

让企业大数据平台 性能更优

Apache Spark
中文实战攻略（下册）

阿里、Databricks、领英、Intel、Facebook 都在用
Spark 企业级最佳实践中文解读全收纳





更多精彩内容扫码关注
Spark 中文社区微信公众号



钉钉扫码加入
Spark 中文社区钉钉群



阿里云开发者“藏经阁”
海量电子书免费下载

I 目录

Spark 最佳实践	4
使用 Databricks 作为分析平台	5
领英如何应对 Apache Spark 的 Scalability 挑战	11
利用闪存优化在 Cosco 基础上的 Spark Shuffle	26
基于 Spark 和 TensorFlow 的机器学习实践	33
在 kubernetes 上运行 apache spark: 最佳实践和陷阱	42
使用 RayOnSpark 在大数据平台上运行新兴的人工智能应用	51
使用 Ray 将可扩展的自动化机器学习 (AutoML) 用于时序预测	58
Apache Spark 3.0 对 Prometheus 监控的原生支持	68
 阿里云开源大数据平台实践	 80
助力云上开源生态 - 阿里云开源大数据平台的发展	81
EMR Spark-SQL 性能极致优化揭秘 概览篇	91
EMR Spark-SQL 性能极致优化揭秘 RuntimeFilter Plus	96
EMR Spark-SQL 性能极致优化揭秘 Native Codegen Framework	102
Spark Codegen 浅析	111
Tablestore 结合 Spark 的流批一体 SQL 实战	124
Tablestore+Delta Lake(快速开始)	132

Spark 最佳实践

使用 Databricks 作为分析平台

简介： SPARK+AI SUMMIT 2020 中文精华版线上峰会将会带领大家一起回顾 2020 年的 SPARK 又产生了怎样的最佳实践，技术上取得了哪些突破，以及周边的生态发展。本文是阿里巴巴高级技术专家章剑锋做的相关分享，介绍了 YipitData 公司基于 Databricks 平台搭建的分析平台。

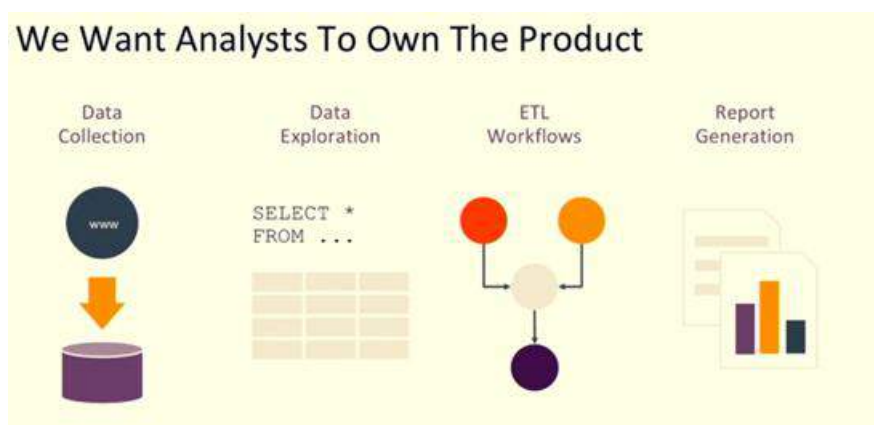
Spark 等引擎都是作为工具被开发者使用的，而我们使用这些工具的最终目的是搭建合适的平台提供给业务方。以下是 YipitData 's Platform 的相关介绍。

一、为什么要用到平台 (Why a platform) ?

YipitData 是一家咨询公司，其客户主要是投资基金以及财富五百强中的一些公司。该公司通过自己的数据产品进行分析，提供给客户相应的数据分析报告。YipitData 的主要产出方式和赚钱方式就是做数据分析，其公司内部有 53 个数据分析师，却只有 3 个数据工程师。数据分析的基础是数据，所以对于该公司来说大数据分析的平台是非常重要的。

二、平台中有什么 (What is in our platform) ?

YipitData 公司希望通过他们自己的数据分析平台能够让数据分析师不需要付出太大的成本就完成数据分析的任务，也就是 Own The Product，而这个过程主要包括如下图所示的 Data Collection、Data Exploration、ETL Workflows 和 Report Generation 四个阶段。



上面我们提到 YipitData 公司的人员主要包括数据分析师和数据工程师，其中数据分析师来分析数据并且提供基于数据的问题解答和分析报告，数据工程师来给数据分析师提供数据和分析数据的平台。Databricks 中的一个产品叫做 Workspace，简单来说它就是一个 Notebook，你可以在其中写 python、Scala、SQL 等语言的代码，然后交由 Databricks 平台去执行并返回结果。YipitData' Platform 是基于 Databricks 平台来搭建的，简而言之就是他们对 Databricks 进行了更深一层的封装，创建了一个 Python Library，更加方便分析师来进行使用。

（一）获取数据 (Ingesting data)

YipitData 公司的数据量是非常大的，有压缩后大小超过 1PB 的 Parquet，60K 的 Tables 和 1.7K 的 Databases。他们的数据收集使用的是 Readpipe，简单理解就是一个网络爬虫，在有了 URL 之后，将网页内存 download 下来然后进行存储，实现从 URLs 到 Parquet。首先，使用 Readpipe 对网页进行爬取，然后以流的方式源源不断的写入 kinesis Firehose，kinesis Firehose 会接着将数据写入 AWS 的 S3 上。在这个阶段所存储的数据都是原始 JSON 数据，是没有 schema 的，这类数据对于数据分析师来说是很难进行使用的。因此，第二步我们要对数据进行一些格式转换和清理，比较典型的做法是将 JSON 文件转换成 Bucket，这一步也自带了压缩效果。转换完成之后会有两个输出，如下图所示，一个是元数据，会写入 Glue Metastore，另外一个数据，会写入 Parquet Bucket 中。通过上面的过程，就完成了数据的收集和清理过程，整个过程是非常经典，非常有参考价值的。

Parquet Makes Data “Queryable”



另外，因为数据流是实时数据，每隔一段时间就会产生一些 JSON 文件，属于小文件，时间久了 S3 上面会存在非常多的小文件，带来性能方面的许多问题，于是要对小文件做相应的 Merge 处理，将小文件汇聚成大文件，这对后续的处理非常有帮助。

YipitData 公司所使用的的数据都是第三方数据，他们本身不生产任何数据，而使用第三方数据会面临一些问题，主要包括如下四类问题：

- Various File Formats
- Permissions Challenges
- Data Lineage
- Data Refreshes



上面几类问题是在实际业务中经常遇到的，如果不解决好自然也不能有很好的成果产出。YipitData 公司解决上面几类问题主要是靠 Databricks 平台，比如上传并利用额外的元数据将文件转为 parquet 等，如下图所示。

Databricks Helps Manage 3rd Party Data

- Upload files and convert to parquet with additional metadata
- Configure data access by assuming IAM roles within notebooks



```

Cmd 13
1. bash
2. mkdir -p ~/aws/

Cmd 14
1. bash
2. cat << EOF > ~/aws/config
3. [default] {
4.   region = us-east-1
5.
6.   [profile s3_access]
7.   role_arn = arn:aws:iam::000000000000:role/example_access
8.   credential_source = S3InstanceMetadata
9. }
3. EOF

Cmd 15
1. from pyspark.sql import functions as F
2.
3. df = spark.read.csv('...',) \
4.   .withColumn("input_file_name", F.input_file_name()) \
5.   .withColumn("data_ingested", F.current_timestamp())
6.
7. df.write.format("parquet").save('...')
  
```

(二) 表实用程序 (Table Utilities)

YipitData's Platform 提供了一些 table utilities 来帮助分析师创建 table 和管理 table。比如下图所示的 create_table 函数，可以帮助数据分析师更快速地创建 table。

Table: Database + Name + Data

```

Cmd 6
1 df = spark.read.parquet('...')
2
3 create_table('demo_database', 'demo_table', df)

Cmd 7
1 df = spark.read.parquet('...')
2
3 # Optionally specify column(s) to partition the data
4 create_table('demo_database', 'demo_partitioned_table', df, partitions=['dt'])
  
```



上图所示的是一个非常典型的 Spark Job 的场景，通常包括 read、processing 和 write 三个模块。但是对于 YipitData 公司来说，上面的过程仍然是一个比较繁琐的过程，因为该公司最重要的任务是进行数据分析，且大多数人员也是数据分析师，如果让数据分析师使用 Spark API 去完成上述过程，还是有一定门槛的。对于 YipitData 公司来说，最好是把一些功能进行封装，不要暴露太多的底层功能，所以有了上面的 create_table 函数，大大降低了数据分析师的使用难度。

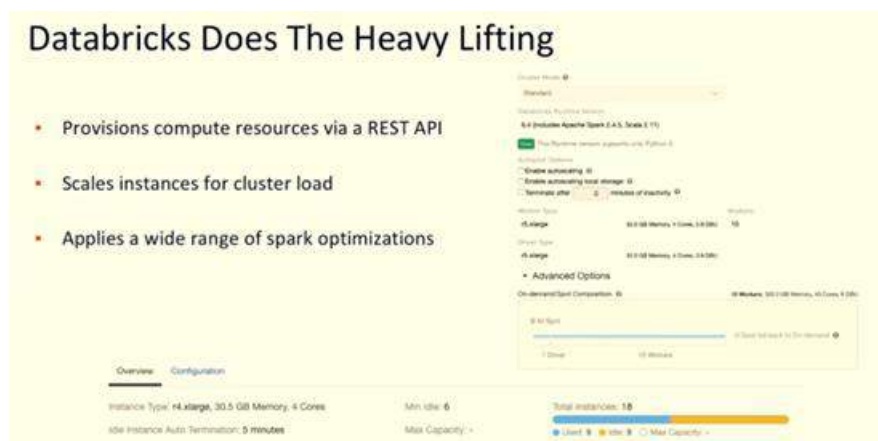
（三）集群管理 (Cluster Management)

对于数据分析师来说，最后还是要进行计算，就牵涉计算资源的管理，那么 YipitData 是怎么做的呢？我们知道，搭建一个 Spark 集群并不是很难，但是如何搭建一个能够最优化地解决问题的 Spark 集群并不是那么容易，因为 Spark 集群有非常多的配置，而这项工作如果交给数据分析师来做的话就更不简单了。为了解决易用性的问题，YipitData 的工程师参照 T-Shirt 的 Size 划分巧妙地将集群划分成 SMALL、MEDIUM、LARGE 三类，如下图所示，数据分析师在使用的时候虽然少了灵活性，但是节省了很多集群配置的时间，大大的提高了工作效率。背后的原理也是进行更深层次的封装，将众多参数设置隐藏起来，数据分析师只需要像选择 T-Shirt 的尺寸一样做选择即可，而无需关心背后的复杂配置如何实现。



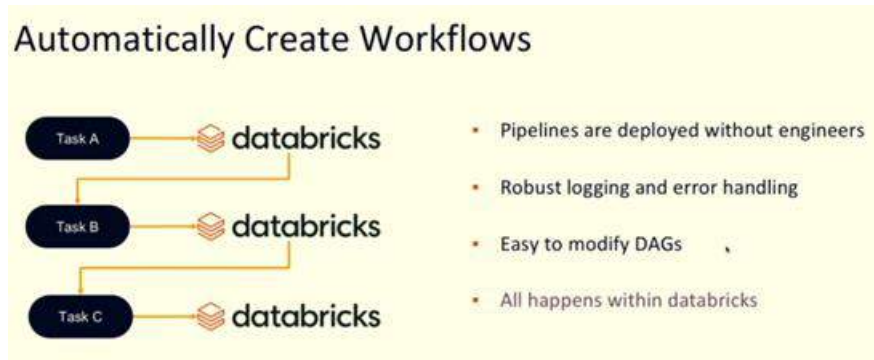


在集群管理方面，Databricks 还提供了许多其他的 API 来对集群的计算资源进行管理，比如可以通过 REST API 控制集群。对集群做各种各样的配置，还可以对集群的配置进行动态调整等等，如下图所示。



(四) ETL Workflow 的自动化 (ETL Workflow Automation)

YipitData 使用 Airflow 来实现 ETL Workflow 的自动化。越来越多的人使用 Airflow 来管理 ETL Workflow, 已经逐渐成为 ETL 的一个标准工具。对于数据工程师来说, Airflow 的使用不是很难: 首先构建一个 DAG, 然后去定义其中的 TASK, 最后定义下这些 TASKS 的依赖关系即可。但是, 终究是要写一段代码来实现这个过程, 就需要有人来维护, 对于大多数员工是数据分析师的 YipitData 来说就不是那么合适了。因此, YipitData 使用 Airflow+databricks 的 API 来自动化构建 DAGs。具体来说, 每个文件夹就代表一个 DAG, 每个 Notebook 就代表一个 Task, Notebook 中指定一些属性 (内部是 python 脚本), 然后通过 API 来自动化构建 DAG 文件。通过上面的过程完成整个 ETL 的自动化, 其中用户只需要指定 Notebook 中的参数值即可。YipitData 自动化创建 Workflows 的过程如下图所示, 整个流程都是在 Databricks 平台上扩展得到的。



三、Q&A

Q1: Databricks 和 Dataworks 都是一站式的数据分析平台，两者的区别是什么？

A1: 两者的侧重点不一样。Dataworks 绑定在阿里云，而 Databricks 可以在各个云上使用；Databricks 绑定了 Spark 引擎，而 Dataworks 可以使用各种引擎；Dataworks 在数据治理上更强一些，而 Databricks 的 Spark 应用更强一些。

Q2: 目前 Zeppelin、Jupyter、Databricks 产品的分析功能有些类似，他们有什么特别推荐的使用场景吗？

A2: 这几个产品最大的特点是提供了交互式的编程环境，和传统的 IDE 开发不同，他们有着更好的开发效率，尤其是在数据分析和机器学习方面。另外，这类产品也不是只能做交互式开发，也可以用来做 ETL。

领英如何应对 Apache Spark 的 Scalability 挑战

简介：在集群计算引擎使用率快速增长的过程当中，会面对多维度的计算基础架构规模扩展性的挑战。同时由于 Spark 团队直接与 Spark 用户打交道，如何提升 Spark 用户生产力，避免“用户支持陷阱”，一直是较为头疼的问题。本次直播将由领英 Spark 团队软件工程师沈旻和林致远为您介绍，领英 Spark 生态系统，构建多元化 Spark 生态系统过程中遇到的挑战，如何提升 Spark 用户生产力以及如何优化 Spark 基础计算架构。

演讲嘉宾简介：沈旻，领英 Spark 团队软件工程师，技术负责人，伊利诺伊芝加哥分校计算机专业博士学位。林致远，领英 Spark 团队软件工程师，卡耐基梅隆大学硕士学位，专攻分布式系统方向。

本次分享主要围绕以下四个方面：

- 一、Overview of Spark Ecosystem @ LinkedIn
- 二、Scaling challenges we have experienced
- 三、Solutions to scale our Spark users' productivity
- 四、Solutions to scale Spark compute infrastructure

一、Overview of Spark Ecosystem @ LinkedIn

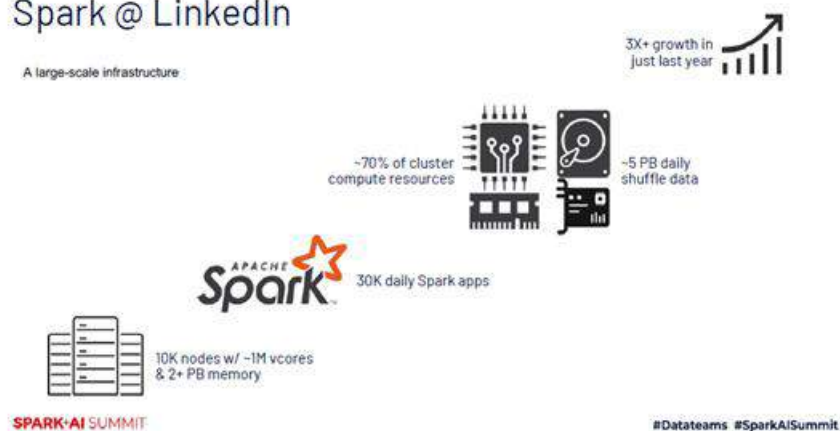
本着协助成就职场人士，在职场上事半功倍的初衷与使命，领英平台已演变成了数字化的全球经济图谱，图谱中所蕴含的信息不仅为各种平台产品提供了所需的数据，也提供了数字化洞悉全球经济的途径。在这样一张庞大的图谱中分析数据获得 insights 才能帮助领英平台中的职场人士。那么大规模的数据平台必不可少，因此 Apache Spark 成为了领英的主要计算引擎。



领英 Spark 生态系统有两个特别显著的特性，一方面是一个大规模计算基础架构，在 Hadoop 平台中领英使用了上万个节点，包括近 100 万的 vcores，以及 2+PB 的内存。每日大概有 3 万个 Spark 应用在集群中运行，这些 Spark 应用程序占用了 70% 的计算资源，每日 shuffle 超过 5PB 的数据。计算基础架构增长也非常迅速，仅仅 2019 年平均就增长了 3 倍以上。

Spark @ LinkedIn

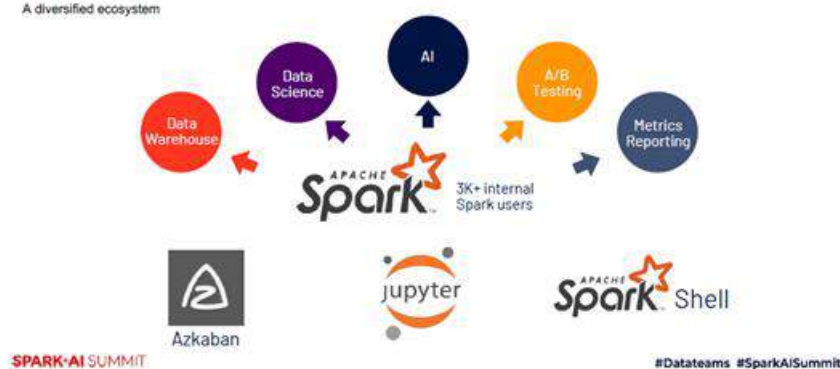
A large-scale infrastructure



领英的 Spark 是一个多元化的生态系统，Spark 用户通过不同的方式在集群中使用 Spark，要么通过 Azkaban 获得预定的流程，或者通过 Jupyter notebook 运行交互式查询。在领英，上千万用户通过 Spark 丰富的 API 开发各种大数据应用。大约 60% 的应用程序是 Spark SQL 应用，通过丰富的大数据应用，在领英涵盖了多种应用场景，包括人工智能、数据分析、A/B Testing、Data Warehouse、指标报告等等。

Spark @ LinkedIn

A diversified ecosystem



二、Scaling challenges we have experienced

面临挑战

快速增长的大规模基础架构所支持的多元化生态系统给领英带来了不小的难题。在扩展 Spark 基础架构，赋能用户，高效开发等方面遇到了不同维度的挑战。首先在资源管理方面，在上千用户的集群中管理用户，并满足不同团队计算资源的需求会带来不少集群运维的挑战。在此之上，计算引擎本身也面临这更高扩展性的问题，计算引擎快速增长意味着需要可扩展的基础架构保证稳定性，否则将大大影响用户使用的便捷性和满意度，同时会带来技术问题且会增加技术运维的开销。在用户生产力方面，当 Spark 使用量增加时，用户支持问题也会接踵而至，为了确保内部 Spark 用户的生产力，团队中更多的人力也会自动的转移到用户支持的问题上。但这同时意味着有限的团队资源无法投入到计算基础架构的改进上，而这本身又会带来更多用户支持问题。这样的恶性循环会形成阻碍团队发展的“用户支持陷阱”。

Challenges of Scaling Spark

The Compute Scaling Pyramid



- User Productivity
 - User productivity vs. "support trap"
- Compute Engine
 - Fast usage growth vs. infrastructure scalability
- Resource Management
 - Growing compute resource demands vs. bounded cluster resources

#DataTeams #SparkAISummit

解决方案

领英展开了系列的动作来解决这些挑战。在资源管理层面，工作中心是让运维团队从集群管理工作中解脱出来，创建了以业务为导向的集群资源队列结构，使得各个部门自己管理自己的资源队列。此外还优化了集成资源调度器，帮助用户弹性管理资源，将集群中的空闲资源以弹性资源的方式分配给最需要的队列。通过这些解决方案，资源管理很大程度上变成了自动化或用户自我管理的方式。在计算引擎层面，领英花了很多精力优化 Spark Shuffle，Spark Shuffle 是最大的规模扩展瓶颈。领英还尝试了多种 SQL 优化技术，SQL 优化有助于减少同等 Spark 应用数量下的计算工作量，从而进一步缓解在快速增长的规模下计算引擎所面临的压力。在用户生产力方面，领英的目标是拜托用户支持的陷阱，尽量通过自动化的 Spark 系统回答最常见的用户问题，分别是为什么我的 Spark 运行失败了，为什么我的 Spark 运行很慢，以及如何让它运行的更快。下面分别从用户生产力及 Spark Shuffle 方面展开详细的介绍。

Tackling Scaling Challenges

The Compute Scaling Pyramid



- User Productivity
 - Automated Spark job understandability
 - Failure root cause
 - Performance bottlenecks
 - Job tuning recommendations
- Compute Engine
 - Next-gen Spark shuffle service
 - Compute optimization
- Resource Management
 - Self-served queue management (YARN-5734)
 - Automated queue elasticity management (YARN-9869)

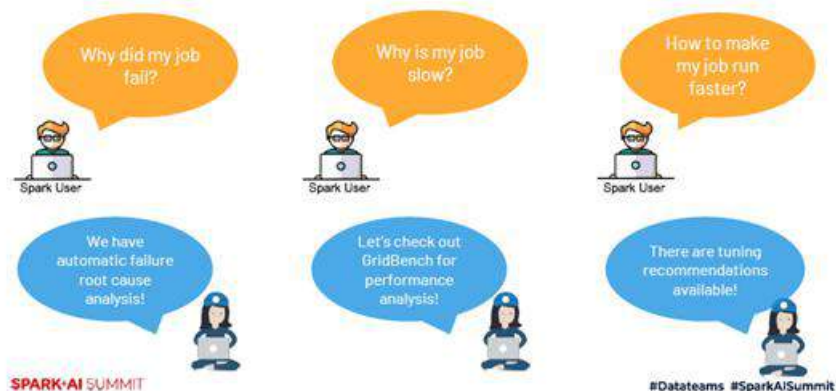
#Datateams #SparkAISummit

三、Solutions to scale our Spark users' productivity

提升 Spark 用户生产力

提升用户生产力，并帮助用户理解 Spark 应用的各个方面对于团队工作至关重要。鉴于 Spark 是非常复杂的计算引擎，对于 Spark 用户而言，调试和调参往往都非常繁杂。由于 Spark 团队资源相对有限，考虑到领英 Spark 规模非常庞大，如何更好的提供用户支持也是一个极大的挑战。在 Spark 中运行时，错误可能会出现在任何地方，用户至少需要很多步骤才能获取到相关日志，寻找出错原因，有时即使找到出错日志但想找到根本原因也不是很容易的事情。而且 Spark 用户花费了很多功夫终于调试好了，但运行时还是有很多不尽人意的地方，调整应用性能瓶颈，再提升性能也是一项很头疼的工作。针对用户的痛点，领英开发了一套自动化的解决方案，帮助用户查找用户出错的原因，分析并查找应用性能瓶颈，并提供各种调参建议。借助这些解决方案，可以大大提高用户生产力，同时减轻 Spark 团队在用户支持上的负担。

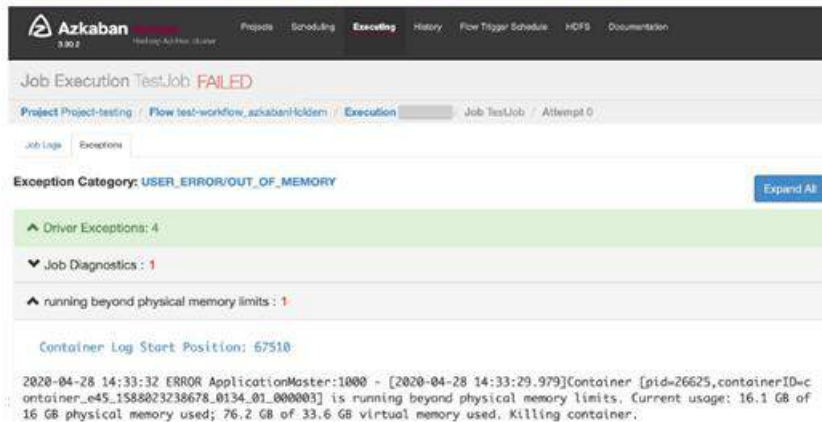
Typical Spark User Questions



第一类问题：为什么我的 Spark 应用失败了？

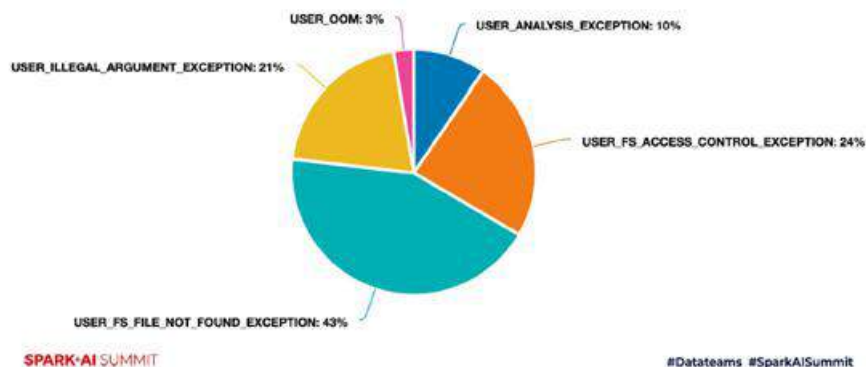
针对第一类问题，领英开发了一套自动出错分析工具，作为领英的 Spark 用户，大家可以在工作流失败的情况下，到工作流管理器 Azkaban 上找出错原因，免去了繁琐的获取相关日志，查询出错原因的步骤。用户只需要点击异常选项卡就能直观的找到所有相关异常，同时也能看到应用运行失败的根本原因。如下图中，出错原因是内存不足，用户也可以点击链接查看完整日志。

Automatic Failure Root Cause Analysis



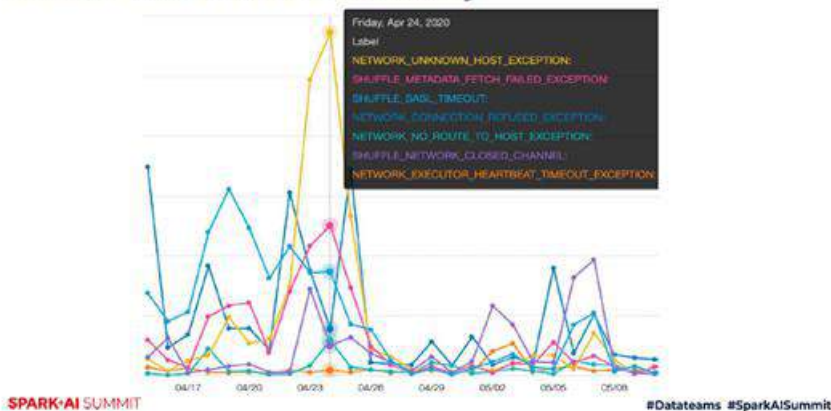
此外，领英还做了一系列出错原因分析可视化仪表，提供了各种原因的占比，可以为 Spark 用户带来更多帮助。如下图，可以发现最主要的出错类型是输入数据的缺失，以及数据访问权限问题。通过这些信息，团队可以更方便的知道其故障原因所在，从而采取相应措施。

Platform Failure Reason Breakdown



自动故障分析可以时运维团队收益。下图是集群上各种错误知识趋势，包括网络问题和数据 Shuffle 问题等等。这种趋势监控对集群健康至关重要，有助于提早发现异常行为，识别集群中的潜在问题。

Cluster Infra Failure Trending



第二类问题：如何找到运行时的性能瓶颈？

针对第二类问题，领英为此开发了性能分析工具 GridBench，它可以通过各种报告帮助用户理解性能指标，可以对同一个 Spark 应用多次运行后的结果自动分析，从而发现性能瓶颈点。GridBench 也可以作为很好的衡量工具，帮助用户了解存储和计算模块的性能指标。下图中展示了 GridBench 针对某个应用做出的性能比较报告，通过对比两组不同时间下运行间的之间执行记录。GridBench 可以确定应用性能是否有变化，可以发现 Executor CPU Time 有了明显的提升，直观的告诉用户性能瓶颈所在，在优化性能时更加的有针对性。

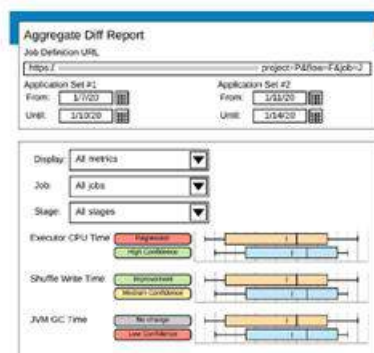
GridBench - Performance Analysis

Generate Reports

- Single application report
- Aggregate diff report
- Regression analysis report

Capability

- Performance Bottleneck Discovery
- Regression Detection
- Benchmark Infra (HDFS, Spark, etc)



第三类问题：如何调参，使得应用运行的更快？

针对第三类问题，领英为用户提供了自动化参数调优建议。通过一系列预定义调参方案，自动化检查应用配置，资源设置等等，给出对应的建议。如下图，某个调优方案显示的是红色，表明有进一步优化的空间，如果显示绿色表示参数设置已经比较合理。下图中应用的内存设置太高，建议将其设置为较低的值，避免资源浪费。

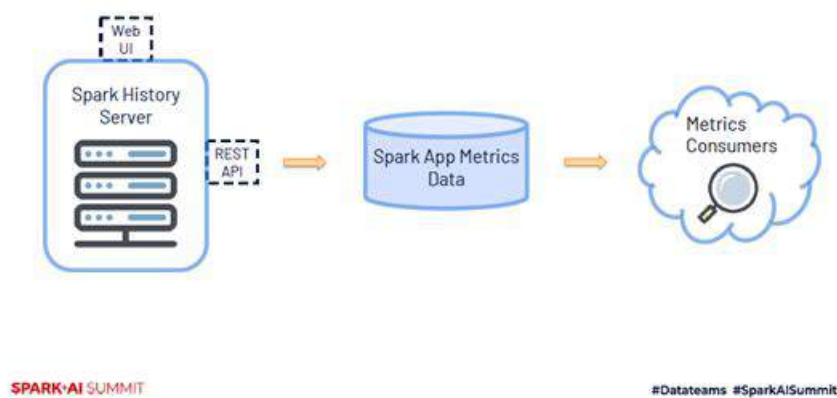
Tuning Heuistics & Recommendations



解决方案设计框架

上述解决方案背后的设计框架中, 各种 Spark 应用指标是框架核心元素, 追踪集群里运行的每个指标, 将指标汇总成一个数据集, 数据集会被前面提到的自动化错误分析, 性能分析及调参建议工具使用到。在设计框架中, 用 Spark History Server 获取所需要的数据, 它是领英 Spark 生态系统中重要的一环, 为所有 Spark 应用提供历史日志记录, 通过网页和 Rest API 的方式提供 Spark 应用的详细数据信息。再通过 Metric API 获取指标数据。

Path to Automation



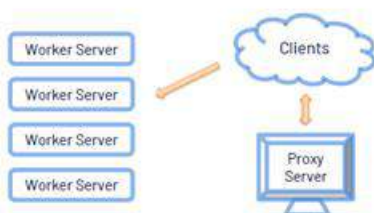
Spark History Server

当日均 Spark 应用数据达到上万个时, 通过 Spark History Server 获取每个指标数据, 还是遇到了功能扩展问题。一方面在集群高峰时段, 单位时间内结束的 Spark 应用数量比较大, Spark History Server 不能很好的处理大量的并发请求。其次, Spark History Server 在解析历史日志, 提取较大的日志文件时会非常耗时, 影响应用指标的及时性。为了解决并发请求问题, 设计了分布式的 Spark History Server, 架构中包含一个 proxy server 和多个 worker server, 通过使用多台服务器可以很好的横向扩展。为了解决第二个问题, 设计了增量 Spark History Server 解析 (Incremental Parsing)。一般情况下, 在 Spark

应用运行结束后才解析历史日志，Incremental Parsing 可以在运行时开始解析，一点点的增量解析日志文件。当 Spark 应用结束后，Spark History Server Incremental Parsing 可以在很短的时间内提供所需要的指标数据。目前大约只需要 20 秒，Spark History Server Incremental Parsing 就可以完成对 99%Spark 应用的日志解析。

Scaling Spark History Server

Distributed Spark History Server



SPARK·AI SUMMIT

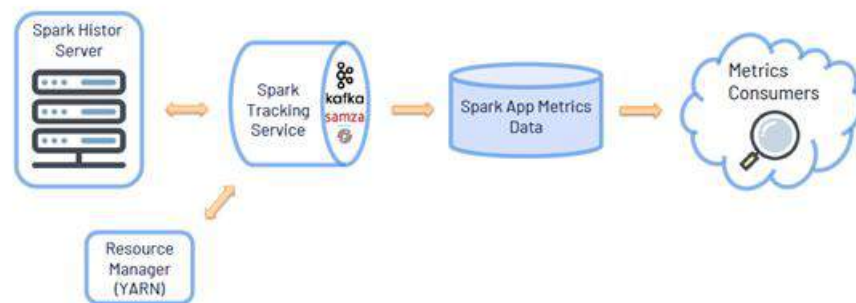
Incremental Parsing

	Average Delay	99th Percentile Delay
Clients	193s (3+ mins)	3035s (~1h)
After	6.82s	18 s

#Datateams #SparkAISummit

通过对 Spark History Server 扩展性和实时性的提升,得以以较低延迟提供所有 Spark 应用指标的数据。为了驱动各种用户生产力数据,领英搭建了一套基于 Kafka 和 samza 的 Spark Tracking Service。基于 Kafka 和 samza 是领英开源的流处理系统, Spark Tracking Service 首先会读取来自集群 Resource Manager 的数据流,获取运行结束时的 Spark 应用 ID, 查询 Spark History Server 以获取每个 Spark 应用的数据,在对原始数据处理后 Spark Tracking Service 会进一步解析用户所需要的指标数据。由此,前面提到的自动故障分析、性能分析及调参工具就有了数据来源。

A Low-Latency Solution



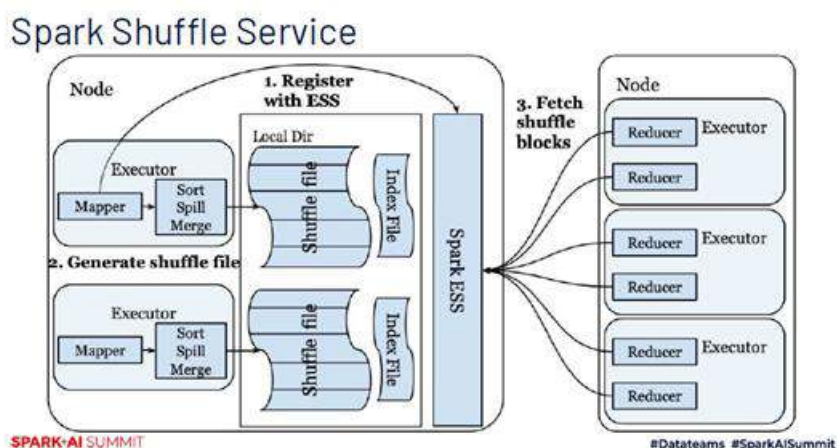
SPARK·AI SUMMIT

#Datateams #SparkAISummit

四、Solutions to scale Spark compute infrastructure

Spark Shuffle Service

有了提升用户生产力的各种工具之后，Spark 团队可以更多的投入的优化计算引擎之上。Spark 本身是一个复杂的系统，应该首先改进哪个组件呢？随着 Spark 在领英内部使用率的快速增长，Spark Shuffle Service 成为了最先扩展瓶颈的的 Spark 组件之一。领英使用了 External Spark Shuffle Service 管理 Shuffle 文件，启用 Spark 动态资源分配功能，这种配置对多租户集群中 Spark 应用间的公平资源共享至关重要。在这样的部署中，集群中的每个计算节点都将部署一个 Spark Shuffle Service，每个 Spark Executor 在启动时会和本地的 Spark Shuffle Service 对接，并提供注册信息。之后 Spark Executor 中 Shuffle Map Tasks 会生成 Shuffle 文件，每个文件都包含对应不同 Shuffle 分区的 Shuffle Block，Shuffle 文件被 External Spark Shuffle Service 统一管理。当 Shuffle Reducer Tasks 开始运行时，都会从远程的 Shuffle Service 当中获取相应的 Shuffle Block。在繁忙的生成集群当中，单个 Shuffle Service 可以轻易的接收到数千个 Shuffle 并发连接，这些连接来自数十个应用中的 Shuffle Reducer Tasks。由于 Spark Shuffle Service 共享性质，在大规模部署应用服务时遇到了很多问题。



Spark Shuffle Service 问题

首先是 Shuffle 可靠性问题，在生成集群当中，在集群高峰时段 Reducer Tasks 经常无法与 Shuffle 进行连接，连接失败将导致 Shuffle Block 的获取失败。这种问题导致工作流中的 SLA 无法满足，甚至运行失败。除此之外，还遇到了 Shuffle 效率问题，在集群当中，Shuffle 文件存储在硬盘之上，由于 Reducer Tasks 请求陆续发出，Shuffle Service 也将访问数据，如果 Shuffle Block 大小很小，那么 Shuffle Service 生成的少数据随机获取操作将严重硬盘的数据吞吐量，从而延长 Shuffle 等待时间。第三个问题是 Shuffle 规模扩展性问题，由于 Shuffle Service 的共享属性，一个需要 Shuffle 很多小 Blocks 的应用，在获取 Shuffle Block 时很容易对 Shuffle Service 造成过大压力，导致性能的下降。这不仅影响对 Shuffle 不友好的应用，还会影响共享同一个 Shuffle Service 的相邻应用。对于这些应用而言，调整 Shuffle Block 并不容易，这种现象发生时也会导致其它正常应用运行时间的延长。

Issues with Spark Shuffle Service

- Reliability issue
 - Unreliable connection establishment to shuffle services under heavy load
- Efficiency issue
 - Small shuffle blocks hurt disk throughput, prolonging shuffle wait time
- Scalability issue
 - Abusive jobs choking shared shuffle services, causing performance degradations to neighboring jobs

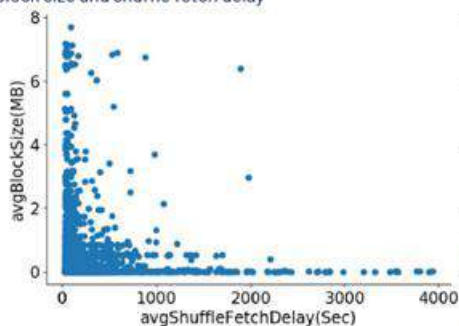
SPARK-AI SUMMIT

#Datateams #SparkAISummit

下图很直观的展示了小 Shuffle Block 带来的问题，图中采用了 5000 个 Shuffle Reduce Stage，并在图中展示了每个 Stage 平均 Shuffle Block 的大小，以及每个任务的 Shuffle 等待时间。数据来源是领英 2019 年生产集群当中所运行的 Spark 应用。可以发现，经历了较长 Shuffle 等待时间的 Stage，大多数也是应用了小 Shuffle Block Stage。

Issues with Spark Shuffle Service

Correlation between block size and shuffle fetch delay



SPARK-AI SUMMIT

#Datateams #SparkAISummit

Stage1: 提升 Shuffle Service 可靠性

为了解决这些问题，领英分了三个阶段来系统性的解决集群中的 Spark Shuffle 组件。第一阶段就是提升 Shuffle Service 在集群高峰时段的可靠性。通过前面介绍的集群故障分析工具，发现在集群中 Shuffle 失败的主要原因与网络故障并没有特别大的关系。而是由于 Shuffle Service 在处理 Shuffle 验证请求时超时。这个问题日均达到了 1000 次，某个日子甚至达到大约 6000 次。

Next-gen Spark shuffle service

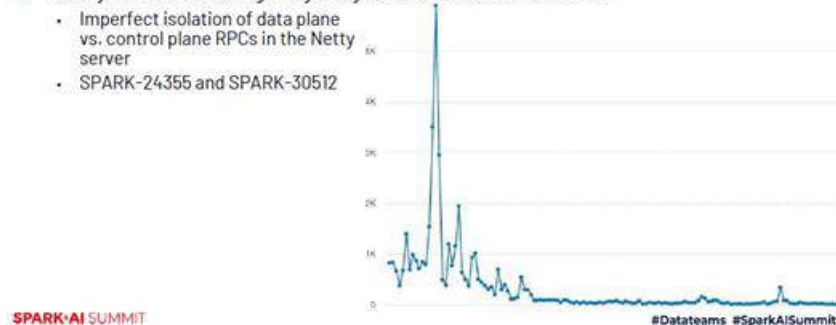
Stage 1: Hardening shuffle service Netty server



经过更进一步的研究，发现这是因为 Shuffle Service 背后使用的 Next-gen 服务器的问题。在较轻量层的控制层 RPC，如身份验证请求并没有很好的与更加耗时的数据处理 RPC 隔离开来，在集群高峰时段，大量的数据层的 RPC 占用了 Shuffle Service 的处理时间，直接导致控制层 RPC 请求超时。领英修复了这个问题，在集群上部署了改进后的 Shuffle Service 之后，看到了立杆见影的效果。大大减少了此类问题的出现，提升了集群中 Shuffle 组件的可靠性。

Next-gen Spark shuffle service

Netty issue causing majority of the shuffle failures



Stage2: Shuffle Service 端限流机制

在下一阶段，着重于对 Shuffle Service 端限流来帮助解决集群内 abusive 应用的影响。这类应用所带来的最大的负面影响是 Shuffle Block 索取的速度。这些 abusive 应用会生成大量的小 Shuffle Block，因此当 Reducer Tasks 获取 Shuffle Block 时，Shuffle Service 会开始大量的小数据随机读取操作，很容易导致硬盘带宽过载。在领英生产环境中，abusive 应用会延长 Shuffle 获取等待时间。领英开发了 Shuffle Service 限流机制，实时追踪每个连接的应用 Shuffle Block 获取速率，当 Shuffle Block 获取速率超过阈值时，Shuffle Service 可以让相应应用的 Reducer Tasks 回退，通过减少并发 Shuffle Block 获取数据流的方式作出响应。

Next-gen Spark shuffle service

- Stage 2: Shuffle service with throttling
 - Shuffle abusive jobs significantly increases shuffle fetch delay of their neighboring jobs
 - Measures per application shuffle block fetch rate
 - Throttle reducers from the most abusive job to reserve bandwidth for other applications
 - Throttled reducers would reduce block fetch rate or # concurrent block fetch streams

SPARK·AI SUMMIT

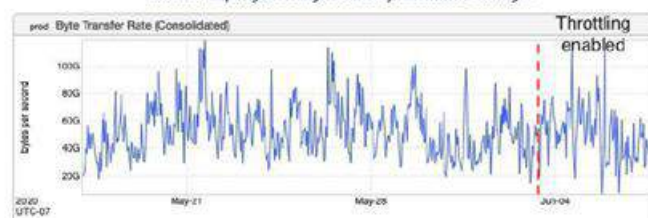
#Datateams #SparkAISummit

领英最近在生产环境中使用了 Shuffle Service 限流机制，观察到集群中所有节点上的 Shuffle Service Block 获取速率的波动幅度明显降低，这意味着 abusive job 对 Shuffle Service 以及相邻应用的影响开始受到了控制。同时，还观察到集群上 Shuffle 数据传输速率并没有特别明显的变化。这意味着当限制那些少数 abusive 应用时并不会伤害整个集群 Shuffle 数据吞吐量。

Next-gen Spark shuffle service

- Reduce block fetch spikiness

Aggregated shuffle bytes per second across all nodes in the same cluster. Spiking this significantly reduced change.



SPARK·AI SUMMIT

#Datateams #SparkAISummit

Stage3: Magnet

尽管限流可以保护 Shuffle Service 免受 abusive 应用的影响，但依然不能根本的解决小 Shuffle Block 的问题。受到限流的作业可能是高优先级的作业，并且有严格的 SLA 要求，这些应用无法承受 Shuffle Service 受导致的运行时间的影响。另一方面，挑战这些应用的参数，以增加 Shuffle Block 大小也不容易，这会导致任务处理数据量的增加，同时对该应用的其它方面产生影响。为了解决这个问题，领英对 Shuffle Service 采取了根本的改进，设计并实现了 Magnet，一种在 Spark 之上的全新的 push base Shuffle 实现方式。相关工作论文也被 VLDB2020 接收，目前领英也准备将这方面工作贡献至开源社区。更详细的 Magnet 可以关注后续工程博客文章。

Next-gen Spark shuffle service

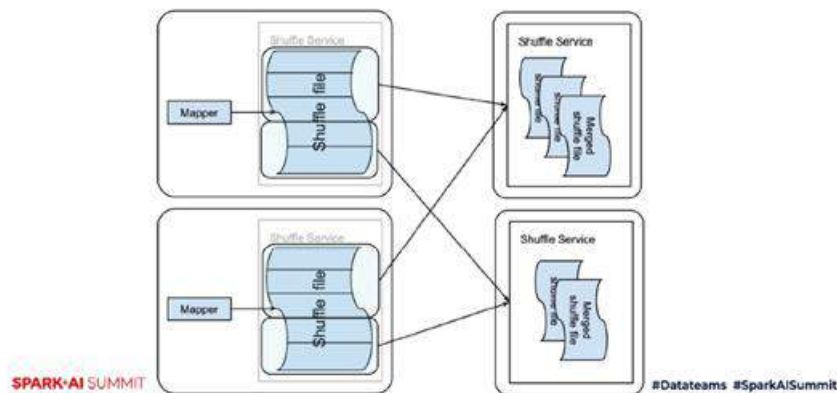
- Stage 3: Magnet Shuffle Service
 - Throttling does not solve the small block issue
 - Certain high-priority tight-SLA jobs cannot afford being throttled
 - Not always straightforward to tune the jobs to shuffle larger blocks
 - M. Shen, Y. Zhou, C. Singh. "Magnet: Push-based Shuffle Service for Large-scale Data Processing" *Proceedings of the VLDB Endowment*, 13(12)(2020)
 - SPIP in SPARK-30602

SPARK-AI SUMMIT

#Datateams #SparkAISummit

Magnet 采用了推送和合并的 Shuffle 机制, 生成 Shuffle 文件之后, Map Tasks 会将生成的 Shuffle Block 划分为多个组, 每个组包含了连续的 Shuffle byte 组成的数据, 分组之后另外单独的线程会读取一整个组, 将其中的 Shuffle Block 传输到远程的 Shuffle Service 中。Shuffle Service 中 Shuffle Block 按照不同的 Shuffle 分区被合并。Shuffle Driver 会在 Shuffle Map Stage 一开始会选择一系列的 Shuffle Service, 每个 Map Tasks 都将收到同样的一系列 Services, 这样可以确保属于同一个分区的 Shuffle Block 始终被分配到同一个远程的 Shuffle Service 上。在 Shuffle Service 端将以 Best effort 方式把收到 Shuffle Block 按分区合并至对应的 Shuffle 文件当中。

Push-Merge Shuffle

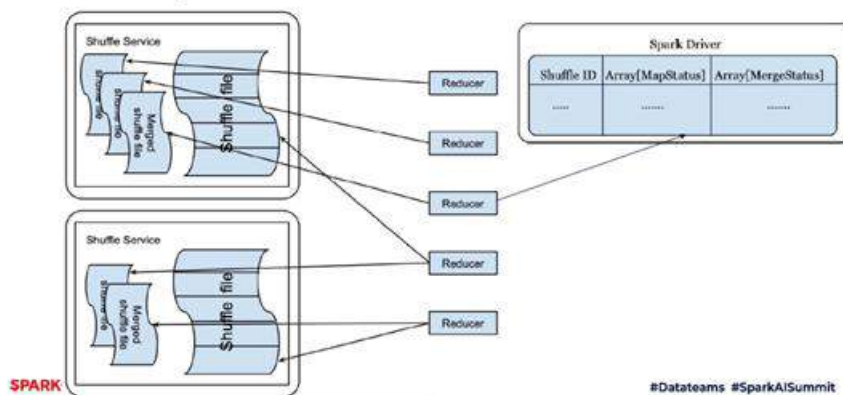


SPARK-AI SUMMIT

#Datateams #SparkAISummit

当推送和合并的过程完成时，除了原本并没有被合并的 Shuffle Block 大小和位置之外，Spark Driver 也会收到这些分区合并的 Shuffle 文件大小和位置。当 Reducer Tasks 开始运行时，通过 Spark Driver 查询所需 Shuffle Block 的位置和大小，这将大大减少 Reducer Tasks 所需获取的 Shuffle Block 数量，从而避免小 Shuffle Block 的问题。尽管 Shuffle Service 端以 Best effort 方式把收到 Shuffle Block 按分区合并，会导致小部分 Shuffle Block 没有被合并，Reducer Tasks 仍然可以获取那些没有被合并的 Shuffle Block，从而保证数据的完整性。此外，由于目前 Reducer Tasks 大部分输入数据都被合并在集群的一个节点之上，Spark Driver 在调用 Reducer Tasks 时考虑到这点，有助于进一步提高 Shuffle 性能。

Fetch Merged Shuffle Data



基于 Magnet 的 Shuffle 过程中，Map Tasks 生成的 Shuffle Block 会推送的远程 Shuffle Service 当中，并按照不同的 Shuffle 分区合并，这个过程有助于将 Shuffle 当中的小数据随机读取操作转化为大数据的顺序读取操作。此外，Shuffle Block 合并的过程，相当于为 Shuffle 中间数据创造两个副本，有助于进一步提高 Shuffle 的可靠性。同时 Reducer Tasks 调取合并后的 Shuffle 分区的位置，有助于进一步提高 Shuffle 的性能。

Magnet Shuffle Service Recap

- Magnet Shuffle Service
 - Mapper generated blocks get pushed to remote shuffle services to be merged per-partition
 - Convert the shuffle small random reads into large sequential reads
 - Shuffle data gets 2-replicated, further improving reliability
 - Locality-aware scheduling of reduce tasks to co-locate with the merged shuffle partition
 - Significant performance improvement observed, also help to prevent being throttled
 - Currently rolling out to production flows

下图所示，可以观察到基于 Magnet 的生产性能有了显著的提升。使用 Gridbench 性能分析工具，对同一个 Spark 应用，在使用 Magnet 后的性能进行了分析。下图中使用了较为复杂的生成机器学习特征数据的生产流程，处理了集群中的真实数据。原本在应用执行过程中，占据较大比重的是 Shuffle 获取等待时间被极大的缩短，带来了接近 30% 的应用运行时间的缩短。目前，领英正在将这种全新的 Shuffle 机制推广到生产集群中。

Magnet Shuffle Service in Action

Metric	2003	2004	2005	Metric Description
Block Split Size	960	960	26.18.421	Total data spilled to disk (in GB)
Executor CPU Time	21040	21039	26.12053	Total executor CPU time spent by the executor running the main task thread (in minutes)
Executor Runtime	17609	16612	-58% (-5904%)	Total elapsed time spent by the executor running tasks (in minutes)
Executor Runtime with Shuffle	16706	16076	-4% (-32%)	Executor run time including shuffle time (in minutes)
First Launch SLS Completed	165	174	6% (5.42)	Duration between launching the first task and stage completion (in minutes)
Input Records	181327175	181326187	0% (-0.5%)	Total number of records consumed by tasks (in thousands)
Input Size	17089	17084	-0% (-0.3%)	Total input data consumed by tasks (in GB)
IO GC Time	489	445	-9% (-8.8%)	Total time JVM spent in garbage collection (in minutes)
Memory Split Size	5403	5242	-3% (-3%)	Total data spilled to memory (in GB)
Net IO Time	2768	2740	-10% (-5%)	Total time spent accessing external storage (in minutes)
Output Records	4337	4337	0%	Total number of records produced by tasks (in thousands)
Output Size	19.22	19.22	0% (0)	Total output data produced by tasks (in GB)
Shuffle Read Records	13403148	11832128	11% (-12.3%)	Total number of shuffle records consumed by tasks (in thousands)
Shuffle Read Size	1793	1758	-2% (-2.3%)	Total shuffle data consumed by tasks (in GB)
Shuffle Read Wait Time	14016	9.40	-99% (-93.5%)	Total time during which tasks were blocked waiting for remote shuffle data (in minutes)
Shuffle Write Records	13403150	11832128	11% (-12.3%)	Total number of shuffle records produced by tasks (in thousands)
Shuffle Write Size	1751	1758	0% (0.3%)	Total shuffle data produced by tasks (in GB)
Shuffle Write Time	19.36	15.88	-18% (-17.5%)	Total shuffle write time spent by tasks (in minutes)
Submission to Launch Delay	440	411	-7% (-6.6%)	Delay between stage submission and launching of the first task (in minutes)
Total Runtime	414	165	-59% (-59%)	Total elapsed running time (in minutes)

SPARK AI SUMMIT

#Datateams #SparkAISummit

利用闪存优化在 Cosco 基础上的 Spark Shuffle

简介： SPARK+AI SUMMIT 2020 中文精华版线上峰会将会带领大家一起回顾 2020 年的 SPARK 又产生了怎样的最佳实践，技术上取得了哪些突破，以及周边的生态发展。本文中，来自 Databricks 开源项目组的软件工程师吴一介绍了利用 Flash 闪存优化在 Cosco 基础上的 Spark Shuffle。原标题：Flash for Spark Shuffle with Cosco

Cosco 是 Facebook 开发的一种服务，主要用于优化 Spark Shuffle 的性能，下文主要介绍用 Flash 闪存（以下简称：闪存）进一步优化 Cosco。

一、Cosco

Cosco 作为一种服务主要优化 Spark Shuffle 的性能，其优势有：

- 相较于原生的 Spark Shuffle，能够提升大约 3 倍的 I/O 性能，能够有效降低磁盘的读写时间；
- 引入闪存以后 Cosco 能够以更少的资源支撑更多的场景；
- 引入闪存之后有更大的可能降低 Query 的延迟；
- 利用闪存优化 Cosco 的过程中用到的技术也可以用于 Cosco 之外的领域。

（一）Cosco 产生背景

在 Spark Shuffle 中有 Map Task 和 Reduce Task 两种 Task，每个 Map Task 都会生成 Map Output Files，然后根据 Partition 进行分组，决定将文件写入本地磁盘还是分布式文件系统中；所有 Map Task 执行完毕之后，Reduce Task 就会去读取 Map Output Files 中某个分区的数据，将其合并成某个大的 Partition，在必要的时候还会进行排序。在上面的过程中，主要会存在两个与 I/O 性能相关的问题：

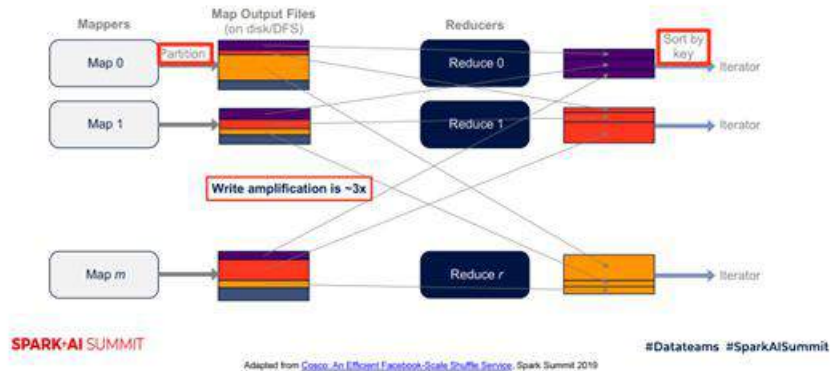
（1）Write amplification problem

如下图，这个问题也就说在最坏的情况下，同一个 shuffle 产生的字节会被写入磁盘三次：

- 第一次是在 Map Task 产生 shuffle data 的过程中如果内存不足，会先把 Shuffle Data Spill 到磁盘；
- 第二次是写入 Map Output Files 的过程；
- 第三次是在 Reduce Task 中，如果进行 sort 的时候内存不够，也会先 Spill 到磁盘。

Spark Shuffle Recap

Write amplification problem

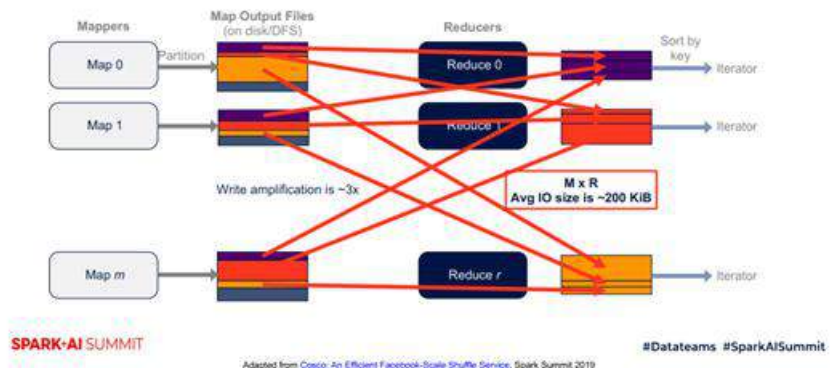


(2) small IOs problem

在 Spark Shuffle 模型中，其 I/O 请求总数会有 $M \times R$ 个，而在生产中观察到的 I/O 请求的 size 平均为 200kb，相对于磁盘来说是非常小的，当整个作业的并行度提升之后会产生大量的小 I/O 请求，会急剧增加磁盘开销，比如寻址时间等，从而导致 Shuffle 的性能变差。

Spark Shuffle Recap

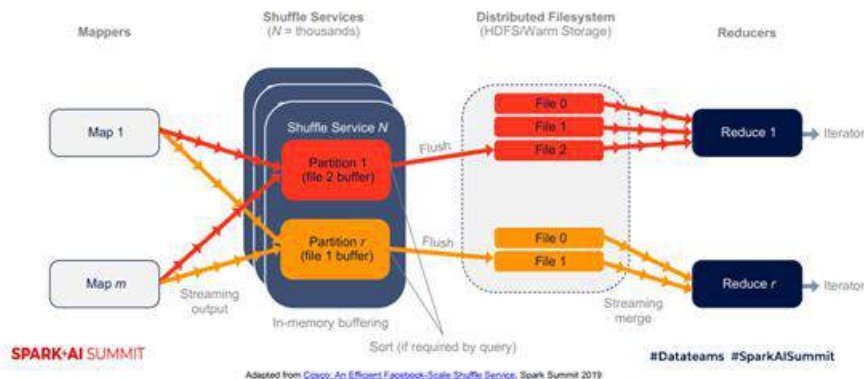
And small IOs problem



基于上述问题，Cosco 应运而生，其工作原理如下图所示。相较于原生的 Spark 每个 Task 生成一个自己的 Map Output Files，Cosco 允许不同的 Map Task 将同一个 Partition 写入到同一个内存缓存中，缓存到达一个阈值后，会将这部分数据 Flush 到分布式文件系统的文件中。这种情况下，同一个 Partition 可能产生多个对应的 Flush 文件，等到 Reduce Task 执行的时候，只需要读取 HDFS 系统中的文件即可，且文件的数据量在十几 M 的级别，且文件数量远小于之前的 $M \times R$ 数量级。因此，也就解决了小 I/O 的问题。

Cosco Shuffle for Spark

Reducers do a streaming merge after map stage completes

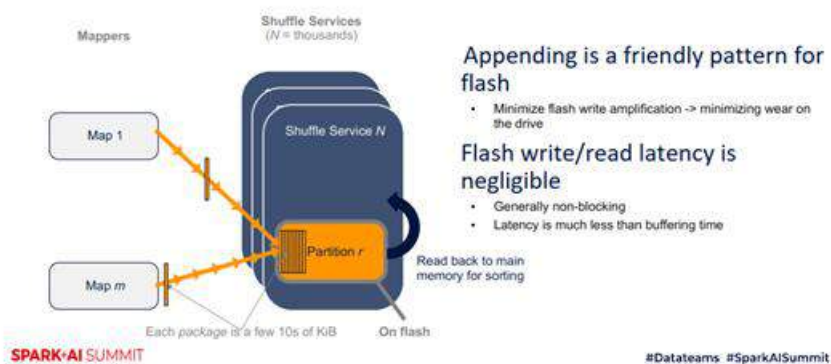


(二) 用 Flash 替换内存缓冲

使用 Flash 来作为缓冲的话是通过追加写的方式将 Shuffle 的数据写入缓存中,之所以这么做是因为闪存的可擦写次数是有限的,追加写可以延长闪存的寿命。闪存相对于内存存在着一定的延迟,但是总体而言这个延迟相对于整个 shuffle 在 Cosco 中缓存的时间是可以忽略不计的。

Replace DRAM with Flash for Buffering

Simply buffer to flash instead of memory



现在存在一个如何选择的问题,比如在 1GB 的内存和每天能够写 100GB 的闪存之间让我们用来部署集群,我们如何抉择呢?这里有一条不精确的经验:1GB 的内存和每天能够写 100GB 的闪存这两种选择的效果是一样的,也就是说能够支撑同样的场景,但是内存比闪存需要更多的能耗。大家在部署集群的时候可以根据这条经验来实际操作,选择最优的配置。

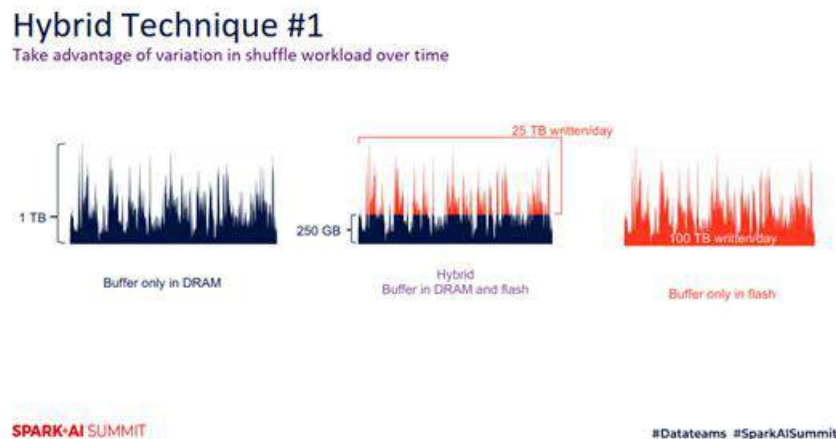
（三）基于内存和闪存混合的缓存优化

基于内存和闪存混合的缓存优化技术主要有两种：

- 第一种是优先缓存内存，当内存达到一定阈值之后再 Flush 到闪存；
- 第二种是利用 partition 加载速度不一样的特性，对于加载速度快的 partition 用内存缓存，对于加载速度慢的 partition 用闪存缓存。

（1）第一种

第一种优化技术利用了 Shuffle 数据随时间变化的特性，如下图所示，我们发现 Shuffle 数据随时间变化的统计中，峰值情况是占据小部分的，于是我们用闪存来处理峰值情况，最终只用 250GB 的内存和每天能写 25TB 的闪存就能达到和原来一样的效果，这样就实现了 Cosco 的优势，用更少的硬件资源来支撑了同样的场景。这种混合存储的优化技术比之纯用内存有着更强的伸缩性，不会在某些特殊情况下造成系统崩溃。比如单纯用内存的时候，如果出现内存不足就会崩溃，但是混合使用的时候就可以用闪存来处理异常情况，避免造成严重后果。



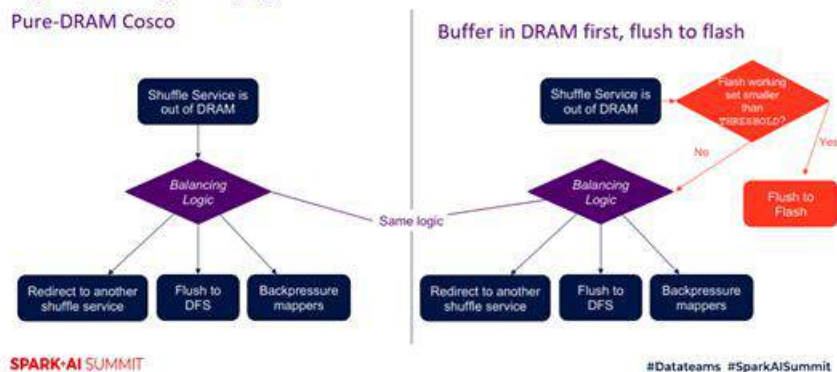
原来的 Cosco 集群有负载均衡的逻辑，更了获得更好的效果，我们使用插件的方式将闪存的优化集成到负载均衡逻辑中，如下图所示，引入一个阈值，当 Shuffle Service 内存不够的时候，就会利用阈值来进行判断是否将数据 Flush 到闪存中，这样有两个好处：

- 实现简单；
- 便于对集群的性能做评估。

Hybrid Technique #1

Plug into pre-existing balancing logic

Pure-DRAM Cosco



总的来说，第一种优化技术有如下特点：

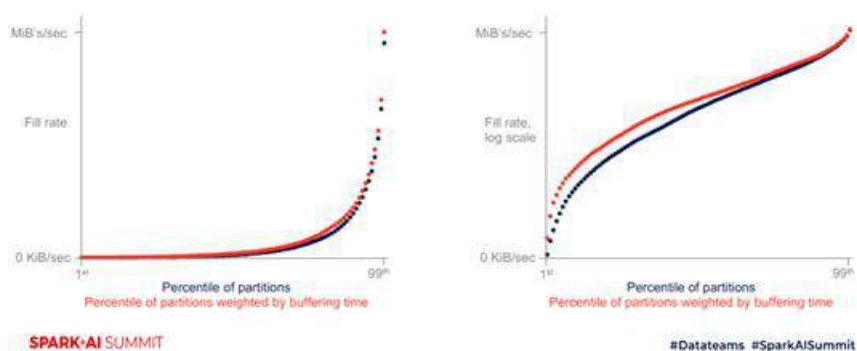
- 利用了 Shuffle Data 的分布随时间变化而变化的特性；
- 采用了优先在内存内进行缓存的策略；
- 巧妙的适配了原来的负载均衡逻辑。

(2) 第二种

第二种是利用 partition 加载速度不一样的特性，对于加载速度快的 partition 用内存缓存，对于加载速度慢的 partition 用闪存缓存。其策略主要是周期性的检测 Partition 的加载速率，当速率小于某个阈值的时候，就使用闪存来缓存，当速率大于某个阈值的时候就使用内存来缓存。从下图可以看出，在实际生产中大多数 Partition 的加载速度是比较慢的，少部分加载速度比较快，加载速度比较慢的 Partition 占用了少部分内存，造成内存的低使用率，因此我们用闪存来承载这些 Partition，达到优化的目的。

Hybrid Technique #2

Real-world partition fill rates



二、未来工作

（一）低延迟查询

引入闪存之后，我们可以让 Reduce Task 直接从闪存中读取缓存的数据，而不是从 HDFS 中的文件读取数据，这样子提高了数据的读取速率。另外，在引入闪存之后，Shuffle 的数据块会变得更大，在 Reduce 端合并数据块的次数会变少，让整个查询变得更快。

（二）性能提升

当前，Cosco 为了保证容错性，每一份 Shuffle 数据在写入持久化的文件之前，会在不同节点的 Shuffle Service 保存两份数据。如果我们引入闪存之后，因为闪存具有不易失性，这样子 Shuffle Service 在恢复之后可以从闪存恢复数据，减少了拷贝的副本。另外，在引入闪存之后，数据块变得更大，在整个 DFS 上的读写也会更加高效。

三、性能评估技术

在上述的优化过程中，主要有如下四种类型的评估技术：

- Discrete event simulation
- Synthetic load generation on a test cluster
- Shadow testing on a test cluster
- Special canary in a production cluster

（1）Discrete event simulation

Discrete event simulation，也就是离散时间模拟的方法，是一种比较通用的评估方法。我们把每个 Shuffle Data 到达闪存的行为作为一个离散事件，记录其到达的时间、此时闪存中写入的数据总量以及最后闪存被 Flush 到 DFS 文件的数据总量。最终我们会得到如下图所示统计表，包含了最终数据块的大小和缓存的时间，由此我们就可以推算出数据块的加载速率，也就是对应 Partition 的加载速率，并且把这个速率应用于上文中讲到的第二种优化技术来进行决策。

Discrete Event Simulation

Drive simulation based on production data
cosco_chunks dataset

Partition	Shuffle Service ID	Chunk (DFS file) number	Chunk Start Time	Chunk Size	Chunk Buffering Time	Chunk Fill Rate (derived from size and buffering time)
3	10	5	2020-05-19 00:00:00.000	10 MiB	5000ms	2 MiB/s
42	10	2	2020-05-19 00:01:00.000	31 MiB	10000ms	3.1 MiB/s
...						
...						

(2) Special canary in a production cluster

如果我们要在一个生产环境中来验证我们所进行的优化是否有效是比较困难的,因为在 Cosco 中一个 Task 可以与多个 Shuffle Service 进行通信,所以很难确定是因为加入了闪存提升了性能还是因为其他原因而提升。因此,我们将整个生产集群分成两个互不干扰的子集群,然后进行对比试验,比如对子集群 A 增加闪存优化,而子集群 B 保持原来的部署模式。之后,我们再对两个子集群进行评估,就可以得知增加闪存优化是否起到了优化效果。

基于 Spark 和 TensorFlow 的机器学习实践

简介： 大数据以及计算能力的提升，使得 AI 技术有了突飞猛进的发展。在大数据和 AI 技术的热潮下，在 2019 杭州云栖大会机器学习技术专场，阿里云高级技术专家吴威和阿里云技术专家江宇向大家分享了 EMR E-Learning 平台和平台上新开发的核心特性 TensorFlow on Spark。

EMR E-Learning 平台

EMR E-Learning 平台基于的是大数据和 AI 技术，通过算法基于历史数据来构建机器学习模型，从而进行训练与预测。目前机器学习被广泛应用到很多领域，如人脸识别、自然语言处理、推荐系统、计算机视觉等。近年来，大数据以及计算能力的提升，使得 AI 技术有了突飞猛进的发展。



机器学习中重要的三要素是算法、数据和算力。而 EMR 本身是一个大数据平台，平台之上拥有多种数据，比如传统的数据仓库数据、图像数据；EMR 有很强的调度能力，可以很好地调度 GPU 和 CPU 资源；其结合机器学习算法，就可以成为一个比较好的 AI 平台。



典型的 AI 开发流程如下图所示：首先是数据收集，手机、路由器或者日志数据进入大数据框架 Data Lake；然后是数据处理，收集到的数据需要通过传统的大数据 ETL 或特征工程进行处理；其次是模型训练，经过特征工程或 ETL 处理后的数据会进行模型的训练；最后对训练模型进行评估和部署；模型预测的结果会再输入到大数据平台进行处理分析，整个过程循环往复。



下图展示了 AI 开发的流程，左侧是单机或者集群，主要进行 AI 训练和评估，包含数据存储；右侧是大数据存储，主要进行大数据处理，如特征工程等，同时可以利用左侧传输的机器学习模型进行预测。

AI 开发的现状主要有以下两点：

- 两套集群运维复杂：从图中可以看出，AI 开发涉及的两套集群是分离的，需要单独维护，运维成本复杂，容易出错。
- 训练效率较低：左右两侧集群需要大量数据传输和模型传输，带来较高的端到端训练的延迟。



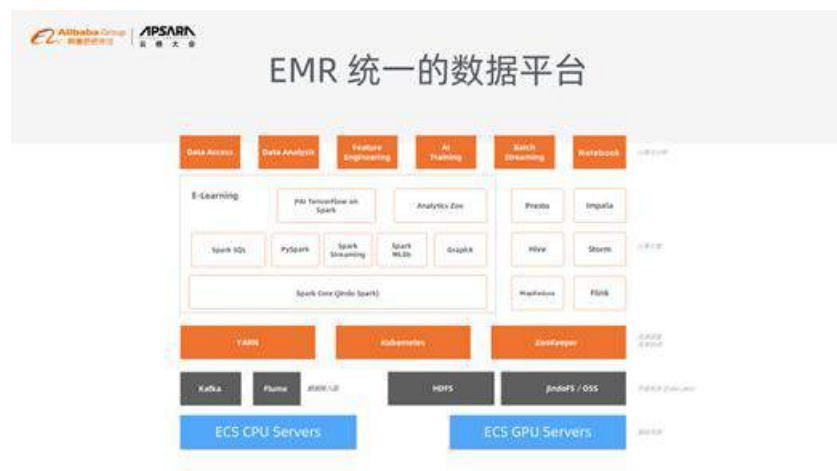
两套集群运维复杂

运维成本复杂，难于比较

训练效率较低

大数据集群与训练集群需要大量数据传输与模型传输，较高的网络延迟延迟

EMR 作为统一的大数据平台，包含了很多特性。最底层基础设施层，其支持 GPU 和 CPU 机器；数据存储层包括 HDFS 和阿里云 OSS；数据接入层包括 Kafka 和 Flume；资源调度层计算引擎包括 YARN、K8S 和 Zookeeper；计算引擎最核心的是 E-learning 平台，基于目前比较火的开源系统 Spark，这里的 Spark 用的是 jindo Spark，是 EMR 团队基于 Spark 改造和优化而推出的适用于 AI 场景下的版本，除此之外，还有 PAI TensorFlow on Spark；最后是计算分析层，提供了数据分析、特征工程、AI 训练以及 Notebook 的功能，方便用户来使用。



EMR 平台的特性主要有以下几点：

- 统一的资源管理与调度：支持 CPU、Mem 和 GPU 的细粒度的资源调度和分配，支持 YARN 和 K8S 的资源调度框架；
- 多种框架支持：包括 TensorFlow、MXNet 和 Caffe 等；
- Spark 通用的数据处理框架：提供 Data Source API 来方便各类数据源的读取，MLlib pipeline 广泛用于特征工程；
- Spark+深度学习框架：Spark 和深度学习框架的集成支持，包括高效的 Spark 和 TensorFlow 之间的数据传输，Spark 资源调度模型支持分布式深度学习训练；
- 资源监控与报警：EMR APM 系统提供完善的应用程序和集群监控多种报警方式；
- 易用性：Jupyter notebook 以及 Python 多环境部署支持，端到端机器学习训练流程等。



EMR E-Learning 集成了 PAI TensorFlow，其支持对深度学习的优化和对大规模稀疏场景的优化。



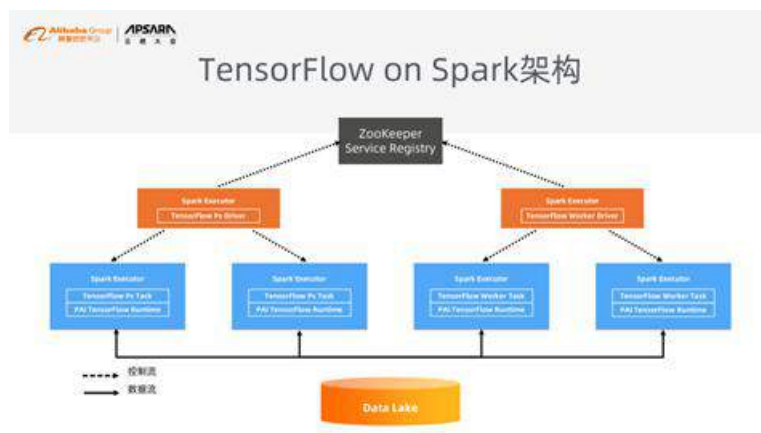
TensorFlow on Spark

经过市场调研发现, 大多数的客户在深度学习之前的数据 ETL 和特征工程阶段使用的都是开源计算框架 Spark, 之后的阶段广泛使用的是 TensorFlow,因此就有了将 TensorFlow 和 Spark 有机结合的目标。TensorFlow on Spark 主要包含了下图中的六个具体设计目标。

TensorFlow on Spark设计目标

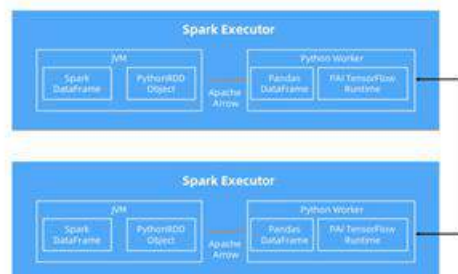
- 方便的与现有的Spark 数据处理流程结合
- 用户现有TensorFlow程序不需要改动就可以迁移
- 支持TensorFlow的所有功能
- 支持高效的数据传输, 加速从特征工程到训练时间
- PAI TensorFlow 底层的通信优化及大量的算法组件
- 快速支持各种框架接入, MXNet, Caffe

TensorFlow on Spark 从最底层来讲实际上是 PySpark 应用框架级别的封装。框架中实现的主要功能包括：首先调度用户特征工程任务，然后再调度深度学习 TensorFlow 任务，除此之外还需要将特征工程的数据高效快速地传输给底层的 PAI TensorFlow Runtime 进行深度学习和机器学习的训练。由于 Spark 目前不支持资源的异构调度, 假如客户运行的是分布式 TensorFlow, 就需要同时运行两个任务 (Ps 任务和 Worker 任务), 根据客户需求的资源来产生不同的 Spark executor, Ps 任务和 Worker 任务通过 Zookeeper 来进行服务注册。框架启动后会将用户写的特征工程任务调度到 executor 中执行, 执行后框架会将数据传输给底层的 PAI TensorFlow Runtime 进行训练, 训练结束后会将数据保存到 Data Lake 中, 方便后期的模型发布。



在机器学习和深度学习中，数据交互是可以提升效率的点，因此在数据交互部分，TensorFlow on Spark 做了一系列优化。具体来讲采用了 Apache Arrow 进行高速数据传输，将训练数据直接喂给 API TensorFlow Runtime, 从而加速整个流程

数据交互



TensorFlow on Spark 的容错机制如下图所示：最底层依赖 TensorFlow 的 Checkpoints 机制，用户需要定时的将训练模型 Checkpoint 到 Data Lake 中。当重新启动一个 TensorFlow 的时候，会读取最近的 Checkpoint 进行训练。容错机制会根据模式不同有不同的处理方式，针对分布式任务，会启动 Ps 和 Worker 任务，两个任务直接存在 daemon 进程，监控对应任务运行情况；对于 MPI 任务，通过 Spark Barrier Execution 机制进行容错，如果一个 task 失败，会标记失败并重启所有 task，重新配置所有环境变量；TF 任务负责读取最近的 Checkpoint。

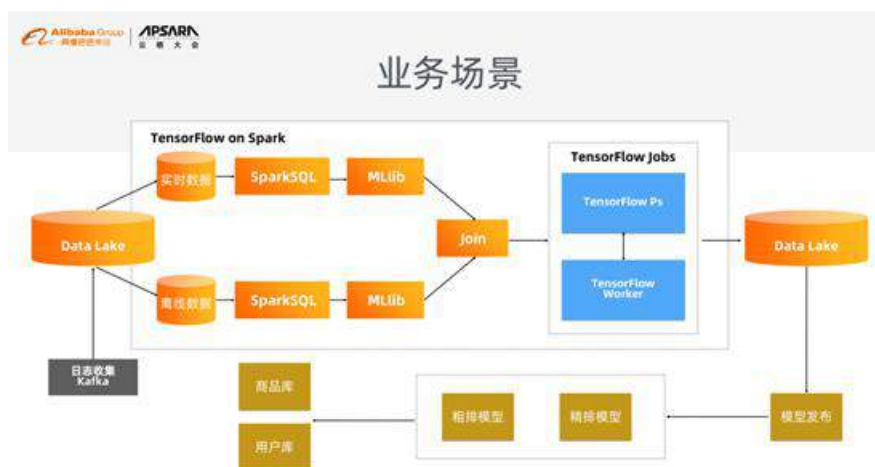


TensorFlow on Spark 的功能和易用性主要体现在以下几点：

- 部署环境多样：支持指定 conda，打包 python 运行时 virtual env 支持指定 docker
- TensorFlow 架构支持：支持分布式 TensorFlow 原生 PS 架构和分布式 Horovod MPI 架构
- TensorFlow API 支持：支持分布式 TensorFlow Estimator 高阶 API 和分布式 TensorFlow Session 低阶 API
- 快速支持各种框架接入：可以根据客户需求加入新的 AI 框架，如 MXNet



EMR 客户有很多来自于互联网公司, 广告和推送的业务场景比较常见, 下图是一个比较典型的广告推送业务场景。整个流程是 EMR 客户通过 Kafka 将日志数据实时推送到 Data Lake 中, TensorFlow on Spark 负责的是上半部分流程, 其中可以通过 Spark 的工具如 SparkSQL、MLlib 等对实时数据和离线数据进行 ETL 和特征工程, 数据训练好之后可以通过 TensorFlow 框架高效地喂给 PAI TensorFlow Runtime 进行大规模训练和优化, 然后将模型存储到 Data Lake 中。



在 API 层面, TensorFlow on Spark 提供了一个基类, 该基类中包含了三个方法需要用户去实现: `pre_train`、`shutdown` 和 `train`。 `pre_train` 是用户需要做的数据读取、ETL 和特征工程等任务, 返回的是 Spark 的 `DataFrame` 对象; `shutdown` 方法实现用户长连接资源的释放; `train` 方法是用户之前在 TensorFlow 中实现的代码, 如模型、优化器、优化算子的选择。最后通过 `pl_submit` 命令来提交 TensorFlow on Spark 的任务。



下图是推荐系统 FM 的样例，FM 是一个比较常见的推荐算法，具体场景是给电影评分，根据客户对之前电影评分、电影类型和发布时间为用户推荐潜在的电影。左侧是一个特征工程，用户可以使用 Spark data source API 读取电影和评分信息，原生支持 Spark 所有操作，如 join、ETL 处理等；右侧是 TensorFlow，进行模型、优化器的选择。目前整个系统的代码已经开源到 Github。



<https://github.com/aliyun/aliyun-emapreduce-demo/tree/master-2/src/main/python/deeplearning>

最后总结一下，EMR E-Learning 平台将大数据处理、深度学习、机器学习、数据湖、GPUs 功能特性紧密的结合，提供一站式大数据与机器学习平台；TensorFlow on Spark 提供了高效的数据交互流程以及完备的机器学习训练流程，将 Spark 与 TensorFlow 结合，借助 PAI TensorFlow，助力用户加速训练；目前 E-Learning 平台在公有云服务不同的客户，成功案例，CPU 集群规模超过 1000，GPU 集群规模超过 1000。



The slide features the Alibaba Cloud and APSARA logos in the top left corner. The title '总结' (Summary) is centered at the top. Below the title, there are three bullet points summarizing the platform's capabilities. At the bottom, two blue buttons highlight the scale of the clusters: 'CPU集群规模 1000+' and 'GPU集群规模 100+'.

总结

- EMR E-Learning 平台将大数据处理、深度学习、机器学习、数据湖、GPUs功能特性紧密的结合，提供一站式大数据与机器学习平台
- TensorFlow on Spark 提供了高效的数据交互流程以及完备的机器学习训练流程，将Spark与TensorFlow结合，借助PAI TensorFlow，助力用户加速训练
- E-Learning平台成功案例

CPU集群规模
1000+

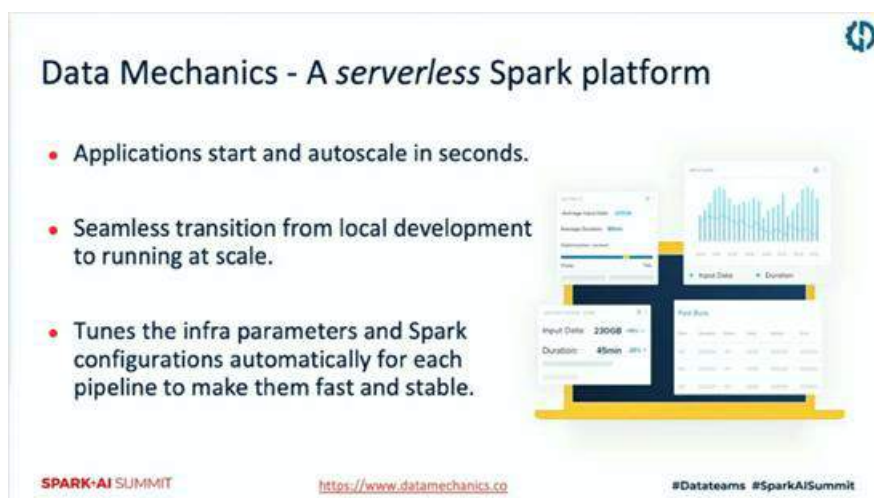
GPU集群规模
100+

在 kubernetes 上运行 apache spark：最佳实践和陷阱

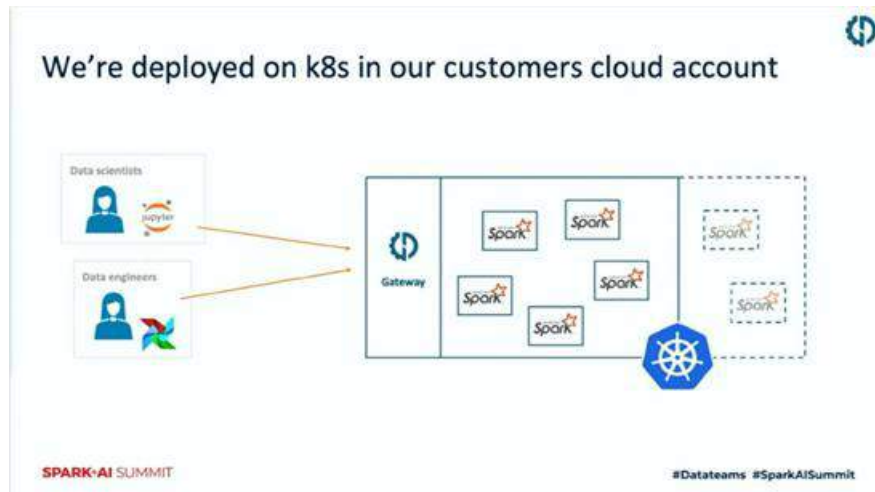
简介： 阿里云高级技术专家范振为大家带来在 kubernetes 上运行 apache spark 的介绍。内容包括 Data Mechanics 平台介绍，Spark on k8s，以及 EMR 团队云原生的思考和实践。以下由 Spark+AI Summit 中文精华版峰会的精彩内容整理。

一、Data Mechanics 平台介绍

这块是 data mechanics 平台的一个介绍。首先，它是一个 serverless 的平台，即一个全托管的平台，用户不用去关心自己的机器。所有的应用的启动和扩容都是在这种秒级的。然后，对于这种本地的开发转到这种线上的部署，它是一种无缝的转换。然后，它还能提供这种配置自动的 spark 的一些参数调优，整个这条 pipeline 都会做得非常的快，已经非常地稳定。



然后他们相当于是把整个的平台部署在用户的账号里边的 K8S 集群。这样的话，对整个的安全是一个非常大的保证。然后数据权限，包括运行权限，都在统一的账号里面。



二、Spark on k8s

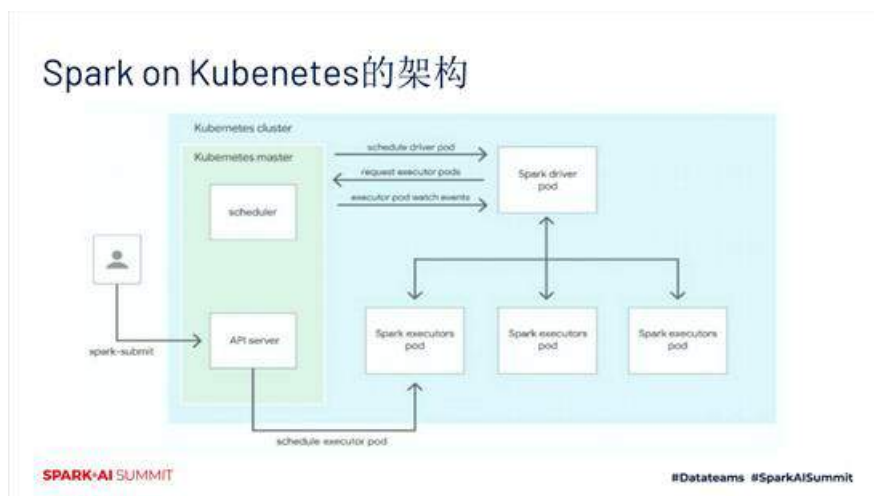
(一) 核心概念

首先，k8s 和 spark 的结合是出现在 spark 2.3 版本以后的事情，在此之前有几种方式。第一种就是 Standalone，大家使用的并不是非常的多。第二种是 Apache mesos，在国外用的比较多，但是市场规模也在逐渐缩小。第三种是 Yarn，我们现在绝大多数的企业都是跑在 Yarn 的集群里面了。第四种是 Kubernetes，现在大家也逐渐的把 spark 跑在 k8s 上面。

Kubernetes和spark的结合

- Standalone
- Apache mesos
- Yarn
- Kubernetes

Spark on k8s 的架构如下图所示：



提交应用的方式有两种。一种是 Spark submit，另一种是 Spark-on-k8s operator。它们各自的特点如下图所示：



然后我们再对比一下 Yarn 和 k8s 的依赖的管理。这块是区分点比较大的一个地方。Yarn 提供一个全局的 spark 版本，包括 python 的版本，全局的包的依赖，缺少环境隔离。而 k8s 是完全的环境隔离，每一个应用可以跑在完全不同的环境、版本等。Yarn 的包管理方案是上传依赖包到 HDFS。K8s 的包管理方案是自己管理镜像仓库，将依赖包打入 image 中，支持包依赖管理，将包上传到 OSS/HDFS，区分不同级别任务，混合使用以上两种模式。

依赖管理

Yarn

- 缺少环境隔离
 - 全局的spark版本
 - 全局的python版本
 - 全局的包依赖
- 包管理方案
 - 上传依赖包到HDFS

SPARK-AI SUMMIT

kubernetes

- 完全环境隔离
 - 每一个application可以跑在完全不同的环境、版本等
 - K8s Operator云原生方式部署
- 包管理方案
 - 自己管理镜像仓库
 - 将依赖包打入image中
 - 支持包依赖管理，将包上传到OSS/HDFS
 - 区分不同级别任务，混合使用以上两种模式

#Datateams #SparkAISummit

（三）配置和性能

然后我们说一下配置 spark executors 的小坑。举个例子，假设 k8s node 为 16G-RAM，4-core 的 ECS，下面的配置会一个 executor 都申请不到！如下图所示。

配置spark executors的小坑

假设k8s node为16G-RAM，4-core的ECS
下面的配置会一个executor都申请不到~！

- `spark.executor.cores=4`
- `spark.executor.memory=11g`

SPARK-AI SUMMIT

#Datateams #SparkAISummit

原因是什么，就是说 Spark pod 只能按照 node 资源的一定比例来申请资源，而 `spark.executor.cores=4` 占用了 node cores 全部资源。如下图所示，假设我们计算得到的可用资源是 85%，那么我们应该这样配置资源，`spark.kubernetes.executor.request.cores=3400m`。

K8s aware executor sizing

- 发生了什么？
 - Spark pod只能按照node资源的一定比例来申请资源
 - spark.executor.cores=4占用了node cores全部资源
- 计算可用资源
 - 预估node allocatable 95%
 - Daemonsets的资源占用 (say 10%)
 - 85%的pods可用资源
- 配置资源
 - spark.executor.cores=4
 - spark.kubernetes.executor.request.cores=3400m



SPARK-AI SUMMIT

#Datateams #SparkAISummit

然后这块是一个比较重要的特点，就是动态资源。动态资源的完整支持目前做不到。比如说，Kill 一个 pod，shuffle file 会丢失，会带来重算。

动态资源

- 动态资源的完整支持目前做不到。Kill一个pod，shuffle file会丢失，会带来重算。
- Ongoing work: spark-24432
- Spark3.0的特性

spark.dynamicAllocation.shuffleTracking.enabled	false	Experimental. Enables shuffle file tracking for executors, which allows dynamic allocation without the need for an external shuffle service. This option will try to keep alive executors that are storing shuffle data for active jobs.	3.0.0
spark.dynamicAllocation.shuffleTracking.timeout	infinity	When shuffle tracking is enabled, controls the timeout for executors that are holding shuffle data. The default value means that Spark will rely on the shuffles being garbage collected to be able to release executors. If for some reason garbage collection is not clearing up shuffles quickly enough, this option can be used to control when to time out executors even when they are storing shuffle data.	3.0.0

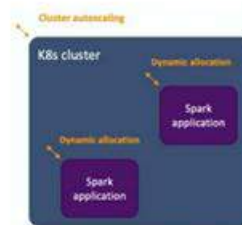
SPARK-AI SUMMIT

#Datateams #SparkAISummit

这一块是 Cluster autoscaling 和 dynamic allocation。上文中，我们看到 PPT 的某一页，它有一个实线框，还有一个虚线框。实际上说的是，k8s cluster autoscaler：当 pod 处于 pending 状态不能被分配资源时，扩展 node 节点。然后，Autoscaling 和动态资源配合起来工作，就是说，当有资源时，executor 会在 10s 内注册到 driver。而当没有资源时，先通过 autoscaling 添加 ECS 资源，再申请 executors。大约在 1min~2min 内完成 executor 申请过程。

Cluster autoscaling & dynamic allocation

- k8s cluster autoscaler: 当pod处于pending状态不能被分配资源时, 扩展node节点
- Autoscaling和动态资源配合起来工作:
 - 当有资源时, executor会在10s内注册到driver
 - 当没有资源时, 先通过autoscaling添加ECS资源, 再申请executors
 - 大约在1min~2min内完成executor申请过程
- Aliyun ACK/ASK, AWS EKS, GOOGLE GCP, AZURE AKS都有该功能, 需要安装autoscaler
<https://github.com/kubernetes/autoscaler>



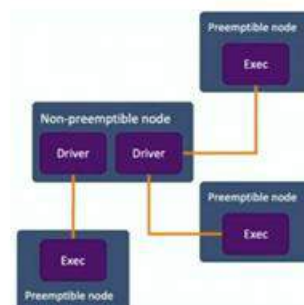
SPARK AI SUMMIT

#Datateams #SparkAISummit

这个实际上也是更好的保证了我们运行时的弹性, 还有一个我自己的理解, 比较有意思的一个玩法, 就是说更加省成本。Spot instance 会使得成本降低 75%, 它是可以被抢占的一种资源方式, 跑一些 SLA 不高、价格敏感的应用。它在架构上整体设计, 使得成本最低。如果 executor 被 kill, 会 recover。如果 driver 被 kill, 就会很危险。配置 node selector 和 affinities 来控制 Driver 调度在非抢占的 node, executor 调度在 spot instance。

更加省成本: spot (preemptible) instance

- Spot instance会使得成本降低75%
 - 可以被抢占的一种资源方式
 - 跑一些SLA不高、价格敏感的应用
- 架构上整体设计, 使得成本最低
- 如果executor被kill, 会recover
- 如果driver被kill, game over 危险~!
- 配置node selector和affinities来控制 Driver调度在非抢占的node, executor 调度在spot instance



SPARK AI SUMMIT

#Datateams #SparkAISummit

然后, 下一个就是对象存储的IO问题。Spark on k8s 通常都是针对对象存储, rename, list, commit task/job 会非常费时。如果没有 S3A committers, Jindofs JobCommitter, 应该设置 `spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version=2`。

对象存储的IO问题

- Spark on k8s通常都是针对对象存储（OSS，S3，WASB）
- rename, list, commit task/job会非常费时
- S3A committers, Jindofs JobCommitter
- 如果没有以上的committer, 应该设置
 - spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version=2
 - 之前EMR团队给过一些介绍
- 阿里云客户需求:
 - 打通ACK与EMR集群可以读写EMR的HDFS

SPARK·AI SUMMIT

#Datateams #SparkAISummit

还有 Shuffle 性能。 I/O 性能是 shuffle bound workload 的关键点，spark2.x 版本用 docker file system。而 Docker file system 是非常慢的，需要用 volume 来代替。

Shuffle性能

- I/O性能是shuffle bound workload的关键点，spark2.x版本用docker file system（可以配置成emptydir）
- Docker file system是非常慢的，需要用volume来代替
 - emptyDir：3.0的默认方式
 - Hostpath：可以利用宿主机的快速磁盘（nvme-based SSD）
 - Tmpfs：利用ram，危险

SPARK·AI SUMMIT

#Datateams #SparkAISummit

（四）未来工作

未来的工作，我认为比较重要的就是 shuffle 的提升，中间数据的存储与计算分离。这块是一个比较大的工作。另外，还有 Node decommission，支持上传 python 依赖文件等等。

Future works

- Shuffle提升：中间数据的存储与计算分离
 - Spark-25299
 - Enable完整的动态资源特性
 - Spark容忍executor pod和node级别的lost
 - Facebook cosco
 - Aliyun EMR jindo shuffle service
- Node decommission
 - Spark-20624
 - Copy shuffle data和cache data
- 支持上传python依赖文件
 - Spark-27936
- Job queue&资源管理

SPARK-AI SUMMIT

#Datateams #SparkAISummit

我们选择 k8s 的优缺点，如下图所示：

We choose Kubernetes, should you

Pors

- 云原生扩缩容，天然容器隔离性
- 所有技术栈统一为Kubernetes
- 在线离线服务统一调度
- 成本更低
- 运维一致性

Cons

- 需要学习Kubernetes
- 大规模调度性能、启动性能受限
- Shuffle & 动态资源

SPARK-AI SUMMIT

#Datateams #SparkAISummit

部署 spark on k8s 的具体步骤，如下图所示：

部署spark on k8s checklist

- Setup the infrastructure
 - 建立k8s cluster
 - 部署spark operator
 - 准备spark镜像
 - 部署spark history service
 - 部署log系统、Prometheus for monitor and metrics
- Config your apps for success
 - 配置好node池子，包括node labels等，确定好executor pod的规格打下
 - 配置好IO相关的jar，包括对象存储file committer等，配置好对应的volumes
 - 可选：配置好k8s cluster autoscaling，动态资源等
 - 可选：Spot instance等相关配置以节省成本
- Enjoy the Ride!

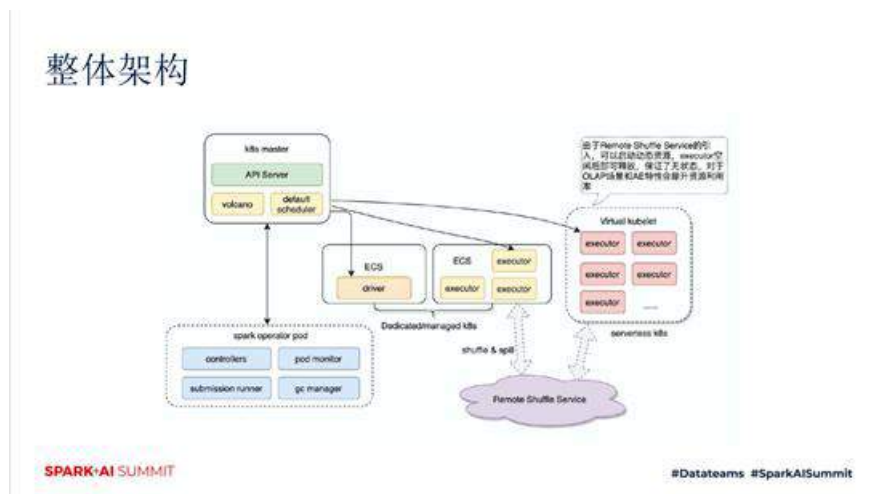
SPARK-AI SUMMIT

#Datateams #SparkAISummit

三、EMR 团队云原生的思考和实践

（一）整体架构

这块是我们的一个整体架构，如下图所示：



（二）动态资源&shuffle 提升

Shuffle service 解决核心问题：

- 解决动态资源问题
- 解决挂载云盘贵，事前不确定大小的痛点
- 解决 NAS 作为中心存储的扩展性以及性能问题
- 避免 task 由于 fetch 失败重新计算的问题，提升中大作业的稳定性
- 通过 Tiered 存储提升作业性能

（四）EMR Spark 云原生规划

EMR 产品体系，创建 EMR 集群类型为 ON ACK：

- JindoSpark 镜像
- JindoFSService/JindoFSSDK 提升访问 OSS 数据湖能力
- JindoJobCommitter 增强提交 OSS 作业能力
- JindoShuffleService&动态资源能力增强
- ACK 集群打通 EMR 原有集群，可以互访老集群表和 HDFS 数据
- Operator 增强，Dependency 管理，提供一站式管控命令
- 云原生日志、监控一站式平台

使用 RayOnSpark 在大数据平台上运行新兴的人工智能应用

简介：RayOnSpark 能够让 Ray 的分布式应用直接无缝地集成到 Apache Spark 的数据处理流水线中，省去集群间数据传输的 overhead，支持用户使用 Spark 处理的数据做新兴人工智能应用的开发。本次直播将由 Intel 大数据团队软件工程师黄凯为您介绍 Ray 和 Intel 的开源项目 Analytics Zoo，开发 RayOnSpark 的动机和初衷，同时结合实际案例分享 RayOnSpark 的落地实践。

演讲嘉宾简介：黄凯，Intel 大数据团队软件工程师，大数据和人工智能开源项目 Analytics Zoo 和 BigDL 的核心贡献者之一

本次分享主要围绕以下五个方面：

- 一、Overview of Analytics Zoo
- 二、Introduction to Ray
- 三、Motivations for Ray On Apache Spark
- 四、Implementation details and API design
- 五、Real-world use cases

一、Overview of Analytics Zoo

AI on Big Data

英特尔大数据团队近几年在助力人工智能落地方面做了很多工作，先后开源了两个项目。在 2016 年底开源了 BigDL，是基于 Apache Spark 开发的分布式高性能的深度学习框架，首次将深度学习引入到大数据平台中，让用户在大数据平台上更容易使用深度学习的算法。用 BigDL 写的深度学习应用是一个标准的 Spark 程序，可以运行在标准的 Spark 或 Hadoop 集群上，对集群不需要做任何特殊的修改。BigDL 在深度学习方面对标了现在流行的其他深度学习框架，和它们一样提供了丰富的深度学习功能。在性能方面 BigDL 利用并行计算，以及依赖于英特尔底层的库，如 MKL 等，使得 BigDL 基于 CPU 能有良好的性能。在可扩展性方面，BigDL 能通过 Spark 扩展到成百上千个节点上做对深度学习模型做分布式的训练和预测。

开源了 BigDL 之后，英特尔又开源了统一的数据分析和 AI 平台 Analytics Zoo，用户可以根据不同的需求，在大数据的平台上直接运行由使用 TensorFlow、PyTorch、Keras、Ray、等框架构建的应用。Analytics Zoo 可以将用户的大数据平台作为数据存储、数据处理挖掘、特征工程、深度学习等一体化的 pipeline 平台。

AI on Big Data

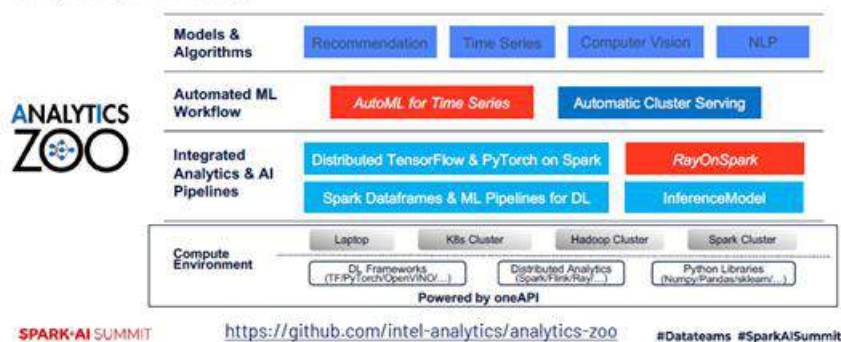


Analytics Zoo

Analytics Zoo 底层依赖于一系列现有的常用框架, 包括主流的深度学习框架、分布式计算框架、Python 数据处理库等, 在这些框架之上搭建了一套非常完整的数据分析和人工智能的流水线, 包括支持用户在 Spark 上跑分布式的 TensorFlow 和 PyTorch, 只需要做很小的代码改动就可以在大数据平台运行主流的深度学习框架; 对 Spark DataFrame 和 ML Pipeline 提供了原生的深度学习支持; 也提供了轻量级的 API 对训练好的模型做线上推理。在流水线之上, Analytics Zoo 提供了 ML workflow, 帮助用户自动化地去构建大规模的人工智能应用, 比如对时间序列做可扩展的预测, 以及分布式 Cluster Serving。最上层对很多常见领域, 如推荐、时间序列、计算机视觉、自然语言处理等等, 提供了开箱即用的模型以及参考案例。

Analytics Zoo

Unified Data Analytics and AI Platform for distributed TensorFlow, Keras and PyTorch on Apache Spark/Flink & Ray



实际工作中, 开发部署一条数据分析和 AI 的流水线通常需要经历三个步骤: 开发者首先在笔记本上使用样本数据完成开发的原型, 然后使用历史几个月的数据在集群上做实验, 实验结果没有问题的话再到生产环境中进行大规模的部署。我们希望在执行三个步骤中, 用户几乎不需要改动, 就能将单机的代码无缝地部署在生成环境中, 并且简化和自动化搭建整个 pipeline 的过程, 这也是开发 Analytics Zoo 和 RayOnSpark 的初衷和目的。

Unified Data Analytics and AI Platform



Seamless Scaling from Laptop to Distributed Big Data Clusters



Easily prototype end-to-end pipelines that apply AI models to big data.

"Zero" code change from laptop to distributed cluster.

Seamlessly deployed on production Hadoop/K8s clusters.

Automate the process of applying machine learning to big data.

SPARK AI SUMMIT

#Datateams #SparkAISummit

二、Introduction to Ray

Ray 是由 UC Berkeley 开源的一个能够非常快速和简单地构建分布式应用的框架, Ray Core 提供了非常友好的 API, 帮助用户更容易地并行处理任务。Python 用户只需要增加几行代码就可以直接并行地执行 Python 函数和对象。简单来说, 用户首先需要 import ray, 调用 ray.init() 启动 Ray 服务。正常情况下, 在一个循环中调用多次 Python 函数是顺序执行的, 但是如果加上 @ray.remote(num_cpus, ...) 的 Python 修饰器, 就可以去并行执行这些 Python 函数, 最后通过 ray.get 得到返回值。同样对 Python class 也能加上 @ray.remote, 变成 Ray actor 能够被 Ray 去远程地启动。在 @ray.remote 中还可以指定运行所需资源, 比如需要多少 CPU 等, 在运行过程中 Ray 会预留这些资源。Ray 可以支持单机和集群上的并行运行。

Ray



Ray is a fast and simple framework for building and running distributed applications.

Ray Core provides easy Python interface for parallelism by using remote functions and actors.

```
import ray
ray.init()

@ray.remote(num_cpus=...)
def f(x):
    return x * x

# Executed in parallel
ray.get([f.remote(i) for i in range(5)])

@ray.remote(num_cpus=...)
class Counter(object):
    def __init__(self):
        self.n = 0
    def increment(self):
        self.n += 1
        return self.n

counters = [Counter.remote() for i in range(5)]
# Executed in parallel
ray.get([c.increment.remote() for c in counters])
```



SPARK AI SUMMIT

#Datateams #SparkAISummit

除了直接使用 Ray Core 实现简单的并行之外, Ray 还提供了一些 high-level 的 library, 加速人工智能 workload 的构建。其中 Ray Tune 能自动去调参, RLlib 提供统一的 API 去执行不同强化学习任务, Ray SGD 在 PyTorch 和 TensorFlow 原生的分布式模块之上实现了一层 wrapper 来简化部署分布式训练的过程。

Ray



Ray is packaged with several high-level libraries to accelerate machine learning workloads.

Tune: Scalable Experiment Execution and Hyperparameter Tuning

RLlib: Scalable Reinforcement Learning

RaySGD: Distributed Training Wrappers

<https://github.com/ray-project/ray/>



SPARK AI SUMMIT

#Datateams #SparkAISummit

三、Motivations for Ray On Apache Spark

Ray 可以让用户很容易的构建新兴的人工智能的应用，在实际工作过程中也越来越需要将这些新兴的人工智能技术应用在生产数据上，来创造更多的价值。但其实用户在这个过程中会往往面临一些挑战：首先，生产环境中的数据通常是大规模存储在大数据集群上，而直接在大数据集群上部署 Ray 并不容易。其次，如何提前在集群的所有节点上准备好运行所需要的 Python 环境和依赖，同时不给整个集群带来副作用。第三，如果用两个不同的系统分别进行数据处理和 AI 任务，不可避免地会带来数据传输的 overhead，还需要额外的资源去维护不同的系统和工作流。这些挑战促使了英特尔开发 RayOnSpark，希望用户可以直接在大数据分析的流水线上嵌入用 Ray 开发的新兴人工智能应用。

Motivations for RayOnSpark



Demand to embrace emerging AI technologies on production data.

Efforts required to directly deploy Ray applications on existing Hadoop/Spark clusters.

Challenge to prepare the Python environment on each node without modifying the cluster.

Need a unified system for big data analytics and advanced AI applications.

SPARK AI SUMMIT

#Datateams #SparkAISummit

四、Implementation details and API design

RayOnSpark 架构

开发 RayOnSpark 是为了 Ray 的分布式应用能直接无缝地集成到 Spark 数据处理的流水线中。顾名思义，RayOnSpark 把 Ray 跑在了 Spark 大数据集群之上，后面的介绍以 YARN 集群为例，同样的思路也可用于 Kubernetes 或者 Apache Mesos 等集群。在环境准备方面，我们使用 conda-pack 打包 Python 环境，在运行时分发到各个节点上，这样一来用户不需要在每个节点上提前装好 Python 依赖，程序结束之后集群环境也不会受到影响。下图右侧是 RayOnSpark 整体架构，Spark 会在 Driver 节点上起一个 SparkContext 的实例，SparkContext 会在整个集群起多个 Spark Executor 执行 Spark 的任务。除了 SparkContext 之外，RayOnSpark 设计中还会在 Spark Driver 中创建一个 RayContext 的实例，利用现有的 SparkContext 将 Ray 在集群里启动起来，Ray 的进程会伴随着在 Spark Executor，包括一个 Ray Master 进程和它的 Raylet 进程。RayContext 也会在 Spark Executor 中创建 RayManager 来管理这些 Ray 的进程，任务结束后自动将 Ray 的进程关掉，同时释放 Ray 所占用的资源。在同一个 YARN 集群上同时有 Spark 和 Ray，这样我们就能够将 in-memory 的 Spark RDD 或 DataFrame 直接运行在 Ray 的应用中，使用 Spark 的数据做新兴人工智能应用的开发。

Implementation of RayOnSpark

RayOnSpark allows Ray applications to seamlessly integrate into Spark data processing pipelines.

Leverage conda-pack and Spark for runtime Python package distribution.

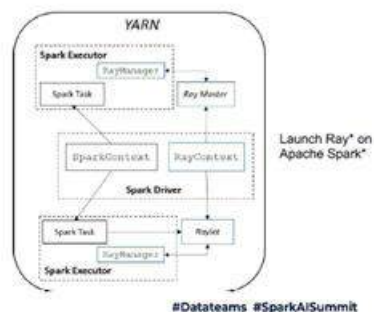
RayContext on Spark driver launches Ray across the cluster.

Ray processes exist alongside Spark executors.

For each Spark executor, a Ray Manager is created to manage Ray processes.

Able to run in-memory Spark RDDs or DataFrames in Ray applications.

SPARK AI SUMMIT



RayOnSpark 使用方法

RayOnSpark 的使用非常简单，只需要三步。首先要 import Analytics Zoo 中的包，通过 `init_spark_on_yarn` 方法创建 `SparkContext` object，会自动将指定 conda 环境的 Python 依赖打包好分发给所有的 Spark Executor。第二步，创建 `RayContext` object，这是连接 Ray 和 Spark 的桥梁，在创建的时候可以定义 Ray 的参数，如给多大的 `object_store_memory` 等。下图右侧红色框是需要加的 RayOnSpark 代码，黑色框是用 Ray 直接写的代码。在 Ray 项目执行完成后，调用 `ray_ctx.stop()` 就可以关掉 Ray 的集群。更多的介绍可以参见：<https://analytics-zoo.github.io/master/#ProgrammingGuide/rayonspark/>

Interface of RayOnSpark

Three-step programming with minimum code changes:

- Initiate or use an existing *SparkContext*.
- Initiate *RayContext*.
- Shut down *SparkContext* and *RayContext* after tasks finish.

More instructions at: <https://analytics-zoo.github.io/master/#ProgrammingGuide/rayonspark/>

SPARK AI SUMMIT



```
import ray
from zoo import init_spark_on_yarn
from zoo.ray import RayContext

sc = init_spark_on_yarn(hadoop_conf, conda_name,
num_executors, executor_cores,...)
ray_ctx = RayContext(sc, object_store_memory,...)
ray_ctx.init()
```

RayOnSpark code

```
@ray.remote
class Counter(object):
    def __init__(self):
        self.n = 0
    def increment(self):
        self.n += 1
        return self.n
counters = [Counter.remote() for i in range(5)]
ray.get([c.increment.remote() for c in counters])
```

Pure Ray code

```
ray_ctx.stop()
sc.stop()
```

#Datateams #SparkAISummit

五、Real-world use cases

RayOnSpark 的第一个应用是我们在 Analytics Zoo 里基于 Ray Tune 和 RayOnSpark 开发的 AutoML 模块。在大数据平台上构建时序应用非常复杂，需要很多流程，如特征提取、选择模型、调整超参等等。

利用 AutoML 可以将这些过程自动化，简化搭建时间序列模型过程。感兴趣的同学可以参见：

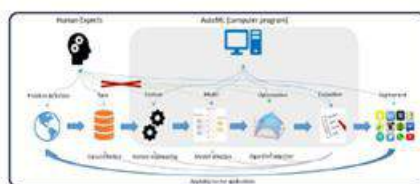
<https://github.com/intel-analytics/analytics-zoo/tree/master/pyzoo/zoo/automl>，了解更多的使用方法和 use cases。

Use Cases of RayOnSpark

Scalable AutoML for time series prediction.

- Automate the feature generation, model selection and hyperparameter tuning processes.

- See more at: <https://github.com/intel-analytics/analytics-zoo/tree/master/pyzoo/zoo/automl>.



Source: Yao, G., Wang, et. al Taking the Human out of Learning Applications: A Survey on Automated Machine Learning.

SPARK AI SUMMIT

#Datateams #SparkAISummit

除了 AutoML，我们还基于 RayOnSpark 实现了数据并行的神经网络训练的 pipeline。用户可以使用 PySpark 或者 Ray 并行进行数据加载和处理，我们对不同深度学习框架使用 RayOnSpark 实现了 wrapper，去自动化地搭建分布式训练的环境。对用户来说，不再需要关心很多复杂的分布式环境搭建问题，只需要在单机上实现模型原型，使用 RayOnSpark，通过简单的代码修改就可以完成大数据集群上分布式模型的训练。

Use Cases of RayOnSpark



Data parallel pre-processing and distributed training pipeline of deep neural networks.

- Use PySpark or Ray for parallel data loading and processing.
- Use RayOnSpark to implement thin wrappers to automatically setup distributed environment.
- Run distributed training with either framework native modules or Horovod (from Uber) as the backend.
- Users only need to write the training script on the single node and make minimum code changes to achieve distributed training.

SPARK-AI SUMMIT

#Datateams #SparkAISummit

合作案例：Drive-thru Recommendation System at Burger King

英特尔和汉堡王合作，针对 drive-thru 场景（即用户开车到快餐门店，不需要下车，直接通过门口的麦克风对话），基于 RayOnSpark 构建了一个完整的推荐系统流水线。汉堡王作为全球最大的快餐品牌之一，每天都会收集很多的数据，这些数据会在 Spark 集群上面做数据清洗和预处理，再做分布式训练。汉堡王选择使用 MXNet 作为深度学习框架。在与英特尔合作之前，他们单独使用了一个 GPU 集群做 MXNet 分布式训练。从 Spark 集群拷贝数据到 GPU 集群上，无疑使得他们耗费了很多时间。英特尔提供的解决方案是使用 RayOnSpark，直接在 Spark 的集群上做分布式的训练，这样一来数据不需要再额外进行拷贝，且非常容易扩展。类似于 RaySGD，在 MXNet 上实现了一层轻量级的 wrapper layer，使得分布式 MXNet 训练能很容易地在 YARN 集群上部署。MXNet Worker 和 Server 都在 Ray 进程中运行，通过 Ray Manager 管理。整个 pipeline 只需要一个集群就可以处理分布式训练任务，目前基于 RayOnSpark 的解决方案已经被汉堡王部署到了他们的生产环境中，证明了这种方案更加高效、更容易维护并且有更好的扩展性。

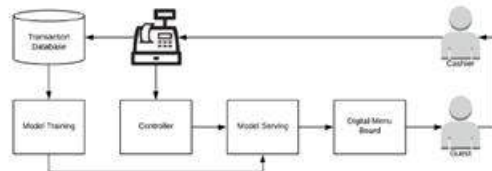
Drive-thru Recommendation System at Burger King



Burger King performs Spark ETL tasks first, followed by distributed MXNet training.

Similar to RaySGD, we implement a lightweight shim layer around native MXNet modules for easy deployment on YARN cluster.

The entire pipeline runs on a single cluster. No extra data transfer needed.



SPARK-AI SUMMIT

#Datateams #SparkAISummit



使用 Ray 将可扩展的自动化机器学习 (AutoML) 用于时序预测

简介：机器学习和深度学习在时序预测上有更好的表现，前提是生成好的模型。但训练出好的模型并不是那么容易的，尤其是那些新手，这也就说明了为什么 AutoML 越来越火。在 Analytics Zoo 当中用户可以使用 AutoML，在很短的时间内得到满足准确度要求的模型。在 2020 Spark+AI 峰会直播中，由 Intel 高级架构师黄晟盛为您介绍时序应用典型场景，基于 AutoML 的时序解决方案，同时结合实际案例与大家分享与合作客户的合作经验和反馈。

演讲嘉宾简介：黄晟盛，Intel 高级架构师，Apache Spark committer，PMC member Analytics Zoo 和 BigDL 重要贡献者。

本次分享主要围绕以下四个方面：

- 一、Background
- 二、Scalable AutoML for Time Series
- 三、Use Case Sharing & Learnings
- 四、Future Work

Background

Analytics Zoo

BigDL 是英特尔 Spark 团队的第一个开源项目，基于 Spark 的深度学习框架，对标的是 TensorFlow、Caffe 等库，在 BigDL 中实现了大量的神经网络层，整个模型的训练可以以 Spark job 的形式跑在大数据集群上。后来在 BigDL 基础上又开源了 Analytics Zoo，为大数据用户提供了统一的端到端的大数据+深度学习的平台，为用户提供单机到集群的无缝式体验。Analytics Zoo 集成了多种软硬件加速库，为主流的深度学习框架 TensorFlow，PyTorch，Keras 等等提供了更便捷的分布式工作流支持。同时提供了大量的预定义的模型和参考案例，使得用户使用起来更方便更高效。下图展示了 Analytics Zoo 架构概览，Analytics Zoo 可以跑在各种环境当中，包括笔记本、K8S 集群、Hadoop Cluster、Spark Cluster 等等。Analytics Zoo 的底层有 pipeline 的支持，这些流水线组件可以使得用户更方便的将深度学习框架使用到工作学习当中，比如分布式的 TensorFlow 和 PyTorch 支持、RayOnSpark、Spark DataFrame、ML pipelines for DL、Inference Model 等等。流水线之上提供了 Workflow 方面的支持，如自动调参，自动 Cluster Serving 等。最上层针对不同的用户场景，提供了场景的算法，使得用户模型的搭建更加简便。



Time Series

时间序列数据指的是在时间轴上有序的样本，一般是数值类型，可以是标量也可以是矢量。标量是单变量的预测，矢量是多变量预测。常见的时序数据预测包括一段时间的股票价格预测、商品销量、监控指标、传感器设备数据。下图展示的是 2014 年到 2015 年出租车乘客数量时序数据，可以看到这份时序数据是有一定周期规律的，但也不是非常的规整，现实世界的。时序数据是多种多样的，各自可能有有不同的特征。

对时序数据的分析有很多种类。时序预测是首先是用过去可以预测未来。时序异常检测，另一个是可以发现异常时序特征的样本。此外还有有时序分类及时序聚类等等。时序预测可以作为在线异常检测的前序步骤，首先根据过去样本数据预测将来数据，当实际值到达时，去比较预测值和实际值之间的差距，当差距过大时可以认为新的数据可能是异常。时序分析本身也有多种应用，销量或资源的预测可以用于仓储管理和资源人力分配，对通信网络的 KPI 分析可以用于通信网络质量检测，针对设备传感器数据传感器设备的预测可以用于高价值设备的维护，针对服务监控指标分析可以用于智能运维。AIOps 是一个常常用到时间序列分析的领域，企业的系统迁移到云端之后，传统的运维方法很难管理大规模的虚拟机和服务，AIOps 试图使用大数据和 ML、AI 的技术在监控数据中进行挖掘和分析，自动化探测和修复问题，从而节省运营成本，减少错误修复平均时间。常见的数据分析在 AIOps 的使用场景包括：是对虚拟机状态的异常监测和告警，以及异常和告警的根源分析，再根据长期或短期的趋势预测指导资源的规划和配置。

Time Series In a Nutshell



Time Series data

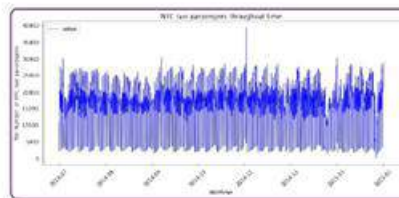
- A series of data that is observed sequentially in time.
- stock prices, sales volume, CPU/I/O monitoring metrics, KPIs in telecom networks ...

Time Series Analysis

- Time Series Forecasting
- Anomaly Detection
- Time Series Classification, clustering, etc.

Applications

- Demand forecasting
- network quality management
- predictive maintenance
- AIOps



Total volume of taxi passengers in NYC from 2014/07-2015/02 (source : https://github.com/mherzig/nytaxi/blob/master/nytaxi_anomaly_detection/nytaxi_anomaly_detection.py#L49)

SPARK AI SUMMIT

#Datateams #SparkAISummit

Time Series Forecasting

时序预测这个问题可以定义为：已知认为从第 1 个观测点到第 t 个观测点的数值是 y_1 到 y_t ，目标是预测点是 y_{t+1} 到 y_{t+h} ， h 可以大于等于 1，取决于具体场景。理论上，从第一个观测点开始的所有历史数据都是可以用来预测未来使用的，但是相隔太久远的数值对当前预测的意义不大，也会加大模型的负担，所以有。一般我们经常实际上，使用 t 时刻之前最邻近的 k 个样本点数值，如下图中标记的紫色部分，从 y_{t-k+1} 到 y_t 。很多种方法可以用于时序预测，如自回归、指数平滑回归、ARIMA 等等在现实中有很多使用基础，相对简单，不擅长捕捉复杂的时序模式，也不擅长捕捉曲线之间的关联。最近有很多基于机器学习和深度学习做时序预测。传统方法通常需要依赖于领域的专业知识，而且对数据的统计分布有假设。基于机器学习和深度学习的方法是大多是数据驱动的，而且很多实验结果表明，这些方法会比传统方法得到更好的预测效果。

Time Series Forecasting



Problem Definition

- Given all history t observations y_1, \dots, y_t , Predict values of next h steps, y_{t+1}, \dots, y_{t+h}
- Usually only lookback k steps, y_{t-k+1}, \dots, y_t

Forecasting Methods

- Autoregression, Exponential Smoothing, ARIMA, ...
- Machine Learning and Deep Learning methods



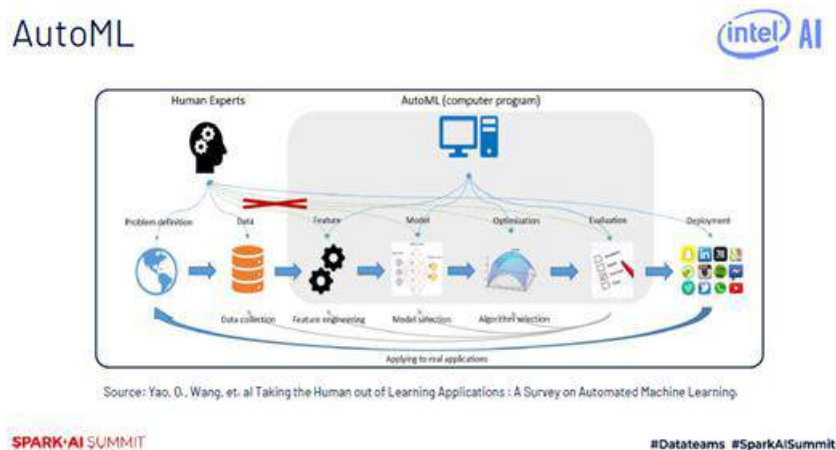
SPARK AI SUMMIT

#Datateams #SparkAISummit

时序预测要预测数值，本质上是回归问题，但时序预测相比普通的回归问题有一些特殊性，除了模型本身需要考虑序列和趋势的特征之外，而且 train 数据时间戳需要按照顺序划分，在交叉验证时注意使用时序验证方法。还需要特别关注采样区间、采样频率、采样不规则等问题。

AutoML

机器学习和深度学习在时序预测上比传统方法有更好的表现，前提是生成好的模型。但训练出好的模型并不是那么容易的，尤其是那些没有多少经验的用户，这也就说明了为什么 AutoML 越来越火。下图中展示了使用机器学习构建模型的大致步骤，从问题定义、收集数据、特征工程、建模、训练、评估到部署等。特征工程到评估的步骤是非常耗时且需要多轮迭代，还需要基于建模者的经验调参。AutoML 希望将耗时的机器学习步骤进行自动化，以此来减轻建模者调优的工作量，降低机器学习门槛。AutoML 初级版本基本思路是将模型参数进行排列组合，逐一跑一遍，找最好的一组参数。现代 AutoML 把自动寻找最优模型的这个任务抽象成了一个新的优化问题，也就是：在一个给定的预算之内去寻找一个最优化的目标指标，所使用的方法呢也远比简单的网格搜索和随机搜索要更加复杂。目前，AutoML 将这些步骤切换成了新的优化问题，在给定的范围之内寻找最优的目标指标，所使用的方法远比简单的网络搜索或随机搜索更加复杂。AutoML 不仅可以用于网络搜索，还可以用于机器学习的深度搜索。在 Analytics Zoo 当中用户可以用 AutoML 得到更优的时序模型。同时减少预处理和特征工程方面的工作。



SPARK AI SUMMIT

#Databricks #SparkAISummit

Ray and RayOnSpark

在考虑如何实现 Analytics Zoo 当中的 AutoML 模块时，英特尔选择了 Ray Tune。Ray Tune 在 Ray 上进行可扩展实验和超参调优库，Ray 是为新型的 AI 用户打造的分布式框架。RayOnSpark 是 Analytics Zoo 上的新功能，使得用户可以方便的将 Ray 跑在 Spark 大数据集群之上。

Ray and RayOnSpark



Ray

- A distributed framework for emerging AI applications

Ray Tune

- A library on Ray for experiment execution and hyperparameter tuning

RayOnSpark

- a feature in Analytics Zoo
- Directly run Ray programs on big data cluster
- Seamlessly integrate ray into spark data processing pipeline

<https://ray.readthedocs.io/en/latest/>
<https://analytics-zoo.github.io/projects/#programmabledata/rayonspark>

SPARK AI SUMMIT

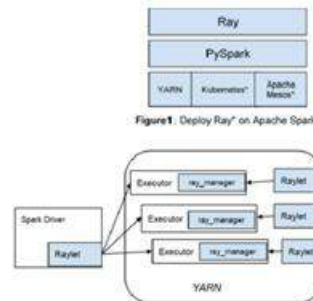


Figure 1: Deploy Ray* on Apache Spark*

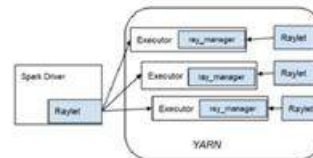


Figure 2: Launch Ray* process on Apache Spark*

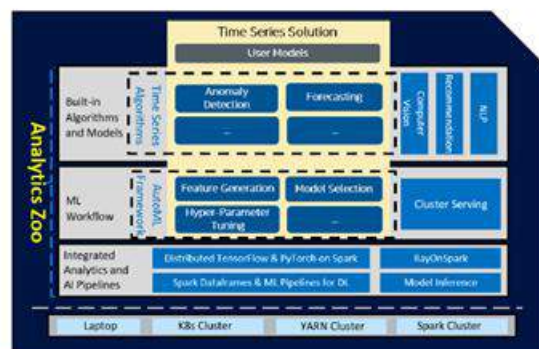
#Datateams #SparkAISummit

三、Scalable AutoML for Time Series

Time Series Solution In Analytics Zoo

下图中黄色部分是 Analytics Zoo 当中的 Time Series Solution，最底层是 AutoML 框架，框架中对流水线的各个部分进行了自动化，如特征生成、模型选择、调优等。上一层是专转为时序模型定制的算法模块，如时序预测和异常检测。用户可以基于自己的场景构造时序应用。Time Series Solution 有三种特性，首先是提供了丰富的算法，包括统计、神经网络、混合模型等。第二是提供了基于 AutoML 的自动调参，可以对特征生成，模型选择，超参，调优等步骤进行自动化，大量节约调参人工消耗。第三了由于背靠 Analytics Zoo，可以使用更多 AI pipeline 中的功能，使得整个自动化训练过程更加高效，还可扩展在集群之上。

Time Series Solution In Analytics Zoo



SPARK AI SUMMIT

Rich algorithms

statistical, neural-networks, hybrid state-of-art models, etc.

AutoML

for automatic feature generation, model selection, hyper-parameter tuning, etc.

Seamless scaling

with integrated analytics and AI pipelines

#Datateams #SparkAISummit

Software Stack

下图中，中间的 AutoML 框架中包含四个组件，用蓝色方框表示，。FeatureTransformer 和 Model 是机器学习不同环节的抽象，可以把它们看作接受一些当作参数的黑盒子——这些参数被它们各自用于构造内部的流水线结构，以及控制后续的执行流程。，具体来说，FeatureTransformer 可以做特征生

成，特征选择等任务，Model 可以做模型构建及训练，以及多模型选择。SearchEngine 负责超参组合和启动实验，并且负责所有实验在分布式集群上的调度。当超参过程结束之后，实验结果进行回收，通过最好的模型结果和超参组合构建 pipeline。Pipeline 是端到端的处理过程，包含预处理，特征工程和模型。Pipeline 可以被存储下来后再加载，做推理和增量式训练，所有模块都可以被扩展用于其他种类的被分析任务，不仅限于并且用于时序场景。在 AutoML 框架中有两个灰色的框，在计划中但目前还没有实现。

在这个 general 的 automl 的框架之上基于 general 的 AutoML 框架，实现了针对时序特征的模块，其中 TimeSequencePredictor 负责自动化模型训练过程，TimeSequencePipeline 作为时序预测 pipeline 的输出。在 AutoML 框架中有两个灰色的框，目前还没有实现。

Software Stack

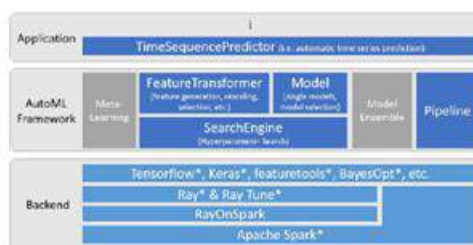


AutoML Framework

- FeatureTransformer
- Model
- SearchEngine
- Pipeline

Time Series upon AutoML

- TimeSequencePredictor
- TimeSequencePipeline



<https://medium.com/mindia/hybrid-automl-for-time-series-prediction-with-ray-and-mlflow-search-pipeline-711e11e>

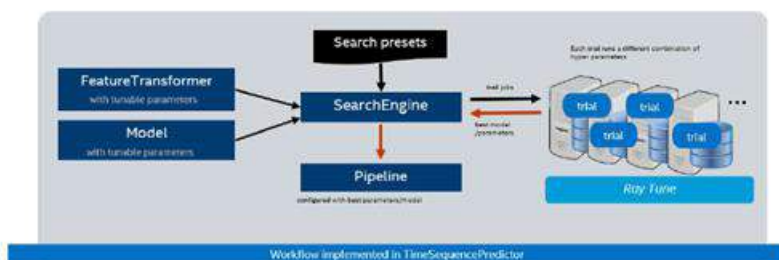
SPARK·AI SUMMIT

#Datateams #SparkAISummit

自动化训练过程

大体而言，使用 SearchEngine 驱动整个训练过程。SearchEngine 在构造的时候首先接收一个 FeatureTransformer 和一个 Model，同时需要 Search presets。SearchEngine 会根据 Search presets 确定超参组合，启动实验，并且将多个实验在 Ray 的集群上进行调度。每个实验都有不同的参数组合来构造内部流水线和最终执行的训练流程。当实验结束时，将结果结构写到 HDFS 中。全部实验结果都被回收和分析，根据最好的参数组合和训练好的模型构建 pipeline，最终返回给用户。

Training at Runtime



SPARK·AI SUMMIT

#Datateams #SparkAISummit

A glimpse of API

要训练一个模型，首先需要创建 TimeSequencePredictor，然后调用在 fit 中调用 Predictor，启动自动训练过程。fit 中有两个重要的参数，一个是 recipe，具体表示用什么策略调参，如参数搜索范围，采样分布等等。目前已经设置了一些公用的 recipe，用户可以简单的调用，同时可以自定义 recipe。另一个参数是 distributed，表明搜索过程是分布式还是单机的，在单机模式下可以适当减少开销。Fit 结束后返回 TimeSequencePipeline，可以被回收加载，Pipeline 还提供 evaluate, predict 和 fit(incremental) 等 API 用于评估，预测和增量式训练。

A glimpse of API



Training a Pipeline

- fit (w/ automi)
- recipe
- distributed mode

Using a Pipeline

- save/load
- evaluate/predict
- fit (incremental)

```
from zoo.automi.regression.time_sequence_predictor import timesequencepredictor
tsp = timesequencepredictor( dt_col="datetime",
                             target_col="value",
                             extra_features_col=None,
                             future_seq_len=1)

pipeline = tsp.fit(train_df,
                   metric="mean_squared_error",
                   recipe=RandomRecipe(num_samples=100),
                   distributed=True)
```

```
pipeline.save("/tmp/saved_pipeline/my.ppl") #save

from zoo.automi.pipeline.time_sequence import load_ts_pipeline
pipeline = load_ts_pipeline("/tmp/saved_pipeline/my.ppl") #load
ex = pipeline.evaluate(test_df, metric=["r_square"]) # evaluation
result_df = pipeline.predict(test_df) # inference
pipeline.fit(newtrain_df, epoch_num=1) # incremental training
```

SPARK AI SUMMIT

#Datateams #SparkAISummit

Use Case Sharing & Learnings

Project Zouwu

Zouwu 是在前面所介绍的模块的加持上实现的新的项目，专门为电信领域的时序应用所打造的，不仅仅提供基于 AutoML 的时序预测方案，还会提供了一些单独的时序预测模型给用户，同时会为电信用户的典型场景提供了参考解决解决方案，如网络流量预测。

Project Zouwu



Use case

- reference time series use cases for Telco (such as network traffic forecasting, etc.)

Models

- built-in models for time series analysis (such as LSTM and MTNet)

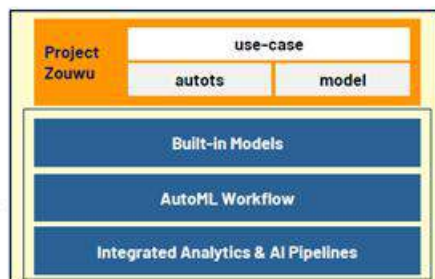
"AutoTS"

- AutoML support for building E2E time series analysis pipelines (including automatic feature generation, model selection and hyperparameter tuning)

<https://github.com/intel-analytics/analytics-zoo/tree/master/pyzoo/zoo/zouwu>

SPARK AI SUMMIT

#Datateams #SparkAISummit



电信网络流量预测

电信网络在实际运营过程当中有很多 KPI，如上下行速率，连接速率等，这些 KPI 的预测在电信应用中非常广泛，可以做节能，资源调度。电信网络的 KPI 可以反映实时的客户需求，如活跃用户数量等。电信运营商如果可以预测每个小区的 KPI 的话，就可以预测哪些小区未来一段时间实际承载的业务量比较低。运营商可以根据这些的 KPI 预测进行有计划的实施资源调节，比如在特定时间段关闭一些闲置的小区，这样就可以在不影响终端用户体验的情况下显著降低基站能耗。除了节能检查之外，网络流量的预测可以指导自适应的 network slicing。正因为 KPI 预测有如此广泛的应用因为，在 Project Zouwu 中我们选择了使用网络流量预测作为典型案例给用户提供参考解决方案。我们选择了 WIDE 项目维护的公开数据集，其中收集的都是真实的网络流量数据，我们将总吞吐量和平均传输速率作为预测 KPI 目标，使用过去一周数据预测未来两个小时的 KPI。Project Zouwu 为同一个用例提供了两种不同的方案，一个模型直接使用了独立的 Forecaster 模型，另一个模型使用了有 AutoML 加持的 AutoTS 模块。两个部分提供了两个 notebook，展示两种 KPI 的预测情况。用户发现他们只需要对模型做少量改动就可以应用在自己的场景中。

Network Traffic KPI Forecasting in Telco

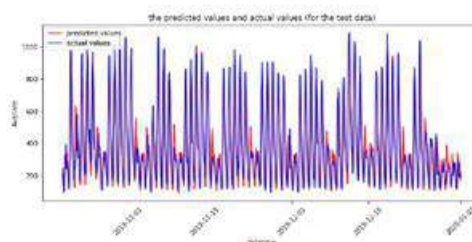


Usage Scenario

- KPI/metrics forecasting is widely used in Telco applications (e.g., energy saving, network slicing, etc.)
- aggregated traffic KPI's (i.e. total bytes, average rate in Mops/Gbps) in the past week to forecast the KPI in the next two hours.

2 ways to solve this problem using Zouwu

- Use built-in "Forecaster" models for training, and forecasting ([notebook link](#))
- Use "AutoTS" (with built-in AutoML support) to train an E2E Time Series Analysis Pipeline, and forecast ([notebook link](#))



Example result of network traffic average rate forecasting on the test period

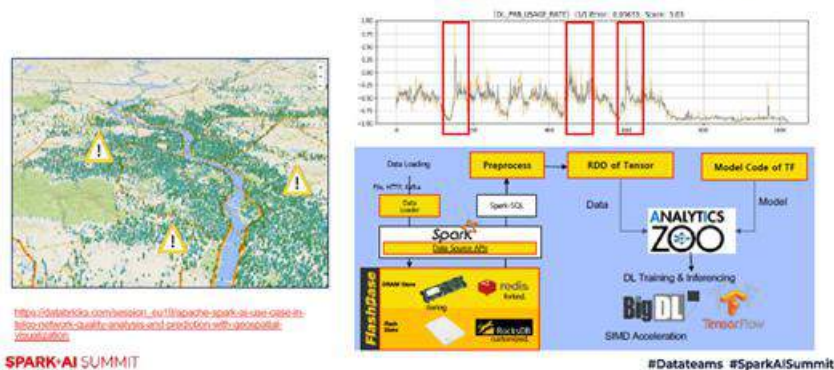
SPARK AI SUMMIT

#Datateams #SparkAISummit

SK Telecom

下图展示了与韩国电信所合作的项目，是 KPI 预测的又一个实践应用。韩国电信使用 KPI 预测检查小区当中基站的潜在异常情况，并且将有潜在异常的小区在 3D 地图中展示出来。这类异常检查是非常常见的需求，尽管在本案例中没有使用到前面所介绍的时序解决方案，但其它客户正在使用时序解决方案解决类似的网络异常检测问题。韩国电信在 2019 年的 Spark 峰会中有关于本案例的 talk，感兴趣的同学可以进一步了解。

Network Quality Prediction in SK Telecom



NeuSoft

NeuSoft 是一家 IT 解决方案提供商，他们有很多客户，包括宝马之类的企业。在 NeuSoft 所开发的智能运维系统中需要实时监控虚拟机，容器，服务等健康指标，对这些指标进行预测和异常检测。NeuSoft 使用了基于 AutoML 的时序预测方案，在很短很多时间之内就训练出除了达到准确度要求的模型。

Forecast-based Analysis for AIOps at NeuSoft



经验分享

通过与大量客户的合作经验和反馈，有以下几点需要与大家进行分享：

亮点：首先，我们基于 AutoML 的模型允许输入额外特征。很多传统模型不允许输入额外特征。但是 analytics-zoo 的基于 AutoML 的 pipeline 会获得自动生成额外的特征，如是否处于节假日，忙时还是闲时等。客户也有自己领域相关的特征。利用这些额外特征可以得到相对而言可以得到更优的模型效果。另外，采用 analytics-zoo 内置的模型和自动训练辅助，通过自动调优的策略，大部分用户可以在较很短的时间内得到比手动调优更高的满足准确度要求的模型。

数据质量：当然并不是所有案例在一开始都能获得很好的效果，其中但绝大部分都是由于数据质量不好所造成的。数据质量上的问题其中又是缺失数据最普遍，不同的填充方法对于预测的准确度影响可能很大。同一个数据集中都要采取不同的方法。

数据量大：很多客户在面临数据量大的问题，很多时候这并不是因为时间维度上样本数量大，而是有太多的时间序列需要预测。比如跟我们合作的一个客户有几十万的小区，每个小区有很多 KPI 需要预测，相当于一次要预测数百万的时序曲线。在云系统的智能运维场景下，百万级别的容器和服务，及上百个监控指标需要预测上千万的时序。在如此大规模的时序预测场景中，为每个时序做一个模型显然不是高效的做法。英特尔也正在尝试为客户解决此类问题，同时也取得了一些进展。

Takeaways from Early Users



Highlights

- Additional features allowed
- Less efforts in tuning
- Satisfactory accuracy

Data quality matters

- Missing values, outliers, etc.

Scale, scale, scale

- hundreds of thousands of cells X KPIs => millions of time series
- millions of servers/containers X metrics => hundreds of millions of time series

SPARK AI SUMMIT

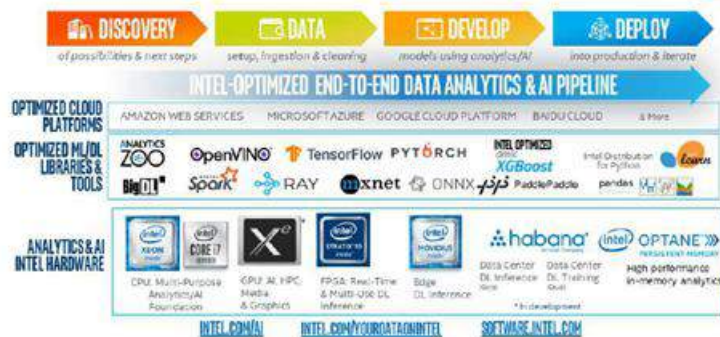
#Datateams #SparkAISummit

四、未来规划

英特尔正在为超高维度时间序列预测寻找更高效的解决方案。其次，还在计划改进搜索方法，如 meta-learning，集成学习等功能。同时结合用户的需求增加更多的机器学习模型和特征方面的处理功能。英特尔还在考虑添加自动化数据预处理功能，解决数据质量问题。

从下图可以发现，英特尔提供了一系列端到端的 AI 技术和产品，从硬件到软件，以及集成方案。结合不同需求提供多样化选择，感兴趣的同学可以重点关注下方三个链接获取更多的信息。

Accelerate Your Data Analytics & AI Journey with Intel



SPARK AI SUMMIT

#Datateams #SparkAISummit

Apache Spark 3.0 对 Prometheus 监控的原生支持

简介： 阿里云 EMR 技术专家周康为大家带来 Apache Spark 3.0 对 Prometheus 监控的原生支持的介绍。内容包括 spark 3.0 以前是怎么用 Prometheus 进行监控的，以及 spark 3.0 是如何实现对 Prometheus 更好的本地化的支持。

一、用 Prometheus 监控 Apache Spark

在使用 Apache Spark 去做 ETL，去做数据分析和处理的过程中，我们肯定都会涉及到监控 spark 程序这么一项工作。一般来说，有三种方式去做程序的监控。第一个就是使用 Web UI。第二块主要是日志。第三种是 Metrics。这三个信息，最大的一个问题是说，我们一般是在 ETL 夯住了或者失败之后，才会去查看。如果有这么一个系统能够帮助我们，在任务性能下降或者失败之前就提醒、甚至是自愈，这样就能避免很多故障的发生。



在分布式系统中，比较大的挑战之一就是如何做到在故障发生之前就能够有些规避的措施，或者一些提醒的措施。Metrics 在一个分布式系统里边是非常关键和核心的一个模块。在 spark 里边，同样也是有很多的 metrics。使用 metrics 我们能够发现内存泄露，一些配置错误，在性能下降之前的一些监控，以及对于流作业的一些监控等等。

Metrics are useful to handle gray failures

Early warning instead of post-mortem process

- Monitoring and alerting Spark jobs' gray failures
 - Memory Leak or misconfiguration
 - Performance degradation
 - Growing streaming job's inter-states

SPARK-AI SUMMIT #Datateams #SparkAISummit

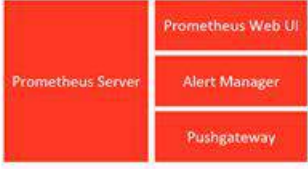
Prometheus 是非常有用的一个 metrics 监控的系统。它是一个开源的系统，能够支持监控和报警。它主要有以下四个特点。第一个特点是，它的数据模型是一个多维度的数据模型。第二个特点是，部署和运维是非常方便的。第三个特点是，在采集数据的扩展性这块，支持得非常好。最后一个特点是，提供了一个非常强大的查询语言。

Prometheus

An open-source systems monitoring and alerting toolkit

- Provides
 - a multi-dimensional data model
 - Operational simplicity
 - Scalable data collection
 - a powerful query language

A good option for Apache Spark Metrics



SPARK-AI SUMMIT #Datateams #SparkAISummit

在 spark 3.0 以前，我们一般怎么去支持 Prometheus。其中第一个方法就是基于 java agent 的方式。主要主要分为四个步骤，如下图所示。

Spark 2 with Prometheus (1/3)

Using JmxSink and JMXExporter combination

- Enable Spark's built-in JmxSink in Spark's conf/metrics.properties
- Deploy Prometheus' JMXExporter library and its config file
- Expose JMXExporter port,9404,to Prometheus
- Add '-javaagent' option to the target (master/worker/executor/driver/...)

```
-javaagent:./jmx_prometheus_javaagent-0.12.0.jar=9404:config.yaml
```

SPARK-AI SUMMIT

#Datateams #SparkAISummit

第二种方式，基于 GraphiteSink 和 GraphiteExporter 的组合。

Spark 2 with Prometheus (2/3)

Using GraphiteSink and GraphiteExporter combination

- Set up Graphite server
- Enable Spark's built-in Graphite Sink with several configurations
- Enable Prometheus's GraphiteExporter at Graphite

SPARK-AI SUMMIT

#Datateams #SparkAISummit

第三种方式，基于 Custom sink 和 Pushgateway server。

Spark 2 with Prometheus (3/3)

Custom sink (or 3rd party Sink) + Pushgateway server

- Set up Pushgateway server
- Develop a custom sink (or use 3rd party libs) with Prometheus dependency
- Deploy the sink libraries and its configuration file to the cluster

SPARK-AI SUMMIT

#Datateams #SparkAISummit

我们来看一下，前面三种方法有哪些好处和坏处。首先说好处，已经有很多用户通过三种方式实现了跟 Prometheus 的对接，是一个比较通用的方式。但是它也有坏处，在新环境中它有一些比较繁琐的 setup 的工作。另外，第三方插件可能对特定版本会有一些比较强的依赖，这对我们后续的运维和升级会带来困扰。

Pros and Cons

- Pros
 - Used already in production
 - A general approach
- Cons
 - Difficult to setup at new environments
 - Some custom libraries may have a dependency on Spark versions

SPARK-AI SUMMIT

#Datateams #SparkAISummit

所以在 spark 3.0 中，就提出了新的目标。主要有两个需要关注的设计点。一个就是只使用到新的 endpoint，不去跟原先的 pipeline 耦合，不引入对其他一些组件的依赖。另外一块就是尽量重新使用已经存在的一些资源，包括端口的资源，等等。

Goal in Apache Spark 3

Easy usage

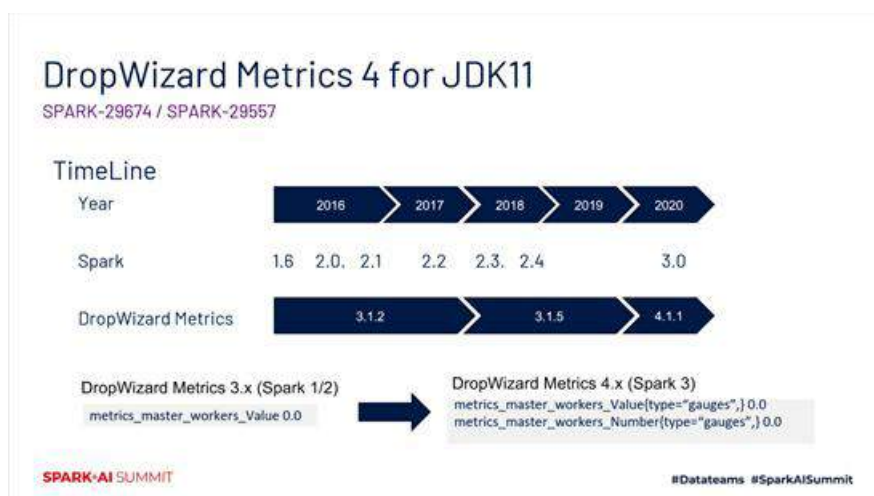
- Be independent from the existing Metrics pipeline
 - Use new endpoints and disable it by default
 - Avoid introducing new dependency
- Reuse the existing resources
 - Use official documented ports of Master/Worker/Driver
 - Take advantage of Prometheus Service Discovery in K8s as much as possible

SPARK-AI SUMMIT

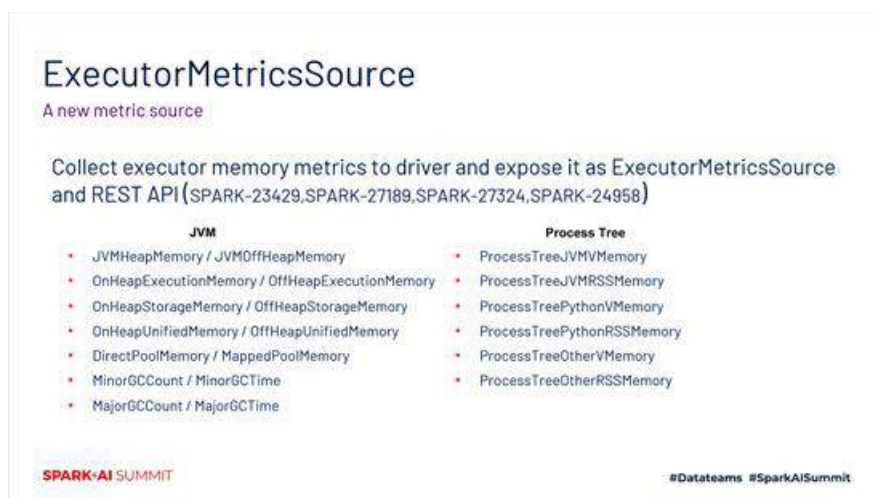
#Datateams #SparkAISummit

二、本地化支持 Prometheus (Support Prometheus monitoring natively)

接下来，我会介绍一下在 spark 3.0 中如何实现刚才提到的目标，更好的本地化支持 Prometheus。Spark 的 metrics 模块依赖比较重的是 DropWizard 这么一个组件。它在 spark 3.0 中也做了一次升级，带来的好处是能够支持 jdk11，但是也存在小的负面点，就是它的数据格式有所变化，如下图所示。



第二个增加的组件就是 `ExecutorMetricsSource`。主要是对 `executor memory` 相关的一些 `metrics` 做了增强。



为了更好的做一些本地化支持的工作，主要加了两个组建。一个是 `PrometheusServlet`，它会生成 `Prometheus` 兼容的一个格式，它的配置方式跟以前的 `metrics system` 配置是一样的。同时也不会引入其他的一些库，也不会对特定端口进行依赖。第二个组件是 `PrometheusResource`。针对 `executor memory metrics` 相关的信息，提供了这么一个独立的 `endpoint`。

Support Prometheus more natively (1/2)

Prometheus-format endpoints

- **PrometheusServlet: A friend of MetricServlet**
 - A new metric sink supporting Prometheus-format (SPARK-29032)
 - Unified way of configurations via `conf/metrics.properties`
 - No additional system requirements (services/libraries/ports)
- **PrometheusResource: A single endpoint for all executor memory metrics**
 - A new metric endpoint to export **all executor metrics** at driver (SPARK-29064/SPARK-29400)
 - The most efficient way to discover and collect because driver has all information already
 - Enabled by `'spark.ui.prometheus.enabled'` (default: false)

SPARK-AI SUMMIT #Datateams #SparkAISummit

在 3.0 中增加了 `spark_info` metric，它的作用是针对不同版本的监控和报警。另外，对 `driver service annotation` 做了一些增强，能够实现利用 Prometheus service discovery 的特性，从而更方便的进行一些 metrics 的采集和监控。

Support Prometheus more natively (2/2)

`spark_info` and service discovery

- **Add `spark_info` metric (SPARK-31743)**
 - A standard Prometheus way to expose version and revision
 - Monitoring Spark jobs per version
- **Support driver service annotation in K8S (SPARK-31696)**
 - Used by Prometheus service discovery

SPARK-AI SUMMIT #Datateams #SparkAISummit

三、在 K8s 集群中监控 (Monitoring in K8s cluster)

在 K8s 环境下面，怎么去基于已经集成 Prometheus 的 spark 版本，对我们的作业进行监控，主要有三个场景。第一个是对一些批作业的内存的情况进行监控。第二个是对动态调度和动态资源分配的监控。第三个是对流作业的监控。

Key Monitoring Scenarios on K8s clusters

- Monitoring **batch job memory** behavior

Monitoring **dynamic allocation** behavior

Monitoring **streaming job** behavior
- => A risk to be killed?

=> Unexpected slowness?

=> Latency?

SPARK-AI SUMMIT

#Datateams #SparkAISummit

我们来看一下这三种场景。首先，对批作业内存的监控场景，主要用到的是 Prometheus 的 service discovery 这一个特性。如下图所示，有四个配置。

Monitoring batch job memory behavior (1/2)

Use Prometheus Service Discovery

Configuration	Port
spark.ui.prometheus.enabled	true
spark.kubernetes.driver.annotation.prometheus.io/scrape	true
spark.kubernetes.driver.annotation.prometheus.io/path	/metrics/executors/prometheus
spark.kubernetes.driver.annotation.prometheus.io/port	4040

SPARK-AI SUMMIT

#Datateams #SparkAISummit

我们来看一个简单的例子，是基于 SparkPi 的。假设我们现在通过 spark-submit 提交一个作业到 K8s。主要是把前面提到的几个配置，也就是下图这个例子中加粗的部分给配置上就可以了。

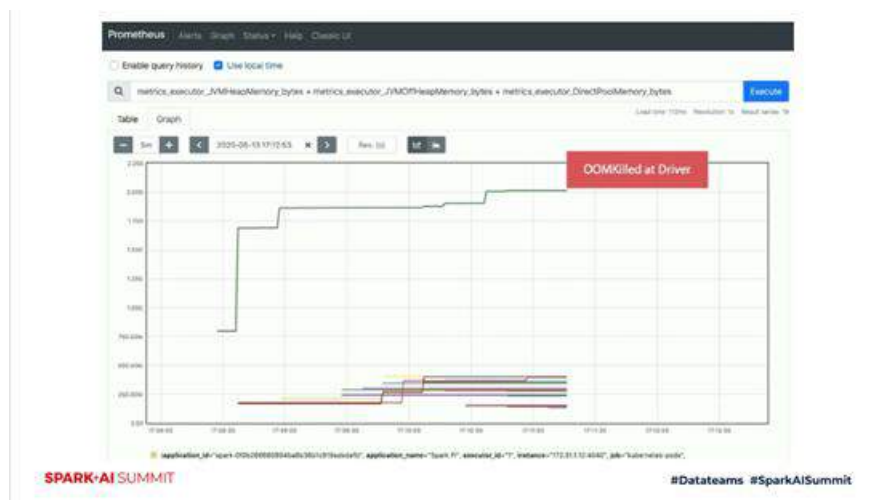
Monitoring batch job memory behavior (2/2)

```
spark-submit --master k8s://$K8S_MASTER_URL --deploy-mode cluster \
-c spark.driver.memory=2g \
-c spark.executor.instances=30 \
-c spark.ui.prometheus.enabled=true \
-c spark.kubernetes.driver.annotation.prometheus.io/scrape=true \
-c spark.kubernetes.driver.annotation.prometheus.io/path=/metrics/executors/prometheus \
-c spark.kubernetes.driver.annotation.prometheus.io/port=4040 \
-c spark.kubernetes.container.image=$spark_image \
--class org.apache.spark.examples.SparkPi \
local:///path/to/spark-examples_xx.jar 200000
```

SPARK-AI SUMMIT

#Datateams #SparkAISummit

如下图所示，是一个针对内存的监控。它是 `executor_id` 为 1 的时候，对它的内存使用的监控。可以看到绿色的这条线，我们从 `driver` 端拿到这个状态之后，就可以提前知道已经有 OOM 的风险，通过 Prometheus 从 `driver` 端采集到监控信息，就可以及时的做一些报警和预处理。



第二个场景就是动态调度。Spark 3.0 支持在 K8s 上动态调度。主要就是把一些配置打开，如下图中的黑体部分所示。配置打开之后，在作业提交的时候，就能够开启在 K8s 环境的动态调度。

Monitoring streaming job behavior (1/2)

Set `spark.sql.streaming.metricsEnabled=true` (default: false)

- **Metrics**
 - Latency
 - inputRate-total
 - processingRate-total
 - states-rowsTotal
 - states-usedBytes
 - eventTime-watermark
- **Prefix of streaming query metric names**
 - `metrics_[namespace]_spark_streaming_[queryName]`

SPARK-AI SUMMIT

#Datateams #SparkAISummit

前面提到的 6 种 metrics 对于流作业都是比较关键的，都应该去监控并且做一些报警处理。这里重点再提两个。一个就是前面提到的延时，如果 `latency > micro-batch interval`，就可能存在一些性能问题，需要关注。另外一个报警就是 `States-rowsTotal`。如果它无限的增长，可能会导致流作业发生 OOM。这个也是需要做好监控和报警的。

Monitoring streaming job behavior (2/2)

All metrics are important for alert

- **latency > micro-batch interval**
 - Spark can endure some situations, but the job needs to be re-design to prevent future outage
- **States-rowsTotal grows indefinitely**
 - These jobs will die eventually due to OOM
 - SPARK-27340 Alias on TimeWindow expression cause watermark metadata lost (Fixed at 3.0)
 - SPARK-30553 Fix structured-streaming java example error

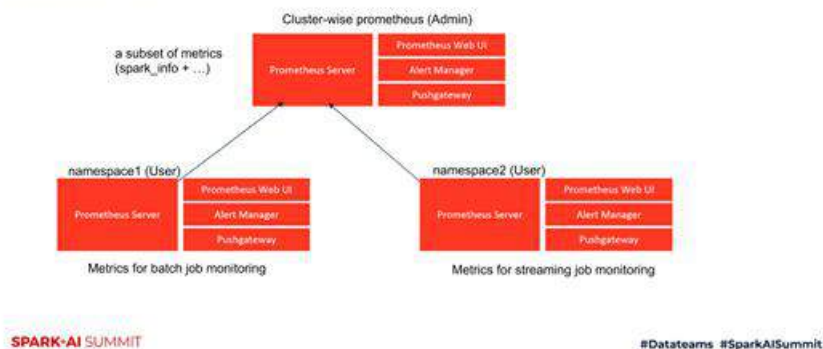
SPARK-AI SUMMIT

#Datateams #SparkAISummit

在针对 Prometheus 的使用方式这块，社区在 3.0 建议的是基于 federation 的模式。如下图所示，左边是在 namespace1 下面对批作业的一个监控。右边是在 namespace2 下面对流作业的一个监控。他们都可以同时发送到一个 Cluster-wise Prometheus 下面，然后通过 `spark_info` 等等这些信息做一些数据的细分。

Prometheus Federation and Alert

Separation of concerns



以上就是 spark 3.0 对于 Prometheus 监控的一些支持。目前来说还是处在一个实验性的阶段，所以还存在一些问题。第一个问题就是新的 endpoint 暴露的 metrics 是只包含 metrics_ 或者 spark_info 开头的这么一些信息。第二个问题是，PrometheusServlet 目前的命名格式没有遵循 Prometheus 的命名风格。第三个问题是，在流作业这块，如果没有把 namespace 做一些很好的配置，可能会导致 metrics 不断的增加，甚至互相影响。

Limitations and Tips

New endpoints are experimental

- New endpoints expose only Spark metrics starting with `metrics_` or `spark_info`
 - `javaagent` method can expose more metrics like `jvm_info`
- PrometheusServlet does not follow Prometheus naming convention
 - Instead, it's designed to follow Spark 2 naming convention for consistency in Spark
- The number of metrics grows if we don't set the followings

```
spark.metrics.namespace=spark
```

```
writeStream.queryName("spark")
```

SPARK-AI SUMMIT

#Datateams #SparkAISummit

阿里云 EMR 团队在做的 spark on k8s，目前已经对 Prometheus 监控进行了原生化的支持。目前我们实现的方式是基于 javaagent，后续也会计划去引进更多的原生支持。

Spark on K8s: E-MapReduce - Alibaba Cloud

Prometheus monitoring has been supported

- Currently we use 'javaagent' to integrate with Prometheus
- We are planning to integrate with more natively support for Prometheus

阿里云开源大数据平台实践

助力云上开源生态 – 阿里云开源大数据平台的发展

简介： 阿里云 E-MapReduce (EMR) 是构建在阿里云云服务器 ECS 上的开源 Hadoop、Spark、HBase、Hive、Flink 生态大数据 PaaS 产品。提供用户在云上使用开源技术建设数据仓库、离线批处理、在线流式处理、即时查询、机器学习等场景下的大数据解决方案。在 2019 杭州云栖大会大数据生态专场上，阿里巴巴高级产品专家夏立为大家分享了阿里云 EMR 如何助力云上开源生态。

本次分享的内容主要分为四个部分：

1. 发展历程
2. 云上现状
3. 云上开源生态的最佳实践
4. 开源大数据平台的发展展望

一、发展历程

在 2015 年，阿里巴巴刚开始做开源大数据平台的时候，摆在面前的有三种选择，分别是使用开源的 Hadoop 体系、CDH 和 HDP，以及当时的 ODPS（现在的 MaxCompute）。在那个时候，在大洋彼岸的 AWS 有一款大数据产品叫做 EMR，因此阿里云当时也希望借鉴 AWS 的经验来做开源大数据平台，希望将大数据能力和云原生能力进行深度结合。



阿里云在 2015 年 6 月份的时候就开始研发自己的开源大数据平台并实现了第一个“镜像+脚本”的版本，这个版本可以实现在最短的时间内将 Spark 环境搭建起来，而且这个版本很快上线并且发布到 GitHub 上。当时使用的 API 很像现在的编排服务，但是这样服务的缺点是只能一次性搭建，因此维护起来非常麻烦。



在 2015 年 11 月份，阿里云正式将 E-MapReduce 独立云产品推出市场。大家都知道 MapReduce 的思想来自于谷歌，其代表了大数据理论，因此阿里巴巴将这款大数据产品命名为 E-MapReduce，使得大家看到名字就知道其主要作用。



阿里云 E-MapReduce 上线之后经过四年的时间发展到现在，E-MapReduce 4.0 即将发布版本，并且在新版本中将会支持 Hadoop 3.0 以及其他新功能。

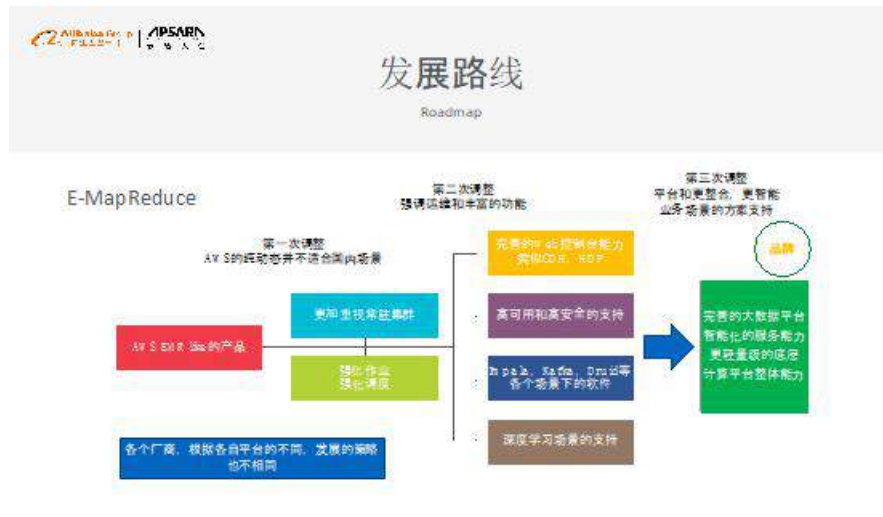


发展到今天，阿里云 E-MapReduce 将会为开源生态提供基础平台，在这个平台上能够让大家选择各种各样的开源产品，并且不会限制大家使用自定义的能力。此外，阿里云 E-MapReduce 也希望能够将阿里巴巴整个云智能平台的计算能力输出给大家，为大家提供云原生能力和弹性调度能力。未来，E-MapReduce 会陆续集成各种开源技术和能力，并且会在这些开源技术之上为大家提供更好的优化的技术，一方面提高稳定性，另外一方面也提升性能，并且也会进行能力的适配。最后一点就是实现云原生的结合，大家使用开源或者自建的大数据方案往往难以和云原生技术或者基础设施进行结合或者难以获得较高的性能，因此阿里巴巴希望通过 E-MapReduce 能够更好地和云原生技术进行结合。



总结阿里巴巴 E-MapReduce 的发展历程，最开始就是实现了一个 AWS EMR Like 的产品，运行一年之后发现 AWS 的纯动态方式并不适合国内的场景，因此实现了第一次调整，更加重视常驻集群，并且强化作业调度能力。经过第一次调整之后，阿里云发现 E-MapReduce 的能力还是无法满足需求，因此在第二次调整中提供了完善的 Web 控制台能力，并且支持了集群的高可用和高安全，也在外围支持了 Impala、Kafka、Druid 等各个场景下的软件，进而可以更好地支持各个业务场景。除此之外，还支持了深度学习的场景，将经过阿里巴巴自身优化的机器学习算法提供到平台之上。如今，

E-MapReduce 仍在继续调整，希望能够提供更加完善的大数据平台，更加智能化的服务能力，并且使得底层更加轻量，也使得计算平台整体能力能够对外输出出去。



二、云上现状

云上的生态概览

下图展示了阿里云的大数据生态概览。在数据来源方面，开源方面有 HDFS 和 Kafka，阿里巴巴则提供了 OSS、SLS、RDS 以及消息队列等服务。所有数据可以通过开源的 Hive、Spark、Flink、Rresto、TensorFlow 以及阿里巴巴的 MaxCompute、Flink/TensorFlow 等服务进行计算，并且还可以与阿里的自身体系如 DataWorks、DataV 以及 QuickBI 进行融合。目前，云上大数据方案可以认为是半托管的服务，阿里云能够帮助客户进行运维并且提供运维支撑服务。



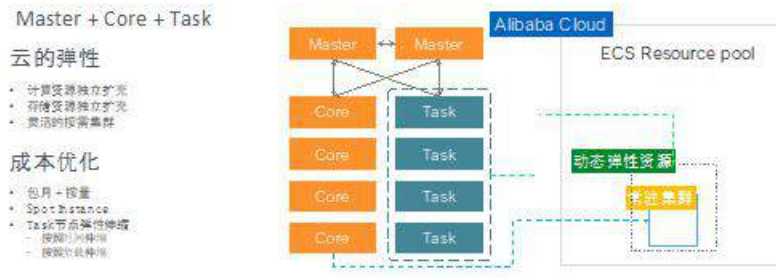
多样的存储选择

在阿里云上，大数据存储主要有三种选择，分别为 Hadoop HDFS、Alibaba HDFS 和 OSS。Hadoop HDFS 有三种存储方式，EBS 云盘存储数据可靠，但是后台有多个数据副本，因此成本较高，同时通过网络获取数据性能较低；D1 本地磁盘以及 I1/I2 本地词盘性能比较高，成本也比较低，但是数据容易丢失，并且运维成本较高。另外一种选择是 Alibaba HDFS，这种方式数据可靠，成本中等，并且数据全部通过网络传输，没有本地计算。OSS 标准存储经过阿里巴巴的改造和优化之后可以直接在 Hadoop 中进行读写，这就是所谓的 NativeOSS，NativeOSS 存储数据可靠，成本较低，并且通用性比较好，但是性能比较低。因此，进一步在 NativeOSS 上进行了强化，实现了 JindoFS，JindoFS 做到了数据可靠，成本较低，性能高并且通用性较好，但是需要额外的存储成本。



弹性实践

云上做计算需要充分发挥出弹性能力，否则就无法发挥出云的真正价值。而为了发挥出云弹性能力，各大云产商的所有大数据能力都会有 Master 节点以及一组工作节点 Task，Task 节点只进行计算但是不会进行数据存储，因此在云上执行计算任务时，Task 节点可以进行弹性伸缩，还可以通过 Stop Instance 来降低成本。在计算任务的高峰期购买 Task 节点，当高峰期过去之后，就可以释放 Task 节点。阿里云还为客户提供了一套伸缩机制，既可以按照时间伸缩，也可以按照负载伸缩。



集群架构

下图展示的是阿里云比较推荐的云上集群架构。如图中左侧所示的是建立的 Hadoop 集群，其底层全部使用 OSS 做数据存储，而在 OSS 之上存在几个独立的计算集群，比如 Hive、Spark 以及 Presto 等，而且这些集群全部都是灵活可销毁的。右侧同样建立 Hadoop 集群，而外侧则提供了 Gateway 以及 Client 来接受请求。此外，很多客户可能在云上没有使用 OSS 或者混合使用 OSS 和 HDFS，借助这种集群架构，可以帮助用户跨越数据存储的障碍。



三、云上开源生态的最佳实践

存储的选择和优化

在 2015 年的时候，想要在阿里云上部署大数据平台只有云盘存储可以选择，比如使用 SSD 等高效存储盘，因此这样的成本会非常高。到 2017 年左右，阿里云智能团队和 ECS 团队合作做了本地盘机型，

后来还和 ECS 团队合作做了 D1，并且适配了一些国内更能够适应的场景。在 2016 年，E-MapReduce 实现了和 OSS 的结合，当时因为带宽限制，因此使用的客户较少。到如今，针对之前的发展和合作经验，E-MapReduce 可以选择使用 JindoFS、Alibaba HDFS 等进行存储。



IaaS 层升级

为了让客户更好地使用 E-MapReduce，IaaS 层也经历了多次升级。第一代是 D1 和 I1，第二代是 D2 和 I2，提供了更高的网络带宽，本地磁盘提供了极高的性能，但同时也带来了运维成本的增加。而通过磁盘的热更换，提供了更好的体验整套的运维的支持链路，并且提供了整套硬件的监测、预警、通知、更换等操作完成主动运维流程。



存储访问优化方案 JindoFS

JindoFS 的目的在于让大家更好地使用存储与计算分离的架构。在 JindoFS 的架构之下，所有数据都会存储在 OSS 上面，所有的计算都放在动态集群上面进行，并且可以随时进行计算伸缩，JindoFS 为客户提供了高性能的数据存取能力，和高性价比、无限扩展的弹性存储能力。这里面临的最大的挑战就是 OSS 和计算集群之间的网络带宽，而 JindoFS 方案中通过本地缓存技术大大降低了时延，提升了性

能效率。同时，因为 JindoFS 采用了基于存储计算分离的架构，因此客户不用担心缓存数据的丢失问题。



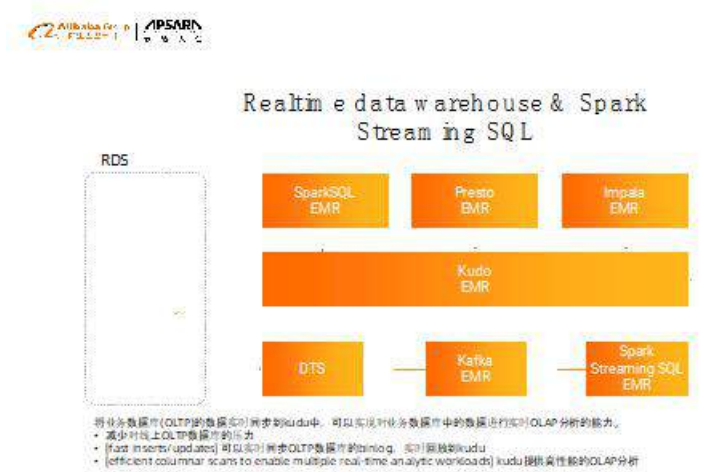
更多的产品的融合和增强

阿里云 E-MapReduce 融合了更多的产品，比如 Spark、Flink、TensorFlow、Elasticsearch、Dataworks 等，并在这些产品的基础之上做了增强。

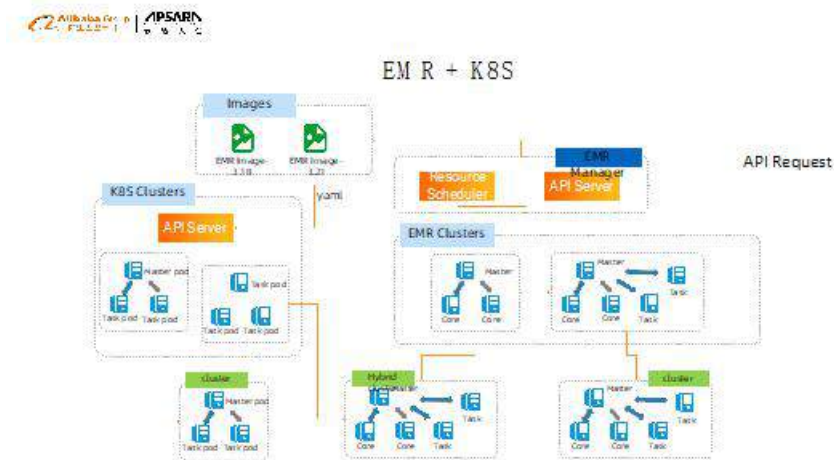


四、开源大数据平台的发展展望

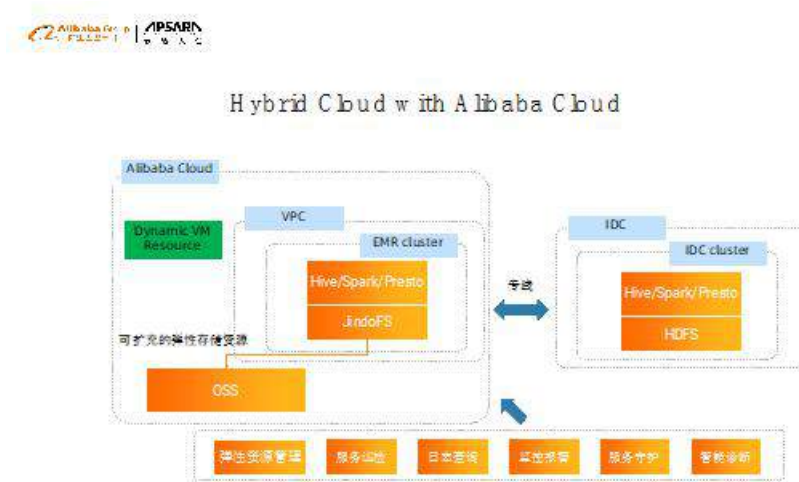
阿里云 E-MapReduce 希望基于平台实现更多的方案，希望能够更好地赋能客户的业务场景。比如在实时数仓方案和 Spark Streaming SQL 中，实现了将业务数据库的数据实时同步到 kudo 中，可以实现对业务数据库中的数据进行实时 OLAP 分析的能力。



未来, EMR 将会实现与 K8S 的融合, 希望能够帮助客户更好地节约成本, 让用户在阿里云内部可以腾挪自己的资源来完成各种工作, 支持客户在业务低峰时将 K8S 节点加入到 Hadoop 节点中作为计算补充, 在业务高峰时将集群还给业务, 这样在不增加额外成本的情况下增加计算能力, 更加充分地利用资源。



很多的用户希望实现多云和混合云，因此阿里巴巴希望为客户提供在线下 IDC 用法不变的情况下，将冷数据通过专线传输到动态存储，并且使用 E-MapReduce 等进行动态赋能和线下集群结合起来使用的能力，并且与此同时充分利用线下和线上的能力。









EMR Spark-SQL 性能极致优化揭秘 概览篇

简介：这次的优化里面，还有一个很好玩的优化，就是我们引入的 Native Runtime，如果说上述的优化器优化都是一些特殊 Case 的杀手锏，Native Runtime 就是一个广谱大杀器，根据我们后期统计，引入 Native Runtime，可以普适性的提高 SQL Query 15~20% 的 E2E 耗时，这个在 TPCDS Perf 里面也是一个很大的性能提升点。

作者：林学维，阿里云智能 EMR 团队技术专家，目前主要专注于 EMR 产品中开源计算引擎的优化工作

引子

最近阿里云 E-MapReduce 团队在 TPCDS-Perf 榜单中提交了最新成绩，相比第二名(其实也是 EMR 团队于 2019 年提交的记录)，无论从性能还有性价比都取得了 2 倍+的优秀成绩！详细看 [TPCDS Perf](#)

Rank	Company	System	QphDS	Price/QphDS	Watts/QphDS	System Availability	Database	Operating System	Date Submitted	Cluster
1		Alibaba Cloud E-MapReduce	11,569,838	.24 CNY	NR	04/17/20	Alibaba Cloud E-MapReduce 4.0.1	CentOS Linux Release 7.4	04/16/20	Y
2		Alibaba Cloud E-MapReduce	5,261,414	.53 CNY	NR	09/16/19	Alibaba Cloud E-MapReduce 3.21.2	CentOS Linux Release 7.4	09/16/19	Y
3		Supermicro A+ Server 2123BT-HNCR	4,418,054	.12 USD	NR	08/31/19	Transwarp ArgoDB V1.2.1	Red Hat Enterprise Linux Server 7.6	08/07/19	Y
4		Alibaba Cloud AnalyticDB	2,684,357	1.19 CNY	NR	04/26/19	Alibaba Cloud AnalyticDB 2.7	Alibaba Group Enterprise Linux Server 7.2 (Paelelin)	04/26/19	Y
5		Alibaba Cloud E-MapReduce	1,824,283	.31 USD	NR	03/20/19	Alibaba Cloud E-MapReduce 3.16.1	CentOS Linux Release 7.4	03/19/19	Y
6		Cisco UCS Integrated Infrastructure for Big Data	1,580,649	.64 USD	NR	03/05/18	Transwarp Data Hub v5.1	Red Hat Enterprise Linux Server 6.7	03/05/18	Y

阿里云 E-MapReduce 团队，除了在产品、易用性、安全性等维度上投入了大量的研发资源和精力，打造了 EMR 这样一个广受好评的大数据产品；在引擎层面上也长期投入，持续深耕，目的就是要在保持开源软件的 100% 兼容性的同时，要利用团队的技术深度去打造产品的技术壁垒，让客户在使用开源软件栈的时候，能够获得更多的性价比，真真切切的把云上成本降低到极致，让客户能够在上云的过程中没有疑虑和后顾之忧。

阿里云 E-MapReduce 团队在 TPCDS Perf 中取得的成绩也足以验证，团队在 SPARK 引擎的技术深度以及技术实力，接下来会有一个系列的文章，去介绍我们 2020 年度打榜过程的一些优化点还有思考，欢迎社区里的 spark 引擎开发者或者 spark 应用开发者可以关注我们的系列文章，也欢迎来和我们交流，最关键的是，欢迎多投简历，加入阿里云 E-MapReduce 团队，我们求贤若渴！！

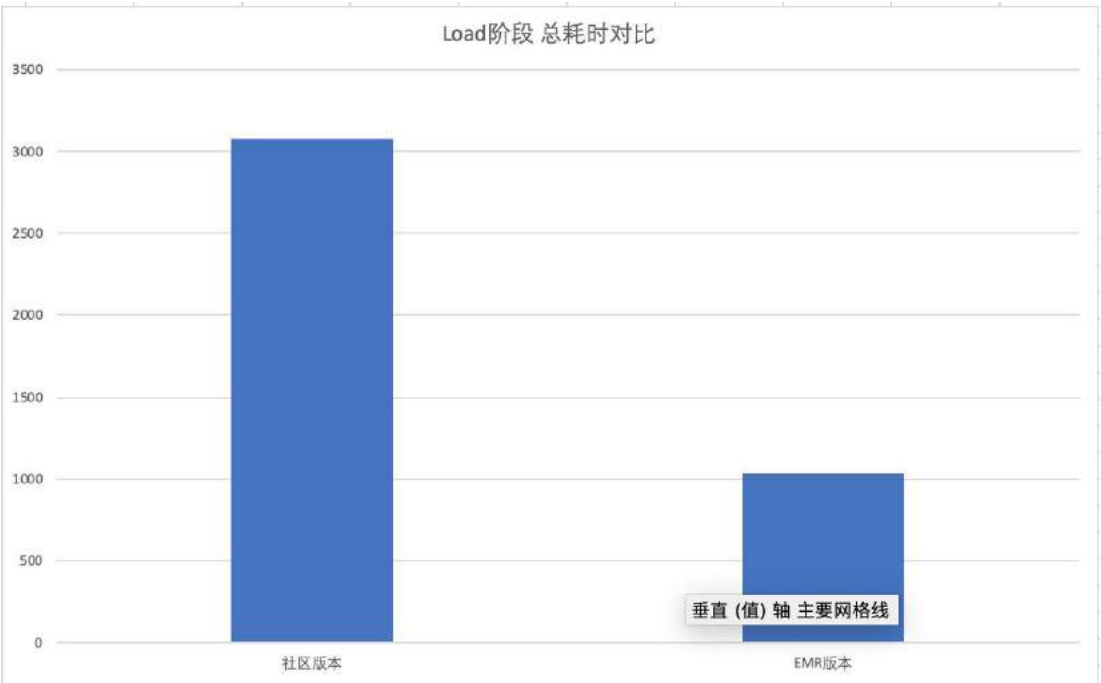
第三次刷榜的 Flag

从上述的 TPCDS Perf 链接中，我们可以看到，其实 EMR 团队在 10TB 规模总共提交了三次成绩。第三次也就是这一次打榜，背后还有一个小故事。因为在 Perf 页面中，最终 TPCDS 关注的指标有两个，一个是性能指标一个是性价比指标。这次项目立项的时候，我们就给自己立下了一个艰难的 Flag，我们要在物理硬件保持不变的条件下，纯靠软件优化提升 2 倍+，这样子性能指标和性价比指标就都能翻倍了。

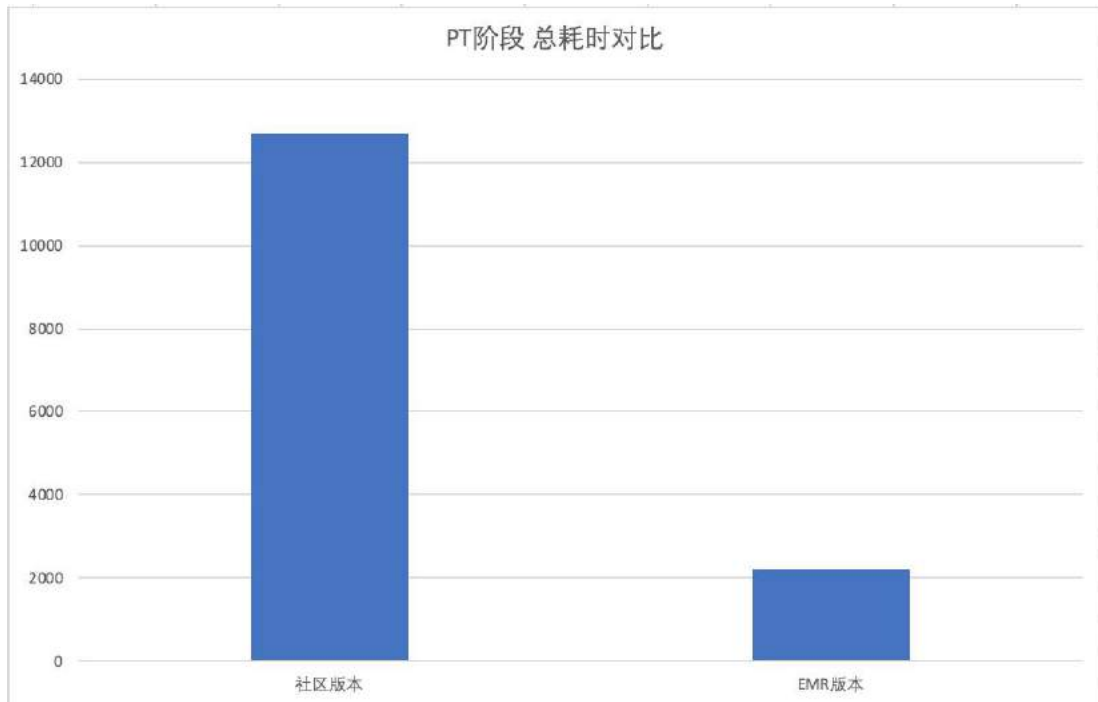
与开源 Spark 版本的一些对比数据

在提交完成绩后，我们用开源 Spark V2.4.3 版本进行了 TPCDS 99 Query 测试，以下是性能数据对比

Load 阶段性能提升约 3 X

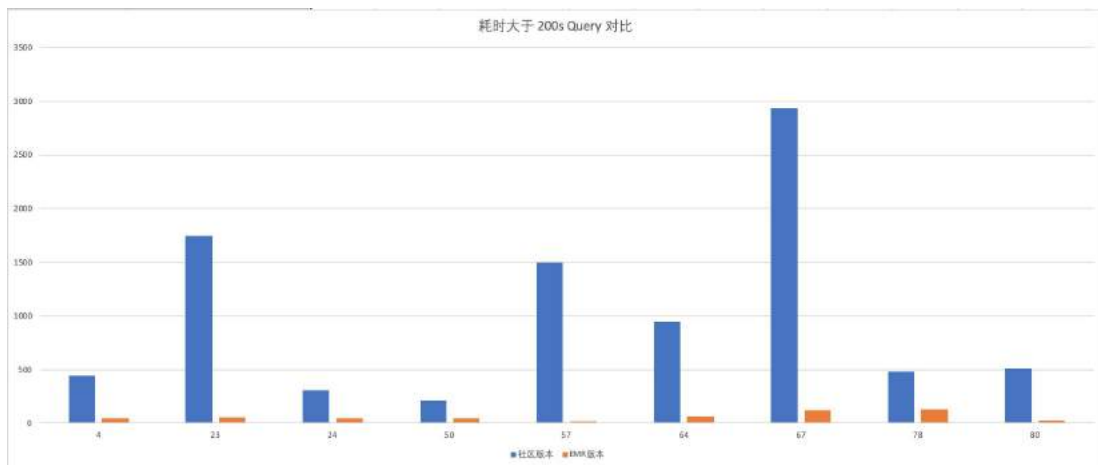


PT 阶段性能提升约 6 X



PS. 其中社区 Spark V2.4.3 版本中 Query 14 以及 Query 95 因为 OOM 的原因没法跑出来，不纳入计算

社区 Spark 版本运行时间大于 200s 的 Query 单独拿出来对比



PS. 这几个 Query 最低的 Query 78 有 3X 性能提升，Query 57 有接近 100 倍的性能提升。

优化点概述

优化器

- 基于 InMemoryTable Cache 的 CTE 物化

简单来说，就是尽量更合理的利用 InMemoryTable Cache 去减少不必要的重复计算，比如说 Query 23A/B 中的标量计算，本身是非常重的操作，并且又必须重复的计算，通过 CTE 优化的模式匹配，识别出需要重复计算且比较耗时的操作，并利用 InMemoryTable 缓存，整体减少 E2E 时间

- 更加有效的 Filter 相关优化

- Dynamic Partition Pruning

这个在社区最新的 3.0 版本才有这个功能

- 小表广播复用

一个具有过滤性的小表，如果可以过滤 2 个或以上的打表数据时，可以复用该小表的过滤效果 Query 64 就是一个好例子

- BloomFilter before SMJ

在 SMJ 真正实施之前，通过前置 BloomFilter，Join 过程的数据进一步减少，最大限度的消除 SpillDisk 的问题

- PK/FK Constraint 优化

通过主键外键信息，对优化器提供更多的优化建议

- RI-Join

去除事实表与维表于主键外键上做 Join，但是维表的列并没有被 Project 的情况下，这次 Join 其实完全没有必要执行

- GroupBy Keys 去除非主键列

当 GroupBy Keys 中同时包括主键列以及非主键列，其实非主键列对 GroupBy 结果已经没有影响了，因为主键列已经隐含了 Unique 的信息

- GroupBy Push Down before

- Fast Decimal

基于 Table Analyze 以及运行时中的 Stat 信息，优化器可以决定把某些 Decimal 优化为 Long 或者 Int 的计算，这会有极大的提升，而 TPCDS 99 Query 里有大量的 Decimal 计算

运行时

这次的优化里面，还有一个很好玩的优化，就是我们引入的 Native Runtime，如果说上述的优化器优化都是一些特殊 Case 的杀手锏，Native Runtime 就是一个广谱大杀器，根据我们后期统计，引入 Native Runtime，可以普适性的提高 SQL Query 15~20% 的 E2E 耗时，这个在 TPCDS Perf 里面也是一个很大的性能提升点。

大致的介绍一下 Native Runtime

基于开源版本的 WholeStageCodeGeneration 的框架，在原有的生成的 Java 代码，替换成 Weld IR 来真实运行。Weld 详细参考 <http://weld.stanford.edu/>。在整个项目里，Weld IR 的替换其实是非常小的一部分工作，为了 Weld IR 能够运行起来，我们还需要做以下的工作

- Expression Weld IR CodeGen (TPCDS 范围内全支持)
- Operators Weld IR CodeGen (除了 SortMergeJoin 用 C++ 实现，其他均可以用 Weld IR 代替)
- 统一内存布局 (OffHeap UnsafeRow => C++ & Weld Runtime)
- Batch 化执行框架 (因为如果按照 Java 运行时，每次都是一条记录的在生成代码里流转，在 NativeRuntime 的时间里代价太高，JNI 以及 WeldRuntime 明显不能这么玩)
- 其他高性能 Native 算子 SortMergeJoin、PartitionBy、CSV Parsing，这几个算子目前用 Weld IR 提供的接口无法直接实现，我们通过 C++ 来实现这些算子的 Native 执行

结语

这个文章只是大概的介绍了这次性能优化的一些优化点，在接下来的系列文章里，我们会针对每一个优化点细致的展开、分析，希望对 Spark-SQL 有兴趣的同学们可以多多关注，多多捧场。同时，我们也希望对 EMR 团队有兴趣的同学，积极联系我们，我们真的求贤若渴，海量 HC，请有兴趣者联系 林学维(峰七) 18518298234，也可邮箱 xuwei.linuxuewei@alibaba-inc.com !!!

推荐阅读

[EMR Spark-SQL 性能极致优化揭秘 RuntimeFilter Plus](#)

EMR Spark-SQL 性能极致优化揭秘

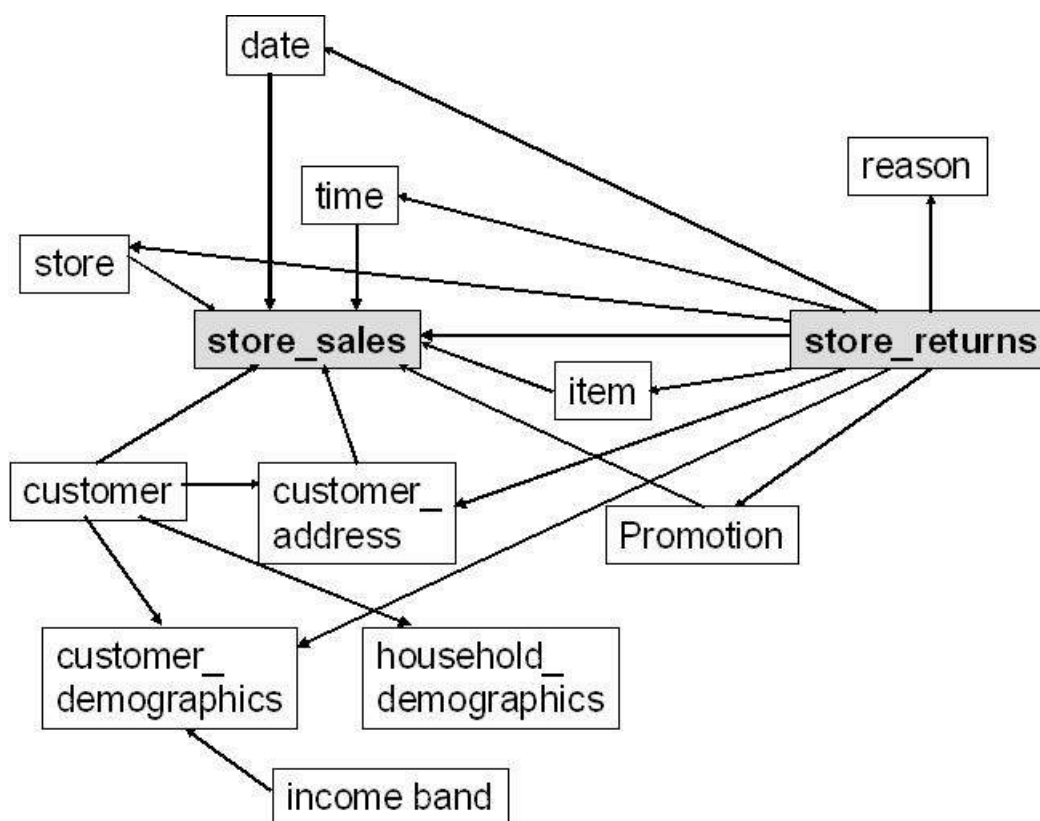
RuntimeFilter Plus

简介： 在 2019 年的打榜测试中，我们基于 Spark SQL Catalyst Optimizer 开发的 RuntimeFilter 优化 对于 10TB 数据 99 query 的整体性能达到 35% 左右的提升。

作者：陆路，花名世仪，阿里巴巴计算平台事业部 EMR 团队高级开发工程师，大数据领域技术爱好者，对 Spark、Hive 等有浓厚兴趣和一定的了解，目前主要专注于 EMR 产品中开源计算引擎的优化工作。

背景介绍

TPC-DS 测试集采用星型和雪花型等多维数据模型，包含 7 张事实表和 17 张维度表，以 store channel 为例，事实表和维度表的关联关系如下所示：



分析 TPC-DS 全部 99 个查询语句不难发现，绝大部分语句的过滤条件都不是直接作用于事实表，而是通过过滤维度表并将结果集与事实表 join 来间接完成。因此，优化器很难直接利用事实表索引

来减少数据扫描量。如何利用好查询执行时的维度表过滤信息，并将这些信息下推至存储层来完成事实表的过滤，对于性能提升至关重要。

在 2019 年的打榜测试中，我们基于 Spark SQL Catalyst Optimizer 开发的 RuntimeFilter 优化对于 10TB 数据 99 query 的整体性能达到 35% 左右的提升。简单来说，RuntimeFilter 包括两点核心优化：

- 动态分区裁剪：事实表以日期列（date_sk）为分区列建表，当事实表与 date_dim 表 join 时，optimizer 在运行时收集 date_dim 过滤结果集的所有 date_sk 取值，并在扫描事实表前过滤掉所有未命中的分区文件。
- 非分区列动态过滤：当事实表与维度表的 join 列为非分区列时，optimizer 动态构建和收集维度表结果集中 join 列的 Min-Max Range 或 BloomFilter，并在扫描事实表时下推至存储层，利用存储层索引（如 Parquet、ORCFile 的 zone map 索引）来减少扫描数据量。

问题分析

为了进一步挖掘 RuntimeFilter 优化的潜力，我们选取了部分执行时间较长的 query 进行了细致的性能剖析。这些 query 均包含大于一个事实表和多个维度表的复杂 join。在分析了 RuntimeFilter 对各个 query 的性能提升效果后，我们发现：

- 动态分区裁剪的性能提升效果明显，但很难有进一步的优化空间
- 非分区列动态过滤对整体提升贡献相比分区裁剪小很多，主要是因为很多下推至存储层的过滤条件并没有达到索引扫描的效果

聪明的同学应该已经发现，只有 date_dim 这张维度表和分区列相关，那么所有与其它维度表的 join 查询从 RuntimeFilter 优化中受益都较为有限。对于这种情况，我们做了进一步的拆解分析：

- 绝大部分 join 列均为维度表的自增主键，且与过滤条件没有相关性，因此结果集取值常常均匀稀疏地散布在该列的整个取值空间中
- 对于事实表，考虑最常见的 Zone Map 索引方式，由于 load 阶段没有针对非分区列做任何聚集操作（Clustering），每个 zone 的取值一般也稀疏分散在各个列的值域中。
- 相比 BloomFilter，Min-Max Range 的构建开销和索引查询开销要低得多，但由于信息粒度太粗，索引过滤命中的效果也会差很多

综合以上几点考虑，一种可能的优化方向是在 load 阶段按照 join 列对事实表进行 Z-Order 排序。但是这种方式会显著增加 load 阶段执行时间，有可能导致 TPC-DS 评测总分反而下降。同时，由于建表阶段优化的复杂性，实际生产环境的推广使用也会比较受限。

RuntimeFilter Plus

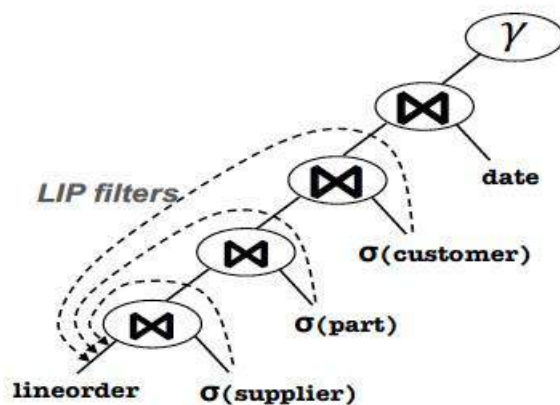
基于上述分析，我们认为依赖过滤条件下推至存储层这一方式很难再提升查询性能，尝试往其它方向进行探索：

- 不依赖存储层索引
- 不仅优化事实表与维度表 join

最终我们提炼两个新的运行时过滤优化点：维度表过滤广播和事实表 join 动态过滤，并在原版 RuntimeFilter 优化的基础上进行了扩展实现。

维度表过滤广播

这一优化的思想来源于 Lookahead Information Passing(LIP), 在论文《[Looking Ahead Makes Query Plans Robust](#)》中首次提出。其针对的场景如下图所示：



当事实表（lineorder）连续与多个维度表过滤结果做 multi-join 时，可将所有维度表的过滤信息下推至 join 之前。该方法与我们的 RuntimeFilter 的主要不同在于下推时考虑了完整的 multi-join tree 而不是局部 binary-join tree。其优化效果是即使 join ordering 为 bad case，无用的事实表数据也能够被尽早过滤掉，即让查询执行更加 robust。

我们参考论文算法实现了第一版过滤下推规则，但并没有达到预期的性能提升，主要原因在于：

- Spark CBO Join-Reorder 结合我们的遗传算法优化，已经达到了接近最优的 join ordering 效果
- 前置的 LIP filters 执行性能并没有明显优于 Spark BroadcastHashJoin 算子

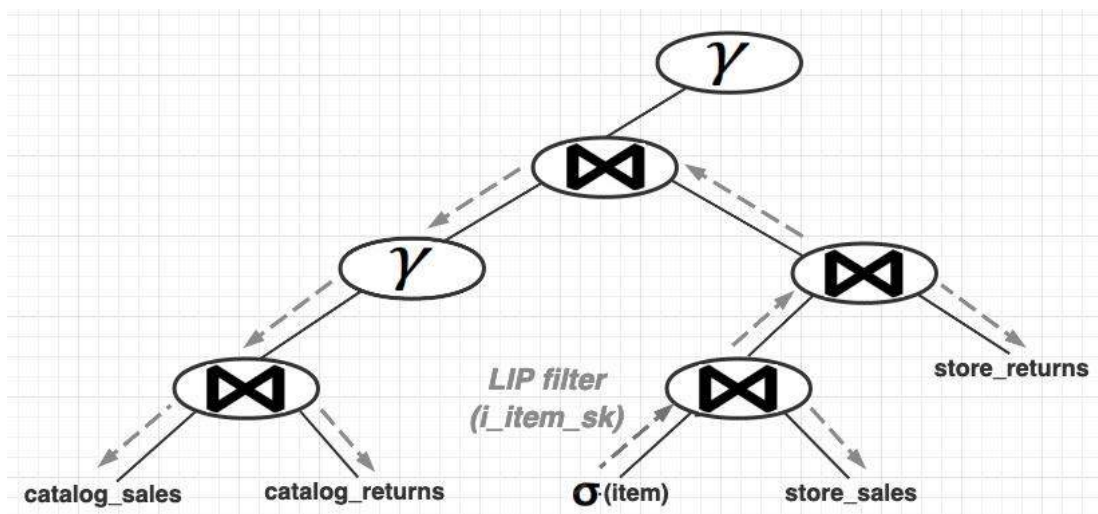
基于过滤条件可以传递至复杂 multi-join tree 的任意节点这一思想去发散思考，我们发现，当 multi-join tree 中存在多个事实表时，可将维度表过滤条件广播至所有的事实表 scan，从而减少后续事实表 SortMergeJoin 等耗时算子执行时所需处理的数据量。以一个简化版的 query 64 为例：

```

with cs_ui as
(select cs_item_sk
,sum(cs_ext_list_price) as sale
from catalog_sales
,catalog_returns
where cs_item_sk = cr_item_sk
and cs_order_number = cr_order_number
group by cs_item_sk)
select i_product_name product_name
,i_item_sk item_sk
,sum(ss_wholesale_cost) s1
from store_sales
,store_returns
,cs_ui
,item
where ss_item_sk = i_item_sk and
ss_item_sk = sr_item_sk and
ss_ticket_number = sr_ticket_number and
ss_item_sk = cs_ui.cs_item_sk and
i_color in ('almond','indian','sienna','blue','floral','rosy') and
i_current_price between 19 and 19 + 10 and
i_current_price between 19 + 1 and 19 + 15
group by i_product_name
,i_item_sk

```

该查询的 plan tree 如下图所示：



考虑未实现维度表过滤广播的执行流程，store_sales 数据经过 RuntimeFilter 和 BroadcastHashJoin 算子进行过滤，但由于过滤后数据仍然较大，后续的所有 join 都需要走昂贵的 SortMergeJoin 算子。但如果将 LIP filter 下推至 4 张事实表的 scan 算子（无需下推至存储层），不仅减少了 join 数据量，也减少了 catalog_sales 和 catalog_returns 表 join 后的 group-by aggregation 数据量。

LIP 实现

在 optimizer 层，我们在原版 RuntimeFilter 的 SyntheticJoinPredicate 规则后插入 PropagateDynamicValueFilter 规则，将合成的动态谓词广播至所有合法的 join 子树中；同时结合原有的谓词下推逻辑，保证动态谓词最终传播到所有相关的 scan 算子上。在算子层，LIP filters 的底层实现可以是 HashMap 或 BloomFilter，针对 TPC-DS 的数据特性，我们选择 BitMap 作为广播过滤条件的底层实现。由于 BitMap 本身是精确的（Exact Filter），可以结合主外键约束信息进一步做 semi-join 消除优化。基于主外键约束的优化规则将在系列后续文章做详细介绍。

应用该优化后，query 64 执行时间由 177 秒降低至 63 秒，加速比达到 2.8 倍。

事实表 Join 动态过滤

使用 BloomFilter 来优化大表 join 是一种常见的查询优化技术，比如在论文《Building a Hybrid Warehouse: Efficient Joins between Data Stored in HDFS and Enterprise Warehouse》中提出对 join 两表交替应用 BloomFilter 的 zig-zag join 方法，降低分布式 join 中的数据传输总量。对于 TPC-DS 测试集，以 query 93 为例，store_sales 与 store_returns join 后的结果集大小远小于 store_sales 原始数据量，非常适合应用这一优化。

BloomFilter 的构建和应用都存在较高的计算开销，对于 selectivity 较大的 join，盲目使用这一优化可能反而导致性能回退。基于静态 stats 的 join selectivity 估算往往误差，Spark 现有的 CBO 优化规则难以胜任鲁棒的 BloomFilter join 优化决策。因此，我们基于 Spark Adaptive Execution（AE）运行时重优化机制来实现动态的 BloomFilter join 优化规则。AE 的基本原理是在查询作业的每个 stage 执行完成后，允许优化器根据运行时采集的 stage stats 信息重新调整后续的物理执行计划。目前主要支持三种优化：

- （1）reduce stage 并发度调整；
- （2）针对 skew 情况的 shuffle 数据均衡分布；
- （3）SortMergeJoin 转换为 BroadcastHashJoin

基于 AE 的优化规则流程如下：

- 根据静态 stats 判断 join 的一端的 size 是否可能适合构建 BloomFilter（build side），如果是，则 build side 和 stream side 的 scan stage 会依次串行提交执行；否则这两个 stage 将并行执行。
- 在 build side 的 scan stage 执行完成后，AE 根据运行时收集的 size 和 join 列 histogram 进行代价估算，并决定最终走 BroadcastHashJoin、BloomFilter-SortMergeJoin 还是原本的 SortMergeJoin。
- 当物理执行计划为 BloomFilter-SortMergeJoin，优化器会插入一个新的作业并行扫描 build

side 的 shuffle 数据来构建 BloomFilter，并下推至 stream side 的 scan stage 中。

BloomFilter 算子实现

为了减少 BloomFilter 带来的额外开销，我们重新实现了高效的 BuildBloomFilter 和 Native-InBloomFilter 的算子。在构建阶段，使用 RDD aggregate 来合并各个数据分片的 BloomFilter 会导致 driver 成为数据传输和 bitmap 合并计算的性能瓶颈；使用 RDD treeAggregate 实现并行分层合并显著降低了整体的构建延迟。在过滤阶段，Native-InBloomFilter 的算子会被推入 scan 算子中合并执行。该算子直接访问 Spark 列式读取内存格式，按批量数据来调用 SIMD 优化的 native 函数，降低 CPU 执行开销；同时，我们将原版算法替换为 Blocked BloomFilter 算法实现，该算法通过牺牲少量的 bitmap 存储空间来换取访存时更低的 CPU cache miss 率。

应用该优化后，query 93 执行时间由 225 秒降低至 50 秒，加速比达到 4.5 倍。

EMR Spark-SQL 性能极致优化揭秘 Native Codegen Framework

简介： SparkSQL 多年来的性能优化集中在 Optimizer 和 Runtime 两个领域。前者的目的是为了获得最优的执行计划，后者的目的是针对既定的计划尽可能执行的更快。

作者：周克勇，花名一锤，阿里巴巴计算平台事业部 EMR 团队技术专家，大数据领域技术爱好者，对 Spark 有浓厚兴趣和一定的了解，目前主要专注于 EMR 产品中开源计算引擎的优化工作。

背景和动机

SparkSQL 多年来的性能优化集中在 Optimizer 和 Runtime 两个领域。前者的目的是为了获得最优的执行计划，后者的目的是针对既定的计划尽可能执行的更快。

相比于 Runtime, Optimizer 是更加通用的、跟实现无关的优化。无论是 Java 世界(Spark, Hive)还是 C++ 世界(Impala, MaxCompute), 无论是 Batch-Based(Spark, Hive)还是 MPP-Based(Impala, Presto), 甚至无论是大数据领域还是传统数据库领域亦或 HTAP 领域(HyPer, ADB), 在 Optimizer 层面考虑的都是非常类似的问题: Stats 收集, Cost 评估以及计划选择; 采用的优化技术也比较类似, 如 JoinReorder, CTE, GroupKey Elimination 等。尽管因为上下文不同(如是否有索引)在 Cost Model 的构造上会有不同, 或者特定场景下采用不同的空间搜索策略(如遗传算法 vs. 动态规划), 但方法大体是相同的。

长期以来, Runtime 的优化工作基本聚焦在解决当时的硬件瓶颈。如 MapReduce 刚出来时网络带宽是瓶颈, 所以 Google 做了很多 Locality 方面的优化; Spark 刚出来时解决的问题是磁盘 IO, 内存缓存的设计使得性能相比 MapReduce 有了数量级的提升; 后来 CPU 成为了新的瓶颈[1], 因此提升 CPU 性能成了近年来 Runtime 领域重要的优化方向。

提升 CPU 性能的两个主流技术是以 MonetDB/X100[2](如今演化为 VectorWise[3])为代表的向量化(Vectorized Processing)技术和以 HyPer[5][6]为代表的代码生成(CodeGen)技术(其中 Spark 跟进的是 CodeGen[9])。简单来说, 向量化技术沿用了火山模型, 但与其让 SQL 算子每次计算一条 Record, 向量化技术会积攒一批数据后再执行。逐批计算相比于逐条计算有了更大的优化空间, 例如虚函数的开销分摊, SIMD 优化, 更加 Cache 友好等。这个技术的劣势在于算子之间传递的数据从条变成了批, 因此增大了中间数据的物化开销。CodeGen 技术从另外一个角度解决虚函数开销和中间数据物化问题: 算子融合。简单来说, CodeGen 框架通过打破算子之间的界限把火山模型“压平”了, 把原来迭代器链压缩成了大的 for 循环, 同时生成语义相同的代码(Java/C++/LLVM), 紧接着用对应的工具链编译生成的代码, 最后用编译后的 class(Java)或 so(C++/LLVM)去执行, 从而把解释执行转变成了编译执行。

此外，尽管还是逐条执行，由于抹去了函数调用，一条 Record 从(Stage 内的)初始算子一直执行到结束算子都基本处于寄存器中，不会物化到内存。CodeGen 技术的劣势在于难以应用 SIMD 等优化。

两个门派相爱相杀，在经历了互相发论文验证自家优于对方后[4][8]两家走向了合作，合作产出了一系列项目和论文，而目前学界的主流看法也是两者融合是最优解，一些采用融合做法的项目也应运而生，如进化版 HyPer[6], Pelonton[7]等。

尽管学界已走到了融合，业界主流却没有很强的动力往融合的路子走，探究其主要原因一是目前融合的做法相比单独的优化并没有质的提升；二是融合技术目前没有一个广为接受的最优做法，还在探索阶段；三是业界在单一的技术上还没有发挥出最大潜力。以 SparkSQL 为例，从 2015 年 SparkSQL 首次露面自带的 Expression 级别的 Codegen，到后来参考 HyPer 实现的 WholeStage Codegen，再经过多年的打磨，SparkSQL 的 Codegen 技术已趋成熟，性能也获得了两次数量级的跃升。然而，也许是出于可维护性或开发者接受度的考虑，SparkSQL 的 Codegen 一直限制在生成 Java 代码，并没有尝试过 NativeCode(C/C++, LLVM)。尽管 Java 的性能已经很优，但相比于 Native Code 还是有一定的 Overhead，并缺乏 SIMD(Java 在做这方面 feature)，Prefetch 等语义，更重要的是，Native Code 直接操作裸金属，易于极致压榨硬件性能，对一些加速器(如 GPU)或新硬件(如 AEP)的支持也更方便。

基于以上动机，EMR 团队探索并开发了 SparkSQL Native Codegen 框架，为 SparkSQL 换了引擎，新引擎带来 20%左右的性能提升，为 EMR 再次获取世界第一立下汗马功劳，本文讲详细介绍 Native Codegen 框架。

核心问题

做 Native Codegen，核心问题有三个：

- 1.生成什么？
- 2.怎么生成？
- 3.如何集成到 Spark？

生成什么

针对生成什么代码，结合调研的结果以及开发同学的技术栈，有三个候选项：C/C++, LLVM, Weld IR。C/C++的优势是实现相对简单，只需对照 Spark 生成的 Java 代码逻辑改写即可，劣势是编译时间过长，下图是 HyPer 的测评数据，C++的编译时间比 LLVM 高了一个数量级。

	HyPer + C++	HyPer + LLVM
TPC-C [tps]	161,794	169,491
total compile time [s]	16.53	0.81

编译时间过长对小 query 很不友好，极端 case 编译时间比运行时间还要长。基于这个考虑，我们排除了 C/C++ 选项。上图看上去 LLVM 的编译时间非常友好，而且很多 Native CodeGen 的引擎，如 HyPer, Impala，以及阿里云自研大数据引擎 MaxCompute, ADB 等，均采用了 LLVM 作为目标代码。LLVM 对我们来说(对你们则不一定:D)最大的劣势就是过于底层，语法接近于汇编，试想用汇编重写 SparkSQL 算子的工作量会有多酸爽。大多数引擎也不会用 LLVM 写全量代码，如 HyPer 仅把算子核心逻辑用 LLVM 生成，其他通用功能(如 spill，复杂数据结构管理等)用 C++ 编写并提前编译好。即使 LLVM+C++ 节省了不少工作量，对我们来说依然不可接受，因此我们把目光转向了第三个选项: Weld IR(Intermediate Representation)。

首先简短介绍以下 Weld。Weld 的作者 Shoumik Palkar 是 Matei Zaharia 的学生，后者大家一定很熟悉，Spark 的作者。Weld 最初想解决的问题是不同 lib 之间互相调用时数据传输的开销，例如要在 pandas 里调用 numpy 的接口，首先 pandas 把数据写入内存，然后 numpy 读取内存进行计算，对于极度优化的 lib 来说，内存的写入和读取的时间可能会远超计算本身。针对这个问题，Weld 开发了 Common Runtime 并配套提供了一组 IR，再加上惰性求值的特性，只需(简单)修改 lib 使其符合 Weld 的规范，便可以做到不同 lib 共用 Weld Runtime，Weld Runtime 利用惰性求值实现跨 lib 的 Pipeline，从而省去数据物化的开销。Weld Runtime 还做了若干优化，如循环融合，循环展开，向量化，自适应执行等。此外，Weld 支持调用 C 代码，可以方便调用三方库。

我们感兴趣的是 Weld 提供的 IR 和对应的 Runtime。Weld IR 面向数据分析进行设计，因此语义上跟 SQL 非常接近，能较好的表达算子。数据结构层面，Weld IR 最核心的数据结构是 vec 和 struct，能较好地表达 SparkSQL 的 UnsafeRow Batch；基于 struct 和 vec 可以构造 dict，能较好的表达 SQL 里重度使用的 Hash 结构。操作层面，Weld IR 提供了类函数式语言的语义，如 map, filter, iterator 等，配合 builder 语义，能方便的表达 Project, Filter, Agg, BroadCastJoin 等算子语义。例如，以下 IR 表达了 Filter + Project 语义，具体含义是若第二列大于 10，则返回第一列：

```
|v: vec[{i32,i32}]| for(v,appender,|b,i,n| if(n.$1 > 10, merge(b,n.$0), b))
```

以下 IR 表达了 groupBy 的语义，具体含义是按照第一列做 groupBy 来计算第二列的 sum：

```
|v: vec[{i32,i32}]| for(v,dictmerger[i32,i32,+],|b,i,n| merge(b,{n.$0,n.$1}))
```

具体的语法定义请参考 Weld 文档

(<https://github.com/weld-project/weld/blob/master/docs/language.md>)。

Weld 开发者 API 提供了两个核型接口：

1. weld_module_compile, 把 Weld IR 编译成可执行模块(module)。
2. weld_module_run, 执行编译好的模块。

基本流程如下图所示，最终也是生成 LLVM 代码。



由此，Weld IR 的优势就显而易见了，既兼顾了性能(最终生成 LLVM 代码)，又兼顾了易用性(CodeGen Weld IR 相比 LLVM, C++方便很多)。基于这些考虑，我们最终选择 Weld IR 作为目标代码。

怎么生成

SparkSQL 原有的 CodeGen 框架之前简单介绍过了, 详见 <https://developer.aliyun.com/article/727277>。我们参考了 Spark 原有的做法，支持了表达式级别，算子级别，以及 WholeStage 级别的 Codegen。复用 Producer-Consumer 框架，每个算子负责生成自己的代码，最后由 WholeStageCodeGenExec 负责组装。

这个过程有两个关键问题：

1. 算子之间传输的介质是什么？
2. 如何处理 Weld 不支持的算子？

传输介质

不同于 Java，Weld IR 不提供循环结构，取而代之的是 vec 结构和其上的泛迭代器操作，因此 Weld IR 难以借鉴 Java Codegen 在 Stage 外层套个大循环，然后每个算子处理一条 Record 的模式，取而代之的做法是每个算子处理一批数据，IR 层面做假物化，然后依赖 Weld 的 Loop-Fusion 优化去消除物化。例如前面提到的 Filter 后接 Project，Filter 算子生成的 IR 如下，过滤掉第二列<=10 的数据：

```
|v:vec[{i32,i32}]| let res_fil = for(v,appender,|b,i,n| if(n.$1>10, merge(b,n), b)
```

Project 算子生成的 IR 如下，返回第一列数据：

```
let res_proj = for(res_fil,appender,|b,i,n| merge(b,n.$0))
```

表面上看上去 Filter 算子会把中间结果做物化，实际上 Weld 的 Loop-Fusion 优化器会消除此次物化，优化后代码如下：

```
|v: vec[{i32,i32}]| for(v,appender,|b,i,n| if(n.$1 > 10, merge(b,n.$0), b))
```

尽管依赖 Weld 的 Loop-Fusion 优化可以极大简化 CodeGen 的逻辑,但开发中我们发现 Loop-Fusion 过程非常耗时,对于复杂 SQL(嵌套 3 层以上)甚至无法在有限时间给出结果。当时面临两个选择:修改 Weld 的实现,或者修改 CodeGen 直接生成 Loop-Fusion 之后的代码,我们选择了后者。重构后生成的代码如下,其中 1,2,11 行由 Scan 算子生成,3,4,5,6,8,9,10 行由 Filter 算子生成,7 行由 Project 算子生成。

```
|v: vec[{i32,i32}]|
  for(v,appender,|b,i,n|
    if(
      n.$1 > 10,
      merge(
        b,
        n.$0
      ),
      b
```

这个优化使得编译时间重回亚秒级别。

Fallback 机制

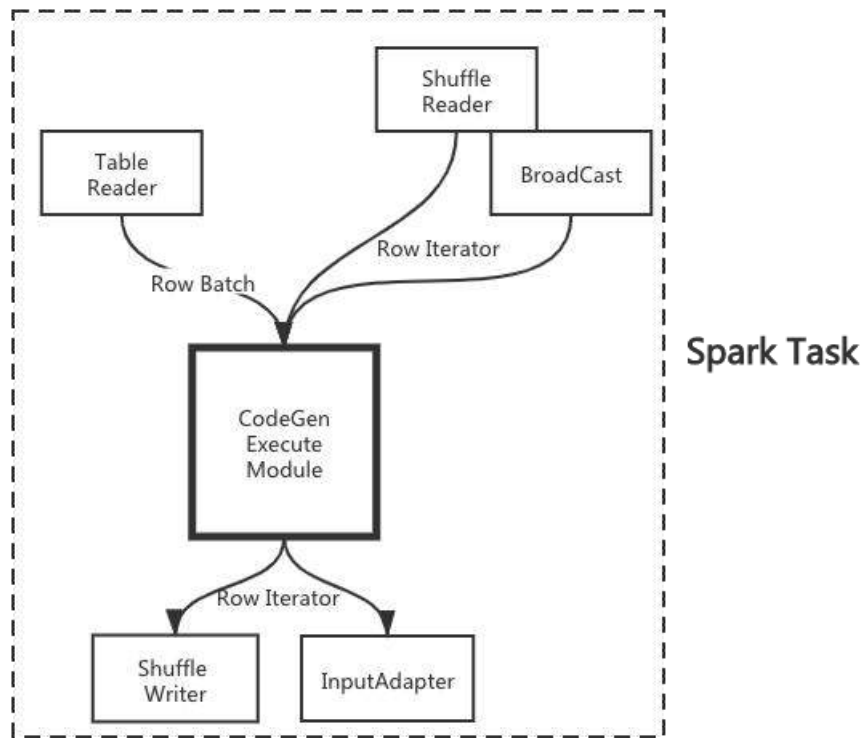
受限于 Weld 当前的表达能力,一些算子无法用 Weld 实现,例如 SortMergeJoin, Rollup 等。即使是原版的 Java CodeGen,一些算子如 Outter Join 也不支持 CodeGen,因此如何做好 Fallback 是保证正确性的前提。我们采用的策略很直观:若当前算子不支持 Native CodeGen,则由 Java CodeGen 接管。这里涉及的关键问题是 Fallback 的粒度:是算子级别还是 Stage 级别?

抛去实现难度不谈,虽然直观上算子粒度的 Fallback 更加合理,但实际上却会导致更严重的问题:Stage 内部 Pipeline 的断裂。如上文所述,CodeGen 的一个优势是把整个 Stage 的逻辑 Pipeline 化,打破算子之间的界限,单条 Record 从初始算子执行到结束算子,整个过程不存在物化。而算子粒度的 Fallback 则会导致 Stage 内部一部分走 Native Runtime,另一部分走 Java Runtime,则两者连接处无可避免存在中间数据物化,这个开销通常会大于 Native Runtime 带来的收益。

基于以上考虑,我们选择了 Stage 级别的 Fallback,在 CodeGen 阶段一旦遇到不支持的算子,则整个 Stage 都 Fallback 到 Java CodeGen。统计显示,整个 TPCDS Benchmark,命中 Native CodeGen 的 Stage 达到 80%。

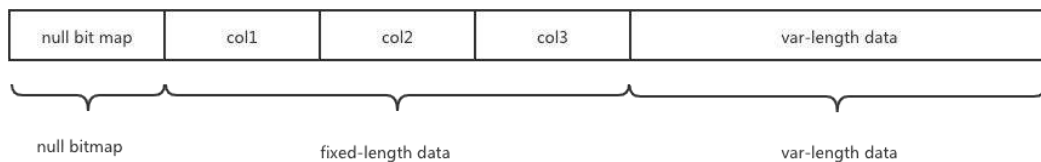
Spark 集成

完成了代码生成和 Fallback 机制,最后的问题就是如何跟 Spark 集成了。Spark 的 WholeStageCodegenExec 的执行可以理解为一个黑盒,无论上游是 Table Scan, Shuffle Read, 还是 Broadcast, 给到黑盒的输入类型只有两种: RowBatch(上游是 Table Scan)或 Row Iterator(上游非 Table Scan),而黑盒的输出固定为 Row Iterator, 如下图所示:



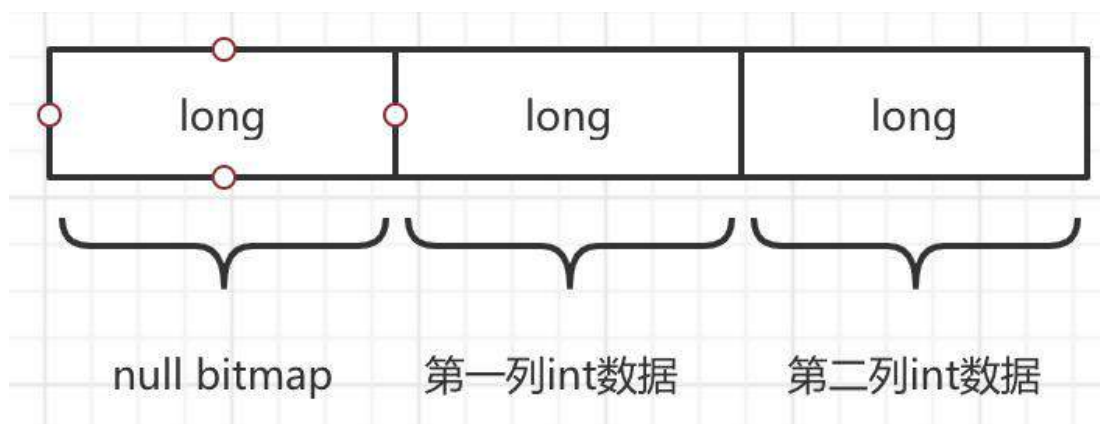
上文介绍我们选择了 Stage 级别的 Fallback，也就决定了黑盒要么是 Java Runtime，要么是 Native Runtime，不存在混合的情况，因此我们只需要关心如何把 Row Batch/Row Iterator 转化为 Weld 认识的内存布局，以及如何把 Weld 的输出转化成 Row Iterator 即可。为了进一步简化问题，我们注意到，尽管 Shuffle Reader/BroadCast 的输入是 Row Iterator，但本质上远端序列化的数据结构是 Row Batch，只不过 Spark 反序列化后转换成 Row Iterator 后再喂给 CodeGen Module，RowBatch 包装成 Row Iterator 非常简易。因此 Native Runtime 的输入输出可以统一成 RowBatch。

解决办法呼之欲出了：把 RowBatch 转换成 Weld vec！但我们更进了一步，何不直接把 Row Batch 喂给 Weld 从而省去内存转换呢？本质上 Row Batch 也是满足某种规范的字节流而已，Spark 也提供了 OffHeap 模式把内存直接存堆外(仅针对 Scan Stage。Shuffle 数据和 Broadcast 数据需要读到堆外)，Weld 可以直接访问。Spark UnsafeRow 的内存布局大致如下：



针对确定的 schema, null bitmap 和 fixed-length data 的结构是固定的, 可以映射成 struct, 而针对 var-length data 我们的做法是把这些数据 copy 到连续的内存地址中。如此一来, 针对无变长数据的 RowBatch, 我们直接把内存块喂给 Weld; 针对有变长部分的数据, 我们也只需做大粒度的内存拷贝(把定长部分和变长部分分别拷出来), 而无需做列级别的细粒度拷贝转换。

继续举前文的 Filter+Project 的例子, 一条 Record 包含两个 int 列, 其 UnsafeRow 的内存布局如下(为了对齐, Spark 里定长部分最少使用 8 字节)。



显而易见, 这个结构可以很方便映射成 Weld struct:

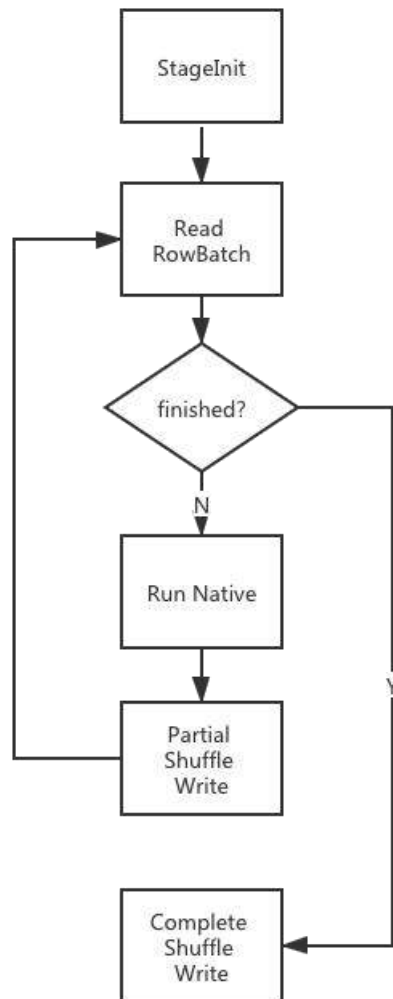
```
{i64,i64,i64}
```

而整个 Row Batch 便映射成 Weld vec:

```
vec[{i64,i64,i64}]
```

如此便解决了 Input 的问题。而 Weld Output 转 RowBatch 本质是以上过程的逆向操作, 不再赘述。

解决了 Java 和 Native 之间的数据转换问题, 剩下的就是如何执行了。首先我们根据当前 Stage 的 Mode 来决定走 Java Runtime 还是 Native Runtime。在 Native 分支, 首先会执行 StageInit 做 Stage 级别的初始化工作, 包括初始化 Weld, 加载编译好的 Weld Module, 拉取 Broadcast 数据(若有)等; 接着是一个循环, 每个循环读取一个 RowBatch(来自 Scan 或 Shuffle Reader)喂给 Native Runtime 执行, Output 转换并喂给 Shuffle Writer。如下图所示:



总结

本文介绍了EMR团队在Spark Native Codegen方向的探索实践,限于篇幅若干技术点和优化没有展开,后续可另开文详解,例如:

- 1.极致 Native 算子优化
- 2.数据转换详解
- 3.Weld Dict 优化

大家感兴趣的任何内容欢迎沟通:)

- [1] Making Sense of Performance in Data Analytics Frameworks. Kay Ousterhout
- [2] MonetDB/X100: Hyper-Pipelining Query Execution. Peter Boncz
- [3] Vectorwise: a Vectorized Analytical DBMS. Marcin Zukowski
- [4] Efficiently Compiling Efficient Query Plans for Modern Hardware. Thomas Neumann
- [5] HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots.
Alfons Kemper
- [6] Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation.
Harald Lang
- [7] Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and
Prefetching Work Together At Last. Prashanth Menon
- [8] Vectorization vs. Compilation in Query Execution. Juliusz Sompolski
- [9] <https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html>

Spark Codegen 浅析

简介：Codegen 是 Spark Runtime 优化性能的关键技术，核心在于动态生成 java 代码、即时 compile 和加载，把解释执行转化为编译执行。Spark Codegen 分为 Expression 级别和 WholeStage 级别，分别针对表达式计算和全 Stage 计算做代码生成，都取得了数量级的性能提升。本文浅析 Spark Codegen 技术原理。

作者：周克勇，花名一锤，阿里巴巴计算平台事业部 EMR 团队技术专家，大数据领域技术爱好者，对 Spark 有浓厚兴趣和一定的了解，目前主要专注于 EMR 产品中开源计算引擎的优化工作。

背景介绍

SparkSQL 的优越性能背后有两大技术支柱：Optimizer 和 Runtime。前者致力于寻找最优的执行计划，后者则致力于把既定的执行计划尽可能快地执行出来。Runtime 的多种优化可概括为两个层面：

1. **全局优化**。从提升全局资源利用率、消除数据倾斜、降低 IO 等角度做优化，包括自适应执行(Adaptive Execution), Shuffle Removal 等。
2. **局部优化**。优化具体的 Task 的执行效率，主要依赖 Codegen 技术，具体包括 Expression 级别和 WholeStage 级别的 Codegen。

本文介绍 Spark Codegen 的技术原理。

Case Study

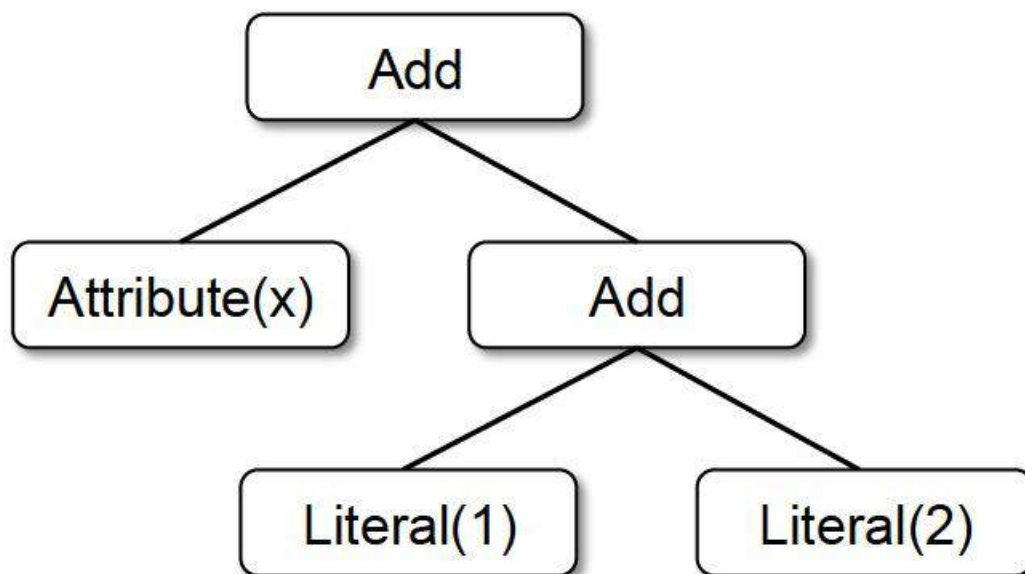
本节通过两个具体 case 介绍 Codegen 的做法。

Expression 级别

考虑下面的表达式计算： $x + (1 + 2)$ ，用 scala 代码表达如下：

```
Add(Attribute(x), Add(Literal(1), Literal(2)))
```

语法树如下：



递归求值这棵语法树的常规代码如下：

```

tree.transformUp {
  case Attribute(idx) => Literal(row.getValue(idx))
  case Add(Literal(c1),Literal(c2)) => Literal(c1+c2)
  case Literal(c) => Literal(c)
}

```

执行上述代码需要做很多类型匹配、虚函数调用、对象创建等额外逻辑，这些 overhead 远超对表达式求值本身。

为了消除这些 overhead，Spark Codegen 直接拼成求值表达式的 java 代码并进行即时编译。具体分为三个步骤：

1. **代码生成**。根据语法树生成 java 代码，封装在 wrapper 类中：

```

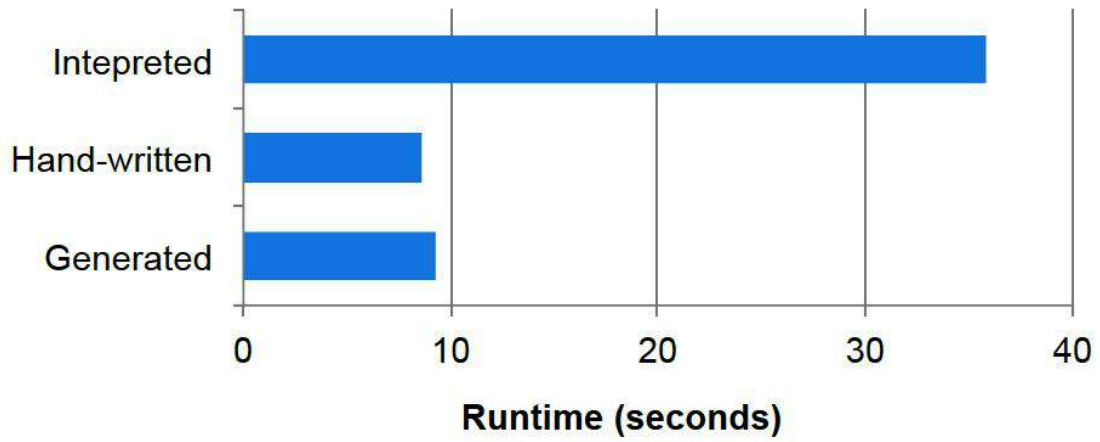
... // class wrapper
row.getValue(idx) + (1 + 2)
... // class wrapper

```

2. **即时编译**。使用 Janino 框架把生成代码编译成 class 文件。

3. **加载执行**。最后加载并执行。

优化前后性能有数量级的提升。

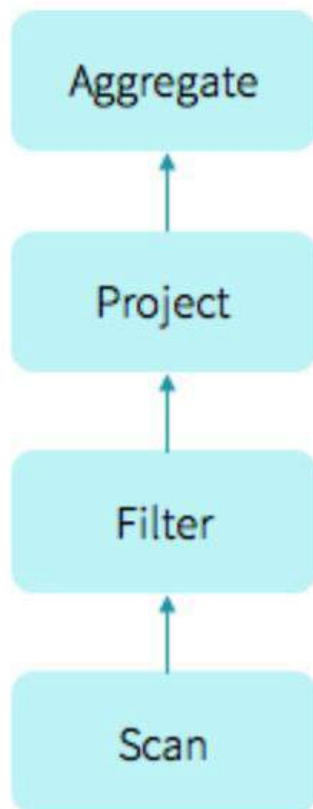


WholeStage 级别

考虑如下的 sql 语句:

```
select count(*) from store_sales  
where ss_item_sk=1000;
```

生成的物理执行计划如下:



执行该计划的常规做法是使用火山模型(volcano model), 每个 Operator 都继承了 Iterator 接口, 其 next() 方法首先驱动上游执行拿到输入, 然后执行自己的逻辑。代码示例如下:

```
class Agg extends Iterator[Row] {
  def doAgg() {
    while (child.hasNext()) {
      val row = child.next();
      // do aggregation
      ...
    }
  }
  def next(): Row {
    if (!doneAgg) {
      doAgg();
    }
    return aggIter.next();
  }
}

class Filter extends Iterator[Row] {
  def next(): Row {
    var current = child.next()
    while (current != null && !predicate(current)) {
      current = child.next()
    }
    return current;
  }
}
```

从上述代码可知, 火山模型会有大量类型转换和虚函数调用。虚函数调用会导致 CPU 分支预测失败, 从而导致严重的性能回退。

为了消除这些 overhead, Spark WholestageCodegen 会为该物理计划生成类型确定的 java 代码, 然后类似 Expression 的做法即时编译和加载执行。本例生成的 java 代码示例如下(非真实代码, 真实代码片段见后文):


```
var count = 0
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1
  }
}
```

优化前后性能提升数据如下:



Spark Codegen 框架

Spark Codegen 框架有三个核心组成部分

1. 核心接口/类
2. CodegenContext
3. Produce-Consume Pattern

接下来详细介绍。

接口/类

四个核心接口:

1. CodegenSupport(接口)

实现该接口的 Operator 可以将自己的逻辑拼成 java 代码。重要方法:

```
produce() // 输出本节点产出 Row 的 java 代码
consume() // 输出本节点消费上游节点输入的 Row 的 java 代码
```

实现类包括但不限于: ProjectExec, FilterExec, HashAggregateExec, SortMergeJoinExec。

2. WholeStageCodegenExec(类)

CodegenSupport 的实现类之一, Stage 内部所有相邻的实现 CodegenSupport 接口的 Operator 的融合, 产生的代码把所有被融合的 Operator 的执行逻辑封装到一个 Wrapper 类中, 该 Wrapper 类作为 Janino 即时 compile 的入参。

3. InputAdapter(类)

CodegenSupport 的实现类之一, 胶水类, 用来连接 WholeStageCodegenExec 节点和未实现 CodegenSupport 的上游节点。

4. BufferedRowIterator(接口)

WholeStageCodegenExec 生成的 java 代码的父类, 重要方法:

```
public InternalRow next() // 返回下一条 Row
public void append(InternalRow row) // append 一条 Row
```

CodegenContext

管理生成代码的核心类。主要涵盖以下功能:

- 1.命名管理。**保证同一 Scope 内无变量名冲突。
- 2.变量管理。**维护类变量, 判断变量类型(应该声明为独立变量还是压缩到类型数组中), 维护变量初始化逻辑等。
- 3.方法管理。**维护类方法。
- 4.内部类管理。**维护内部类。
- 5.相同表达式管理。**维护相同子表达式, 避免重复计算。
- 6.size 管理。**避免方法、类 size 过大, 避免类变量数过多, 进行比较拆分。如把表达式块拆分成多个函数; 把函数、变量定义拆分到多个内部类。
- 7.依赖管理。**维护该类依赖的外部对象, 如 Broadcast 对象、工具对象、度量对象等。
- 8.通用模板管理。**提供通用代码模板, 如 genComp, nullSafeExec 等。

Produce-Consume Pattern

相邻 Operator 通过 Produce-Consume 模式生成代码。

Produce 生成整体处理的框架代码, 例如 aggregation 生成的代码框架如下:

```

if (!initialized) {
    # create a hash map, then build the aggregation hash map
    # call child.produce()
    initialized = true;
}
while (hashmap.hasNext()) {
    row = hashmap.next();
    # build the aggregation results
    # create variables for results
    # call consume(), which will call parent.doConsume()
    if (shouldStop()) return;
}

```

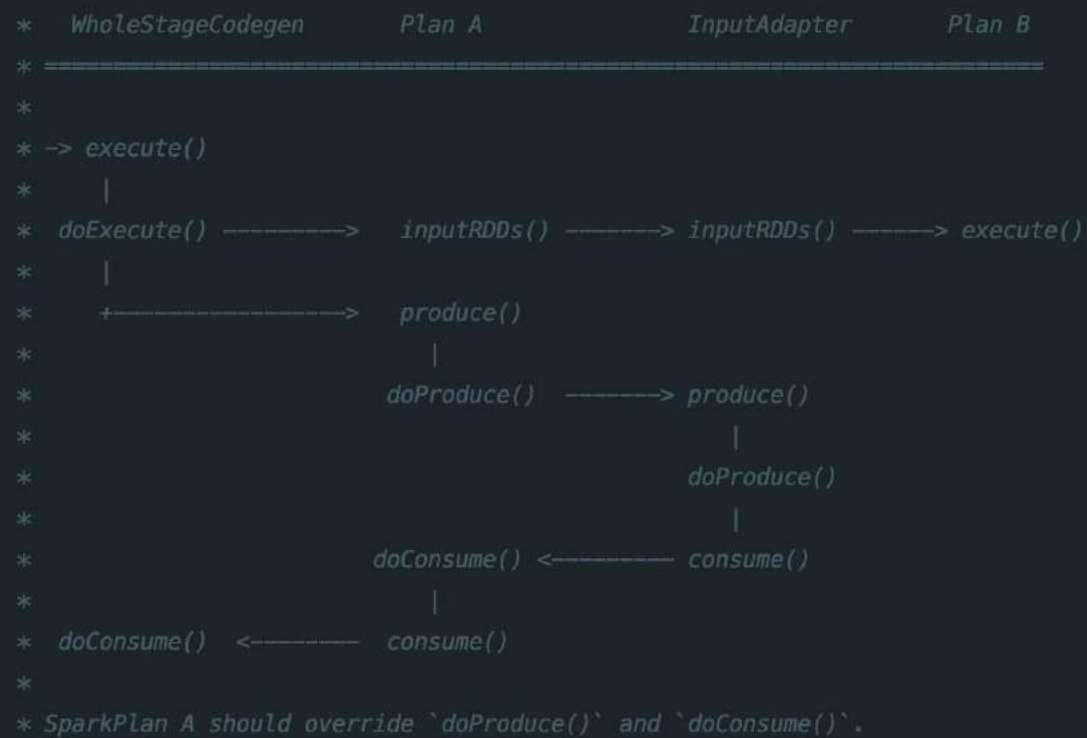
Consume 生成当前节点处理上游输入的 Row 的逻辑。如 Filter 生成代码如下:

```

# code to evaluate the predicate expression, result is isNull1 and value2
if (!isNull1 && value2) {
    # call consume(), which will call parent.doConsume()
}

```

下图比较清晰地展示了 WholestageCodegen 生成 java 代码的 call graph:



Case Study 的示例，生成的真实代码如下：

```

== Subtree 1 / 2 ==
*(2) HashAggregate(keys=[], functions=[count(1)], output=[count(1)#326L])
+- Exchange SinglePartition
  +- *(1) HashAggregate(keys=[], functions=[partial_count(1)], output=[count#329L])
    +- *(1) Project
      +- *(1) Filter (isnotnull(ss_item_sk#13L) && (ss_item_sk#13L = 1000))
        +- *(1) FileScan parquet [ss_item_sk#13L] Batched: true, Format: Parquet, Location:
InMemoryFileIndex[file:/home/admin/zhoukeyong/workspace/tpc/tpcds/data/parquet/10/store_sale
s/par..., PartitionFilters: [], PushedFilters: [IsNotNull(ss_item_sk), EqualTo(ss_item_sk,1000)],
ReadSchema: struct<ss_item_sk:bigint>

```

Generated code:

```

/* 001 */ public Object generate(Object[] references) {
/* 002 */     return new GeneratedIteratorForCodegenStage2(references);
/* 003 */ }
/* 004 */
/* 005 */ // codegenStageId=2
/* 006 */ final class GeneratedIteratorForCodegenStage2 extends
org.apache.spark.sql.execution.BufferedRowIterator {
/* 007 */     private Object[] references;
/* 008 */     private scala.collection.Iterator[] inputs;
/* 009 */     private boolean agg_initAgg_0;
/* 010 */     private boolean agg_bufIsNull_0;
/* 011 */     private long agg_bufValue_0;
/* 012 */     private scala.collection.Iterator inputadapter_input_0;
/* 013 */     private org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter[]
agg_mutableStateArray_0 = new
org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter[1];
/* 014 */
/* 015 */     public GeneratedIteratorForCodegenStage2(Object[] references) {
/* 016 */         this.references = references;
/* 017 */     }
/* 018 */
/* 019 */     public void init(int index, scala.collection.Iterator[] inputs) {
/* 020 */         partitionIndex = index;

```

```

/* 021 */    this.inputs = inputs;
/* 022 */
/* 023 */    inputadapter_input_0 = inputs[0];
/* 024 */    agg_mutableStateArray_0[0] = new
org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter(1, 0);
/* 025 */
/* 026 */    }
/* 027 */
/* 028 */    private void agg_doAggregateWithoutKey_0() throws java.io.IOException {
/* 029 */        // initialize aggregation buffer
/* 030 */        agg_bufIsNull_0 = false;
/* 031 */        agg_bufValue_0 = 0L;
/* 032 */
/* 033 */        while (inputadapter_input_0.hasNext() && !stopEarly()) {
/* 034 */            InternalRow inputadapter_row_0 = (InternalRow) inputadapter_input_0.next();
/* 035 */            long inputadapter_value_0 = inputadapter_row_0.getLong(0);
/* 036 */
/* 037 */            agg_doConsume_0(inputadapter_row_0, inputadapter_value_0);
/* 038 */            if (shouldStop()) return;
/* 039 */        }
/* 040 */
/* 041 */    }
/* 042 */
/* 043 */    private void agg_doConsume_0(InternalRow inputadapter_row_0, long agg_expr_0_0)
throws java.io.IOException {
/* 044 */        // do aggregate
/* 045 */        // common sub-expressions
/* 046 */
/* 047 */        // evaluate aggregate function
/* 048 */        long agg_value_3 = -1L;
/* 049 */        agg_value_3 = agg_bufValue_0 + agg_expr_0_0;
/* 050 */        // update aggregation buffer
/* 051 */        agg_bufIsNull_0 = false;
/* 052 */        agg_bufValue_0 = agg_value_3;
/* 053 */
/* 054 */    }
/* 055 */
/* 056 */    protected void processNext() throws java.io.IOException {
/* 057 */        while (!agg_initAgg_0) {
/* 058 */            agg_initAgg_0 = true;
/* 059 */            long agg_beforeAgg_0 = System.nanoTime();
/* 060 */            agg_doAggregateWithoutKey_0();
/* 061 */            ((org.apache.spark.sql.execution.metric.SQLMetric) references[1] /* aggTime
*/)
.add((System.nanoTime() - agg_beforeAgg_0) / 1000000);
/* 062 */

```

```

/* 063 */      // output the result
/* 064 */
/* 065 */      ((org.apache.spark.sql.execution.metric.SQLMetric) references[0] /*
numOutputRows */).add(1);
/* 066 */      agg_mutableStateArray_0[0].reset();
/* 067 */
/* 068 */      agg_mutableStateArray_0[0].zeroOutNullBytes();
/* 069 */
/* 070 */      agg_mutableStateArray_0[0].write(0, agg_bufValue_0);
/* 071 */      append((agg_mutableStateArray_0[0].getRow()));
/* 072 */      }
/* 073 */      }
/* 074 */
/* 075 */ }

```

== Subtree 2 / 2 ==

* (1) HashAggregate(keys=[], functions=[partial_count(1)], output=[count#329L])

+ - * (1) Project

+ - * (1) Filter (isnotnull(ss_item_sk#13L) && (ss_item_sk#13L = 1000))

+ - * (1) FileScan parquet [ss_item_sk#13L] Batched: true, Format: Parquet, Location: InMemoryFileIndex[file:/home/admin/zhoukeyong/workspace/tpc/tpcds/data/parquet/10/store_sales/par..., PartitionFilters: [], PushedFilters: [IsNotNull(ss_item_sk), EqualTo(ss_item_sk,1000)], ReadSchema: struct<ss_item_sk:bigint>

Generated code:

```

/* 001 */ public Object generate(Object[] references) {
/* 002 */     return new GeneratedIteratorForCodegenStage1(references);
/* 003 */ }
/* 004 */
/* 005 */ // codegenStageId=1
/* 006 */ final class GeneratedIteratorForCodegenStage1 extends
org.apache.spark.sql.execution.BufferedRowIterator {
/* 007 */     private Object[] references;
/* 008 */     private scala.collection.Iterator[] inputs;
/* 009 */     private boolean agg_initAgg_0;
/* 010 */     private boolean agg_bufIsNull_0;
/* 011 */     private long agg_bufValue_0;
/* 012 */     private long scan_scanTime_0;
/* 013 */     private boolean outputMetaColumns;
/* 014 */     private int scan_batchIdx_0;
/* 015 */     private org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter[]
scan_mutableStateArray_3 = new
org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter[3];

```



```
/* 016 */ private org.apache.spark.sql.vectorized.ColumnarBatch[] scan_mutableStateArray_1 =
new org.apache.spark.sql.vectorized.ColumnarBatch[1];
/* 017 */ private scala.collection.Iterator[] scan_mutableStateArray_0 = new
scala.collection.Iterator[1];
/* 018 */ private org.apache.spark.sql.execution.vectorized.OffHeapColumnVector[]
scan_mutableStateArray_2 = new org.apache.spark.sql.execution.vectorized.OffHeapColumnVector[1];
/* 019 */
/* 020 */ public GeneratedIteratorForCodegenStage1(Object[] references) {
/* 021 */     this.references = references;
/* 022 */ }
/* 023 */
/* 024 */ public void init(int index, scala.collection.Iterator[] inputs) {
/* 025 */     partitionIndex = index;
/* 026 */     this.inputs = inputs;
/* 027 */
/* 028 */     scan_mutableStateArray_0[0] = inputs[0];
/* 029 */     outputMetaColumns = false;
/* 030 */     scan_mutableStateArray_3[0] = new
org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter(1, 0);
/* 031 */     scan_mutableStateArray_3[1] = new
org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter(1, 0);
/* 032 */     scan_mutableStateArray_3[2] = new
org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter(1, 0);
/* 033 */
/* 034 */ }
/* 035 */
/* 036 */ private void agg_doAggregateWithoutKey_0() throws java.io.IOException {
/* 037 */     // initialize aggregation buffer
/* 038 */     agg_bufIsNull_0 = false;
/* 039 */     agg_bufValue_0 = 0L;
/* 040 */
/* 041 */     if (scan_mutableStateArray_1[0] == null) {
/* 042 */         scan_nextBatch_0();
/* 043 */     }
/* 044 */     while (scan_mutableStateArray_1[0] != null) {
/* 045 */         int scan_numRows_0 = scan_mutableStateArray_1[0].numRows();
/* 046 */         int scan_localEnd_0 = scan_numRows_0 - scan_batchIdx_0;
/* 047 */         for (int scan_localIdx_0 = 0; scan_localIdx_0 < scan_localEnd_0; scan_localIdx_0++) {
/* 048 */             int scan_rowIdx_0 = scan_batchIdx_0 + scan_localIdx_0;
/* 049 */             if (!scan_mutableStateArray_1[0].validAt(scan_rowIdx_0)) { continue; }
/* 050 */             do {
/* 051 */                 boolean scan_isNull_0 =
scan_mutableStateArray_2[0].isNullAt(scan_rowIdx_0);
```

```
/* 058 */          if (!filter_value_2) continue;
/* 059 */
/* 060 */          ((org.apache.spark.sql.execution.metric.SQLMetric) references[2] /*
numOutputRows */).add(1);
/* 061 */
/* 062 */          agg_doConsume_0();
/* 063 */
/* 064 */          } while(false);
/* 065 */          // shouldStop check is eliminated
/* 066 */          }
/* 067 */          scan_batchIdx_0 = scan_numRows_0;
/* 068 */          scan_mutableStateArray_1[0] = null;
/* 069 */          scan_nextBatch_0();
/* 070 */      }
/* 071 */      ((org.apache.spark.sql.execution.metric.SQLMetric) references[1] /* scanTime
*/).add(scan_scanTime_0 / (1000 * 1000));
/* 072 */      scan_scanTime_0 = 0;
/* 073 */
/* 074 */  }
/* 075 */
/* 076 */  private void scan_nextBatch_0() throws java.io.IOException {
/* 077 */      long getBatchStart = System.nanoTime();
/* 078 */      if (scan_mutableStateArray_0[0].hasNext()) {
/* 079 */          scan_mutableStateArray_1[0] =
(org.apache.spark.sql.vectorized.ColumnarBatch)scan_mutableStateArray_0[0].next();
/* 080 */          ((org.apache.spark.sql.execution.metric.SQLMetric) references[0] /*
numOutputRows */).add(scan_mutableStateArray_1[0].numRows());
/* 081 */          scan_batchIdx_0 = 0;
/* 082 */          scan_mutableStateArray_2[0] =
(org.apache.spark.sql.execution.vectorized.OffHeapColumnVector) (outputMetaColumns ?
/* 083 */              scan_mutableStateArray_1[0].column(0, true) :
scan_mutableStateArray_1[0].column(0));
/* 084 */
/* 085 */      }
/* 086 */      scan_scanTime_0 += System.nanoTime() - getBatchStart;
/* 087 */  }
/* 088 */
/* 089 */  private void agg_doConsume_0() throws java.io.IOException {
```

```
/* 090 */    // do aggregate
/* 091 */    // common sub-expressions
/* 092 */
/* 093 */    // evaluate aggregate function
/* 094 */    long agg_value_1 = -1L;
/* 095 */    agg_value_1 = agg_bufValue_0 + 1L;
/* 096 */    // update aggregation buffer
/* 097 */    agg_bufIsNull_0 = false;
/* 098 */    agg_bufValue_0 = agg_value_1;
/* 099 */
/* 100 */ }
/* 101 */
/* 102 */ protected void processNext() throws java.io.IOException {
/* 103 */     while (!agg_initAgg_0) {
/* 104 */         agg_initAgg_0 = true;
/* 105 */         long agg_beforeAgg_0 = System.nanoTime();
/* 106 */         agg_doAggregateWithoutKey_0();
/* 107 */         ((org.apache.spark.sql.execution.metric.SQLMetric) references[4] /* aggTime
/* 108 */         */).add((System.nanoTime() - agg_beforeAgg_0) / 1000000);
/* 109 */
/* 110 */         // output the result
/* 111 */         ((org.apache.spark.sql.execution.metric.SQLMetric) references[3] /*
/* 112 */         numOutputRows */).add(1);
/* 113 */         scan_mutableStateArray_3[2].reset();
/* 114 */         scan_mutableStateArray_3[2].zeroOutNullBytes();
/* 115 */
/* 116 */         scan_mutableStateArray_3[2].write(0, agg_bufValue_0);
/* 117 */         append((scan_mutableStateArray_3[2].getRow()));
/* 118 */     }
/* 119 */ }
/* 120 */
/* 121 */ }
```

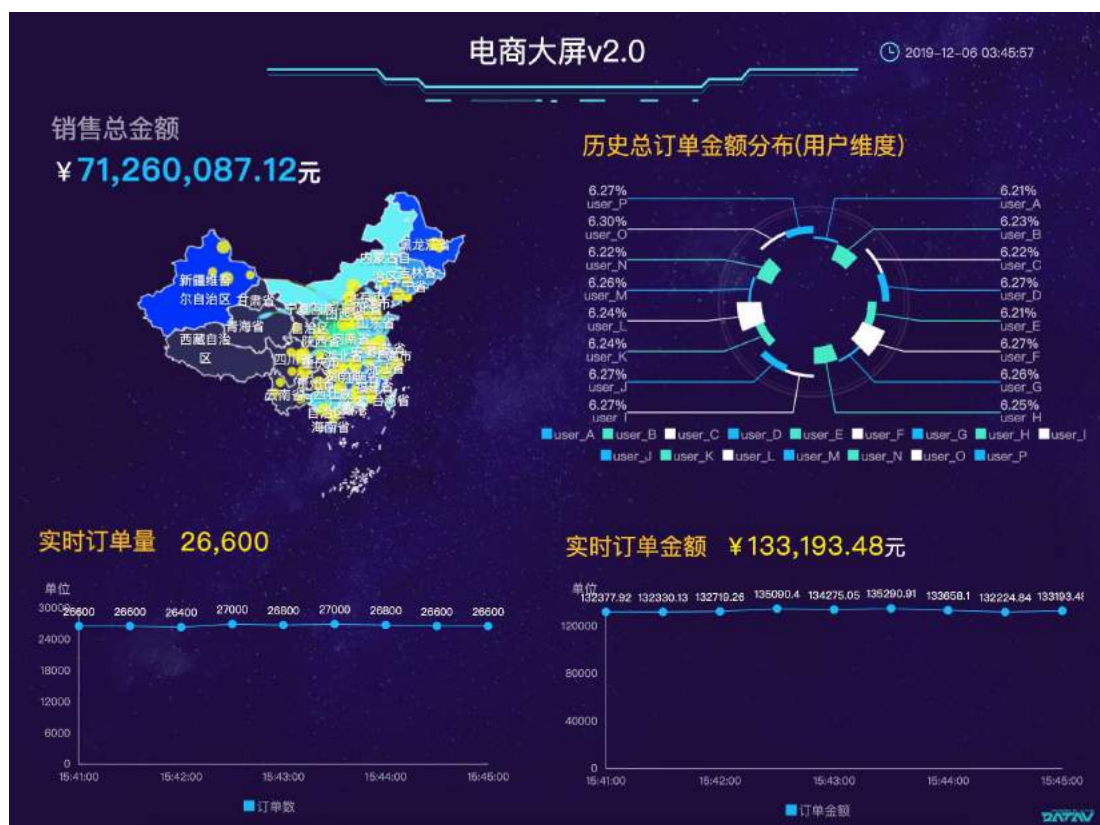
Tablestore 结合 Spark 的流批一体 SQL 实战

简介： 本文将通过结合 Tablestore 和 Spark 的流批一体存储和计算，来自建电商大屏完成电商数据的分析和可视化。

作者：王卓然 花名卓然 阿里云存储服务技术专家

背景介绍

电子商务模式是指在网络环境和大数据环境下基于一定技术基础的商务运作方式和盈利模式，对于数据的分析和可视化是电商运营中最重要的部分之一，而电商大屏提供了数据分析和可视化的完美结合。电商大屏包含有全量订单和实时订单的聚合，全量订单的聚合提供的是全景的综合数据视图，而实时订单的聚合展示的是实时的运营指标数据。本文将通过结合 Tablestore 和 Spark 的流批一体存储和计算，来自建电商大屏完成电商数据的分析和可视化，其效果图如下。



架构设计

在本次的电商大屏实战中，客户端会实时向 Tablestore 插入原始订单数据，实时流计算会通过 Spark Structured Streaming 实时统计一个窗口周期时间内的订单数和订单金额统计，并将聚合结果写回 Tablestore，最终在 DataV 大屏上进行展示，而离线批计算通过 Spark SQL 进行原始订单数据的总金额和用户维度总金额的离线聚合，聚合结果也会写回 Tablestore，并最终在 DataV 大屏上进行展示，整个场景的架构图如下图所示。



准备工作

1. 创建阿里云 E-MapReduce 的 Hadoop 集群，文档参见[创建集群](#)。
2. 下载 E-MapReduce 的最新 SDK 包，包名的格式为`js-emr-datasources_shaded_*.jar`
里面会包含有 Tablestore 相关的 Spark 批流 Source 和 Sink。

数据源说明

数据源是一张简单的原始订单表 OrderSource，表有两个主键 UserId(用户 ID)和 OrderId(订单 ID)和两个属性列 price(价格)和 timestamp(订单时间)，数据示例如下图所示。

数据源: OrderSource		表格数据最多显示50行。			
<input type="checkbox"/>	详细数据	Userid(主键)	OrderId(主键)	price	timestamp
<input type="checkbox"/>	详细数据	user_A	000c51aa-b6b4-49de-a...	1.24	1575600298455
<input type="checkbox"/>	详细数据	user_A	000c9907-955d-43f2-9...	7.74	1575600257702
<input type="checkbox"/>	详细数据	user_A	001a1952-7431-4158-8...	0.92	1575600310047
<input type="checkbox"/>	详细数据	user_A	003d9dd7-6ea3-48d4-b...	6.67	1575600282824
<input type="checkbox"/>	详细数据	user_A	00403eb1-6d80-43d5-9...	3.17	1575600299848
<input type="checkbox"/>	详细数据	user_A	00615679-4fe0-4cd5-a...	9.85	1575600304933

批流 SQL 流程详解

创建数据源表

1.登陆 EMR Header 机器，执行以下命令，启动 sql 客户端，该客户端用于批流 SQL 计算，其中 emr-datasources_shaded_*.jar 为准备工作中下载的 EMR 最新版的 SDK 包。

```
streaming-sql --driver-class-path emr-datasources_shaded_*.jar --jars emr-datasources_shaded_*.jar
--master yarn-client --num-executors 8 --executor-memory 2g --executor-cores 2
```

1. 创建原始订单数据表(Source 表)的外表 order_source，该外表将用于后续的流批 SQL 执行。

```
DROP TABLE IF EXISTS order_source;
CREATE TABLE order_source
USING tablestore
OPTIONS(
  endpoint="http://vehicle-test.cn-hangzhou.vpc.tablestore.aliyuncs.com",
  access.key.id="",
  access.key.secret="",
  instance.name="vehicle-test",
  table.name="OrderSource",
  tunnel.id="2b7bbf3d-d6c4-4cea-89fe-71998bccaf19",
  catalog='{"columns": {"UserId": {"col": "UserId", "type": "string"}, "OrderId": {"col": "OrderId", "type":
"string"}, "price": {"cols": "price", "type": "double"}, "timestamp": {"cols": "timestamp", "type":
"long"}}}'
);
```


参数说明：

参数名	解释
endpoint	表格存储实例的访问地址
access.key.id	阿里云账号 AK ID
access.key.secret	阿里云账号 AK Secret
instance.name	表格存储实例名
table.name	表格存储表名
tunnel.id	表格存储的增量通道 ID, 该参数用于实时的增量 SQL, 批量 SQL 时非必须。
catalog	表的字段 Schema 定义, 上述示例中对应的四个列为 UserId(主键), OrderId(主键), price, timestamp, 数据类型分别为 string, string, double, long。

实时流计算

实时流计算将实时统计一个窗口周期时间内的订单数和订单金额统计, 并将聚合结果写回 Tablestore。首先创建流计算的 Sink 外表 order_stream_sink(对应 Tablestore 表 OrderStreamSink), 然后运行流计算 SQL 进行实时聚合, 最后将聚合结果实时写回 Tablestore 目的表中。

Sink 表的各参数含义和 Source 表一致, 其中 catalog 字段的内容有所不同, 对应的 Sink 表中有四个字段, begin(开始时间, 主键列, 格式为 2019-11-27 14:54:00), end(结束时间, 主键列), count(订单数), totalPrice(订单总金额)。

```
// 创建 Sink 表 order_stream_sink 对应 Tablestore 的表 OrderStreamSink(主键为 begin 和 end 两列)
DROP TABLE IF EXISTS order_stream_sink;
CREATE TABLE order_stream_sink
USING tablestore
OPTIONS(
  endpoint="http://vehicle-test.cn-hangzhou.vpc.tablestore.aliyuncs.com",
  access.key.id="",
  access.key.secret="",
  instance.name="vehicle-test",
  table.name="OrderStreamSink",
  catalog='{
    "columns": {
      "begin": {
        "col": "begin",
        "type": "string"
      },
      "end": {
        "col": "end",
        "type": "string"
      },
      "count": {
        "col": "count",
        "type": "long"
      },
      "totalPrice": {
        "col": "totalPrice",
        "type": "double"
      }
    }
  }'
);
```

```
// 在 order_source 表上创建视图 order_source_stream_view
CREATE SCAN order_source_stream_view ON order_source USING STREAM OPTIONS
("maxoffsetsperchannel"="10000");

// 在视图 order_source_stream_view 上运行 STREAM SQL 作业，以下样例会按 30s 粒度进行订单
数和订单金额的聚合，
// 聚合结果将写回 Tablestore 表 OrderStreamSink。
CREATE STREAM job1
options(
checkpointLocation='/tmp/spark/cp/job1',
outputMode='update'
)
INSERT INTO order_stream_sink
SELECT CAST(window.start AS String) AS begin, CAST(window.end AS String) AS end, count(*) AS count,
CAST(sum(price) AS Double) AS totalPrice FROM order_source_stream_view GROUP BY
```

在运行 Stream SQL 后，可以实时得到聚合结果，聚合结果样例如下图所示，聚合结果存放在 OrderStreamSink 表中，通过 Tablestore 和 DataV 的[直连功能](#)，可以很容易的将结果绘制在 DataV 的大屏上。

数据源: OrderStreamSink		表格数据最多显示50行。			
<input type="checkbox"/>	详细数据	begin(主键)	end(主键)	count	totalPrice
<input type="checkbox"/>	详细数据	2019-12-06 10:55:30	2019-12-06 10:56:00	18800 ▼	93988.24999999994 ▼
<input type="checkbox"/>	详细数据	2019-12-06 10:55:00	2019-12-06 10:55:30	23800 ▼	119194.13 ▼
<input type="checkbox"/>	详细数据	2019-12-06 10:54:30	2019-12-06 10:55:00	25400 ▼	127011.12999999998 ▼
<input type="checkbox"/>	详细数据	2019-12-06 10:54:00	2019-12-06 10:54:30	23600 ▼	118741.59999999998 ▼
<input type="checkbox"/>	详细数据	2019-12-06 10:53:30	2019-12-06 10:54:00	26200 ▼	130843.28000000006 ▼
<input type="checkbox"/>	详细数据	2019-12-06 10:53:00	2019-12-06 10:53:30	22800 ▼	113727.05000000002 ▼
<input type="checkbox"/>	详细数据	2019-12-06 10:52:30	2019-12-06 10:53:00	26200 ▼	131344.82999999993 ▼
<input type="checkbox"/>	详细数据	2019-12-06 10:52:00	2019-12-06 10:52:30	24800 ▼	122971.66000000004 ▼

离线批计算

离线批计算将进行原始订单数据的总金额和用户维度总金额的离线聚合，首先会创建两张 Sink 表分别存放历史总金额和用户维度总金额的聚合数据，然后直接在源表 order_source 上运行批计算 SQL，最后得到聚合结果。

```
// 批计算任务
// 用户维度结果表: OrderBatchSink(主键 UserId, 属性列 count, totalPrice)
// 总数据维度结果表: OrderTotalSink(主键 Count, 属性列 totalPrice)
DROP TABLE IF EXISTS order_batch_sink;
CREATE TABLE order_batch_sink
USING tablestore
OPTIONS(
endpoint="http://vehicle-test.cn-hangzhou.vpc.tablestore.aliyuncs.com",
access.key.id="",
access.key.secret="",
instance.name="vehicle-test",
table.name="OrderBatchSink",
tunnel.id="",
catalog='{"columns": {"UserId": {"col": "UserId", "type": "string"}, "count": {"col": "count", "type":
"long"}, "totalPrice": {"col": "totalPrice", "type": "double"}}}'
);

DROP TABLE IF EXISTS order_totol_sink;
CREATE TABLE order_total_sink
USING tablestore
);
OPTIONS(
endpoint="http://vehicle-test.cn-hangzhou.vpc.tablestore.aliyuncs.com",
access.key.id="",
access.key.secret="",
instance.name="vehicle-test",
table.name="OrderTotalSink",
tunnel.id="",
catalog='{"columns": {"count": {"col": "count", "type": "long"}, "totalPrice": {"col": "totalPrice", "type":
"double"}}}'
);
```

运行以下批计算 SQL 进行用户维度聚合结果的更新。

```
// SQL 命令
INSERT INTO order_batch_sink SELECT UserId, count(*) AS count, sum(price) AS totalPrice FROM
order_source GROUP BY UserId;
// 实际运行
spark-sql> INSERT INTO order_batch_sink SELECT UserId, count(*) AS count, sum(price) AS totalPrice
FROM order_source GROUP BY UserId;
Time taken: 5.107 seconds
```

数据源: OrderBatchSink		表格数据最多显示50行。		
<input type="checkbox"/>	详细数据	UserId(主键)	count	totalPrice
<input type="checkbox"/>	详细数据	user_A	75688	376216.4799999986
<input type="checkbox"/>	详细数据	user_B	75475	377251.8799999907
<input type="checkbox"/>	详细数据	user_C	75249	376723.3000000127
<input type="checkbox"/>	详细数据	user_D	75634	379666.5100000123
<input type="checkbox"/>	详细数据	user_E	75404	375893.8099999992
<input type="checkbox"/>	详细数据	user_F	75950	379500.2100000018
<input type="checkbox"/>	详细数据	user_G	75931	379052.8499999999
<input type="checkbox"/>	详细数据	user_H	75766	378606.32000000036
<input type="checkbox"/>	详细数据	user_I	75943	379577.4199999996
<input type="checkbox"/>	详细数据	user_J	75754	379299.70000000094

运行以下批计算 SQL 进行总数据维度结果的更新。

```
// SQL 命令
INSERT INTO order_total_sink SELECT count(*) AS count, sum(price) AS totalPrice FROM order_source;
// 实际运行
spark-sql> INSERT INTO order_total_sink SELECT count(*) AS count, sum(price) AS totalPrice FROM
order_source;
Time taken: 4.272 seconds
```

数据源: OrderTotalSink		表格数据量多显示50行。	
<input type="checkbox"/>	详细数据	count(主键)	totalPrice
<input type="checkbox"/>	详细数据	14251626	7.126008712000002E7 ▼
<input type="checkbox"/>	详细数据	10468050	5.234753223000053E7 ▼
<input type="checkbox"/>	详细数据	8060201	4.030721644000016E7 ▼
<input type="checkbox"/>	详细数据	2967266	1.4834135849999965E7 ▼
<input type="checkbox"/>	详细数据	2248788	1.1244914189999966E7 ▼

写在最后

本文通过使用一套存储(Tablestore)和一套计算(Spark)完成了批流计算的有效结合, 更多有关批流一体的细节和干货可以参见 [Tablestore 结合 Spark 的云上流批一体大数据架构](#)。

对 Tablestore 有任何问题, 随时欢迎同我们进行交流, 钉钉群号: 11789671(1 群)、23307953(2 群)。

Tablestore+Delta Lake(快速开始)

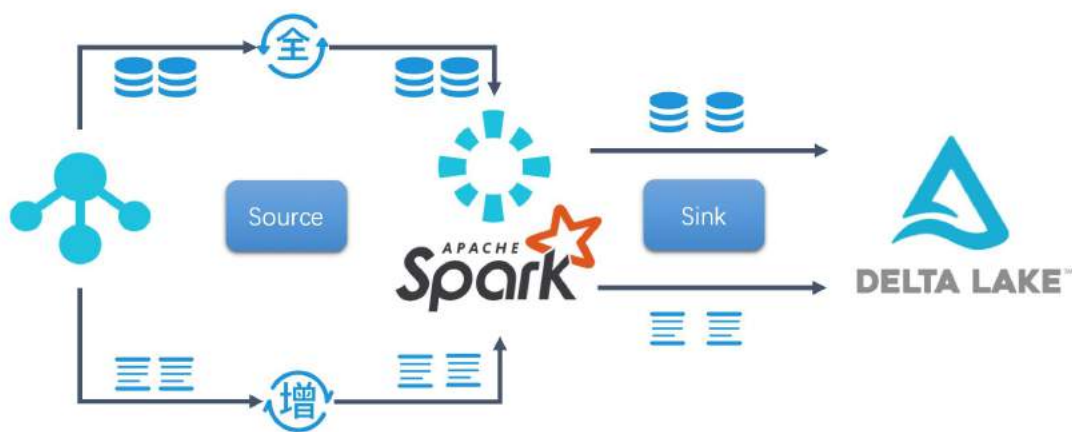
简介：本文介绍如何在 E-MapReduce 中通过 Tablestore Spark Streaming Source 将 TableStore 中的数据实时导入到 Delta Lake 中。

简介：本文介绍如何在 E-MapReduce 中通过 Tablestore Spark Streaming Source 将 TableStore 中的数据实时导入到 Delta Lake 中。

背景介绍

近些年来 HTAP(Hybrid transaction/analytical processing)的热度越来越高,通过将存储和计算组合起来,既能支持传统的海量结构化数据分析,又能支持快速的事务更新写入,是设计数据密集型系统的一个成熟的架构。

表格存储(Tablestore)是阿里云自研的 NoSQL 多模型数据库,提供海量结构化数据存储以及快速的查询和分析服务(PB 级存储、千万 TPS 以及毫秒级延迟),借助于表格存储的底层引擎,能够很好的完成 OLTP 场景下的需求。Delta Lake 类似于支持 Delta 的 Data Lake(数据湖),使用列存来存 base 数据,行的格式存储新增 delta 数据,进而做到支持数据操作的 ACID 和 CRUD,完全兼容 Spark 的大数据生态,通过结合 Delta Lake 和 Spark 生态,能够很好的完成 OLAP 场景下的需求。下图展示的是 Tablestore 和 Delta Lake 结合的 HATP 场景的一个简要的逻辑结构图,有关结构化大数据分析平台设计的更多细节和干货,可以参阅文章 [结构化大数据分析平台设计](#)。



准备工作

- 登录阿里云 E-MapReduce 控制台
- 创建 Hadoop 集群(若已创建, 请跳过)
- 确保将 Tablestore 实例部署在 E-MapReduce 集群相同的 VPC 环境下

步骤一 创建 Tablestore 源表

详细开通步骤请参考[官方文档](#), 本文 demo 中所创建出来的表名为 Source, 表的 Schema 如下图所示, 该表有 PKString 和 PkInt 两个主键, 类型分别为 String 和 Integer。

Source

基本详情

数据管理

触发器管理

数据监控

通道管理

索引管理

基本信息

数据表名称: Source [调整生命周期与最大版本](#)

数据生命周期: -1

最大数据版本: 1

数据有效版本偏差: 86400

最近一次调整时间: 2019-09-25 12:16:44

表格大小: 0 B

主键:

name	type	other
PkString	STRING	(分片键)
PkInt	INTEGER	

为表 Source 建立一个增量通道, 如下图所示, 通道列表里面会显示该通道的名字、ID 以及类型。



技术注解:

通道服务（Tunnel Service）是基于 Tablestore 数据接口之上的全增量一体化服务，包含三种通道类型：

- 全量：对数据表中历史存量数据消费处理
- 增量：对数据表中新增数据消费处理
- 全量加增量：先对数据表总历史存量数据消费，之后对新增数据消费

通道服务的详细介绍可查询 [Tablestore 官网文档](#)。

步骤二 获取相关 jar 包并上传到 hadoop 集群

- 获取环境依赖的 JAR 包。

Jar 包 | 获取方法 |

----- | ----- |

emr-tablestore-X.X.X.jarX.X.X: Since 1.9.0+ |Maven 库中下载：

<https://mvnrepository.com/artifact/com.aliyun.emr/emr-tablestore>

tablestore-X.X.X-jar-with-dependencies.jar | 下载 EMR SDK 相关的 Tablestore 依赖包。

<https://repo1.maven.org/maven2/com/aliyun/openservices/tablestore/5.3.0/tablestore-5.3.0-jar-with-dependencies.jar>

- 在**集群管理**页面，单击已创建的 Hadoop 集群的**集群 ID**，进入**集群与服务管理**页面。
- 在左侧导航树中选择**主机**列表，然后在右侧查看 Hadoop 集群中 **emr-header-1** 主机的 IP 信息。
- 在 SSH 客户端中新建一个命令窗口，登录 Hadoop 集群的 **emr-header-1** 主机。
- 上传所有 JAR 包到 emr-header-1 节点的某个目录下。

步骤三 运行 Spark Streaming 作业

1. 以一个基于 `emr demo` 修改的代码为样例，编译生成 JAR 包，JAR 包需要上传到 Hadoop 集群的 `emr-header-1` 主机中(参见步骤二)，完整的代码由于改动较大，不在本文中一一说明，后续会合到 `emr demo` 官方项目中。
2. 该样例以 Tablestore 表作为数据源，通过结合 Tablestore CDC 技术，Tablestore Streaming Source 和 Delta Sink，演示的是 TableStore 到 Delta Lake 的一个完整链路。
3. 按以下命令，启动 spark streaming 作业，开启一个实时同步 Tablestore Source 表中数据到 Delta Lake Table 的监听程序。

各个参数说明如下：

参数	参数说明
<code>com.aliyun.emr.example.spark.sql.streaming.DeltaTableStoreCDC</code>	所要运行的主程序类
<code>emr-tablestore-X.X.X-SNAPSHOT.jar</code>	包含 Tablestore source 的 jar 包
<code>tablestore-X.X.X-jar-with-dependencies.jar</code>	EMR SDK 相关的 Tablestore 依赖包
<code>examples-X.X.X-shaded.jar</code>	基于 EMR demo 修改的包(包含主程序类)
<code>instance</code>	Tablestore 实例名
<code>tableName</code>	Tablestore 表名
<code>tunnelId</code>	Tablestore 表的通道 Id
<code>accessKeyId</code>	Tablestore 的 <code>accessKeyId</code>
<code>accessKeySecret</code>	Tablestore 的秘钥
<code>endPoint</code>	Tablestore 实例的 <code>endPoint</code>
<code>maxOffsetsPerChannel</code>	Tablestore 通道 Channel 在每个 Spark Batch 周期内同步的最大数据条数，默认 10000。
<code>catalog</code>	同步的列名，详见 Catalog 字段说明

步骤四 数据 CRUD 示例

首先在 TableStore 里插入两行，本次示例中，我们建了 8 列的同步列，包括两个主键(PkString, PkInt)和六个属性列(col1, col2, col3, timestamp, col5 和 col6)。由于表格存储是 Free-Schema 的结构，我们可以任意的插入属性列，TableStore 的 Spark Source 会自动的做属性列的筛选。如下面两张图所示，在插入两行数据后，Delta Table 中同步也可以马上读取到两行，且数据一致。

表格数据

插入数据

查询数据

更新数据

删除数据

数据源: Source

表格数据最多显示50行。

<input type="checkbox"/>	详细数据	PkString(主键)	PkInt(主键)	col1	col2	col5	timestamp
<input type="checkbox"/>	详细数据	TestString	123456	hello emr	12345678		
<input type="checkbox"/>	详细数据	TestString2	123456			3.1415926	1569395874446
<input type="checkbox"/>							

共有2条, 每页显示: 10条

1

```
scala> val df = spark.read.format("delta").load("/delta/streamrecords")
df: org.apache.spark.sql.DataFrame = [PkString: string, PkInt: bigint ... 6 more fields]

scala> df.show()
+-----+-----+-----+-----+-----+-----+-----+
|PkString|PkInt|col1|col2|col3|timestamp|col5|col6|
+-----+-----+-----+-----+-----+-----+-----+

scala> df.show()
+-----+-----+-----+-----+-----+-----+-----+
| PkString| PkInt| col1| col2|col3| timestamp| col5|col6|
+-----+-----+-----+-----+-----+-----+-----+
| TestString|123456|hello emr|12345678|null| null| null|null|
|TestString2|123456| null| null|1732811406|3.1415926|null|
+-----+-----+-----+-----+-----+-----+-----+
```

接着, 在 Tablestore 中进行一些更新行和插入行的操作, 如下面的两个图所示, 等待一小段 micro-batch 的数据同步后, 表格存储中的数据同步变化能够即时的更新到 Delta Table 中。

表格数据

插入数据

查询数据

更新数据

删除数据

数据源: Source

表格数据最多显示50行。

<input type="checkbox"/>	详细数据	PkString(主键)	PkInt(主键)	col1	col2	col5	col6	timestamp
<input type="checkbox"/>	详细数据	TestString	123456	hello emr	12345678			
<input type="checkbox"/>	详细数据	TestString2	123456		1234567	2.22222222		1569395874446
<input type="checkbox"/>	详细数据	TestString3	654321		13579	6.666666	true	
<input type="checkbox"/>								

共有3条, 每页显示: 10条

1

```
scala> df.show()
+-----+-----+-----+-----+-----+-----+-----+
| PkString| PkInt| col1| col2|col3| timestamp| col5|col6|
+-----+-----+-----+-----+-----+-----+-----+
|TestString2|123456| null| 1234567|null|1732811406|2.22222222|null|
|TestString3|654321| null| 13579|null| null| 6.666666|true|
| TestString|123456|hello emr|12345678|null| null| null|null|
+-----+-----+-----+-----+-----+-----+-----+

scala>
```

将 Tablestore 中的数据全部清空，如下面两图所示，Delta Table 也同步的变成了空。



```
scala> df.show()
+-----+-----+-----+-----+-----+-----+-----+
| PkString|PkInt|col1|col2|col3|timestamp|col5|col6|
+-----+-----+-----+-----+-----+-----+-----+
|TestString2|123456|null|1234567|null|1732811406|2.22222222|null|
|TestString3|654321|null|13579|null|null|6.666666|true|
|TestString|123456|hello emr|12345678|null|null|null|null|
+-----+-----+-----+-----+-----+-----+-----+

scala> df.show()
+-----+-----+-----+-----+-----+-----+-----+
|PkString|PkInt|col1|col2|col3|timestamp|col5|col6|
+-----+-----+-----+-----+-----+-----+-----+

scala>
```

在集群上，Delta Table 默认存放在 HDFS 中，如下图所示，_delta_log 目录中存放的 json 文件是 Transaction log，parquet 格式的文件是底层的数据文件。

```
[root@emr-header-1 conf]# hdfs dfs -ls -h /delta/streamrecords
Found 13 items
drwxr-x--x - root hadoop 0 2019-09-25 15:40 /delta/streamrecords/_delta_log
-rw-r----- 2 root hadoop 826 2019-09-25 15:18 /delta/streamrecords/part-00000-044b1ce3-e602-423a-a443-90f72ea14416-c000.snappy.parquet
-rw-r----- 2 root hadoop 826 2019-09-25 15:18 /delta/streamrecords/part-00000-09412853-16ac-4045-8e43-04f25dcab226-c000.snappy.parquet
-rw-r----- 2 root hadoop 826 2019-09-25 15:12 /delta/streamrecords/part-00000-1ada679c-da16-45fc-a25d-34eeec68951-c000.snappy.parquet
-rw-r----- 2 root hadoop 826 2019-09-25 15:12 /delta/streamrecords/part-00000-4309805c-c93c-47de-80fe-2d0809cf6d8f-c000.snappy.parquet
-rw-r----- 2 root hadoop 826 2019-09-25 15:17 /delta/streamrecords/part-00000-4450738-6053-4647-e283-0e893f05230f-c000.snappy.parquet
-rw-r----- 2 root hadoop 826 2019-09-25 15:37 /delta/streamrecords/part-00000-86dd0f97-39a9-4018-87ad-65b39a9d7ee9-c000.snappy.parquet
-rw-r----- 2 root hadoop 826 2019-09-25 15:38 /delta/streamrecords/part-00000-f0fc1942-e855-4b13-bf73-90c8bfa6787-c000.snappy.parquet
-rw-r----- 2 root hadoop 826 2019-09-25 15:40 /delta/streamrecords/part-00000-f6f6b928-8d4c-4445-930b-9304720d2472-c000.snappy.parquet
-rw-r----- 2 root hadoop 1.7 K 2019-09-25 15:18 /delta/streamrecords/part-00072-3b3d839d-02c8-43cb-9cab-fee706afeca-c000.snappy.parquet
-rw-r----- 2 root hadoop 1.8 K 2019-09-25 15:38 /delta/streamrecords/part-00072-6915cd74-bb46-4e96-ab4c-a77b813ce0e9-c000.snappy.parquet
-rw-r----- 2 root hadoop 1.8 K 2019-09-25 15:37 /delta/streamrecords/part-00121-17679369-f81f-4caf-8a52-bf7a0fd0491-c000.snappy.parquet
-rw-r----- 2 root hadoop 1.7 K 2019-09-25 15:17 /delta/streamrecords/part-00121-fed12613-87c2-490a-9bca-683dd11e7c13-c000.snappy.parquet

[root@emr-header-1 conf]# hdfs dfs -ls -h /delta/streamrecords/_delta_log
Found 0 items
-rw-r----- 2 root hadoop 1.2 K 2019-09-25 15:12 /delta/streamrecords/_delta_log/00000000000000000000.json
-rw-r----- 2 root hadoop 589 2019-09-25 15:12 /delta/streamrecords/_delta_log/00000000000000000001.json
-rw-r----- 2 root hadoop 751 2019-09-25 15:17 /delta/streamrecords/_delta_log/00000000000000000002.json
-rw-r----- 2 root hadoop 751 2019-09-25 15:18 /delta/streamrecords/_delta_log/00000000000000000003.json
-rw-r----- 2 root hadoop 589 2019-09-25 15:18 /delta/streamrecords/_delta_log/00000000000000000004.json
-rw-r----- 2 root hadoop 751 2019-09-25 15:37 /delta/streamrecords/_delta_log/00000000000000000005.json
-rw-r----- 2 root hadoop 893 2019-09-25 15:38 /delta/streamrecords/_delta_log/00000000000000000006.json
-rw-r----- 2 root hadoop 1006 2019-09-25 15:40 /delta/streamrecords/_delta_log/00000000000000000007.json

[root@emr-header-1 conf]#
```



更多精彩内容扫码关注
Spark 中文社区微信公众号



钉钉扫码加入
Spark 中文社区钉钉群



阿里云开发者“藏经阁”
海量电子书免费下载