一、项目概述

随着大数据技术的不断发展，各种云产品相继推出多元化的大数据存储解决方案。作为新一代数据编排组件，Alluxio 可以作为分布式缓存组件，实现对各种云产品的无缝集成和兼容，带来了可观的性能提升。然而，虽然 Alluxio 在离线计算与在线计算的场景下表现出良好的性能特点，但目前对于其集成不同云产品所带来的性能提升缺乏充分的测评，导致潜在客户缺乏清晰的测试结果，无法选择具有指导意义的信息进行产品选型。

本项目旨在研究和分析数据编排系统 Alluxio 在实际场景中的应用和性能表现。为此，我们搭建了两套数据系统，一套包含 Alluxio 作为数据编排层，另一套不包含 Alluxio。通过进行基准测试（使用 TPC-DS 标准）和具体场景下的测试（量化交易场景）来比较两套系统在各方面的性能和表现。最终对测评结果进行横向对比分析，总结出 Alluxio 在绝大多数情况下都能有效地提升数据系统的性能。

具体步骤:

1. 系统搭建: 我们在实验环境中搭建了两套数据系统，一套包含 Alluxio 作为数据编排层，另一套不包含 Alluxio。搭建过程中确保系统配置和硬件规格相似，以便进行公平的比较。
2. 基准测试: 我们使用 TPC-DS 标准来进行基准测试，该标准模拟了典型的决策支持查询场景。我们分别在两套系统上执行相同的查询集合，并主要记录以下指标作为性能评估的依据：挂钟时间。
3. 具体场景测试: 我们选择了量化交易场景作为具体场景测试的应用场景。与量化公司 Altopia 合作，通过实验，定量分析在量化交易领域，数据编排系统 Alluxio 的性能提升方面的作用。
4. 结果分析: 通过比较两套系统在基准测试和具体场景测试中的性能表现和指标，我们将分析 Alluxio 在数据编排和场景应用方面的优势和适用性。我们将评估 Alluxio 对数据系统性能方面的影响，并提供具体的数据和统计结果支持。

该项目将为数据系统的设计和优化提供有价值的指导，帮助用户更全面地了解和应用 Alluxio 在实际场景中的潜力和价值。同时，该研究成果还将为公司的产品改进提供数据支持，并为公司宣传增添有力的支撑，有助于潜在客户更好地理解、选择和使用 Alluxio 产品，提升产品在市场的竞争力。最后，通过本项目的研究成果，我们也希望能够促进数据系统和数据管理领域的发展和创新，为行业的进步做出贡献。

二、系统架构

存算耦合架构在资源分配上缺乏足够的灵活度，造成不必要的浪费。伴随大数据技术的高速发展，网络性能显著提高，云计算作为一种通过网络向用户提供计算资源的服务，为存算分离提供了条件。存算分离旨在实现数据计算与存储系统独立管理和调度，与存算耦合相比，存算分离架构能够独立扩展计算和存储，具有更高的弹性和资源共享程度，也更有利于降低整体成本。目前存算分离架构已经被各大公有云厂商广泛使用，从存算耦合到存算分离已然成为趋势。本次作业采用存算分离架构，在物理层面上将计算引擎和底层存储分离，通过数据编排技术，加速基于云的数据分析。

2.1 存储层：对象存储

对象存储是一种以非结构化格式（称为对象）存储和管理数据的技术。云对象存储系统将这些数据分布在多个物理设备上，但允许用户从单个虚拟存储库有效地访问内容。对象存储提供了基于桶和对象的扁平化存储方式，对象存储系统可以用桶和对象来表征。桶（Bucket）是存储对象的容器，对象保存在单个存储桶中，桶中的所有对象都处于同一逻辑层。如图 2.1 所示，对象存储去除了文件系统中的多层级树形目录结构，这将创建一个称为存储桶的平面结构，而不是分层或分级存储。

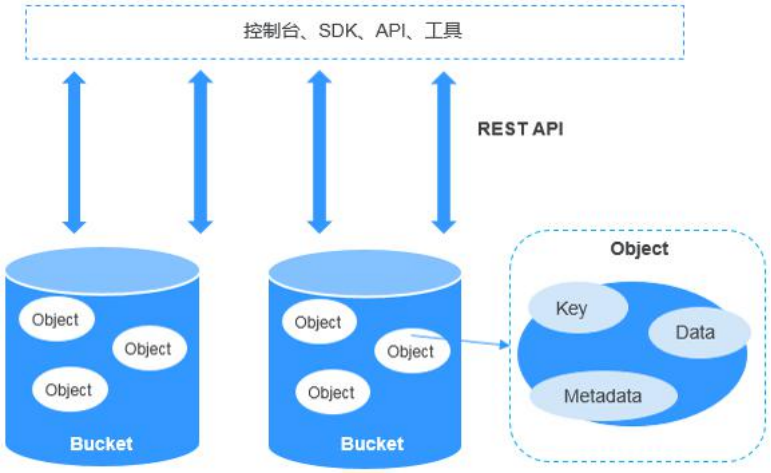

图 2.1 对象存储结构

对象存储服务是一种基于对象的海量存储服务。对象存储服务为客户提供海量、安全、高可靠、低成本的数据存储能力，包括创建、修改、删除桶，上传、下载、删除对象等。对象存储以 REST 接口的形式提供服务， REST 接口使用标准的 HTTP 标头和状态代码，使用标准的 HTTP 请求（GET、POST、PUT、DELETE 等）创建、提取和删除存储桶和对

象。借助 REST 接口，用户能够使用任何网页浏览器和 HTTP 工具包在 Web 上和对象存储服务进行交互。用户也可以使用各大云服务厂商提供的客户端访问对象存储服务。

Amazon 最早提供对象存储服务——Amazon S3，Amazon S3 使用 REST 接口。Amazon 制定的对象存储协议称为 S3 协议。目前市面上流行的对象存储服务基本兼容 S3 协议。本次作业中，我们采用 Amazon S3 作为分布式对象存储系统。

2.2 数据仓库层：Hive/Hadoop

Hive 是一个基于 Hadoop 的数据仓库工具，Hive 的设计目标是提供一个简单、易用的接口来处理大规模数据。它通过将查询转换为 MapReduce 任务来实现高性能的数据处理。Hive 的架构由三个主要组件组成：Hive 客户端、Hive 服务器和 Hive 元数据存储。三个组件共同构成了 Hive 的体系结构，提供了一个方便的数据仓库基础设施，使用户能够使用 SQL 类似的语言对大规模数据进行查询、处理和分析。

Hive 客户端是与 Hive 交互的用户界面，通常是通过命令行界面（CLI）或图形用户界面（GUI）提供的。Hive 客户端允许用户执行 Hive 查询和管理 Hive 表、分区、视图等。它提供了一种类似于 SQL 的查询语言，称为 HiveQL（Hive Query Language），允许用户使用 SQL 语法进行数据查询和处理。Hive 客户端将用户提交的查询转换为 Hive 任务，并将其发送到 Hive 服务器执行。

Hive 服务器是运行 Hive 服务的主机或进程。它接收来自 Hive 客户端的查询请求，并将其转发给底层的 Hadoop 集群进行处理。Hive 服务器负责协调查询的执行，并将结果返回给 Hive 客户端。Hive 服务器可以以不同的方式部署，例如独立模式、本地模式或与 Hadoop 集群中的其他组件（如 YARN）集成的远程模式。

Hive 元数据存储是 Hive 使用的关键组件，它用于存储有关 Hive 表、分区、列、数据位置等的元数据信息。Hive 元数据存储记录了 Hive 表的结构和属性，并跟踪数据在 HDFS 上的位置和格式。它还包含有关 Hive 表的统计信息，可以用于查询优化和执行计划生成。Hive 元数据存储通常使用关系型数据库（如 MySQL 或 Derby）来存储元数据，通过 Hive Metastore 服务进行管理和访问。

本次作业在创建数据库和数据表时使用了 Hive 客户端和 Hive 服务器，并且在 Hive 元数据存储更新存储表、分区、列等元数据信息。当计算引擎驱动数据查询和分析任务时，Hive 元数据存储跟踪数据在 HDFS 上的位置和格式，为用户提供方便的数据分析和查询能力。

由于 Hive 的运行依赖于 Hadoop，所以需要安装 Hadoop。Hive 使用 Hadoop 的 MapReduce 框架或 YARN 资源管理器来执行查询和数据处理任务。本次作业需要配置 MapReduce 和 YARN，包括设置作业调度、资源管理等参数。Hive 需要使用 Hadoop 的核心配置文件，例如 core-site.xml、hdfs-site.xml、mapred-site.xml（对于 MapReduce）、

yarn-site.xml（对于 YARN）。这些配置文件包含 Hadoop 集群的相关设置，如 HDFS 和 MapReduce（或 YARN）的连接信息、存储位置、日志路径等。Hive 需要连接到类 Hadoop 的底层存储服务，因此需要配置底层存储服务的主机名、端口号等连接信息，以确保 Hive 能够与底层存储服务进行通信。

2.3 数据编排层：Alluxio

存算分离将存储系统和计算引擎拆分为独立的模块，如图 2.2 所示，Alluxio 用于存储系统和计算引擎之间，**连接多个数据存储系统，为上层数据驱动型应用提供统一的客户端接口和全局数据访问服务**。Alluxio 为计算型应用（如 Apache Spark、Presto）和存储系统（如 Amazon S3、Alibaba OSS）的数据访问构建了桥梁，详见 Alluxio 官网 https://www.alluxio.io/。

Alluxio 支持多种数据源，如 HDFS、S3、OSS 等，可以将这些数据源中的数据缓存到内存或磁盘等介质中，以提高数据的访问速度和效率。Alluxio 并非简单的将底层存储系统的进行拷贝，而是根据其访问热度生成数据副本，以数据块（Block）的形式存储，且访问频率越高，生成的数据副本越多；反之，访问频率低的数据块会逐渐被替换出 Alluxio 存储层；而对于普通的存储系统（如 HDFS），通常只会生成固定数量的数据副本，无法根据热度动态使用存储资源。

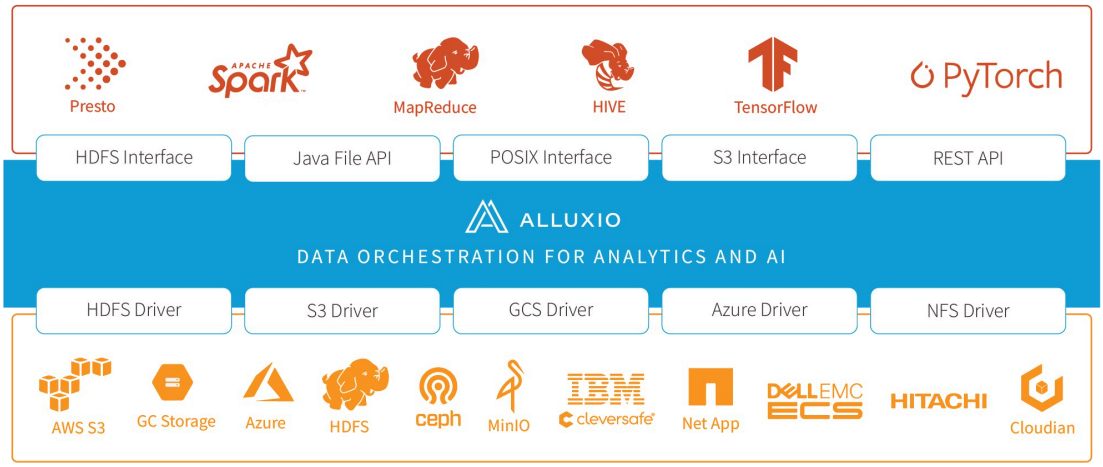

图 2.2 Alluxio 架构

Alluxio 基于客户端-服务端架构，应用程序通过 Alluxio 客户端与服务端进行交互。如图 2.3 所示，Alluxio 服务端由 Alluxio master、Alluxio worker 构成。Alluxio master 负责管理 Alluxio 文件系统元数据，处理元数据请求和集群注册。Alluxio worker 管理分配内存、硬盘等本地资源，从 Alluxio 存储或底层文件存储读写数据。Alluxio 客户端为用户提供了一个与 Alluxio 服务器交互的网关。Alluxio 客户端与 Alluxio master 通信以执行元数据操作，Alluxio 客户端与 Alluxio worker 通信以读取和写入存储在 Alluxio 中的数据。

图 2.3 Alluxio 集群

Alluxio 客户端提供多种多样的接口，其中包括 FUSE 基础上的 POSIX 接口和文件系统接口。Alluxio FUSE 服务通过提供 Unix/Linux 下的标准 POSIX 文件系统接口，让应用程序在不修改代码的前提下，以访问本地文件系统的方式访问 Alluxio 分布式文件系统中的数据。

Alluxio-FUSE 可以在一台 Unix 机器上的本地文件系统中挂载一个 Alluxio 分布式文件系统。通过使用该特性，一些标准的命令行工具（例如 ls、 cat 以及 echo）可以直接访问 Alluxio 分布式文件系统中的数据。Alluxio 为数据驱动型应用程序（如 Spark、Trino 等）提供兼容 Hadoop 文件系统的接口。Alluxio 文件系统接口实现了 Alluxio 集群和类 Hadoop 底层存储集群的无感知转换，使得底层存储集群的数据可以透明地缓存到 Alluxio 分布式存储中。

2.4 计算层：Trino

我们采用 Trino 计算引擎。Trino 是一个开源的分布式 SQL 查询引擎，可以跨多个服务器并行处理数据，在大规模数据上运行交互式分析查询。

Trino 执行 SQL 语句并将这些语句转换为查询，这些查询在 Coordinator 和 Worker 的分布式集群上执行。如图 2.4 所示，Trino 集群由一个 Coordinator 和许多 Worker 组成。Trino Coordinator 是负责解析语句、规划查询和管理 Trino 工作节点的服务器。Coordinator 是客户端连接以提交执行语句的节点，Trino 用户使用 SQL 查询工具连接到 Coordinator。Coordinator 跟踪每个 Worker 的活动并协调查询的执行。Coordinator 创建涉及一系列阶段的查询的逻辑模型，然后将其转换为在 Trino 工作集群上运行的一系列连接任务。Trino Worker 负责执行任务和处理数据。Worker 节点从连接器获取数据并相互交换中间数据。Coordinator 负责从 Worker 那里获取结果并将最终结果返回给客户端。Worker 使用 REST API 与其他 Worker 和 Trino Coordinator 进行通信。

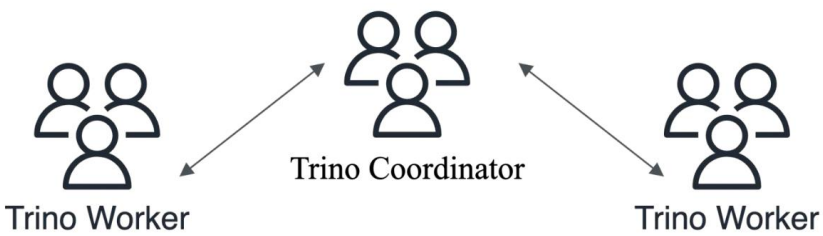

图 2.4 Trino 集群

2.5 整体架构

根据上述介绍，系统整体架构从上至下包含了计算层、数据仓库层、缓存层和底层存储层。计算层采用数据分析引擎 Trino，数据仓库层采用 Hive 以及 Hadoop 配置文件，数据编排层采用 Alluxio，存储层采用对象存储的范例——Amazon S3。

2.5.1 两种部署方式

为了验证 Alluxio 在存算分离系统中的有效性，我们搭建了两组测试环境：

1) 计算引擎 Trino 直接查询底层存储 S3（如图 2.5 所示）。

2) 通过 Alluxio 缓存，计算引擎 Trino 间接查询底层存储 S3（如图 2.6 所示）。

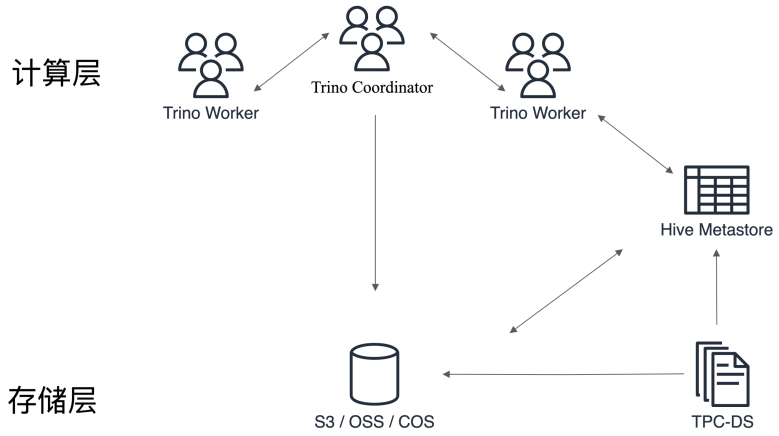

图 2.5 Trino + Hive/Hadoop + S3

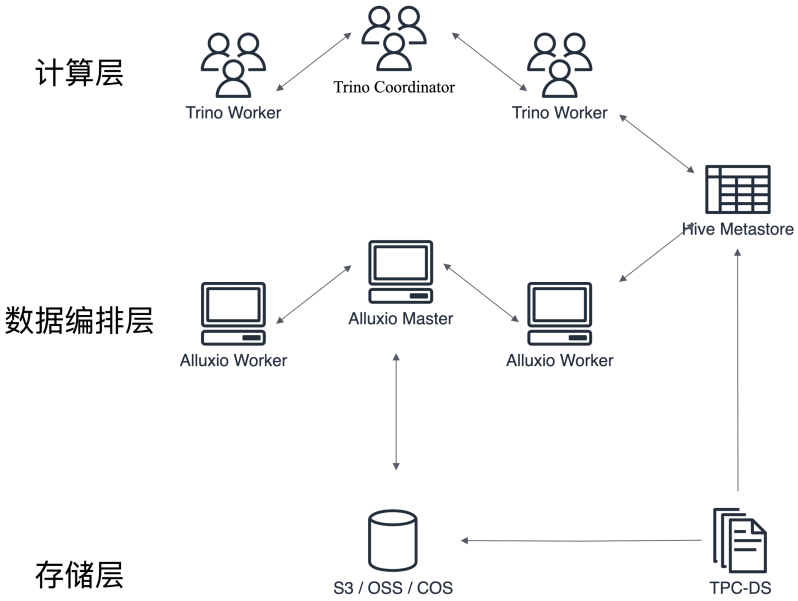

图 2.6 Trino + Hive/Hadoop + Alluxio + S3

2.5.2 Alluxio 与 Trino 联合部署

Alluxio 作为分布式缓存层，将 Trino 访问的数据透明地缓存到 Alluxio 分布式存储中。如图 2.7 所示， 将 Alluxio master 与 Trino coordinator 部署在一起，将 Alluxio worker 与 Trino worker 部署在一起，可以提高数据本地性，避免网络传输引发的时延，从而加速数据 I/O。

图 2.7 Alluxio 与 Trino 联合部署

三、数据分析

3.1 TPC-DS 数据集

3.2 量化交易数据集

3.1.1 简介

我们采用 2020 年 1 月 2 日的 A 股交易数据，共有 20 多万条数据，包含以下 16 个指标：日期，股票代码，前一日收盘价，开盘价，最高价，最低价，收盘价，成交量，成交额，成交笔数，调整因子，涨停价，跌停价，总股本，流通股本，自由流通股本。这些字段提供了关于股票交易数据的重要信息，可以用于分析股票价格的变动、成交活跃度以及股本结构等方面的情况。

如下图所示，取第一条数据为例，在 2020 年 1 月 2 日这天，代码是 000001.SH 的股票，其开盘价是 24.08，最高价与最低价分别是 64.19 和 20.20，收盘价是 54.27 等。

	date_day	sym	prevclose	open	high	low	close	volume	turnover	tradecount	af	upperlimit	lowe
0	2020/1/2	000001.SH	00001.SZ	24.08	69.14	20.20	54.37	47.85	36.42	11.04	56.74	3.87	
1	2020/1/2	000001.SZ	00001.SZ	41.18	58.75	69.50	16.86	97.72	45.39	33.43	20.42	29.50	
2	2020/1/2	000002.SH	00001.SZ	30.68	21.37	43.01	37.10	10.12	92.32	26.22	22.41	33.30	
3	2020/1/2	000002.SZ	00001.SZ	90.61	46.97	62.82	30.12	34.61	68.18	74.01	23.99	33.16	
4	2020/1/2	000003.SH	00001.SZ	55.95	5.26	16.19	91.84	79.97	92.53	59.91	23.35	41.37	
...	...	...	...	...	...	...	...	...	...	...	...	...	
218615	2020/1/3	S00959.SZ	000010.SZ	53.77	65.94	21.85	9.63	83.08	43.16	35.16	94.84	30.51	
218616	2020/1/3	S00972.CSI	000010.SZ	25.45	90.06	81.32	63.84	89.54	1.02	63.10	25.31	93.57	
218617	2020/1/3	SHHKSI.CSI	000010.SZ	88.43	48.21	33.86	38.65	51.11	61.79	19.90	89.74	63.42	
218618	2020/1/3	VALHKDG.CNI	000010.SZ	79.40	50.90	4.71	65.23	98.61	20.17	53.00	20.21	46.54	
218619	2020/1/3	YLVHKDG.CNI	000010.SZ	81.78	60.30	10.98	21.93	20.69	47.97	7.89	64.93	37.84	

218620 rows × 16 columns

图 3.1 量化交易数据集概览

此外，对数据集中的各项指标进行简单统计，结果如图 3.2 所示。

	open	high	low	close	volume	turnover	tradecount	af	upperlimit	lowerlimit	tt
count	218620.00	218620.00	218620.00	218620.00	218620.00	218620.00	218620.00	218620.00	218620.00	218620.00	21862
mean	50.59	50.54	50.58	50.48	50.45	50.56	50.48	50.42	50.50	50.49	5
std	28.59	28.59	28.56	28.58	28.63	28.62	28.58	28.59	28.53	28.60	2
min	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	
25%	25.83	25.74	25.86	25.64	25.67	25.73	25.72	25.60	25.89	25.73	2
50%	50.66	50.54	50.59	50.52	50.38	50.55	50.45	50.37	50.44	50.45	5
75%	75.37	75.24	75.35	75.27	75.30	75.31	75.23	75.16	75.20	75.32	7
max	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	10

图 3.2 量化交易数据集各项指标统计值

四、实验

启动环境需要运行的命令（一步也不能写错，否则实验将无法顺利进行！）

```
service ssh restart
source /etc/profile
cd /root
./docker-env-start.sh

# 挂载 Alluxio Fuse
mkdir -p /mnt/fuse
chown $(whoami) /mnt/fuse
chmod 755 /mnt/fuse
/root/alluxio/integration/fuse/bin/alluxio-fuse mount /mnt/fuse /
```

关闭环境

```
cd /root
./docker-env-stop.sh
```

4.1 环境搭建

4.1.1 服务器

使用实例类型：

- AWS（亚马逊云服务）
- 区域：美国东部 (弗吉尼亚北部)
- us-east-1
- r5a.2xlarge：CPU：8 核；内存：64GB；架构：x86_64

配置 root 用户和 SSH 登录：

```
# 为 root 添加密码
sudo passwd root

# 切换 root 用户
su root
```

```
# 进入 /etc/ssh/sshd_config，修改 PasswordAuthentication 和 PermitRootLogin 的值。
vim /etc/ssh/sshd_config
PasswordAuthentication yes
PermitRootLogin yes

# 启用 ssh 登录
service sshd restart
```

Docker

- 使用 ubuntu:latest 镜像

4.1.2 Hadoop 3.3.4

core-site.xml

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:8020</value>
    </property>
    <property>
        <name>hadoop.tmp.dir</name>
        <value>/var/hadoop</value>
    </property>
    <property>
        <name>hadoop.http.staticuser.user</name>
        <value>root</value>
    </property>
<property>
        <name>hadoop.proxyuser.namenode.hosts</name>
        <value>root</value>
    </property>
    <property>
```

```
        <name>hadoop.proxyuser.namenode.groups</name>
        <value>root</value>
    </property>
</configuration>
```

hdfs-site.xml

```
<configuration>
    <!-- nn web 端访问地址 -->
    <property>
        <name>dfs.namenode.http-address</name>
        <value>localhost:9870</value>
    </property>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
    <property>
            <name>dfs.client.use.datanode.hostname</name>
            <value>true</value>
            <description>only config in clients</description>
    </property>
</configuration>
```

yarn-site.xml

```
<configuration>
            <property>
                    <name>yarn.nodemanager.aux-services</name>
                    <value>mapreduce_shuffle</value>
            </property>
            <property>
                <name>yarn.resourcemanager.hostname</name>
                <value>localhost</value>
        </property>

        <property>
```

```
                    <name>yarn.nodemanager.aux-services</name>
                <value>mapreduce_shuffle</value>
        </property>
            <property>
                <name>yarn.nodemanager.resource.cpu-vcores</name>
                <value>2</value>
            </property>
            <property>
                    <name>yarn.nodemanager.resource.memory-mb</name>
                    <value>2048</value>
                    <description>每个节点可用内存,单位 MB</description>
            </property>
</configuration>
```

mapred-site.xml

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
</configuration>
```

hadoop-env.sh

Hadoop 建议配置 Java 8

```
export JAVA_HOME=xxx
export PATH=$PATH:$JAVA_HOME/bin
```

Hadoop 启动脚本

```
#!/bin/bash

case $1 in
"start")
        echo "------------------- 启动 hdfs -------------------"
        /root/hadoop/sbin/hadoop-daemon.sh start namenode
        /root/hadoop/sbin/hadoop-daemon.sh start datanode
```

```
        echo "------------------- 启动 yarn -------------------"
        /root/hadoop/sbin/yarn-daemon.sh start resourcemanager
        /root/hadoop/sbin/yarn-daemon.sh start nodemanager
        /root/hadoop/yarn-daemon.sh start nodemanager

        echo "------------------- 关闭安全模式 -------------------"
        /root/hadoop/bin/hdfs dfsadmin -safemode leave

;;
"stop")
        echo "------------------- 关闭 yarn -------------------"
        /root/hadoop/sbin/stop-yarn.sh
        echo "------------------- 关闭 hdfs -------------------"
        /root/hadoop/sbin/stop-dfs.sh

;;
*)
            echo "error"
;;
esac
```

4.1.3 Hive 3.1.3

初始化

```
cd ~
schematool -dbType derby -initSchema
```

hive 启动脚本

```
#!/bin/bash

HIVE_LOG_DIR=$HIVE_HOME/logs
if [ ! -d $HIVE_LOG_DIR ]
```

```
then
mkdir -p $HIVE_LOG_DIR
fi

#检查进程是否运行正常，参数 1 为进程名，参数 2 为进程端口
function check_process()
{
pid=$(ps -ef 2>/dev/null | grep -v grep | grep -i $1 | awk '{print $2}')
ppid=$(netstat -nltp 2>/dev/null | grep $2 | awk '{print $7}' | cut -d '/' -f 1)
echo $pid
[[ "$pid" =~ "$ppid" ]] && [ "$ppid" ] && return 0 || return 1
}

function hive_start()
{
metapid=$(check_process HiveMetastore 9083)
cmd="nohup hive --service metastore >$HIVE_LOG_DIR/metastore.log 2>&1&"
[ -z "$metapid" ] && eval $cmd || echo "Metastroe 服务已启动"
server2pid=$(check_process HiveServer2 10000)
cmd="nohup hiveserver2 >$HIVE_LOG_DIR/hiveServer2.log 2>&1 &"
[ -z "$server2pid" ] && eval $cmd || echo "HiveServer2 服务已启动"
}

function hive_stop()
{
metapid=$(check_process HiveMetastore 9083)
[ "$metapid" ] && kill $metapid || echo "Metastore 服务未启动"
server2pid=$(check_process HiveServer2 10000)
[ "$server2pid" ] && kill $server2pid || echo "HiveServer2 服务未启动"
}

case $1 in
"start")
hive_start
;;
"stop")
```

```
hive_stop
;;
"restart")
hive_stop
sleep 2
hive_start
;;
"status")
check_process HiveMetastore 9083 >/dev/null && echo "Metastore 服务运行正常"
|| echo "Metastore 服务运行异常"
check_process HiveServer2 10000 >/dev/null && echo "HiveServer2 服务运行正常"
|| echo "HiveServer2 服务运行异常"
;;
*)
echo Invalid Args!
echo 'Usage: '$(basename $0)' start|stop|restart|status'
;;
esac
```

4.1.4 Alluxio 2.9.0

软件及源码下载地址：https://github.com/Alluxio/alluxio/releases/tag/v2.9.0。

alluxio-site.properties

```
alluxio.master.hostname=Master01
alluxio.master.mount.table.root.ufs=s3://zhaozihao-us-east-2/alluxio/
alluxio.underfs.s3.region=us-east-2
s3a.accessKeyId=xxx
s3a.secretKey=xxx
```

alluxio-env.sh

alluxio 建议配置 Java 8

```
export JAVA_HOME=xxx
export PATH=$PATH:$JAVA_HOME/bin
```

4.1.5 Trino 433

Trino Coordinator

node.properties

```
node.environment=production
node.id=ffffffff-ffff-ffff-ffff-ffffffffffff0
node.data-dir=/root/trino/trino-01/data
```

jvm.config

```
-server
-Xmx12G
-XX:+UseG1GC
-XX:G1HeapRegionSize=32M
-XX:+UseGCOverheadLimit
-XX:+ExplicitGCInvokesConcurrent
-XX:+HeapDumpOnOutOfMemoryError
-XX:+ExitOnOutOfMemoryError
-Djdk.attach.allowAttachSelf=true
-Dfile.encoding=UTF-8
```

config.properties

```
coordinator=true
node-scheduler.include-coordinator=true
http-server.http.port=8090
discovery.uri=http://localhost:8090
```

log.properties

```
io.trino=INFO
```

etc/catalog/hive.properties

```
connector.name=hive
hive.metastore.uri=thrift://localhost:9083
hive.config.resources=/root/hadoop/etc/hadoop/core-site.xml,/root/hadoop/etc/ha
doop/hdfs-site.xml
```

Trino Worker 1

node.properties

```
node.environment=production
node.id=ffffffff-ffff-ffff-ffff-fffffffffff1
node.data-dir=/root/trino/trino-02/data
```

jvm.config

```
-server
-Xmx12G
-XX:+UseG1GC
-XX:G1HeapRegionSize=32M
-XX:+UseGCOverheadLimit
-XX:+ExplicitGCInvokesConcurrent
-XX:+HeapDumpOnOutOfMemoryError
-XX:+ExitOnOutOfMemoryError
-Djdk.attach.allowAttachSelf=true
-Dfile.encoding=UTF-8
```

config.properties

```
coordinator=false
http-server.http.port=8091
discovery.uri=http://localhost:8090
```

etc/catalog/hive.properties

```
connector.name=hive
hive.metastore.uri=thrift://localhost:9083
hive.config.resources=/root/hadoop/etc/hadoop/core-site.xml,/root/hadoop/etc/ha
doop/hdfs-site.xml
```

Trino Worker 2

node.properties

```
node.environment=production
node.id=ffffffff-ffff-ffff-ffff-fffffffffff2
```

```
node.data-dir=/root/trino/trino-03/data
```

jvm.config

```
-server
-Xmx12G
-XX:+UseG1GC
-XX:G1HeapRegionSize=32M
-XX:+UseGCOverheadLimit
-XX:+ExplicitGCInvokesConcurrent
-XX:+HeapDumpOnOutOfMemoryError
-XX:+ExitOnOutOfMemoryError
-Djdk.attach.allowAttachSelf=true
-Dfile.encoding=UTF-8
```

config.properties

```
coordinator=false
http-server.http.port=8092
discovery.uri=http://localhost:8090
```

etc/catalog/hive.properties

```
connector.name=hive
hive.metastore.uri=thrift://localhost:9083
hive.config.resources=/root/hadoop/etc/hadoop/core-site.xml,/root/hadoop/etc/hadoop/hdfs-site.xml
```

4.1.6 Amazon S3

S3 是云存储，但是如果要使用的话仍需进行下列若干步骤的配置。

配置 Hive 连接 S3

由于 Hive 依赖 Hadoop 运行，因此只需要配置 Hadoop 连接 S3 即可。

依赖包默认不在 hadoop classpath 下面。可以使用以下两种方法引入这两个包：

```
# 在 hadoop-env.sh 中加入 export
# 更改完毕后可以使用 hadoop classpath 确定。
```

```
HADOOP_CLASSPATH=$HADOOP_CLASSPATH:$HADOOP_HOME/share/hadoop
/tools/lib/*

# 通过软链接:
ln -s $HADOOP_HOME/share/hadoop/tools/lib/*aws*
$HADOOP_HOME/share/hadoop/common/lib/
```

在 hadoop/etc/hadoop/core-site.xml 中添加

注意要写 fs.s3.endpoint，不可以写 fs.s3a.endpoint！！！

```
<property>
    <name>fs.s3a.access.key</name>
    <value> xxx </value>
</property>
<property>
    <name>fs.s3a.secret.key</name>
    <value> xxx </value>
</property>
<property>
    <name>fs.s3a.connection.ssl.enabled</name>
    <value>false</value>
</property>
<property>
    <name>fs.s3a.path.style.access</name>
    <value>true</value>
</property>
<property>
    <name>fs.s3.endpoint</name>
    <value>https://zhaozihao-us-east-2.s3.us-east-2.amazonaws.com</value>
</property>
<property>
    <name>fs.s3a.impl</name>
    <value>org.apache.hadoop.fs.s3a.S3AFileSystem</value>
</property>
```

配置 Trino 连接 S3

```
hive.s3.path-style-access=true
hive.s3.endpoint=https://s3.us-east-2.amazonaws.com
hive.s3.aws-access-key=xxx
hive.s3.aws-secret-key=xxx
hive.s3.ssl.enabled=false
```

4.2 基准测试

4.2.1 简介

选择公开的、公认的、权威性较高的数据集和测试负载，定量分析在通常情况下，数据编排系统 Alluxio 在性能提升方面的作用。TPC-DS 数据集是 TPC 协会制作的用于 OLTP 基准测试的数据集。

4.2.2 数据集

生成数据

```
# 编译
apt-get install -y gcc make flex bison byacc git
make -f Makefile.suite
# 生成数据
./dsdgen   -scale 1 -dir ../../data/sf1
```

4.2.3 代码

共 25 张表，这里仅展示一个表的例子，其余的表格请见 github 代码仓库。

S3

```
create table dbgen_version (
   dv_version varchar(16),
   dv_create_date date,
   dv_create_time TIMESTAMP,
   dv_cmdline_args varchar(200)
```

```
) row format delimited fields terminated by '|' stored as textfile location
's3a://zhaozihao-us-east-2/alluxio/tpcds/sf1';
```

alluxio

```
create table dbgen_version (
   dv_version varchar(16),
   dv_create_date date,
   dv_create_time TIMESTAMP,
   dv_cmdline_args varchar(200)
) row format delimited fields terminated by '|' stored as textfile
location 'alluxio://localhost:19998/tpcds/sf1';
```

测试负载代码（共 20 个测试负载，这里只展示 1 个 q03.sql，其余的请见 github 代码仓库）

```
SELECT
   dt.d_year,
   item.i_brand_id brand_id,
   item.i_brand brand,
   SUM(ss_ext_sales_price) sum_agg
FROM date_dim dt, store_sales, item
WHERE dt.d_date_sk = store_sales.ss_sold_date_sk
   AND store_sales.ss_item_sk = item.i_item_sk
   AND item.i_manufact_id = 128
   AND dt.d_moy = 11
GROUP BY dt.d_year, item.i_brand, item.i_brand_id
ORDER BY dt.d_year, sum_agg DESC, brand_id
LIMIT 100;
```

本项目 Trino 直接读 S3 的自动化测试运行方法

```
/root/trino/trino-test/test07/auto-test.sh
```

本项目 Trino 通过 Alluxio 读 S3 的自动化测试运行方法

```
/root/trino/trino-test/test08/auto-test.sh
```

4.2.4 结果与可视化

我们进行对比测试，运用 Alluxio 的系统比不运用 Alluxio 的系统性能提升量为 27.86%

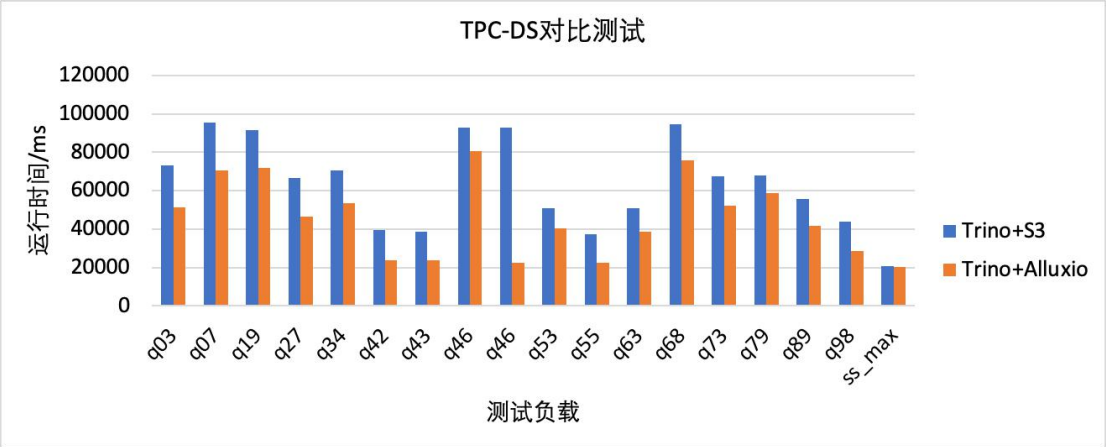

图 4.2.1 Trino+S3 与 Trino+Alluxio 性能对比

可以看到 Alluxio-2.10.0-SNAP 比 Alluxio-2.9.0 在性能方面有所提升。

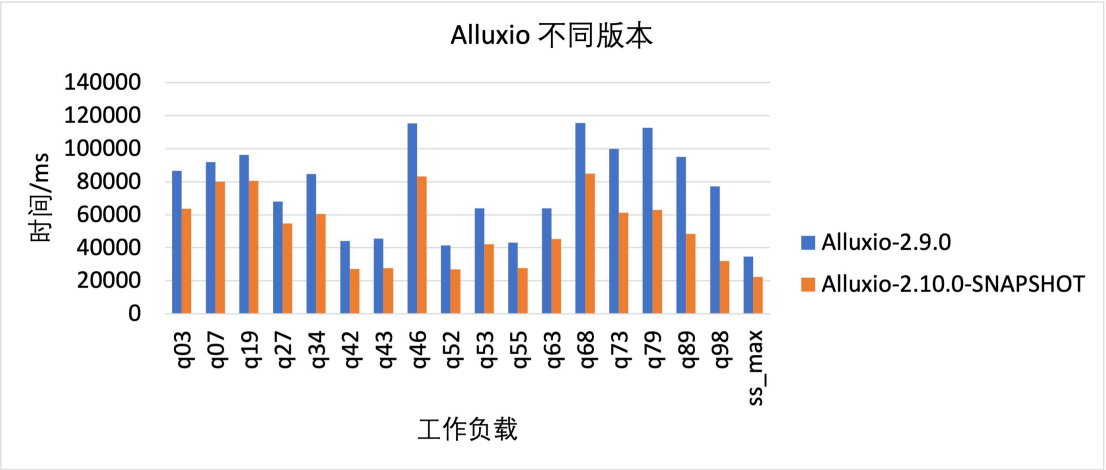

图 4.2.2 Alluxio-2.9.0 和 Alluxio-2.10.0-SNAPSHOT 性能对比

4.3 量化交易

4.3.1 简介

与量化公司 Altopia 合作，定量分析在量化交易领域，数据编排系统 Alluxio 的性能提升方面的作用。

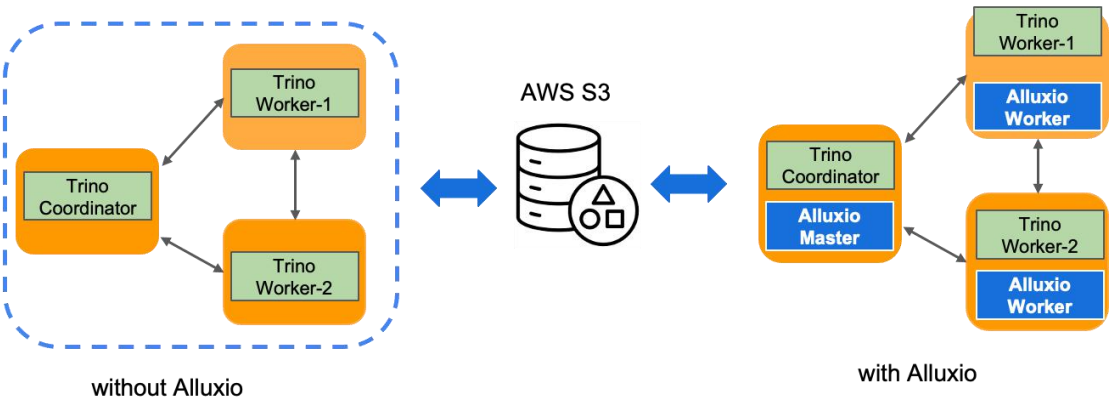

图 4.3.1 系统架构示意图

4.3.2 数据集

使用 Altopia 公司提供的 A 股交易数据。

4.3.3 代码

建表（仅列举一个 table 表结构，其余的请看 github 仓库的代码）

Trino 直接查询 S3

```
CREATE EXTERNAL TABLE IF NOT EXISTS csbar_1d_08 (
date_day VARCHAR(256),
sym VARCHAR(256),
prevclose DOUBLE,
open DOUBLE,
high DOUBLE,
low DOUBLE,
close DOUBLE,
volume DOUBLE,
turnover DOUBLE,
tradecount DOUBLE,
af DOUBLE,
upperlimit DOUBLE,
lowerlimit DOUBLE,
ttlshr DOUBLE,
fltshr DOUBLE,
```

```
frshr DOUBLE
)
row format delimited fields terminated by ','
STORED AS textfile
LOCATION 's3a://zhaozihao-us-east-2/alluxio/aitop/csbar_1d_08';
```

Trino 通过 Alluxio 查询 S3

```
CREATE EXTERNAL TABLE IF NOT EXISTS csbar_1d_08 (
date_day VARCHAR(256),
sym VARCHAR(256),
prevclose DOUBLE,
open DOUBLE,
high DOUBLE,
low DOUBLE,
close DOUBLE,
volume DOUBLE,
turnover DOUBLE,
tradecount DOUBLE,
af DOUBLE,
upperlimit DOUBLE,
lowerlimit DOUBLE,
ttlshr DOUBLE,
fltshr DOUBLE,
frshr DOUBLE
)
row format delimited fields terminated by ','
STORED AS textfile
LOCATION 'alluxio://localhost:19998/aitop/csbar_1d_08';
```

我们基于样例数据和样例查询：

两种查询类型：多表连接类查询和聚合计算类查询

多表连接类型

```
  select *, default.csbar_1d.af as "af"
      from default.csbar_1m
      left join default.csbar_1d
      on default.csbar_1m.date = default.csbar_1d.date and default.csbar_1m.sym =
default.csbar_1d.sym
```

聚合计算类型

```
select date,avg(volume) from csbar_1d group by date;
```

我们针对每一类查询，制做了 20 个数据表，每个表的大小都相同。

使用提供的 csbar_1d, csbar_1m, cstick 的原始数据提供的字段，date, sym, time 保持和原有的表格一致。其余交易数据随机生成。

多表连接运算和聚合运算累计查询的数据量约为 40GB

多表连接：执行运算之后将数据写入临时表，然后查询临时表（分别用 1 线程和 2 线程各测试一轮）。每个原始数据表生成 30 张临时表；每张临时表查询 50 次。（相当于总计查询 1500 次），实际执行的 sql 举例：

```
INSERT INTO result_test01_00_00
SELECT
csbar_1m_00.date_day AS "date_day",
csbar_1m_00.time_minute AS "time_minute",
csbar_1m_00.sym AS "sym",
csbar_1m_00.open AS "open",
csbar_1m_00.high AS "high",
csbar_1m_00.low AS "low",
csbar_1m_00.close AS "close",
csbar_1m_00.volume AS "volume",
csbar_1m_00.turnover AS "turnover",
csbar_1d_00.prevclose AS "prevclose",
csbar_1d_00.tradecount AS "tradecount",
csbar_1d_00.upperlimit AS "upperlimit",
csbar_1d_00.lowerlimit AS "lowerlimit",
csbar_1d_00.ttlshr AS "ttlshr",
csbar_1d_00.fltshr AS "fltshr",
csbar_1d_00.frshr AS "frshr",
csbar_1d_00.af AS "af"
FROM
```

```
csbar_1m_00 LEFT JOIN csbar_1d_00 ON csbar_1m_00.date_day =
csbar_1d_00.date_day AND csbar_1m_00.sym = csbar_1d_00.sym;

select * from result_test01_00_00;
select * from result_test01_00_00;
select * from result_test01_00_00;
select * from result_test01_00_00;
select * from result_test01_00_00;
select * from result_test01_00_00;
select * from result_test01_00_00;
select * from result_test01_00_00;
select * from result_test01_00_00;
select * from result_test01_00_00;
```

聚合运算：执行运算之后将数据写入临时表，然后查询临时表（分别用 1 线程和 2 线程各测试一轮）。每个原始数据表生成 30 张临时表；每张临时表查询 50 次。（相当于总计查询 1500 次），实际执行的 sql 举例：

```
INSERT INTO result_test02_00_00
select
date_day as "date_day",
avg(volume) as "avg_volume"
from csbar_1d_00 group by date_day;

select * from result_test02_00_00;
select * from result_test02_00_00;
select * from result_test02_00_00;
select * from result_test02_00_00;
select * from result_test02_00_00;
select * from result_test02_00_00;
select * from result_test02_00_00;
select * from result_test02_00_00;
select * from result_test02_00_00;
select * from result_test02_00_00;
```

然而在实际生产中，更常见的情景是使用 Python 连接 Trino Coordinator, S3 或 Alluxio

进行测试。为了更加贴近真实场景，我们使用 Python 又进行测试

Python 连接 S3（以多表连接查询的 1 个线程的情况为例）

```
import trino
import boto3
import csv
import time

# Trino 连接细节
trino_host = '127.0.0.1'
trino_port = 8090
trino_user = 'zzh'
trino_catalog = 'hive'
trino_schema = 'aitop'

# S3 连接细节
s3_access_key = 'my_access_key'
s3_secret_key = 'my_secret_key'
s3_region = 'us-east-2'
s3_bucket_name = 'zhaozihao-us-east-2'
s3_bucket_path = 'alluxio/aitop-test07'

#  查询细节
queries = [
    {
        'table_suffix': str(i).zfill(2),
        'output_file_prefix': 'result_test07_' + str(i).zfill(2),
    } for i in range(20)
]

# Trino 连接
conn = trino.dbapi.connect(
    host=trino_host,
    port=trino_port,
    user=trino_user,
    catalog=trino_catalog,
```

```
    schema=trino_schema
)

# S3 连接
s3 = boto3.client('s3',
aws_access_key_id=s3_access_key,
aws_secret_access_key=s3_secret_key,
region_name=s3_region)

# 计时装饰器
def timer(func):
    def wrapper(*args, **kwargs):
        start_time = time.time_ns()
        result = func(*args, **kwargs)
        end_time = time.time_ns()
        execution_time = (end_time - start_time) // 1000000 # 将纳秒转换为毫秒
        print(f"zzhTime-{func.counter}: {execution_time}")
        func.counter += 1
        return result
    func.counter = 1
    return wrapper

# 查询并写入 CSV 文件的函数
@timer
def execute_query_and_write_to_csv(query_details):
    table_suffix = query_details['table_suffix']
    output_file_prefix = query_details['output_file_prefix']

# 构建查询语句
query = f"""
SELECT
csbar_1m_{table_suffix}.date_day AS "date_day",
csbar_1m_{table_suffix}.time_minute AS "time_minute",
csbar_1m_{table_suffix}.sym AS "sym",
csbar_1m_{table_suffix}.open AS "open",
csbar_1m_{table_suffix}.high AS "high",
```

```
csbar_1m_{table_suffix}.low AS "low",
csbar_1m_{table_suffix}.close AS "close",
csbar_1m_{table_suffix}.volume AS "volume",
csbar_1m_{table_suffix}.turnover AS "turnover",
csbar_1d_{table_suffix}.prevclose AS "prevclose",
csbar_1d_{table_suffix}.tradecount AS "tradecount",
csbar_1d_{table_suffix}.upperlimit AS "upperlimit",
csbar_1d_{table_suffix}.lowerlimit AS "lowerlimit",
csbar_1d_{table_suffix}.ttlshr AS "ttlshr",
csbar_1d_{table_suffix}.fltshr AS "fltshr",
csbar_1d_{table_suffix}.frshr AS "frshr",
csbar_1d_{table_suffix}.af AS "af"
FROM
csbar_1m_{table_suffix} LEFT JOIN csbar_1d_{table_suffix}
ON csbar_1m_{table_suffix}.date_day = csbar_1d_{table_suffix}.date_day
AND csbar_1m_{table_suffix}.sym = csbar_1d_{table_suffix}.sym
"""

# 执行查询
cur = conn.cursor()

# Execute the query 30 times
for i in range(30):
    # Execute query
    cur.execute(query)

# 提取结果
results = cur.fetchall()
file_name = f'{output_file_prefix}_{str(i).zfill(2)}'
output_file_path = f'files/{file_name}.csv'
with open(output_file_path, 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(results)

# Upload CSV file to S3
s3_file_path = f'{s3_bucket_path}/{file_name}.csv'
```

```
s3.upload_file(output_file_path, s3_bucket_name, s3_file_path)

# 从 S3 多次下载 CSV 文件
for j in range(50):
    local_file_path = f'files/{file_name}_{str(j).zfill(2)}.csv'
    s3.download_file(s3_bucket_name, s3_file_path, local_file_path)

# 执行查询并写入 CSV 文件
for query in queries:
    execute_query_and_write_to_csv(query)
```

Python 通过 Alluxio 连接 S3（以多表连接查询的 1 个线程的情况为例）

```
import trino
import alluxio
import csv
import time

# Trino 连接细节
trino_host = '127.0.0.1'
trino_port = 8090
trino_user = 'zzh'
trino_catalog = 'hive'
trino_schema = 'aitopalluxio'

# Alluxio 连接细节
alluxio_host = '127.0.0.1'
alluxio_port = 39999
alluxio_path = '/aitop-test09'

# 查询细节
queries = [
{
'table_suffix': str(i).zfill(2),
```

```
'output_file_prefix': 'result_test09_' + str(i).zfill(2),
}
for i in range(20)
]

# Trino 连接
conn = trino.dbapi.connect(
host=trino_host,
port=trino_port,
user=trino_user,
catalog=trino_catalog,
schema=trino_schema
)

# Alluxio 连接
alluxio_client = alluxio.Client(alluxio_host, alluxio_port)

# 计时装饰器
def timer(func):
def wrapper(*args, **kwargs):
start_time = time.time_ns()
result = func(*args, **kwargs)
end_time = time.time_ns()
execution_time = (end_time - start_time) // 1000000 # 将纳秒转换为毫秒
print(f"zzhTime-{func.counter}: {execution_time}")
func.counter += 1
return result
func.counter = 1
return wrapper

# 查询并写入 CSV 文件的函数
@timer
def execute_query_and_write_to_csv(query_details):
table_suffix = query_details['table_suffix']
output_file_prefix = query_details['output_file_prefix']
```

```
# 构建查询语句
query = f"""
SELECT
csbar_1m_{table_suffix}.date_day AS "date_day",
csbar_1m_{table_suffix}.time_minute AS "time_minute",
csbar_1m_{table_suffix}.sym AS "sym",
csbar_1m_{table_suffix}.open AS "open",
csbar_1m_{table_suffix}.high AS "high",
csbar_1m_{table_suffix}.low AS "low",
csbar_1m_{table_suffix}.close AS "close",
csbar_1m_{table_suffix}.volume AS "volume",
csbar_1m_{table_suffix}.turnover AS "turnover",
csbar_1d_{table_suffix}.prevclose AS "prevclose",
csbar_1d_{table_suffix}.tradecount AS "tradecount",
csbar_1d_{table_suffix}.upperlimit AS "upperlimit",
csbar_1d_{table_suffix}.lowerlimit AS "lowerlimit",
csbar_1d_{table_suffix}.ttlshr AS "ttlshr",
csbar_1d_{table_suffix}.fltshr AS "fltshr",
csbar_1d_{table_suffix}.frshr AS "frshr",
csbar_1d_{table_suffix}.af AS "af"
FROM
csbar_1m_{table_suffix} LEFT JOIN csbar_1d_{table_suffix}
ON csbar_1m_{table_suffix}.date_day = csbar_1d_{table_suffix}.date_day
AND csbar_1m_{table_suffix}.sym = csbar_1d_{table_suffix}.sym
"""

# 执行查询
cur = conn.cursor()

# Execute the query 30 times
for i in range(30):
# Execute query
cur.execute(query)

# 提取结果
results = cur.fetchall()
```

```
file_name = f'{output_file_prefix}_{str(i).zfill(2)}'
output_file_path = f'files/{file_name}.csv'
with open(output_file_path, 'w', newline='') as f:
writer = csv.writer(f)
writer.writerows(results)

# 读取文件
file_content = ''
with open(output_file_path, 'r') as f:
file_content = f.read()

# Upload CSV file to Alluxio
alluxio_file_path = f'{alluxio_path}/{file_name}.csv'

with alluxio_client.open(alluxio_file_path, 'w') as f:
f.write(file_content)

# 从 Alluxio 多次下载 CSV 文件
for j in range(50):
local_file_path = f'files/{file_name}_{str(j).zfill(2)}.csv'
with alluxio_client.open(alluxio_file_path, 'r') as f:
with open(local_file_path, 'w') as this_file:
this_file.write(f.read().decode('utf-8'))

# 执行查询并写入 CSV 文件
for query in queries:
execute_query_and_write_to_csv(query)
```

Python 通过 Alluxio Fuse 连接 S3（以多表连接查询的 1 个线程的情况为例）

```
import trino
import alluxio
import csv
import time

# Trino 连接细节
```

```
trino_host = '127.0.0.1'
trino_port = 8090
trino_user = 'zzh'
trino_catalog = 'hive'
trino_schema = 'aitopalluxio'

# 查询细节
queries = [
{
'table_suffix': str(i).zfill(2),
'output_file_prefix': 'result_test11_' + str(i).zfill(2),
}
for i in range(20)
]

# Trino 连接
conn = trino.dbapi.connect(
host=trino_host,
port=trino_port,
user=trino_user,
catalog=trino_catalog,
schema=trino_schema
)


# 计时装饰器
def timer(func):
def wrapper(*args, **kwargs):
start_time = time.time_ns()
result = func(*args, **kwargs)
end_time = time.time_ns()
execution_time = (end_time - start_time) // 1000000 # 将纳秒转换为毫秒
print(f"zzhTime-{func.counter}: {execution_time}")
func.counter += 1
return result
func.counter = 1
```

```
return wrapper

# 查询并写入 CSV 文件的函数
@timer
def execute_query_and_write_to_csv(query_details):
table_suffix = query_details['table_suffix']
output_file_prefix = query_details['output_file_prefix']

# 构建查询语句
query = f"""
SELECT
csbar_1m_{table_suffix}.date_day AS "date_day",
csbar_1m_{table_suffix}.time_minute AS "time_minute",
csbar_1m_{table_suffix}.sym AS "sym",
csbar_1m_{table_suffix}.open AS "open",
csbar_1m_{table_suffix}.high AS "high",
csbar_1m_{table_suffix}.low AS "low",
csbar_1m_{table_suffix}.close AS "close",
csbar_1m_{table_suffix}.volume AS "volume",
csbar_1m_{table_suffix}.turnover AS "turnover",
csbar_1d_{table_suffix}.prevclose AS "prevclose",
csbar_1d_{table_suffix}.tradecount AS "tradecount",
csbar_1d_{table_suffix}.upperlimit AS "upperlimit",
csbar_1d_{table_suffix}.lowerlimit AS "lowerlimit",
csbar_1d_{table_suffix}.ttlshr AS "ttlshr",
csbar_1d_{table_suffix}.fltshr AS "fltshr",
csbar_1d_{table_suffix}.frshr AS "frshr",
csbar_1d_{table_suffix}.af AS "af"
FROM
csbar_1m_{table_suffix} LEFT JOIN csbar_1d_{table_suffix}
ON csbar_1m_{table_suffix}.date_day = csbar_1d_{table_suffix}.date_day
AND csbar_1m_{table_suffix}.sym = csbar_1d_{table_suffix}.sym
"""

# 执行查询
cur = conn.cursor()
```

```
# Execute the query 30 times
for i in range(30):
# Execute query
cur.execute(query)

# 提取结果并写入 alluxio fuse
results = cur.fetchall()
file_name = f'{output_file_prefix}_{str(i).zfill(2)}'
output_file_path = f'/mnt/fuse/aitop-test11/{file_name}.csv'
with open(output_file_path, 'w', newline='') as f:
writer = csv.writer(f)
writer.writerows(results)

# 读取文件
file_content = ''
with open(output_file_path, 'r') as f:
file_content = f.read()

# 从 Alluxio 多次下载 CSV 文件
for j in range(50):
local_file_path = f'files/{file_name}_{str(j).zfill(2)}.csv'
with open(local_file_path, 'w') as this_file:
this_file.write(file_content)

# 执行查询并写入 CSV 文件
for query in queries:
execute_query_and_write_to_csv(query)
```

4.3.4 结果与可视化

- 我们针对每一类查询，制做了 20 个数据表（q01, q02, q03, …, q20），每个表的大小都相同
- 针对每个表分别使用 1 个线程、2 个线程去查询数据（作为两组测试），进行三组对照

测试：

- 使用 Trino 直接查询 S3
- 使用 Trino 通过 Alluxio 查询 S3
- 不使用 Trino，使用 Alluxio 工具（Alluxio Fuse）直接从 Alluxio 下载数据

测试结果图表的横轴 q01，q02，… 表示查询第 1 个数据表、第二个数据表...
测试结果图标的纵轴表示每个测试 sql 的耗时，单位是毫秒。

查询类型	并发线程	Trino+Alluxio 平均提升量	Alluxio FUSE 平均提升量	Trino+S3 平均运行时间	Trino+Alluxio 平均运行时间	Alluxio FUSE 平均运行时间
多表连接查询	1	25.25%	76.90%	279s	208s	64s
	2	23.55%	79.39%	312s	239s	64s
聚合查询	1	45.71%	78.36%	128s	70s	28s
	2	37.39%	77.85%	138s	86s	31s

结论：

- 单个查询的时间在从单线程变为双线程时间基本保持一致，因此总查询时间双线程仅为单线程的一半。

为了更加贴近真实场景，我们不通过 Trino Client 连接 Trino Coordinator，而是直接通过 Python 连接 Trino Coordinator。将查询的结果以 CSV 的形式存入 S3、Alluxio 或 Alluxio Fuse 中。

查询类型	并发线程	Python+Trino+Alluxio	Python+Trino+Alluxio Fuse	Python+Trino+S3 平均时间	Python+Trino+Alluxio 平均时间	Python+Trino+Alluxio Fuse 平均运行时间
多表连接查询	1	55.89%	65.25%	132s	58s	46s
	2	54.68%	66.86%	132s	60s	44s
聚合查询	1	85.41%	92.83%	150s	22s	11s
	2	83.64%	93.68%	149s	24s	9s

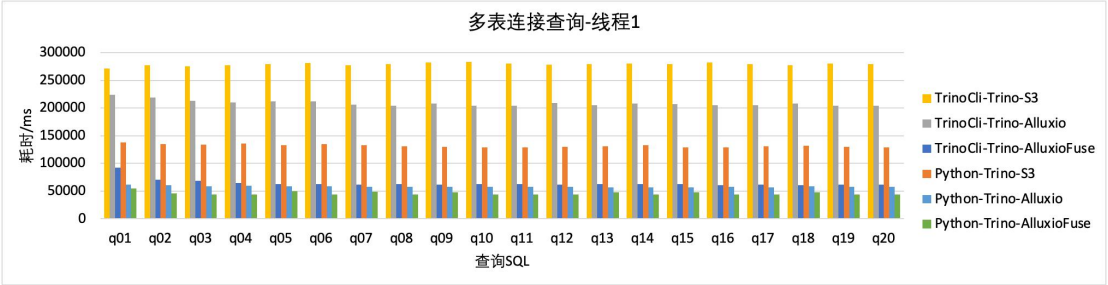

图 4.3.3 多表连接查询-线程 1-性能对比图

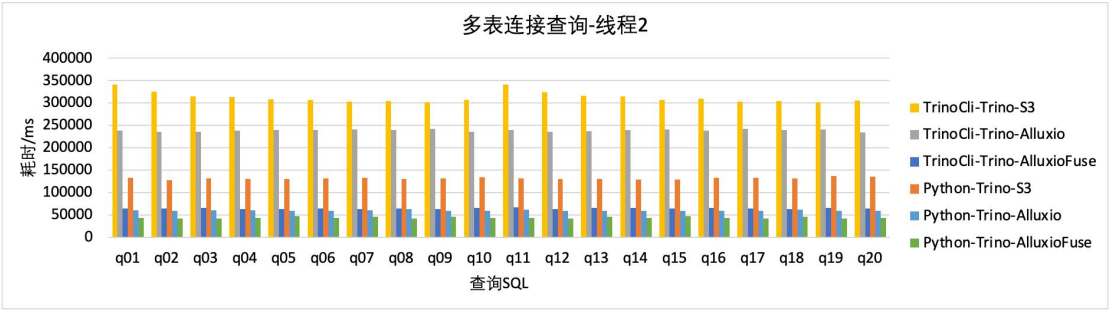

图 4.3.4 多表连接查询-线程 2-性能对比图

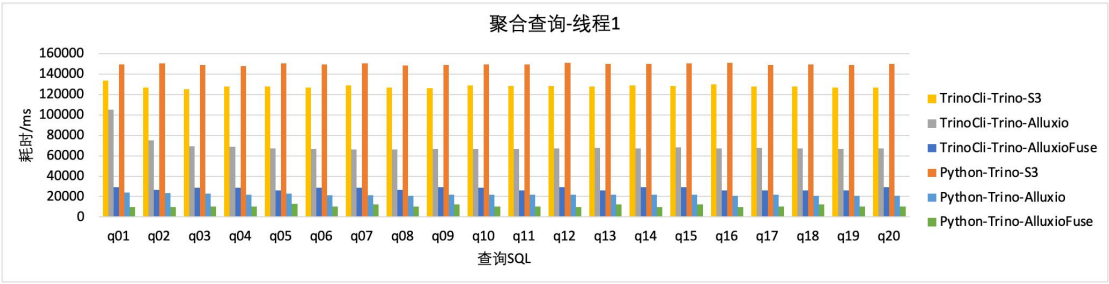

图 4.3.5 聚合查询-线程 1-性能对比图

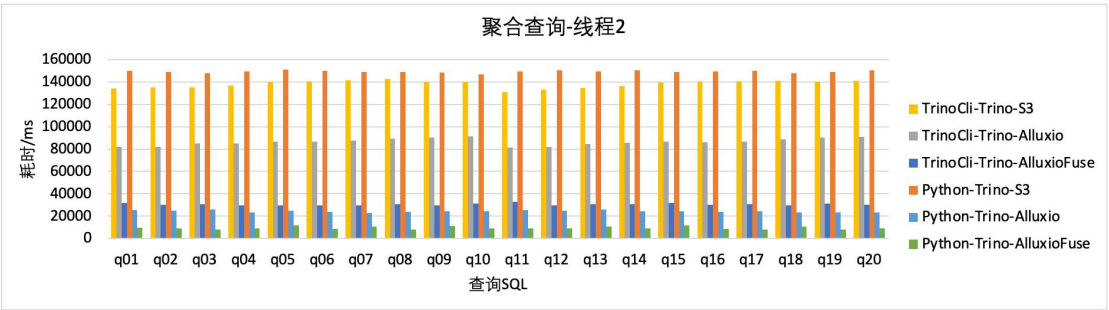

图 4.3.6 聚合查询-线程 2-性能对比图

可见 Alluxio 在提升整体系统性能的方面还是很有优势的。

五、总结与展望

我们成功搭建了两套数据系统，一套使用 Alluxio 作为数据编排层，另一套则不使用 Alluxio。经过横向对比实验，我们证实了使用 Alluxio 能够有效提升数据系统的处理性能。在基准测试中，我们发现使用 Alluxio 作为数据编排层的系统明显提升了数据系统的处理性能。Alluxio 作为一个中间数据层，能够切实提供高速缓存和数据管理的功能，在普遍的数据处理情景下，有效地减少了数据访问延迟，并提升了查询性能和系统吞吐量。

在具体场景测试中，我们以量化交易场景为例进行了测试。实验发现使用 Alluxio 作为数据编排系统，能够显著提升查询效率和性能。特别是在多表连接查询和聚合查询方面，通过使用 Alluxio FUSE 直接从 Alluxio 下载数据，平均提升量达到了较高的百分比，且平均运行时间明显减少。这表明 Alluxio 在量化交易场景下具有明显的优势，能够为实际应用场景提供高效的数据处理和查询支持。

总而言之，通过本项目的研究和分析，我们对 Alluxio 在数据编排和场景应用方面的优势和适用性有了更深入的了解。我们相信 Alluxio 将在数据系统的设计和优化中发挥重要作用，并为用户提供更高效、可靠和灵活的数据管理解决方案。

在未来，我们希望对 Alluxio 在更多不同场景下的应用进行进一步研究和探索。我们可以考虑扩展测试范围，包括大规模数据分析、机器学习任务等，以评估 Alluxio 在这些场景下的优势和适用性。此外，我们也可以结合其他数据管理和处理工具，对 Alluxio 进行深度集成，进一步提升系统的性能和功能。

同时，我们还将继续关注 Alluxio 的发展和新特性的推出。Alluxio 作为一个开源项目，持续不断地进行改进和优化，我们将密切关注其最新的版本和功能，并将其应用到我们的实际项目中。我们期待 Alluxio 在未来能够更好地满足数据系统的需求，并成为数据管理和处理领域的重要工具和组件之一。同时，我们也将继续关注和推动数据系统和数据管理领域的发展和创新，以更好地满足用户的需求和挑战。