基于Spark SQL的分布式全文检索框架的设计与实现

崔光范[1,2] 许利杰[2] 刘 杰[2] 叶 丹[2] 钟 华[2]
(中国科学院大学 北京 100049)[1] (中国科学院软件研究所 北京 100049)[2]

摘 要 随着信息化的深入,大数据在各个领域产生了巨大的价值,海量数据的存储和快速分析成为新的挑战。传统的关系型数据库由于性能、扩展性的不足以及价格昂贵等方面的缺点,难以满足大数据的存储和分析需求。Spark SQL是基于大数据处理框架Spark的数据分析工具,目前已支持TPC-DS基准,成为大数据背景下传统数据仓库的替代解决方案。全文检索作为一种文本搜索的有效方式,能够与一般的查询操作结合使用,提供更加丰富的查询和分析操作。目前,Spark SQL仅支持简单的查询操作,不支持全文检索。为了满足传统业务迁移和现有业务的使用需求,提出了分布式全文检索框架,涵盖了SQL文法、SQL翻译转换框架、全文检索并行化、检索优化4个模块,并在Spark SQL上进行了实现。实验结果表明相比于传统的数据库,在两种检索优化策略下,该框架的索引构建时间、查询时间分别减少到传统数据库的0.6%/0.5%和1%/10%,索引存储量减少为传统数据库的55.0%。

关键词 Spark SQL,全文检索,翻译转换框架,检索并行化,检索优化
中图法分类号 TP391.3 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2018.09.016

Design and Implementation of Distributed Full-text Search Framework Based on Spark SQL

CUI Guang-fan[1,2] XU Li-jie[2] LIU Jie[2] YE Dan[2] ZHONG Hua[2]
(University of Chinese Academy of Sciences, Beijing 100049, China)[1]
(Institute of Software, Chinese Academy of Sciences, Beijing 100049, China)[2]

Abstract With the development of information technology, big data has generated great value in various fields. Huge data storage and rapid analysis have become new challenges. The traditional relational database is difficult to meet the needs of big data storage and analysis because of its shortcomings in terms of performance, scalability and high cost. Spark SQL is a data analysis tool based on Spark, which is a big data processing framework. Spark SQL currently supports the TPC-DS benchmark and has become an alternative solution to the traditional data warehouse under the background of big data. Full-text search, as a kind of effective method of text search, can be used in combination with general query operation to provide richer queries and analysis operations. Spark SQL doesn't support full-text search now. In order to meet the needs of traditional business migration and existing business, this paper proposed a Spark SQL distributed text retrieval framework, covering the design and implementation of 4 modules including SQL grammar, SQL translation framework, full-text search parallelism and search optimization. The results of experiment show that, under the two search optimization strategies, index construction time and query time of this framework are reduced to 0.6%/0.5% and 1%/10% respectively compared with the traditional database, and index storage volume is reduced to 55.0%.

Keywords Spark SQL, Full-text search, Translation framework, Search parallelism, Search optimization

1 引言

随着云计算、物联网等技术的发展,以及以博客、社交网络、基于位置的服务LBS为代表的应用模式的出现[1-2],数据的种类和规模正以前所未有的速度增长,大数据中蕴含的宝贵价值成为人们存储和处理大数据的驱动力[3]。数百TB甚至数十至数百PB规模的行业或企业的大数据以及数据的复杂性已远远超出了现有传统的计算技术和信息系统的处理能力,因此,寻求有效的大数据处理和分析技术已经成为现实世界的迫切需求。

传统的关系型数据库管理技术经过40多年的发展,在扩展性方面遇到了巨大的障碍,无法胜任海量数据的分析任务。以谷歌MapReduce[4]为代表的非关系型数据的处理和分析技术及其社区开源实现Hadoop[5]以良好的扩展性、并行性、容错性,成为了大数据处理的标准之一。基于Hadoop的数据仓库工具Hive[6]提供了更高层的SQL操作,支持数据的

到稿日期:2017-10-11 返修日期:2018-01-10 本文受北京市科技重大项目(D171100003417002)资助。
崔光范(1991-),男,硕士生,主要研究方向为分布式计算;许利杰(1987-),男,博士,助理研究员,CCF会员,主要研究方向为大数据系统,E-mail:xulijie@iscas.ac.cn(通信作者);刘 杰(1982-),男,博士,副研究员,CCF会员,主要研究方向为大数据挖掘分析;叶 丹(1971-),女,博士,研究员,CCF高级会员,主要研究方向为大数据挖掘分析;钟 华(1971-),男,博士,研究员,CCF会员,主要研究方向为分布式计算、软件工程。

提取、转化和加载(ETL)，避免了用户编写 MapReduce 程序带来的复杂性。由于 Hadoop 频繁写磁盘、缺乏作业的规划、迭代处理能力差等缺点，UC Berkeley AMPLab 提出了类 MapReduce 通用并行处理框架 Spark[7]，有效弥补了 Hadoop 在迭代计算和交互性方面的不足。其中，Spark SQL[8] 作为 Spark 软件栈中的一员，在大数据分析、机器学习、深度学习等方面发挥着工具和桥梁作用。Spark SQL 兼容 Hive，拥有比 Hive 更好的性能，目前已支持 TPC-DS 基准，是大数据背景下优良的数据仓库解决方案之一，因此，研究方向是 Spark SQL 的功能完备性以及扩展性。

在关系型数据库中，全文检索是衡量数据库易用性和功能完备性的重要指标。全文检索是通过将关键词和存储的文档数据进行匹配，找到关联度高的若干文档的信息检索技术。在众多关系型数据库中，如 MySQL，SQL Server，都已具备全文检索功能。

然而，Spark SQL 作为传统数据仓库的替代系统，不支持全文检索的 SQL 语句及其并行化。现有分布式全文检索引擎(如 Solr 和 ElasticSearch)虽然提供了 Hive 和 Spark 的连接器，但是仍不支持全文检索 SQL 文法，无法满足边查询边计算的需求，部署的复杂性和学习成本使其难以使用。

为了满足传统业务迁移以及现有业务对于检索的需求，本文设计和实现了 Spark SQL 分布式全文检索框架。本文的主要贡献如下：

1)查询语言到检索模型的流程翻译转换，包括全文检索 SQL 文法以及将 SQL 语句翻译为执行引擎并行任务的方法。

2)提出了全文检索任务的并行化方法，包括索引构建和查询并行化。

3)提出了两种检索优化方案。两种不同的方案分别侧重性能优化和存储优化，每种方案包括索引存储和原表数据还原两部分。针对存储优化场景，提出了时间复杂度为 $O(n)$ 的查询结果与原表数据连接算法。

4)使用大规模数据集对框架的性能、扩展性进行了评测，并与传统关系型数据库进行了比较。实验表明，相比于传统关系型数据库，在两种检索优化策略下，该框架索引的构建时间、查询时间分别是传统数据库的 0.6%/0.5%，1%/10%，索引存储量减少为传统数据库的 55.0%。

本文第 2 节介绍了大数据查询语言及框架、SQL 执行计划生成与优化器、全文检索并行化的相关工作；第 3 节介绍了全文检索、Spark 以及 Spark 执行计划生成与优化器的相关概念和原理；第 4 节介绍了分布式检索框架的总体设计，概括说明了各层的功能和作用；第 5 节介绍了框架中 SQL 文法、检索并行化以及检索优化的设计和实现；第 6 节介绍了框架应用于 Spark SQL 的情况及 Spark SQL 内核修改的总览；第 7 节对框架的性能以及扩展性进行了实验；最后总结全文，并指出下一步的研究工作。

2　相关工作

与框架相关的研究工作可以分为大数据处理系统的关系型接口、SQL 执行计划生成与优化器、全文检索并行化 3 个方面。

1)大数据查询语言及框架

基于 MapReduce 的大数据处理系统能够给予用户强大的但低层次、过程式的编程接口。基于此类系统的编程是繁琐的，为了实现高效的性能，用户需要自行优化。因此，众多系统(不限于 MapReduce 编程模式)如 Pig[9]，Hive，Impala[10]，Dremel[11]，BlinkDB[12] 和 Spark 提供了查询语言接口和自动优化技术，无须编写底层代码和关注底层执行引擎细节，增强了用户体验。

2)SQL 执行计划生成与优化器

SQL 执行计划生成与优化器将 SQL 翻译成底层执行引擎能够认知的物理执行计划。Hive，Dremel 和 Spark SQL 都有优化器。Hive 通过 ANTLR[13] 工具识别 SQL 语句，并通过定义的转换规则将其转换为底层的 MapReduce 任务。Spark 则是通过 Catalyst[8] 引擎进行解析。此类优化器都是通过将 SQL 解析成语法树，根据一系列分析和优化策略将其转换为底层的执行任务。过程通常分为：根据词法和语法解析得到抽象语法树；规范校验；与数据字典进行元数据绑定得到计划树；计划树优化；最优计划选择；物理执行。

3)全文检索并行化

全文检索包括建立索引和利用索引进行快速查询两个过程。全文检索的并行化是这两个过程在分布式环境下的并行化实现，全文检索引擎 Solr Cloud 和 ElasticSearch 均通过分片机制将索引分布到不同机器上，当一个搜索请求到来时将其分发到索引分片所在机器上并行执行，并返回查询结果。其中，每台机器上索引的操作由 Lucene 完成，多台机器检索的并行化过程由系统进行通信和调度。

目前，针对 Spark 场景下的全文检索工具有 spark-lucenerdd[14]，但是 spark-lucenerdd 只能在一次作业中使用建立的索引，无法在多次作业中重用历史索引，并且没有对索引的存储优化，更无法利用 SQL 进行全文检索，其面向群体为对 Spark 熟悉的开发人员。

3　相关概念和原理

3.1　全文检索

全文检索是指计算机索引程序通过扫描文档，对每一个词建立索引，记录该词出现的位置和次数，当用户查询时，检索程序根据事先建立好的索引进行查找，并将查询结果反馈给用户的检索方式。

全文检索系统是按照全文检索理论建立起来的提供全文检索服务的软件系统。全文检索系统的结构如图 1 所示，一般来说，全文检索系统需要基本的建立索引和查询的功能，以及文本分析、对外接口等模块。

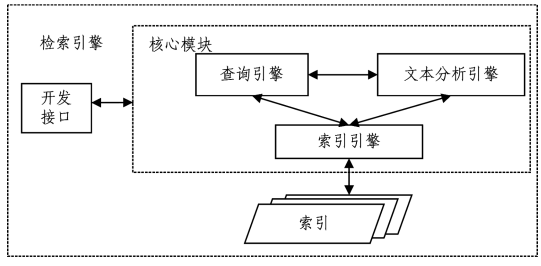

图 1　检索引擎架构

Fig. 1　Search engine architecture

3.2 Spark 介绍

Spark 是由 UC Berkeley AMPLab 提出的类 Hadoop Mapreduce 大数据处理框架。弹性分布式数据集(RDD)以及宽窄依赖是 Spark 进行作业调度及实现的核心。

1)RDD 的概念

RDD 是一个容错的、并行的数据结构,可以让用户显式地将数据存储到磁盘和内存中,并能控制数据的分区;同时,RDD 还提供了一组丰富的接口来操作这些数据。RDD 相互依赖从而形成有向无环图(DAG),Spark 通过分析 DAG 划分任务调度和执行,并提供 Cache 机制来支持多次迭代计算时的数据共享,大大减少了迭代计算之间重复读取数据的开销,这对于需要多次迭代的数据挖掘和分析应用的性能有很大的帮助。

2)数据依赖与性能

RDD 作为数据结构,本质上是一个只读的分区记录集合。一个 RDD 可以包含多个分区,每个分区就是数据集的一部分。RDD 可以相互依赖从而形成有向无环图。

如果 RDD 的每个分区最多只能被一个子 RDD 的一个分区使用,则称之为窄依赖;若多个子 RDD 分区都可以依赖,则称之为宽依赖。不同的操作依据其特性可能会产生不同的依赖。

区分宽窄依赖对于 Spark 作业调度和性能分析都非常重要。窄依赖意味着可以在同一台机器上进行 PipeLine 的操作,相当于对数据操作的算子进行叠加,避免了更多任务的生成;而宽依赖通常伴随着数据的 Shuffle 操作,容易出现性能问题,因此良好的算法和框架设计应尽量避免产生宽依赖的情况。

3.3 Spark SQL 翻译引擎

翻译引擎是将 SQL 转换成底层分布式计算引擎能够认知的任务的重要步骤。

在 Spark SQL 中,对 SQL 的解析过程都是通过 Catalyst 进行的,Catalyst 是通用的 SQL 翻译引擎,负责执行计划的生成与优化,解析过程为:

1)使用 ANTLR 进行文法解析。

2)Parser 通过 visitor 模式将 ANTLR 形成的语法树替换为 Catalyst 中定义的树节点构成的计划树。

3)Analyzer 将计划树与元数据信息相关联;Optimizer 对计划树进行优化,如常量折叠、谓词下推等。

4)物理计划转换器(Spark Planner)则是将计划树的每个节点转换成与底层 Spark 执行引擎相匹配的物理计划,每一个物理计划都包含了对 RDD 或者数据源的操作。RDD 是 Spark 进行任务分配且调度之前的最后一步,它代表了底层数据以及对数据操作算子的封装。

4 框架设计

本文框架包含 4 层:接收用户查询语句的 SQL 客户端层;SQL 翻译引擎,负责执行计划的解析和优化;并行计算层,用来提供分布式的全文检索功能;分布式索引存储层,负责存储原始数据和索引数据。

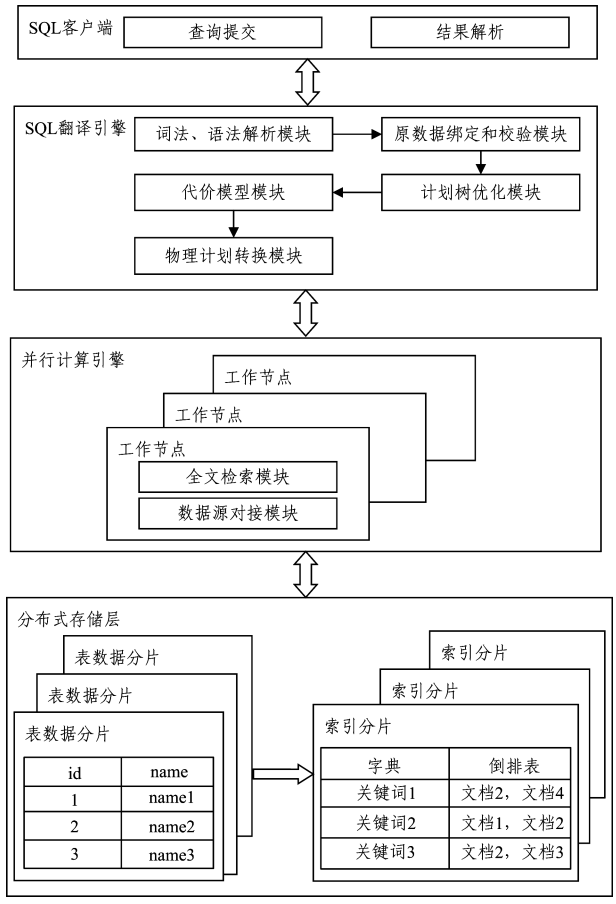

图 2 系统架构

Fig. 2 System architecture

4.1 SQL 客户端

SQL 客户端能够接收用户输入的 SQL 语句,查询提交模块将 SQL 提交给翻译引擎,结果解析模块将查询结果解析并返回给客户端,采用 Spark SQL 的 CLI 实现。

4.2 SQL 翻译引擎

SQL 翻译引擎模块是将 SQL 翻译成执行计划树并进行优化的模块。类似于编译原理的解析过程,首先,根据词法和语法规则对 SQL 语句进行切分,从而形成一棵语法树。语法树从底向上包含了对表的一系列语义操作。之后,根据一系列替换和优化规则,对语法树结构进行改变,通过 cost model 选择适合执行引擎的最优物理执行计划树交给底层执行引擎,执行结果返回 SQL 客户端。

在 Spark 中,Catalyst 作为执行计划生成与优化框架将 SQL 语句解析为多个 RDD 的操作,RDD 之间形成的 DAG 交给并行计算引擎进行作业规划和执行。SQL 解释器采用 Catalyst,通过修改 Catalyst,识别全文检索文法,并将全文检索操作映射为包含全文检索完整功能的 SearchRDD 的操作,并交给并行计算引擎执行。

4.3 并行计算层

并行计算层将 SQL 解释层的全文检索操作以多任务并行的方式运行。并行计算层包含了全文检索和数据源对接模块。表由多个分区数据构成,全文检索模块负责对输入文档

进行词法、语法分析，建立索引等操作；数据源对接模块将表中各分区数据建立的索引并行写入分布式存储层；而查询的并行化则是基于各分区的索引并行查询，最后通过全局 Reduce 操作返回得分最高的 K 个结果。

4.4　分布式索引存储层

分布式索引存储层采用 HDFS 作为存储文件系统，索引的存储采用分片和副本方式，从而提升了并行计算层作业执行的并发度和效率。本文设计和实现了两种检索优化方案，即分别侧重性能优化和存储优化两种场景的索引存储和原表数据还原策略，即全量存储和索引指定列策略；同时针对存储优化场景，提出了时间复杂度为 $O(n)$ 的查询结果与原表数据连接算法。全量存储适用于以最短时间获得查询结果的场景，但索引存储量较大；而索引指定列适用于存储空间有限的场景，索引存储量在表拥有上百、上千列时的优势非常明显，而且，高性能的数据连接算法能够保证在可接受的时间内返回查询结果。

5　各层设计

在大数据处理和分析系统中，SQL 是用户进行操作的直接接口，SQL 翻译器识别用户提交的 SQL 查询语句，通过内部定义的转换和优化规则，生成物理执行计划。物理执行计划包含了如何执行规划底层作业的细节，包括操作的定义以及与文件系统的交互。

框架在查询语言到检索模型的流程翻译转换、检索模型的并行化以及索引读写和查询优化上进行了设计与实现。

5.1　查询语言到检索模型的流程翻译转换

翻译器是将 SQL 转换成底层分布式计算引擎能够认知的任务的重要步骤。

5.1.1　问题描述

翻译器并不支持全文检索。

5.1.2　解决方案

本文在文法解析模块上，增加了全文检索文法以及全文检索文法识别规则；在物理计划模块上，增加了全文检索文法的转换规则，支持将查询操作下推到数据源；在物理计划执行模块，实现了 SearchRDD 类，包含索引建立和索引查询功能。其中，文法设计参考了 MySQL 中的全文检索文法，MySQL 有 3 种方式建立索引，此处只参考一种，这种方式相比其他两种更加符合用户习惯。

建立全文索引的 SQL 语句与 MySQL 类似，如表 1 所列，可以针对某个表的若干列建立索引，同时支持两类索引存储。而对于全文检索方面，表 2 提供了比 MySQL 形式更加丰富、功能更强大的接口，如词项查询(TermQuery)、模糊查询(FuzzyQuery)、段查询(PhraseQuery)、前缀查询(PrefixQuery)、表达式查询(QueryParser)等，以满足用户复杂查询的需要。

表 1　索引文法的建立

Table 1　Construction of index grammar

Spark SQL	MySQL
CREATE INDEX index_name ON TABLE talbe_name (column1,column2) [USING org.apache.spark.sql.index] [STRATEGY QUICKWAY|NOQUICK]	CREATE FULLTEXT INDEX index_name ON table_name (column1, column2)

表 2　查询文法

Table 2　Query grammar

Spark SQL	MySQL
SELECT * FROM index_name WHERE TERMQUERY|FUZZY- QUERY|PHRASEQUERY| PREFIXQUERY|QUERYPARSER ('field','queryString', ['maxEdits',] 'topK')	SELECT * MATCH(col1,col2) AGAINST('query string' [search_modifier]) FROM table_name WHERE MATCH(col1,col2) AGAINST('query string' [search_modifier])

全文检索文法的翻译过程如图 3 所示，SQL 语句通过转换为语法树最终转化为物理执行计划，物理执行计划包含了对 RDD 的操作。

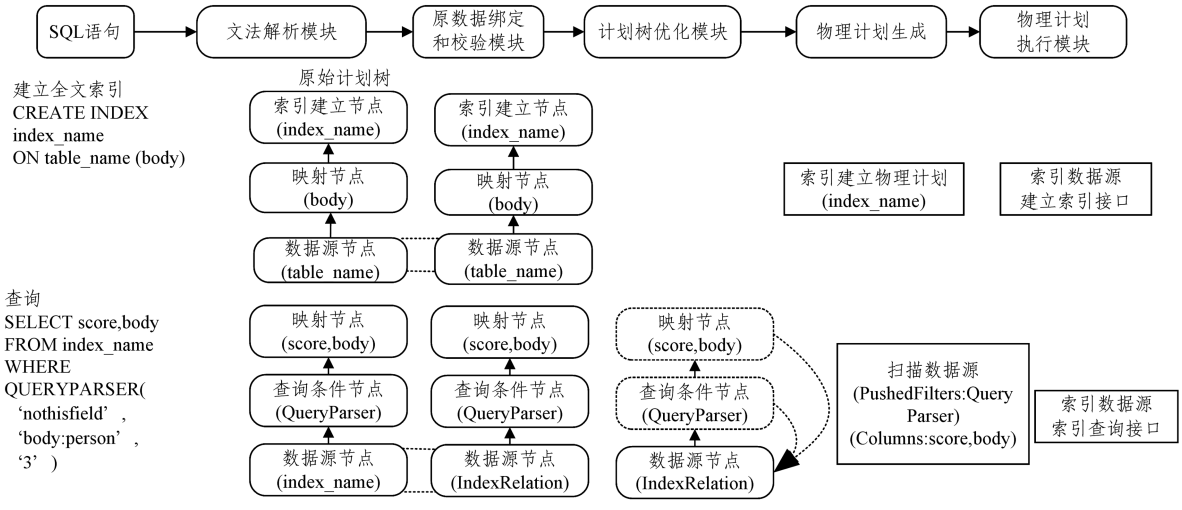

图 3　全文检索文法的翻译过程

Fig. 3　Translation process of full-text search grammar

建立索引时,文法解析模块首先将 SQL 解析为具有祖先关系的 3 个节点(即索引建立节点、映射节点以及数据源节点)的逻辑计划树。该树表明选取 table_name 表的 body 列数据建立索引,并且索引存储为另一张表 index_name;在元数据分析和校验模块中,数据源节点与元数据进行关联;在物理计划生成模块中,索引建立逻辑计划节点转换为索引建立物理计划节点;在物理计划执行模块中,调用索引数据源的索引建立接口。

进行全文检索时,文法解析模块首先将 SQL 解析为具有 3 个节点的逻辑计划树,即映射节点、查询条件节点和数据源节点,该树表明利用 QUERYPARSER 对索引 index_name 进行检索,分区检索结果只返回 body 列和新添加的 score 列的数据;在元数据分析和校验模块中,数据源节点与元数据进行关联;在计划树优化模块中,将映射节点和查询条件节点下推至数据源逻辑计划节点;在物理计划生成模块中,索引查询逻辑计划节点转换为索引查询物理计划节点;在物理计划执行模块中,调用索引数据源的查询接口。

5.2 并行计算层

检索模型描述了从数据源进行信息抽取,根据用户输入的查询语句返回符合条件的结果列表的一系列步骤。

5.2.1 问题描述

目前,Spark 并不支持全文检索的并行化,主要包括索引建立的并行化和查询匹配计算的并行化。

5.2.2 解决方案

本文通过使用 3 种方法来解决该问题:数据分片;多节点多线程并行任务;索引分片。

单机环境与分布式环境下的对比如图 4 所示。分布式环境下,表数据由多个分区组成,建立索引时,输入包含多个分区数据的表;对于每个分区启动相同处理逻辑的任务,每个任务使用全文检索组件处理该分区数据并写入分布式索引存储层,每个分区具有独立的索引文件,查询时,针对每一个分区索引启动一个任务,得到基于分区形式的查询结果,最后汇总得到全局得分 Top K 的结果(只有一个分区)。

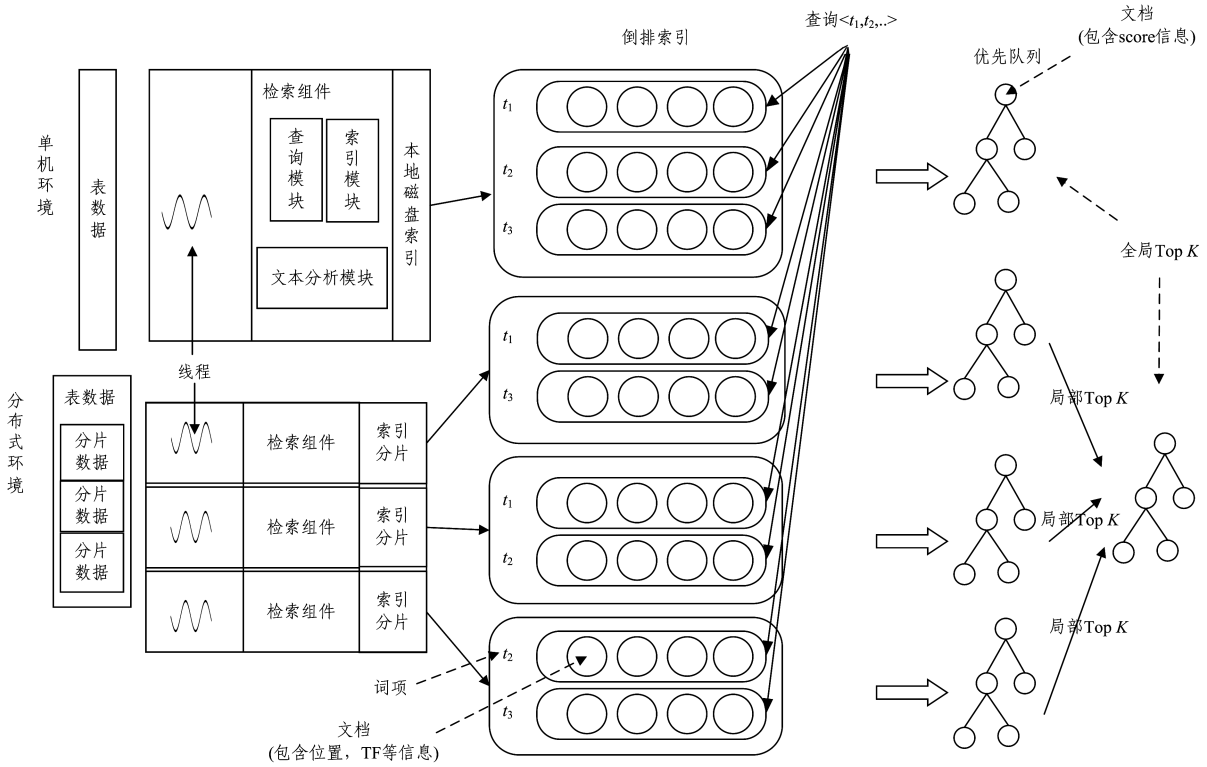

图 4 检索模型并行化

Fig. 4 Parallelization of search model

具有多个分区的数据表和并行查询结果由 RDD 表示;每个线程的任务由 Task 表示,Task 中使用 Lucene 作为索引构建和查询模块的实现工具;索引存储使用 HDFS;并行查询结果需要使用 Reduce 操作得到全局 Top K。

5.3 检索优化

在海量数据的全文检索中,除了提高数据处理的并行度,合理的检索优化方案对于提升海量数据的查询性能、减少索引数据的存储量至关重要。

全文检索中索引的存储与文档、域、字典、倒排表以及存储文档时的存储方式有关。

文档是加入索引和查询结果的最小单位,其中包含了若干的域。在数据库中,文档对应于表中的某一行,而域对应于某一列。

字典是针对若干域(列)进行分词后形成的非重复的单词表,倒排列表保存了单词在哪些文档出现以及出现的位置(见图 4)。

索引的存储方式分为两种:分词和索引(Token 和 Index),存储(Store)。分词和索引(Token 和 Index)是指对该域(列)进行分词并将分词结果加入字典和倒排表。根据倒排表,如果该列中的某一行中包含该单词,那么该单词就能被搜索出来,即得分大于 0;否则得分为 0。存储(Store)是指是否对该域(列)进行存储。如果用户需要在索引中获取若干域的

原始信息，那么需要存储该域。

5.3.1　问题描述

在传统的索引建立过程中，存储的内容除了基本的字典和倒排索引信息，还有存储文档的其他信息，从而可以使用户在查询时既可以获取评分，又能还原得到其他域的原始信息。如图 5 所示，SQL 语句需要返回查询的结果以及相关的所有列信息，但由于没有存储相应的域，导致查询结果中相应域(列)返回为空。因此，为了适应所有情况下 SELECT 返回结果的需求，原始的策略需要存储所有列的数据。

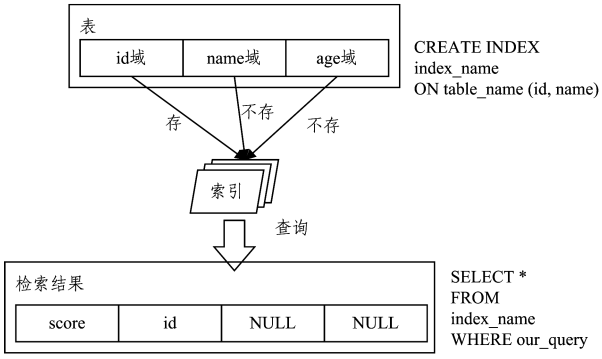

图 5　SQL 与域(列)存储

Fig. 5　SQL and fields(columns) storage

在单机数据量少的情况下，存储策略只需要额外存储用户需要的域(列)信息，并不会出现存储和性能瓶颈问题。随着数据量的上升，情况会产生非常大的差异。在海量数据的 SQL 检索下，表通常包含千万行或者上千列的数据，这产生了巨大的额外存储开销(在 Spark 与 Solr 或 ES 的联合使用中，也存在该问题)。这种情况下难以使用简单的存储策略。

缺失的域(列)数据可以通过获取原表相应位置的数据来填补，即查询结果通过与原表数据关联找到缺失的数据。由于关联操作的存在，会产生相应的性能代价。不存储任何列的数据，而仅仅索引 SQL 中指定的需要索引的列，并通过关联操作得到缺失的数据，这种方法可有效减少额外数据的存储量，但关联操作会降低查询性能。

5.3.2　解决方案

前文说明了海量数据检索中遇到的两类问题，即存储量和性能的综合考量。基于这两类问题，本文提出了两种存储策略：全量存储策略和索引(Token 和 Index)指定列策略。针对不同的现实需求，本文归纳和总结了两种存储策略的适用场景、存储和索引规则以及第二种策略中的关联算法。

1)适用场景

采用全量索引的策略时，使用的 Lucene 支持随机读，因此相应域的数据可以在 O(1)时间获得。

在只索引(Token 和 Index)指定列的数据策略中，返回结果只包含得分以及文档 ID，虽然可以通过文档 ID 与该文档在原始分区数据中的偏移量一致的特性找到原始数据，但是大部分海量存储系统对于数据的访问采用迭代器模式，不支持随机读，无法在 O(1)时间内完成。

因此，对于性能要求较高的场景，希望系统能够迅速返回原始数据，采用全量索引；对于性能要求不高但数据量异常庞大的场景，采用只索引指定列的方式。图 6 给出了两种存储策略的比较情况。

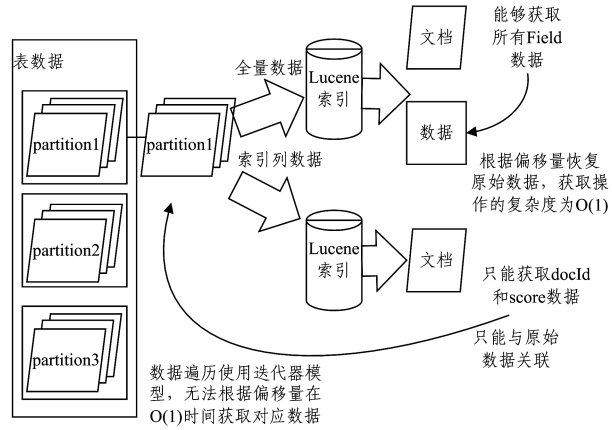

图 6　两类存储策略

Fig. 6　Two storage strategies

2)存储和索引规则

存储和索引规则描述了针对域(列)的索引和存储策略，以及原始数据获取的方式。表 3 列出了存储规则与原数据获取。

表 3　存储规则与原数据获取

Table 3　Storage rules and raw data acquisition

	全量存储	索引指定列
存储规则	A. 只有 SQL 指定(ON 关键字)的索引域(列)才会被分词和索引 B. 存储所有域(列)的数据	A. 只有 SQL 指定(ON 关键字)的索引域(列)才会被分词和索引 B. 除了分区 ID，不存储任何域(列)的数据
原表数据还原	从索引中获取	通过分区连接算法从原表中获取

分区对齐连接算法描述了具有多分区的查询结果与具有多分区的原表进行连接的过程。

由于针对每一个原表分区建立了一块索引，而查询时基于每块索引生成一个查询结果的分区，即查询结果的每一块分区是针对原表每一块分区的查询结果，因此分区形式一一映射。查询结果与原表均以 RDD 表示。

算法提出的目的是通过分区 ID 以及偏移量(文档 ID)找到这两个 RDD 中每一个映射分区内相同的对应点，使得原表数据与查询结果数据相拼接，返回包含了得分和原表相关列的完整结果。算法复杂度为 $O(n)$，n 为表中数据的总行数。

该算法的步骤如下：

步骤 1　将查询结果与原表数据的分区对齐(基于 mapPartitionsWithIndex)。

步骤 2　在查询结果的分区内，使用字典记录查询结果需要获取的原数据的所有偏移量和偏移量对应的评分。

步骤 3　在原表的分区内通过迭代和记录偏移量的方式找到在字典内的偏移量，最后将符合的原数据与得分拼接。

步骤 4　遍历所有映射分区，直到连接操作全部完成。

6　Spark SQL 架构与修改总览

Spark SQL 内核修改过程如图 7 所示。

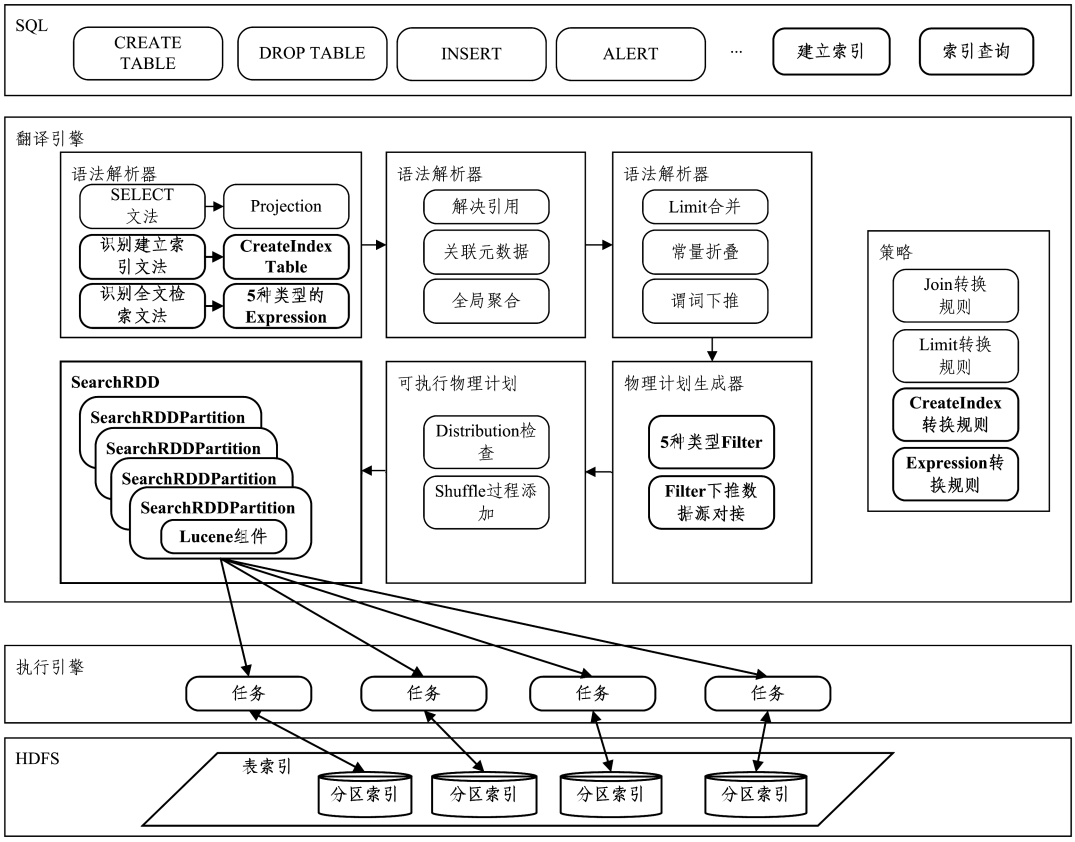

图 7 Spark SQL 内核修改

Fig. 7 Spark SQL kernel modification

其中,加粗部分为 Spark SQL 内核修改或添加的部分。系统架构与 Spark SQL 的架构保持一致,共分为 4 层,上层 SQL 客户端接受用户的 SQL 查询,翻译引擎使 SQL 经过语法分析、元数据绑定、计划优化、物理计划转换,最终转换为对 RDD(此处为 SearchRDD)的操作。执行引擎执行 SearchRDD 中分区索引的读写和查询操作,其中,索引的读写都是基于 HDFS 并行进行的,每个分区都会形成一块索引。

7 性能分析和扩展性实验

本文实验使用了 10 台物理机(1 台 master,9 台 slave),每台物理机的内存为 16 GB,CPU 为 Intel(R) Core(TM) i7-2600 CPU @3.40 GHz 8 核心,Hadoop 版本为 2.7.1,Spark 版本为基于社区最新 master 分支并加入全文检索模块的分支版本,运行在 standalone 模式下,集群最大的有效 Executor 数目为 36。

HDFS 和 Spark 的关键配置信息如表 4 所列。

表 4 HDFS 和 Spark 的关键配置信息

Table 4 Key configuration information of HDFS and Spark

配置项	描述	取值
HDFS Block Size	块大小	128 M
spark.driver.memory	客户端内存	8 GB
spark.executor.cores	executor 使用的核心数量	2
spark.executor.memory	executor 可用内存	3 GB
SPARK_WORKER_CORES	worker 端可用核数	8
SPARK_WORKER_MEMORY	worker 端可用内存	12 GB
SPARK_WORKER_INSTANCES	各节点 worker 数量	1

本文使用了 The Westbury Lab USENET Corpus 测试集,该测试集共包含 32440001 篇文档。针对实验环境,选取了前 M_i 篇文档作为实验数据,由于运算能力有限,M_i 的最大值为 3243904。文档数与文本形式的空间占用量如表 5 所列。

表 5 文档数与文本形式的空间占用量

Table 5 Numbers of documents and amount of space

文档数	数据量 M_i/MB
25343	32
50686	64
101372	128
202744	256
405488	512
810976	1024
1621952	2048
3243904	4096

使用表的形式存储文档,底层以 SequenceFile 作为存储格式。数据分片方式 S_i 有:1,2,4,8,16,32,64。分片方式用于测试任务并发度与分片数、空间占用量、集群环境配置等因素的关系。

表的命名为:usenet_corpus_M_i_S_i_test,其中 M_i = {25343, 50686, 101372, 202744, 405488, 810976, 1621952, 3243904},S_i = {1,2,4,8,16,32,64}。

性能分析:由于索引的读写和查询都是基于原始数据(表)的分区并行的,即窄依赖,执行时间为 $T_{\text{IndexPerformace}}$。得到全局 Top K 的结果需要两个步骤:在每个分区内获取原数据,将评分数据与原数据进行拼接,性能为

$T_{\text{GetOriginalDataPerformance}}$；根据 score 利用 Reduce 操作（宽依赖）得到全局 Top K，性能为 $T_{\text{ReducePerformance}}$。不考虑作业启动、调度等其他操作的占用时间，Spark SQL 下全文检索作业的执行时间为：

$$T_{\text{Perfermance}} = T_{\text{IndexPerformace}} + T_{\text{GetOriginalDataPerformance}} + T_{\text{ReducePerformance}}$$

本文从 3 个方面来评测不同存储策略、分片、文档数对于系统的影响，以及索引建立时间、全文检索时间和索引存储量。

7.1　建立索引

全量存储策略 SQL 语句为：

```
CREATE INDEX
usenet_corpus_Mi_Si_test_index
ON TABLE usenet_corpus_Mi_Si_test
STRATEGY QUICKWAY;
```

索引指定列策略 SQL 语句为：

```
CREATE INDEX
usenet_corpus_Mi_Si_test_index_noquick
ON TABLE usenet_corpus_Mi_Si_test
STRATEGY NOQUICK;
```

实验结论如下：

结合图 8 和图 9，分析不同文档数下的执行时间可以发现，Spark SQL 在全量存储和索引指定列策略下建立索引的平均时间分别为 MySQL 执行时间的 0.6% 和 0.5%，因此 MySQL 难以适应海量数据的全文检索。

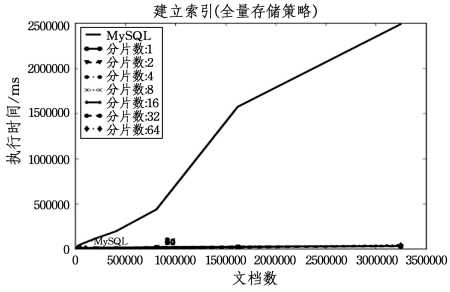

图 8　全量存储策略的对比实验

Fig. 8　Comparsion experiment of total storage strategy

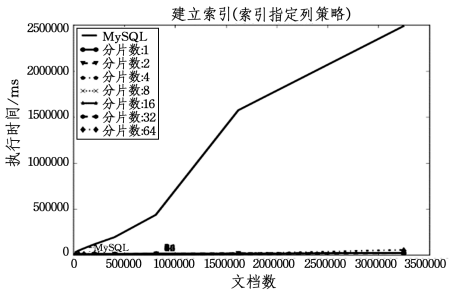

图 9　索引指定列策略的对比实验

Fig. 9　Expansibility experiment of index designated columns strategy

结合图 10 和图 11，对不同文档数下的执行时间分析发现，当数据量固定时，随着分片数的增大，由于多个任务能够并行执行，且所分配到的数据量减小，建立索引的性能得到提升。当分片数固定时，数据量以 2 的指数倍上升，执行时间与文档数的斜率小于 1，说明检索的并行化有效缓解了由数据量的快速上升而带来的性能瓶颈问题。

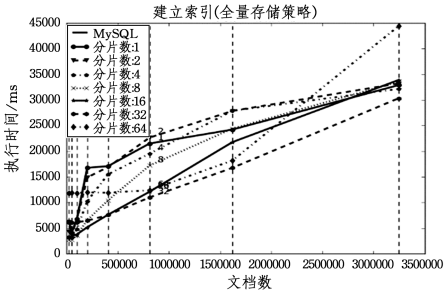

图 10　全量存储策略的扩展性实验

Fig. 10　Comparsion experiment of total storage columns

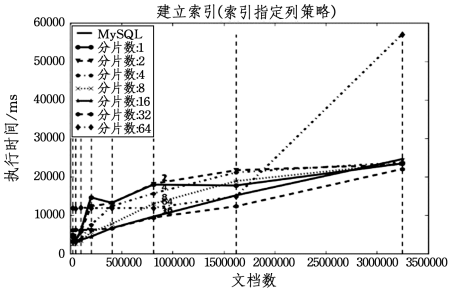

图 11　索引指定列策略的扩展性实验

Fig. 11　Expansibility experiment of index designated columns strategy

问题：在分片数为 64 时，大于集群中最多的 Executor 数量为 35，并行度的不足导致作业需要两轮才能执行完成，虽然每个作业分配到的数据量变少，但是总作业处理时间增加（包括作业规划、启动等因素）。

随着文档数的增加，虽然分片数已增加，但作业总时间趋近，原因是分片的空间占用量大于或等于 *Block Size*，导致额外作业启动。

索引指定列策略只存储一部分数据，节省了大量的磁盘 IO 操作，相比于全量存储策略，平均执行时间减少了 17%，但是仍然存在并行度不足的问题。

7.2　全文检索

全量存储策略 SQL 语句为：

```
SELECT *
FROM usenet_corpus_Mi_Si_test_index
WHERE
QUERYPARSER('nothisfield','body:person','3')
```

索引指定列策略 SQL 语句为：

```
SELECT *
FROM usenet_corpus_Mi_Si_test_index_noquick
WHERE
QUERYPARSER('nothisfield','body:person','3')
```

实验结论如下：

结合图 12 和图 13，分析不同文档数下的执行时间可以发现，Spark SQL 在全量存储和索引指定列策略下全文检索的平均执行时间分别是 MySQL 的 1% 和 10%。

结合图 14 和图 15，对不同文档数下的执行时间分析发现，当数据量固定时，随着分片数的增多，查询时间缩短；当分片数固定时，随着数据量的上升，查询时间增长得十分缓慢，因此框架拥有良好的扩展性。

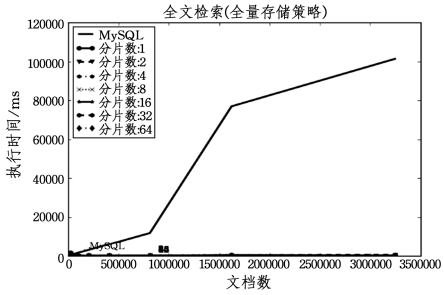

图 12 全量存储策略的对比实验

Fig. 12 Comparsion experiment of total storage strategy

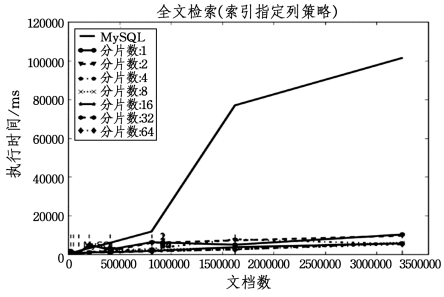

图 13 索引指定列策略的对比实验

Fig. 13 Comparison experiment of index designated columns strategy

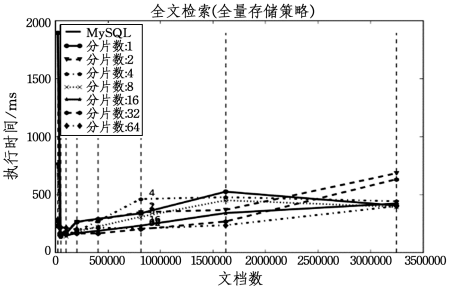

图 14 全量存储策略的扩展性实验

Fig. 14 Expansibility experiment of total storage strategy

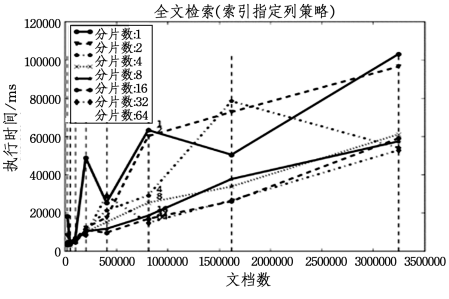

图 15 索引指定列策略的对比实验

Fig. 15 Comparison experiment of index designated columns strategy

由于获取原表数据需要执行分区对齐连接算法,因此索引指定列策略的执行时间比全量存储策略的执行时间长。

索引指定列策略中,索引存储量的下降使得分片数与作业执行时间有较好的对应关系,在目前的数据量下,没有出现明显的性能下滑趋势。

7.3 索引存储量

结合图 16 和图 17,对不同文档数下的索引存储量进行分析发现,索引指定列策略的索引存储量是 MySQL 的 55.0%,是全量存储策略的 36.7%。由于只存储必要的分词和索引信息,不存储原始文档,因此随着数据量和列数的增加,索引指定列策略的优势将更加明显。

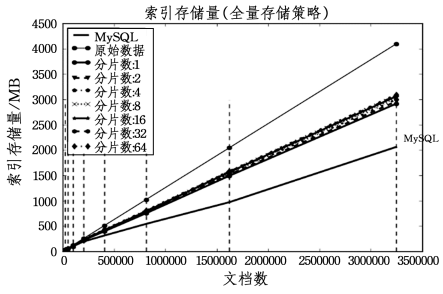

图 16 全量存储策略的索引量

Fig. 16 Index storage volume of total storage strategy

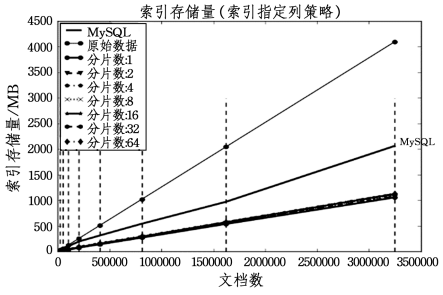

图 17 索引指定列策略的索引量

Fig. 17 Index storage volume of index designated columns strategy

7.4 实验总结

实验结果表明:在现有实验条件下,对比 MySQL,在全量存储和索引指定列策略中,该框架索引构建时间缩短为原来的 0.6%和 0.5%,查询时间分别缩短为原来的 1%和 10%,索引存储量在索引指定列策略下减少为原来的 55.0%,并且随着数据量和列数的增加,索引量与 MySQL 之间的差别增大。

结束语 传统的关系型数据库系统难以满足海量数据下全文检索的需求,而目前主流的大数据处理系统 Spark 只提供了简单的数据查询和分析功能,对于全文检索并没有一套完整的框架设计和实现。

本文系统从 SQL 文法设计、检索并行化、检索优化 3 方面介绍了 Spark SQL 分布式全文检索框架的设计与实现。在文法设计中,支持在若干列上建立索引,同时提供了丰富的检索功能,用来应对不同的检索需求;在索引并行化方面,将基于单机单进程/线程的全文检索扩展到基于 Spark 的多节点多任务并行处理,有效解决了海量数据下传统数据库全文检索的瓶颈问题;在检索优化方面,提出了两种存储策略应对不同场景下对于性能和存储的要求,并且提出了拥有 $O(n)$ 时间复杂度的分区对齐连接算法。最后,通过对比不同分片(并行度)和存储策略下的索引建立、全文检索、存储量,证明了分布式全文检索框架在性能、索引存储方面远远超过传统关系型数据库。

下一步工作将增强全文检索的功能,使其支持维度搜索和空间搜索,优化索引分片策略,并将其贡献给 Spark 社区。

参 考 文 献

[1] SUN D W,ZHANG G Y,ZHENG W M. Big Data Flow computation:Key technology and system examples[J]. Journal of Software,2014,25(4):839-862. (in Chinese)
孙大为,张广艳,郑纬民. 大数据流式计算:关键技术及系统实例[J]. 软件学报,2014,25(4):839-862.

(下转第 145 页)

白宝明，孙成，陈佩瑶，等. 信道编码技术新进展[J]. 无线电通信技术，2016，42(2)：1-8.

[5] JIAO X P，WEI H Y，MU J J. Improved ADMM penalized decoder for irregular low-density parity-check codes[J]. IEEE Communications Letters，2015，19(6)：913-916.

[6] ANASSI O，CONDE-CANENCIA L，MANSOUR M，et al. Non-binary Low-Density Parity-Check coded cyclic Code-Shift Keying[C]// IEEE Wireless Communications and Networking Conference. Shanghai，China，2013：3890-3894.

[7] MA Z，SHI Z，ZHOU C，et al. Design of signal space diversity based on non-binary LDPC code [C]// International Conference on Communications. Fujian，China，2008：31-34.

[8] RONG B，JIANG T，LI X，et al. Combine LDPC codes over GF(q) with q-ary modulations for bandwidth efficient transmission [J]. IEEE Transactions on Broadcasting，2008，54(1)：78-84.

[9] LI G，FAIR I J，KRZYMIEN W A. Density evolution for nonbinary LDPC codes under Gaussian approximation[J]. IEEE Transactions on Information Theory，2009，55(3)：997-1015.

[10] CHEN Y M，GAO X L，WANG Z X，et al. Performance Analysis of Nonbinary and Binary LDPC Codes[J]. Electronic Design Engineering，2013，23(21)：94-95. (in Chinese)
陈明阳，高兴龙，王中训，等. 多元 LDPC 码与二元 LDPC 码的性能分析[J]. 电子设计工程，2013，23(21)：94-95.

[11] SAMAD A M，KAMARULZAMAN N，HAMDANI M A，et al. The potential of Unmanned Aerial Vehicle(UAV) for civilian and mapping application [C]// Proceedings of 2013 IEEE 3rd International Conference on System Engineering and Technology. Shah Alam：IEEE，2013：313-318.

[12] XU Y H，ZHOU S K，ZHU Q M，et al. Simulation of UAV Communication Channel Based on Flight Trajectory[J]. Telecommunication Engineering，2013，53(5)：656-660. (in Chinese)
徐仪华，周生奎，朱秋明，等. 基于飞行轨迹无人机通信信道仿真[J]. 电讯技术，2013，53(5)：656-660.

[13] SALAMANCA L，OLMOS P M，Murillo-Fuentes J J，et al. Tree Expectation Propagation for ML Decoding of LDPC Codes over the BEC[J]. IEEE Transactions on Communications，2013，61(2)：465-473.

[14] SONG H，CRUZ J R. Reduced-complexity decoding of Q-ary LDPC codes for magnetic recording[J]. IEEE Transactions on Magnetics，2003，39(2)：1081-1087.

[15] WYMEERSCH H，STEENDAM H，MOENECLAEY M. Log-domain decoding of LDPC codes over GF(q)[C]// 2004 IEEE International Conference on Communcations. 2004：772-776.

[16] ZHAO S，WANG X，WANG T，et al. Joint detection-decoding of majority-logic decodable nonbinary LDPC coded modulation systems：An iterative noise reduction algorithm[C]// IEEE China Summit & International Conference on Signal and Information Processing. 2013：412-416.

[17] LACRUZ J O，GARCIA-HERRERO F，VALLS J，et al. One minimum only trellis decoder for non-binary low-density parity-check codes[J]. IEEE transactions on circuitsand systems I：regular papers，2015，62(1)：177-184.

[18] PENG T，YI X X，LI H，et al. OFDM-IDMA System with LDPC [J]. Journal of Chongqing Institute of Technology，2012，26(11)：80-82. (in Chinses)
彭涛，益晓新，李辉，等. LDPC 码在正交频分复用-交织多址系统中的应用[J]. 重庆理工大学学报，2012，26(11)：80-82.

(上接第 112 页)

[2] MENG X F，CI X. Big Data Management：Concept，technology and challenge[J]. Computer Research and Development，2013，51(1)：146-169(in Chinese).
孟小峰，慈祥. 大数据管理：概念、技术与挑战[J]. 计算机研究与发展，2013，51(1)：146-169.

[3] CHENG X Q，JIN X L，WANG Y Z，et al. A summary of large data systems and analysis techniques[J]. Journal of Software，2014(9)：1889-1908. (in Chinese)
程学旗，靳小龙，王元卓，等. 大数据系统和分析技术综述[J]. 软件学报，2014(9)：1889-1908.

[4] DEAN J，GHEMAWAT S. MapReduce：simplified data processing on large clusters[J]. Communications of the ACM，2008，51(1)：107-113.

[5] SHVACHKO K，KUANG H，RADIA S，et al. The hadoop distributed file system[C]// 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies(MSST). IEEE，2010：1-10.

[6] THUSOO A，SARMA J S，JAIN N，et al. Hive：a warehousing solution over a map-reduce framework[C]// Proceedings of the VLDB Endowment. 2009：1626-1629.

[7] ZAHARIA M，CHOWDHURY M，DAS T，et al. Resilient distributed datasets：A fault-tolerant abstraction for in-memory cluster computing[C]// Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. USENIX Association，2012：2.

[8] 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. ACM，2015：1383-1394.

[9] OLSTON C，REED B，SRIVASTAVA U，et al. Pig latin：a not-so-foreign language for data processing[C]// Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. ACM，2008：1099-1110.

[10] KORNACKER M，BEHM A，BITTORF V，et al. Impala：A Modern，Open-Source SQL Engine for Hadoop[C]// Proceedings of the 7th Biennial Conference on Innovative Data Systems Research. 2015.

[11] MELNIK S，GUBAREV A，LONG J J，et al. Dremel：interactive analysis of web-scale datasets[J]. Proceedings of the VLDB Endowment，2010，3(1/2)：330-339.

[12] AGARWAL S，MOZAFARI B，PANDA A，et al. BlinkDB：queries with bounded errors and bounded response times on very large data[C]// Proceedings of the 8th ACM European Conference on Computer Systems. ACM，2013：29-42.

[13] PARR T J，QUONG R W. ANTLR：A Predicated[J]. Software—Practice and Experience，1995，25(7)：789-810.

[14] ZOUZIAS A. Spark-lucenerdd (Version 0.3.0)[EB/OL]. http://github.com/zouzias/spark-lucenerdd.