分配，对内存，CPU 资源进行设置，这里 CPU 在线离线通过 Cpuset 的方式绑定到同样的核上，区别在于隔离策略上离线优先级比较低，但离线有重要任务需要高优先级资源保障，这也是为什么提出资源优先级方案的原因。混部 V1 的模式资源其实是静态 offer 的模式，两个调度没有太多直接交互，并且没有优先级的体系和 Quota 管理，当大规模下，很可能会导致调度资源管理出现问题，导致业务 SLO 无法保障。下面我们在混部规模化的讨论里看看是如何去解决的。

2-3-3. 混部规模化做了什么？

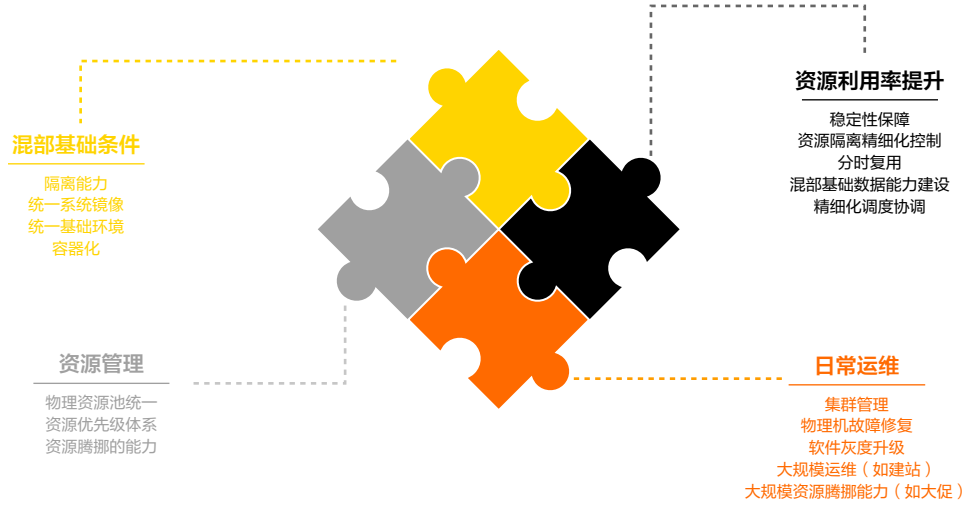
我们先看和 2017 混部 V1 相比，我们混部规模化的目标：

- 在离线为离线节省几千台的资源，并承担在线 14% 的交易峰值；
- 离在线为双 11 支撑在线 28% 的交易。

SLA：

- 在线业务性能影响日常在 10% 以内，压测与大促峰值在 5% 以内；
- 在离线集群内 10% 的离线资源保障；
- 保障离线在压测和大促时不整体降级。

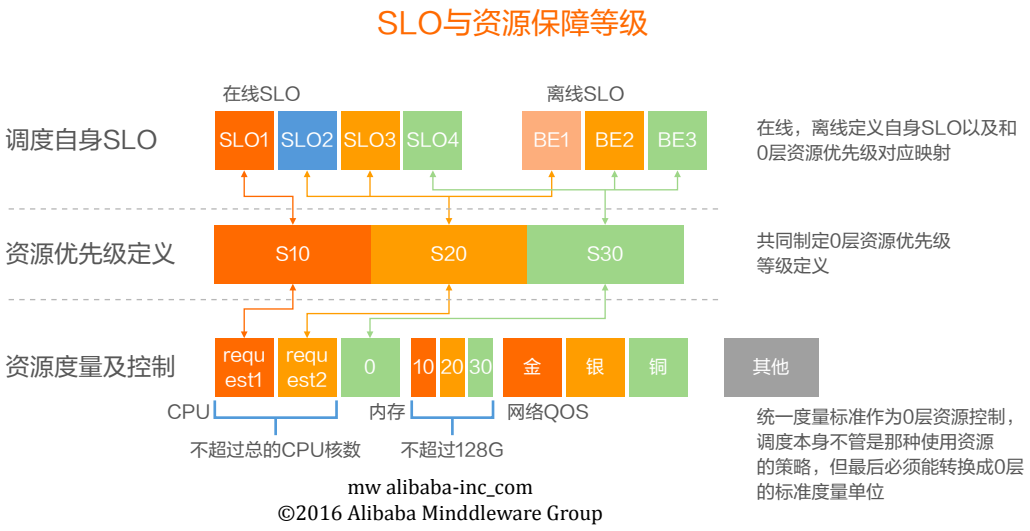
以此看看大规模混部要解决的问题：



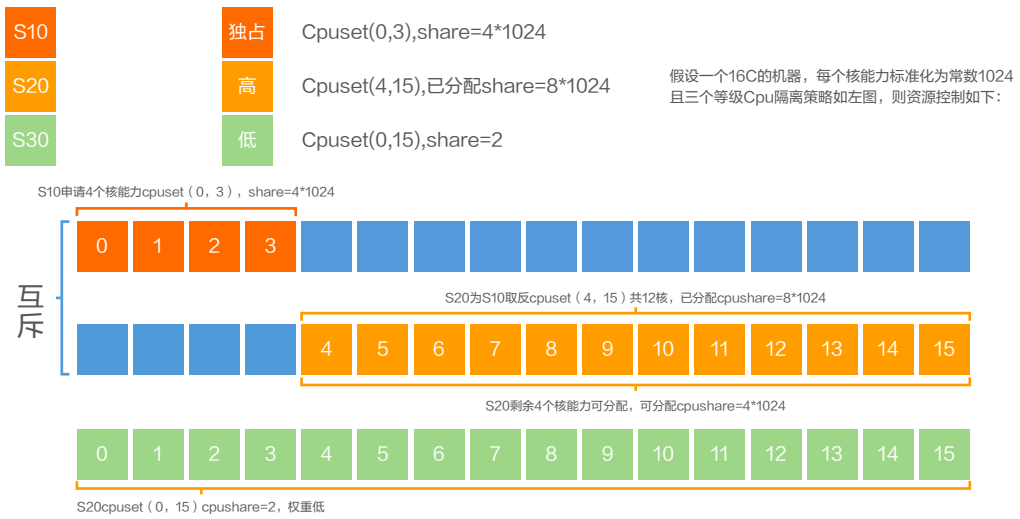
以上问题，有些是未来我们需要持续去解决的，而有些则是我们规模化的基础条件，针对这些问题，我们在保持上文介绍的基础架构的基础上，对局部架构方案进行调整，也使得混部整体进入新的阶段。这里会着重介绍下资源优先级的体系，基础运维的融合以及内核提供的能力增强等。

混部新架构–资源优先级体系

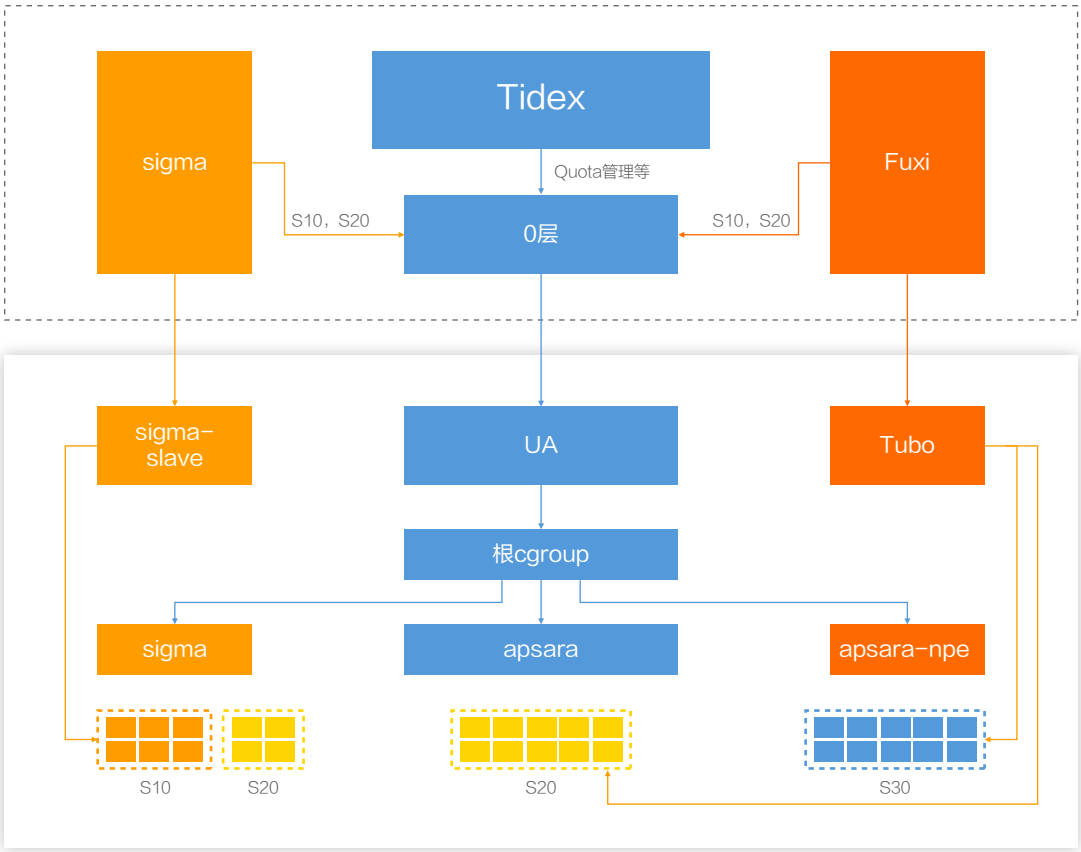
上面我们在混部 V1 里探讨了资源管理的问题，V1 的方案比较简单明了，就是在线完全高优先级，而离线则没有太多保障，而这显然不合理，比如离线的 8 级基线任务非常重要，是下游任务基础依赖，不希望长尾拖累整体进程，必须要优先级比较高的资源来保证。我们看看是如何通过资源优先级来解决的。在线离线针对这种情况，在离线两侧提出在 0 层架构中实现一套资源优先级，并将两侧自定义的 SLO 对齐到各个资源的优先级中来。



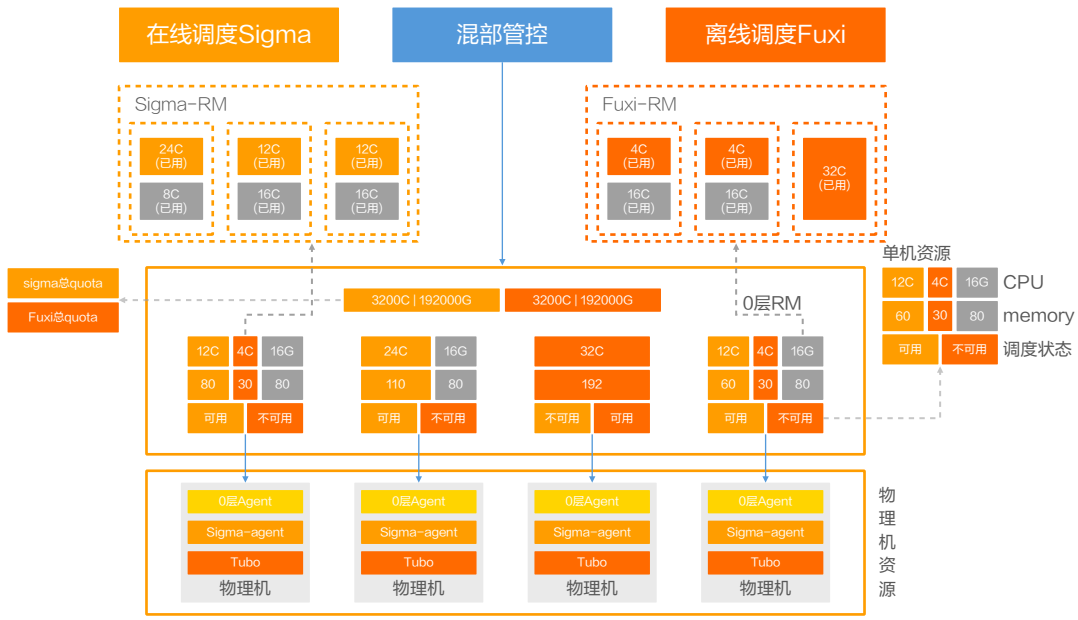
以下以 CPU 的优先级为例，我们来看看是怎么做的



这里 S10 对应的一般是在线延迟特别敏感的业务，S20 则采用 cpushare 的方式，对应的是在线普通业务和离线重要任务，S30 则对应离线普通任务和超卖任务。目前针对资源的优先级机制，我们调整了 Tidex、0 层、UA 以及调度之间的关系，形成新的资源管理架构。



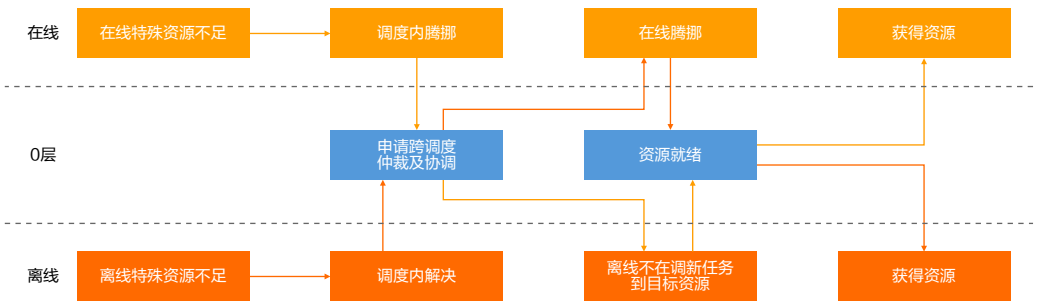
其中 UA 对 sigma，apsara，apsara-npe 三个 cgroup 根组进行管理，设置根组的资源，而调度再自行管理子组。这里和 V1 的资源视图比较，新的资源视图变为



资源优先级体系建设使得混部从资源静态 offer 式迈向类似于共享式的方式。

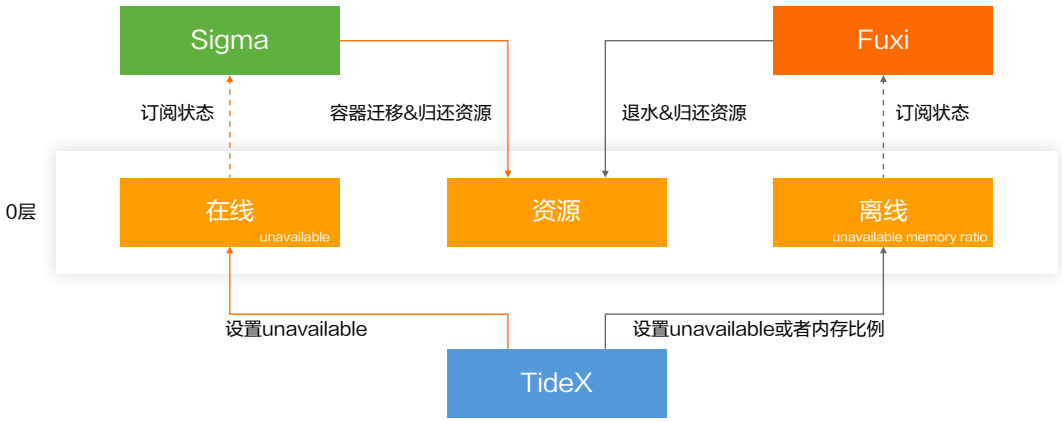
混部新架构–腾挪能力

考虑场景一：当混部规模大了之后，在线和离线之间会出现当前的资源分布不符合需求的场景，比如 Fuxi 有了新任务，需要大规格的 S20 CPU，或者 Sigma 对某部分机器有偏好，希望 Fuxi 出让该部分机器的部分资源给 Sigma。对这种混部腾挪能力，我们在 0 层增加了一层腾挪协议，由两个调度配合做腾挪并归还资源操作，最终达到资源转让腾挪的能力。



mw alibaba-inc.
©2016 Alibaba Minddleware Group

考虑场景二：离在线集群大促建站、压测以及大促峰值时差，需要资源（这里主要指内存）在在线离线间大规模的腾挪，因此需要有大规模的资源腾挪能力，而且涉及的每台宿主机可能资源需求不同，因此我们对单机增加了一些管控能力，这是大促态分时复用以及一些人工资源腾挪的基础能力。



内核–能力增强

2017 年双十一的混部，我们大促峰值期间，其实离线是全降级的，实质意义上，

这个瞬间，其实没有真正混部。这样做的原因一方面离线的资源规模不大，可以接受全降级，另外稳定压倒一切，我们认为混部还需要更全面的技术验证。2018 年进一步增强内核各项能力，有了更加全面的验证计划，另外离线提供的规模远远大于去年，必须要确保 7、8 级基线任务的产出，因此 2018 年我们做到了真正意义上的混部；最终我们也做到了混部集群在线业务 SLO 的保证，兼顾了两边业务的资源需求。这里重点讲下内核在内存以及磁盘隔离方面所做的工作：

内存

其实 2018 年规模化混部，我们也遇到比较多的稳定性问题，对于业务的影响主要表现在 Load 偶发性飚高，困扰业务开发同学，有时甚至影响到业务的 RT。内存 2018 年的优化有：

- 离线调度调整内存申请并发度，尽量避免高并发申请内存，从而触发内存大面积回收；
- 离线退出 pagecache 回收；
- 内核提供整机异步回收优化，避免整机的直接回收；
- 增加子组异步回收水位 wmarkratio 设置；
- 内核升级了 10 几个 hotfix，解决了内存回收死锁，cgroup 清理等等问题；
- 冷内存回收方案。

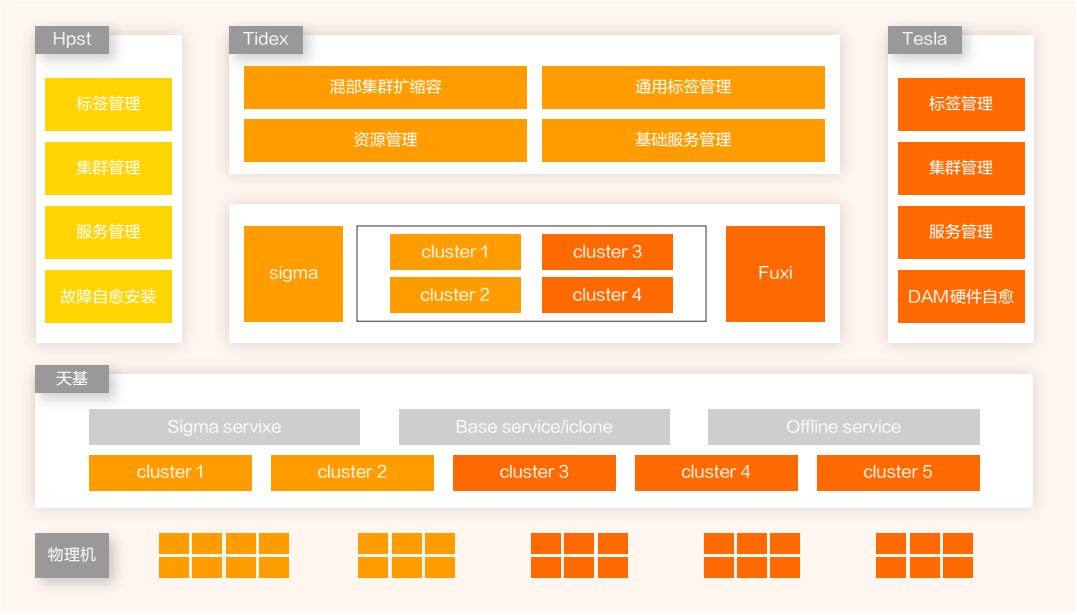
磁盘 IO

2018 年双 11 针对不同机型对离线进行了不同的 blkio 限制方案，第一次对在离线间的 IO 干扰上具备了一定的隔离确定性。

混部新架构-运维融合

在线和离线在混部之前，一直都有自己的运维体系，在线以 Sigma-host 为宿主机

运维，往上则是 Atom 以及 PSP 等面向容器和业务的运维体系。而离线则以天基为基座，以 Tesla 为面向业务的运维体系。混部的机器部署了两侧业务，会涉及到两侧运维系统共同管理，必然会面临一系列运维冲突问题。这里我们以运维融合为最终目标，对于不能短期融合的部分，采用解耦协调的思路，优先实现两侧运维系统的打通，解决运维冲突。



上图是混部规模化所做的运维融合的部分工作。主要：

1.定义集群：如上图所示，共同使用天基为运维底座，对齐混部调度集群和天基集群概念，使得运维所看到的集群和调度 Quota 管理的集群概念统一；

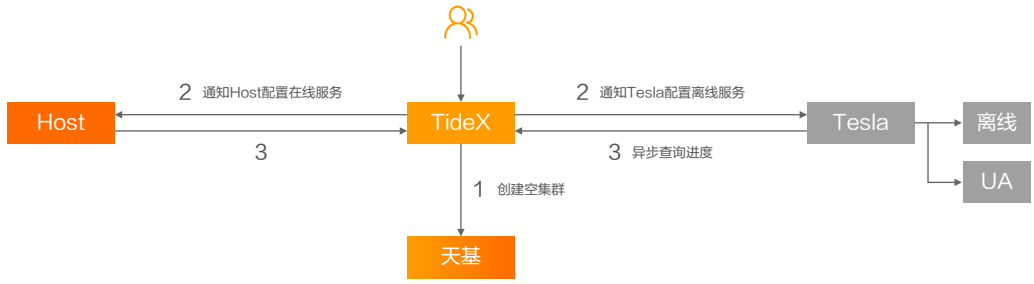
2.基础环境运维统一：在线和离线将基础服务统一交给天基进行安装升级维护，确保安装服务载体统一，防止冲突，这块在线运维系统 sigma-host 做了比较多的改造；

3.集群机器导入：利用 Tidex 为集群管理的中间系统，串接两侧运维系统，实现集群的扩缩容的自动化；

4.故障机维修：利用天基 DAM 维修机制，实现故障机维修流程的对接打通。

集群生命周期

这里集群概念对齐和基础环境运维统一是基础，看看是如何解决的：



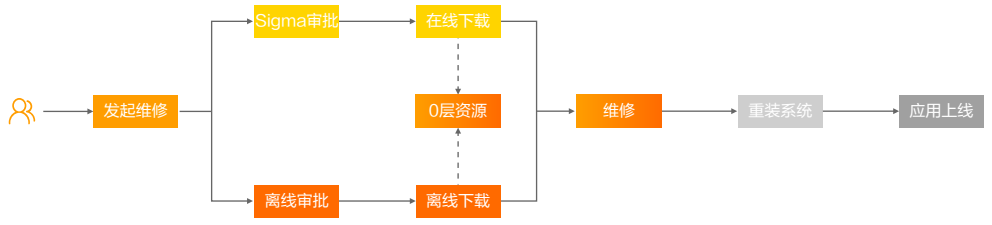
可以看到，通过将天基、离线 Tesla、在线 Host 系统打通，信息通过系统间流转，不再通过人肉传递，避免集群创建错误或者在线离线配置基线等信息不一致问题。

机器生命周期

集群创建和基础环境问题解决后，要解决的是混部集群机器扩缩容的问题，这个部分放到下面双 11 混部资源交付里进行讨论。

机器维修

机器导入混部集群后，我们再看看怎么解决日常的故障机维修流程，我们知道机器维修或者做一些冷升级，需要对机器上的作业进行迁移，原来两侧运维只需要关心自己的业务，但混部之后，则面临两侧业务的协调迁移，才能进行维修。这里我们共用天基 Decider 事件来进行故障流程审批，利用天基 DAM 维修机制为基础，实现维修流程和任务迁移的统一协调。



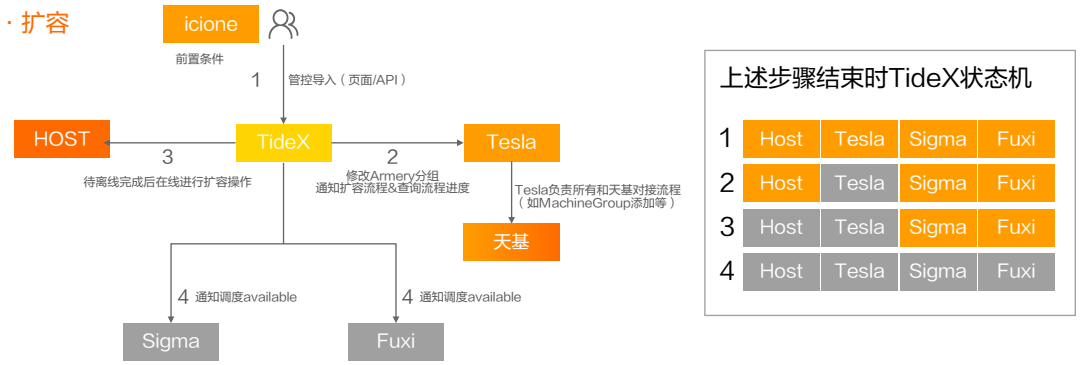
2-3-4. 混部和双 11

文章开头我们也说了双 11 是混部最大的使用场景之一。如果说在离线混部（在线出机器）作为常态化的混部方案，解决了集团日常机器资源利用率问题，帮助在线离线整体上节约了成本。那离在线混部（离线出机器）就是为大促而生，帮助大幅减少了集团大促的成本。从前面大促的资源变化情况，我们可以看出大促涉及两个比较大的问题，一个是短期快速的混部资源交付，一个是压测和大促峰值时大范围的资源腾挪以及快速部署业务的能力。

混部资源交付

混部集群宿主机的扩缩流程之前一直是在线离线分别安装的模式，流程里有非常多的人工流转环节，容易出现基础环境不一致，信息流转脱节等问题，效率通常以周记，大促资源交付风险非常大。再就是 2018 年混部规模非常大，需要交付万级的物理机资源。

前面我们在混部运维融合部分也讨论了集群创建以及流程化串接基础服务到天基的能力，遗留了混部集群扩缩容部分内容，放在这里介绍。混部扩缩容流程化方案并没有完全融合两边的机器导入流程，因为两个调度的导入流程是比较复杂的，短期融合也不太现实。因此我们由 Tidex 串接起 Tesla 和 sigma-host 来将机器导入到 0 层以及两个调度，并解决导入流程里冲突点。这里以扩容为例：



这个流程可以看到和集群创建流程配合，很好的解决了混部集群及机器的生命周期管理，减少了很多人工流转环节，确保了混部资源交付流程里的信息透明，将导入周期从周降低到天，很好的保障了混部的资源按时交付。但同时我们也看到了，由于整体流程里还有比较多的人工重试审批流程，以及面向工单模式的卡单问题等，导致长尾机器影响整体交付效率，因此后续再规划里我们会讨论进一步优化的方案。

其实除了宿主机资源交付，还有在线快速建站的能力，这个可以去关注大促建站的相关分享。另外在线建站过程中需要资源快速大规模腾挪的能力，和大促以及压测分时复用非常像，因此这部分我们放到大促态分时复用流程里讨论。

大促大规模快速资源腾挪–大促态分时复用（离在线集群）

压测和大促峰值期间，我们需要在线离线之间进行资源腾挪，并快速上下线两侧业务，我们将这个整体过程称之为大促态分时复用。这个流程的效率非常关键，如果流程执行时间过长，离线失去资源时间过长，可能会造成离线破基线，并减小压测时间，导致压测轮次变多；而大促当天则影响更大，会打乱业务预热的计划，后果更不可估量。

上图是大促整体的执行流程，其中大促态分时复用占据了头尾部分，执行效率非常关键，如果有偏差，则可能导致在线或者离线的资源无法保障，从而引起大规模故障。这其中离线任务的快速优雅退出，在线业务的快速拉起是整个效率提升的关键，我们做了比较多的优化，比如应用停但镜像保持更新这样的优化，可以减少镜像更新带来的时间消耗，保证在线能够快速拉起。

2018 年混部双 11 的表现

整体表现：电商 MaxCompute 混部在离线支撑 14% 的交易峰值，另外离在线大促态分时复用为双11支撑了 28% 的交易峰值，为大促节省了非常大的机器成本。

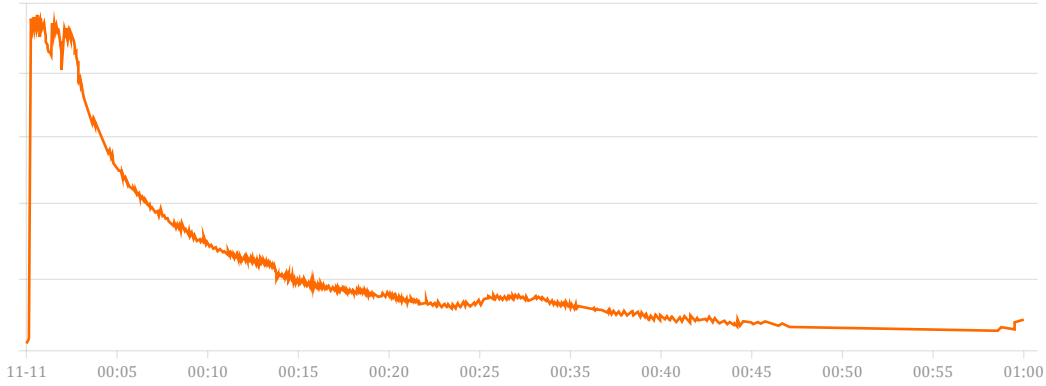
大促态分时复用表现（离在线集群）

- 快上 2 小时：离线退水+退资源 0.5 小时；在线资源申请（0.5 小时）+应用快上整体 1.5 小时；

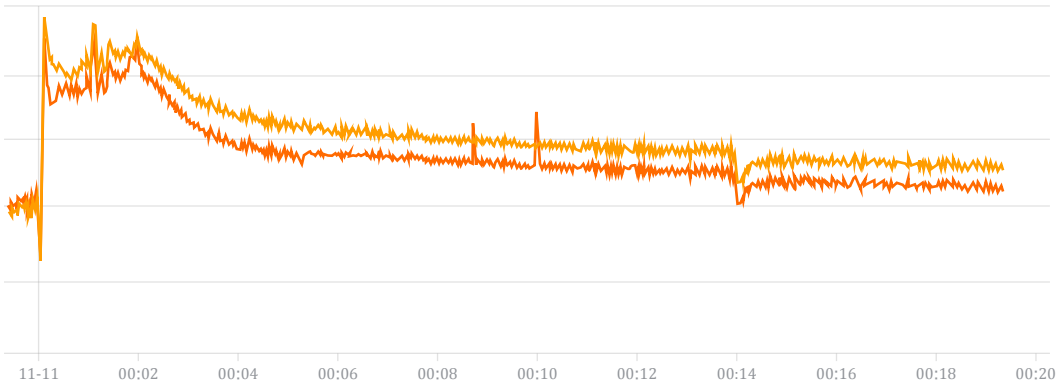
- 快下 0.5 小时：在线快下+资源归还+离线上任务整体 0.5 小时（95% 资源）；长尾 1 小时。

混部集群实际表现

下面是某混部集群大促当天的内存资源变化情况，可以看到大促当天表现符合预期，和平时压测保持一致，达成了支撑峰值交易的目标。



从混部集群对业务的 RT 表现影响来看，也是符合预期。下图是某混部集群和某非混部集群的 RT 对比。其中绿色为混部集群，可以看到混部集群的 RT 表现甚至好于非混部集群，当然这并不表示混部带来了 RT 的提升，但至少能个显示混部对业务的 RT 影响是符合预期。



2-3-5. 更高的要求：全网混部

当前的混部规模化方案还只是打下基础，而未来混部目标是全网混部，帮助集团利用率提升。一方面规模更大，涉及业务更加多元化，调度融合，运维融合的方案要更加高效，分层要更加清晰。另外一方面利用率提升要达到更高目标，这也使得混部进入深水区，涉及到精细化的混部调度，更确定性的内核隔离能力，网络建设，存储计算分离以及机房规划。

基于“伏羲 2.0”的行业最佳实践

基于“伏羲 2.0”的行业最佳实践

本文作者 喻奎 、汤志鹏 、CHEN, Yingda、王家忙、冯亦挥、王璀 | 阿里云智能

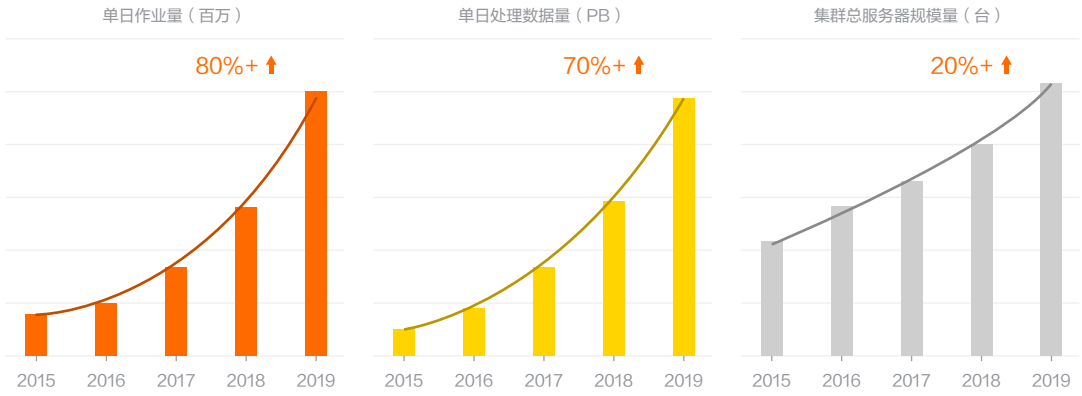
伏羲（Fuxi）是十年前创立飞天平台时的三大服务之一（分布式存储 Pangu，分布式计算 MaxCompute，分布式调度 Fuxi），当时的设计初衷是为了解决大规模分布式资源的调度问题（本质上是多目标的最优匹配问题）。

随着阿里经济体和阿里云业务需求（尤其是双十一）的不断丰富，伏羲的内涵也不断扩大，从单一的资源调度器（对标开源系统的 YARN）扩展成大数据的核心调度服务，覆盖数据调度（Data Placement）、资源调度（Resource Management）、计算调度（Application Manager）、和本地微（自治）调度等多个领域，并在每一个细分领域致力于打造超越业界主流的差异化能力。

过去十年来，伏羲在技术能力上每年都有新的进展和突破，2013 年 5K，2015 年 Sortbenchmark 世界冠军，2017 年超大规模离在/在离线混部能力，2019 年的 Yugong 发布并且论文被 VLDB2019 接受等。随着 Fuxi 2.0 首次亮相 2019 双 11，2019 年 飞天大数据平台在混部侧支持和基线保障两个方面均顺利完成了目标。其中，混部支持了双十一 60% 在线交易洪峰的流量，超大规模混部调度符合预期。在基线保障方面，单日数据处理 970PB，较 2018 年增长超过 60%。在千万级别的作业上，不需要用户额外调优，基本做到了无人工干预的系统自动化。

3-1. 双 11 带来的全新挑战

随着业务和数据的持续高速增长，MaxCompute 双十一的作业量和计算数据量每年的增速都保持在 60% 以上 。2019 年双十一，MaxCompute 日计算数据量规模已接近 EB 级，作业量也到了千万量级，在如此大规模和资源紧张的情况下，要确保双十一稳定运行，所有重要基线作业按时产出压力相当之大。



MaxCompute 历年双十一增长曲线

在双十一独特的大促场景下，2019 双 11 的挑战主要来自以下几个方面：

1. 超大规模计算场景下，以及资源紧张的情况下，如何进一步提升平台的整体性能，来应对业务的持续高速增长；

2. 双十一会给 MaxCompute 带来全方面超压力的极端场景，如几亿条的热点 key、上千倍的数据膨胀等，这对集群磁盘 IO 的稳定性、数据文件读写性能、长尾作业重跑等各方面都是挑战；

3. 近千万量级作业的规模下，如何做到敏捷、可靠、高效的分布式作业调度执行；

4. 以及对高优先级作业（如重要业务基线）的资源保障手段；

5. 2019 年也是云上集群首次参与双十一，并且开始支持混部。

3-2. 3 项调优化解 EB 级数据压力

为了应对上述挑战，与往年相比，除了常规的 HBO 等调整之外，飞天大数据平台加速了过去 1-2 年中技术积累成果的上线，尤其是 Fuxi 2.0 首次亮相双十一，最终在单日任务量近千万、单日计算量近千 PB 的压力下，保障了基线全部按时产出。

- 在平台性能优化方面，对于挑战 #1 和 #2，StreamlineX + Shuffle Service 根据实时数据特征自动智能化匹配高效的处理模式和算法，挖掘硬件特性深度优化 IO，

内存，CPU 等处理效率，在减少资源使用的同时，让全量 SQL 平均处理速度提升将近 20%，出错重试率下降至原来的几十分之一，大大提升了 MaxCompute 平台整体效能；

- 在分布式作业调度执行方面，对于挑战#3，DAG 2.0 提供了更敏捷的调度执行能力，和全面去阻塞能力，能为大规模的 MR 作业带来近 50% 的性能提升。同时 DAG 动态框架的升级，也为分布式作业的调度执行带来了更灵活的动态能力，能根据数据的特点进行作业执行过程中的动态调整；

- 在资源保障方面，为应对挑战 #4，Fuxi 对高优先级作业(主要是高优先级作业) 采取了更严格、更细粒度的资源保障措施，如资源调度的交互式抢占功能、基于时间预估的全局最优抢占、高优先级作业全局Quota保障等。目前线上最高优先级的作业基本能在 90s 内抢占到资源；

- 其他如业务调优支持等：如业务数据压测配合，与作业调优等。

3-3. 关键作战模块：StreamlineX + Shuffle Service

挑战

上面提到 2019 年双十一数据量翻倍接近 EB 级，作业量接近千万，整体资源使用也比较紧张，通过以往经验分析，双十一影响最关键的模块就是 Streamline（在其他数据处理引擎也被称为 Shuffle 或 Exchange)，各种极端场景层出不穷，并发度超过 5 万以上的 Task，多达几亿条的热点 Key，单 Worker 数据膨胀上千倍等全方位覆盖的超压力数据场景，都将极大影响 Streamline 模块的稳定运行，从而对集群磁盘 IO 的稳定性，数据文件读写性能，机器资源竞抢性能，长尾 Worker PVC（Pipe Version Control，提供了某些特定情况下作业失败重跑的机制）重跑等各方面产生影响，任何一个状况没有得到及时的自动化解决，都有可能导致基线作业破线引发故障。

Streamline 与 Shuffle Service 概述

- Streamline

在其他 OLAP 或 MPP 系统中，也有类似组件被称为 Shuffle 或Exchange，在 MaxCompute SQL 中该组件涉及的功能更加完善，性能更优，主要包含但不限于分布式运行的 Task 之间数据序列化，压缩，读写传输，分组合并，排序等操作。SQL 中一些耗时算子的分布式实现基本都需要用到这个模块，比如 join，group-by，window 等等，因此它绝对是 CPU，memory，IO 等资源的消耗大户，在大部分作业中运行时间占比整个 sql 运行时间 30% 以上，一些大规模作业甚至可以达到 60% 以上，这对于 MaxCompute SQL 日均近千万任务量，日均处理数据接近 EB 级的服务来说，性能每提升 1 个多百分点，节省的机器资源都是以上千台计，因此对该组件的持续重构优化一直是 MaxCompute SQL 团队性能提升指标的重中之重。 2019 年双十一应用的 SLX 就是完全重写的高性能 Streamline 架构；

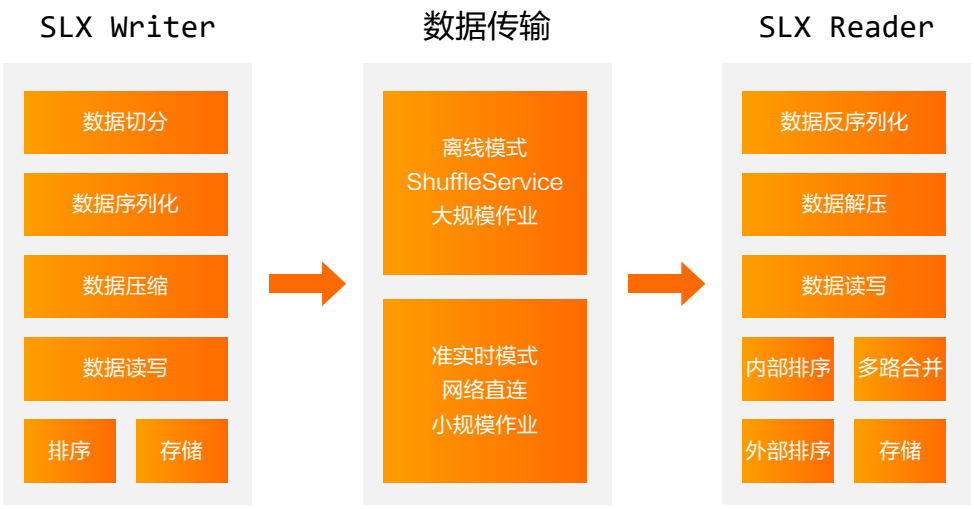
- Shuffle Service

在 MaxCompute SQL 中，它主要用于管理全集群所有作业 Streamline 数据的底层传输方式和物理分布。调度到不同机器上成千上万的 Worker 需要通过精准的数据传递，才能协作完成整体的任务。在服务 MaxCompute 这样的数据大户时，能否高效、稳定地完成每天 10 万台机器上千万量级 Worker 间传输几百 PB 数据量的数据 shuffle 工作，很大意义上决定了集群整体的性能和资源利用效率。Shuffle Service 放弃了以磁盘文件为基础的主流 shuffle 文件存储方式，突破了机械硬盘上文件随机读的性能和稳定性短板；基于 shuffle 数据动态调度的思想，令 shuffle 流程成为了作业运行时实时优化的数据流向、排布和读取的动态决策。对准实时作业，通过解构 DAG 上下游调度相比 network shuffle 性能相当，资源消耗降低 50%+。

StreamlineX + Shuffle Service 关键技术

- StreamlineX (SLX) 架构与优化设计

SLX 逻辑功能架构如下图所示，主要包含了 SQL runtime 层面的数据处理逻辑重构优化，包括优化数据处理模式，算法性能提升，内存分配管理优化，挖掘硬件性能等各方面来提升计算性能，而且底座结合了全新设计的负责数据传输的 Fuxi ShuffleService 服务来优化 IO 读写以及 Worker 容错性等方面，让 SLX 在各种数据模式以及数据规模下都能够有很好的性能提升和高效稳定的运行。



（SLX逻辑功能架构图）

SQL Runtime SLX 主要包含 Writer 和 Reader 两部分，下面简要介绍其中部分优化设计：

1. 框架结构合理划分: Runtime Streamline 和 Fuxi SDK 解耦，Runtime 负责数据处理逻辑，Fuxi SDK 负责底层数据流传输。代码可维护性，功能可扩张性，性能调优空间都显著增强；

2. 支持 GraySort 模式: Streamline Writer 端只分组不排序，逻辑简单，省去数据内存拷贝开销以及相关耗时操作，Reader 端对全量数据排序。整体数据处理流程 Pipeline 更加高效，性能显著提升；

3. 支持 Adaptive 模式: StreamlineReader 支持不排序和排序模式切换，来支持一些 AdaptiveOperator 的需求，并且不会产生额外的 IO 开销，回退代价小，Adaptive 场景优化效果显著；

4. CPU 计算效率优化: 对耗时计算模块重新设计 CPU 缓存优化的数据结构和算法，通过减少 cache miss，减少函数调用开销，减少 cpu cache thrashing，提升 cache 的有效利用率等手段，来提升运算效率；

5. IO 优化：支持多种压缩算法和 Adaptive 压缩方式，并重新设计 Shuffle 传输数据的存储格式，有效减少传输的 IO 量；

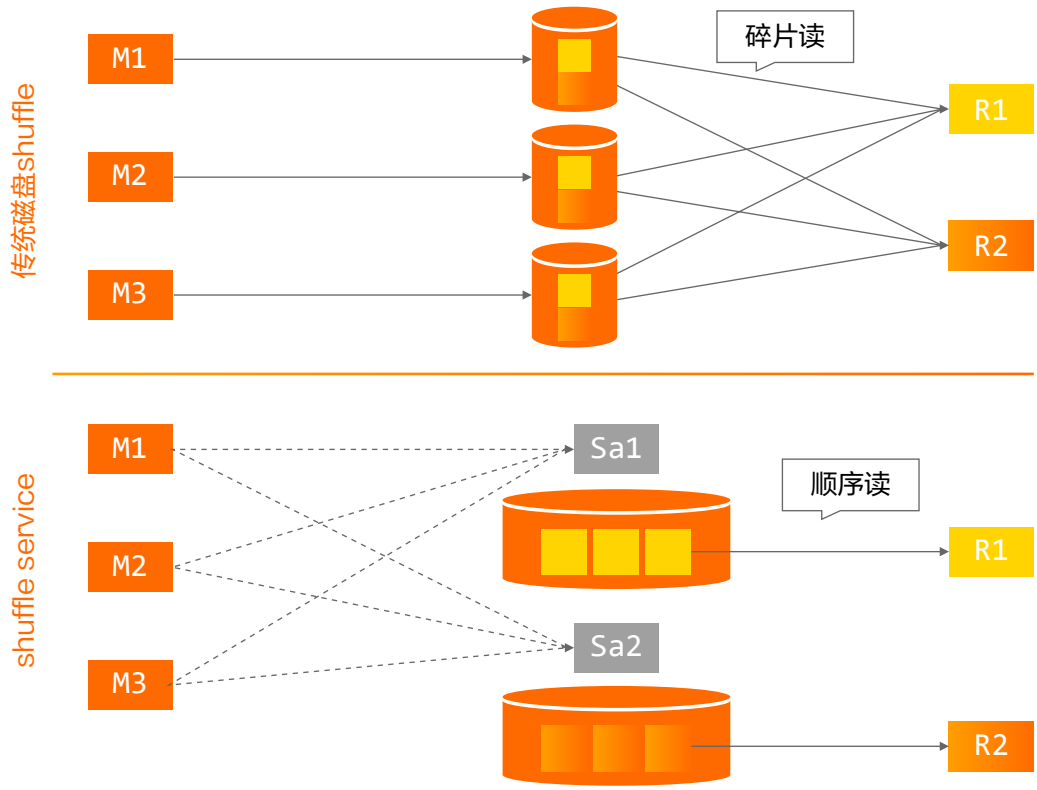
6. 内存优化: 对于 Streamline Writer 和 Reader 内存分配更加合理，会根据实际数据量来按需分配内存，尽可能减少可能产生的 Dump 操作。

根据以往双十一的经验，11 月 12 日凌晨基线任务数据量大幅增加，shuffle 过程会受到巨大的挑战，这通常也是造成当天基线延迟的主要原因，下面列出了传统磁盘 shuffle 的主要问题：

- 碎片读：一个典型的 2k*1k shuffle pipe 在上游每个 mapper 处理 256MB 数据时，一个 mapper 写给一个 reducer 的数据量平均为 256KB，而从 HDD 磁盘上一次读小于 256KB 这个级别的数据量是很不经济的，高 iops 低 throughput 严重影响作业性能；

- 稳定性：由于 HDD 上严重的碎片读现象，造成 reduce input 阶段较高的出错比率，触发上游重跑生成shuffle数据更是让作业的执行时间成倍拉长。

Shuffle Service 彻底解决了以上问题。经过 2019 年双 11 的考验，结果显示线上集群的压力瓶颈已经从之前的磁盘，转移到 CPU 资源上。双十一当天基线作业执行顺畅，集群整体运行持续保持稳定。



Shuffle Service 主要功能有：

- agent merge：彻底解决传统磁盘 shuffle 模式中的碎片读问题；

- 灵活的异常处理机制：相较于传统磁盘 shuffle 模式，在应对坏境异常时更加稳定高效；

- 数据动态调度：运行时为任务选择最合适的shuffle数据来源；

- 内存 &PreLaunch 对准实时的支持：性能与 network shuffle 相当的情况下，资源消耗更少。

StreamlineX + Shuffle Service 双十一线上成果

为了应对上面挑战，突破现有资源瓶颈，2018 年前 MaxCompute SQL 就启动性能持续极限优化项目，其中最关键之一就是 StreamlineX (SLX) 项目，它完全重构了现有的 Streamline 框架，从合理设计高扩展性架构，数据处理模式智能化匹配，内存动态分配和性能优化，Adaptive 算法匹配，CPU Cache 访问友好结构设计，基于 Fuxi Shuffle Service 服务优化数据读写 IO 和传输等各方面进行重构优化升级后的高性能框架。至2019 年双十一前，日均 95% 以上弹内 SQL 作业全部采用 SLX，90% 以上的 Shuffle 流量来自 SLX，0 故障，0 回退的完成了用户透明无感知的热升级，在保证平稳上线的基础上，SLX在性能和稳定性上超预期提升效果在双十一当天得到充分体现，基于线上全量高优先级基线作业进行统计分析：

- 在性能方面，全量准实时 SQL 作业 e2e 运行速度提升 15%+，全量离线作业的 Streamline 模块 Throughput (GB/h) 提升 100%；

- 在 IO 读写稳定性方面，基于 FuxiShuffleService 服务，整体集群规模平均有效 IO 读写 Size 提升 100%+，磁盘压力下降 20%+；

- 在作业长尾和容错上，作业 Worker 发生 PVC 的概率下降到仅之前的几十分之一。

在资源优先级抢占方面，ShuffleService 保障高优先级作业 shuffle 数据传输比低优先级作业降低 25%+。

正是这些超预期优化效果，MaxCompute 双十一当天近千万作业，涉及到近 10 万台服务器节省了近 20% 左右的算力，而且针对各种极端场景也能智能化匹配最优处理模式，做到完全掌控未来数据量不断增长的超大规模作业的稳定产出。上面性能数据统计是基于全量高优先级作业的一个平均结果，实际上 SLX 在很多大规模数据场景下效果更加显著，比如在线下 tpch 和 tpcds 10TB 测试集资源非常紧张的场景下，SQL e2e 运行速度提升近一倍，Shuffle 模块提升达 2 倍。

StreamlineX+Shuffle Service 展望

高性能 SLX 框架经过 2019 年双十一大考绝不是一个结束，而是一个开始，后续我们还会不断的完善功能，打磨性能。比如持续引入高效的排序，编码，压缩等算法来 Adaptive 匹配各种数据 Parttern，根据不同数据规模结合 ShuffleService 智能选择高效数据读写和传输模式，RangePartition 优化，内存精准控制，热点模块深度挖掘硬件性能等各方向持续发力，不断节省公司成本，技术上保持大幅领先业界产品。

3-4. 敏捷、智能，DAG 2.0 如何融入双 11 场景

挑战

双十一大促场景下，除了数据洪峰和超过日常作业的规模，数据的分布与特点也与平常大不相同。这种特殊的场景对分布式作业的调度执行框架提出了多重挑战，包括：

- 处理双十一规模的数据，单个作业规模超过数十万计算节点，并有超过百亿的物理边连接。在这种规模的作业上要保证调度的敏捷性，需要实现全调度链路 overhead 的降低以及无阻塞的调度；

- 在基线时段集群异常繁忙，各个机器的网络/磁盘/CPU/内存等等各个方面均会收到比往常更大的压力，从而造成大量的计算节点异常。而分布式调度计算框架在这个时候，不仅需要能够及时监测到逻辑计算节点的异常进行最有效的重试，还需要能够智能化的及时判断/隔离/预测可能出现问题的物理机器，确保作业在大的集群压力下依然能够正确完成；

- 面对与平常特点不同的数据，许多平时的执行计划在双十一场景上可能都不再适用。这个时候调度执行框架需要有足够的智能性，来选择合理的物理执行计划；以及

足够的动态性，来根据实时数据特点对作业的方方面面做出及时的必要调整。这样才能避免大量的人工干预和临时人肉运维。

2019 年双十一，适逢计算平台的核心调度执行框架全新架构升级– DAG 2.0 正在全面推进上线，DAG 2.0 很好的解决了上述几个挑战。

DAG 2.0 概述

现代分布式系统作业执行流程，通常通过一个有向无环图（DAG）来描述。DAG 调度引擎，是分布式系统中唯一需要和几乎所有上下游（资源管理，机器管理，计算引擎, shuffle 组件等等）交互的组件，在分布式系统中起了重要的协调管理，承上启下作用。作为计算平台各种上层计算引擎（MaxCompute, PAI 等）的底座，伏羲的 DAG 组件在过去十年支撑了上层业务每天数百万的分布式作业运行，以及数百 PB 的数据处理。而在计算引擎本身能力不断增强，作业种类多样化的背景下，对 DAG 架构的动态性，灵活性，稳定性等多个方面都提出了更高的需求。在这个背景下，伏羲团队启动了 DAG 2.0 架构升级。引入了一个，在代码和功能层面，均是全新的 DAG 引擎，来更好的支持计算平台下个十年的发展。

这一全新的架构，赋予了 DAG 更敏捷的调度执行能力，并在分布式作业执行的动态性，灵活性等方面带来了质的提升，在与上层计算引擎紧密结合后，能提供更准确的动态执行计划调整能力，从而为支持各种大规模作业提供了更好的保障。例如在最简单的 MR 作业测试中，DAG 2.0 调度系统本身的敏捷度和整个处理流程中的全面去阻塞能力, 能为大规模的MR作业(10 万并发)带来将近 50% 的性能提升。而在更接近线上 SQL workload 特点的中型（1TB TPCDS）作业中，调度能力的提升能带来 20%+ 的 e2e 时间缩短。

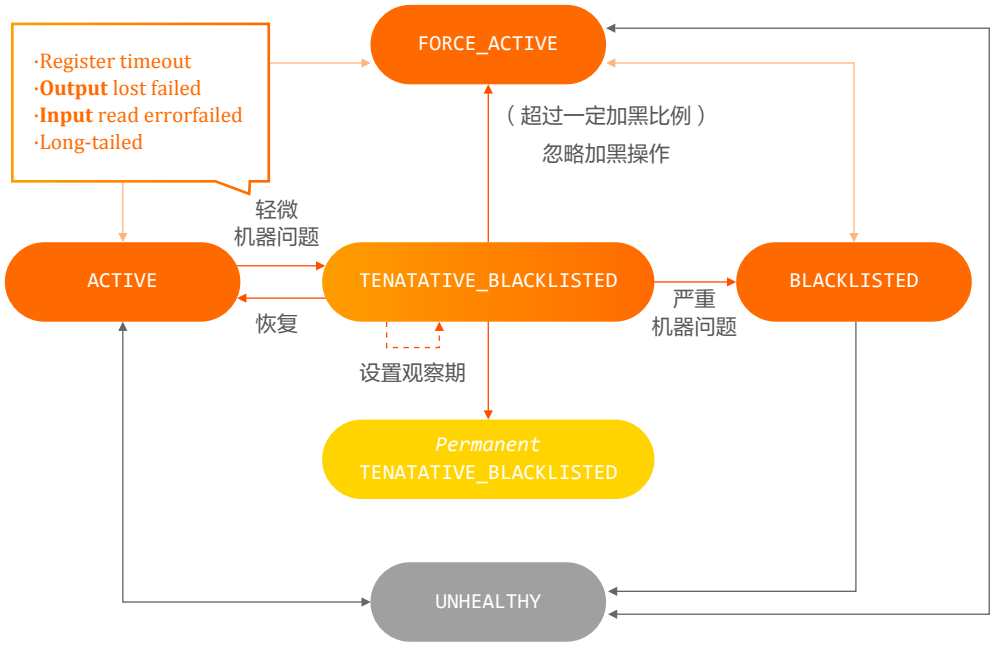
DAG 2.0 的架构设计中结合了过去10年支持集团内外各种计算任务的经验，系统的对实时机器管理框架，backup instance 策略以及容错机制等方面进行了考虑和设计，为支持线上多种多样的实际集群环境打下了重要基础。而另一挑战是，DAG 2.0 架构要在日常数百万分布式作业的体量上做到完全的上线，在飞行中换引擎。从 FY20 财年初开始，DAG2.0 推进线上升级，至今已经实现了在 MaxCompute 离线作业，PAI 平台 Tensorflow CPU/GPU 作业等每天数百万作业的完全覆盖。并经过项目组所有成员的共同努力，在双十一当天交出了一份满意的答卷。

DAG 2.0 关键技术

能取得上述线上结果，和 DAG 2.0 众多的技术创新密不可分，受篇幅限制，这里主要选取和双 11 运行稳定性相关的两个方面做重点介绍。

- 完善的错误处理能力

在分布式环境中由于机器数量巨大，单机发生故障的概率非常高，因此容错能力是调度系统的一个重要能力。为了更好的管理机器状态，提前发现故障机器并进行主动规避，DAG 2.0 通过完整的机器状态管理，完善了机器错误的处理能力：

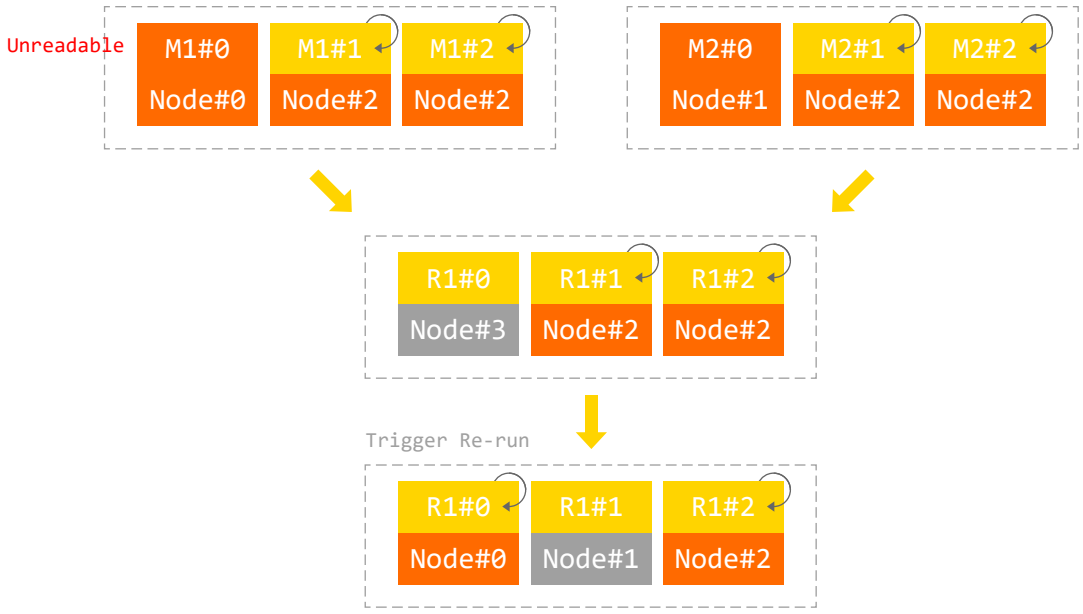


Node State Machine

如上图所示，DAG 2.0 将机器分为多个状态。并根据一系列不同的指标来触发在不同状态之间的转换。对于不同状态的机器，根据其健康程度，进行主动规避，计算任务迁移，以及计算任务主动重跑等操作。将机器问题造成的作业运行时间拉长，甚至作业失败的可能性降到最低。

另一方面，在一个 DAG 上，当下游读取数据失败时，需要触发上游的重跑，而在发生严重机器问题时，多个任务的链式重跑，会造成作业的长时间延迟，对于基线作业的及时产出造成严重影响。DAG 2.0 上实现了一套基于血缘回溯的主动容错策

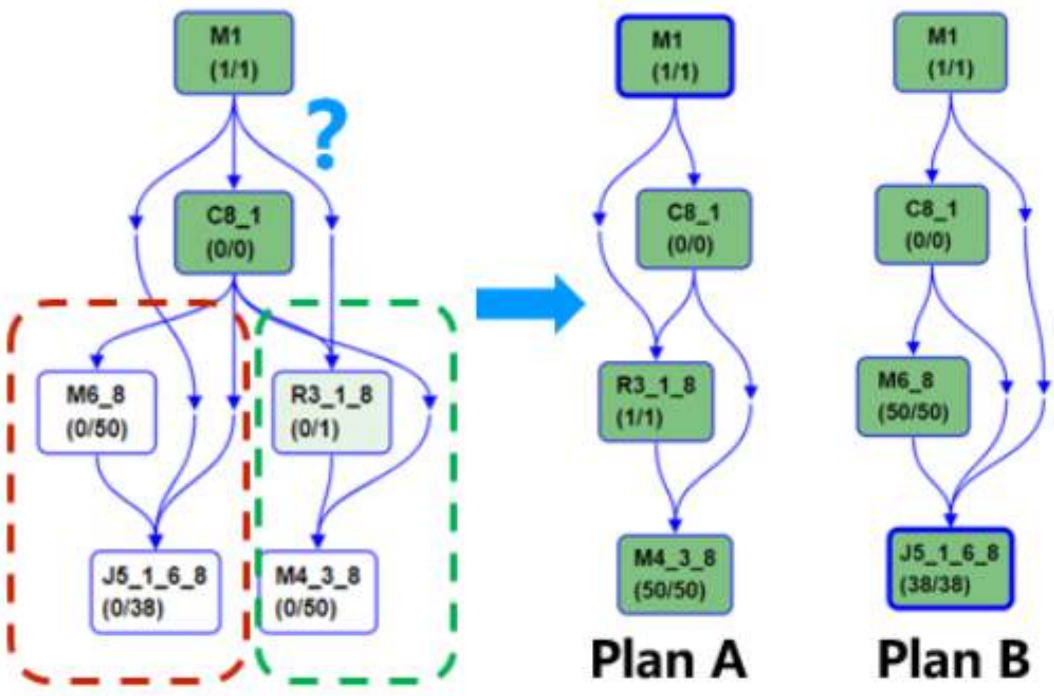
略（如下图），这样的智能血缘回溯，能够避免了层层试探，层层重跑，在集群压力较大时，能够有效的节约运行时间，避免资源浪费。



- 灵活的动态逻辑图执行策略--Conditional join

分布式 SQL 中，map join 是一个比较常见的优化，其实现原理对小表避免 shuffle，而是直接将其全量数据 broadcast 到每个处理大表的分布式计算节点上，通过在内存中直接建立 hash 表，完成 join 操作。map join 优化能大量减少额外 shuffle 和排序开销并避免 shuffle 过程中可能出现的数据倾斜，提升作业运行性能。但是其局限性也同样显著：如果小表无法 fit 进单机内存，那么在试图建立内存中的hash表时就会因为 OOM 而导致整个分布式作业的失败。所以虽然 map join 在正确使用时，可以带来较大的性能提升，但实际上优化器在产生 map join 的 plan 时需要偏保守，导致错失了很多优化机会。而即便是如此，依然没有办法完全避免 map join OOM 的问题。

基于 DAG 2.0 的动态逻辑图执行能力，MaxCompute 支持了 conditional join 功能：在对于 join 使用的算法无法被事先确定时，允许优化器提供一个 conditional DAG，这样的 DAG 同时包括使用两种不同 join 的方式对应的不同执行计划支路。在实际执行时，AM 根据上游产出数据量，动态选择一条支路执行（plan A or plan B)。这样子的动态逻辑图执行流程，能够保证每次作业运行时都能根据实际作业数据特性，选择最优的执行计划，详见下图：



出于对上线节奏把控的考虑，双十一期间 conditional join 尚未覆盖高优先级作业。双十一期间，我们也看到了重要基线上由于数据膨胀导致 Mapjoin hint 失效，作业 OOM 需要临时调参；以及因为 Mapjoin 未能被正确选中，而导致作业未能选中优化执行计划而延迟完成的情况。在 conditional join 在重要基线上线后，能够有效的避免这一情况，让基线的产出更加流畅。

DAG 2.0 双十一线上成果

双十一作为阿里集团所有技术线的大考，对于 DAG2.0 这一全新的组件，同样是一个重要的考验，也是 DAG2 线上升级的一个重要的里程碑：

- 双 11 当天，DAG 2.0 覆盖支持线上 80%+project。截至目前已完成全面上线，日均支持几百万的离线作业执行。对于 signature 相同的基线作业，DAG 2.0 下 instance 运行的 overhead 开销有 1 到 2 倍的降低；

- 双 11 当天，使用 DAG 2.0 的高优先级基线在双十一数据洪峰下，没有任何人工干预的前提下，未发生任何作业失败重跑。其中，DAG 2.0 提供的实时机器管理，backup instance 策略等智能容错机制发挥了重要作用；

- 支持开发环境中近百万量级作业，在作业平均规模更大的前提下，双 11 期间毫秒级（执行时间小于 1s 的作业）分布作业占比相比 1.0 提升 20%+。新框架上更高效的资源流转率也带来了资源利用率的明显提升：等待在线资源超时而回退的在线作业比例降低了将近 50%；

- DAG 2.0 还支持了 PAI 引擎，为双十一期间的搜索、推荐等业务的模型提前训练提供了有力的支持。双十一前 PAI 平台的所有 TensorFlow CPU/GPU 作业，就已经全量迁移到 DAG 2.0 上，其更有效的容错和资源使用的提升，保证了各条业务线上模型的及时产出。

除了基础调度能力提升带来的性能红利外，DAG 2.0 在动态图亮点功能上也完成了新的突破。包括增强 Dynamic Parellism， LIMIT 优化，Conditional Join 等动态图功能完成上线或者正在上线推动中。其中 Conditional Join 一方面保证了优化的执行计划能尽可能的被选用，同时也保证了不会因为错误选择而带来 OOM 导致作业失败，通过运行时数据统计来动态决定是否使用Mapjoin，保证更加准确决策。双十一前在集团内部做了灰度上线，线上日均生效的 conditionl 节点 10 万+，其中应用 Map join 的节点占比超过了 90%，0 OOM 发生。推广的过程中我们也收到了各个 BU 多个用户的真实反馈，使用 conditional join 后，因为能选择最优的执行计划，多个场景上作业的运行时间，都从几个小时降低到了 30 分钟以下。

DAG 2.0 展望

在双十一值班的过程中，我们依然看到了大促场景下因为不同的数据分布特点，数据的倾斜/膨胀对于分布式作业整体的完成时间影响非常大。而这些问题在 DAG 2.0 完备的动态图调度和运行能力上，都能得到较好的解决，相关功能正在排期上线中。

一个典型的例子是 dynamic partition insert 的场景，在某个高优先级作业的场景上，一张重要的业务表直接采用动态分区的方式导入数据导致表文件数过多，后续基线频繁访问该表读取数据导致 pangu master 持续被打爆，集群处于不可用状态。采用 DAG 2.0 的 Adaptive Shuffle 功能之后，线下验证作业运行时间由 30+ 小时降低到小于 30 分钟，而产生的文件数相比于关闭 reshuffle 的方式降低了一个数量级，在保障业务数据及时产出的前提下，能极大缓解 pangu master 的压力。动态分区场景在弹内生产和公共云生产都有广阔的应用场景，随着 Adaptive Shuffle 的上线，dynamic insert 将是第一个解决的比较彻底的数据倾斜场景。此外，

DAG 2.0 也持续探索其他数据倾斜（data skew）的处理，例如 join skew 等，相信随着在 2.0 上更多优化功能的开发，我们的执行引擎能做到更动态，更智能化，包括数据倾斜问题在内的一众线上痛点问题，将可以得到更好的解决。今天最好的表现，是明天最低的要求。我们相信 2020 年的双十一，在面对更大的数据处理量时，计算平台的双十一保障能够更加的自动化，通过分布式作业运行中的动态化调整，在更少人工干预的前提下完成。

3-5. 高优先级作业资源保障

挑战

FuxiMaster 是 Fuxi 的资源调度器，负责将计算资源分配给不同的计算任务。针对 MaxCompute 超大规模计算场景下，不同应用间多样的资源需求，过去几年资源调度团队对核心调度逻辑做了极致的性能优化，调度延时控制在了 10 微秒级别，集群资源的高效流转为 MaxCompute 历年双十一大促的稳定运行提供了强有力的保障。而重要高先级作业的按时完成是双十一大促成功的重要标志，也是资源保障中的重中之重。

然而 MaxCompute 集群通常非常的繁忙，用户总的请求量通常是集群规模的若干倍。在这么大的压力下，想要保证高优先级作业能够快速拿到资源，天然会想到使用直接抢占的方式来抢夺低优先级作业的资源，但是每一次暴力的抢占都会使得作业在执行过程中被杀掉，无论从作业角度还是集群有效利用率角度都是一种浪费。

所以我们希望提供一种新的抢占方式，能够在保障高优先级作业在有限时间内获得资源的同时，尽量避免直接抢占低优先级的作业，这种方式我们称之为交互式抢占，它的行为简述如下：高优先级作业发起交互式抢占时，会让低优先级作业在跑完当前 instance 后，把资源还给调度器（不要继续使用执行下一个 instance）；如果在规定时间内没有归还，调度器就强行回收这份资源并分配给高优先级作业。

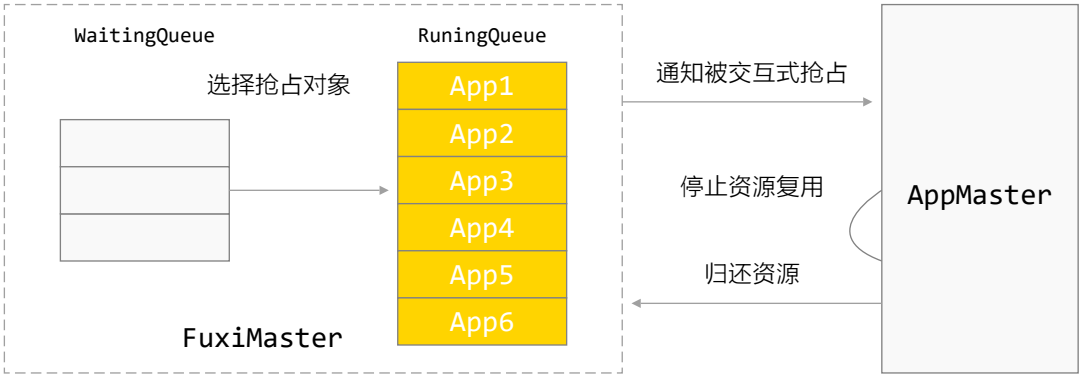
抢占过程本身涉及到抢谁的问题，这里需要综合考虑被抢作业的优先级、被抢作业的执行时间等因素。对于交互式抢占，我们希望能够把分配到所有机器上的全局最低优先级的资源抢过来，如果不够再抢占次低优先级的作业，以此类推；同样，在若干低优先级作业可供挑选时，我们还希望选择归还资源时间最短的作业，因为这样可以很大概率上避免强行回收。

此外，我们观察到有些 quota 组在一小段时间内的高优先级的请求已经超过了它的可用 quota 额度，同时其他 quota 组在运行时着大量的低优先级作业，那么从全局高优先级作业保障的角度来考虑，我们是否可以考虑通过有限的全局 quota 借调来保障高优先级作业的资源供给？这里面就涉及到管谁借、怎么借、借多少的复杂问题。

a. 交互式抢占

通常一个作业的 Task 可能会有多个 instance，为了避免频繁申请归还资源，作业会复用申请到的资源，也就是说一份资源可能会串行地被多个 instance 来使用，这会增加对资源的使用时间。如果是直接抢占，则会强行回收掉已经分配的资源，这会导致作业运行过程中被杀，显然是一种浪费。

而交互式抢占的设计思路是: 当高优先级作业持续一段时间分配不到资源，可以预定一些低优先级作业的资源，同时通过协议的方式告诉这些低优先级作业需要尽快还资源。当资源被还回来的时候，资源确保被优先分配给曾经有过预定的高优先级作业。如下图所示:



它具备两个关键属性:

- 资源预留：首先预定了这部分低优先级作业持有的资源，这些资源被归还之后一定会保障优先分配给预留过机器的高优先级作业，如果机器 Free 资源无法满足，则会继续保持预留，也不会分配给其他非预留的作业；

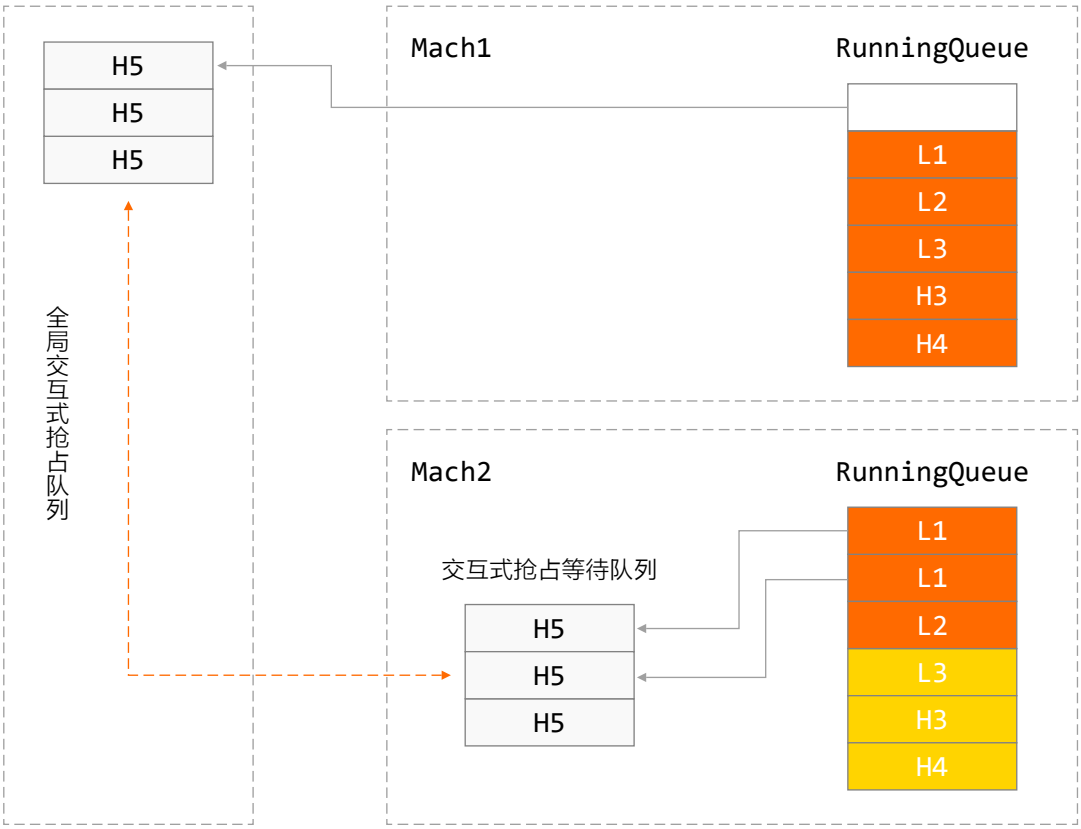
- 抢占属性：以协议的方式通知低优先级作业需要停止资源复用，等待当前的 instance 运行完，即刻归还资源，而还未运行的instance可以等待下次再次申请资

源来运行；相比于直接抢占来回收掉资源，导致运行中的作业被杀，能够避免资源的浪费。

此外，我们对交互式抢占做了进一步优化。

交互式抢占发起之后，这部分请求会以等待队列的方式记录在每台机器上，当机器有free资源的时候，优先分配给这个队列上的请求。但是集群中如果有其他机器有free资源，我们期望也是可以对这个队列进行供给的。因此，我们增加了预留外资源分配的能力，为每个 Quota 组维护一个全局视角的队列（或称为全局交互式抢占队列），发生交互式抢占后在机器上建立等待队列的同时，也同步更新全局交互式抢占队列。同时在调度逻辑上保证，当机器有Free资源的时候，总是优先分配给机器上的等待队列， 然后剩余资源再分配给全局交互式抢占队列，最后再分配给普通的等待队列，通过这种方式保证了资源总是优先供给高优先级的作业。这里面需要保障机器的等待队列和全局交互式抢占队列之间信息的同步。

下图为预留外资源分配的处理模型，左侧为 Quota 组级别的全局交互抢占队列，右侧 Mach1 有一份空闲资源，Mach2 有两份资源由交互式抢占的 L1 作业释放。



然而即使增加了资源供给渠道，在巨大的资源压力下还是可能出现资源不足的问题。所以我们在交互式抢占发起之后，对被抢占的对象增加了超时 Revoke 机制，即当超过规定的超时时间，机器上如果还是存在等待队列，就发起直接抢占，这是不得已而为之的办法。因此我们还是期望超时 Revoke 的代价和调度上的代价都尽可能的小，这里就涉及到了超时时间、超时 revoke 对象选择、revoke 资源量、revoke 资源如何供给的问题。

a. 对于超时时间选择，我们分析了线上离线 Job 的 Instance 的运行时间，85% 的 instance 运行时间都集中在 90s 以内, 因此我们选择了 90s 作业超时的配置，保证了大多数作业仍然是可以通过作业正常运行结束将资源还掉；

b. 而对于超时 Revoke 谁的问题，我们则选择超时时间最长的作业作为目标，给其他被交互式抢占的作业尽可能留足正常运行结束的机会。而如果机器上不存在等待队列的时候，即使到了超时时间，我们也会让其正常跑完；

c. 从调度效率上考虑，对于被 revoke 的资源无需遵循从某个低优先级作业到某个高优先级作业的定向供给，这样会增加大量信息维护的代价。所以我们设计上遵循了资源从低到高的流转过程，对于被 Revoke 释放后的资源仍旧回到资源池，依赖调度逻辑保障资源总会优先分配给高优先级的作业。

通过这种方式，高优先级作业的等待资源时间得到了明显的缩短，如下表所示。

等资源时间	现状	交互式抢占
< 30s	33.4%	45.8%
< 60s	45.0%	63%
< 90s	57.3%	72.4%
> 90s	42.7%	27.6%

b. 基于时间预估的全局最优抢占

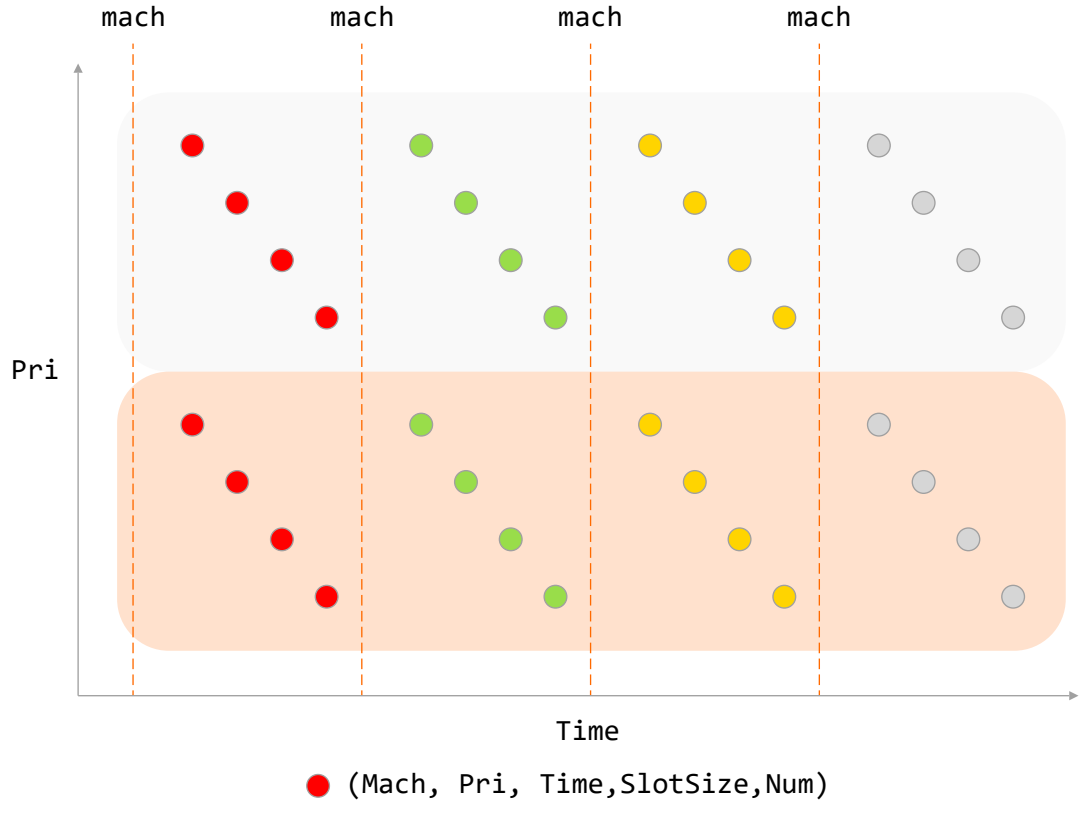
上述交互式抢占在选择抢占对象策略是非常简单的，随机从一台台机器遍历过去，贪心地抢占所有优先级比发起抢占作业低的作业，直到抢够为止。可以想象这个策略有很多不好的地方：

1、由于不知道哪台机器上分布着可抢的低优先级资源，所以会产生非常多的无用遍历，抢占的 latency 很高；

2、由于不知道后面机器上是否存在可抢的更低优先级的作业，所以当前只能贪心地把能抢占的作业全部抢占过来；

3、同 2 原因，由于不知道后面是否存在运行时间更短的低优先级作业，所以当前只能贪心地把能抢占的作业全部抢占过来。

可以看到，已有的抢占策略从抢占效果上是非常不理想的，虽然最终能够保障高优先级作业能够抢占到资源，但是对于被抢的作业是非常不友好的。因此我们做了进一步的改进方案。



如上图所示， 对任意一台机器而言，上面运行的作业都可以按照优先级划分为若干个组，我们称之为 P1、P2...Px，数字越大，优先级越低；Px 中有若干对应的正在运行的低优先级作业。随着每次资源的分配\归还，我们都更新Px中的运行作业的资源总和；当发起抢占时，我们面临的就是不同机器上的 Px，怎么能够快速、准确的挑选出 Px 就是我们需要解决的问题，这里我们利用了 Fuxi 内部高效的队列管理及挑选模型，此处不做赘述。

关于时间维度的利用，我们是如下考虑的：首先我们要有能力来获取作业的时间信息，我们统计了线上集群的 instance 平均运行时间，发现:

- 70% 的离线 Job instance 运行时间<=30s；
- 85% 的离线 Job instance 运行时间<=90s；
- 90% 的离线 Job instance 运行时间<=3min。

综合上述数据来看， HBO 提供的作业执行时间的信息对调度的可信度较高， 覆盖面也能接受。

综合考虑，在作业提交时，如果有 HBO 的时间信息，我们就相信这个时间；如果没有，由于 70% 的 instance < 30s, 所以我们先假设这些作业都是执行很快的，他们在被抢占上的优先级是高的；随着 job 的运行，调度器会收集运行时间并动态调整被抢占的优先级（时间维度），让跑的快的作业优先被抢占。

那么上述模型上线后，交互式抢占 latency 均值下降为原来的 1/10， 被抢占的7级\6级优先级作业(不合理抢占)次数从 32530次下降到 3617 次，下降为原来的 1/10, Instance 运行时间超过 180s 的作业被抢占的次数从均值 1 次/s 下降到 0.07 次/s。

c. 全局 Quota 保障

上述 a，b 方案我们都是基于抢占本身，然而引起抢占本质还是资源供给不足导致的，所以抢占在整个保障方案里依然是一个最终手段。 因此我们希望从 Quota 角度进一步寻找到增加资源供给的可能，来降低抢占的发生。

这里首先简要介绍下 Fuxi 对 Quota 的管理模型，区别于大多数静态 Quota 划分的策略，在 Fuxi 内部为了保持高效的资源利用率，采用动态 Quota 的计算模型，来实现集群总资源在各个 Quota 组之间的有效流转。 这里涉及到几个关键术语:

- MaxQuota：表示当前 Quota 组能划分的最大资源上限；

- MinQuota：当前 Quota 组最低资源保障，当 Quota 组请求超过 MinQuota 的时候，至少要保障这么多；

- RuntimeQuota：通过 Quota 模型计算出的最终各个 Quota 组能够划分的总资源。

我们分析了 Quota 使用情况并观察到了以下几个现象：

- 生产时段内某些 Quota 组的高优先级请求会超过整个 Quota 组的资源，这种情况下无论如何都是无法全部满足的；

- 同一时刻存在其他 Quota 组的高优先级作业请求并不多，但是最终 Quota 组的总资源还是比较多，也超过了其 MinQuota 的保障；

- 在不影响各个组 MinQuota 保障的前提下，多数情况下各个组可以腾挪的资源足以满足高优先级请求无法满足的部分。

综合上述分析结果，再结合高优先级作业在集群范围内都需要重点保障的考量，我们考虑进行跨组资源借调，如果高优先级作业请求超过了 RuntimeQuota，就适当从其他组借调一部分，实现全局的 Quota 保障。那么这里就涉及到找谁借调、怎么借调等重要的问题。

在对 RuntimeQuota 计算模型分析后发现，出现上述现象主要是由于我们在 Quota 划分的时候并未引入高\低优先级的考虑，在划分集群资源的时候没有针对高优先级请求有所侧重。因此我们引入了对高优先级请求的考虑，当某些 Quota 组高优先级请求超过 RuntimeQuota 的时候，适当提高该组的 MinQuota，然后借助 RuntimeQuota 的计算过程，一方面保障了划分 Quota 时对高优先级请求有所侧重，另一方面天然的保障了各个组的 MinQuota 的需求。

在上述方案实施后，资源时间得到了进一步缩短。

等资源时间	现状	交互式抢占	全局 Quota 保障
< 30s	33%	45%	98.3%
< 60s	45%	63%	99.0%
< 90s	57%	72%	99.8%
> 90s	43%	28%	0.2%

总结

交互式抢占显著提高了高优先级作业分配资源的效率，但是在集群资源有限的情况下，高优先级作业收益则必然有低优先级作业受损，而基于时间预估的全局最优抢占则从抢占的质量上出发，筛选出全局更优的抢占对象，以减少抢占带来的代价。高优先级作业全局 Quota 保障则是从资源供给的角度出发，尽可能提供充足的资源供给，来避免发生抢占的发生。高优先级作业资源保障始终是离线资源调度的重要关注内容，随着场景的丰富我们也面临更多的挑战，我们在资源管理上仍会持续关注重要高优先级作业，未来提供更加精细化的管理手段。

“伏羲”神算
阿里巴巴经济体核心调度系统揭秘

扫一扫二维码图案，关注我吧

MaxCompute 开发者社区钉钉群

SaaS 模式云数据仓库 MaxCompute

飞天大数据平台钉钉群

阿里云开发者“藏经阁”海量免费电子书下载

阿里云 | 飞天大数据平台 APSARA BIG DATA PLATFORM × 飞天AI平台 APSARA AI PLATFORM

藏经阁

阿里云开发者电子书系列

“伏羲”神算

阿里巴巴经济体核心调度系统揭秘