个推
GeTui

个推技术实践系列白皮书

大数据 BIG DATA 降本提效

实战手册

每日互动股份有限公司

股票代码：300766

序言

董霖

每日互动首席数据官

每日互动践行“数据让产业更智能”的使命，致力于用数据为互联网、品牌广告、交通旅游等行业降本提效，为行业带来全新的生态和激动人心的未来。与此同时，大数据技术自身的降本提效，也同样能释放出可观的存储和算力资源，让更海量的数据和更强大的算法融合起来产生更绚丽的化学反应。

这，不是内卷！

监制：叶新江、董霖

出品：袁凯、孟显耀、段永康

总编辑：章玉珍

责任编辑：张瑶

编辑：王雯、郑立群、孙慧丽、罗斯琦

美术编辑：许箫、吴昊澜、高路路

索阅、投稿、建议和意见反馈，请联系每日互动股份有限公司 市场部

Email：tech@getui.com

地址：杭州市西湖区西斗门路7号千岛湖智谷大厦A座14楼

电话：4006-808-606

目录 CONTENTS

个推大数据降本提效实战

出品团队

关于我们

Data Intelligence

Spark性能调优实践：性能提升60%↑ 成本降低50%↓

个推采用Hive元数据管理+Spark计算引擎的大数据架构，以支撑自身大数据业务发展，并将Spark广泛应用到报表分析、机器学习等场景中，为行业客户和政府部门提供实时人口洞察、群体画像构建等服务。

本文解读了Spark的底层原理，梳理standalone、Yarn-client、Yarn-cluster等3种常见任务提交方式；并从存储格式、数据倾斜、参数配置等3个方面展开，为大家分享个推进行Spark性能调优的进阶姿势。

实战一：Spark性能调优实践

前言

Spark是目前主流的大数据计算引擎，功能涵盖了大数据领域的离线批处理、SQL类处理、流式/实时计算、机器学习、图计算等各种不同类型的计算操作，应用范围与前景非常广泛。作为一种**内存计算框架**，Spark**运算速度快**，并能够满足UDF、大小表Join、多路输出等多样化的数据计算和处理需求。

作为国内专业的数据智能服务商，**个推从早期的1.3版本便引入Spark**，并基于Spark建设数仓，进行大规模数据的离线和实时计算。由于Spark在2.x版本之前的优化重心在计算引擎方面，而在元数据管理方面并未做重大改进和升级。因此个推仍然使用Hive进行元数据管理，采用**Hive元数据管理+ Spark计算引擎**的大数据架构，以支撑自身大数据业务发展。个推还将Spark广泛应用到报表分析、机器学习等场景中，为行业客户和政府部门**提供实时人口洞察、群体画像构建**等服务。

存储格式	操作	SQL引擎	耗时/s	SQL引擎	耗时/s	提升倍数
parquet+gzip	pv-count(1)	sparksql2.3	159	hive1.2(mr)	29734	187
orc+snappy	pv-count(1)		2340		6198	2
orc+zlib	pv-count(1)		1973		2883	1
parquet+gzip	带条件的pv		3511		17772	5
orc+snappy	带条件的pv		4338		8339	1
orc+zlib	带条件的pv		4553		4867	1
parquet+gzip	uv-去重gid		2544.2		16971	6
orc+snappy	uv-去重gid		4716.5		13552	2
orc+zlib	uv-去重gid		4595		9728	2
parquet+gzip	点查-（匹配条件较多）		3960		19671	4
orc+snappy	点查-（匹配条件较多）		6840		6481	0
orc+zlib	点查-（匹配条件较多）		7012		6322	0
parquet+gzip	点查-（单个匹配条件）		4103		14910	3
orc+snappy	点查-（单个匹配条件）		6009		3998	0
orc+zlib	点查-（单个匹配条件）		9317		2966	0
parquet+gzip	抽样后，和大表join		148		8203	55
orc+snappy	抽样后，和大表join		272		3747	13
orc+zlib	抽样后，和大表join		147		2551	17

▲个推在实际业务场景中，分别使用SparkSQL 和 HiveSQL对一份3T数据进行了计算，上图展示了跑数速度。数据显示：在锁死队列（120G内存，<50core）前提下， SparkSQL2.3 的计算速度是Hive1.2 的5-10倍。

实战一：Spark性能调优实践

对企业来讲，效率和成本始终是其进行海量数据处理和计算时所必须关注的问题。如何充分发挥Spark的优势，在进行大数据作业时真正实现降本增效呢？个推将多年积累的**Spark性能调优妙招**进行了总结，与大家分享。

Spark性能调优-基础篇

众所周知，正确的参数配置对提升Spark的使用效率具有极大助力。因此，**针对不了解底层原理的Spark使用者，我们提供了可以直接抄作业的参数配置模板**，帮助相关数据开发、分析人员更高效地使用Spark进行离线批处理和SQL报表分析等作业。

推荐参数配置模板如下：

1.spark-submit 提交方式脚本

```
/xxx/spark23/xxx/spark-submit --master yarn-cluster  \
--name ${mainClassName} \
--conf spark.serializer=org.apache.spark.serializer.KryoSerializer \
--conf spark.yarn.maxAppAttempts=2 \
--conf spark.executor.extraJavaOptions=-XX:+UseConcMarkSweepGC \
--driver-memory 2g \
--conf spark.sql.shuffle.partitions=1000 \
--conf hive.metastore.schema.verification=false \
--conf spark.sql.catalogImplementation=hive \
--conf spark.sql.warehouse.dir=${warehouse} \
--conf spark.sql.hive.manageFilesourcePartitions=false \
--conf hive.metastore.try.direct.sql=true \
--conf spark.executor.memoryOverhead=512M \
--conf spark.yarn.executor.memoryOverhead=512 \
--executor-cores 2 \
--executor-memory 4g \
--num-executors 50 \
--class 启动类 \
${jarPath} \
-M ${mainClassName}
```

实战一：Spark性能调优实践

2.spark-sql 提交方式脚本

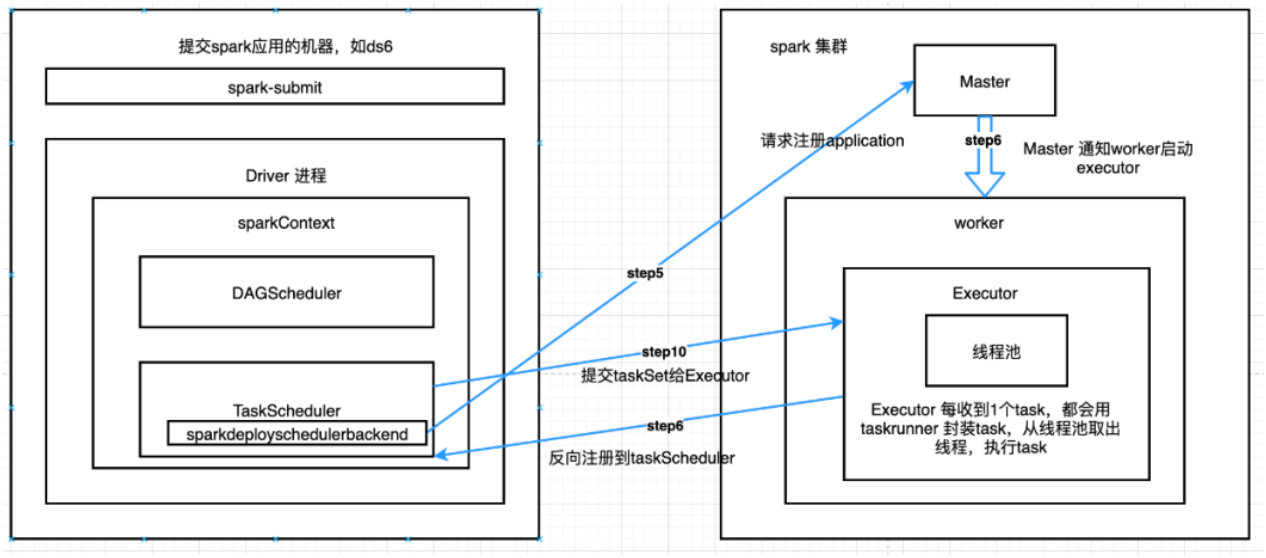
```
option=/xxx/spark23/xxx/spark-sql
export SPARK_MAJOR_VERSION=2
${option} --master yarn-client \
--driver-memory 1G \
--executor-memory 4G \
--executor-cores 2 \
--num-executors 50 \
--conf "spark.driver.extraJavaOptions=-Dlog4j.configuration=file:log4j.properties" \
--conf spark.sql.hive.caseSensitiveInferenceMode=NEVER_INFER \
--conf spark.sql.auto.repartition=true \
--conf spark.sql.autoBroadcastJoinThreshold=104857600 \
--conf "spark.sql.hive.metastore.try.direct.sql=true" \
--conf spark.dynamicAllocation.enabled=true \
--conf spark.dynamicAllocation.minExecutors=1 \
--conf spark.dynamicAllocation.maxExecutors=200 \
--conf spark.dynamicAllocation.executorIdleTimeout=10m \
--conf spark.port.maxRetries=300 \
--conf spark.executor.memoryOverhead=512M \
--conf spark.yarn.executor.memoryOverhead=512 \
--conf spark.sql.shuffle.partitions=10000 \
--conf spark.sql.adaptive.enabled=true \
--conf spark.sql.adaptive.shuffle.targetPostShuffleInputSize=134217728 \
--conf spark.sql.parquet.compression.codec=gzip \
--conf spark.sql.orc.compression.codec=zlib \
--conf spark.ui.showConsoleProgress=true
-f pro.sql

pro.sql 为业务逻辑脚本
```

实战一：Spark性能调优实践

Spark性能调优-进阶篇

针对有意愿了解Spark底层原理的读者，本文梳理了standalone、Yarn-client、Yarn-cluster等3种常见任务提交方式的交互图，以帮助相关使用者更直观地理解Spark的核心技术原理、为阅读接下来的进阶篇内容打好基础。

standalone



1）spark-submit 提交，通过反射的方式构造出1个DriverActor 进程；

2）Driver进程执行编写的application，构造sparkConf，构造sparkContext；

3）SparkContext在初始化时，构造DAGScheduler、TaskScheduler，jetty启动webui；

4）TaskScheduler 有sparkdeployschedulebackend 进程，去和Master通信，请求注册Application；

5）Master 接受通信后，注册Application，使用资源调度算法，通知Worker，让worker启动Executor；

6）worker会为该application 启动executor，executor 启动后，会反向注册到TaskScheduler；

7）所有Executor 反向注册到TaskScheduler 后，Driver 结束sparkContext 的初始化；

8）Driver继续往下执行编写的application，每执行到1个action，就会创建1个job；

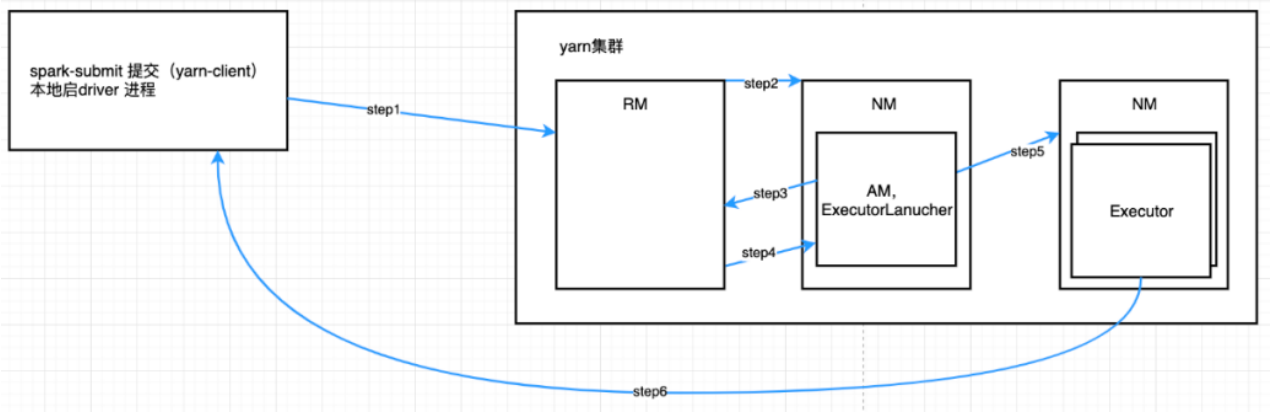
9）job 会被提交给DAGScheduler，DAGScheduler 会对job 划分为多个stage（stage划分算法），每个stage创建1个taskSet；

10）taskScheduler会把taskSet里每1个task 都提交到executor 上执行（task 分配算法）；

11）Executor 每接受到1个task，都会用taskRunner来封装task，之后从executor 的线程池中取出1个线程，来执行这个taskRunner。（task runner：把编写的代码/算子/函数拷贝，反序列化，然后执行task）。

实战一：Spark性能调优实践

Yarn-client



1） 发送请求到ResourceManager（RM），请求启动ApplicationMaster（AM）；

2） RM 分配container 在某个NodeManager（NM）上，启动AM，实际是个ExecutorLauncher;

3） AM向RM 申请container;

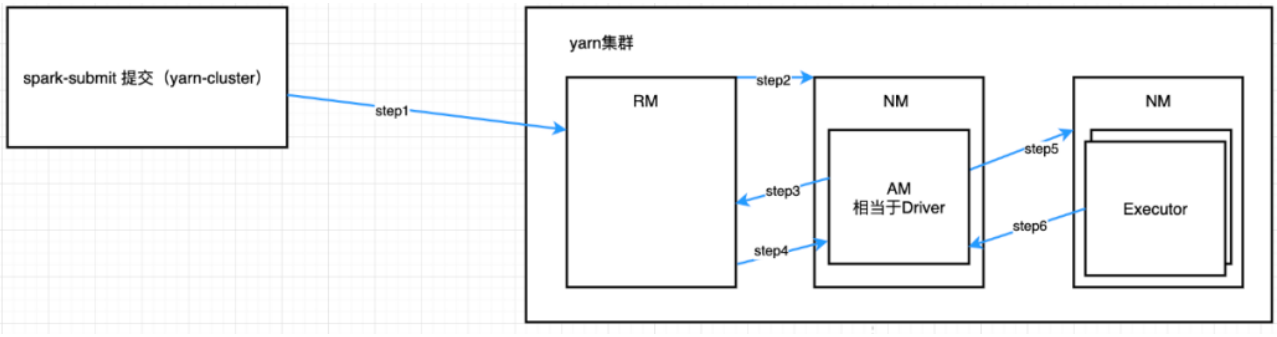
4） RM给AM 分配container;

5） AM 请求NM 来启动相应的Executor;

6） executor 启动后，反向注册到Driver进程;

7） 后序划分stage，提交taskset 和standalone 模式类似。

Yarn-cluster



1） 发送请求到ResourceManager（RM），请求启动ApplicationMaster（AM）；

2） RM 分配container 在某个NodeManager（NM）上，启动AM;

3） AM向RM 申请container;

4） RM给AM 分配container;

5） AM 请求NM 来启动相应的Executor;

实战一：Spark性能调优实践

6） executor 启动后，反向注册到AM;

7） 后序划分stage，提交taskset 和standalone 模式类似。

理解了以上3种常见任务的底层交互后，接下来本文从**存储格式、数据倾斜、参数配置**等3个方面来展开，为大家分享个推进行Spark性能调优的进阶姿势。

存储格式（文件格式、压缩算法）

众所周知，不同的SQL引擎在不同的存储格式上，其优化方式也不同，比如Hive更倾向于orc，Spark则更倾向于parquet。同时，在进行大数据作业时，**点查、宽表查询、大表join操作相对频繁**，这就要求文件格式最好采用**列式存储，并且可分割**。因此我们推荐以parquet、orc为主的列式存储文件格式和以gzip、snappy、zlib 为主的压缩算法。在组合方式上，我们建议使用**parquet+gzip、orc+zlib**的组合方式，这样的组合方式**兼顾了列式存储与可分割**的情况，相比txt+gz这种行式存储且不可分割的组合方式更能够适应以上大数据场景的需求。

个推以线上500G左右的数据为例，在不同的集群环境与SQL引擎下，对不同的存储文件格式和算法组合进行了性能测试。测试数据表明：**相同资源条件下，parquet+gz存储格式较text+gz存储格式在多值查询、多表join上提速至少在60%以上。**

结合测试结果，我们对不同的集群环境与SQL引擎下所推荐使用的存储格式进行了梳理，如下表：

HDP环境	SQL引擎	推荐使用
2.7.x	hive 1.2	1 orc+zlib 2 parquet+gz
2.7.x	spark 2.3	1 parquet+gz 2 orc+zlib
3.x	hive 3	1 orc+zlib 2 parquet+gz
3.x	spark 2.3	1 parquet+gz 2 orc+zlib

同时，我们也对parquet+gz、orc+zlib的内存消耗进行了测试。**以某表的单个历史分区数据为例，parquet+gz、orc+zlib比txt+gz 分别节省26%和49%的存储空间。**

实战一：Spark性能调优实践

完整测试结果如下表：

序号	数据	分区	存储格式	集群环境	数据如何生产	单副本大小	基准	压缩
1	某表	某个历史分区	parquet+gz	Hadoop2.7.x	MR	3.9T	5.3T	74%
2			orc+snappy		sparksql2.3	3.6T		68%
3			text+gz		MR	5.3T		100%
4			orc+zlib		sparksql2.3	2.7T		51%

可见，parquet+gz、orc+zlib确实在降本提效方面效果显著。**那么，如何使用这两种存储格式呢？步骤如下：**

➤ Hive 与 Spark 开启指定文件格式的压缩算法

```
spark:
set spark.sql.parquet.compression.codec=gzip;
set spark.sql.orc.compression.codec=zlib;

hive:
set hive.exec.compress.output=true;
set mapreduce.output.fileoutputformat.compress=true;
set mapreduce.output.fileoutputformat.compress.codec=org.apache.hadoop.io.compress.GzipCodec;
```

➤ 建表时指定文件格式

```
parquet 文件格式(序列化,输入输出类)
CREATE EXTERNAL TABLE `test`(rand_num double)
PARTITIONED BY (`day` int)
ROW FORMAT SERDE
'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
STORED AS INPUTFORMAT
'org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat'
;


orc 文件格式(序列化,输入输出类)
ROW FORMAT SERDE
```

实战一：Spark性能调优实践

```
'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
STORED AS INPUTFORMAT
'org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat'
;
```

➤ 线上表调

```
ALTER TABLE db1.table1_std SET TBLPROPERTIES ('parquet.compression'='gzip');
ALTER TABLE db2.table2_std SET TBLPROPERTIES ('orc.compression'='ZLIB');
```

➤ ctas 建表

```
create table tablename stored as parquet as select ......;
create table tablename stored as orc TBLPROPERTIES ('orc.compress'='ZLIB')  as select ......;
```

数据倾斜

数据倾斜分为map倾斜和reduce倾斜两种情况。本文着重介绍reduce 倾斜，如SQL 中常见的group by、join 等都可能是其重灾区。数据倾斜发生时，一般表现为：部分task显著慢于同批task，task 数据量显著大于其他task，部分taskOOM、spark shuffle 文件丢失等。如下图示例，在duration 列和shuffleReadSize/Records列，我们能明显发现部分task处理数据量显著升高，耗时变长，造成了数据倾斜：

Tasks

Page: 1 2 >　　2 Pages. Jump to

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Shuffle Read Size / Records
0	4035	0	SUCCESS	PROCESS_LOCAL	123 / -ds88	2021/08/16 10:06:21	3.2 min	0.3 s	50 MB / 1384444
1	4036	0	SUCCESS	PROCESS_LOCAL	87 / -ds17	2021/08/16 10:06:21	1 s		523.8 KB / 1428
2	4037	0	SUCCESS	PROCESS_LOCAL	76 / -ds7	2021/08/16 10:06:21	0.5 s		550.7 KB / 1496
3	4038	0	SUCCESS	PROCESS_LOCAL	45 / -ds19	2021/08/16 10:06:21	2 s	28 ms	539.4 KB / 1473

实战一：Spark性能调优实践

如何解决数据倾斜？

我们总结了7种数据倾斜解决方案，能够帮助大家解决常见的数据倾斜问题：

解决方案一：使用 Hive ETL预处理数据

即在数据血缘关系中，把倾斜问题前移处理，从而使下游使用方无需再考虑数据倾斜问题。

▸ 该方案适用于下游交互性强的业务，如秒级/分钟级别提数查询。

解决方案二：过滤少数导致倾斜的key

即剔除倾斜的大key，该方案一般结合百分位点使用，如99.99%的id记录数为100条以内，那么100条以外的id 就可考虑予以剔除。

▸该方案在统计型场景下较为实用，而在明细场景下，需要看过滤的大key 是否为业务所侧重和关注。

解决方案三：提高shuffle操作的并行度

即对spark.sql.shuffle.partitions参数进行动态调整，通过增加shuffle write task写出的partition数量，来达到key的均匀分配。SparkSQL2.3 在默认情况下，该值为200。开发人员可以在启动脚本增加如下参数，对该值进行动态调整：

```
conf spark.sql.shuffle.partitions=10000
conf spark.sql.adaptive.enabled=true
conf spark.sql.adaptive.shuffle.targetPostShuffleInputSize=134217728
```

▸该方案非常简单，但是对于key的均匀分配却能起到较好的优化作用。比如，原本10个key，每个50条记录，只有1个partition，那么后续的task需要处理500条记录。通过增加partition数量，可以使每个task 都处理50条记录，10个task 并行跑数，耗时只需要原来1个task 的1/10。但是该方案对于大key较难优化，比如，某个大key记录数有百万条，那么大key 还是会被分配到1个task 中去。

解决方案四：将reducejoin转为mapjoin

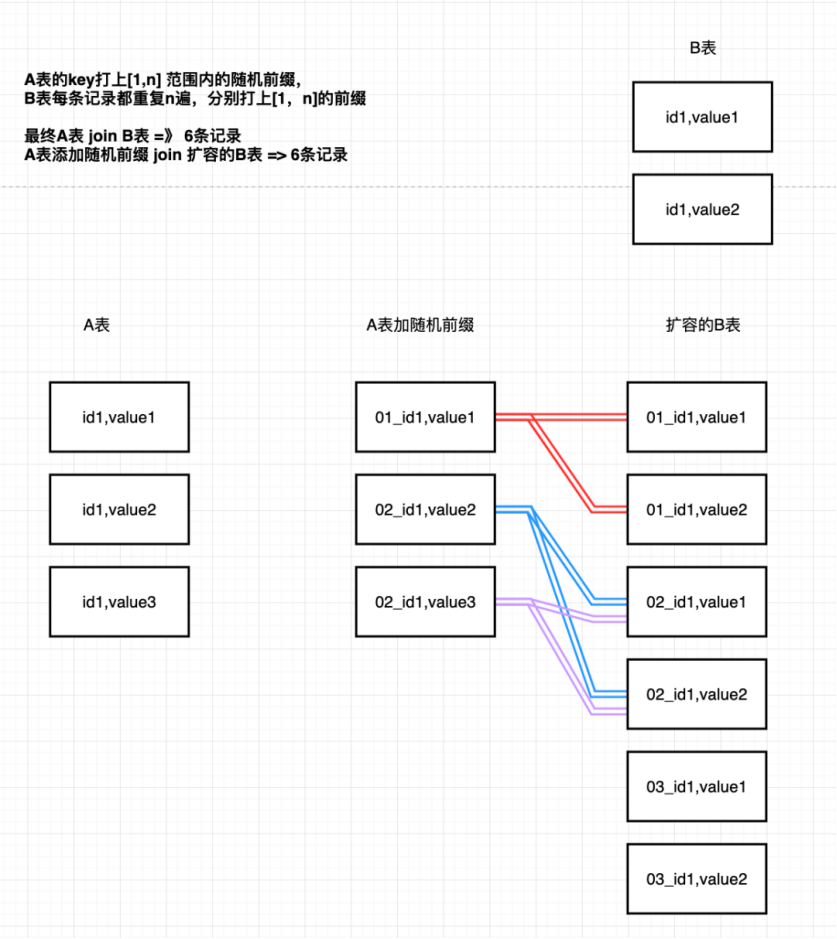
指的是在map端join，不走shuffle过程。以Spark为例，可以通过广播变量的形式，将小RDD的数据下发到各个Worker节点上（Yarn 模式下是NM），在各个Worker节点上进行join。

▸该方案适用于小表join大表场景（百G以上的数据体量）。此处的小表默认阈值为10M，低于此阈值的小表，可分发到worker节点。具体可调整的上限需要小于container分配的内存。

实战一：Spark性能调优实践

解决方案五：采样倾斜key并分拆join操作

如下图示例：A表 join B表，A表有大key、B表无大key，其中大key的id为1，有3条记录。



如何进行分拆join操作呢？

- 首先将A表、B表中id1单独拆分出来，剔除大key的A' 和 B' 先join，达到非倾斜的速度；
- 针对A表大key添加随机前缀，B表扩容N倍，单独join；join后剔除随机前缀即可；
- 再对以上2部分union。

▸ 该方案的本质还是减少单个task 处理过多数据时所引发的数据倾斜风险，适用于大key较少的情况。

解决方案六：使用随机前缀和扩容RDD进行join

比如，A 表 join B表，以A表有大key、B表无大key为例：

- 对A表每条记录打上[1,n] 的随机前缀，B表扩容N倍，join。
- join完成后剔除随机前缀。

▸ 该方案适用于大key较多的情况，但也会增加资源消耗。

实战一：Spark性能调优实践

解决方案七：combiner

即在map端做combiner操作，减少shuffle拉取的数据量。

- 该方案适合累加求和等场景。

在实际场景中，建议相关开发人员具体情况具体分析，针对复杂问题也可将以上方法进行组合使用。

Spark 参数配置

针对无数据倾斜的情况，我们梳理总结了参数配置参照表帮助大家进行Spark性能调优，这些参数的设置适用于2T左右数据的洞察与应用，基本满足大多数场景下的调优需求。

--executor-memory 4G \	spark executor 运行所需内存。
--executor-cores 2 \	每个 executor 可以并发运行的 task 数。 spark 任务并发量=executor_cores * num_executors 如果 task > 1，每个 task 的内存=executor_memory/executor_cores。
--num-executors 50 \	executor 数量。
--conf spark.sql.autoBroadcastJoinThreshold=104857600 \	大小表 join 时，小表 map join 上限，目前设置 100M，可根据机器内存调大。小表会被广播到各个 Woker 节点上。
--conf spark.dynamicAllocation.enabled=true \ --conf spark.dynamicAllocation.minExecutors=1 \ --conf spark.dynamicAllocation.maxExecutors=200 \	动态调整 spark-sql executor 数量(1~200 个 executor)。特别适用于在终端节点上启动，不运行任务时，会只在后台常驻 1 个 executor。
--conf spark.executor.memoryOverhead=512M \ --conf spark.yarn.executor.memoryOverhead=512 \	堆外内存，建议开启。 和 shuffle 有关，序列化成二进制的 record，在堆外排序,spark NIO 拉取数据时也会先放到堆外。 典型报错场景：shuffle file cannot find，executor lost、task lost，out of memory
--conf spark.sql.shuffle.partitions=10000 \ --conf spark.sql.adaptive.enabled=true \ --conf spark.sql.adaptive.shuffle.targetPostShuffleInputSize=134217728 \	通过增加 shuffle write task 写出的 partition 的数量，来达到 key 的均匀分配，sparksql2.3 默认情况为 200，实际可以在启动脚本增加参数来动态调整。
--conf spark.sql.parquet.compression.codec=gzip \	paruqet 文件格式启用 gzip 压缩。
--conf spark.sql.orc.compression.codec=zlib \	orc 文件格式启用 zlib 压缩。
--conf spark.ui.showConsoleProgress=true	查看 stage，task 进度。

总结

目前，Spark已经发展到了Spark3.x，最新版本为Spark 3.1.2 released (Jun 01, 2021)。Spark3.x的许多新特性，如动态分区修剪、Pandas API的重大改进、增强嵌套列的裁剪和下推等亮点功能，为进一步实现降本增效提供了好思路。未来，个推也将继续保持对Spark演进的关注，并持续展开实践和分享。

基于Flink SQL 建设实时数仓实践

个推使用SQL模式进行Flink作业，SQL模式虽然更为简单、通用性也更强，但是在性能调优方面，却存在较大难度。

本文将个推Flink SQL使用和调优经验进行了总结，从中间表注册入手，分享Flink SQL正确使用姿势，帮助大家在使用Flink SQL进行实时计算作业时少走弯路、提升效率。

实战二：基于Flink SQL建设实时数仓实践

前言

作为一家数据智能企业，个推在服务垂直行业客户的过程中，会涉及到很多数据实时计算和分析的场景，比如在服务开发者时，需要对App消息推送的下发数、到达数、打开率等后效数据进行实时统计；在服务政府单位时，需要对区域内实时人口进行统计和画像分析。为了更好地支撑大数据业务发展，个推也建设了自己的实时数仓。相比Storm、Spark等实时处理框架，Flink不仅具有高吞吐、低延迟等特性，同时还支持**精确一次语义（exactly once）、状态存储**等特性，拥有很好的**容错**机制，且**使用门槛低、易上手、开发难度小**。因此，个推主要基于Flink SQL来解决大部分的实时作业需求。

目前个推主要使用3种方式进行Flink作业：Zeppelin模式、Jar模式和SQL模式。相比另外两种模式，使用SQL模式进行Flink作业，虽然更为简单、通用性也更强，但是在性能调优方面，却存在较大难度。**本文将个推Flink SQL使用和调优经验进行了总结，旨在帮助大家在基于Flink SQL进行实时计算作业时少走弯路、提升效率。**

个推Flink SQL使用现状

在SQL模式下，个推通过**jar+SQL文件+配置参数**的方式使用Flink。其中jar是基于Flink封装的执行SQL文件的执行jar，提交命令示例如下：

```
/opt/flink/bin/flink run -m yarn-cluster -ynm KafkaSourceHbaseSinkCaseTestSql \
-c ${mainClassName} \
${jarPath} \
--flink.parallelism 40 \
--mode stream \
--sql.file.path ${sqlFile}
```

SQL文件内容示例如下：

```
create table kafka_table(
 ts  bigint,
 username string,
 num bigint
) with (
 'connector' = 'kafka',
```

实战二：基于Flink SQL建设实时数仓实践

```
7   'topic' = 'test',
8    'properties.bootstrap.servers' = ' ',
9    'properties.group.id' = 'test-consumer001',
10   'scan.startup.mode' = 'latest-offset',
11   'format' = 'csv'
12  )
13
14  create table sink_table(
15   ts  bigint,
16   username string,
17   num bigint
18  ) with (
19   'connector' = 'kafka',
20   'topic' = 'test2',
21   'properties.bootstrap.servers' = ' ',
22   'format' = 'json'
23  )
21
25  insert into sink_table select * from kafka_table
```

在将原有的Spark Streaming实时计算任务改造成SQL的过程中，我们发现了许多原生Flink SQL无法支持的需求，比如：

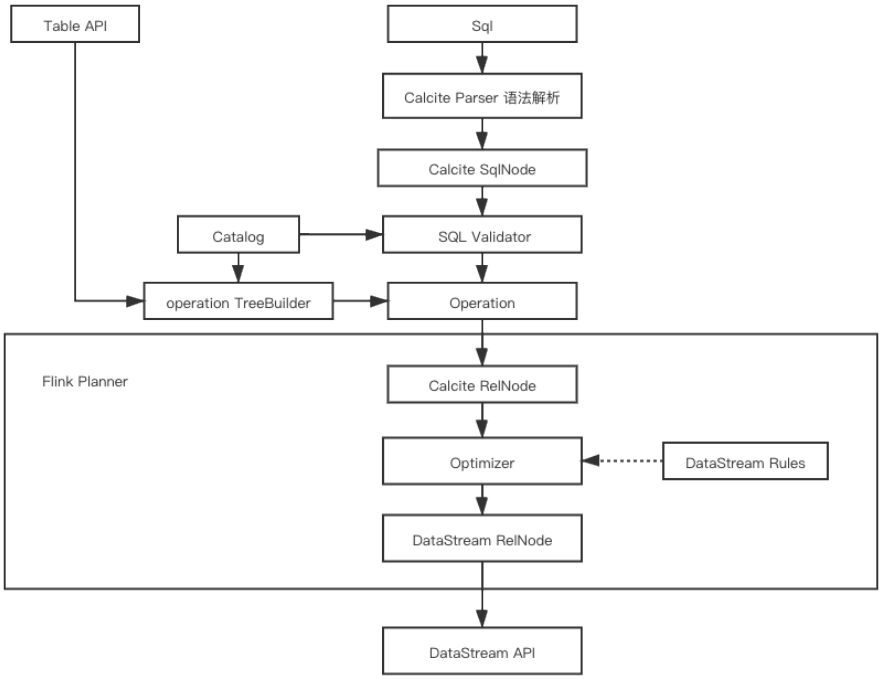
- **写hbase指定时间戳**：原生Flink SQL写hbase的时间戳无法由数据时间指定。
- **写hbase支持数据字段指定qualifier**：原生Flink SQL注册hbase表时就需要指定qualifier，无法使用数据字段的值作为qualifier。
- **中间表注册**：目前注册表只能调用table api实现。
- **kafka source数据预截断**：由于业务原因，部分数据源写入kafka的数据默认增加指定前缀，解析前需要预截断。
- **kafka schema不匹配**：由于业务原因，上游写入csv格式数据前会追加字段，导致和schema不匹配，数据无法解析。

针对以上大部分场景，我们均结合业务特色需求，对Flink SQL进行了拓展适配。**本文从中间表注册入手，分享Flink SQL正确使用姿势。**

实战二：基于Flink SQL建设实时数仓实践

Flink中SQL的处理流程

为了帮助大家更好地理解中间表注册问题，我们先整体梳理下Flink中SQL的执行逻辑，如下图：



整个流程可以大致拆解为以下几个步骤：

1、SqlParser解析阶段(SQL -> SqlNode)

Flink的Calcite使用JavaCC，根据Parser.jj生成SqlParser(实际类名为SqlParserImpl)。SqlParser负责将SQL解析为AST语法树，数据类型为SqlNode。

2、Validator验证阶段

第一阶段后生成的AST树中，对字段、函数等并没有进行验证。第二阶段会进行校验，校验内容包括表名、字段名、函数名、数据类型等。

3、逻辑计划(SqlNode -> RelNode/RexNode)

经过语法校验的AST树经过SqlToRelConverter.convertQuery调用，将SQL转换为RelNode，即生成逻辑计划LogicalPlan。需要注意的是，Flink为了统一“table api”和“sql执行”两种方式，会在这个阶段将RelNode封装成Operation。

实战二：基于Flink SQL建设实时数仓实践

4、优化器(RelNode -> LogicalNode -> ExecNode)

优化器的作用是将关系代数表达式(RelNode)转换为执行计划，用于执行引擎执行。优化器会使用过滤条件的下压、列裁剪等常见的优化规则进行优化，以生成更高效的执行计划。

Flink主要使用Calcite的优化器，采用HepPlanner和VolcanoPlanner这两种优化方式进行优化。

- **HepPlanner**: 是基于规则优化（RBO）的实现，它是一个启发式的优化器，按照规则进行匹配，直到达到次数限制（match 次数限制）或者遍历一遍后不再出现rule match的情况才算完成。
- **VolcanoPlanner**: 是基于成本优化（CBO）的实现，它会一直迭代rules，直到找到cost最小的plan。

需要注意的是，在调用规则优化前，Flink会有一个内部的**CommonSubGraphBasedOptimizer优化器**用于提取多个执行计划的共用逻辑。CommonSubGraphBasedOptimizer是Flink对于多流场景（常见为多个insert）的优化器，主要作用是**提取共用的逻辑，生成有向无环图，避免对共用逻辑进行重复计算。**

根据运行环境不同（批和流式），CommonSubGraphBasedOptimizer优化器有BatchCommonSubGraphBasedOptimizer和 StreamCommonSubGraphBasedOptimizer两种实现方式。从执行结果来看，**CommonSubGraphBasedOptimizer优化类似于Spark表的物化，最终目的都是避免数据重复计算。**

源码中RelNodeBlock类注释很形象地描述了优化效果：

```
1  * {{{-
2  *  val sourceTable = tEnv.scan("test_table").select('a, 'b, 'c)
3  *  val leftTable = sourceTable.filter('a > 0).select('a as 'a1, 'b as 'b1)
4  *  val rightTable = sourceTable.filter('c.isNotNull).select('b as 'b2, 'c as 'c2)
5  *  val joinTable = leftTable.join(rightTable, 'a1 === 'b2)
6  *  joinTable.where('a1 >= 70).select('a1, 'b1).writeToSink(sink1)
7  *  joinTable.where('a1 < 70 ).select('a1, 'c2).writeToSink(sink2)
8  * }}}
9  *
10 * the RelNode DAG is:
```

实战二：基于Flink SQL建设实时数仓实践

```
11  *
12  * {{{-
13  * Sink(sink1)    Sink(sink2)
14  *   |            |
15  * Project(a1,b1)  Project(a1,c2)
16  *   |            |
17  * Filter(a1>=70)  Filter(a1<70)
18  *     \        /
19  *      Join(a1=b2)
20  *     /        \
21  * Project(a1,b1)  Project(b2,c2)
22  *     |        |
23  * Filter(a>0)    Filter(c is not null)
24  *     \        /
25  *     Project(a,b,c)
26  *       |
27  *     TableScan
28  * }}}
29  * This [[RelNode]] DAG will be decomposed into three [[RelNodeBlock]]s, the break-point
30  * is the [[RelNode]](`Join(a1=b2)`) which data outputs to multiple [[LegacySink]]s.
31  * <p>Notes: Although `Project(a,b,c)` has two parents (outputs),
32  * they eventually merged at `Join(a1=b2)`. So `Project(a,b,c)` is not a break-point.
33  * <p>the first [[RelNodeBlock]] includes TableScan, Project(a,b,c), Filter(a>0),
34  * Filter(c is not null), Project(a1,b1), Project(b2,c2) and Join(a1=b2)
35  * <p>the second one includes Filter(a1>=70), Project(a1,b1) and Sink(sink1)
36  * <p>the third one includes Filter(a1<70), Project(a1,c2) and Sink(sink2)
37  * <p>And the first [[RelNodeBlock]] is the child of another two.
38  *
39  * The [[RelNodeBlock]] plan is:
40  * {{{-
41  * RelNodeBlock2  RelNodeBlock3
42  *     \        /
43  *     RelNodeBlock1
44  * }}}
```

实战二：基于Flink SQL建设实时数仓实践

可以看到，在这段注释中，sink1和sink2这两个sink流有一段逻辑是共用的，即Join(a1=b2)。那么，优化器在优化阶段会将这段逻辑切分成3个block，其中共用的逻辑为单独的RelNodeBlock1，优化器将把这部分共用逻辑提取出来，避免重复计算。

中间表注册语法扩展

问题描述

值得注意的是，**原生的Flink SQL只能通过调用table api来提取共用逻辑。**在非table api的场景下，比如，数据经过计算后将根据字段条件被写入不同的kafka topic，SQL示例如下：

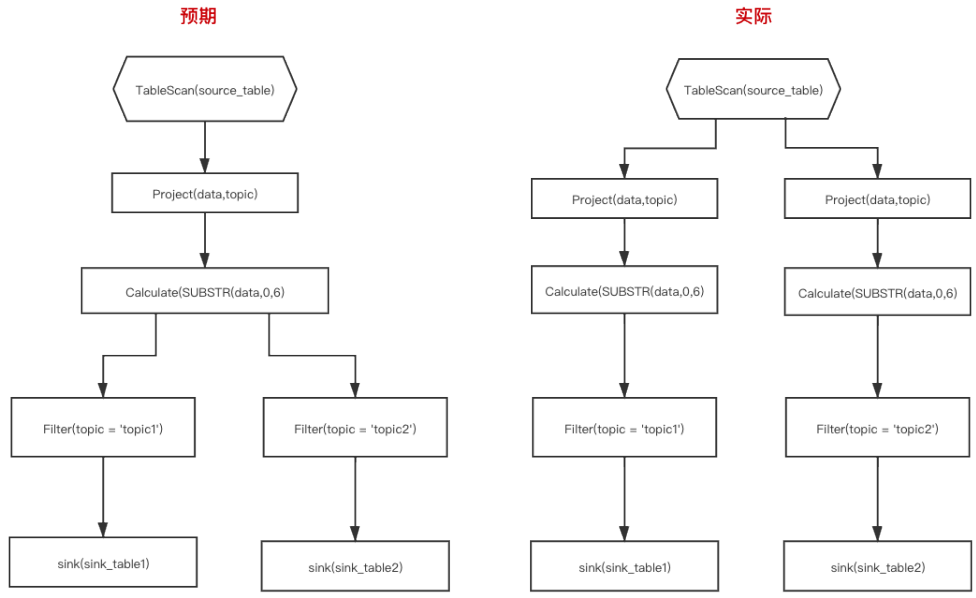
```
create table source_table(
 data string,
 topic string,
) with (
 'connector' = 'kafka',
 'topic' = 'topic',
 'properties.bootstrap.servers' = '',
 'format' = 'csv'
)

create table sink_table1(
 data string,
) with (
 'connector' = 'kafka',
 'topic' = 'topic2',
 'properties.bootstrap.servers' = '',
 'format' = 'csv'
)

create table sink_table2(
 data string,
) with (
```

实战二：基于Flink SQL建设实时数仓实践

```
23  'connector' = 'kafka',
24  'topic' = 'topic2',
25  'properties.bootstrap.servers' = '',
26  'format' = 'csv'
27 )
28
29 insert into sink_table1 select * from (select SBSTR(data, 0, 6) data,topic from source_table) where topic='topic1'
30
31 insert into sink_table2 select * from (select SBSTR(data, 0, 6) data,topic from source_table) where topic='topic2'
```

如果使用 StreamTableEnviroment.executeSql()去分别执行这两条insert sql，最终会异步生成两个任务，因此需要使用Flink提供的statementset先缓存多条insert sql，最后调用执行，在一个任务中完成多条数据流的处理。

可以发现，在这两条insert sql中存在复用逻辑，即select SBSTR(data, 0, 6) data、topic from source_table。预期的结果是Flink能够识别到这段共用逻辑并复用，但是实际情况并非预期中的，如下图：



问题分析

分析出现该问题的原因是：Flink在解析阶段将select SBSTR(data, 0, 6) data、topic from source_table解析成SqlNode(SqlSelect)并生成相应的RelNode。由于即便是相同逻辑的

实战二：基于Flink SQL建设实时数仓实践

SQL，其解析为RelNode的摘要也是不同的。而**Flink正是通过摘要来寻找复用的RelNode。**因此，Flink也就不能识别到这段逻辑是可以共用的。

判断逻辑共用的源码如下：

```
/**
  * Reuse common sub-plan in different RelNode tree, generate a RelNode dag
  *
  * @param relNodes RelNode trees
  * @return RelNode dag which reuse common subPlan in each tree
  */
 private def reuseRelNodes(relNodes: Seq[RelNode], tableConfig: TableConfig): Seq[RelNode] = {
val findOpBlockWithDigest = tableConfig.getConfiguration.getBoolean(
    RelNodeBlockPlanBuilder.TABLE_OPTIMIZER_REUSE_OPTIMIZE_BLOCK_WITH_DIGEST_ENABLED)
  if (!findOpBlockWithDigest) {
    return relNodes
  }

  // reuse sub-plan with same digest in input RelNode trees.
  val context = new SubplanReuseContext(true, relNodes: _*)
  val reuseShuttle = new SubplanReuseShuttle(context)
  relNodes.map(_.accept(reuseShuttle))
}
```

解决思路

那么如何解决呢？首先想到的思路就是**将这段共用逻辑注册成表**，这样Flink就能知道这段逻辑是共用的。

目前有**“注册视图”（create view as query）和“注册表”（registerTable）**两种方式能够将共用逻辑注册成表。在Flink中，当执行‘create view as query' 创建视图或者调用registerTable注册表时，底层都会在catalog中创建临时表，区别在于create view创建表的实现类为CatalogViewImpl，而registerTable创建表的实现类为QueryOperationCatalogView。

实战二：基于Flink SQL建设实时数仓实践

前者CatalogViewImpl的查询逻辑使用字符串表示，而后者 QueryOperationCatalogView的查询逻辑已经被解析为QueryOperation。

也就是说，执行创建视图的语句时，**最终创建的临时表仅仅是缓存了查询部分的SQL语句，当其他命令使用这个临时表时还需要重新解析临时表中的查询语句**，而重新解析带来的问题就是创建新的RelNode，产生不同的摘要，这样Flink仍然不能够识别到这段共用逻辑并复用。

相反，**regiterTable这样的方式就不需要对临时表中的查询语句进行重新解析。**因此可以采用regiterTable将共用逻辑注册成表。示例代码如下：

```
// 创建视图
StreamTableEnvironment.executeSql("create view as select SBSTR(data, 0, 6) data,topic from source_table);

// 注册表
Table table = StreamTableEnvironment.executeSql("select SBSTR(data, 0, 6) data,topic from source_table");

StreamTableEnvironment.registerTable("tmp", table);
```

但是，**为了SQL化，就需要有语法去支持中间表注册，以屏蔽底层的api调用**，实现用户无感知。新语法预期如下：

```
REGISTER TABLE TABLE_NAME AS SELECT_QUERY

// 示例
register table tmp as select SBSTR(data, 0, 6) data,topic from source_table
```

实现语法支持

如何实现新的语法，来支持中间表注册呢？目前有2种解决方案：

方案1：框架先使用正则匹配判断SQL类型，之后提取出临时表名和查询逻辑，比如上面的SQL经过正则匹配提取组之后可以得到表名为tmp，查询逻辑为'select SBSTR(data, 0, 6) data,topic from source_table'，之后框架去调用table api进行注册。

实战二：基于Flink SQL建设实时数仓实践

方案2：修改flink-table模块源码扩展语法，实现对register table语法的支持。

从实现难度上来说，**方案1的改动少，难度也较小，而方案2虽然改动较大，但是通用性更好**。下面主要围绕方案2的实现展开。

register table语法扩展大致分为以下3个步骤：

Step1 SQL解析与校验

即修改Java CC相关文件，使得 SqlParser可以识别新的语法并解析为AST。

➤ 1.1 增加关键字“REGISTER”

首先需要让解析器识别新的关键字“REGISTER”，因此修改Parser.tdd，在keywords和nonReservedKeywords中分别增加“REGISTER”关键字。

➤ 1.2 创建SqlNode (SqlRegisterTable)

由于SqlParser解析SQL生成的AST数据类型为SqlNode，因此需要增加相应的SqlNode。在org.apache.flink.sql.parser.ddl下创建SqlRegisterTable：

```
1  package org.apache.flink.sql.parser.ddl;
2
3  import org.apache.calcite.sql.*;
4  import org.apache.calcite.sql.parser.SqlParserPos;
5  import javax.annotation.Nonnull;
6
7  import java.util.Collections;
8  import java.util.List;
9
10 public class SqlRegisterTable extends SqlCall {
11 public static final SqlSpecialOperator OPERATOR = new SqlSpecialOperator("REGISTER TABLE", SqlKind.OTHER_DDL);
12
13 private SqlIdentifier tableName;
14 private SqlNode query;
15    // 部分代码省略
16 }
```

修改Parser.tdd，在imports增加org.apache.flink.sql.parser.ddl.SqlRegisterTable，即刚才新建的 SqlRegisterTable的全类名路径。

实战二：基于Flink SQL建设实时数仓实践

➤ 1.3 增加语法解析模版

除了关键字和SqlNode，还需要相应的语法模版，让解析器能够把SQL解析为SqlRegisterTable。

修改parserImpls.ftl，增加语法解析模版：

```
SqlRegisterTable SqlRegisterTable() :
 {
   SqlIdentifier tableName = null;
   SqlNode query = null;
   SqlParserPos pos;
   }
 {
<REGISTER> <TABLE> { pos = getPos();}
   tableName = CompoundIdentifier()
   <AS>
   query = OrderedQueryOrExpr(ExprContext.ACCEPT_QUERY)
   {
      return new SqlRegisterTable(pos, tableName, query);
      }
}
```

Step2：SqlNode转为Operation

根据calcite在Flink中的执行流程，Flink会将SqlNode封装为Operation，因此需要创建相应的RegisterTableOperation，并修改相关的转换逻辑。

➤ 2.1 新建 RegisterTableOperation

```
package org.apache.flink.table.operations.ddl;

import org.apache.flink.table.catalog.ObjectIdentifier;
import org.apache.flink.table.operations.Operation;
import org.apache.flink.table.operations.QueryOperation;

```

实战二：基于Flink SQL建设实时数仓实践

```
7  public class RegisterTableOperation implements Operation{
8   private final ObjectIdentifier tableIdentifier;
9   private final QueryOperation query;
10
11    // 省略部分代码
12 }
```

➤ 2.2 增加解析逻辑

生成RegisterTableOperation之后，还需要让Flink能将SqlNode转换成对应的Operation，因此我们要修改SqlToOperationConverter.convert内部代码，增加解析逻辑，代码如下：

```
1  private Operation convertRegisterTable(SqlRegisterTable sqlRegisterTable) {
2    UnresolvedIdentifier unresolvedIdentifier = UnresolvedIdentifier.of(sqlRegisterTable.fullTableName());
3    ObjectIdentifier identifier = catalogManager.qualifyIdentifier(unresolvedIdentifier);
4    PlannerQueryOperation operation = toQueryOperation(flinkPlanner, validateQuery);
5
6    return new RegisterTableOperation(identifier, operation);
7   }
```

Step3 Operation执行

即底层调用RegisterTable实现注册。由于流处理底层使用TableEnvironmentImpl进行相关SQL操作，比如常见的 executeSql(String statement) 操作：

```
1  @Override
2   public TableResult executeSql(String statement) {
3    List<Operation> operations = parser.parse(statement);
4
5    if (operations.size() != 1) {
6     throw new TableException(UNSUPPORTED_QUERY_IN_EXECUTE_SQL_MSG);
7    }
8
9    return executeOperation(operations.get(0));
10  }
```

实战二：基于Flink SQL建设实时数仓实践

因此在executeOperation()方法中，需要识别 RegisterTableOperation进行额外操作，因此增加operation执行逻辑如下：

```
                .build();
        } else if (operation instanceof DescribeTableOperation) {
            DescribeTableOperation describeTableOperation = (DescribeTableOperation) operation;
            Optional<CatalogManager.TableLookupResult> result =
                    catalogManager.getTable(describeTableOperation.getSqlIdentifier());
            if (result.isPresent()) {
                return buildDescribeResult(result.get().getResolvedSchema());
            } else {
                throw new ValidationException(String.format(
                        "Tables or views with the identifier '%s' doesn't exist",
                        describeTableOperation.getSqlIdentifier().asSummaryString()));
            }
        } else if (operation instanceof QueryOperation) {
            return executeInternal((QueryOperation) operation);
        } else if (operation instanceof RegisterTableOperation) {
            RegisterTableOperation registerTableOperation = (RegisterTableOperation) operation;
            Table table = createTable( registerTableOperation.getQuery());
            registerTable(registerTableOperation.getTableIdentifier().getObjectName(), table);
            return TableResultImpl.TABLE_RESULT_OK;
        } else {
            throw new TableException(UNSUPPORTED_QUERY_IN_EXECUTE_SQL_MSG);
        }
    }
```

通过以上三步，完成源码修改，然后将flink-parser打包替换当前依赖，即可实现对register table语法的扩展。

Flink的SQL执行基于calcite，语法拓展的实现简要概括分为语法解析、转换、优化和执行4个阶段，其中会涉及到Java CC、Planner等知识，有兴趣的同学可以查阅相关内容做深入了解。

总结

本文围绕中间表注册入手，对个推基于Flink SQL建设实时数仓的实践进行了总结和分享。

后续，我们还将持续梳理在实时业务场景下的Flink SQL应用实践，沉淀包括**SQL稳定性**、**SQL资源配置**等在内的通用解决方案；同时还将展开**批流一体**的探索，通过统一开发标准，来提升大数据作业的整体效率。

我们还将陆续面向行业总结和输出这些实践经验，更多精彩内容，请持续关注个推技术实践公众号。

TiDB调优实践：实现性能提速千倍

TiDB是一种支持水平弹性扩展，能够有效应对高并发、海量数据场景，同时高度兼容MySQL的云原生分布式数据库。

个推将MySQL切换到TiDB，期望实现在数据存储量不断增长的情况下，仍然确保数据的快速查询，满足内外部客户高效分析数据的需求，然而在完成数据迁移却陷入到TiDB的使用“反模式”。

本文从参数配置、热点问题解决等方面入手，分享两个调优技巧，为你打开TiDB的正确使用姿势，使系统性能提速千倍！

实战三：TiDB调优实践

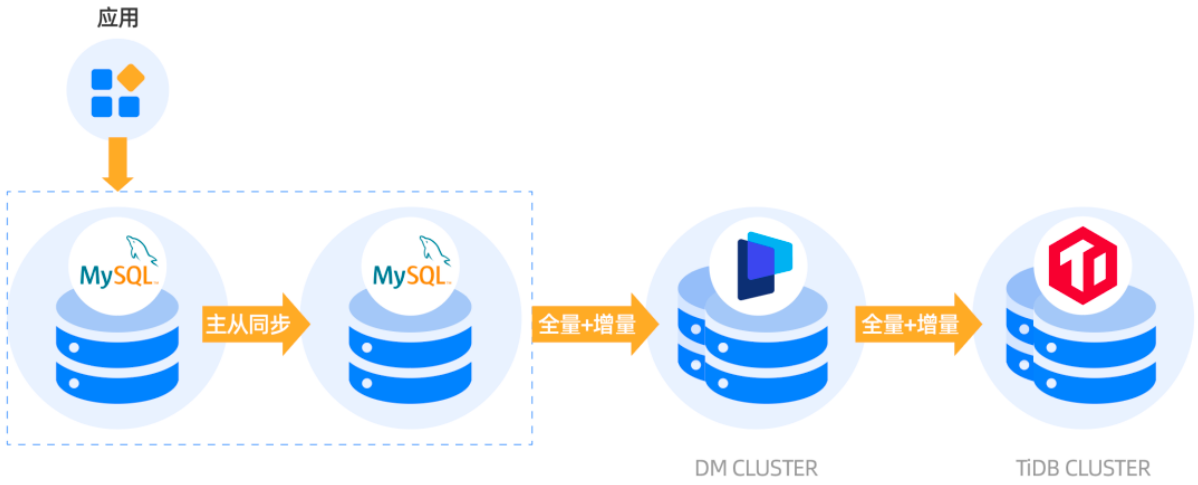
个推与TiDB的结缘

作为一家数据智能企业，个推为数十万APP提供了消息推送等开发者服务，同时为众多行业客户提供专业的数字化解决方案。在快速发展业务的同时，公司的数据体量也在高速增长。随着时间的推移，数据量越来越大，MySQL已经无法满足公司对数据进行快速查询和分析的需求，一种**支持水平弹性扩展，能够有效应对高并发、海量数据场景，同时高度兼容MySQL**的新型数据库成为个推的选型需求。

经过深入调研，我们发现“网红”数据库TiDB不仅具备以上特性，还是**金融级高可用、具有数据强一致性、支持实时HTAP**的云原生分布式数据库。因此，我们决定将MySQL切换到TiDB，期望实现**在数据存储量不断增长的情况下，仍然确保数据的快速查询，满足内外部客户高效分析数据**的需求，比如为开发者用户提供及时的推送下发量、到达率等相关数据报表，帮助他们科学决策。

完成选型后，我们就开始进行数据迁移。本次迁移MySQL数据库实例的数据量有数T左右，我们采用TiDB自带的生态工具Data Migration (DM)进行全量和增量数据的迁移。

- **全量数据迁移**：从数据源迁移对应表的表结构到TiDB，然后读取存量数据，写入到TiDB集群。
- **增量数据复制**：全量数据迁移完成后，从数据源读取对应的表变更，然后写入到TiDB集群。



个推将MySQL数据迁移到TiDB

实战三：TiDB调优实践

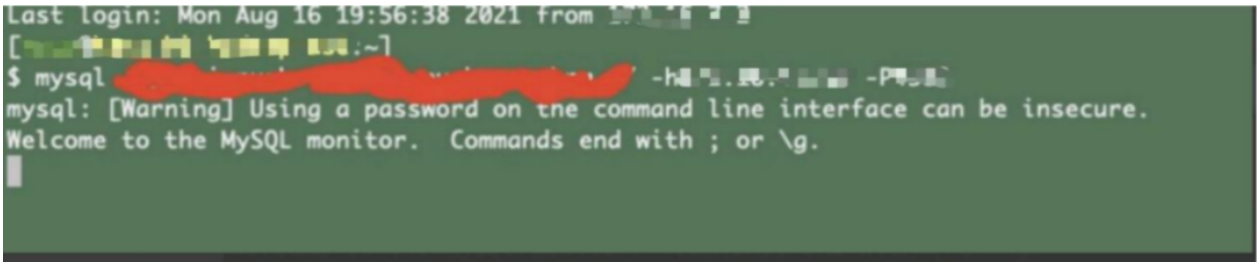
当数据同步稳定之后，将应用逐步迁移到TiDB Cluster。把最后一个应用迁移完成之后，停止DM Cluster。这样就完成了从MySQL到TiDB的数据迁移。

注：DM的具体配置使用详见官方文档。

陷入TiDB使用的“反模式”

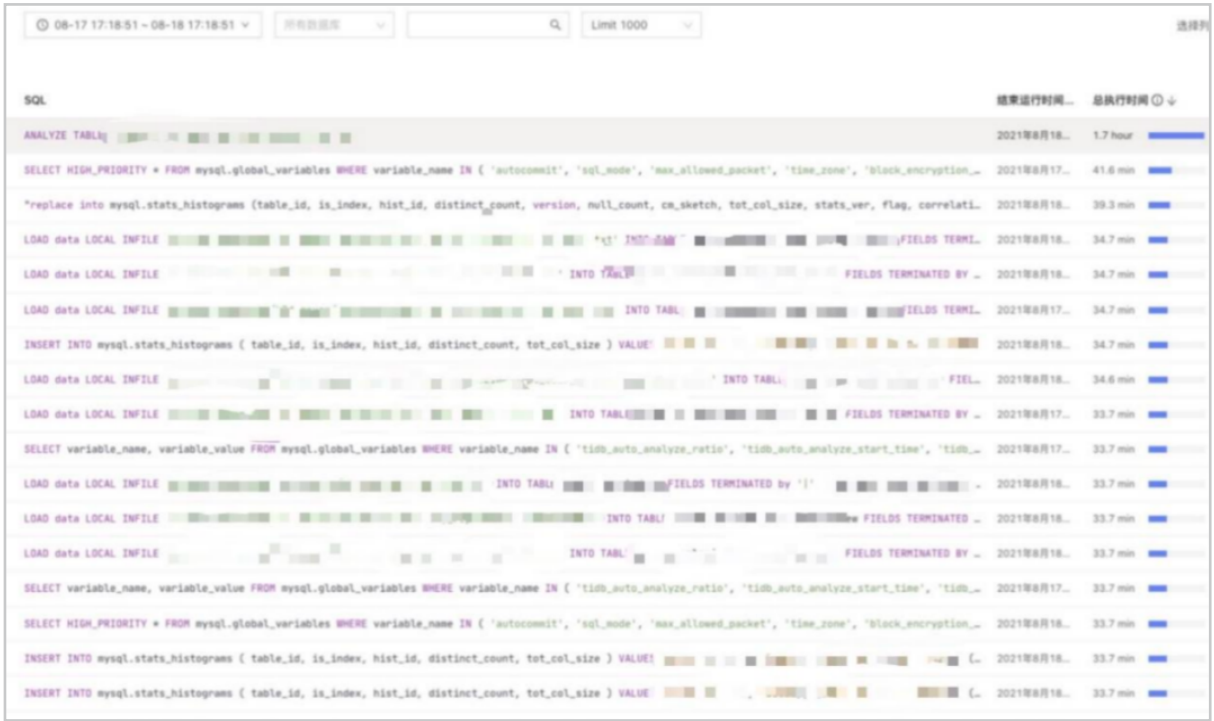
然而，当应用全部迁移到TiDB之后，却出现了数据库反应慢、卡顿，应用不可用等一系列的问题。

如下图：



登陆数据库时遇到卡顿

通过排查，我们发现有大量的慢SQL都是使用load导入数据的脚本。



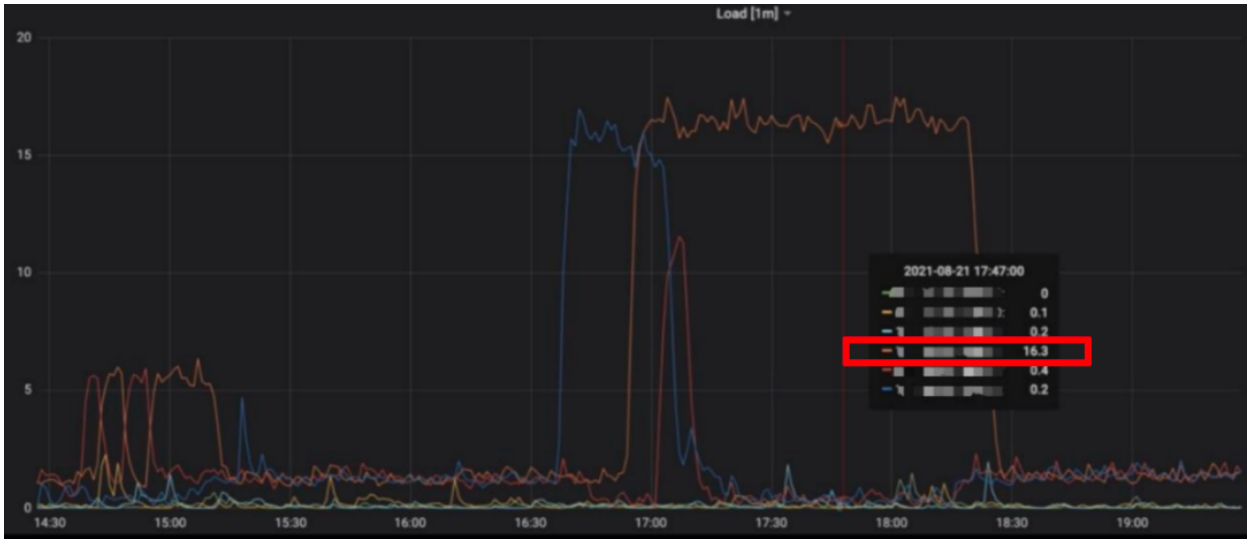
慢SQL的导入耗时几十分钟

实战三：TiDB调优实践

和业务方沟通后，我们发现**有些导入语句就包含几万条记录，导入时间需要耗时几十分钟。**

对比之前使用MySQL，一次导入只需几分钟甚至几十秒钟就完成了，而迁到TiDB却需要双倍甚至几倍的时间才完成，几台机器组成的TiDB集群反而还不如一台MySQL机器。

这肯定不是打开TiDB的正确姿势，我们需要找到原因，对其进行优化。



单个服务器负载过高

通过查看监控，发现服务器负载压力都是在其中一台机器上（如上图，红色线框里标注的这台服务器承担主要压力），这说明我们目前并没有充分利用到所有的资源，未能发挥出TiDB作为分布式数据库的性能优势。

打开TiDB的正确使用姿势

具体如何优化呢？我们首先从配置参数方面着手。众所周知，很多配置参数都是使用系统的默认参数，这并不能帮助我们合理地利用服务器的性能。通过深入查阅官方文档及多轮实测，我们对TiDB配置参数进行了适当调整，从而充分利用服务器资源，使服务器性能达到理想状态。

下表是个推对TiDB配置参数进行调整的说明，供参考：

实战三：TiDB调优实践

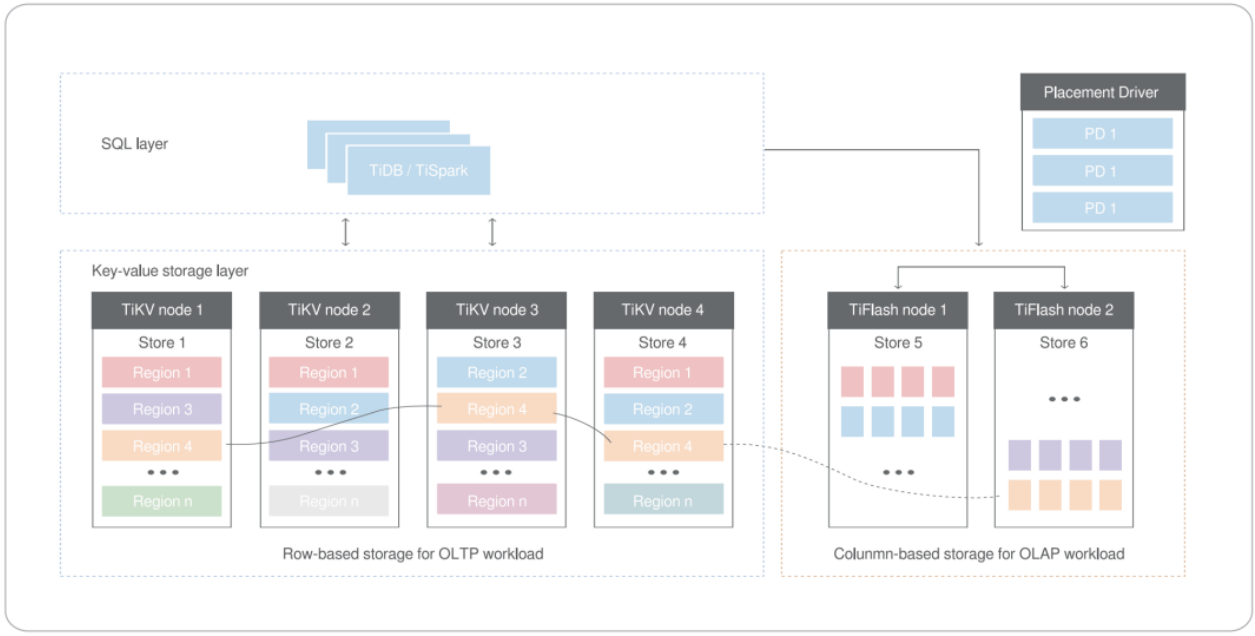
参数	调优前（默认设置）	调优后	备注
readpool.unified.max-thread-count	默认为机器CPU 数的80%（如机器为16核，则默认线程池大小为 12）	25	通常建议根据业务负载特性调整其CPU使用率为在线程池大小的60%～90%之间
storage.block-cache.capacity	系统总内存大小的45%	70GB	共享block cache的大小 默认值：系统总内存大小的 45% 建议：不超过系统内存的60%
raftstore.region-split-check-diff	region大小的 1/16	32MB	允许region数据超过指定大小的最大值
rocksdb.defaultcf.disable-auto-compactions	FALSE	TRUE	开启自动 compaction 的开关
raftstore.region-max-size	144MB	384MB	Region容量空间最大值，超过时系统分裂成多个Region
raftstore.region-split-size	96MB	256MB	分裂后新Region的大小，此值属于估算值
raftstore.split-region-check-tick-interval	10s	300s	检查region是否需要分裂的时间间隔，0 表示不启用
rocksdb.defaultcf.max-write-buffer-number	5	10	指最大memtable个数
rocksdb.writecf.max-write-buffer-number	5	10	指最大memtable个数
rocksdb.compaction-readahead-size	0	2MB	指异步Sync限速速率
readpool.storage.normal-concurrency	8	16	指处理普通优先级读请求的线程池线程数量 当8≤cpu num≤16时，默认值为cpu_num * 0.5；当cpu num大于8时，默认值为4；当cpu num大于16时，默认值为8，建议不超过50%

重点解决热点问题

调整配置参数只是基础的一步，我们还是要从根本上解决服务器负载压力都集中在一台机器上的问题。可是如何解决呢？这就需要我们先深入了解TiDB的架构，以及TiDB中表保存数据的内在原理。

在TiDB的整个架构中，分布式数据存储引擎TiKV Server负责存储数据。在存储数据时，TiKV采用范围切分（range）的方式对数据进行切分，切分的最小单位是region。每个region有大小限制（默认上限为96M），会有多个副本，每一组副本，成为一个raft group。每个raft group中由leader负责执行这个块数据的读&写。leader会自动地被PD组件（Placement Driver，简称“PD”，是整个集群的管理模块）均匀调度在不同的物理节点上，用以均分读写压力，实现负载均衡。

实战三：TiDB调优实践



TiDB架构图（图片来源于TiDB官网）

TiDB会为每个表分配一个TableID，为每一个索引分配一个IndexID，为每一行分配一个RowID（默认情况下，如果表使用整数型的Primary Key，那么会用Primary Key的值当做RowID）。同一个表的数据会存储在以表ID开头为前缀的一个range中，数据会按照RowID的值顺序排列。在插入（insert）表的过程中，如果RowID的值是递增的，则插入的行只能在末端追加。

当Region达到一定的大小之后会进行分裂，**分裂之后还是只能在当前range范围的末端追加，并永远仅能在同一个Region上进行insert操作，由此形成热点（即单点的过高负载）**，陷入TiDB使用的“反模式”。

常见的increment类型自增主键就是按顺序递增的，默认情况下，在主键为整数型时，会将主键值作为RowID，此时RowID也为顺序递增，在大量insert时就会形成表的写入热点。同时，TiDB中RowID默认也按照自增的方式顺序递增，主键不为整数类型时，同样会遇到写入热点的问题。

在使用MySQL数据库时，为了方便，我们都习惯使用自增ID来作为表的主键。**因此，将数据从MySQL迁移到TiDB之后，原来的表结构都保持不变，仍然是以自增ID作为表的主键**。这样就造成了批量导入数据时出现TiDB写入热点的问题，导致Region分裂不断进行，消耗大量资源。

实战三：TiDB调优实践

对此，在进行TiDB优化时，我们从表结构入手，对以自增ID作为主键的表进行重建，删除自增ID，使用TiDB隐式的_tidb_rowid列作为主键，将

```
create table t (a int primary key auto_increment, b int);
```

改为：

```
create table t (a int, b int)SHARD_ROW_ID_BITS=4 PRE_SPLIT_REGIONS=2
```

通过设置SHARD_ROW_ID_BITS，将RowID打散写入多个不同的Region，从而缓解写入热点问题。

此处需要注意，SHARD_ROW_ID_BITS值决定分片数量：

- SHARD_ROW_ID_BITS = 0 表示 1 个分片
- SHARD_ROW_ID_BITS = 4 表示 16 个分片
- SHARD_ROW_ID_BITS = 6 表示 64 个分片

SHARD_ROW_ID_BITS值设置的过大会造成RPC请求数放大，增加CPU和网络开销，这里我们将SHARD_ROW_ID_BITS设置为4。

PRE_SPLIT_REGIONS指的是建表成功后的预均匀切分，我们通过设置PRE_SPLIT_REGIONS=2，实现建表成功后预均匀切分2^(PRE_SPLIT_REGIONS)个Region。

经验总结

以后新建表禁止使用自增主键，考虑使用业务主键

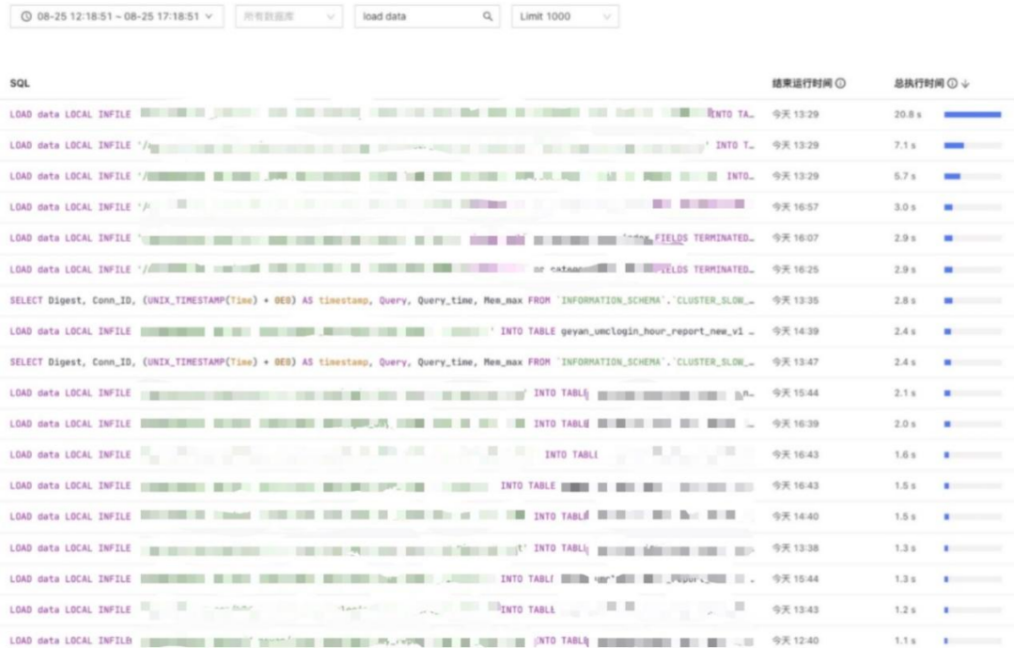
加上参数SHARD_ROW_ID_BITS = 4 PRE_SPLIT_REGIONS=2

此外，由于TiDB的优化器和MySQL有一定差异，出现了相同的SQL语句在MySQL里可以正常执行，而在TiDB里执行慢的情况。我们针对特定的慢SQL进行了深入分析，并针对性地进行了索引优化，取得了不错的成效。

优化成果

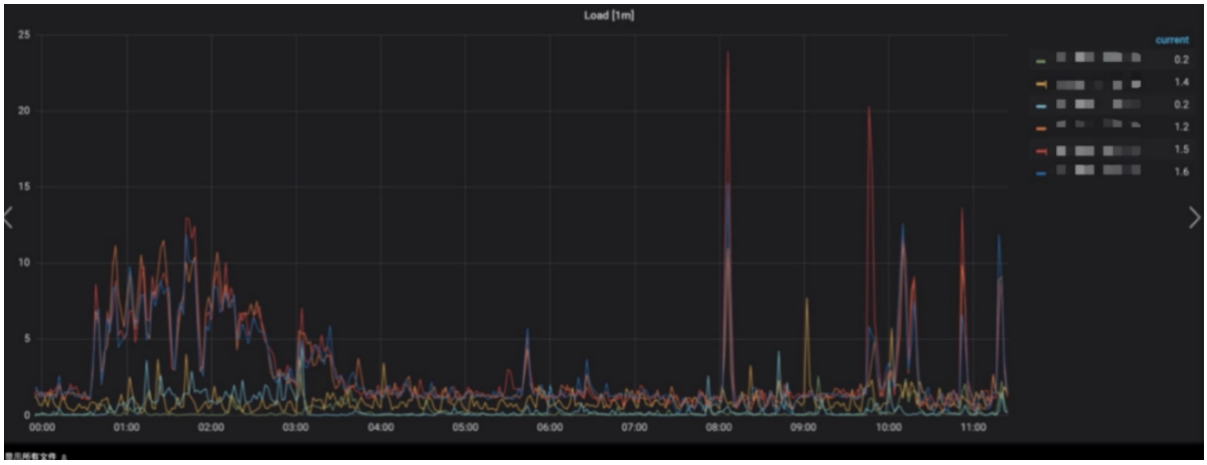
通过慢SQL查询平台可以看到，经过优化，大部分的导入在秒级时间内就完成了，**相比原来的数十分钟，实现了数千倍的性能提升。**

实战三：TiDB调优实践



慢SQL优化结果

同时，性能监控图表也显示，在负载高的时刻，是几台机器同时高，而不再是单独一台升高，这说明我们的优化手段是有效的，TiDB作为分布式数据库的优势得以真正体现。



优化后，实现服务器负载均衡

总结

作为一种新型分布式关系型数据库，TiDB能够为OLTP（Online Transactional Processing）和OLAP（Online Analytical Processing）场景提供一站式的解决方案。个推不仅使用TiDB进行海量数据高效查询，同时也展开了基于TiDB进行实时数据分析、洞察的探索。

在Hadoop2.0上丝滑落地EC的实践

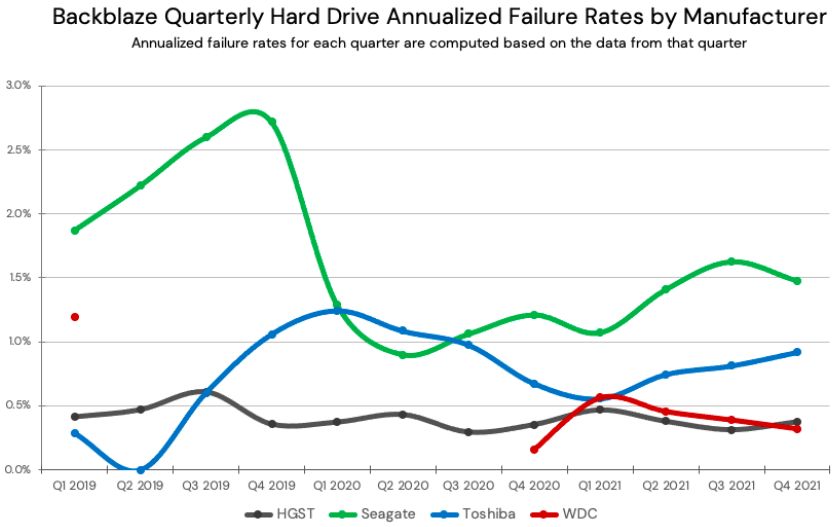
在Hadoop3.0版本上，HDFS引入了EC技术（Erasure Code 纠删码）。作为Hadoop3.0的全新特性之一，EC技术采用什么样的算法来进行编解码？对于未升级Hadoop3.0的企业，又该如何在实际场景中落地EC技术？

本文为您详细解读EC纠删码的技术原理，并结合个推实践分享在Hadoop2.0上丝滑落地EC，使数据存储成本降低约50%的宝贵经验。

实战四：在Hadoop2.0上丝滑落地EC的实践

前言

根据云存储服务商Backblaze发布的2021年硬盘“质量报告”，现有存储硬件设备的可靠性无法完全保证，我们需要在软件层面通过一些机制来实现可靠存储。**一个分布式软件的常用设计原则就是面向失效的设计。**



	Q1 2019	Q2 2019	Q3 2019	Q4 2019	Q1 2020	Q2 2020	Q3 2020	Q4 2020	Q1 2021	Q2 2021	Q3 2021	Q4 2021
HGST	0.41%	0.47%	0.61%	0.36%	0.37%	0.43%	0.30%	0.35%	0.47%	0.38%	0.31%	0.37%
Seagate	1.87%	2.22%	2.60%	2.72%	1.29%	0.90%	1.06%	1.21%	1.07%	1.41%	1.63%	1.48%
Toshiba	0.29%	0.00%	0.60%	1.06%	1.24%	1.09%	0.98%	0.67%	0.56%	0.75%	0.81%	0.92%
WDC	1.20%							0.16%	0.57%	0.46%	0.39%	0.32%

Backblaze

图片来源于Backblaze

作为当前广泛流行的分布式文件系统，HDFS需要解决的一个重要问题就是数据的可靠性问题。3.0以前版本的Hadoop在HDFS上只能采用多副本冗余的方式做数据备份，以实现数据可靠性目标（比如，三副本11个9，双副本8个9）。多副本冗余的方式虽然简单可靠，却浪费了成倍的存储资源，随着数据量的增长，将带来大量额外成本的增加。为了解决冗余数据的成本问题，在Hadoop3.0版本上，HDFS引入了EC技术（Erasure Code 纠删码）。

本文分享EC技术原理及个推的EC实践，带大家一起玩转Hadoop3.0！

EC原理深度解读

EC技术深度应用于RAID和通信领域，**通过对数据编解码以实现在部分数据丢失时仍能够将其恢复。**

我们可以这样理解EC的目标和作用：对n个同样大小的数据块，额外增加m个校验块，使得这n+m个数据中任意丢失m个数据块或校验块时都能恢复原本的数据。

实战四：在Hadoop2.0上丝滑落地EC的实践

以HDFS的**RS-10-4-1024k**策略为例，可以实现在1.4倍的数据冗余的情况下，达到近似于5副本的数据可靠性，也就是说**以更小的数据冗余度获得更高的数据可靠性。**

EC算法

常见的EC算法有XOR、RS，下面一一做简要介绍：

简单EC算法：XOR

XOR是一种基于异或运算的算法，通过对两个数据块进行按位异或，就可以得到一个新的数据块，当这三个数据块中有任意一个数据块丢失时，就可以通过另外两个数据块的“异或”恢复丢失的数据块。

HDFS通过XOR-2-1-1024k的EC策略实现了该算法，这种方式虽然降低了冗余度，但是**只能容忍三个数据块中一个丢失**，在很多情况下其可靠性依然达不到要求。

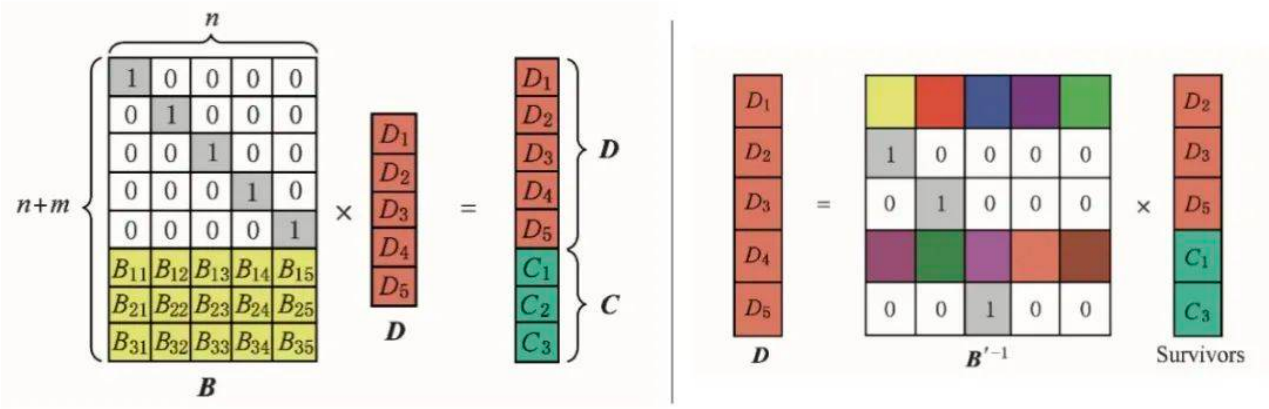
改进的EC算法：RS

另一种降低冗余度的编码方式为Reed-Solomon（RS），它有两个参数，记为RS(n,m)。其中，n表示数据块，m表示校验块，需要注意的是，**RS算法下，有多少个校验块，就表示最多可容忍多少个数据块（包括数据块和校验块）丢失。**

RS 算法使用**生成矩阵**（GT，Generator Matrix）与n个数据单元相乘，以获得具有n个数据单元（data cells）和m个奇偶校验单元（parity cells）的矩阵。如果存储失败，那么只要n+m个cells中的n个可用，就可以通过生成器矩阵来恢复存储。

RS算法克服了XOR算法的限制，使用线性代数运算生成多个parity cells，以便能够容忍多个失败。

下图形象地描述了RS算法的编码与解码过程：



图片来源于网络

实战四：在Hadoop2.0上丝滑落地EC的实践

编码过程（图左）：

- 把m个有效数据组成一个向量D。
- 生成一个变换矩阵B：由一个n阶的单位矩阵和一个n * M的范德蒙特矩阵（Vande-mode）组成。
- 两矩阵B和D，相乘，得到一个新的具备纠错能力的矩阵。

解码过程（图右）：

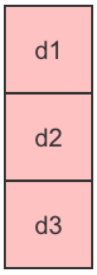
- 取范德蒙矩阵B中没有丢失的行，构成矩阵B`。
- 取编码过程最后计算的矩阵中没有丢失的行，构成矩阵Survivors。
- 用B`的逆，乘以Survivors矩阵，即可得到原始的有效数据。

为了更通俗地说明编解码过程，我们以RS-3-2-1024k策略为例，回顾使用EC算法进行编解码的过程。

假设有三块数据：d1,d2,d3，我们需要额外存储两块数据，使得这五块数据中，任意丢失两块数据，都能将它们完整找回。

首先按编码过程构建纠错矩阵：

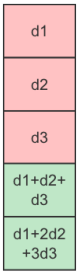
1.得到向量D



2、生成一个变换矩阵B

1	0	0
0	1	0
0	0	1
1	1	1
1	2	3

3、得到纠错矩阵D*B

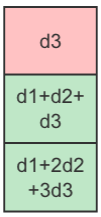


假设d1,d2数据丢失，我们通过解码做数据恢复：

1、取B中没有丢失的行，构成矩阵B`

0	0	1
1	1	1
1	2	3

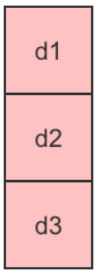
2、取纠错矩阵中没有丢失的行，构成矩阵Survivors



3、计算B`的逆为：

1	2	-1
-2	-1	1
1	0	0

4、B`的逆，乘以Survivors矩阵，即可得到原始的有效数据：



至此，我们完成了**在原本3个数据块外再额外存储2个数据块，使得这5个数据块中任意丢失两个都能将其找回的目标。**

实战四：在Hadoop2.0上丝滑落地EC的实践

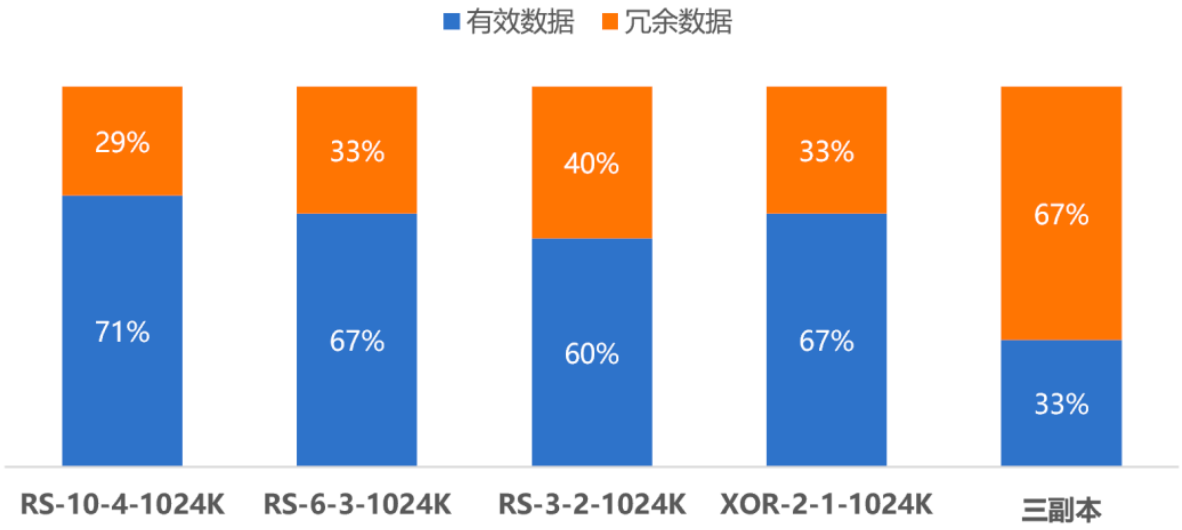
与三副本的方式对比，在**可靠性**方面，三副本方式可以容忍存储该文件（数据d1,d2,d3）的机器中任意两台宕机或坏盘，因为总还有一个副本可用，并通过复制到其他节点恢复到三副本的水平。同样，在RS-3-2策略下，我们也可以容忍5个数据块所在的任意2台机器宕机或坏盘，因为总可以通过另外的3个数据块来恢复丢失的2个数据块。

由此可见，三副本方式和RS-3-2策略，在可靠性方面基本相当。

在**冗余度（冗余度=实际存储空间/有效存储空间）**方面，三副本方式下，每1个数据块都需要额外的2个数据块做副本，冗余度为3/1=3，而在RS-3-2的策略下，每3个数据块只需要额外的2个数据块就能够实现可靠性目标，冗余度为5/3=1.67。

最终，我们通过RS-3-2的方式能够在1.67倍冗余的情况下，实现近似三副本的可靠性。

下图为Hadoop上，不同策略下的有效数据与冗余数据占比示意图。可以看到，三副本方式的存储成本是最高的：



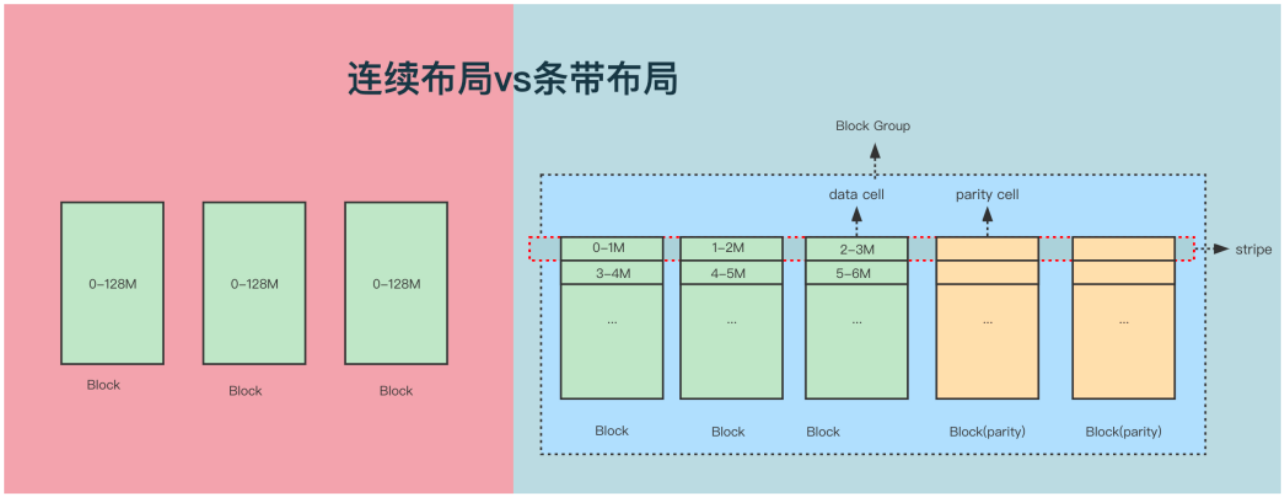
条带布局

副本策略以块（Block）为单位，将数据连续写入Block中，直至达到该Block上限（默认128M），然后再去申请下一个Block。以最常见的三副本方式为例，每个Block会有3个相同数据的副本存储于3个DataNode（DN）上。

HDFS EC策略采用的是**条带式存储布局**（Striping Block Layout）。条带式存储以块组（BlockGroup）为单位，横向式地将数据保存在各个Block上，同一个Block上不同分段的数据是不连续的，写完一个块组再申请下一个块组。

实战四：在Hadoop2.0上丝滑落地EC的实践

下图为连续布局和RS(3,2)策略下一个BlockGroup布局对比：



相比于连续布局，条带布局有以下优势：

- 支持直接写入EC数据，不需要做离线转化
- 对小文件更友好
- I/O并行能力提高

个推在Hadoop2.0上落地EC

个推在很早的时候就对整个集群做了规划，将整个Hadoop集群分为对计算需求比较大的热集群和对存储需求比较大的冷集群。在Hadoop3.x发布以后，我们将冷集群升级到Hadoop3.x版本，进行了包括EC编码在内的新特性试用。考虑到计算引擎的兼容性、稳定性要求，同时为了减少迁移成本，我们**仍将热集群保持在了Hadoop2.7版本。**

计算引擎访问

由于主要承接计算任务的热集群是Hadoop2.x的环境，而内部的计算引擎都不支持Hadoop3.x，所以为了将EC功能在生产环境落地，我们**首先要解决在Hadoop2.x上对Hadoop3.x上EC数据进行访问的能力。**为此，我们对Hadoop2.7的hadoop-hdfs做了定制化开发，移植了Hadoop3.x上的EC功能，核心的变动包括：

- EC编解码和条带相关功能引入
- PB协议的适配
- 客户端读取流程的改造

实战四：在Hadoop2.0上丝滑落地EC的实践

资源本地化

在部署改造代码包的过程中，
我们使用Hadoop的“资源本地化”机制，
简化了灰度和上线流程。

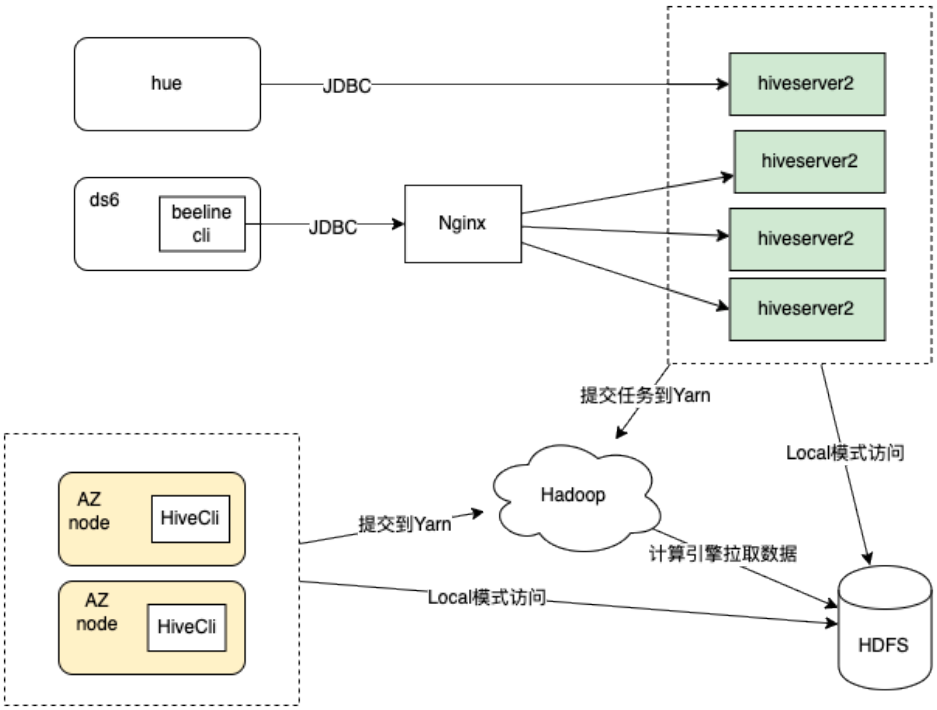
所谓“资源本地化”，指的是NodeManager在启动container以前需要从HDFS上下载该container执行所依赖资源的过程，这些资源包括jar、依赖的jar或者其它文件。借助资源本地化的特性，我们就可以将jar包，定制分发到相应计算任务的container中，以控制application级别任务的Container的jar包环境，使后续的测试、灰度验证和上线非常方便。

SQL访问

个推目前有大量的任务是通过SQL方式提交的，其中，大部分的SQL任务在提交到Yarn上之后，会转化为相应计算引擎的计算任务。针对该部分SQL任务，我们可以直接使用第一种解决方案进行访问。

但是，**仍然存在一部分的任务并不通过Yarn提交，而是直接与HDFS做交互**，比如一些小数据集计算任务或直接通过limit查看几条示例的SQL任务（例如select * from table_name limit 3）。这就需要该部分任务所在的节点，有访问Hadoop3.x上EC数据的能力。

以Hive为例，下图为个推环境上Hive访问HDFS数据的几种方式，这里的HiveCli、Hiveserver2都要做相应的适配：



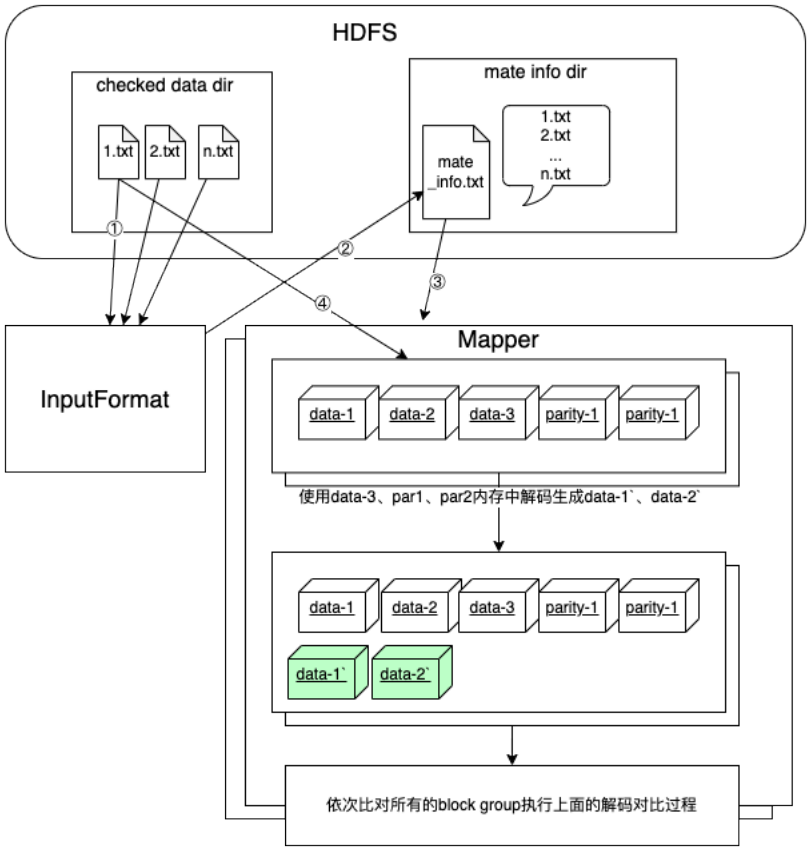
实战四：在Hadoop2.0上丝滑落地EC的实践

损坏校验

在社区提交的EC相关的bug中，我们发现有一些bug会导致编码的数据损坏，例如：HDFS-14768、HDFS-15186、HDFS-15240。为了确保转化后的数据是正确的，我们对编码后的数据做了块级别的额外校验。

在设计校验程序时，我们**不仅要考虑校验程序的便利性，使其能够对新EC的数据做校验，还要对之前已经EC过的数据做一遍校验**。因此，我们主要的思路是利用全部的校验块和部分数据块对编码后的数据做解码，来对比解码后的数据块和原数据块是否一致。

我们以RS-3-2-1024k为例，回顾下校验过程：



基于额外的校验工具，我们抽取了单副本1PB的EC数据做了验证，验证结果显示，故障的文件数和大小占比大约都在百万分之一以下，可靠性达到目标要求。

展望

目前，我们还需要专门统计和筛选出热度比较低的数据并过滤掉小文件，用于后续的EC编码处理。后续我们将探索设计一套系统，使其能**自动感知到数据的热度下降，并完成数据从副本策略到EC策略的转化**。此外，在**intel　ISA-L加速库**等方面，我们还将持续展开探索，以提升整个EC编解码的运算效率。

使用Pika实现Codis存储成本降低90%的实践

个推采用Codis保存大规模的key-value数据，随着公司kv类型数据的不断增加，使用原生Codis搭建的集群所花费的成本越来越高。在一些对性能响应要求不高的场景中，个推计划采用新的存储和管理方案以有效兼顾成本与性能。经过选型，个推引入了360开源的存储系统Pika作为Codis的底层存储，以替换成本较高的codis-server，管理分布式kv数据集群。

本文为大家分享个推如何完美结合Pika和Codis，最终节省90%大数据存储成本的实战经验。

实战五：使用Pika实现Codis存储成本降低90%的实践

前言

作为一家数据智能公司，个推不仅拥有海量的关系型数据，也积累了丰富的key-value等非关系型数据资源。个推采用Codis保存大规模的key-value数据，**随着公司kv类型数据的不断增加，使用原生的Codis搭建的集群所花费的成本越来越高。**

在一些对性能响应要求不高的场景中，个推计划采用新的存储和管理方案以有效兼顾成本与性能。经过选型，个推引入了360开源的存储系统Pika作为Codis的底层存储，以替换成本较高的codis-server，管理分布式kv数据集群。

将Pika接入到Codis的过程并非一帆风顺，为了更好地满足业务场景需求，个推进行了系列设计和改造工作。**本文为大家分享个推如何完美结合Pika和Codis，最终节省90%大数据存储成本的实战经验。**

Codis的四大组件

在了解具体的迁移实战之前，需要先初步认识下Codis的基本架构。Codis是一个分布式Redis解决方案，由codis-fe、codis-dashboard、codis-proxy、codis-server等四个组件构成。

- 其中，**codis-server是Codis中最核心和基础的组件**。基于Redis3版本，codis-server进行了功能扩展，但其本质上还是依赖于高性能的Redis提供服务。codis-server扩展了基于slot的key存储功能（为了实现slot这个功能，codis-server会额外占用超出存储数据所需的内存），并能够在Codis集群的不同Group之间进行slot数据热迁移。
- codis-fe则提供对运维比较友好的管理界面，方便统一管理多套的codis-dashboard。
- codis-dashboard负责管理slot、codis-proxy和ZooKeeper（或者etcd）等组件的数据一致性，整个集群的运维状态，数据的扩容缩容和组件的高可用，类似于k8s的api-server功能。
- codis-proxy主要提供给业务层面使用的访问代理，负责解析请求路由并将key的路由信息路由到对应的后端group上面。此外，codis-proxy还有一个很重要的功能，即在通过codis-fe进行集群的扩缩容时，**codis-proxy会根据group对应的slot的迁移状态触发key迁移的流程，能够实现在不中断业务服务的情况下热迁移数据，以确保业务的可用性。**

实战五：使用Pika实现Codis存储成本降低90%的实践

Pika接入Codis的挑战

我们引入Pika主要是用来替换codis-server。作为360开源的类Redis存储系统，**Pika底层选用RocksDB，它完全兼容Redis协议，并且主流版本提供Codis的接入能力。**但在引入Pika以及将数据迁移到Codis的过程中，我们发现Pika和Codis的结合并非想象中完美。

问题一：语法不统一

在接入之前，我们深入查阅并对比了Pika和Codis源码，**发现Pika实现的命令相对较少，将Pika接入到Codis之后有些功能还能否正常使用有待观察。**

- 位于pika_command.h头文件中的Pika (3.4.0版本) 源码：

```
//Codis Slots
const std::string kCmdNameSlotsInfo = "slotsinfo";
const std::string kCmdNameSlotsHashKey = "slotshashkey";
const std::stringkCmdNameSlotsMgrtTagSlotAsync = "slotsmgrttagslot-async";
const std::string kCmdNameSlotsMgrtSlotAsync = "slotsmgrtslot-async";
const std::string kCmdNameSlotsDel = "slotsdel";
const std::string kCmdNameSlotsScan = "slotsscan";
const std::string kCmdNameSlotsMgrtExecWrapper = "slotsmgrt-exec-wrapper";
const std::string kCmdNameSlotsMgrtAsyncStatus = "slotsmgrt-async-status";
const std::string kCmdNameSlotsMgrtAsyncCancel = "slotsmgrt-async-cancel";
const std::string kCmdNameSlotsMgrtSlot = "slotsmgrtslot";
const std::string kCmdNameSlotsMgrtTagSlot = "slotsmgrttagslot";
const std::string kCmdNameSlotsMgrtOne = "slotsmgrtone";
const std::string kCmdNameSlotsMgrtTagOne = "slotsmgrttagone";
```

- codis-server支持的命令如下：

```
{"slotsinfo",slotsinfoCommand,-1,"rF",0,NULL,0,0,0,0},
{"slotsscan",slotsscanCommand,-3,"rR",0,NULL,0,0,0,0,0},
{"slotsdel",slotsdelCommand,-2,"w",0,NULL,1,-1,1,0,0},
{"slotsmgrtslot",slotsmgrtslotCommand,5,"w",0,NULL,0,0,0,0,0},
{"slotsmgrttagslot",slotsmgrttagslotCommand,5,"w",0,NULL,0,0,0,0,0},
{"slotsmgrtone",slotsmgrtoneCommand,5,"w",0,NULL,0,0,0,0,0},
```

实战五：使用Pika实现Codis存储成本降低90%的实践

```
 {"slotsmgrttagone",slotsmgrttagoneCommand,5,"w",0,NULL,0,0,0,0,0},
 {"slotshashkey",slotshashkeyCommand,-1,"rF",0,NULL,0,0,0,0,0},
{"slotscheck",slotscheckCommand,0,"r",0,NULL,0,0,0,0,0},
 {"slotsrestore",slotsrestoreCommand,-4,"wm",0,NULL,0,0,0,0,0},
 {"slotsmgrtslot-async",slotsmgrtSlotAsyncCommand,8,"ws",0,NULL,0,0,0,0,0},
 {"slotsmgrttagslot-async",slotsmgrtTagSlotAsyncCommand,8,"ws",0,NULL,0,0,0,0,0},
 {"slotsmgrtone-async",slotsmgrtOneAsyncCommand,-7,"ws",0,NULL,0,0,0,0,0},
 {"slotsmgrttagone-async",slotsmgrtTagOneAsyncCommand,-7,"ws",0,NULL,0,0,0,0,0},
 {"slotsmgrtone-async-dump",slotsmgrtOneAsyncDumpCommand,-4,"rm",0,NULL,0,0,0,0,0},
 {"slotsmgrttagone-async-dump",slotsmgrtTagOneAsyncDumpCommand,-4,"rm",0,NULL,0,0,0,0,0},
 {"slotsmgrt-async-fence",slotsmgrtAsyncFenceCommand,0,"rs",0,NULL,0,0,0,0,0},
 {"slotsmgrt-async-cancel",slotsmgrtAsyncCancelCommand,0,"F",0,NULL,0,0,0,0,0},
 {"slotsmgrt-async-status",slotsmgrtAsyncStatusCommand,0,"F",0,NULL,0,0,0,0,0},
 {"slotsmgrt-exec-wrapper",slotsmgrtExecWrapperCommand,-3,"wm",0,NULL,0,0,0,0,0},
 {"slotsrestore-async",slotsrestoreAsyncCommand,-2,"wm",0,NULL,0,0,0,0,0},
 {"slotsrestore-async-auth",slotsrestoreAsyncAuthCommand,2,"sltF",0,NULL,0,0,0,0,0},
 {"slotsrestore-async-select",slotsrestoreAsyncSelectCommand,2,"lF",0,NULL,0,0,0,0,0},
 {"slotsrestore-async-ack",slotsrestoreAsyncAckCommand,3,"w",0,NULL,0,0,0,0,0},
```

此外，**codis-server和Pika支持的语法也有所不同。**例如，如果要查看某一节点上slot 1的详细信息，Codis与Pika执行的命令分别如下：

Codis-server命令	Pika分布式模式下命令
redis-cli -p 6379 SLOTSINFO 1	redis-cli -p 6379 pkcluster info slot 1

也就是说，我们必须在codis-fe层命令调度与管理功能方面加上对Pika语法格式的支持。

针对此问题，我们在codis-dashboard层中，通过修改部分源码逻辑，实现了对Pika主从同步、主从提升等相关命令的支持，从而完成了在codis-fe层面的操作。

问题二：未成功完成数据迁移

完成了以上操作之后，我们便开始将kv数据迁移到Pika。然后，问题来了，我们发现**虽然codis-fe界面上显示数据均已迁移完成，但实际上要迁移的数据并未被迁移到对应的集群。**在codis-fe界面上，我们也未查看到明显的报错信息。

实战五：使用Pika实现Codis存储成本降低90%的实践

到底为何出现此问题呢？

我们继续查看了Pika有关slot的源码：

```
1  void SlotsMgrtSlotAsyncCmd::Do(std::shared_ptr<Partition> partition) {
2    int64_t moved = 0;
3    int64_t remained = 0;
4    res_.AppendArrayLen(2);
5    res_.AppendInteger(moved);
6    res_.AppendInteger(remained);
7  }
```

我们发现，在日常的运行情况下，通过codis-dashboard发送给Pika的指令就是成功返回，这样codis-dashboard在迁移时立马就收到了成功的信号，然后就直接将迁移状态修改为成功，而其实此时数据迁移并没有被真的执行。

针对这种情况，我们查阅了有关Pika的官方文档 **Pika配合Codis扩容案例。**

从官方的文档来看，这种迁移方案是一种可能会丢数据的有损方案，我们需要根据自身情况来重新设计和调整迁移方案。

1.设计开发Pika迁移工具

首先，根据Codis的数据扩缩容原理，我们**参考codis-proxy的架构设计，使用Go语言自行设计并开发了一套Pika数据迁移工具，**目的是实现以下功能需求：

- 将Pika迁移工具**伪装成一个Pika实例接入Codis**并提供服务。
- 把Pika迁移工具**作为一个流量转发工具**，类似于codis-proxy，能够将对应slot的请求转发到指定的Pika实例上面，从而保证迁移过程中的业务可用性。
- 使Pika迁移工具**能够感知到迁移过程中的主从同步情况**，在主从完成的情况下可自动从节点断开，并将新增数据写入新集群，从而在流量分发过程中全力保证数据一致性。

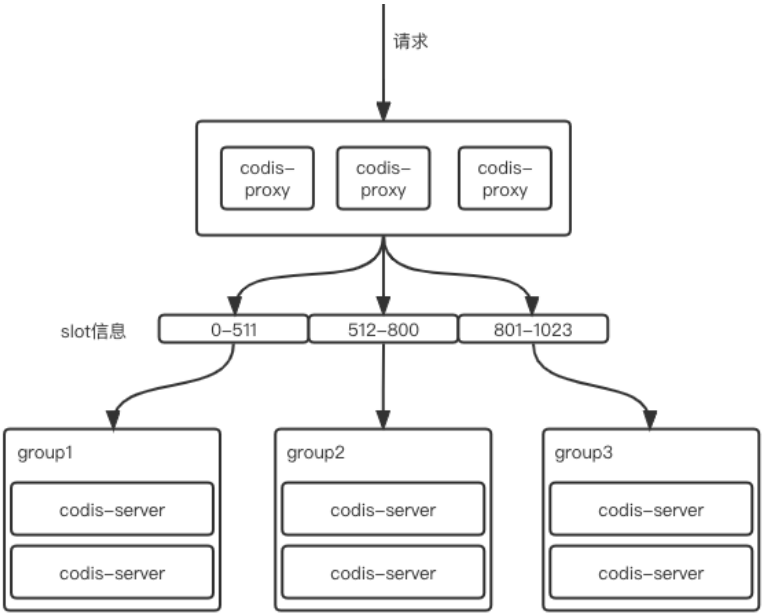
2.使用Pika迁移工具进行数据的热迁移

根据如上需求完成Pika迁移工具的设计开发后，我们就可以使用该工具对数据进行热迁移。迁移过程如下：

➤ **Step1: 集群原始状态**

实战五：使用Pika实现Codis存储成本降低90%的实践

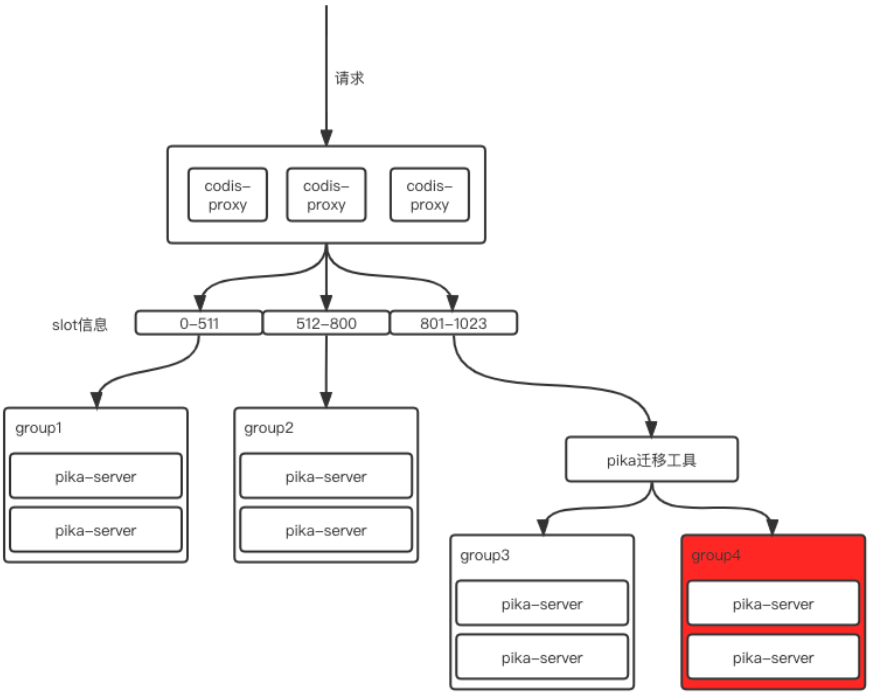
通过下图，可以看到，我们需要将801-1023中901-1023区间的slot信息迁移到新组件即Group4上，作为新实例提供服务。



➤ Step2: 将Pika迁移工具接入Codis提供服务

在Pika迁移工具接入Codis之前，我们需将Group3中待迁移的901-1023作为Group4的主节点，并进行主从数据同步。此时Group3的901-1023作为主，Group4的901-1023作为从。在完成该步骤之后就可将Pika迁移工具接入Codis。

- 首先将801-1023的slot信息迁移到Pika迁移工具。

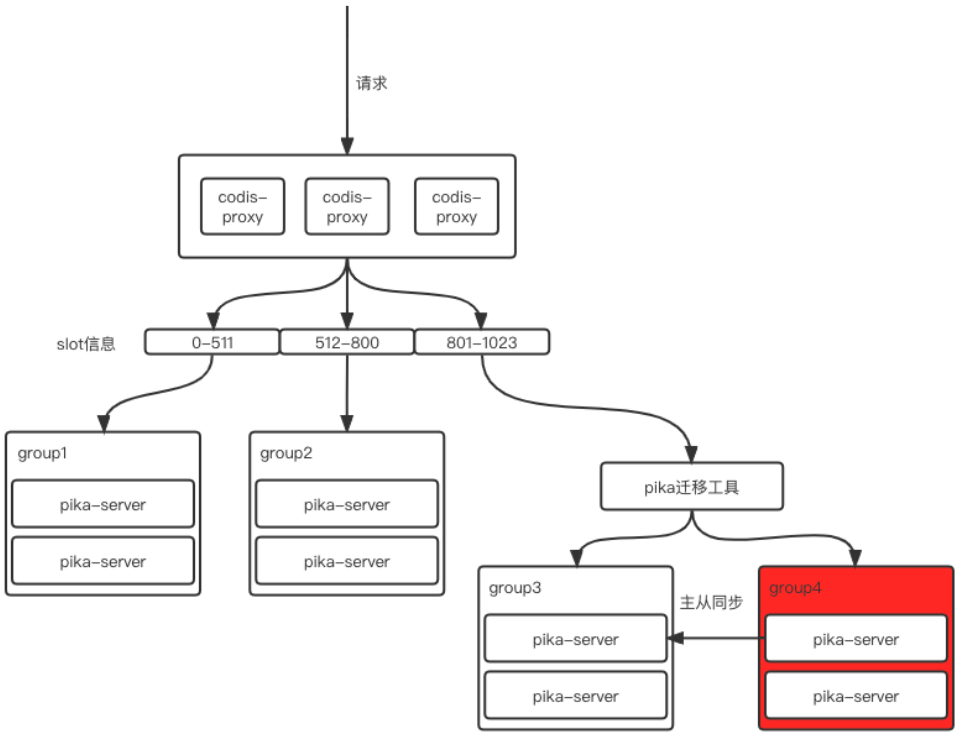


实战五：使用Pika实现Codis存储成本降低90%的实践

- 此时Pika迁移工具将801-900的读写信息写入Group3。
- 在Pika迁移工具中，将901-1023的读写信息同时指向Group4和Group3。然后进入下一步。

➤ Step3: 主从同步数据并动态切换主从

此时Pika迁移工具已经完成接入，它将转发801-1023的slot请求到后端。

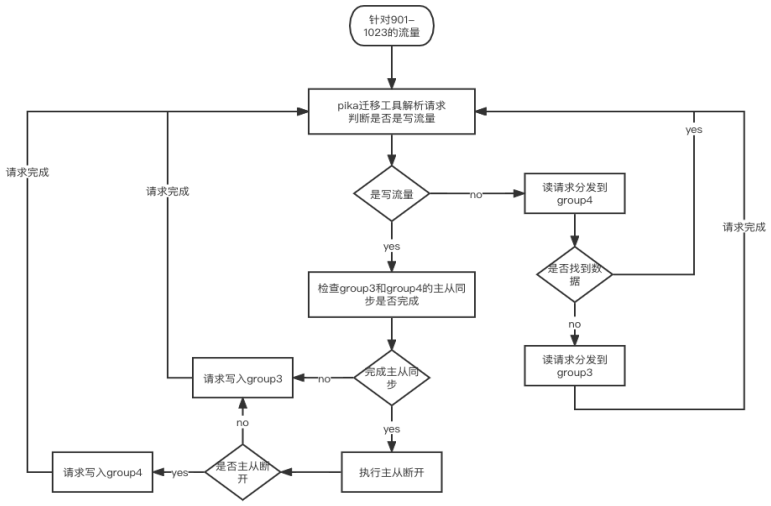


这里需要注意，Pika迁移工具在处理写流量时，会检查主从同步是否完成。

- 如果主从同步完成，Pika迁移工具会直接将Group4中Pika实例的从断掉，并将新数据写入到Group4中，否则就继续将写入的数据路由到Group3。
- 如果是读流量，Pika迁移工具会先尝试获取Group4的数据，如果获取到则返回，否则就去Group3获取数据。
- 如果901-1023的slot中没有写流量，则无法判断该slot主从同步是否完成以及是否要断开主从，那么我们可以向Pika迁移工具发送针对该slot的命令来执行该操作。
- 直到Group4中所有slot的主从同步完成且主从断开，方进行下一步。

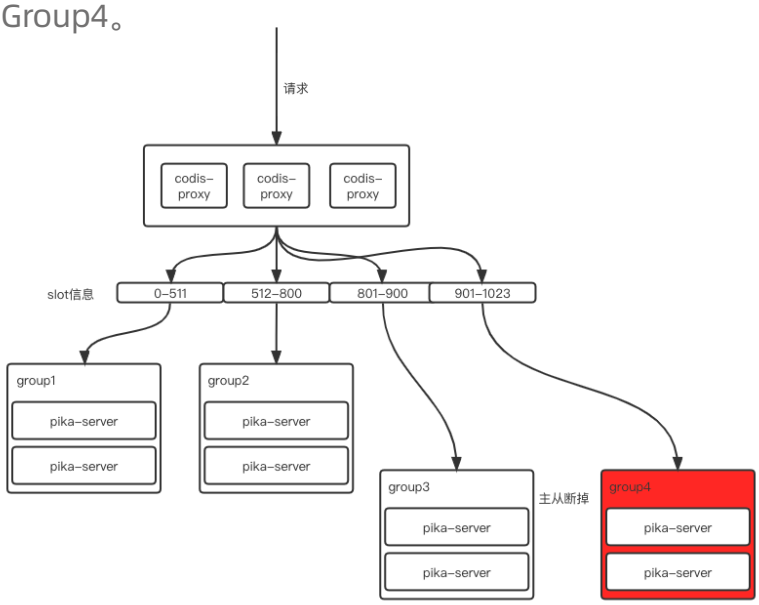
实战五：使用Pika实现Codis存储成本降低90%的实践

下图比较形象地展示了Pika迁移工具的作业逻辑：



➤ Step4: 将待迁移的slot迁入新的Group

在完成步骤3之后，再将Pika迁移工具的slot信息，即801-900，迁移回Group3，将901-1023迁移到Group4。



将901-1023完全迁移到Group4之后，就可将原来Group3中冗余的旧数据删除。至此，我们通过Pika迁移工具完成了对kv集群的扩容。

这里需要说明的是，Pika迁移工具的大部分功能和codis-proxy相似，只不过需要将对应的路由规则进行转换，并添加上支持Pika的语法指令。之所以能够如此设计实现，是因为**在codis-proxy的迁移过程中产生的都是原子性命令的操作，从而能够在Pika迁移工具这一层拦截目标端的数据，并动态地将数据写入到对应的集群中。**

实战五：使用Pika实现Codis存储成本降低90%的实践

方案效果实测

经过以上一系列的操作之后，我们成功使用Pika替换了原有的codis-server。那么我们预先的兼顾成本与性能的目标是否有达成呢？

首先，在性能方面，根据线上业务方的使用反馈，**当前总体的业务服务p99值为250毫秒**（包括对Codis和Pika的多次操作），能够满足当前现网对性能的需求。

再看成本方面，由于存储的key的数据结构类似，占用的实际物理空间基本相同。通过将Pika的数据转换成codis-server的存储量，**内存使用大概为24/4*82 = 480G的内存空间。**

技术方案	数据量与存储情况	集群规模
Codis-server	约4亿key，占用内存为总共约82.02G（主节点）	1主1从
Pika	约24.5亿key，占用硬盘空间总共约为1400G（主节点）	2主2从

根据当前的运维经验，如果实际存储480G的数据，按照每个节点存储10G数据，单节点最大15G，需要48个节点，即需要256G*6台机器（3主3从）提供服务。

存储的数据量	原生codis架构规划	pika架构规划
480G	单节点存储10G，最大内存15G，需要48个节点，即每台机器部署16个节点，即需要256G*6机器(3主3从)	4台机器(2主2从），ssd采用2主2从，存储ssd8.8T（raid10）存储，两台主节点共1400G，各使用700G
>480G	需要在3主3从的集群规模上继续扩展机器	预估容量可以扩展7倍，即总量约150亿数据，在150亿数据量以下ssd都能存下

这样我们就可以得出结论：

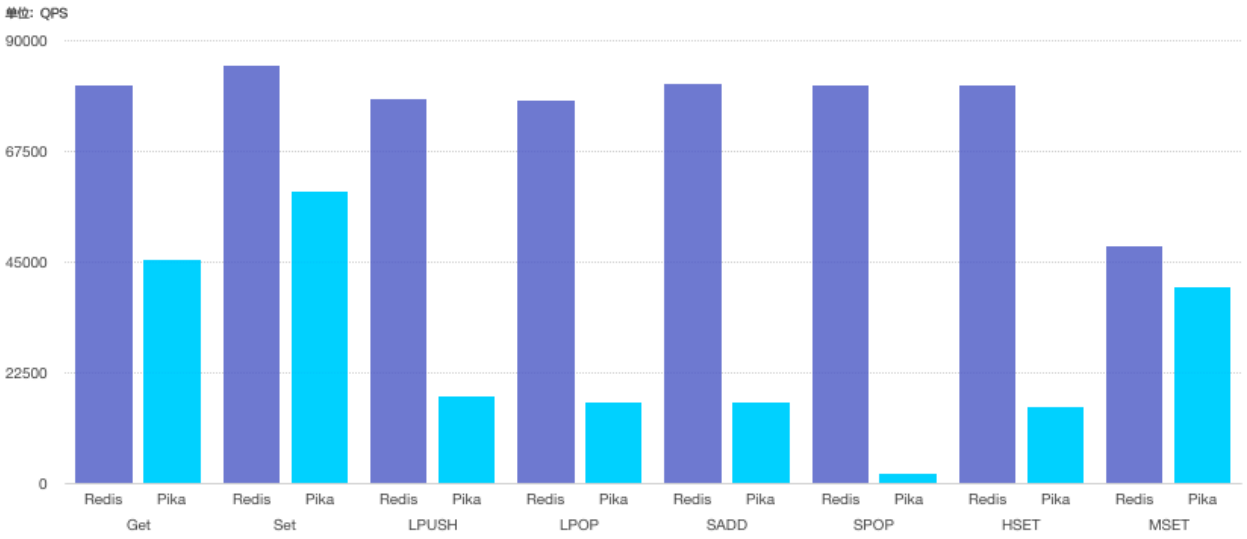
存储同等容量的数据，使用Pika的花费成本仅为Codis的5～10%！

实战五：使用Pika实现Codis存储成本降低90%的实践

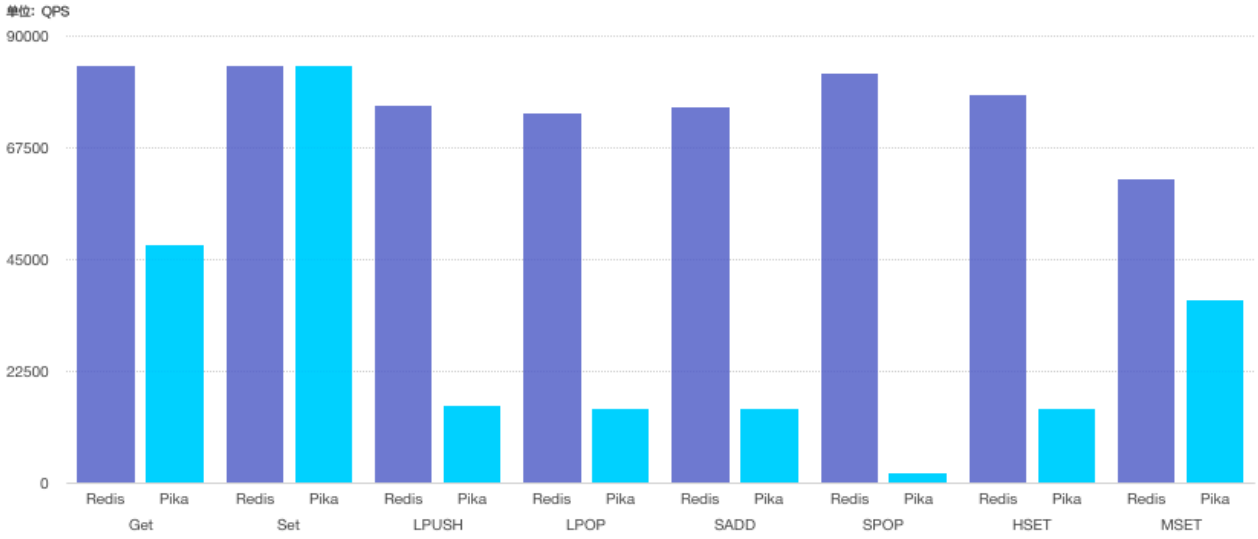
真诚的选型建议

我们还对Pika的单实例与Redis的单实例进行了性能压测对比。

压测命令为redis-benchmark -r 1000000000 -n 1000 -c 50时，性能表现如下：



压测命令为redis-benchmark -r 1000000000 -n 1000 -c 100时，性能表现如下：



从测试环境的压测结果来看，相对而言，单实例压测情况下，Redis表现占优；使用Pika的场景建议为kv类型性能较好，在五种数据结构里面推荐使用String类型。

实战五：使用Pika实现Codis存储成本降低90%的实践

综合压测数据和现网情况，我们对Codis + codis-server和Codis + Pika两种技术栈的优缺点进行了总结：

技术栈	优点	缺点
Codis + codis-server	1. 适合数据量相对较大，响应时延要求较高的场景。 2. slot动态管理，可运维性好。 3. Redis周边生态丰富，运维使用可定制性好。	1. 单个节点的Redis内存不能够太大，太大的内存性能管理的开销反而会拖累性能。 2. Redis需要较贵的内存来存储，使用成本较高。 3. 当前Codis项目官方停更，社区越来越不活跃。
Codis + Pika	1. 支持的kv容量比Codis大指数级。 2. 存储成本较低。 3. Pika进入了开放原子开源基金会，后续社区Pika迭代运维都相对有保障。 4. 当前公司基于Codis的架构可无缝接入Codis+Pika架构，对应用层无影响。	1. 性能没有Codis性能好。 2. 由于使用rocksdb，需要定期的compact来进行数据落盘和空间清理，会对性能有部分影响。 3. 官方对Pika在Codis的使用场景支持还待进一步加强。

针对如上对比，我们的选型建议如下：

场景	选型
延时要求高，访问抖动小，容量在几百G（五种数据结构性能都要求较高）	Codis
延时要求不高，允许性能有些许抖动，容量在百G级别（针对数据结构为String的kv类型）	Pika（经典模式或分布式模式），Pika主从
延时要求不高，允许性能有些许抖动，容量在几十T或T级别（针对数据结构为String的kv类型）	Codis+Pika（分布式模式）

总结

以上是个推使用Pika替换codis-server，以低成本实现海量kv数据存储与读写的实战过程。

个推还将持续关注性能与成本的平衡之道，希望我们的实战经验能帮助大数据从业者们更快地找到大数据降本提效的最优解。

透明存储实践

列式存储（Column-oriented Storage）是大数据场景中面向分析型数据的主流存储方式。与行式存储相比，列式存储只提取部分数据列、同列同质数据，具有更好的编码及压缩方式。目前，个推的核心数据正逐步切换为Parquet等新型数据格式存储以获得更高的I/O性能和更低的存储成本。

本文主要说明Parquet存储格式的收益，以及透明存储如何让历史工程实现数据格式的兼容和切换。

「扫码了解实战详情」

「观看实战讲解视频」

标签存算在
每日治数平台的实践之路

每日互动依托海量数据资源和强大的建模能力，形成3,000余种数据标签，构建了丰富、立体、多维的画像标签体系，从而为行业客户提供数据洞察相关服务，比如APP精细化运营、广告投放人群定向等。

由于业务方的标签组合复杂多样，所以在对大规模数据进行计算和标签构建的过程中，如何加速标签计算，实现秒级人群圈选和洞察便成为我们需要攻克的难题。

本文基于每日治数平台DIOS的开发实践，为您分享有效提升标签存算以及人群圈选效率的核心技术手段。

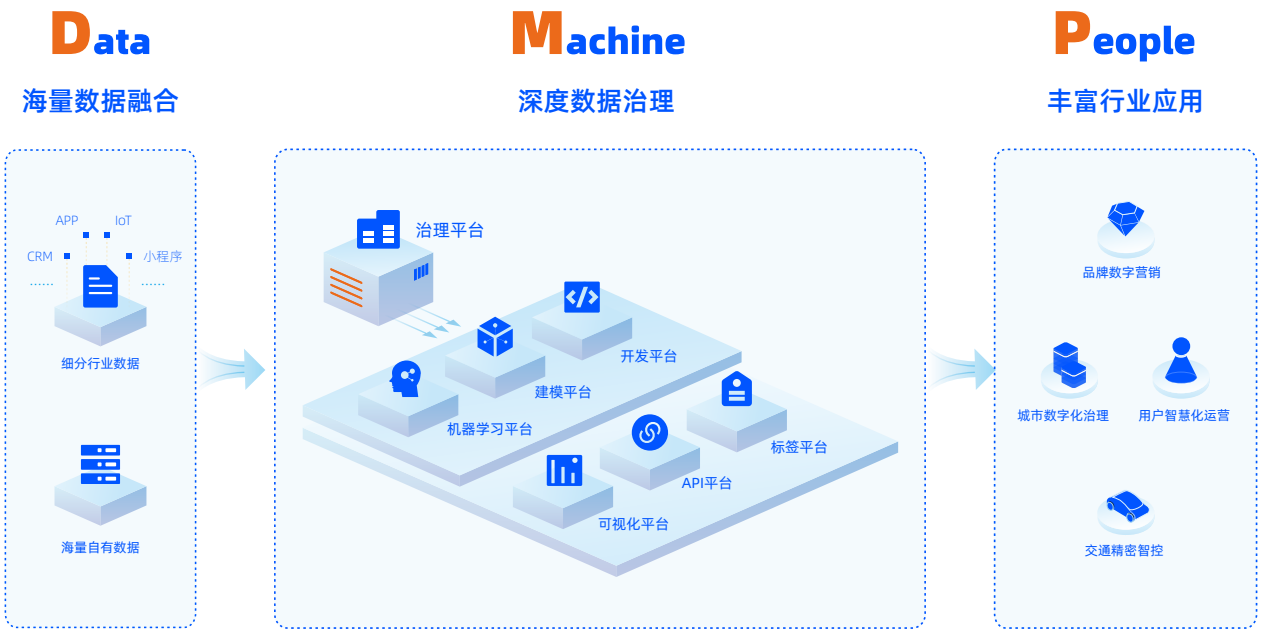
「扫码了解实战详情」

「观看实战讲解视频」

每日治数平台

每日治数平台DIOS（Data Intelligence Operation System）是每日互动推出的数据智能操作系统，它将数据智能的落地过程极致简化，以安全、智能、实时、可视等特性，全新定义数据治理、开发、分析和应用方式，致力于解决企业数据运用难、数据体系缺失、数据源质量差等痛点难题，助力行业数字化敏捷升级。

平台架构



应用场景

政务服务

人口精细化管理 | 基层网格化治理 | 风险企业发现

智慧高速

高速数据治理 | 差异化收费
定向信息发布 | 智能道路疏通

品牌营销

人群深度洞察 | 广告投放定向
线下商圈分析 | 营销归因分析

互联网

用户洞察分析 | 多渠道用户触达
智能拉新促活 | 高效流量变现

普惠金融

智慧运营 | 全渠道精准营销
智能风控 | 数字化高效获客

出品团队

每日互动 大数据平台架构团队

负责每日互动大数据平台的研发工作，设计分布式存储、分布式计算、分布式数据库等大数据平台架构，并基于大数据生态圈开源组件定制优化，助力全公司各项业务的降本提效。

每日互动 治数平台部

全面负责每日互动「数据智能操作系统」每日治数平台（DIOS）的研发，为政府相关部门及行业客户提供数据智能解决方案，致力于帮助行业实现数字化敏捷升级。

添加个推企业微信
了解每日治数平台（DIOS）产品详情

欢迎相关领域的技术牛人加入
简历投递：hrzp@getui.com

关于我们

每日互动股份有限公司（个推）成立于2010年，是专业的数据智能服务商，致力于用数据让产业更智能。公司将深厚的数据能力与行业“Know-How”有机结合，为互联网运营、用户增长、品牌营销、金融风控等各行业客户以及政府部门，提供丰富的数据智能产品、服务与解决方案。公司于2019年3月登陆创业板（股票代码：300766），成为国内率先在A股上市的“数据智能”企业。

每日互动聚焦数据智能赛道十余年，构建了“数据积累-数据治理-数据应用”的服务生态闭环。以开发者服务为基础，公司不断夯实数据底层，强化数据能力，为互联网客户提供便捷、稳定的技术服务与智能运营解决方案。同时，公司通过构建“每日治数平台DIOS”，将数据挖掘、萃取和治理能力向各行各业输出，帮助合作伙伴将数据资源打造成为数据资产，并进一步实现数据的价值兑换。多年来，公司将数据能力深度运用于各细分业务场景，沉淀了深厚的行业知识和丰富的服务经验，并打造了面向企业和政府部门的一系列数据智能产品与解决方案，增能各行业数字化升级。

作为数据智能领域的创新者和实践者，每日互动始终坚持“每日生活 科技改变”的初心，积极通过技术和数据的力量，为客户和社会创造更多价值。

倾情推荐

近几年互联网、物联网高速发展，很多企业积累了海量数据。有了海量数据，企业如何在开展数字运营过程中把数据资产快速用起来？在满足计算效率的同时降低数据存算成本？成为当今许多公司所面临的技术新挑战。个推《大数据降本提效实战手册》，是个推基于多年大数据实战经验，提炼出的大数据降本提效武功秘籍，非常值得一读，也欢迎行业专家指导、交流。

——个推 每日治数平台副总经理 袁凯

随着互联网行业的红利减退，互联网企业从以前的追求规模增长，转变为追求净利润。在此大背景下，各大公司的技术部门开始从多个层面开启“降本提效”之路。个推作为一家数据智能企业，服务器的规模接近5000台，在“降本提效”方面有不少比较容易落地的实战经验，借此机会分享给同行，希望能与大家多多探讨。

——个推 平台运维部总监 孟显耀

营收、效率、成本、质量是企业最关注的维度。对于互联网公司来说，计算和存储的服务器花费占据了不小的成本比重，当业务团队全力以赴做大营收的同时，技术团队也应撸起袖子扛起提效和降本的大旗，不断精进和优化。个推技术部的宗旨是业务成功、技术进步、团队成长，也借《大数据降本提效实战手册》和圈内的各位新老同学一起交流成长。

——个推助理CDO、数据资产团队负责人　段永康

个推GeTui
微信公众号

个推技术实践
微信公众号

添加个推小助手
加入技术交流社群

TECH

每日互动

股票代码：300766