

分类号	<u>TP399</u>	密级	<u>公开</u>
UDC	<u>004</u>	学位论文编号	<u>D-10617-308-(2019)-02047</u>

重庆邮电大学硕士学位论文

中文题目	<u>Spark SQL 结构化数据处理</u>
	<u>及性能优化</u>
英文题目	<u>Structured Data Processing and</u>
	<u>Performance Optimization</u>
	<u>of Spark SQL</u>
学 号	<u>S160201048</u>
姓 名	<u>罗昭</u>
学位类别	<u>工学硕士</u>
学科专业	<u>计算机科学与技术</u>
指导教师	<u>程克非 教授</u>
完成日期	<u>2019 年 5 月 30 日</u>

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的
研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含他人
已经发表或撰写过的研究成果，也不包含为获得 重庆邮电大学 或其他单位的
学位或证书而使用过的材料。与我一同工作的人员对本文研究做出的贡献均已在
论文中作了明确的说明并致以谢意。

作者签名：



日期：2019 年 5 月 31 日

学位论文版权使用授权书

本人完全了解 重庆邮电大学 有权保留、使用学位论文纸质版和电子版
的规定，即学校有权向国家有关部门或机构送交论文，允许论文被查阅和借阅等。
本人授权 重庆邮电大学 可以公布本学位论文的全部或部分内容，可编入有
关数据库或信息系统进行检索、分析或评价，可以采用影印、缩印、扫描或拷贝
等复制手段保存、汇编本学位论文。

(注：保密的学位论文在解密后适用本授权书。)

作者签名：



日期：2019 年 5 月 31 日

导师签名：



日期：2019 年 6 月 1 日

摘要

近年来 Spark 内存计算框架快速崛起, 数据处理速度得到极大的提高, 但是其速度上限却受限于 Spark 内存规模。当数据量小于或接近内存容量时 Spark 性能最好, 反之则性能较差。因此 Spark SQL 在处理以 4G 行业卡数据为代表的通信大数据时暴露出了诸多问题, 如读写速度和查询速度缓慢、系统资源分配不均或不足、大表 Join 效率低等。

本文从 Spark SQL 的数据组织方式、Spark 资源管理机制和 Join 算法三个方面处理结构化数据并进行相关的性能优化。首先提出了改进的数据组织框架以提高 Spark SQL 的读写和查询速度, 其次建立了资源监控模型合理的分配和使用资源, 最后基于改进的数据组织框架和监控模型改进了大表 Join 算法。主要工作如下:

(1) 本文通过分析和对比 Spark SQL 和 Hbase 的数据组织方式, 提出了一种改进的数据组织框架。该框架首先改进了 Parquet 文件格式的读写接口, 其次利用 Hbase+Phoenix 构建了二级索引, 大幅提升了 4G 行业卡数据的读写和查询速度。

(2) 本文进一步研究了 Spark 的内存模型和资源使用情况, 通过性能监控获取集群底层各项参数, 建立了内存监控模型对资源使用情况进行分级和预警。最后将预警结果通过观察者模型反馈给订阅者, 订阅模块就可以根据它的反馈来动态调整数据流量。

(3) 本文基于改进的数据组织框架和监控模型对大表关联算法进行优化, 提出了一种基于内存监控和分批处理的 Join 算法。该算法通过监控模型动态地控制数据流量和 Join 批次, 并通过改进的数据组织方式加快数据读写和查询速度。实验表明该算法从一定程度上缓解了内存不足的问题, 也降低了数据倾斜导致的负载不均衡影响, 总体运行时间优于默认的 Join 算法。

综上所述, Spark SQL 处理结构化数据的性能受数据组织方式和内存模型的影响, 具体表现为 Join 效率低下。本文首先改进了数据组织框架, 然后建立了内存监控模型, 最后优化了 Join 算法, 平均处理时间缩短了 31.49%。

关键词: Spark SQL, 结构化数据, Parquet, 内存监控, Join

Abstract

In recent years, the Spark memory computing framework has risen rapidly, and the data processing speed has been greatly improved. However, the upper limit of speed is limited by the Spark memory size. When the amount of data is less than or close to memory capacity, Spark has the best performance. Otherwise, it has poor performance. Therefore, Spark SQL exposes many problems when dealing with communication big data represented by 4G industry card data, such as slow read-write speed and query speed, uneven or insufficient system resource allocation, and low efficiency of large table Join.

This thesis deals with structured data and performs related performance optimization from three aspects: Spark SQL data organization, Spark resource management mechanism and Join algorithm. Firstly, an improved data organization framework is proposed to improve the read-write and query speed of Spark SQL. Secondly, the resource monitoring model is established to allocate and use the resources reasonably. Finally, the large table Join algorithm is proposed based on the improved data organization framework and monitoring model. The main work is as follows:

(1) By analyzing and comparing the data organization methods of Spark SQL and Hbase, an improved data organization framework is proposed in this thesis. The framework first improves the read-write interface of the Parquet file format, and then constructs the secondary index using Hbase+Phoenix, which greatly improves the speed of reading and writing and querying the 4G card data.

(2) This thesis further studies the memory model and resource usage of Spark, obtains the underlying parameters of the cluster through performance monitoring, and establishes a memory monitoring model to classify and warn the resource usage. Finally, the warning results are fed back to the subscriber through the observer model, and the subscription module can adjust the data traffic dynamically according to its feedback.

(3) This thesis optimizes the large table association algorithm based on the improved data organization framework and monitoring model, and proposes a Join algorithm based on memory monitoring and batch processing. The algorithm dynamically controls the data flow and Join batches through the monitoring model, and accelerates the speed of read-write and query through improved data organization. Experiments show that the algorithm alleviates the problem of insufficient memory to some extent and reduces the

load imbalance caused by data skew. The overall running time is better than that of the default Join algorithm.

In summary, the performance of Spark SQL processing structured data is affected by data organization and memory model, which is characterized by low efficiency of Join. In this thesis, the data organization framework is firstly improved, and then the memory monitoring model is established. Finally, the Join operation is optimized and the average processing time is reduced by 31.49%.

Keywords: Spark SQL, structured data, Parquet, memory monitoring, Join

目录

图录	VII
表录	IX
第 1 章 绪论	1
1.1 研究背景及意义	1
1.2 国内外研究现状	2
1.2.1 数据组织方式研究现状	2
1.2.2 Join 算法研究现状	3
1.2.3 Spark 资源优化研究现状	4
1.3 论文主要工作	5
1.4 论文组织结构	6
第 2 章 Spark SQL 及 Hbase 技术基础	8
2.1 Spark SQL 数据组织框架	8
2.1.1 Spark SQL 查询机制	8
2.1.2 Spark SQL 文件格式	9
2.1.3 Parquet 文件格式	10
2.2 Hbase 数据组织框架	11
2.2.1 Hbase 查询机制	11
2.2.2 Hbase 文件格式	13
2.2.3 Phoenix 映射关系	14
2.3 Spark 资源管理机制	16
2.3.1 Spark 运行机制	16
2.3.2 Spark 内存模型	18
2.4 Spark Join 算法	21
2.4.1 分布式 Join 算法分类	21

2.4.2 BloomFilter 算法.....	22
2.5 本章小结	22
第 3 章 Spark SQL 数据组织方式设计	24
3.1 Spark SQL 问题分析	24
3.1.1 数据读写问题	24
3.1.2 数据存储问题	24
3.1.3 存储格式对比分析	25
3.1.4 Spark SQL 与 Hbase 整合分析	27
3.2 4G 行业卡数据组织框架设计	28
3.2.1 业务场景分析	28
3.2.2 Spark SQL 读写接口改进	31
3.2.3 Spark SQL 与 Hbase 框架整合	34
3.3 实验及结果分析	36
3.3.1 实验环境	36
3.3.2 实验结果分析	36
3.4 本章小结	41
第 4 章 大表关联算法研究	42
4.1 Spark SQL 大表关联问题分析	42
4.1.1 Sort Merge Join 问题分析	42
4.1.2 分批 Join 策略	43
4.2 内存监控模型设计	45
4.2.1 性能指标分析	45
4.2.2 内存监控模型	48
4.2.3 资源分级及预警	49
4.3 分批 Join 算法设计	50
4.3.1 算法概述	50
5.3.2 算法详细流程	51

5.3.3 算法开销分析	53
4.4 实验及结果分析	54
4.4.1 实验环境	54
4.4.2 实验结果分析	54
4.5 本章小结	61
第 5 章 工作总结和展望	62
5.1 论文工作总结	62
5.2 工作展望	63
参考文献	64
致谢	69
攻读硕士学位期间从事的科研工作及取得的成果	70

图录

图 2.1 Catalyst 主要组件及执行流程	9
图 2.2 HBase 不同压缩格式下文件大小对比.....	13
图 2.3 HBase 不同压缩格式下查询时间对比.....	14
图 2.4 tb_imsi_tel 在 Phoenix 中的数据视图	15
图 2.5 先创建 tb_imsi_tel 的结构	15
图 2.6 后创建 tb_imsi_tel 的结构	15
图 2.7 Spark 集群工作框架	17
图 2.8 Spark 作业与集群资源对应关系图	18
图 2.9 JVM 内存模型	18
图 2.10 Spark 内存模型	19
图 3.1 不同存储格式的文件大小对比	26
图 3.2 不同存储格式的查询对比	27
图 3.3 Spark SQL 与 Hbase 的查询对比	28
图 3.4 LTE 网络架构图	29
图 3.5 反射创建 MessageType schema	33
图 3.6 反射获取对象值	33
图 3.7 改进接口与默认 API 读写对比	34
图 3.8 改进的数据组织框架	35
图 3.9 部分实验数据展示	38
图 3.10 FileInfo 数据展示	38
图 3.11 HB_S6A 数据展示.....	39
图 3.12 整合的数据组织框架对比实验	40
图 3.13 横向对比查询实验	41
图 4.1 Spark 作业划分关系图	47

图 4.2 算法框架图	51
图 4.3 监控模型运行截图	57
图 4.4 IM 表数据	59
图 4.5 S6a 表数据	60
图 4.6 两种 Join 算法时间对比图.....	60
图 4.7 综合指标对比图	61

表录

表 2.1 tb_imsi_tel 表的字段定义 10

表 2.2 Hbase 表的逻辑视图 12

表 3.1 四种不同类型的 SQL 测试语句..... 26

表 3.2 S6a 信令表的结构 37

表 3.3 横向对比实验配置信息 40

表 4.1 environment 指标信息 45

表 4.2 executors 指标信息 46

表 4.3 jobs 指标信息..... 47

表 4.4 stages 指标信息..... 48

表 4.5 实验环境参数表 54

表 4.6 接口字段设计说明 55

表 4.7 经验系数影响实验 58

表 4.8 IM 信令表的结构 58

第 1 章 绪论

1.1 研究背景及意义

随着近几年各行各业的数据爆炸，我们已经进入大数据时代，数据成为一种宝贵的信息财富。但是庞大的数据规模和复杂的数据关系对数据分析处理技术提出了严峻的考验，传统的数据处理技术已经不能满足现在的需求。比如 MySQL、Oracle 等传统的关系型数据库已经不能存储和计算海量的关系数据，且面对非结构化和半结构化的数据更是束手无策。于是基于关系型数据库的结构化数据业务纷纷转移到以 Hadoop^[1-2]、Hbase^[3]、Spark^[4]为代表的大数据平台。尤其是随着移动互联网的普及以及终端设备数量的剧增，运营商们迫切地需要分析移动通信大数据来适应用户不断变化的需求。正是基于这样的背景，本文在移动运营商的 4G 行业应用卡(以下简称 4G 行业卡)故障诊断项目和 XDR 信令数据核查项目中应用了 Hadoop+Spark 技术，并且取得了良好的效果。

这两个项目需要 7*24 小时稳定运行，每天处理的数据规模为几百 GB 到几 TB 不等，按行计算则多达数十亿条记录。这些数据都是标准的结构化数据，最开始存储于关系型数据库中，表之间的相互关系通过特定的字段关联起来。但是由于每天处理的数量规模太大，且需要全表扫描、逐行计算以及大量的多维度统计和多表关联等复杂操作，关系型数据库已经无法满足其需求。于是新的方案先采用 HDFS^[5] 存储原始文件，再使用 Spark 采集和筛选数据并存入数据仓库，最后使用 Spark SQL^[6] 处理一些复杂的关联逻辑。其中最为重要的是，Spark SQL 能够使数据的逻辑结构和处理流程与原来的关系型数据库保持一致，开发人员无需做过多的改变就能完成二次开发。当然在项目中也暴露出一些性能问题，如文件组织方式不合理、资源分配不合理导致内存不足、Join 效率低等。这正是本课题提出的契机，期望通过本次研究工作加以解决。

在关系型数据库(如 MySQL)项目中，结构化数据的业务需求灵活多变。通常先使用主键、外键等规则来组织和约束数据，再通过 SQL 语句进行分析处理，如基本的 CURD 操作(创建 Create、更新 Update、读取 Retrieve、删除 Delete)，以及复杂的关联操作(Union、Join、Group by 等)。虽然 Hbase、Spark SQL 等开源框架支

持这些操作，但由于其分布式实现的特殊性，所以对 OLAP^[7]和 OLTP^[8]的支持不如关系型数据库。正是基于这样的背景，本文深入研究了 Spark SQL 处理结构化数据的机制，并通过性能优化以达到快速处理结构化数据的目的。

Spark SQL 是 Spark 框架中专门处理结构化数据的模块，使用它高效地存储和处理这些数据将对我们的生活产生十分重要的意义。比如从海量数据中挖掘有用信息、从物联网数据中做故障诊断、以及对日常行为的统计或预测等工作都是十分必要的。数据之间都是有关联关系的，通常中我们将这些关系进行抽象和建模从而形成二维表，然后使用不同的规则关联起来形成关系型数据库。但是关系型数据库处理不了大数据^[9-10]，于是当大数据开源平台出现的时候，传统项目迁移到大数据平台已经成为一种不可逆的趋势，所以研究在大数据平台上处理结构化数据的重要性不言而喻。

当数据规模较大或实时性要求较高时，使用 Hadoop、Hbase、Spark 等技术来处理它们会极大地提高工作效率。使用数据仓库、Hbase、HDFS 等替代关系型数据库来持久化数据可以有效避免关系型数据库的存储容量限制；使用 Spark 的分布式内存计算替代传统的单机计算可以有效地提高计算速度。但是 Spark 也有一些局限性，如数据规模受限于内存大小。当数据量小于或接近内存容量时 Spark 性能最好^[11]，反之则性能较差，因此 Spark 容易出现内存不足等异常。此外数据分析中最常用的 Join 算法(等值连接，下称 Join)在 Spark 上也表现较差，特别耗时。因此本文基于 4G 行业卡数据深入研究了 Spark 的机制，并针对出现的问题做了性能优化，这对于提升 Spark SQL 处理结构化的性能具有重要的意义，也为物联网数据、5G 行业卡数据的处理提供了参考。

1.2 国内外研究现状

1.2.1 数据组织方式研究现状

Spark SQL 的查询性能受数据组织方式的影响较大，数据组织方式涉及存储形式、压缩算法以及索引结构等方面。为了改善数据组织方式对于查询性能的影响，Spark 社区提供了列式压缩存储格式如：Parquet 文件格式，它可以过滤不需要的字段从而提高查询效率^[6]，但是 Parquet 的读写速度严重受限于 Spark API。由于 Spark

SQL 中的表没有主键和索引，因而 Spark SQL 提出了一种粗粒度的索引策略，即文件分区和分桶策略。但是这种分区需要编程人员手动的去创建和指定，于是 Guo Chenghao 等人在文献[12]中提出了一种自适应的数据分区方案来提高 Spark SQL 的查询效率，但是依然没有为表建立索引。也有研究者尝试在表上建立细粒度的索引，如周亮、李格非等人在文献[13]中针对时态大数据建立时态索引并扩展了 Spark SQL 解析器，使得时态查询性能优于原生 Spark SQL 性能。此外，王亚玲、刘越等人在文献[14]中设计出一套针对电力行业大数据的 OLAP 方案，在该系统中建立了一种基于前缀树的细粒度索引结构 TrieIndex。

由于基于 Spark SQL 内建的数据仓库对于数据的更新和删除支持不如物理数据库，因为数据仓库只是逻辑上的。Hbase 是基于列的物理数据库^[15]，能很好的支持 CURD 操作，如果利用 Hbase 的实时查询机制为 Spark SQL 文件建立索引，将会加快 Spark SQL 的检索速度。所以可将 Hbase 与 Spark SQL 结合，即通过改变 Spark SQL 数据源的组织方式来提高查询效率。但是针对 4G 行业卡数据，目前将 Hbase 与 Spark SQL 结合的相关研究较少，很多研究仅仅是基于 Hbase 本身的优化。Hbase 只有基于 Rowkey 的单独索引，当面对多维度的复杂关联时效率很低，所以王文贤、陈兴蜀等研究者在文献[16]中提出了一种基于 Solr 的 Hbase 海量数据二级索引方案。此外朱明和王志瑞等人在文献[17]中也对 Hbase 建立了二级索引，所以针对结构化数据可通过建立二级索引来提高查询效率。Hbase 只提供了底层的接口，必须借助 Phoenix^[18]才能使用 SQL 语句，因此袁兆争等人在文献[19]中重新构建了解析 SQL 语言的编译器将 SQL 语句转换为对应的 Hbase 操作，但是 Spark SQL 中本身就有基于 Catalyst 的 SQL 解析器^[6]。

综上所述，通过数据组织方式提高 Spark SQL 的查询性能需要考虑存储格式和索引结构，虽然目前有一些高效的存储格式，但是其读写性能受限于 Spark API，因此需要重点关注读写方式的改进。由于目前对 Hbase 二级索引的研究相对成熟，所以利用 Hbase 为 Spark SQL 建立二级索引可以在一定程度上提高数据的检索速度。

1.2.2 Join 算法研究现状

Spark SQL 解析器在执行过程中对于复杂关联查询的优化力度并不够，当数据量过大时会造成程序阻塞或者崩溃，因为并行的多表关联比关系型数据库中的单机

关联要复杂许多。刘容辰等人为了提高 Join 关联效率，在文献[20]中提出了一种基于 BloomFilter 的过滤再分区算法，主要过滤掉不符合条件的无效连接。孙文隽和李建中在文献[21]中对 Join 算法进行了改进，提出一种基于单关系外排序的分治 Join 算法，其效率要高于排序合并 Join 算法。郑晓薇和马琳在文献[22]中提出了一个并行关联多个大表的简便算法 MR_Join，该算法可以有效地实现大表关联。邓亚丹、景宁等人在文献[23]中采用了 Radix Join 算法，即通过 p 趟划分将表重分区为

$$H = \prod_1^p H_p \text{ 个分区, 每次划分中都是从 Hash 值的二进制位最左端开始取 } D = \sum_1^p D_p \text{ 位}$$

将一个块分裂成 $H_p = 2^{D_p}$ 个块，Radix 分区之后再进行 hash 连接，该算法区间划分均匀，具有良好的并行度，但是 Radix 划分的趟数不好确定，有可能导致数据分布的失衡而造成数据倾斜。钱招明、王雷等人在文献[24]中基于 Semi-Join 算法作了优化，提出了 Semi-get-Join 和 Semi-range-Join 来改善获取右表数据的效率，但是只适用于右表数据大且左表少的情况，有较大的局限性。

很多研究都是从纯算法的角度来优化 Join，忽略了数据组织方式对于查询性能的影响。例如高泽等人在文献[25]中参照 MapReduce 模型实现了一种 UGS (Union Group and Segmentation) 算法来提升 Spark Join 的效率，但该算法是基于文本格式的，如果采用 Parquet 格式或者 ORC 格式效果可能会更好。因为 K. Rattanaopas 等人通过文献[26]以及 Li Xiaopeng 等人通过文献[27]都证明了 Parquet 文件在 Spark 上表现最好，ORC 文件在 Hive 上表现最好，且两者都要优于文本文件。因此文献[28]中王华进、黎建辉等研究者提出了基于 ORC 文件的 Join Key 分布估计方法：它通过 Join 列的 ORC 元数据来估算 Join Key 的分布，提升了估算效率，并构造了负载均衡的重分区。

综上所述，目前的 Join 优化研究主要集中于算法角度，而忽略了数据组织方式的影响，所以应该结合数据组织方式的相关研究，即在一种高效的数据组织方案下再进行 Join 算法的改进便能显著地提升计算速度。

1.2.3 Spark 资源优化研究现状

对关联算法的改进只能从某一方面上提高效率，因为多表关联中 Shuffle 过程是不可避免的，这才是真正的性能瓶颈，所以有更多的研究者重点关注 Spark 计算框

架的研究和改进。熊安萍、夏玉冲等人在文献[29]中提出了一种 Shuffle 优化机制，从资源调度的角度来设计了基于 task 本地性等级的重启策略来提升 Shuffle 效率；而侯伟凡等人在文献[30]中则从 Shuffle 内存分配角度出发，根据 task 的需求分配内存降低了 task 的内存溢出率，提升了 Shuffle 的稳定性。因为 Shuffle 会产生很多中间结果，所以 Tang Zhuo 等人在文献[31]中提出了一种偏斜中间数据块的分裂合并算法，该算法仅仅是改善 Reduce 任务的负载均衡。虽然 Shuffle 是基于内存计算的，但是 Spark SQL 查询处理时访问内存的速度比底层组件的速度慢了一个数量级^[32]。于是 P. S. Rao 等人在文献[32]中建议对 Spark SQL 进行修改以提高其支持内存分解的能力，但是这种方式太过繁琐。因为 Spark SQL 引擎最终是转换成 RDD^[33-35]去执行，那么 Huang Chaoqiang 等人在文献[36]中提出了 RDDShare 系统来管理缓存和重用结果，该 RDDShare 系统可以显著地提升 Spark SQL 查询性能。陈康等人在文献[37]中对 RDD 缓存策略进行了优化，提高了任务在资源有限的情况下的稳定性，保证了任务执行效率。孟红涛等人在文献[38]中提出一种新的 RDD 分布式权值缓存策略，根据权值淘汰或缓存 RDD，但是需要通过集群内网获取权值，因此这方式的性能受限于网络状况。

由于 Spark 提供了很多参数，研究者 A. Gounaris 等人在文献[39]中提出通过参数调优如：通过 Shuffle 参数调优提高速度，通过内存参数调优避免内存溢出(OOM)^[40]。如果对 Spark 系统不够熟悉时，众多的参数调优存在很大的困难，于是陈侨安等人在文献[41]中提出了基于历史库的参数优化模型，该历史库存储了以往运行过的任务特征信息和配置信息，这样就减少了人为调优的难度。当然对于众多的 Spark 参数还可以通过 JMX API 和 RESTful API 来监控^[42]，利用监控数据可以查看作业运行状态、GC 信息、系统瓶颈等，有助于开发人员了解底层情况和简化调优工作^[43-44]。

综上所述，Spark 的资源优化研究涉及更深的系统底层知识，主要采用了参数调优、Shuffle 优化、负载均衡等策略。但 Spark 主要受限于内存资源，OOM 等异常还是不可避免。所以基于目前的研究现状，通过资源监控来配置资源是一种有效的解决办法。

1.3 论文主要工作

本文的主要工作为以下三个方面：

(1) 数据仓库和列式数据库的数据组织方式研究

将 Spark SQL 和 Hbase+Phoenix 充分结合起来存储 4G 行业卡数据，提出了一种改进的数据组织框架。该框架基于 Parquet 文件格式进行了读写接口的改进以提升读写速度；并利用 Hbase+Phoenix 构建二级索引以提升 Spark SQL 的检索速度。

(2) 资源监控方案研究

在改进的数据组织方式基础之上，将大量高频使用的性能指标训练成资源监控模型。该模型通过 RESTful API 动态地抓取 Runtime 指标，然后将多项指标量化后作为性能矩阵输入监控模型，运算后输出代价向量，并根据代价向量进行资源评级和预警，最后将预警结果通过观察者模型反馈给订阅者，订阅模块就可以根据它的反馈来动态调整数据流量。

(3) 大表关联算法研究

结合改进的数据组织框架和资源监控模型对大表关联算法进行优化，提出了一种基于内存监控和分批处理的 Join 算法，该算法考虑了数据文件、内存模型、分批策略和运行状态监控四个方面。首先对数据采样和划分 Join 批次；接着启用监控模块动态控制数据流量；然后使用 BloomFilter 过滤非 Join key，生成 HashMap 并广播；最后循环进行分批 Join 操作，每轮 Join 都会根据 HashMap 调入对应的文件进行局部 Hash Join。

1.4 论文组织结构

本文总共有 5 个章节，每个章节的主要内容如下：

第 1 章：绪论。本章首先介绍了论文研究的背景和意义，然后介绍了国内外相关的研究现状，最后介绍了本文的主要工作内容。

第 2 章：Spark SQL 及 Hbase 技术基础。本章主要介绍了 Spark SQL 和 Hbase 相关的技术基础，其中包括了它们各自的数据组织方式、查询机制、内存模型等，最后介绍了分布式 Join 算法和 BloomFilter 算法。

第 3 章：Spark SQL 数据组织方式。本章详细展开了 Spark SQL 对于结构化数据的组织方式，分析了其中的数据读写问题、存储问题，并改进了 Parquet 文件默认的读写接口，大幅提升了读写速度。对比了 Hbase 和 Phoenix 组织方式的优点，并将其整合到 Spark SQL 为其建立二级索引。

第 4 章：大表关联算法研究。本章主要针对 Spark SQL 的大表关联算法即 Sort Merge Join 算法存在的问题(OOM 异常问题, Shuffle 开销问题和负载不均衡问题等)进行了分析, 然后分析了内存监控的各项指标, 并建立内存监控模型来对集群资源进行分级和预警。最后, 结合第 3 章改进的读写接口和数据组织框架, 利用监控模型对 Join 数据进行切割和分批处理, 最后描述了该算法的步骤和实验过程。

第 5 章：工作总结和展望。本章主要对论文的研究工作进行了总结和展望, 提出了工作中的不足, 并展望了未来的研究工作。

第2章 Spark SQL 及 Hbase 技术基础

2.1 Spark SQL 数据组织框架

2.1.1 Spark SQL 查询机制

Spark SQL 是 Spark 框架中专门处理结构化数据的子模块，它简化了数据处理的复杂度，使用 Spark SQL 可以省去大量繁琐的非核心工作，使得用户更加专注于数据处理本身。Spark SQL 在 RDD 上面构建了通用的 SQL 组件，正如 Hive 在 HDFS 上构建的 SQL 组件一样，它们都能让用户通过简单的 SQL 语句实现复杂的数据处理逻辑。更为重要的是，Spark SQL 继承自 Hive 数据仓库，并优化和扩展了其结构，因此它们的语法绝大部分能互相兼容。

Spark SQL 目前拥有自己独立的 SQL 解析器 Catalyst，但也可以使用 Hive 的解析器。完全可以把 Spark SQL 看作 Hive 的升级版本，其数据在数据仓库中是以分块文件（即 Partition 文件）的形式存在，可以手动选择存储格式如 Text、Parquet、ORC 等格式，也可以手动选择压缩算法如 GZIP、LZO、SNAPPY、LZIP 等。

在这种数据仓库中，SQL 语句通过 Spark SQL 的 API(Dataset、Console、JDBC 等)传递给 Catalyst 解析器进行一系列的解析，最终转换成 RDD 执行。Catalyst 主要由以下几个组件组成^[6]：

SQLParse：解析 SQL 语法，然后生成一个未解析的逻辑计划(Unresolved LogicalPlan)。

Analyzer：将上一步的逻辑计划和相应的元数据结合起来生成一个解析过的逻辑计划(Resolved LogicalPlan)。

Optimizer：对上一步的 Resolved LogicalPlan 按照一定的策略进行优化，得到一个优化的逻辑计划(Optimized LogicalPlan)。

Planner：将上一步优化之后的逻辑计划转换成可执行的物理计划(PhysicalPlan)。

CostModel：基于代价模型选择出一个最佳的物理计划，并提交给 Spark 执行。其工作流程如图 2.1 所示。

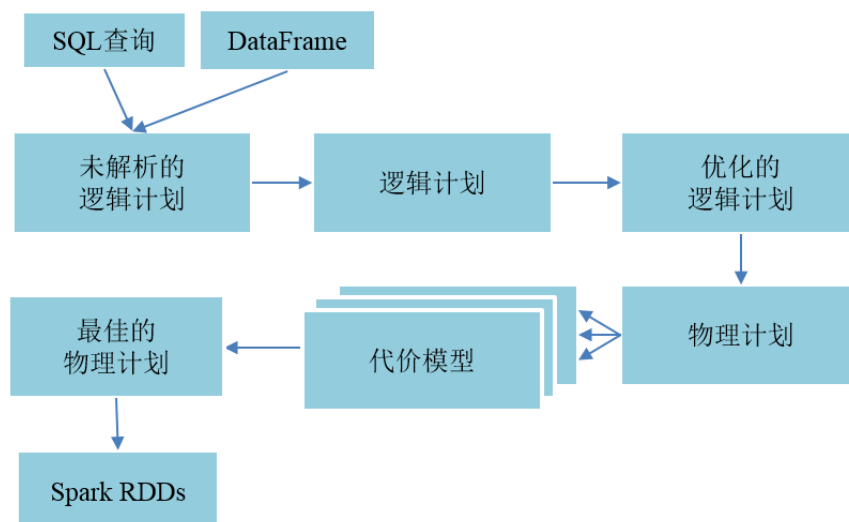


图 2.1 Catalyst 主要组件及执行流程

因此一条 SQL 语句通过 Catalyst 解析器的 SQLParse、Analyzer、Optimizer、Planner、CostModel 等组件完成了从逻辑计划到物理计划的转变，并最终生成 RDD 去执行。该 RDD 最后会加载对应的 Partition 文件完成计算，即本质上 Spark SQL 操作的是 Partition 文件。

2.1.2 Spark SQL 文件格式

Spark SQL 扩展了多个外部数据源^[45]，数据源可以是 Text、JDBC、CSV、JSON、ORC、Parquet 等。数据源加载后默认抽象成结构化的 Dataset 格式，这是一个分布式的结构化数据集合，类似于一张二维表在内存中的视图。Dataset 可用的操作和 RDD 一样，分为用转换和动作算子，但是 Dataset 的封装比 RDD 更简洁易用。

Spark SQL 内置数据仓库中的文件组织方式主要有两种：面向行存储和面向列存储。面向行存储就是把同一行记录存在一起，虽然每一行的记录可以使用一些算法压缩数据，但每次读写都是以行为单位，当只需要某几个字段的时候也会读取整行，浪费了很多读写开销。面向行存储的 SequenceFile^[46]是以键值对的形式存储的二进制文件、而 Avro^[6]是支持数据密集型的二进制文件格式，它比 SequenceFile 有更好地序列化和反序列化效果。

面向列存储格式可以灵活地跳过不需要的字段，因为它是相同列的数据存储在一起，查询特定列时不需要扫描一整行数据，而且每列的数据类型都一样，这使得压缩算法更加高效。常见的列存储格式有 Parquet、ORC 等，其中 ORC 是压缩效率

最高的，它先按行切分，再按列切分。而 Parquet 文件由不同的工具生成其性能也不一样，其中 Spark SQL 生成的 Parquet 文件性能是最优的^[27]。但 Parquet 是综合性能最好的，也是第三方支持最多的格式，所以 Spark SQL 默认采取的就是 Parquet 格式加 SNAPPY 压缩。

因为这些文件格式都采用了一定压缩算法来压缩存储空间，压缩过的数据都是十分精简却不完整的，所以在数据未解析还原之前无法快速定位到某一条数据，也无法修改和删除某一条数据。这也是为什么 Spark SQL 操作表的最小粒度是 Partition 文件，而不是单条记录。

2.1.3 Parquet 文件格式

Parquet 文件格式是被广泛应用的存储格式，其压缩比和解压速度都很出色。它的主要特点是可以跳过不需要的列，扫描性能较好^[47-48]。Parquet 格式是一个可嵌套的数据结构，包含了字段的嵌套关系和每个字段的值，最终形成树结构，这便是 Parquet 的 schema 信息。本文以 4G 行业卡数据的基础表 tb_imsi_tel 为例进行说明，其字段如表 2.1 所示。

表 2.1 tb_imsi_tel 表的字段定义

列名	数据类型	备注
IMSI	int64	国际移动用户识别码
TelNum	int64	用户电话号码
info.CreateTime	string	创建时间
info.APN	string	绑定的 APN 名称

表 2.1 中的 info 表示一个复合字段，它包含了两个子内容即 CreateTime 和 APN，因此这样一个嵌套结构的 schema 信息如下：

```
message tb_imsi_tel
{
  required int64 IMSI;
  repeated int64 TelNum;
  repeated group info
  {
    optional binary CreateTime (UTF8);
    optional binary APN (UTF8);
  }
}
```

```
}  
}
```

Parquet 格式中的 binary 和 UTF8 表示 string 类型, int64 表示 long 类型。如果将该 schema 转成一棵树, 则表名 tb_imsi_tel 为根节点, IMSI、TelNum、CreateTime、APN 则是其叶子节点, 而 info 则是分支节点。即基本类型(如 int、long、string 等)作为叶子节点, 该节点存放值; 复合类型(group)作为分支节点, 本身不含值。其中 required 表示该字段出现 1 次, repeated 表示出现 0 次或多次, optional 表示出现 0 次或 1 次。基本类型的数据即叶子节点还包含了三个属性^[49]: (Repetition level, Definition level, value), 遍历树时需要依赖 Repetition level 和 Definition level 来获取 value。由于字段定义在叶子节点, 具有父亲节点的依赖关系, 当从根节点开始遍历某一个字段的路径时发现某节点为空时(未被定义)的深度作为该字段的 Definition level。而 Repetition level 则表示该字段在哪个深度上是重复的。

从物理结构来看则是一个表的若干行数据先被横向切分成 Row group, 一个 Row group 包含了这若干行数据的所有列, 每一列称为一个 column chunk。一个 column chunk 又被划分成若干个 Page, 一个 Page 是压缩编码的基本单元, 里面存放该列的数据(Repetition level, Definition level, value)。

2.2 Hbase 数据组织框架

2.2.1 Hbase 查询机制

Hbase 作为一个物理的数据库, 而不是像 Spark SQL 那样的逻辑数据仓库, 它的表同时具有逻辑模型和物理模型, 所以数据在 Hbase 中的组织方式是经过精心设计并且要便于查询的。从逻辑上讲 Hbase 的数据是通过键值对定位的: 即通过组合键<rowkey, columnfamily, columnname, timestamp>就可以定位到唯一的值 Cell。行键 rowkey 是表的主键, 通过它可以快速定位数据, 如果不通过 rowkey 则面临着效率低下的全表扫描; 列族(column family)作为表结构(schema)的一部分, 列簇里面包含了若干列, 列名(column name)的前缀是列簇, 列可以动态的增加。相同列簇的列存储在一个文件中, 使用列簇便于控制具有相同属性的列。时间戳 timestamp 则控制单元格 Cell 中数据的版本, 即使向同一单元格插入数据也不会覆盖之前的时间, 因为每份数据都对应着一个时间戳, 即通过<rowkey, columnfamily, columnname,

timestamp>可以唯一的定位为一个 Cell 中的值。如表 2.2 所示的数据其逻辑视图按照 Json 的视角来看整个表结构就类似一个嵌套的 Json 字符串：

```
{
  rowkey1:[cf1:{name:Bob,age:23},cf2:{class: English,addr: Beijing }],
  rowkey2:[cf1:{name: Nick},cf2:{class: Math }]
}
```

表 2.2 Hbase 表的逻辑视图

行键 RowKey	时间戳 Timestamp	列簇 cf1		列簇 cf2	
		name(列名)	age(列名)	class(列名)	addr(列名)
rowkey1	t3	Bob	23		
	t2	Bob		English	
	t1	Bob			Beijing
rowkey2	t5	Nick		Math	
	t4	Nick			

而 Hbase 表的物理模型则是所有的行都按照 rowkey 的字典序排序，先横向将一个 rowkey 范围内的若干行分割为一个 Region，也是说一张表被划分为多个 Region，类似于数据仓库的多个 Partition 分区。然后每个 Region 里面按照列簇划分 Hstore，Hstore 由 MemStore 和 StoreFile 组成，一个列簇里的数据会优先存到内存中的 MemStore，当 MemStore 存满之后会刷写到磁盘上的 StoreFile 里面。而 StoreFile 是基于 HFile 实现的，也就是说数据最终是以 HFile 的格式存储在 HDFS 上的。HFile 是以键值对存储的 Hadoop 二进制文件格式，所以当写入 Hbase 的数据量很大的时候，可以通过 MapReduce 或者 Spark 程序直接生成 HFile 来完成数据的导入。

当查询请求从 Zookeeper 开始经过 ROOT 表到 META 表时就可以定位数据到所在的其中一个 Region，共经过三次请求，每次请求时间复杂度为 $O(1)$ 。而该 Region 里面有一部分数据缓存在内存中，多余的则溢写到磁盘中，为了提高查询速度，各个数据块按照树形结构组织起来。这样的查询机制决定了 Hbase 的查询能在毫秒级返回结果。

2.2.2 Hbase 文件格式

Hbase 的数据最终也是存储在 HDFS 上的数据块, Hbase 默认不会压缩数据块(即存储格式默认为 NONE)。为了压缩存储空间和提升查询速度, HBase 也可以采用压缩的存储格式如 GZ、LZ4、LZO、SNAPPY 等。选取不同的存储格式, 即不同的压缩算法对 Hbase 表的性能影响很大, 尤其是当数据量庞大时影响更为明显。GZ 是最常用的压缩算法, Hadoop 可以直接处理 GZ 格式的压缩文件; LZ4 是一种折衷的压缩算法, 既能保证压缩率, 又能保持很高的压缩和解压速度; LZO 则是偏重于解压速度的无损压缩算法; SNAPPY 也是一种广泛应用的无损压缩算法, 前身是 Zippy。不足之处是 LZO 和 SNAPPY 都需要额外安装复杂的依赖环境才能使用。

图 2.2 显示了 GZ、LZ4、NONE 三种压缩方式下的文件大小对比, 相较于不压缩(NONE), 采用 GZ 压缩最节省空间, LZ4 压缩居中。

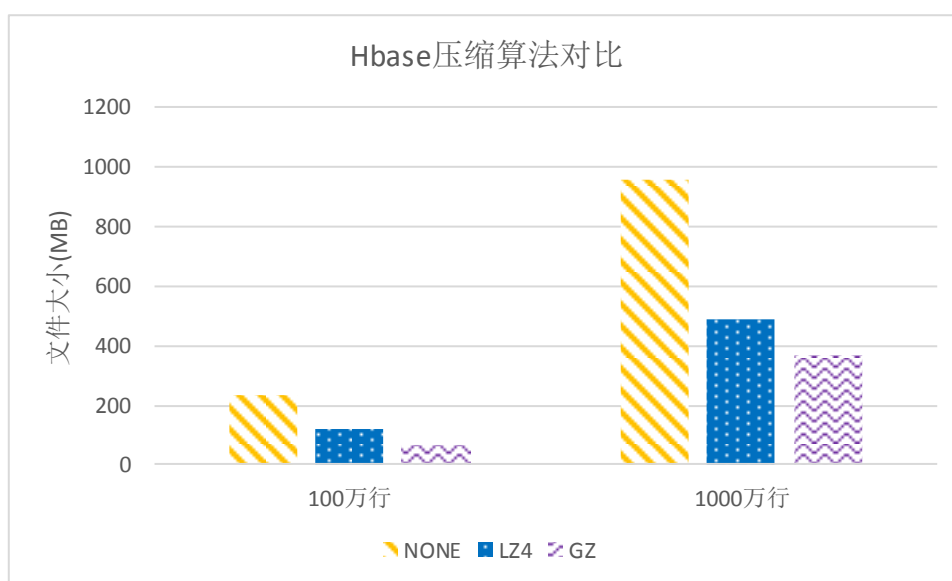


图 2.2 HBase 不同压缩格式下文件大小对比

而图 2.3 进一步显示了 GZ、LZ4、NONE 三种压缩方式下的查询时间对比, 相较于不压缩(NONE), LZ4 的查询时间最快, 而压缩率高的 GZ 查询时间则相对较慢。因此根据文件大小和查询时间的对比, 折衷选取 LZ4 压缩算法作为 Hbase 的文件存储格式可以平衡存储空间和查询时间。

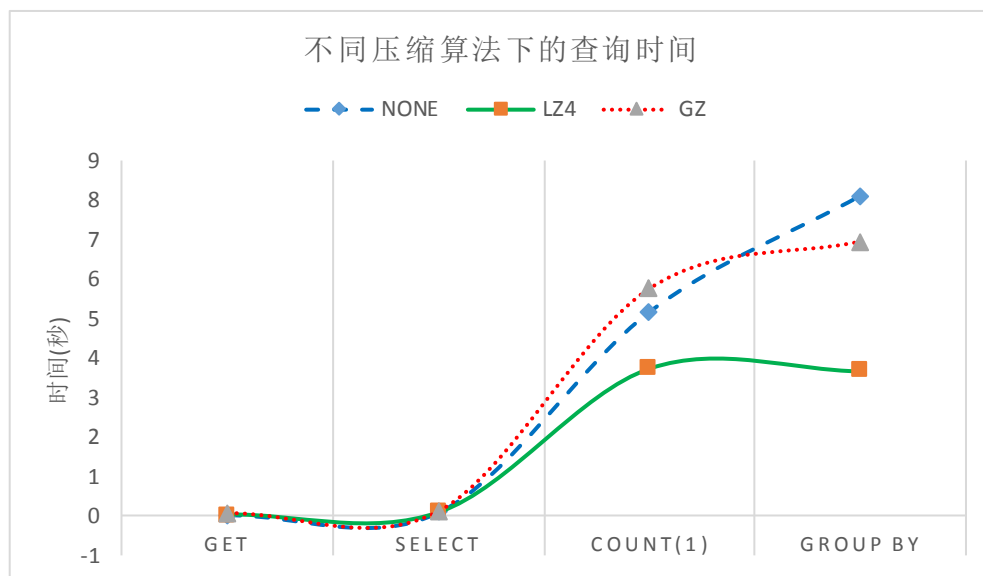


图 2.3 HBase 不同压缩格式下查询时间对比

2.2.3 Phoenix 映射关系

虽然 Hbase 可以快速的随机访问数据,但是它只能通过原生的底层 API(get、put 等)来操作数据,不能像 Spark SQL 那样便捷地使用 SQL 语句操作数据,这为使用人员带来了很大的不便。因此需要将 Hbase 的表与 Phoenix 进行关联映射。Phoenix 为 Hbase 构建了 SQL 解析层,使用 Phoenix API 可以将 SQL 语句转换为 Hbase 的 get、put 等操作。用户可以直接用 Phoenix 创建类似于关系数据库中的表,虽然其底层依然是 Hbase 表,但对用户来说 Hbase 是透明的。当然 Phoenix 表也可以直接与 Hbase 中已有的表关联起来,这样用户既可以按照 Hbase 的方式操作表,也可以按照 Phoenix 的方式操作表。

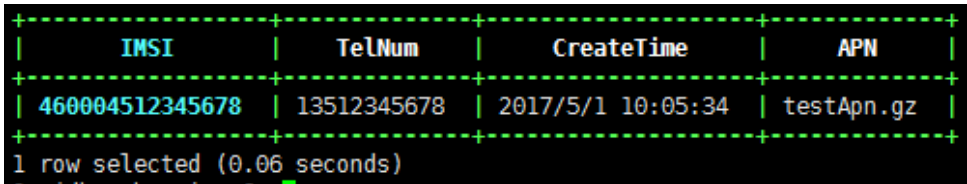
值得注意的是,如果先在 Phoenix 中使用 SQL 语句创建一张表,Phoenix 会默认将所有小写字母变成大写,此时在 Hbase 中查看到的表名便是大写的。下面以 2.1.3 节提到的 tb_imsi_tel 表为例说明两者结构的关联关系^[50]:

首先在 Phoenix 创建 tb_imsi_tel 表,显示为 TB_IMSI_TEL 表,然后 Hbase 会自动关联该表。该表的结构如下:

```
create table tb_imsi_tel (  
  IMSI varchar not null primary key,  
  TelNum varchar ,  
  CreateTime varchar ,
```

APN varchar);

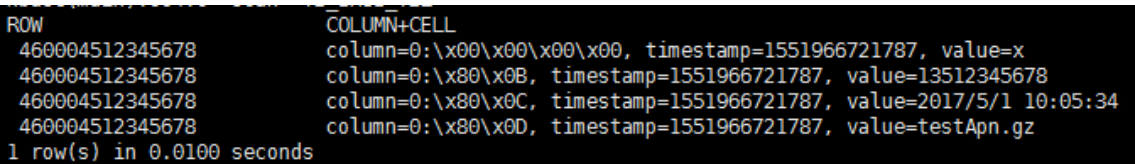
接着向表中插入一条数据(数据已脱敏处理): upsert into tb_imsi_tel values('460004512345678', '13512345678', '2017/5/1 10:05:34','testApn.gz')。则数据在 Phoenix 的显示效果如图 2.4 所示,而在 HBase 中查看到的结构如图 2.5 所示。



IMSI	TelNum	CreateTime	APN
460004512345678	13512345678	2017/5/1 10:05:34	testApn.gz

1 row selected (0.06 seconds)

图 2.4 tb_imsi_tel 在 Phoenix 中的数据视图



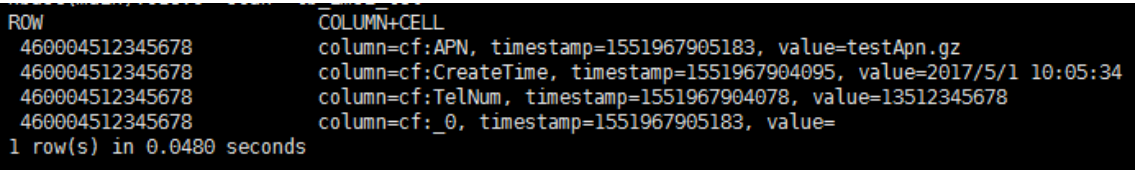
ROW	COLUMN+CELL
460004512345678	column=0:\x00\x00\x00\x00, timestamp=1551966721787, value=x
460004512345678	column=0:\x80\x0B, timestamp=1551966721787, value=13512345678
460004512345678	column=0:\x80\x0C, timestamp=1551966721787, value=2017/5/1 10:05:34
460004512345678	column=0:\x80\x0D, timestamp=1551966721787, value=testApn.gz

1 row(s) in 0.0100 seconds

图 2.5 先创建 tb_imsi_tel 的结构

从中可以看出 tb_imsi_tel 表在 Phoenix 中的主键 IMSI 对应于 HBase 中 RowKey 默认名 ROW, 默认列簇为 0, 冒号后面为十六进制表示的列名, value 表示该列的值。也就是说 Hbase 的 RowKey 对应着 Phoenix 的主键, 而“列簇:列名”则对应着 Phoenix 的列。

但是这样的关联关系并不直观, 所以应该先在 Hbase 创建表 tb_imsi_tel, 然后在 Phoenix 创建同名表"tb_imsi_tel"(表名及字段应加双引号, 如果都用大写字母则可以不用加)。这样做的好处在于 Hbase 表中的列是手动创建的, 并且与 Phoenix 表逐一对应起来了。那么先创建的 Hbase 中 tb_imsi_tel 的数据结构如图 2.6 所示。



ROW	COLUMN+CELL
460004512345678	column=cf:APN, timestamp=1551967905183, value=testApn.gz
460004512345678	column=cf:CreateTime, timestamp=1551967904095, value=2017/5/1 10:05:34
460004512345678	column=cf:TelNum, timestamp=1551967904078, value=13512345678
460004512345678	column=cf:_0, timestamp=1551967905183, value=

1 row(s) in 0.0480 seconds

图 2.6 后创建 tb_imsi_tel 的结构

然后在 Phoenix 中创建关联表 tb_imsi_tel, 其结构则为:

```
create table "tb_imsi_tel" (  
  IMSI varchar not null primary key,
```

```
"cf"."TelNum" varchar ,  
"cf"."CreateTime" varchar ,  
"cf"."APN" varchar );
```

关联之后的 Phoenix 数据显示效果与图 2.4 完全一致,即将 Hbase 与 Phoenix 关联起来后数据都会自动同步。好处在于 Hbase 中对应的列不再是转码后的十六进制字符,而是手动指定的列名,因此在编写程序时能直接通过 Hbase 原生 API 获取指定的列名。

虽然 Phoenix 通过 SQL 映射简化了 Hbase 的读写过程,但 Phoenix 的缺点是当数据量很大时使用 Phoenix SQL 导入数据速度较慢;此外 Phoenix 对复杂的 SQL 语法支持度不够,仅能进行简单的 SQL 语句操作。

2.3 Spark 资源管理机制

2.3.1 Spark 运行机制

Spark 作为一个分布式的内存计算框架,主要是通过 Master 和 Worker 节点的共同作用来完成工作的,因此形成了一个分布式的主从结构。Spark 只是一个计算框架,并不负责数据的持久化,因此外部数据源有多种选择。由于 HDFS 的性能良好,通常 Spark 的外部数据源为 HDFS,即基于 Hadoop+Spark 的解决方案。

因此 Spark 的 Worker 节点和 Hadoop 的 DataNode 一致,每个文件分布在集群中的 DataNode 上,为了保证数据的本地性,每个 Worker 会优先读取属于本地的文件。这样集群的多个 Worker 一起协调工作完成了并行工作,其工作框架如图 2.7 所示。设图中的集群有 1 个 Master 节点, k 个 Worker 节点,每个 Worker 节点上默认运行一个 Executor 进程(也可以配置多个 Executor 来提高并行度,但是会增加 JVM 开销)。Executor 即一个 JVM 进程,是 Worker 节点中真正执行任务的进程。如果 Spark 操作一个大小为 m 的表,假设它被平均分配到 k 个节点存储,则每个节点处理的数据规模降到 $\frac{m}{k}$ 。

每个 Worker 节点只需要处理 $\frac{m}{k}$ 的数据, k 个节点同时工作便缩短了整个程序的运行时间。Worker 节点会给每个 Executor 进程分配若干个 CPU 核心,一个 CPU 核

即是 Executor (JVM)的一个工作线程，对应着一个 Spark 应用程序的 Task，独占一个 Partition 文件。

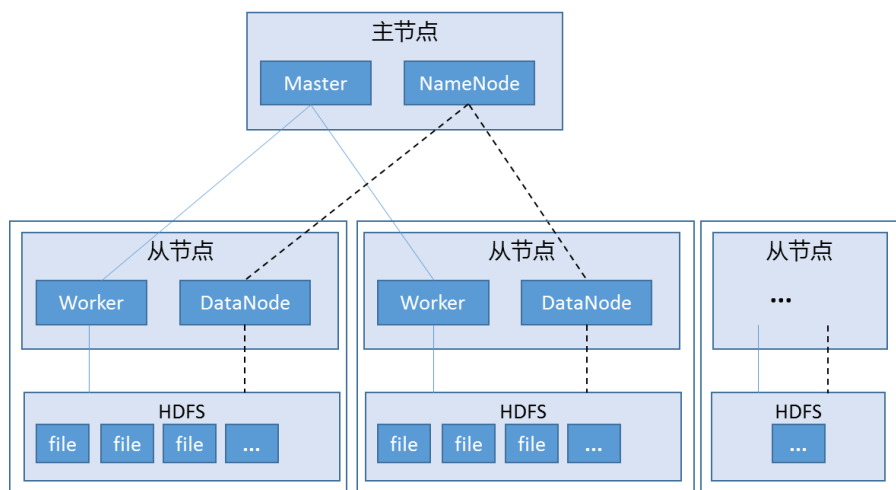


图 2.7 Spark 集群工作框架

每个 Worker 节点可以启动一个或多个 Executor(一个 Executor 进程就是一个 JVM); 一个 Worker 默认情况下分配一个 Executor(默认一个 JVM, 多个 JVM 消耗较大); 每个 Executor 由若干 core 组成, 每个 Executor 的每个 core 一次只能执行一个 Task; 一个 core 就是 Executor (JVM)的一个工作线程。每个 Task 执行的结果就是生成了目标 RDD 的一个 Partiton, 也就是一个 Partition 文件。因此 Spark 应用程序的作业并行度 p 定义如下:

$$p = \tau * k * v \quad (2.1)$$

其中, τ 表示每个 Worker 节点的 Executor 进程数目, 默认为 1 个 Executor 进程, 即 $\tau = 1$ 。 k 表示上述的 k 个 Worker 节点, 因此 $\tau * k$ 表示整个集群的 Executor 数量。 v 表示每个 Executor 的 CPU 核数, 而并行度 p 也就是整个集群所拥有的 CPU 总核数。正是因为 p 个 CPU 核的并行工作才缩短了整个程序的运行时间。

一个应用程序通过层层解析后最终会落到具体的硬件资源进行计算, 因此 Spark 作业与 Spark 集群之间的关系如图 2.8 所示。如果一个 RDD 有 100 个分区(Partition 文件), 那么计算的时候就会生成 100 个 task; 如果集群配置为 10 个计算节点(Worker), 每个两 2 个核, 同一时刻可以并行的 task 数目为 20; 所以计算这个 RDD 的 100 个任务就需要 5 个轮次能执行完。如果有 101 个 task 的话, 就需要 6 个轮次, 在最后一轮中, 只有一个 task 在执行, 其余核都在空转。

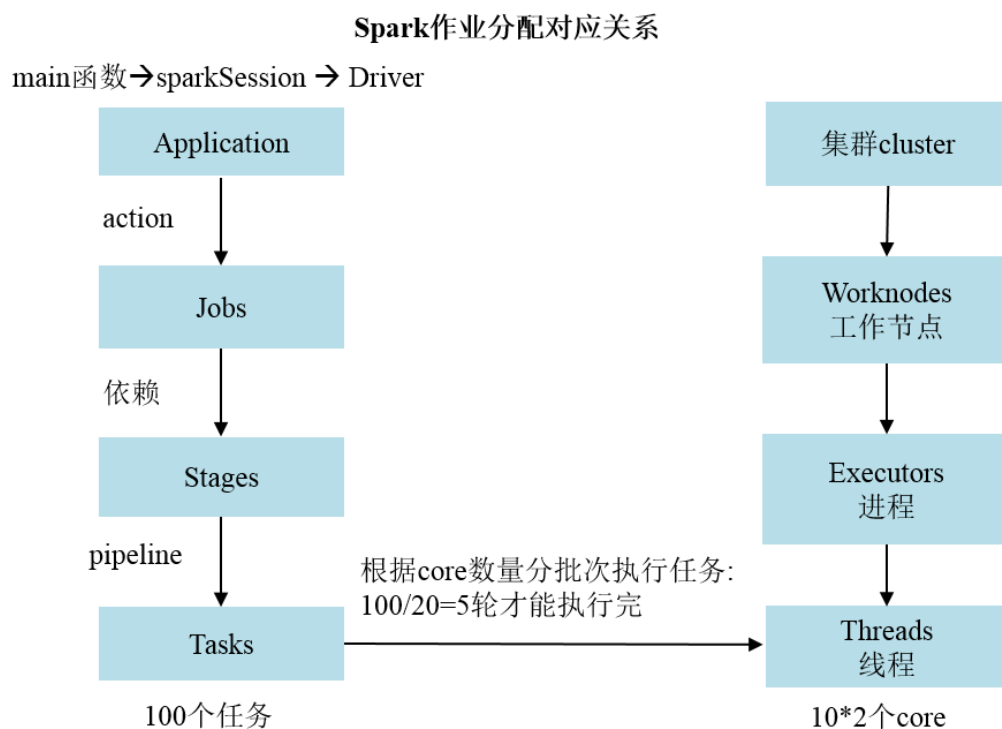


图 2.8 Spark 作业与集群资源对应关系图

2.3.2 Spark 内存模型

Spark 是基于 Java 的运行环境(Java Virtual Machine, JVM), 并在此基础上做了一些优化, 但 JVM 总体架构不变。JVM 内存模型如图 2.9 所示。

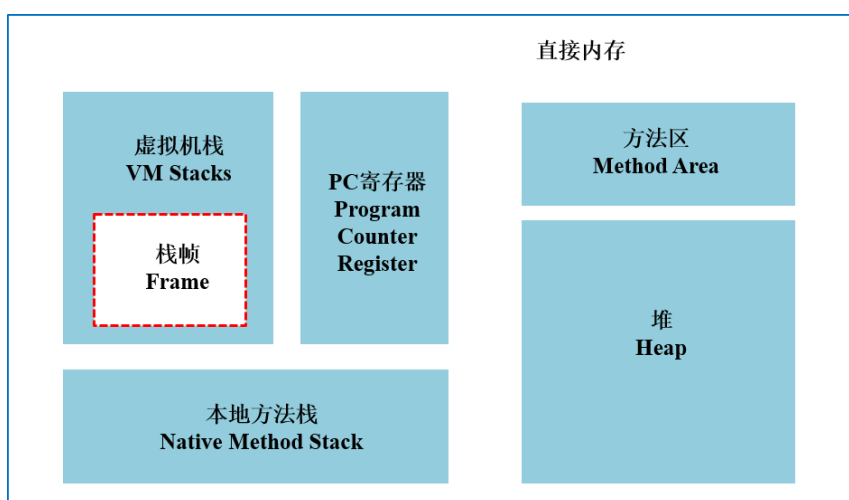


图 2.9 JVM 内存模型

如图 2.9 所示, 虚拟机栈是线程私有的, 每个线程都有一个栈, 栈里面包含了栈帧, 用于存放局部变量, 栈的大小可以调整, 当调用深度超过最大值便会抛出

`StackOverflowError`。本地方法栈主要是调用非 Java 代码的接口,如 C++编写的 SDK。PC 寄存器即程序计数器,也是线程私有的,即每个线程都有自己的计数器,保存了当前执行指令的地址。而方法区是所有线程共享的区域,也就是静态区,本质上是堆的一部分,主要存储类的信息、常量池、方法数据、方法代码等。堆也是所有线程共享的,所有的对象和数组等都在堆上进行分配。这部分空间可通过垃圾回收器 (Garbage Collector, GC)进行回收,当申请不到空间时会抛出 `OutOfMemoryError`。

当 Spark 程序处理的数据规模较大时经常会出现内存不足的情况,抛出 `OutOfMemoryError(OOM)`和 `StackOverflowError` 等异常。因此 Spark 对 JVM 堆空间作了重新划分^[51]: 在 Spark2.x 版本中,默认预留 300M,剩余内存的 40%用于存储用户自定义数据结构或者 Spark 元数据等,剩余内存的 60%作为统一内存,其中 Storage 内存和 Execution 内存各占统一内存的一半,但是二者可以动态地相互借用。Storage 内存用于缓存数据,Execution 内存用于执行 shuffle 时缓存中间数据。Spark 的内存模型如图 2.10 所示。

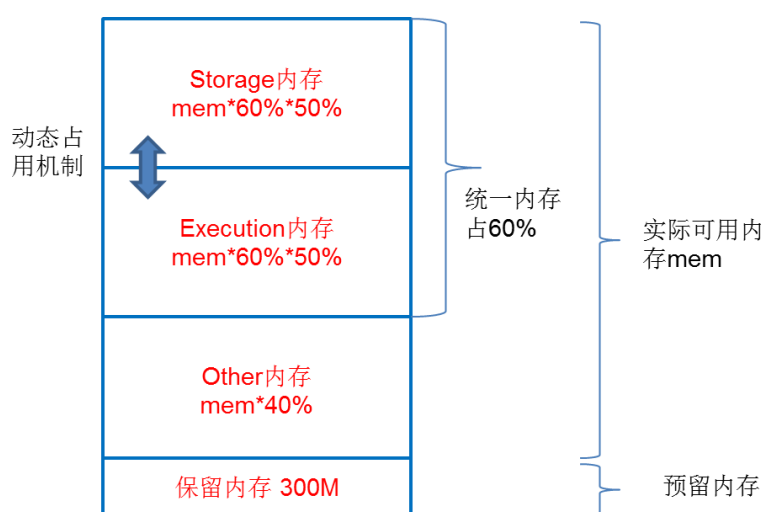


图 2.10 Spark 内存模型

其中, Storage 内存和 Execution 内存的动态占用规则如下:

- (1)当两部分都被占满时,都需要溢写到磁盘。
- (2)当 Storage 满而 Execution 未滿时,Storage 可以占用 Execution 内存,由于 Execution 优先级比 Storage 高,如果此时 Execution 内存不够用则将驱逐 Storage 内存。
- (3)当 Execution 满而 Storage 未滿时,Execution 可以占用 Storage 内存,由于

Execution 优先级比 Storage 高，如果此时 Storage 不够用，则只能等待 Execution 执行完释放内存，不能驱逐 Execution 内存。

当提交一个 Spark 应用程序时首先需要向集群申请固定的内存和 CPU 资源，这作为已知变量。依然采用上一节的假设，即假设一个集群有 1 个 master 节点， k 个 worker 节点，现在为每个节点分配 x GB 内存， y 个 CPU cores，那么该应用程序所获得的总资源为：

Master: mem= x GB, cpu= y cores

Worker: mem= $x*k$ GB, cpu= $y*k$ cores

在 Java8 中元空间占用堆外内存不纳入计算，因此分配的内存 x GB 实际上为年轻代与老年代之和。但是 JVM 的一些其他开销使得实际可使用的内存比 x 稍小，实际可使用内存 $x = \text{Runtime.getRuntime.maxMemory}()$ 。

假设通过参数^[52] “Spark.executor.memory” 设置每个 executor 的内存为 $x=2\text{GB}=2048\text{m}$ ，并指定“Spark.executor.extraJavaOptions=-Xms2048m -XX:NewSize=1024m -XX:NewRatio=1 -XX:SurvivorRatio=6”那么 GC 日志中的堆大小情况如下：

（默认是并行 GC 算法，该 GC 日志经过整理后省略了非关键信息）

PSYoungGen total 917504K=896M, used 49092K [0x00000000c0000000...)

eden space 786432K=768M, 0% used [0x00000000c0000000...)

from space 131072K=128M, 37% used [0x00000000f0000000...)

to space 131072K=128M, 0% used [0x00000000f8000000...)

ParOldGen total 1048576K=1024M, used 80K [0x0000000080000000...)

object space 1048576K, 0% used [0x0000000080000000...)

Metaspace used 20862K, capacity 21128K=20.63281M,committed 21248K, reserved 1067008K

class space used 2932K, capacity 3032K, committed 3072K, reserved 1048576K

其中 PSYoungGen 和 ParOldGen 分别是年轻代和老年代收集器的名称。

因此人为配置的总内存：

$$\begin{aligned}\text{ExecutorMemory} &= \text{eden} + \text{from} + \text{to} + \text{ParOldGen} = 768 + 128 + 128 + 1024 \\ &= \text{PSYoungGen} + \text{to} + \text{ParOldGen} = 896 + 128 + 1024 \\ &= 2048 \text{ MB}\end{aligned}$$

而实际上可用内存：

$$\text{Runtime.getRuntime.maxMemory}() = \text{PSYoungGen} + \text{ParOldGen} = 896 + 1024$$

$$\begin{aligned}
 &= \text{eden} + \text{from} + \text{ParOldGen} = 768 + 128 + 1024 \\
 &= 1920 \text{ MB}
 \end{aligned}$$

因此 Spark 中实际可供存储的内存 $\text{maxMem} = (1920 - 300) * 0.6 = 972 \text{ MB} = 1019215872 \text{ B}$, 且在 Spark 中可以通过上一节中的 executors 监控接口获取。即 2GB 内存的 Executor 真正可用于存储数据的最大空间为 972MB, 但实际上还会比 972MB 更小。

此外内存分配比例可以通过 Spark 参数调整: 参数“Spark.memory.fraction”控制统一内存, 默认为 0.6; 由于统一内存的 Storage 内存和 Execution 内存可以动态借用, 假设初始时统一内存全部被 Storage 内存占用, 则可以最大化的存储数据。设统一内存所占比例的值为 ε , 令 $x = \text{Runtime.getRuntime.maxMemory}()$, x 默认单位是 GB, 那么实际上单个节点的可用于存储数据的内存大小 M' 为:

$$M' = (x * 2^{10} - 300) * \varepsilon \quad (2.2)$$

其中, $M' < x * 2^{10}$, 且单位为 MB。

2.4 Spark Join 算法

2.4.1 分布式 Join 算法分类

Join 是数据库系统中的重要操作, 即匹配两个及以上的表的若干行数据。传统的关系型数据库中 Join 有三种常见的实现算法^[24]: Nest Loop Join、Hash Join、Sort Merge Join。而分布式实现的 Join 算法则有多个变种^[53]: 如基于 MapReduce 框架的 Map Side Join、Reduce Side Join、Semi Join 等; 基于 Hive 的 Bucket Map Join、Sort Merge Bucket Map Join、Skew Join 等; 基于 Spark 的 Broadcast Hash Join、Shuffle Hash Join、Sort Merge Join 等。总结起来, 分布式的 Join 算法都是在传统的 Join 算法的基础上, 加入了具有分布式特点的 Shuffle、Broadcast 等实现。

对于大表和小表的情况, 通常是把小表放到内存里面构建 Hash 表并广播 (Broadcast) 到所有节点, 然后扫描大表并 Hash 查找对应的 Key; 对于大表和较大表的情况, 即内存装不下小表时 (当小表数据增大时称小表为较大表), 就会触发耗时的 Shuffle 操作进行数据的重分区, 然后再进行相关的 Hash Join; 而对于两个大表的情况, 通常采用排序合并连接 (Sort Merge Join) 算法, 即先对两表分别排序再进行

合并连接,这时排序成为了主要的性能消耗,而且排序还可能涉及重分区和 Shuffle 等代价。按照上述原则,Spark 上的 Join 实现对于大表和小表采用 Broadcast Hash Join,对于大表和较大表采用 Shuffle Hash Join,对于大表和大表采用 Sort Merge Join。

2.4.2 BloomFilter 算法

BloomFilter(BF)主要是检索一个元素是否在一个集合中,可以高效地压缩海量数据,还能提高检索速度^[20]。传统的检索结构大多是用时间换空间,或者用空间换时间,而 BloomFilter 创造性地通过牺牲一定的准确率,来同时提升了空间和时间利用率。BloomFilter 判断一个元素不在集合中一定是正确的,如果判断一个元素在集合中则有可能出现误判。

BloomFilter 初始时有一个 m 位的位组(可用数组表示),默认每一位都为 0。有一个长度为 n 的集合 $V = \{v_1, v_2, \dots, v_n\}$,要将集合的每一个元素存入长度为 m 的位组,那么便需要对每一个元素进行一些转换和映射,得到一个“地址”,将这个地址再存入位组便可以大幅地节省存储空间。因此需要用 k 个独立的哈希函数将集合的每个元素(设为第 i 个元素)映射成 k 个值 $\{v_{i1}, v_{i2}, \dots, v_{ik}\}$,这 k 个值将对应着位组中的 k 个位置,再根据每个值将位组中对应的位置 1。如此经过若干个元素的映射后,位组的某一位可能会被覆盖,也就是说多个元素会复用位组的位,因此便达到了压缩数据的目的。

当位组训练完毕后,进行检索的过程也同上。即先将一个元素通过 k 个哈希函数运算成 k 位,再去位组中查这 k 位是否全为 1,如果是即返回 True,反之即为 False。由于多个元素对应的位会重复,因此可能出现某 k 个位置上全为 1,如果某个不存在于集合中的元素经过运算后刚好这 k 位全为 1,于是便出现了误判。此外由于位组的一个位可能被多个元素复用,删除元素的时候将其置 0 可能影响其他元素,因此 BloomFilter 的另外一个缺点是删除元素困难。

2.5 本章小结

本章主要介绍了 Spark 和 Hbase 相关的技术基础。首先介绍了 Spark SQL 的数据组织框架,阐述了 Spark SQL 的查询机制,并分析了它的文件格式。然后介绍了

Hbase 的数据组织框架，阐述了 Hbase 的查询机制和文件格式，分析了 Phoenix 的映射关系。接着讲述了 Spark 的资源管理机制，包含了内存模型以及参数管理等内容。最后描述了 Spark Join 算法相关的技术，分布式 Join 算法分类和 BloomFilter 算法。

第 3 章 Spark SQL 数据组织方式设计

3.1 Spark SQL 问题分析

3.1.1 数据读写问题

Spark SQL 读取数据是通过 `DataFrameReader` 将外部数据源转换为 `Dataset` 的过程，而写数据是通过 `DataFrameWriter` 将 `Dataset` 转换为外部数据格式的过程。虽然 `DataFrameReader` 和 `DataFrameWriter` 接口简单易用，但是其读写速度较慢，因此实际运行中大部分时间和资源消耗在数据读写的过程。一方面原因是由于接口本身的问题，外部数据源与 `Dataset` 之间的相互转换过程复杂，不仅消耗了大量的资源，当数据量较大时还有频繁的 IO 开销；另外一方面则是由于 Spark SQL 操作的粒度是 HDFS Partition 文件，Partition 文件没有索引严重影响了数据的检索速度，如果 Partition 文件过小则会进一步加重这种情况，而且 Partition 文件的性能受不同存储格式的影响较大。

3.1.2 数据存储问题

Spark SQL 受数据存储格式的影响较大，也即是数据的组织方式。Spark SQL 数
据仓库操作的是 HDFS Partition 文件。从 HDFS 目录的视角来看，数据仓库是一个 HDFS 上的一个目录(如: `/user/spark/warehouse`)，数据库则是该目录的一级子目录(如: `/user/spark/warehouse/test.db`)，表则是数据库目录的一级子目录(如: `/user/spark/warehouse/test.db/tb_stu`)，而一个表真正的数据，即 Partition 文件则位于该表目录下。如果该表的数据量很大，就会拆分成多个 Partition 文件存储。当文件数量达到一定程度时，就会严重影响查询速度，因为很有可能触发全表扫描。这时就需要在该表目录下建立子目录将多个 Partition 文件分类存储，即建立表的分区目录如：

```
/user/spark/warehouse/test.db/tb_stu/time=201801  
/user/spark/warehouse/test.db/tb_stu/time=201802  
/user/spark/warehouse/test.db/tb_stu/time=201802/h=12:00:00
```

/user/spark/warehouse/test.db/tb_stu/time=201802/h=13:00:00

这样层层嵌套的目录结构再加上元数据就构成了逻辑上的数据仓库。Spark 的 SQL 语句最终会操作上述目录中的文件，因此 Spark SQL 本质上可以看作是为文件构建了一层 SQL 映射关系。因此一张表(或一个目录下)的数据大小 F 定义为：

$$F = \sum_{i=1}^j f_i \quad (3.1)$$

其中， j 表示 Partition 文件的总数， f_i 表第 i 个文件的大小。可以看出数据仓库的数据组织方式关键在于 Partition 文件本身。通常压缩每个 Partition 文件可以节约存储空间，还能一定程度地提升处理速度。假设一张表的数据总条数为 n 条，总容量为 F 字节(见公式(3.1))，那么平均每 1 条记录所占空间大小 δ 为：

$$\delta = \frac{F}{n} \quad (3.2)$$

其中， δ 就是 Partition 文件的大小衡量标准，假设表的记录总数 n 固定不变，只要降低了每条记录的 δ ，便可以降低 Partition 文件的容量，从而降低整个表的容量。而降低 δ 则可以通过不同的存储算法来实现。

3.1.3 存储格式对比分析

Spark SQL 提供了多种行式或列式存储格式，如 Avro、SequenceFile、Parquet、ORC 等，它们的适用场景和压缩率都不一样。其中 SequenceFile、Avro 等都是按照行存储的格式，而 Parquet、ORC 等都是按照列存储的格式。虽然行式存储和列式存储都有各自的特点，但列式存储更加灵活，可以只读取特定的字段，过滤掉不需要的字段，而行式存储则只能读取一整行。相较之下行式存储浪费读写开销，列式存储更节省读写开销。图 3.1 显示了同一张表转换为不同存储格式时的文件大小对比关系，相较于默认的文本格式(Text)，ORC 文件格式最节省存储空间，Parquet 文件次之，它们均为列式存储格式。而剩余的三种均为行式存储格式，即 Avro、SequenceFile 和 Text 文件格式，它们则占用更多的空间。SequenceFile 是一种基于键值对的二进制存储格式，它最占用空间。其次便是默认的 Text 文件格式，它仅简单的将每一行按照字符串存储，占用空间大小排倒数第二。而 Avro 作为一类 Json 格式的存储形式，它占用的空间大小紧排 Text 其后。综上可以看出 Spark SQL 中数据按照列式组织比按照行式组织更加节省空间。

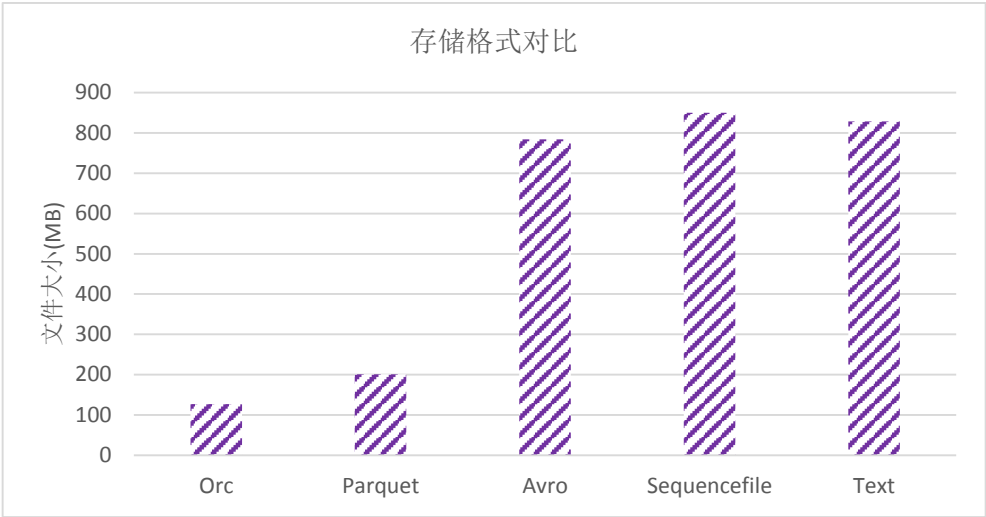


图 3.1 不同存储格式的文件大小对比

文件的组织形式不仅影响占用存储空间的大小，还影响了数据的处理速度。图 3.2 则显示了 4 种不同类型的 SQL 语句(见表 3.1)分别在上述 5 种存储格式下的查询时间对比。

表 3.1 四种不同类型的 SQL 测试语句

语句 1	<pre>with tab_a as (select from_unixtime(cast(startdate/1000000 as int),"yyyy-MM-dd") as etl_date,city as city_id,sum(1) as http_req_times,sum(case when code > 0 and code < 400 then 1 else 0 end) as http_rsp_succ_times,sum(case when code <= 0 or code >= 400 then 1 else 0 end) as http_rsp_fail_times,sum(case when code > 0 and code < 400 then responsetime else 0 end) as first_rsp_succdelay,sum(responsetime) as first_rsp_delay from tb_sque where p_app = 5 and apptype = 5 group by from_unixtime(cast(startdate / 1000000 as int), "yyyy-MM-dd"), city) select a.etl_date,a.city_id,case when http_req_times > 0 then round(http_rsp_succ_times / http_req_times * 100, 2) else 0 end as http_rsp_succ_rate,case when http_rsp_succ_times > 0 then round(first_rsp_succdelay / http_rsp_succ_times / 1000, 2) else 0 end as http_rsp_succdelay,http_req_times as total_http_req_times,http_rsp_succ_times as total_http_rsp_succ_times,first_rsp_succdelay as total_first_rsp_succdelay,first_rsp_delay as total_first_rsp_delay from tab_a a;</pre>
语句 2	<pre>select count(1) from tb_squire;</pre>
语句 3	<pre>select responsetime>windowssize,version from tb_squire where startdate>=1515628800000000 and enddate<=1515718800000000;</pre>
语句 4	<pre>select apn,count(1) from tb_squire group by apn;</pre>

对于 4 种不同类型的 SQL 语句，查询时间最快的是 Parquet 格式，ORC 格式次之。而行式存储的 Avro、SequenceFile、Text 查询时间均比列式存储的 Parquet、ORC 慢很多。此外，结合图 3.1 可知，虽然 ORC 比 Parquet 更节省空间，但是 Parquet 查询时间比 ORC 快。这是因为 Parquet 文件由不同的工具生成其性能也不一样，其

中 Spark SQL 生成的 Parquet 文件性能是最优的,也就是说 ORC 格式在 Hive 上表现较好,而 Parquet 格式在 Spark SQL 上表现较好。由此也可以看出 Spark SQL 默认采用的 Parquet+Snappy 格式是折衷考虑了压缩大小和查询时间的最优选择。

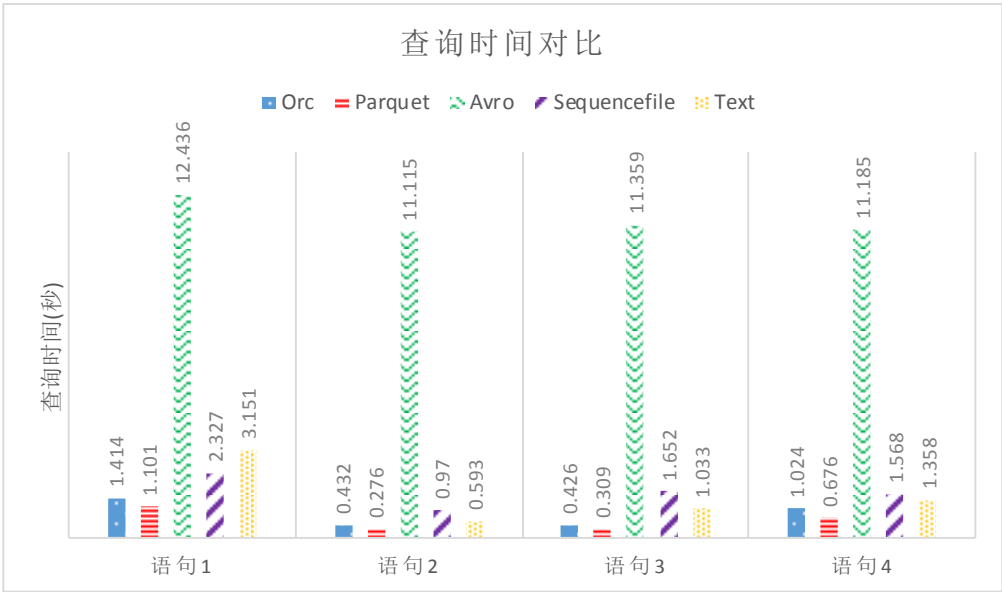


图 3.2 不同存储格式的查询对比

由此可见对于结构化数据的 SQL 查询来讲,列式存储的综合性能表现比行式存储更好,结合应用场景来讲则是 Spark SQL 操作 Parquet 文件格式性能最佳。

3.1.4 Spark SQL 与 Hbase 整合分析

Spark SQL 有独立的 Catalyst 的解析器,可以兼容绝大多数的 SQL 语法,其缺点是数据的加载和持久化较慢,且查询速度也较慢。相较之下 Hbase 的查询速度特别快,能在毫秒级返回结果,但是 Hbase 默认不支持 SQL 操作,要借助 Phoenix 才能实现 SQL 查询。而 Phoenix 的 SQL 语法兼容性不好,仅仅支持常用的简单查询,不支持复杂的 SQL 操作。图 3.3 明确地显示了 Spark SQL 与 Hbase 的查询差异。

如图 3.3 所示,Spark SQL 查询时有分区比没有分区快,这是因为分区表可以看作一种粗粒度的索引机制。而 Hbase 通过 RowKey 查询一条记录耗时仅为 0.154 秒,再和 Phoenix 关联后通过 RowKey 查询也仅为 0.207 秒,即 Hbase 的简单查询 (select/get)比 Spark SQL 快。如果在 Phoenix 关联表上创建二级索引后查询也在毫秒级,但是如果查询非主键和非索引字段则耗时较长。但值得注意的是对于 count()、

sum()、max()等复杂操作，Spark SQL 的查询时间则比 Hbase 快，且 Hbase 原生 API 还不支持 sum()、max()函数，因此图 3.3 中 HBase 的 sum()和 max()查询显示为 0。

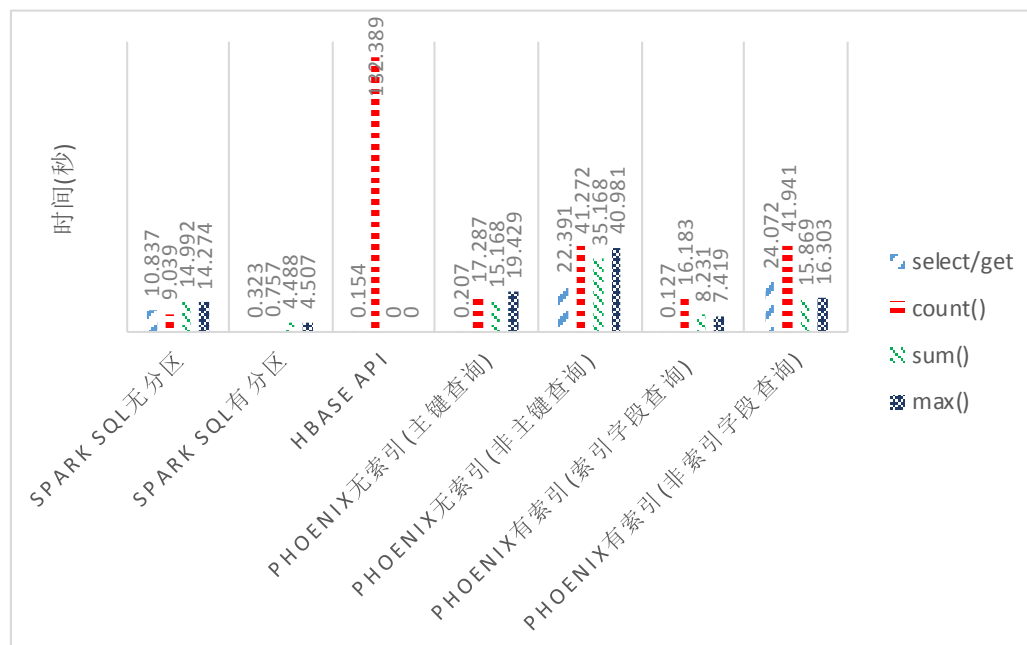


图 3.3 Spark SQL 与 Hbase 的查询对比

由此可知，对于简单查询 Hbase+Phoenix 明显优于 Spark SQL，如果说 Hbase 能像数组一样随机存取，则可以把 Spark SQL 比作链表。但是 Hbase 针对复杂的聚合操作如 count()、sum()、max()、以及 group by 等则不如 Spark SQL，而且 Phoenix SQL 还不支持如表 3.1 中的 with as 等复杂的语法。所以他们都不能独立地完全兼容 CURD 操作和复杂关联操作。

因此为了提升 Spark SQL 的查询速度，本文利用 Hbase+Phoenix 为 Spark SQL 创建索引，即将 Spark SQL 每条记录的文件路径、块位置、大小等元数据信息存入 Hbase，然后通过 Phoenix 建立关联表和二级索引。

3.2 4G 行业卡数据组织框架设计

3.2.1 业务场景分析

4G 行业卡数据是标准的结构化数据，其业务需求变化较大，通常需要全表扫描、逐行计算、多维度统计、多表关联等操作。根据其业务场景总体分为两类：即“基

于历史数据的多维度统计和业务关联”和“实时查询和随机存取业务”。本文所涉及的4G行业卡数据是从移动运营商的长期演进(Long Term Evolution, LTE)网络架构中提取出来的。LTE网络架构如图3.4所示,其中UE表示用户设备,MME表示移动管理实体,HSS表示用户归属签约服务器。本文重点关注的模块便是UE、HSS、MME及其信令数据s1-mme、s6a、s11等,需要从中提取相关的信令数据和业务数据,然后分别按照Spark SQL和Hbase的数据组织格式进行数据建模和导入。

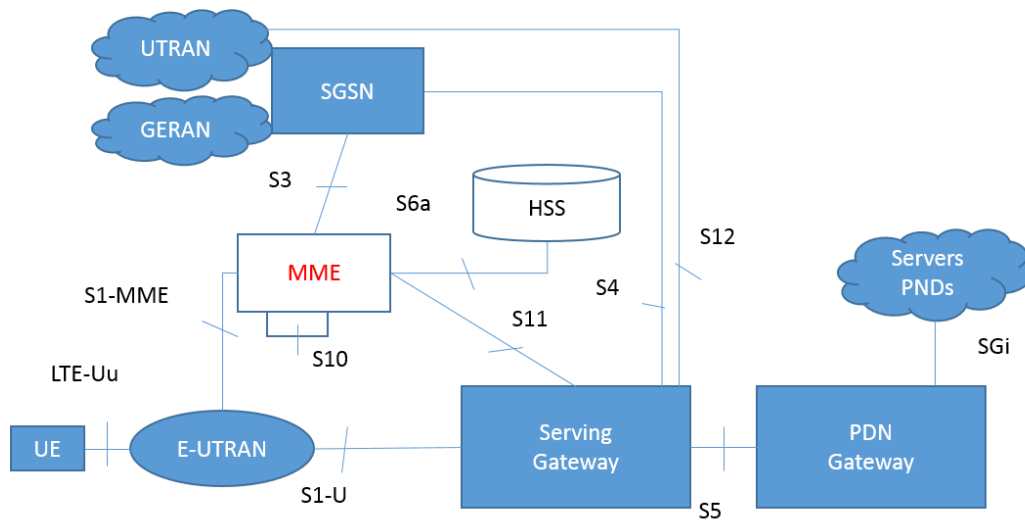


图 3.4 LTE 网络架构图

现有的技术方案主要单独采用Spark SQL/Hive进行离线数据分析,或单独采用Hbase通过代码处理业务逻辑。对于Spark SQL而言,其Spark SQL的操作粒度基于Partition文件,适合处理基于历史数据的多维度统计和业务关联,而Hbase基于LSM树,则适合处理实时查询和随机存取业务。因此目前的技术方案都存在一定的局限性,不能完全适应4G行业卡数据灵活多变的业务需求。具体的分析如下:

由于Spark SQL的操作粒度基于Partition文件,所以它的查询时间复杂度与Partition文件有关。假设一张表有 n 行记录,将其分割为 m 个Partition文件,每个Partition文件存储了 k 行数据,那么满足关系:

$$m = \left\lceil \frac{n}{k} \right\rceil \quad (3.3)$$

其中, n 、 m 、 k 都是可能变化的,当 n 不变时, m 与 k 成反比。首先进行 n 不变的分析:

设当一个文件到达 HDFS 默认的 128MB 时, 该文件最多能存储 k' 行数据(k' 因压缩算法的不同而变化), 即当 $k_{\max} = k'$ 时, $m = m_{\min}$ 。

如果要从 n 行数据中查找一条特定的记录, 设查找并行度为 m' (计算参照公式 (2.1)), 查找将进行 $\lceil m/m' \rceil$ 轮, 每轮同时从 m' 个文件中查找, 直到率先在某个文件中找到这条特定的记录则结束查找。因此在引入并行度 m' 时, 最坏情况下查询时间复杂度满足关系:

$$O(n) = O\left(\left\lceil \frac{m}{m'} \right\rceil * k\right) \quad (3.4)$$

当 $m' < m$, 则最坏需要进轮 $\lceil m/m' \rceil$ 查询, 时间复杂度为如公式 (3.4) 所示;

当 $m' = m$, 则只需要一轮查询完 m 个 Partition 文件, 时间复杂度为 $O(k)$;

当 $m' > m$, 只会使用 $m' = m$ 的并行度查询, 这时会出现资源浪费, 时间复杂度也为 $O(k)$ 。

不难看出采用 $m' = m$ 的并行度进行查询, 即查找并行度等于 Partition 文件个数时, 资源对于查询性能影响最小。如果始终能保证 $m' = m$ 的情况下, k 就成为了影响查询性能的因素:

当 $k=1$ 时, $m = m_{\max}$, 则时间复杂度为 $O(1)$ 。但当数据量很大时, 即当 $n \rightarrow \infty$, 也有 $m \rightarrow \infty$, 而集群资源, 也就是并行度 m' , 始终是有限的, 也就是说执行查询轮次 $\lceil m/m' \rceil$ 很大, 时间复杂度为 $O(\lceil m/m' \rceil)$, 则查询时间 t 正比于 $\lceil m/m' \rceil$;

当 $k=k'$ 时, $m = m_{\min}$, 集群资源能够满足并行地查找 m 个文件, 时间复杂度为 $O(k)$, 那么在 k 很大的情况下, 查询时间 t 正比于 k 。

上述分析是当数据 n 固定的情况下, 然而实际应用中常常出现的情况是, 该表的数据 n 会不断增长, 即使在 k 最大化的情况下, m 个数也会不断增长且远大于并行度 m' , 因此查询瓶颈在于 $O(\lceil m/m' \rceil)$ 。于是需要通过分区和合并小文件等手段来缩小 $\lceil m/m' \rceil$, 因此这种特点决定了 Spark SQL 的查询时间在秒级, 甚至于分钟级, 不能做到实时的统计分析。

与基于文件查询的 Spark SQL 不同, HBase 是基于单条记录的。它使用 RowKey 作为主键, 并借助 Phoenix 建立二级索引, 再加上 LSM 树搜索算法和 HRegion server 缓存, Hbase 实现了毫秒级的“实时查询和随机存取业务”。同样地假设一张表有 n

条记录,那么对应有 n 个 rowkey 且有序排列,如果划分为 m 个 Region,则每个 Region 存储 k 条记录。查询请求从 Zookeeper 开始经过 ROOT 表到 META 表就可以定位数据到所在的其中一个 Region,共经过三次请求,每次请求时间复杂度为 $O(1)$ 。而该 Region 里面有一部分数据缓存在内存中,多余的则溢写到磁盘中。

设 Region 存储 k 条数据平均分为 y 个数据块,每块存储 x 条记录,则满足关系: $x=k/y$ 或 $y=k/x$ 。其中内存里缓存 1 个数据块,磁盘中则有 $y-1$ 个数据块,每块的数据都是按照树结构进行组织,所有的数据块共同组成了一个大的树型结构。

如果查询命中缓存,则时间复杂度为 $O(\log x)$;

如果没有命中缓存,则需要继续查询其他数据块,找到数据所在数据块需要开销为: $O(\log(y-1))$,然后再到该数据块里查找数据,因此总的时间复杂度为:

$$O(n) = O(\log(k/x-1)) + O(\log x) \quad (3.5)$$

通过以上分析可知, Hbase 查询数据的时间开销优于 Spark SQL。由于 Hbase 适合简单的随机查询, Spark SQL 适合复杂的离线分析,因此本文将它们各自的优势结合到一起,即利用 Hbase 为 Spark SQL 构建索引,从而提高 Spark SQL 离线分析的速度。

3.2.2 Spark SQL 读写接口改进

虽然通过改进存储格式和压缩算法可以一定程度地提高 Spark SQL 的查询速度,如采用综合性能最优的 Parquet 文件格式可以大幅提升查询速度,但是受限于 Spark 的 DataFrameReader API 和 DataFrameWriter API, Parquet 文件格式并不能充分发挥其性能。虽然 Parquet 的综合性能最优,但是其读写速度严重限于 Spark 的封闭式 API,因此本文修改了 Parquet 接口的代码,重新定义了读写方式来提升其读写的速度。本文反编译了 \$SPARK_HOME/jars/目录下的部分 jar 包,分析了其中关于 Parquet 文件的读写的源码,然后通过继承和覆盖的方式改写了 Parquet Read API 和 Parquet Write API。主要思路为:首先利用反射技术动态的构造了表的 schema 信息,然后重写覆盖了 ParquetReader 和 ParquetReader API,最后通过多线程的方式调用重写的 API 从而实现了并行地读写 Parquet 文件。

考虑到 API 的统一性和可扩展性,本文对 Read API 和 Write API 的结构定义是一致的,它们公用反射模块 bean2Schema()和 reflectBean()。并且为了便于区分,本

文约定自定义类继承自原 API 类时加上前缀“My”，如 MyParquetReader 是继承自 ParquetReader，并沿用原来的继承关系。

本文改进的 Read API 主要步骤如下：

步骤 1. 自定义 MyParquetReader.class，继承自 ParquetReader.class，使用建造者模式(Builder)构造对象，用于创建 ReadSupport；

步骤 2. 自定义 MyParquetReader 的内部类 MyBuilder，继承自 ParquetReader.Builder；

步骤 3. 自定义 MyParquetReadSupport.class，继承自 ParquetReadSupport.class，用于接收和解析 schema；

步骤 4. 反射创建 schema: MessageType schema = bean2Schema(T.class)；

步骤 5. 创建 ParquetReader 的 handle，用 ParquetReader<String[]> pqReader = new MyBuilder(new Path(inPath),schema).build()；

步骤 6. 遍历源文件并把每一条原始记录转成 String 数组：String[] array = reflectBean(stu)；

步骤 7. 用 ParquetReader 的 handle 读取每一行：pqReader.read(array)。

本文改进的 Write API 主要步骤如下：

步骤 1. 自定义 MyParquetWriter.class，继承自 ParquetWriter.class，使用建造者模式(Builder)构造对象，用于创建 ReadSupport；

步骤 2. 自定义 MyParquetWriter 的内部类 MyBuilder，继承自 ParquetWriter.Builder；

步骤 3. 自定义 MyParquetWriteSupport.class，继承自 ParquetWriteSupport.class，用于接收和解析 schema；

步骤 4. 反射创建 schema: MessageType schema = bean2Schema(T.class)；

步骤 5. 创建 ParquetWriter 的 handle，用 ParquetWriter<String[]> pqWriter = new MyBuilder(new Path(outPath),schema).build()；

步骤 6. 遍历源文件并把每一条原始记录转成 String 数组：String[] array = reflectBean(stu)；

步骤 7. 用 ParquetWriter 的 handle 写入每一行到磁盘：pqWriter.write(array)。

其中，schema 存储在 MySQL 的元数据库中，该元数据库除了表的 schema 之

外, 还包含了路径、分区等信息。Spark 所需的 schema 也即是 Dataset<T>的 schema 类型为 StructType 或 Json 类型(将在后续实验中说明), 而 Parquet 所需的 schema 类型为 MessageType。利用 JavaBean 反射技术自动创建 MessageType 的关键代码封装在方法 bean2Schema()中, 其流程图如图 3.5 所示。

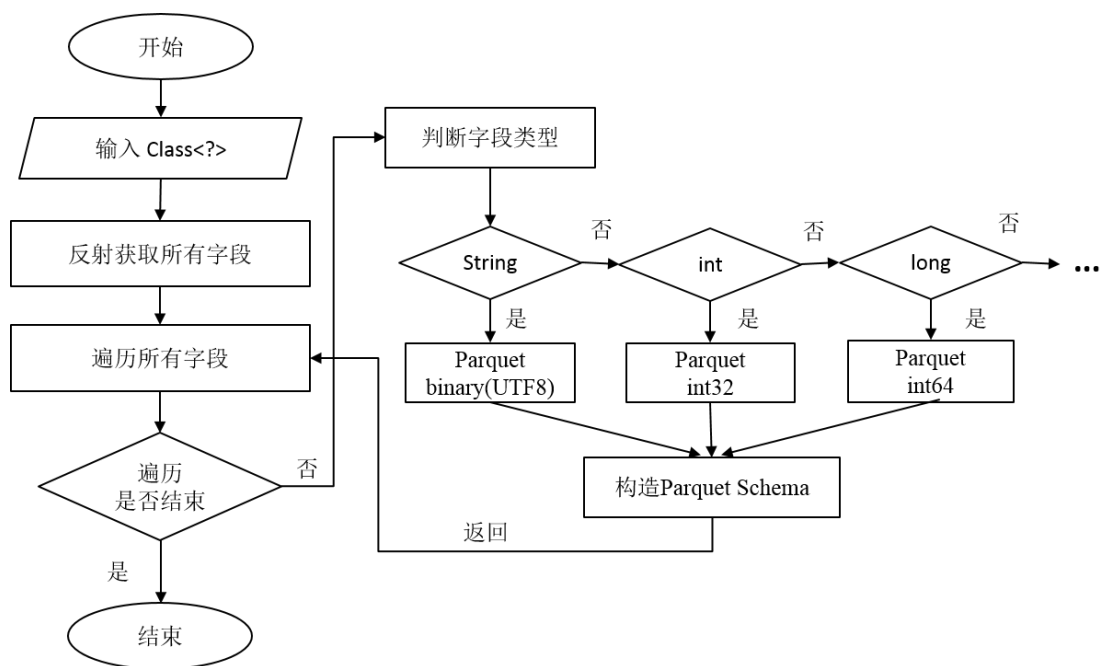


图 3.5 反射创建 MessageType schema

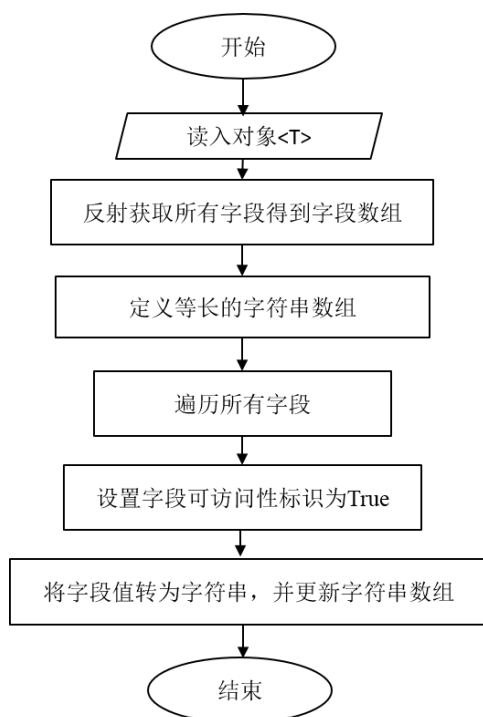


图 3.6 反射获取对象值

此外在遍历过程中需要将对象变成字符串数组，也可以转成 Parquet 提供的 Group 类型，但该类型受异常值干扰较大，经常出现不能识别 Parquet 的异常。因此为了编码的方便，本文选择字符串数组作为读写时的中转格式。将对象转成字符串的关键代码封装在方法 reflectBean() 中，其流程图如图 3.6 所示。

为了验证本文改进的 API 的性能，本文将其与 Spark 默认方法进行对比实验，实验结果如图 3.7 所示。实验表明无论是采用 Spark SQL 读写还是采用本文改进的方法进行读写，最终结果都是读数据比写数据快，这是由于写到磁盘的过程比读取复杂；其次本文改进的方法无论是写数据还是读数据都比 Spark SQL 默认 API 快，这是因为本文定义的数据结构都是轻量级的且缓存于队列中。此外多线程的方式由于线程切换开销大，所以速度比单线程慢，但是总体优于原来的方法。

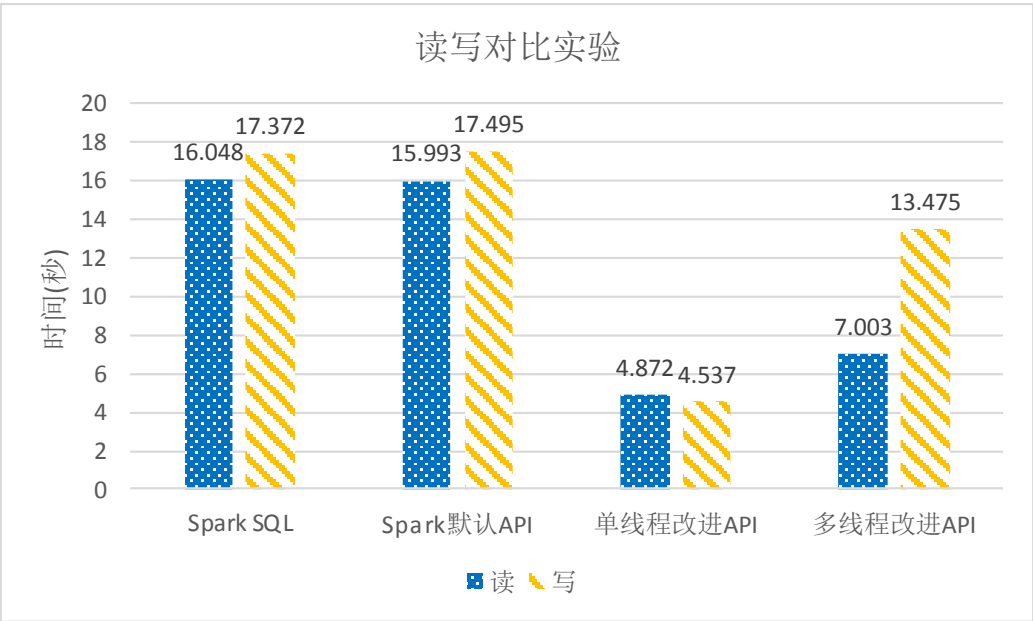


图 3.7 改进接口与默认 API 读写对比

由于本节所改进的读写 API 读写速度较快，所以将用于下一节的数据组织框架，可保证大规模的 4G 行业卡数据入库时的读写速度。

3.2.3 Spark SQL 与 Hbase 框架整合

为了能同时处理 4G 行业卡数据的两种电信业务场景：即“基于历史数据的多维度统计和关联”和“实时查询和随机存取业务”，本文将 Spark SQL 和 HBase 整合到一起，形成了一个整体的数据组织框架，其架构如图 3.8 所示。

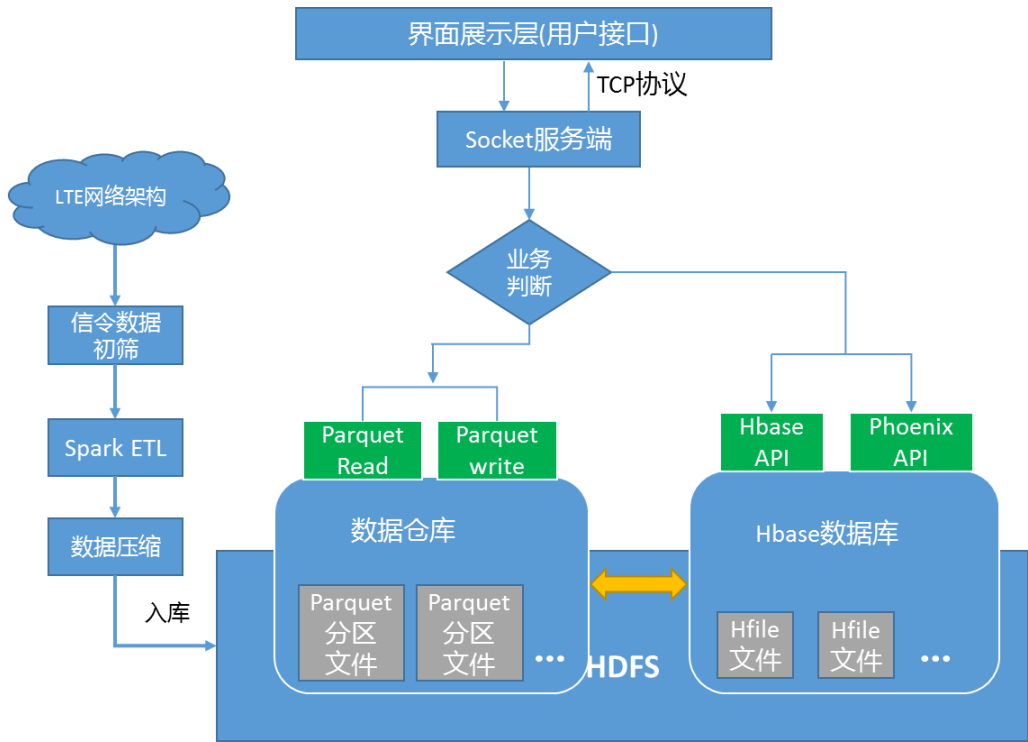


图 3.8 改进的数据组织框架

本文的数据组织框架首先用 Spark 从通信服务器上采集原始信令数据，经过 ETL 和压缩之后存入 HDFS 的数据仓库。写入数据仓库时需要将那些会被前台查询的数据与 Hbase 进行同步，具体表现为每一条待入库数据的文件路径，块位置，大小等信息存入 Hbase，然后通过 Phoenix 建立关联表和二级索引。当前台命令查询时便先通过 Phoenix 定位到数据在数据仓库中的位置信息，再加载对应的数据块文件获取特定的数据。

主要步骤如下：

- 步骤 1. 关联 Spark SQL 与 Hbase+Phoenix 的表结构，实现映射关系；
- 步骤 2. 遍历数据时提取 Spark SQL 块文件的 parquet 文件信息，返回该条数据对应的文件名，大小，位置信息；
- 步骤 3. 获取待写数据中的查询字段，并与 parquet 文件信息进行组合；
- 步骤 4. 利用上一节改进的读写 API 将数据和文件信息分别写入 Spark SQL 与 Hbase；
- 步骤 5. 利用 Phoenix 为多个查询字段建立二级索引，加快检索速度。

其中用户接口定义为：{'command':'hdp.check.SendResult','Time1':'2017/5/1 10:00:00','Time2':'2017/5/1 11:00:00','telnum':'13512345678'}，该接口负责请求服务器

端的查询结果，经过服务端的业务处理后返回 Json 数据。示例如下：

```
{  'Time1':'2017/5/1 10:24:48',
    'Time2':'2017/5/1 10:24:48',
    'IMSI':'460004512345678',
    'telnum': '13512345678',
    'faultlayer':'用户层',
    'faultCase':'用户状态异常',
    'conclusion':'此行卡用户在此查询时间段内欠费'
}
```

本文改进的数据组织方案主要是将 Hbase 随机查询的优点与 Spark SQL 的统计分析能力有机整合为一体，将 Spark SQL 的文件信息存储在 Hbase，并利用 Phoenix 加快检索速度。

3.3 实验及结果分析

3.3.1 实验环境

本文的实验环境是一个包含 5 个节点的集群，同时安装了 Hadoop-3.0.0、Spark-2.4.0、Hbase-2.1.1 和 Phoenix5.0.0。其中一台作为主控节点运行着 Namenode、Master、Hmaster 等守护进程，其余 4 台为从节点，运行着 DataNode、Worker、HregionServer 等守护节点。每个节点的配置均相同，其操作系统均为 Ubuntu 16.04 LTS，内存均为 6GB，CPU 均为 8 核处理器。

3.3.2 实验结果分析

为了验证改进后数据组织框架的性能，本文以 S6a 信令表为例进行说明。即首先从 LTE 网络的原始数据提取出信令表(见图 3.4)，并从中选取了 50 万行数据进行实验。本文首先通过 Spark 程序对源文件进行预处理，将文本格式的信令日志标准化为 S6a 表（数据已经过脱敏处理），如表 3.1 所示，该表共有 7 个字段，每个字段分别对应着不同的含义。

表 3.2 S6a 信令表的结构

序号	字段名	数据类型	源文件对应位置	示例
0	Time1	string	10	2017/5/1 20:56:08
1	Time2	string	11	2017/5/1 20:56:08
2	seat	int	19	0
3	IMSI	long	6	460079812341234
4	Telnum	long	8	13552521234
5	Type	int	9	6
6	Cause	int	13	2001

接着本文利用 Spark SQL 导入该表至数据仓库，则该表的 schema 信息如下：

```
{ "type": "struct",
  "fields": [
    { "name": "Time1", "type": "string", "nullable": true, "metadata": {} },
    { "name": "Time2", "type": "string", "nullable": true, "metadata": {} },
    { "name": "seat", "type": "integer", "nullable": true, "metadata": {} },
    { "name": "IMSI", "type": "long", "nullable": true, "metadata": {} },
    { "name": "Telnum", "type": "long", "nullable": true, "metadata": {} },
    { "name": "Type", "type": "integer", "nullable": true, "metadata": {} },
    { "name": "Cause", "type": "integer", "nullable": true, "metadata": {} }
  ]
}
```

将上述 Json 格式在 schema 信息转成 StructType，其中每一个字段的结构类型为 StructField，该结构有利于遍历、解析以及与其他格式(RDD、字符串、JavaBean 等)进行转换，StructType 结构如下：

```
StructType(
  StructField(Time1,StringType,true),
  StructField(Time2,StringType,true),
  StructField(seat,IntegerType,true),
  StructField(IMSI,LongType,true),
  StructField(Telnum,LongType,true),
  StructField(Type,IntegerType,true),
  StructField(Cause,IntegerType,true)
```


)

然后将该 schema 通过 3.2.2 节改进的 Parquet 接口转换成如 2.1.3 节所述的 schema, 其类型为 MessageType, 其目的是便于用改进的 Parquet 接口快速读写数据, Parquet 格式的 MessageType 如下:

```
message Pair {
  required binary Time1 (UTF8);
  required binary Time2 (UTF8);
  required int32 seat;
  required int64 IMSI ;
  required int64 Telnum;
  required int32 ProcedureType;
  required int32 Cause;
}
```

经过上述 schema 转换之后数据的组织方式为 Parquet+Snappy 文件格式, 部分实验数据如图 3.9 所示。

```
root@a1:~/syl# hdfs dfs -du -h /user/spark/warehouse/lz.db/tb_s6a
0      0      /user/spark/warehouse/lz.db/tb_s6a/ SUCCESS
2.2 M  6.5 M  /user/spark/warehouse/lz.db/tb_s6a/part-00000-ffe4a8fc-60e2-4fad-9e4f-6720cb98b0c4-c000.snappy.parquet
2.2 M  6.5 M  /user/spark/warehouse/lz.db/tb_s6a/part-00001-ffe4a8fc-60e2-4fad-9e4f-6720cb98b0c4-c000.snappy.parquet
2.2 M  6.5 M  /user/spark/warehouse/lz.db/tb_s6a/part-00002-ffe4a8fc-60e2-4fad-9e4f-6720cb98b0c4-c000.snappy.parquet
2.1 M  6.4 M  /user/spark/warehouse/lz.db/tb_s6a/part-00003-ffe4a8fc-60e2-4fad-9e4f-6720cb98b0c4-c000.snappy.parquet
2.2 M  6.5 M  /user/spark/warehouse/lz.db/tb_s6a/part-00004-ffe4a8fc-60e2-4fad-9e4f-6720cb98b0c4-c000.snappy.parquet
2.2 M  6.5 M  /user/spark/warehouse/lz.db/tb_s6a/part-00005-ffe4a8fc-60e2-4fad-9e4f-6720cb98b0c4-c000.snappy.parquet
2.2 M  6.5 M  /user/spark/warehouse/lz.db/tb_s6a/part-00006-ffe4a8fc-60e2-4fad-9e4f-6720cb98b0c4-c000.snappy.parquet
2.2 M  6.5 M  /user/spark/warehouse/lz.db/tb_s6a/part-00007-ffe4a8fc-60e2-4fad-9e4f-6720cb98b0c4-c000.snappy.parquet
```

图 3.9 部分实验数据展示

然后将该数据按照改进的数据组织框架进行入库, 入库的同时要与 Hbase 和 phoenix 进行关联。遍历该 Parquet 文件时将每一条库数据的文件信息(FileInfo:{文件路径, 块位置, 大小})存入 Hbase, 然后通过 Phoenix 建立关联表和二级索引。其中 FileInfo 通过 HdfsUtil 获取, 部分 FileInfo 实验截图见图 3.10。

```
FileInfo [path=/user/spark/warehouse/lz.db/tb_s6a/part-00000-ffe4a8fc-60e2-4fad-9e4f-6720cb98b0c4-c000.snappy.parquet, size=2.16 MB, location=a4]
FileInfo [path=/user/spark/warehouse/lz.db/tb_s6a/part-00001-ffe4a8fc-60e2-4fad-9e4f-6720cb98b0c4-c000.snappy.parquet, size=2.16 MB, location=a5]
FileInfo [path=/user/spark/warehouse/lz.db/tb_s6a/part-00002-ffe4a8fc-60e2-4fad-9e4f-6720cb98b0c4-c000.snappy.parquet, size=2.16 MB, location=a5]
FileInfo [path=/user/spark/warehouse/lz.db/tb_s6a/part-00003-ffe4a8fc-60e2-4fad-9e4f-6720cb98b0c4-c000.snappy.parquet, size=2.15 MB, location=a4]
FileInfo [path=/user/spark/warehouse/lz.db/tb_s6a/part-00004-ffe4a8fc-60e2-4fad-9e4f-6720cb98b0c4-c000.snappy.parquet, size=2.16 MB, location=a5]
FileInfo [path=/user/spark/warehouse/lz.db/tb_s6a/part-00005-ffe4a8fc-60e2-4fad-9e4f-6720cb98b0c4-c000.snappy.parquet, size=2.16 MB, location=a2]
FileInfo [path=/user/spark/warehouse/lz.db/tb_s6a/part-00006-ffe4a8fc-60e2-4fad-9e4f-6720cb98b0c4-c000.snappy.parquet, size=2.16 MB, location=a5]
FileInfo [path=/user/spark/warehouse/lz.db/tb_s6a/part-00007-ffe4a8fc-60e2-4fad-9e4f-6720cb98b0c4-c000.snappy.parquet, size=2.16 MB, location=a4]
FileInfo [path=/user/spark/warehouse/lz.db/tb_s6a/part-00008-ffe4a8fc-60e2-4fad-9e4f-6720cb98b0c4-c000.snappy.parquet, size=2.14 MB, location=a2]
```

图 3.10 FileInfo 数据展示

Hbase 的关联表 HB_S6A 主要有两个字段, 一个是连接条件 CONNKEY, 另外

一个是 FILEINFO。紧接着将每一条记录的 FileInfo 存入 FILEINFO 字段，将 IMSI、Time1、Time2 三个字段组成复合 key 存入 CONNKEY 字段。HB_S6A 数据示例如图 3.11 所示。

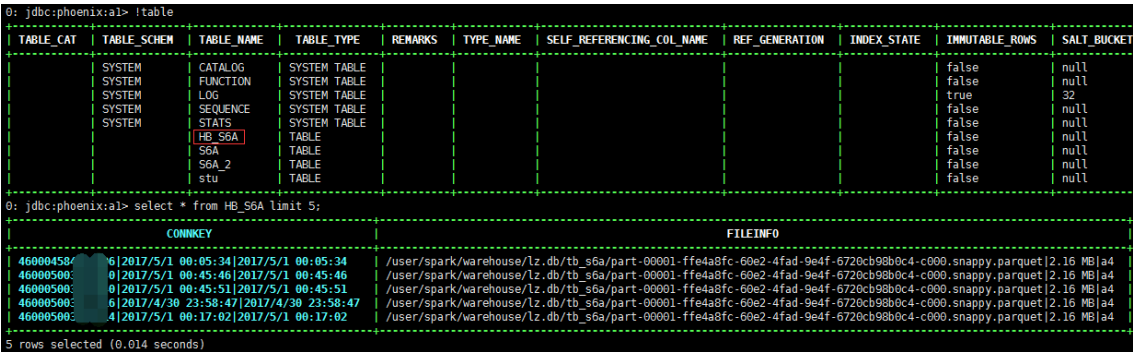


图 3.11 HB_S6A 数据展示

因为用户接口进行故障诊断时会向系统传入三个参数即 IMSI、Time1 和 Time2，其含义是向系统请求诊断某个 IMSI 在时间段[Time1, Time2]内的故障情况。当请求命令通过 Socket 传递到服务端时，便会进行业务判断，对于某些需要关联查询的故障类似便会先启动 Spark SQL 进行处理，对于某些需要随机查询的故障则会先进入 Hbase 关联表进行查询，即先查询 FileInfo，再定位到 Parquet 文件的位置信息，再通过 Parquet 接口加载该文件到内存，再按照条件进行查询。

本文针对改进前后的查询时间进行了实验对比分析，为了保证实验的准确性，实验进行了多次的重复操作，最终实验结果取平均值。实验结果如图 3.12 所示，改进后的方案在数据检索速度上有了明显的提升。实验比较了精确查找和模糊查询两种方式，改进之前查询速度相差不大，改进之后均有明显提升，且精确查找更快。

本章的数据组织框架使用了 Hbase 和 Phoenix 来定于 Parquet 文件信息，由于 Hbase 的随机查询在毫秒级(参照 3.2.1 的分析)，因此定位 Parquet 文件的一个 Partition 文件的时间开销也在毫秒级。接着再使用 3.2.2 节的 Parquet 接口加载数据，由于本章改进的该接口比 Spark 默认的接口快(见图 3.7)，因此去特定的节点上读取特定的一个 Partition 文件并查询其中符合条件的数据的时间开销最多为 $O(n)$ ，其中 n 表示一个 Partition 文件的记录条数。假设一个表有 k 个 Partition 文件，那么 Spark 默认的查询机制下时间开销为 $O(n*k)$ ，所以整合的数据组织框架将时间开销降低到了原来的 $\frac{1}{k}$ 。

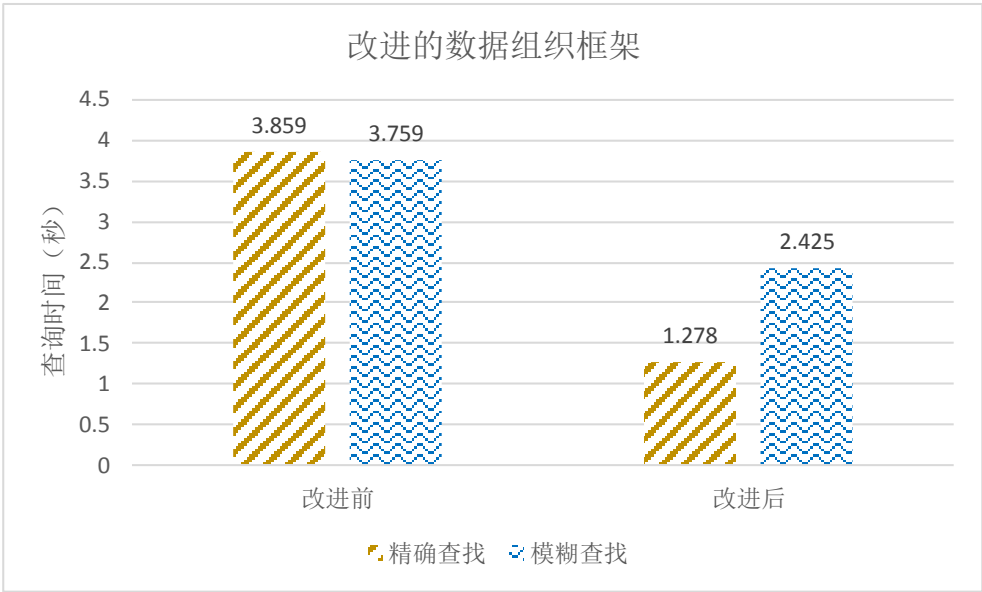


图 3.12 整合的数据组织框架对比实验

为了进一步的验证本文改进的数据组织框架对于 4G 行业卡数据的查询影响，本文将其与 Hive、Hive 索引和 Spark 分区进行了对比实验。由于 Hive 是目前主流的离线处理引擎，且 Spark SQL 与 Hive 绝大部分兼容，两者的 schema 信息完全一样，因此将 4G 行业卡数据迁移到 Hive 仓库中不需要额外的改动。它们的配置信息如表 3.3 所示，其中 Row Format Serde 指定了序列化和反序列化方式，IndexHandler 指定了 Hive 创建索引的方式。

表 3.3 横向对比实验配置信息

配置参数	参数值
Row Format Serde	org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe
InputFormat	org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat
OutputFormat	org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat
IndexHandler	org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler

实验基于 hiveserver2 进行了 Hive 离线查询，又通过 IndexHandler 构建了 Hive 索引进行对比，同时还与 Spark 分区这种粗粒度索引机制进行了对比，实验结果如图 3.13 所示。

本文改进的数据组织框架基于 Hbase+Phoenix 的索引机制查询时间最短，而 Hive 的查询时间则相对较长，这是由于 Hive 基于 MapReduce 引擎导致中间过程的读写次数过多。实验进一步表明 Hive 索引的效果较差，与没有索引的查询时间相差

不大。Hive 的索引机制本质上是将索引列和它在原表中的位置偏移量提取到一个新的表中存储，这个新的表便是索引表。使用索引查询时会先去索引表查找该字段的偏移量，再去原表检索数据。但是 Hive 构建索引的过程较长，中间过程的 HDFS Read 约为:6.874GB,HDFS Write 约为:12.0313GB,而原始数据经过压缩之后仅为98.6MB,相较之下本文的单线程或多线程 API 具有明显的优势。

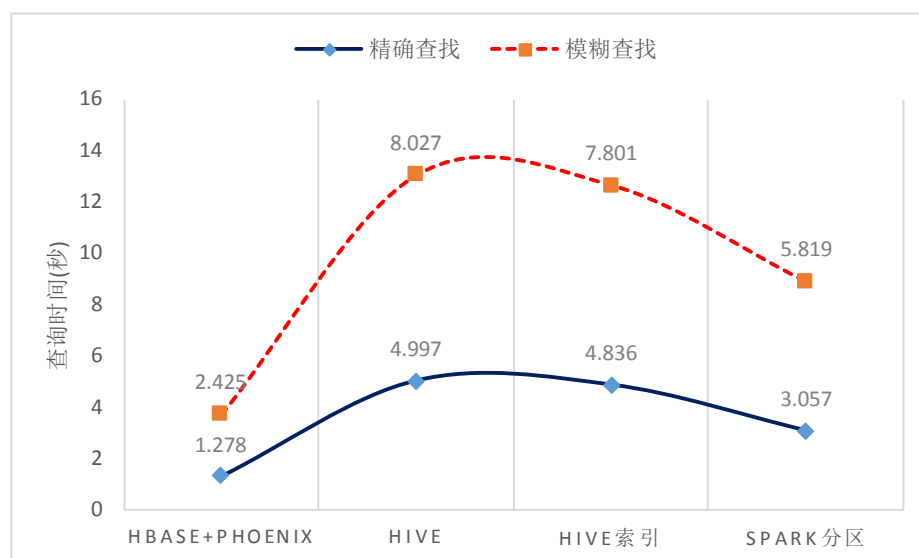


图 3.13 横向对比查询实验

此外，Spark 分区检索在模糊查询时可能会触发全表扫描，因为数据分布可能跨多个分区。如果检索数据分布集中于某个分区，则会大幅加快检索速度。综上所述，本文改进的数据组织框架在读取和查询数据时都具有一定的优势。

3.4 本章小结

大数据环境下不同的数据组织方式对于查询性能的影响十分明显，本章分别从 Spark SQL 和 Hbase 的数据组织方式角度进行了分析和测试。然后结合 4G 行业卡数据业务需求多变，通常需要全表扫描、逐行计算以及大量的多维度统计和多表关联等复杂操作的特点，提出了一个整合的数据组织方案，该方案充分考虑了数据存储格式、改进了读写和索引方式，使得 4G 行业卡数据的处理速度得到了大幅提升。

第4章 大表关联算法研究

4.1 Spark SQL 大表关联问题分析

4.1.1 Sort Merge Join 问题分析

Sort Merge Join 即排序合并连接, Spark SQL 中当左右两张表都很大时, SQL 执行计划便会选择 Sort Merge Join 算法。它与 Hash Join 的思路完全不同, 小表时可以使用 Broadcast Hash Join, 较大表时可以将其分而治之即 Shuffle Hash Join。Sort Merge Join 算法主要思想是用两个指针左右交替地探测数据, 但前提是数据是有序的。又因为数据是分布在 HDFS 的多个不同节点, 所以要先进行 Shuffle 操作, 即按照 Join Key 对数据进行重分区, 使左右两张表的数据分布大致相同; 其次分别对 Shuffle 后的数据进行局部的排序; 再并行地对有序的左右表进行合并操作, 即交替扫描两表并在遇到符合条件的记录时输出合并结果。

由此可以看出 Sort Merge Join 的主要开销在于 Shuffle 和 Sort 阶段, 该阶段本质上是将原有的 Partition 文件读出再重新写入新的文件, 即改变了数据原有的分布规律, 这个 Shuffle 过程即数据重分区。重分区时会出现如文件 1 的记录 A 被拉取到文件 2 中记录 B 的位置的情况, 如果文件 1 和文件 2 不在同一个节点上, 则会有额外的网络开销。因此 Shuffle 时磁盘和网络开销大, 耗时较长。此外重新分区后的数据只是中间结果, 为了加快检索速度会优先缓存在内存中(JVM 堆空间), 如果数据量过大会导致内存不足, 此时会引发频繁的 GC, 而 GC 执行时程序会暂停等待^[54], 因此频繁 GC 的开销也十分巨大。如果多轮 GC 之后内存依然不足, 则会抛出 OOM 错误导致程序崩溃。

OOM 问题主要是因为数据量增大导致 JVM 堆内存不足, 本文通过 JVM dump 文件发现 Spark 默认的 JVM 堆内存分布欠合理: Young Generation 默认分配大小为几百兆到 1GB 左右, 使用率为 30%-50%; Old Generation 默认大小也为几百兆到 1GB 左右, 但是使用率为 80%左右; 而 Metaspace 默认只有一百兆左右, 使用率为 98%左右。虽然可以通过 JVM 参数来适当降低年轻代内存, 提高老年代内存, 同时提高

元空间内存,但是这种调优通常只能采取失败重试策略,需要大量的参数测试和 GC 日志分析,效率低且成果不明显。

此外,Sort Merge Join 的 Shuffle 问题往往伴随着负载不均衡问题,如本文的 4G 行业卡数据呈动态增长的趋势,每间隔一段时间便会传输一个小文件到 HDFS 上,一方面是由于传输时过于集中分配到 HDFS 的某几个 DataNode,另一方面是由于文件本身内容的分布不均匀(峰值时期数据分布密集,闲时数据分布稀疏)。具体表现为:Spark 读取该表时文件分布密集的 DataNode 对应的 executor 过载,而文件稀疏的 executor 空载;处理峰值数据的 executor 耗时较长,处理闲时数据的 executor 空转;还会出现由于某个任务耗时较长(即慢任务)而拖延了整个应用程序。

4.1.2 分批 Join 策略

大表关联算法(Sort Merge Join)由于数据量远远超过物理硬件的限制,因此通常的做法便是分而治之。Join 的优化研究大致分为两类:一是将 Join 分解成多个并行小任务,另一类是优化 Join 的 SQL 执行计划,相对来说优化前者更简单快捷。Spark 虽然以内存计算见长,但它没有控制数据流量的机制,因此 Spark 的 OOM 也饱受诟病。所以本文首先建立一个监控模型来控制数据流量,再对 Join 进行切割和分批处理,使得每一批进入内存 Join 的数据刚好适应内存大小,从而避免频繁 GC。

数据进入内存后会进行一系列操作:如展开(Unroll)、序列化反序列化、按照 RDD 的数据结构重新组织等,这会导致内存中的数据量倍增。设数据在内存中的增长倍数为 ω ,一张表所占磁盘空间大小为 $F = \sum_{i=1}^j f_i$ (见公式(3.1)),那么进入内存之后大小则变为: $F * \omega$,也就是说大小为 F 的数据需要内存空间为 $F * \omega$ 。因此需要先根据采样计算出 ω 。

由于 Spark 对 Parquet 格式支持较好,本文将数据全部转换为 Parquet+Snappy 格式,并且按照某一字段分区存入 SparkSQL 数据仓库,那么平均每条记录所占空间大小为 $\partial = \frac{F}{n}$ (见公式(3.2))。由于 ∂ 并不是恒定不变的,这取决于文件的存储格式和字段数据以及字段类型,即字段少的表比字段多的表更省空间。因此需要预先采样分析,并以此计算出每张表准确的比率关系。

首先表的大小和数据分布是已知的,那么采样抽取若干行数据分别计算其在磁盘和内存中的大小,便可得到数据的增长倍数。为了保证数据的精确性,采样操作应该重复进行几次并取平均值,当数据量很大时,采样得到的误差可以忽略。

一张表会分布在多个节点,本文按照表的原始分布,并行地对每个节点进行分批 Join,则每个节点的实际数据大小为 $F = \sum_{i=1}^j f_i$ (来自公式(3.1))。同时为了保证 Partition 文件的个数为整数(即不进行二次文件分割),则每个节点的实际可用内存大小 M 为:

$$M = \min(M', \omega * \sum_{i=1}^p f_i) \quad (4.1)$$

其中, M' 表示每个节点可用于存储数据的内存大小(来自公式(2.2)), p 表示数据大小最接近 M' 时的 Partition 文件个数(整数), f_i 表示每个文件的实际大小,单位为 MB。又因为 F 远大于每个节点实际可用内存 M , 所以要进行分批 Join, 总批数 t 为:

$$t = \left\lceil \frac{F}{M} \right\rceil = \left\lceil \frac{\sum_{i=1}^j f_i}{\min\left((x * 2^{10} - 300) * \varepsilon, \omega * \sum_{i=1}^p f_i\right)} \right\rceil \quad (4.2)$$

公式(4.2)表明经过 t 轮的 Join 划分, 每轮应该导入最大数据量为 M , 即等于可用内存。由于 k 个节点并行读取, 且每个节点又有 y 个 CPU cores 可以多线程并行操作, 所以经过 t 轮完全能够处理完所有数据。此外为了保证每个节点的数据量 F 大致相同, 还应该对 HDFS 进行负载均衡操作。

按照上述的 Join 批次 t , 那么每轮最多可以导入的数据量为 M 。但这是基于平均情况的假设, 即假设每个节点是平等的, 且数据是负载均衡的。但实际情况中往往由于某些慢任务^[55]会拖累其他作业成为整个瓶颈, 比如某些数据在 Shuffle 时拉取数据耗时较长, 或者 Shuffle 时中间数据太多侵占了 Storage 内存等。所以前面提到的 Join 批次 t 并不一定是最优的, 只能作为程序初始化时的批次划分方案, 在运行过程中还需要根据实际情况动态调整批次, 即根据监控状态来控制数据流量并相应的修改 Join 批次。

4.2 内存监控模型设计

4.2.1 性能指标分析

针对上一节提出的 Sort Merge Join 问题(OOM、Shuffle、负载不均衡等), 资源监控有助于及时了解和排除异常问题。资源监控主要是监控程序运行时(Runtime)的状态(即程序在内存中的运行状态), Runtime 涉及到的参数多达上百个, 主要为内存、CPU、网络、GC 等信息。建立统一的内存监控模型便能一目了然地诊断问题所在, 以便在第一时间解决问题。当向集群提交一个 Spark 应用时会在端口(4040、4041 等)上启动 Web UI, 因此可以通过 JMX 提供的 REST API 来获取其各项指标的监控数据^[56], 即在程序运行期间每间隔一段时间就自动获取各项指标来动态监控程序运行状态。监控数据获取方式为:

`http://<server-url>:4040/api/v1/[app-id]/environment|executors|jobs|stages`

其中 environment 表示程序运行环境的数据, 可以从中获取应用程序的基本资源情况。如应用程序(app)的 ID, driver 进程的内存和 CPU 核数, executor 进程的内存和 CPU 核数, 以及作业的调度模式(FAIR 和 FIFO 两种)。本文重点关注的指标如表 4.1 所示。

表 4.1 environment 指标信息

指标名称	中文含义	数据示例
spark.app.id	所提交作业的 ID	app-20180707164529-0029
spark.cores.max	所申请的 Cpu 总核数	12
spark.scheduler.mode	作业调度模式	FIFO
spark.driver.cores	驱动节点的 Cpu 核数	1
spark.driver.memory	驱动节点的内存	2g
spark.executor.cores	每个执行节点的 Cpu 核数	6
spark.executor.memory	每个执行节点的内存	2g

executors 表示执行器(JVM)监控数据, 可以从指标中获取 driver 进程和每个执行器进程的基本情况。比如通过 maxMemory 和 memoryUsed 可以得到当前内存使用情况, 通过 totalDuration 和 totalGCTime 可以得到当前内存的 GC 状况, 如果

totalGCTime 过大，则说明内存不足，需要减小数据量，而通过 totalInputBytes 可以得到输入的数据量。此外对于上一节提到的采样数据也可以通过 memoryUsed 和 totalInputBytes 指标来计算数据在内存中的增长倍率。因此本文重点关注的指标如表 4.2 所示。

表 4.2 executors 指标信息

指标名称	中文含义	数据示例
id	所分配的执行器 ID	0
isActive	该执行器是否工作	true
maxMemory	最大可用内存	1048785715
memoryUsed	已使用内存	0
diskUsed	已溢出到磁盘	0
totalCores	总 CPU 核数	6
maxTasks	最大任务数	6
totalTasks	总分配任务数	2
completedTasks	已完成任务数	2
activeTasks	活跃任务数	0
failedTasks	失败任务数	0
totalDuration	执行时间	3846
totalGCTime	GC 时间	339
totalInputBytes	输入数据量	39440
totalShuffleRead	Shuffle 拉取数据量	0
totalShuffleWrite	Shuffle 输出数据量	0

Spark 将一个应用程序(application)根据 action 算子划分为若干个作业(job)，每个 job 再根据依赖关系划分为若干阶段(stage)，同样每个 stage 里包含一组并行任务(task)，其数量与 Partition 文件有关，其关系如图 4.1 所示。

因此 jobs 表示 application 的作业信息，比如通过 completionTime 和 submissionTime 的差值得到该作业的执行时间，也能通过 numCompletedTasks 得到当前完成的 task 数量。本文重点关注的指标如表 4.3 所示。

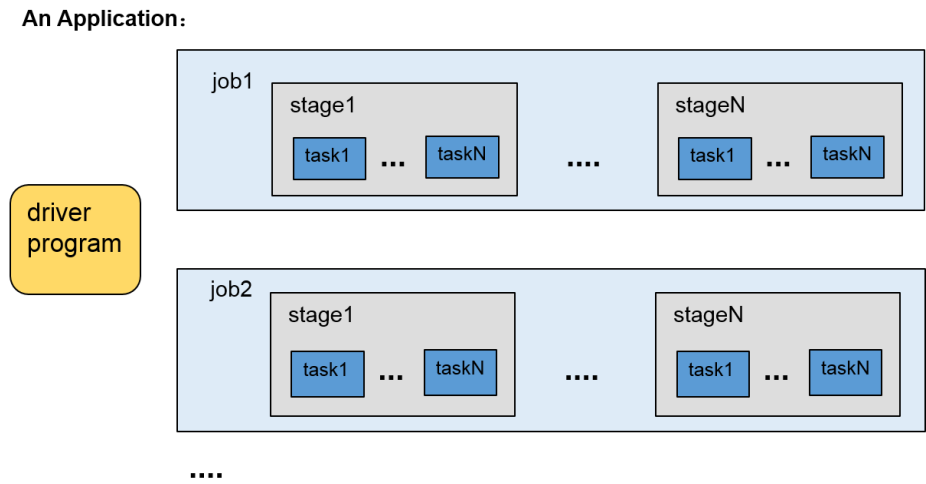


图 4.1 Spark 作业划分关系图

表 4.3 jobs 指标信息

指标名称	中文含义	数据示例
jobId	作业 ID	0
description	作业详情描述	select * from imsi_tel t1 join apn_list t2 on t1.APNName=t2.APN_Name
submissionTime	提交时间	2018-07-05T09:05:39.404GMT
completionTime	结束时间	2018-07-05T09:05:42.948GMT
stageIds	该作业所包含的阶段	[0 ,1]
numTasks	执行任务数	2
numActiveTasks	活跃任务数	0
numCompletedTasks	完成任务数	2
numSkippedTasks	跳过任务数	0
numFailedTasks	失败任务数	0

stages 表示 job 某阶段的信息，可以通过 firstTaskLaunchedTime 和 completionTime 判断当前任务是否在阻塞等待，还可以通过 memoryBytesSpilled 和 diskBytesSpilled 判断是否数据溢出，还能查看当前阶段的输入/输出数据量，数据条数，以及 Shuffle 数据量等，本文重点关注的指标如表 4.4 所示。

表 4.4 stages 指标信息

指标名称	中文含义	数据示例
stageId	阶段 ID	0
status	状态	COMPLETE
submissionTime	提交时间	2018-07-05T09:05:43.193GMT
firstTaskLaunchedTime	第一次执行时间	2018-07-05T09:05:43.194GMT
completionTime	结束时间	2018-07-05T09:05:43.627GMT
inputBytes	输入数据	62548
inputRecords	输入行数	950
outputBytes	输出数据	0
outputRecords	输出行数	0
shuffleReadBytes	Shuffle 拉取数据量	0
shuffleReadRecords	Shuffle 拉取行数	0
shuffleWriteBytes	Shuffle 输出数据量	0
shuffleWriteRecords	Shuffle 输出行数	0
memoryBytesSpilled	内存溢出	0
diskBytesSpilled	磁盘溢出	0

通过上述性能指标便能清晰地分析出当前的程序问题。如指标中包含了内存使用情况、CPU 使用情况以及 GC 情况，这些指标能帮助分析 OOM 异常和进行代码定位；指标中包含了 Shuffle 读写的数据量、任务的持续时长、内存和磁盘溢出字节数等指标可以帮助分析 Shuffle 开销问题；指标中包含了 executors 的任务分配和执行情况、成功和失败的任务数量及耗时等信息可以帮助分析负载不均衡问题。但是众多指标分析起来比较复杂，因此需要建立监控模型来降低分析难度，本文将在下一节进行阐述。

4.2.2 内存监控模型

为了更加清晰地知道内存 CPU 等资源消耗在何处，本文通过上一节提出的性能指标进行监控。首先将内存监控作为一个独立的模块来反馈程序执行状态，然后设置该模块每 10 秒更新一次状态，并通过指标的值来调控数据量。本文参照文献[57]提出的方法建立一个代价评估模型，由于涉及指标较多，首先将数据标准化为统一

的量纲，再建立综合评估模型。对于每一个 JVM 进程，本文重点关注内存使用率、GC 频率、CPU 利用率、任务完成率、Shuffle 频率，定义如下：

$$\text{内存使用率: } d = \frac{\text{memoryUsed}}{\text{maxMemory}} \quad (4.3)$$

$$\text{GC率: } e = \frac{\text{totalGCtime}}{\text{totalDuration}} \quad (4.4)$$

$$\text{CPU利用率: } f = \frac{\text{totalTasks}}{\max(\text{maxTasks}, \text{totalCores})} \quad (4.5)$$

$$\text{任务完成率: } g = \frac{\text{completedTasks}}{\text{totalTasks}} \quad (4.6)$$

$$\text{Shuffle率: } h = \frac{\max(\text{totalShuffleRead}, \text{totalShuffleWrite})}{\text{totalInputBytes}} \quad (4.7)$$

式(4.3)~(4.7)中的参数均来自于 4.2.1 节所述的性能指标。

因此每个 executor 具有一个性能集合 $set = \{d, e, f, g, h\}$ ，假设有 k 个 executor(JVM)，则上述指标构成一个性能矩阵 $A_{k \times 5}$ 作为监控模块的输入，经过评估模型的运算后，输出代价向量 $B = (q_1, q_2, \dots, q_k)^T$ 。因此可以根据 q_i 来调控每一个 executor 的数据流量，而系统的整体性能则为： $p = \frac{1}{k} \sum_{i=1}^k q_i$ 。

为了使 $q_i \in [0, 1]$ ，并且还能较好的反映作业运行时的内存情况，本文参照文献 [57] 中带经验系数的均方根评估模型进行建模，模型如下：

$$q_i = \sqrt{\frac{d' * d^2 + e' * e^2 + f' * f^2 + g' * g^2 + h' * h^2}{d' + e' + f' + g' + h'}} \quad (4.8)$$

其中， d', e', f', g', h' 分别表示内存使用率、GC 率、CPU 利用率、任务完成率、Shuffle 率的经验系数。由于每个指标的量级和重要程度都不一样，因此不同的经验系数产生的均方根有效值也不同，本文将在后续实验中进行分析。

4.2.3 资源分级及预警

通过上一节训练出的内存监控模型先是将数十个参数标准化成五大指标：内存使用率、GC 频率、CPU 利用率、任务完成率、Shuffle 频率，再训练成代价向量 $B = (q_1, q_2, \dots, q_k)^T$ 。其中每一个 q_i 的取值范围都为 $[0, 1]$ ，因此用 q_i 能够综合客观地来表示 Spark 集群第 i 个 executor 的资源评估值。

将 $[0,1]$ 的区间进行分段,即将集群的资源使用情况映射到该区间,那么每一小段便对应着一个实际的资源使用情况。分别判断每一个 q_i 所在的区间段,如果 $q_i \in [0,0.3)$ 则资源充足, $q_i \in [0.3,0.6)$ 则轻度告警, $q_i \in [0.6,0.8)$ 则中度告警, $q_i \in [0.8,1]$ 则严重告警。 q_i 落到该区间的某一段便能立即返回一个资源评估结果,即通过资源分级进行资源预警。

该监控模型通过观察者模式编程实现,每间隔一段时间便向集群请求一次性能指标进行模型训练,然后计算出当前资源的分级情况和预警评价,并发布给订阅了该模型的模块使用。订阅者模块接收到资源预警情况便立即调整本模块的数据流量以适应当前的资源使用情况,并将触发监控模型再次请求新的性能指标,从而实现实时的监视和调控。

4.3 分批 Join 算法设计

4.3.1 算法概述

本文改进的算法主要分为两部分,第一部是准备阶段,对数据采样统计和划分 Join 批次;第二部分是对分割后的数据进行 Join 操作。首先采样是为了计算数据在内存中的增长倍数,以此计算出所需的内存空间并完成初始数据分批的计划,并且要根据监控指标动态调整数据流量。第二部分使用基于 BloomFilter 和局部 Hash Join 算法来处理对分割后的数据,也就是对每一轮的数据采用优化的 Join 运算。

本文提出的根据内存监控实现分批次调控的 Join 算法,即将数据分批次导入内存计算,并根据内存大小动态地调整 Join 批次。首先计算数据规模和内存大小的关系来确定初始 Join 批次,其次是在 Join 过程中根据集群内存监控数据(CPU、内存、GC 等)来动态修改批次和每批处理数据量。该算法有两个显著特点:(1)考虑了数据存储格式对于 Join 性能的影响。(2)根据内存监控动态的调整数据流量大小来适应当前资源情况。

整体的算法框架流程如图 4.2 所示。

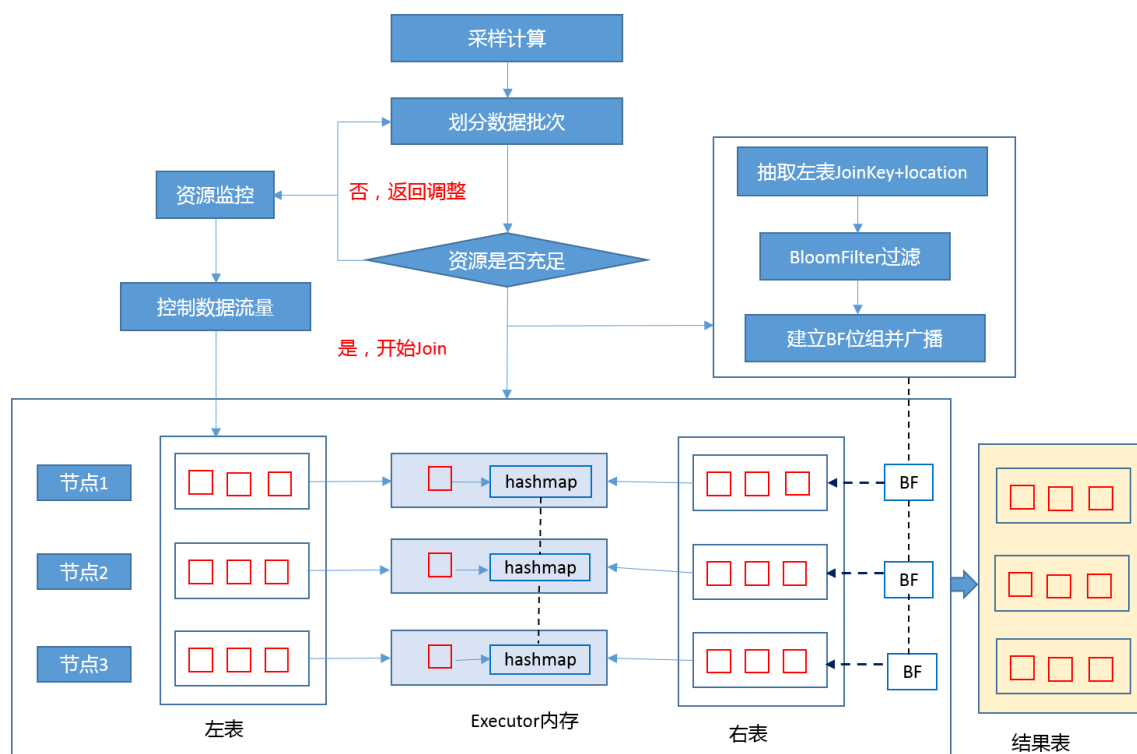


图 4.2 算法框架图

4.3.2 算法详细流程

步骤 1.统计 HDFS 上表的数据大小、分布情况

按照 3.1.2 节所述的方式，首先计算 HDFS 上左表的大小 m ，再根据表名路径获取该表的所有 Partition 文件，返回 $\text{List}\langle\text{FileInfo}\rangle$ ，其中 FileInfo 包含每个文件的{路径 path，大小 disksize，块位置 location}，并根据 location 对文件重新聚合分类，计算每个 node 上的 Partition 文件列表，即返回 $\text{Map}\langle\text{location}, \text{List}\langle\text{path}\rangle\rangle$ 。

步骤 2.左表随机采样，计算数据在内存中的增长倍数 ω

随机加载 $\text{List}\langle\text{FileInfo}\rangle$ 的一个文件，并统计此时内存的使用情况，返回 $\text{List}\langle\text{Executor}\rangle$ ，其中 Executor 包含了 4.2.1 节所述的性能指标如数据输入量 totalInputBytes 、已使用内存 memoryUsed 、和最大可使用内存 maxMemory 等。再根据文件的 location 找到对应的 Executor ，并计算数据增长倍数： $w=\text{memoryUsed}/\text{disksize}$ 。

步骤 3.按照 4.2.2 节所述启动监控模型计算可用内存 M ，并持续监控程序

单独使用一个线程运行监控模块，该模块使用观察者模型动态的监控集群资源变化情况，每间隔 10 秒就请求一次性能指标。首先按照 4.2.2 节所述的方法根据 Executor 的性能矩阵计算代价向量，再根据代价向量的值进行评级和资源使用情况告警：{safety, warn, serious, danger}，然后把此信息通过观察者模型反馈给订阅者，订阅模块就可以根据它的反馈来动态调整数据流量。

步骤 4.按照 4.1.2 节所述划分左表的 Join 批次和确定每批导入的数据量

根据步骤 1 的表大小 m 和步骤 2 的增长倍数 ω 计算该表所需的内存大小 $f=m*\omega$ ，再根据步骤 3 的可用 M 计算初始的 Join 批次 t ，且在运行过程中根据步骤 3 动态的更新剩余表大小 m 和剩余可用内存 M ，并据此重新计算 Join 批次 t 。

步骤 5.建立 Join 字段位组 JoinKey BloomFilter

用多线程的方式快速抽取左表 tb1 的 Join key，并加上它所在的块位置(location)为后缀，组成复合 key(Joinkey+location)用于建立 JoinKey BloomFilter 位组并广播到 Spark 集群中。其中 BloomFilter 选取 k 个质数并构建 k 个 hash 函数对每个 Join key 进行去重后压缩存储，既能节省存储空间和又能提高 key 的查询速度。

步骤 6.过滤右表并建立右表 Join HashMap

并行遍历右表 tb2，并从广播变量中获取步骤 5 的 JoinKey BloomFilter 来过滤右表记录：如果右表的 Joinkey+location 包含在 JoinKey BloomFilter 中，则该条记录是符合等值条件的记录，应将其加入 Join HashMap 中，即返回 Map<Joinkey+location, tb2_Row>。由于右表过滤后符合条件的数据量剧减，因此应将此 Map 广播到 Executor 节点。此时需要结合步骤 3 反馈的资源使用情况，如果资源充足则将该 Map 作为一个广播变量广播到 Spark 集群中；如果资源匮乏或该 Map 过大则将它按照 location 分组拆分后再分别广播 Spark 集群对应的 Executor。

步骤 7.分批遍历左表进行最终 Join

每个节点并行地以 Partition 文件为单位进行 Join，首先加载左表的 Partition 文件开始遍历，遍历过程中 hash 查找 Map<Joinkey+location, tb2_Row>：如果左表 key+location 构成的复合 key 包含在 Map 中，则进行最终 Join，即将该 key 的左右记录联合输出到结果表中。如此一趟下来，便完成了一个 Partition 文件的 Join，结果生成到一个新的 Partition 文件。假设集群有 k 个节点，每个节点有 y 个线程可以并行，

那么一轮 Join 后左表便有 $k*y$ 个 Partition 文件完成了 Join 操作。此时还需要结合步骤 3 反馈的资源使用情况调整 Join 的速度，以便 Executor 来得及 GC 和释放内存。

步骤 8.根据步骤 4 重新计算 Join 批次，并返回步骤 7 继续执行下一批数据，直到所有的数据 Join 完成。

4.3.3 算法开销分析

本文提出的 Join 优化算法主要分为两部分，第一部是准备阶段，包含了步骤 1、步骤 2、步骤 3、步骤 4；第二部分是对分割后的数据进行 Hash Join 操作，包含了步骤 5、步骤 6、步骤 7、步骤 8。

设左表 tb1 有 p 个 Partition 文件，有 m 行数据；右表 tb2 有 q 个 Partition 文件， n 行数据；Spark 集群有 k 个节点，每个节点 y 个 core，则算法每一步的主要开销为：

$t1$: 统计左表的 Partition 文件的 FileInfo{文件路径、大小、块位置}，时间开销为 $O(p)$ ， p 是一个比较小的常数；

$t2$: 随机采样读取一个 Partition 文件计算增长倍数，平均时间开销为 $O(m/p)$ ；

$t3+t4$: 计算 Executor 的内存大小和 Join 批次，时间开销为 $O(k)$ ， k 是一个比较小的常数；

$t5$: 建立 JoinKey BloomFilter 需要遍历左表，时间开销为 $O(m)$ ；

$t6$: 建立 Join HashMap 需要遍历右表，时间开销为 $O(n)$ ；

$t7+t8$: 这部分是 t 轮分割的 Join 过程，分批遍历左表进行局部的 Hash Join，局部 Hash Join 会遍历一个左表的 Partition 文件，平均时间开销为 $O(m/p)$ ， t 轮并行 Join 的平均时间开销为 $O(m/p)*t$ ， t 是动态变化的，但总体来说变化不会很大，因为 t 的变化只是为了保证不会发生 OOM。

因此算法的总体时间开销为：

$$T=t1+t2+t3+t4+t5+t6+t7+t8=O(p)+O(m/p)+O(k)+O(m)+O(n)+O(m/p)*t$$

其中第一部分的开销相较于其他 Join 算法属于额外的开销，但它是必要的准备工作，对于加快后续的 Join 过程有关键作用，且能保证程序不会发生 OOM 异常，因此第一部分的开销是必要的。又因为大表的行数 m 和 n 远远大于其他常数如 p 、 q 、 k 、 y ，因此主要的时间开销为：

$$T = O(m) + O(n) + (t + 1) * O(\frac{m}{p})$$

(4.9)

4.4 实验及结果分析

4.4.1 实验环境

本文的实验环境是基于 Hadoop-3.0.0 和 Spark-2.4.0 的大数据平台。该集群一共包含 5 个节点，其中一个作为主控节点，其余的作为从节点。每个节点物理环境均相同，操作系统为 Ubuntu 16.04 LTS，内存为 6GB，CPU 为 8 核处理器。Hadoop 作为基础数据平台，数据存储在 HDFS 上，Spark 集群为 standalone 模式，主控节点运行客户端驱动程序(driver)。

在向 Spark 集群提交一个 Application 时，本文向 Spark 集群申请了 4 个 executor 和 1 个 driver，共 5 个 JVM。其中每个 JVM 分配 2GB 内存，4 个 cpu 核心，并且为了尽可能地减少 GC 影响，又重定义了 JVM 堆中年轻代和老年代的比例。具体的实验参数如表 4.5 所示。

表 4.5 实验环境参数表

spark.driver.memory	2g
spark.driver.extraJavaOptions	-Xms2048m -XX:NewSize=1024m -XX:NewRatio=1 -XX:SurvivorRatio=6
spark.executor.memory	2g
spark.executor.extraJavaOptions	-Xms2048m -XX:NewSize=1024m -XX:NewRatio=1 -XX:SurvivorRatio=6
spark.cores.max	16
spark.executor.cores	4
spark.locality.wait	60s
spark.serializer	org.apache.Spark.serializer.KryoSerializer

4.4.2 实验结果分析

实验一（建立监控模型）：本实验根据 4.2 节的分析建立了监控模型以实现资源分级和预警的目标。其中监控接口定义为：hdp.spark.SendSparkInfo，即可通过命

令{'command': 'hdp.spark.SendSparkInfo'}请求该接口获取监控数据,通过一系列地处理和转换后返回集群总的概况数据如下:

```
{  "memory": "24.0GB"
  "completedAppsNum": 10
  "masterUrl": "spark://a1:7077"
  "runningAppsNum": 1
  "coresNum": 32
  "aliveWorkersNum": 4
  "coresusedNum": 6
  "status": "ALIVE"
  "memoryused": "2.0GB"
  "listWorkers":
  [
    {"workerId": " worker-20190101115035-10.10.10.24-45443"
    "address": " 10.10.10.24:45443"
    "memory": "6.0GB"
    "coresNum": 8
    "coresusedNum": 0
    "state": "ALIVE"
    "memoryused": "0MB"
    }
    ....
  ]
  "listCompletedApps": [...]
  "listRunningApps": [...]
}
```

以上即是实验设计的接口数据,本文由于篇幅原因省略了部分数据,仅将接口字段的设计说明展示如表 4.6 所示。

表 4.6 接口字段设计说明

Json 字段名	中文含义
masterUrl	主控节点地址
aliveWorkersNum	活跃工作节点数量
coresNum	Cpu 核总数
coresusedNum	Cpu 核已使用数量
memory	集群内存总量

表 4.6 接口字段设计说明（续）

Json 字段名	中文含义	
memoryused	已使用内存	
runningAppsNum	正在运行的应用数量	
completedAppsNum	已完成的应用数量	
status	集群状态	
listWorkers 工作节点列表	workerId	工作节点 ID
	address	地址
	state	状态
	coresNum	节点 Cpu 核数
	coresusedNum	Cpu 核已使用数量
	memory	节点内存总量
	memoryused	已使用内存
listRunningApps 运行的应用列表	appID	应用进程 ID
	appName	应用名称
	cores	使用 cpu 核数
	memoryPerNode	每个工作节点使用内存
	submittedTime	应用提交时间
	user	提交用户
	state	应用状态(运行/等待)
	duration	应用执行时间
listCompletedApps 已完成的应用列表	appID	应用进程 ID
	appName	应用名称
	cores	使用 cpu 核数
	memoryPerNode	每个工作节点使用内存
	submittedTime	应用提交时间
	user	提交用户
	state	应用状态(完成/杀死)
	duration	应用执行时间

接着按照 4.2 节所述对各项指标进行训练和建模，实验向 Spark 集群申请了 4 个 executor 和 1 个 driver(a1 节点)，每个 executor 默认分配 1G 内存和 4 cores。因此实验过程如图 4.3 所示，该图显示了监控模型运行时某一时刻的截图，性能矩阵和代价向量也如图 4.3 所示。

```

监控线程-----当前系统时间: 2019-02-28 09:27:33
活跃的appname->appid 的 map----- {SparkSQL::10.10.10.21=app-20190228092621-0038, Mem---test=app-20190228092646-0039}
活跃的appID->port 的 map----- {app-20190228092646-0039=4041, app-20190228092621-0038=4040}
09:27:33 watch.mode.Executor:46:Executor id= driver --- al:34858
09:27:33 watch.mode.Executor:46:Executor id= 3 --- 10.10.10.22:40021
09:27:33 watch.mode.Executor:46:Executor id= 2 --- 10.10.10.23:36320
09:27:33 watch.mode.Executor:46:Executor id= 1 --- 10.10.10.24:36315
09:27:33 watch.mode.Executor:46:Executor id= 0 --- 10.10.10.25:40696
-----list.size()=5
性能矩阵:
al:34858: d=0.000, e=0.000, f=0.000, g=0.000, h=0.000
10.10.10.22:40021: d=0.000, e=0.000, f=0.000, g=0.000, h=0.000
10.10.10.23:36320: d=0.000, e=0.000, f=0.000, g=0.000, h=0.000
10.10.10.24:36315: d=0.000, e=0.000, f=0.000, g=0.000, h=0.000
10.10.10.25:40696: d=0.000, e=0.000, f=0.000, g=0.000, h=0.000
代价向量:
al:34858= 0.0 资源充足! needMem=0.0, restMem=0.0, 批次t=0
10.10.10.22:40021= 0.0 资源充足! needMem=0.0, restMem=0.0, 批次t=0
10.10.10.23:36320= 0.0 资源充足! needMem=0.0, restMem=0.0, 批次t=0
10.10.10.24:36315= 0.0 资源充足! needMem=0.0, restMem=0.0, 批次t=0
10.10.10.25:40696= 0.0 资源充足! needMem=0.0, restMem=0.0, 批次t=0
totalMem=618.50 MB maxMem=1.78 GB usedMem=469.08 MB freeMem=149.42 MB
PS Scavenge: 15 times 0.181 s
PS MarkSweep: 7 times 0.577 s

```

图 4.3 监控模型运行截图

本实验将监控模块(watchMonitor)单独启动为一个线程来运行性能监控评估模型(CostMode)，并采用观察者模式以松耦合的状态进行模块间的通信，即在 CostMode 中将资源预警情况发送到订阅模块 watchDeal。如 CostMode 发送消息为 Event.sendEvent(Msg.warn)，则能在 watchDeal(implements EventHandler)中接收到 Msg. warn。

监控模型中主要有 5 个指标参数(见公式(4.8))即：内存使用率、GC 率、CPU 利用率、任务完成率、Shuffle 率，每个指标都有一个经验系数。由于每个指标的量级和重要程度都不一样，因此不同的经验系数产生的均方根有效值也不同。本文中内存使用率最重要，GC 频率其次，再则是 shuffle 频率，而 CPU 利用率和任务完成率则相对不那么重要，因此本文与标准的均方根模型进行多次对比实验。标准的均方根模型定义为：

$$q_i = \sqrt{\frac{d^2 + e^2 + f^2 + g^2 + h^2}{5}} \quad (4.10)$$

该模型是将公式(4.8)的经验系数全部置为 1，即不带经验参数，其余变量与公式(4.8)一样。

实验结果如表 4.7 所示。实验表明本文选取的经验系数[10000,100,1,0.1,10]能比较合理的反映内存使用情况。然后对每一个 q_i 分别判断其所在的区间，如果 $q_i \in [0, 0.3)$ 则资源充足， $q_i \in [0.3, 0.6)$ 则轻度告警， $q_i \in [0.6, 0.8)$ 则中度告警，

$q_i \in [0.8, 1]$ 则严重告警。如表 4.7 所示, 内存使用率为 0.56 时, 综合指标为 0.057, 则表示轻度告警。

表 4.7 经验系数影响实验

经验系数	参数说明	综合指标	待测性能矩阵
[1,1,1,1,1]	标准的均方根模型, 每个指标同等重要	q1=0.465	{ 0.091 , 0.097, 0.250, 1.000, 0.000} { 0.000 , 0.005, 0.750, 1.000, 0.018} { 0.560 , 0.200, 0.000, 1.000, 0.000}
		q2=0.559	
		q3=0.515	
[10000,100,1,0.1,10]	内存使用率指标最重要, 且它与综合指标相近	q1=0.09	
		q2=0.008	
		q3=0.557	
[5,4,3,2,1]	各指标的重要程度递减, 但递减梯度较小	q1=0.389	
		q2=0.496	
		q3=0.49	

因此该监控模型可以用于 Join 数据监控以及分批处理, 同时也能适用于其他操作的资源监控和数据限流。

实验二(执行分批 Join): 本实验所用的测试数据为 LTE 网络中的信令数据表: IM 和 S6a, 将 IM 表中的 StartTime、EndTime 和 IMSI 与 S6a 表中的 StartTime、EndTime 和 IMSI 进行连接, 计算出每个 IMSI 号码在某一个时间段[StartTime, EndTime]的连接信息。实验选取数据量较大的 IM 表作为左表, 选取 S6a 作为右表, 且在大、小两种不同的数据规模下进行。小表实验中 IM 表有 100 万行数据, S6a 表有 70 万行数据; 大表实验中 IM 表有 2000 万行数据, S6a 表有 1400 万行数据。S6a 表结构同 3.3.2 节实验所述(见表 3.2), IM 表结构如表 4.8 所示, 该表有 18 个字段, 每个字段对应着不同的含义。数据均提取自 LTE 网络架构(已脱敏处理), 对源文件进行预处理后转换为标准的结构数据。

表 4.8 IM 信令表的结构

序号	字段名	数据类型	源文件对应位置	示例
0	Time1	string	20	2017/5/1 20:56:08
1	Time2	string	21	2017/5/1 20:56:08
2	seat	int	17	0
3	IMSI	long	6	460079812341234

表 4.8 IM 信令表的结构 (续)

序号	字段名	数据类型	源文件对应位置	示例
4	Telnum	long	8	13552521234
5	APN	string	18	cmnet.apnName.name
6	uIp	string	27	10.155.89.71
7	sIP	string	31	111.30.210.153
8	UpLink	int	44	1396
9	DownLink	int	45	7804
10	UPacket	int	46	10
11	DPacket	int	47	6
12	UDisorder	int	48	0
13	DDisorder	int	49	0
14	URetrans	int	50	0
15	DRetrans	int	51	0
16	ResponseDelay	int	52	33
17	ConfirmDelay	int	53	26

接着用 Spark SQL 将数据存入 HDFS，并转为 Parquet+Snappy 格式，构成的 IM 表 Partition 文件如图 4.4 所示，构成的 S6a 表 Partition 文件如图 4.5 所示。按照前文所述 Partition 文件是 Spark SQL 处理的粒度，也决定了并行度。本实验中的 4G 行业卡数据是基于第三章提出的数据组织框架进行组织的，因此数据的加载、处理和写入速度都得到了极大的提升。

```

root@al:~# hdfs dfs -du -h /user/spark/warehouse/lc.db/j_im
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00000-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00001-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00002-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00003-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00004-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00005-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00006-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00007-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00008-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00009-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00010-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00011-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00012-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00013-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00014-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00015-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00016-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00017-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00018-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet
6.4 M 19.3 M /user/spark/warehouse/lc.db/j_im/part-00019-c0295f35-80b7-4e38-9b7b-595dd4b5b751-c000.snappy.parquet

```

图 4.4 IM 表数据

```
root@al:~# hdfs dfs -du -h /user/spark/warehouse/lc.db/j_s6a
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00000-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00001-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00002-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00003-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00004-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00005-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00006-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00007-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00008-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00009-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00010-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00011-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00012-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00013-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00014-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00015-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00016-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00017-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00018-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
4.3 M   12.8 M   /user/spark/warehouse/lc.db/j_s6a/part-00019-d087fabc-a033-4da7-9c41-7d656f47804d-c000.snappy.parquet
```

图 4.5 S6a 表数据

接着启动监控模型检测性能指标，并按照算法流程进行数据切割和分批 Join，每轮实验重复进行多次再取平均值，实验结果如图 4.6 所示。从图中的时间对比可以看出分批 Join 算法在小表、大表的情况下运行时间均要优于 Spark 默认的 Join。分批Join算法的额外时间开销在数据采样和计算批次，而默认Join实际为Sort Merge Join，其主要开销在排序。分批 Join 算法则避免了排序开销，总体来说时间比默认Join 快。

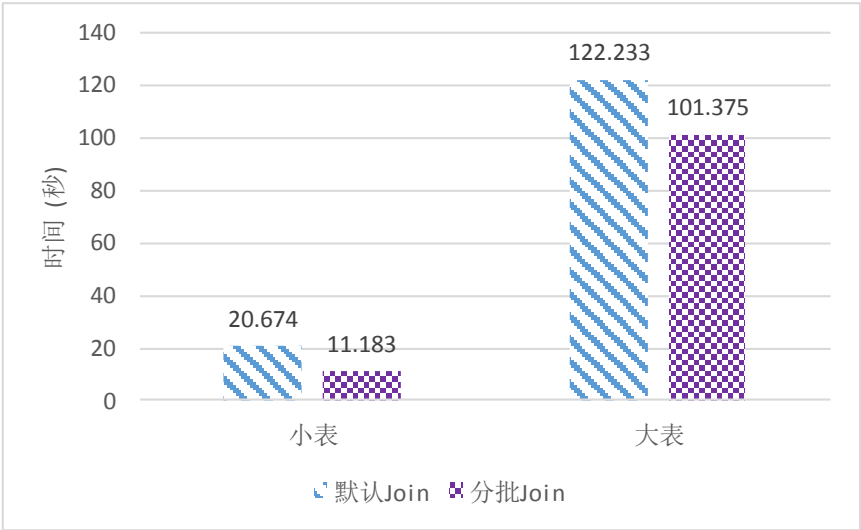


图 4.6 两种 Join 算法时间对比图

除了时间对比外，本文还深入分析了两种算法过程中的综合指标数据(其计算过程参照 4.2.2 节)，实验结果如图 4.7 所示。从图 4.7 的综合指标可以看出默认 Join 算法受数据分布影响较大，即当数据倾斜时各个 executor 负载不均衡，表现为某些 executor 负载特别重、指标特别高，其他 executor 负载很小，指标特别低。而分批

Join 算法则不受数据分布的影响，各个 executor 表现比较均衡，其指标也表现为相对一致。

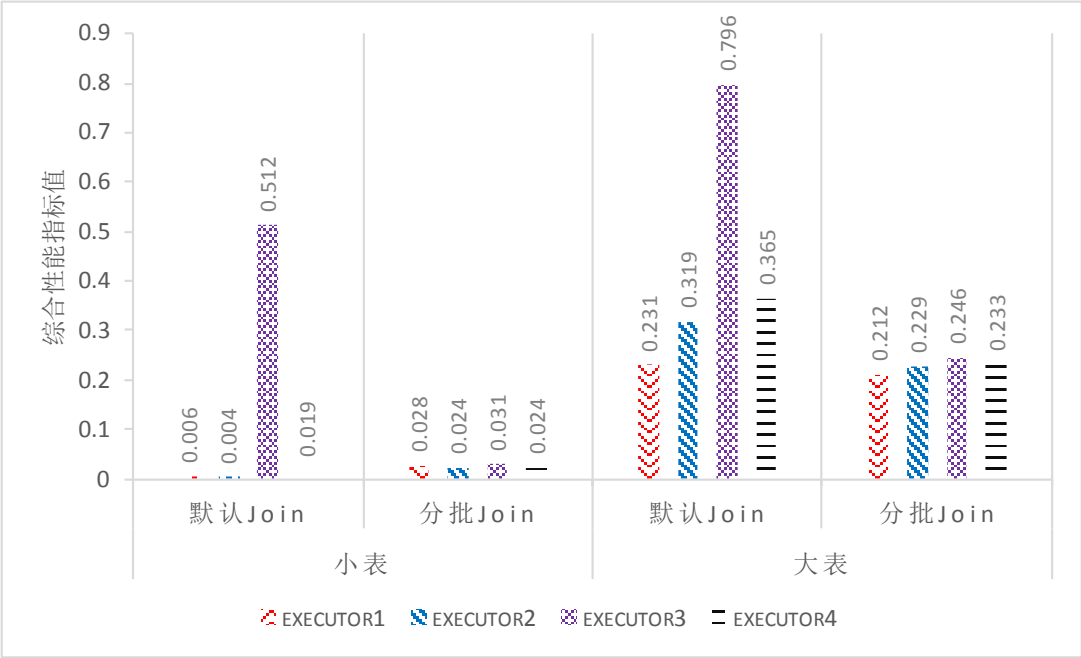


图 4.7 综合指标对比图

4.5 本章小结

本章首先分析了 Spark 大表关联算法(Sort Merge Join)存在的问题，接着又分析了内存监控的各项参数以建立监控模型和实现资源分级以及预警。随后根据监控模型来控制数据和进行分批 Join：即首先对数据采样和划分 Join 批次；然后启用监控模块动态控制数据流量；接着使用 BloomFilter 过滤非 Join key，生成 HashMap 并广播；最后循环进行分批 Join 操作，每轮 Join 都会根据 HashMap 调入对应的文件进行局部 Hash Join。分批 Join 从一定程度上缓解了内存不足的问题，也降低了数据倾斜导致的负载不均衡影响，总体运行时间优于默认的 Join。

第5章 工作总结和展望

5.1 论文工作总结

以4G行业卡数据为代表的通信大数据呈爆发式的增长,使用Hadoop、Spark、Hbase等分布式框架处理4G行业卡数据能显著地提高计算能力,但是也有一些局限性。如基于Hadoop的解决方案通常采用数据仓库或者NoSQL来存储数据,但是由于其分布式实现的特殊性,它对OLAP和OLTP的支持却不如关系型数据库;再比如Spark框架本身的一些局限性也限制了整个处理过程,对于Spark性能的调优会耗费大量的精力。

本文正是基于这样的背景展开了研究工作,主要研究了Spark的数据组织方式、查询机制等,改进了Spark的Parquet文件读写接口,并将其与Hbase和Phoenix框架进行整合以提高Spark SQL的查询速度。其次本文还深入研究了Spark的运行机制、内存模型和参数管理等,在分析了Spark的不足之处后提出了一个内存监控模型,用于实时地检测集群资源和进行资源分级、预警。最后将该监控模型用于改进Spark大表关联算法,即通过监控数据来控制流量,并分批次进行Join操作,从而避免了OOM、数据倾斜等问题。因此本文的主要贡献在于:

(1) 本文综合考虑了压缩存储格式、索引和NoSQL特征,将Spark SQL和Hbase+Phoenix充分结合起来存储4G行业卡数据。主要是将Spark SQL的文件信息存储在Hbase,再利用Phoenix加快Spark的检索速度,还采用了Parquet文件格式压缩和存储数据,并通过多线程和反射技术提高了其读写速度;

(2) 针对Spark的Sort Merge Join算法经常出现的OOM、GC严重、Shuffle开销大、耗时严重等问题,本文主要从数据文件、内存使用、分批处理、运行状态监控四个方面进行了分析,结合Parquet+Snappy的数据组织方式,对Spark Join进行了优化,提出了一种基于内存监控和分批处理的Join算法。主要表现为建立内存监控模型来预警资源使用情况,并据此动态调整Spark Join的轮次和每轮的数据量。

5.2 工作展望

本文所做的研究工作虽然在一定程度上提高了数据处理的速度，但是还存在着一下不足，本文考虑在下一步研究中进行改进。主要有以下几个方面：

(1) 本文的研究主要是基于 4G 行业卡数据，在应用场景上具有一定的局限性。本文考虑将其处理架构抽象为更加通用的结构，如果换成其他的结构化数据，不需要大的结构变动也能处理。

(2) 本文的监控模型仅考虑了常用的重点指标，且各个指标的权重参数最后被模型弱化了，本文考虑下一步完善改模型。将更多的指标纳入考虑范围，重新训练更为全面和综合的监控模型。

(3) 本文的 Join 算法仅是对内存进行监控，再进行分批处理，主要思想是控制数据流量。本文考虑下一步改进 Spark 的内存源码，从内存模型本身上改进使其能承受住高并发和数据峰值压力。

参考文献

- [1] Ghazi M R, Gangodkar D. Hadoop, MapReduce and HDFS: A developers perspective[J]. Procedia Computer Science, 2015, 48: 45-50.
- [2] Hashem I A T, Anuar N B, Gani A, et al. MapReduce: Review and open challenges[J]. Scientometrics, 2016, 109(1): 389-422.
- [3] Naheman W, Wei Jianxin. Review of NoSQL databases and performance testing on HBase[C]// Proceedings 2013 International Conference on Mechatronic Sciences, Electric Engineering and Computer. Shengyang: IEEE, 2013: 2304-2309.
- [4] Salloum S, Dautov R, Chen Xiaojun, et al. Big data analytics on Apache Spark[J]. International Journal of Data Science and Analytics, 2016, 1(3-4): 145-164.
- [5] Francis N, K S K. Data processing for big data applications using Hadoop framework[J]. International Journal of Advanced Research in Computer and Communication Engineering, 2015, 4(3): 177-180.
- [6] Armbrust M, Xin R S, Lian C, et al. Spark SQL: Relational data processing in Spark[C]// Proceedings of the 2015 ACM SIGMOD international conference on management of data. Melbourne: ACM, 2015: 1383-1394.
- [7] 梁远铭. 基于 Spark 的联机分析处理的研究[D]. 武汉: 华中科技大学, 2015.
- [8] Song Jie, Guo Chaopeng, Wang Zhi, et al. HaoLap: A Hadoop based OLAP system for big data[J]. Journal of Systems and Software, 2015, 102:167-181.
- [9] Lawrence R. Integration and virtualization of relational SQL and NoSQL systems including MySQL and MongoDB[C]// International Conference on Computational Science & Computational Intelligence. Las Vegas: IEEE, 2014: 285-290.
- [10] Gadiraju K K, Verma M, Davis K C, et al. Benchmarking performance for migrating a relational application to a parallel implementation[J]. Future Generation Computer Systems, 2016, 63: 148-156.
- [11] Chang Baorong, Tsai H F, Wang Yo-Ai, et al. Resilient distributed computing platforms for big data analysis using Spark and Hadoop[C]// International Conference on Applied System Innovation. Okinawa: IEEE, 2016: 1-4.
- [12] Guo Chenghao, Wu Zhigang, He Zhengying, et al. An adaptive data Partitioning scheme for accelerating exploratory Spark SQL queries[C]// International Conference on Database Systems for Advanced Applications. Cham: Springer, 2017: 114-128.

- [13] 周亮, 李格非, 邵伟鹏, 等. 基于 Spark 的时态查询扩展与时态索引优化研究[J]. 计算机工程, 2017, 43(7): 22-28.
- [14] 王亚玲, 刘越, 洪建光, 等. 基于 Spark/Shark 的电力用采大数据 OLAP 分析系统[J]. 中国科学技术大学学报, 2016(1): 66-75.
- [15] Siddiqua A, Karim A, Gani A. Big data storage technologies: a survey[J]. Frontiers of Information Technology & Electronic Engineering, 2017, 18(8): 1040-1070.
- [16] 王文贤, 陈兴蜀, 王海舟, 等. 一种基于 Solr 的 HBase 海量数据二级索引方案[J]. 信息网络安全, 2017(8): 39-44.
- [17] 朱明, 王志瑞. 基于 Hbase 的大数据查询优化[J]. 智能计算机与应用, 2017, 7(4): 59-61.
- [18] Gugnani S, Lu Xiaoyi, Qi Houliang, et al. Characterizing and accelerating indexing techniques on distributed ordered tables[C]// 2017 IEEE International Conference on Big Data (Big Data). Boston: IEEE, 2017: 173-182.
- [19] 袁兆争, 邵秀丽, 闫凯境, 等. 基于 SQL 的 Hbase 查询的设计与实现[J]. 计算机与现代化, 2017, 7: 20-26.
- [20] 刘容辰, 周明强, 皮兴杰, 等. 基于 Spark 的大数据统计中等值连接问题的优化[J]. 现代计算机, 2017, 12: 3-6.
- [21] 孙文隽, 李建中. 排序合并 Join 算法的新结果[J]. 软件学报, 1999, 10(3): 264-269.
- [22] 郑晓薇, 马琳. 基于 Hadoop 集群的多表并行关联算法及应用[J]. 微型机与应用, 2013, 32(4): 91-93.
- [23] 邓亚丹, 景宁, 熊伟. 多核处理器中基于 Radix-Join 的嵌套循环连接优化[J]. 计算机研究与发展, 2010, 47(6): 1079-1087.
- [24] 钱招明, 王雷, 余晟隽, 等. 分布式系统中 Semi-Join 算法的实现[J]. 华东师范大学学报 (自然科学版), 2016, 2016(5): 75-80.
- [25] 高泽, 李常宝, 杨淙钧, 等. 基于 MapReduce 模式的多表联查算法[J]. 现代电子技术, 2015(14): 81-84.
- [26] Rattanaopas K, Kaewkeerat S, Chuchuen Y. A comparison of ORC-Compress performance with big data workload on virtualization[J]. Applied Mechanics & Materials, 2016, 855: 153-158.

- [27] Li Xiaopeng, Zhou Wenli. Performance comparison of Hive, Impala and Spark SQL[C]// International Conference on Intelligent Human-Machine Systems and Cybernetics. Hangzhou: IEEE, 2015: 418-423.
- [28] 王华进, 黎建辉, 沈志宏, 等. 基于 ORC 元数据的 Hive Join 查询 Reducer 负载均衡方法[J]. 计算机科学, 2018, 45(3):158-164.
- [29] 熊安萍, 夏玉冲, 杨方方. 一种 Spark 集群下的 shuffle 优化机制[J]. 计算机工程与应用, 2018, 54(4): 72-76.
- [30] 侯伟凡, 樊玮, 张宇翔. 改进的 Spark Shuffle 内存分配算法[J]. 计算机应用, 2017, 37(12): 3401-3405.
- [31] Tang Zhuo, Zhang Xiangshen, Li Kenli, et al. An intermediate data placement algorithm for load balancing in Spark computing environment[J]. Future Generation Computer Systems, 2018, 78: 287-301.
- [32] Rao P S, Porter G. Is memory disaggregation feasible? A case study with Spark SQL[C]//Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems. Santa Clara: ACM, 2016: 75-80.
- [33] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing[C]// Usenix Conference on Networked Systems Design and Implementation. San Jose: USENIX Association, 2012: 2-15.
- [34] Armbrust M, Das T, Davidson A, et al. Scaling Spark in the real world: performance and usability[J]. Proceedings of the Vldb Endowment, 2015, 8(12): 1840-1843.
- [35] Shi Weiwei, Zhu Yongxin, Huang Tian, et al. An integrated data preprocessing framework based on Apache Spark for fault diagnosis of power grid equipment[J]. Journal of Signal Processing Systems, 2017, 86(2-3): 221-236.
- [36] Huang Chaoqiang, Yang Shuqiang, Tang Jianchao, et al. RDDShare: Reusing results of Spark RDD[C]// IEEE International Conference on Data Science in Cyberspace. Changsha: IEEE, 2016: 370-375.
- [37] 陈康, 王彬, 冯琳. Spark 计算引擎的数据对象缓存优化研究[J]. 中兴通讯技术, 2016, 22(2): 23-27.
- [38] 孟红涛, 余松平, 刘芳, 等. Spark 内存管理及缓存策略研究[J]. 计算机科学, 2017, 44(6): 31-35.
- [39] Gounaris A, Torres J. A methodology for Spark parameter tuning[J]. Big data research, 2018, 11: 22-32.

- [40] Xu Lijie, Dou Wensheng, Zhu Feng, et al. Experience report: A characteristic study on out of memory errors in distributed data-parallel applications[C]// 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). Gaithersbury: IEEE, 2015: 518-529.
- [41] 陈侨安, 李峰, 曹越, 等. 基于运行数据分析的 Spark 任务参数优化[J]. 计算机工程与科学, 2016, 38(1): 11-19.
- [42] Saraswati S, Chatterjee S, Ramachandra R. Steal-A-GC: Framework to trigger GC during idle periods in distributed systems[C]//2016 IEEE 23rd International Conference on High Performance Computing. Hyderabad: IEEE, 2016: 392-400.
- [43] 吴雯祺. Spark 性能数据收集分析系统的设计与实现[D]. 哈尔滨: 哈尔滨工业大学, 2015.
- [44] 皮艾迪, 喻剑, 周笑波. 基于学习的容器环境 Spark 性能监控与分析[J]. 计算机应用, 2017, 12: 248-253.
- [45] Zaharia M, Xin R S, Wendell P, et al. Apache spark: A unified engine for big data processing[J]. Communications of the ACM, 2016, 59(11): 56-65.
- [46] Li Sanmiao, Li Longshu. Performance analysis of four methods for handling small files in Hadoop[J]. Computer Engineering & Applications, 2016, 52(9): 44-49.
- [47] Saavedra M Z N, Yu W E. A Comparison between Text, Parquet, and PCAP Formats for use in distributed network flow analysis on Hadoop[J]. Journal of Advances in Computer Networks, 2017, 5(2): 59-64.
- [48] Plase D , Niedrite L , Taranovs R . Accelerating data queries on Hadoop framework by using compact data formats[C]// 2016 IEEE 4th Workshop on Advances in Information, Electronic and Electrical Engineering. Vilnius: IEEE, 2016: 1-7.
- [49] Melnik S, Gubarev A, Long J J, et al. Dremel: interactive analysis of web-scale datasets[J]. Communications of the ACM, 2011, 54(6): 114-123.
- [50] Liao Yingti, Zhou Jiazheng, Lu ChiaHung, et al. Data adapter for querying and transformation between SQL and NoSQL database[J]. Future Generation Computer Systems, 2016, 65: 111-121.
- [51] Lu Liang. Apache Spark 内存管理详解[EB/OL]. 北京: IBM developerWorks, 2017-3-28 [2019-1-13]. <https://www.ibm.com/developerworks/cn/analytics/library/ba-cn-apache-spark-memory-management/index.html>.

-
- [52] Apache Spark. Viewing Spark properties[EB/OL]. Apache Software Foundation, 2018-11-02 [2019-2-25].
<http://spark.apache.org/docs/2.4.0/configuration.html#available-properties>.
- [53] Xu Shilei, Wang Lei, Hu Huiqi, et al. Parallel Join based on distributed system OceanBase[J]. Journal of East China Normal University, 2017, 5: 1-10.
- [54] Ganesan K, Chen Y M, Pan X. Scaling Java Virtual Machine on a many-core system[C]// 2014 International Symposium on Integrated Circuits. Singapore: IEEE, 2014: 336-339.
- [55] Guo Yanfei, Rao Jia, Jiang Changjun, et al. Moving Hadoop into the cloud with flexible slot management and speculative execution[J]. IEEE Transactions on Parallel & Distributed Systems, 2017, 28(3): 798-812.
- [56] Apache Spark. Monitoring and instrumentation[EB/OL]. Apache Software Foundation, 2018-11-02 [2019-2-25].
<https://spark.apache.org/docs/latest/monitoring.html>.
- [57] 王欢, 李红辉, 张骏温. 改进 K-means 聚类的云任务调度算法[J]. 计算机与现代化, 2017, 2: 1-5.

致谢

转眼间已经研三了，三年的研究生学习生涯即将结束，三年来我不断地提升和充实自己，这首先要感谢学校为我们提供的良好学习环境和 Learning 资源。在重庆邮电大学的三年里我亲身感受到了同学们热烈好学的人文气氛，感受到了老师们兢兢业业的学术氛围，感受到了社会各界对母校的认可与好评。我自己学习了到很多专业知识，专业技能得到了极大的提升。通过与老师和同学们的交流，我的学术能力和写作水平也得到了提升，思考问题和认知世界的方式、角度、层次等都得到了一定的提升。这里要特别感谢学校、感谢老师和同学们的帮助和支持，希望重邮越办越好，希望老师和同学们取得丰硕的成果！

我要特别感谢我的导师程克非教授，感谢他的谆谆教导。从考研结束后联系导师时第一次见到程老师，他就给我留下了深刻的印象。在我看来，他是一个专业素养极高的导师，同时还不失谦和幽默的态度，显得十分温文尔雅，因此我十分敬佩程老师的人格魅力。程老师耐心地给我指导专业知识和项目开发，使得我通过这些专业知识和项目经验提升了自己的编程技术和实践经验。程老师还不厌其烦地给我指导论文中的错误和不足，使得我论文写作的能力也得到了极大的提升。因此我衷心的感谢程老师给了我们丰富的学习资源和悉心的指导。

其次要感谢实验室的各位伙伴们以及室友和同学们，感谢大家在学习和生活上的互相帮助和理解。平时大家一起讨论学术问题、攻克技术难题、分享学习经验和各种信息资讯，这使我们大家都融入在一个浓烈的人文氛围中。我还要感谢贵州力创科技有限公司提供的数据和平台，实习期间同事们对我的生活和工作都提供了极大的帮助和支持。此外还要感谢家人和朋友对我学习的支持和理解，你们都是我进步的后盾。

最后，非常感谢评审专家们的辛苦工作以及对本论文提出的宝贵意见。

攻读硕士学位期间从事的科研工作及取得的成果

参与科研项目:

- [1] 4G 行业应用卡综合故障定位分析平台, 企业横向, 2017.04-2018.11.
- [2] 大数据能力开放平台, 企业横向, 2017.11-2018.03.
- [3] XDR 信令数据集中性能核查平台, 企业横向, 2017.12-2018.05.

发表及完成论文:

- [1] Cheng Kefei, **Luo Zhao**, Zhou Ke, et al. A Spark Join Algorithm Based on Memory Monitoring and Batch Processing[C]// 2018 9th IEEE International Conference on Software Engineering and Service Science. Beijing: IEEE, 2018: 1096-1103.
- [2] 程克非, **罗昭**. 大数据环境下数据组织方式对于查询性能的影响[J]. 重庆邮电大学学报(自然科学版). 已投.
- [3] 程克非, 邓先均, 周科, **罗昭**, 等. 基于微博多维度及综合权值的热点话题检测模型[J]. 重庆邮电大学学报(自然科学版). 已录用.
- [4] Kefei Cheng, Xudong Chen, Ke Zhou, Xianjun Deng, **Zhao Luo**. Research of Spark SQL query optimization based on massive small files on HDFS[C]// The 2018 International Conference on High Performance Computing, Cloud Storage, Big Data, and Internet of Things. Xian: Atlantis Press, 2019: 180-190.

获奖:

- [1] **罗昭**, 邓先均, 吕媛媛. 重庆市高校数据库应用程序设计大赛, 一等奖(省部级). 2018.
- [2] **罗昭**, 邓先均, 吕媛媛. 重庆市高校数据库应用程序设计大赛, 新技术应用奖(省部级). 2018.