背景

随着大数据时代的到来，越来越多的数据流向了Hadoop生态圈，同时对于能够快速的从TB甚至PB级别的数据中获取有价值的数据对于一个产品和公司来说更加重要，在Hadoop生态圈的快速发展过程中，涌现了一批开源的数据分析引擎，例如Hive、Spark SQL、Impala、Presto等，同时也产生了多个高性能的列式存储格式，例如RCFile、ORC、Parquet等，本文主要从实现的角度上对比分析ORC和Parquet两种典型的列存格式，并对它们做了相应的对比测试。

列式存储

由于OLAP查询的特点，列式存储可以提升其查询性能，但是它是如何做到的呢？这就要从列式存储的原理说起，从图1中可以看到，相对于关系数据库中通常使用的行式存储，在使用列式存储时每一列的所有元素都是顺序存储的。由此特点可以给查询带来如下的优化：

- 查询的时候不需要扫描全部的数据，而只需要读取每次查询涉及的列，这样可以将I/O消耗降低N倍，另外可以保存每一列的统计信息(min、max、sum等)，实现部分的谓词下推。
- 由于每一列的成员都是同构的，可以针对不同的数据类型使用更高效的数据压缩算法，进一步减小I/O。
- 由于每一列的成员的同构性，可以使用更加适合CPU pipeline的编码方式，减小CPU的缓存失效。

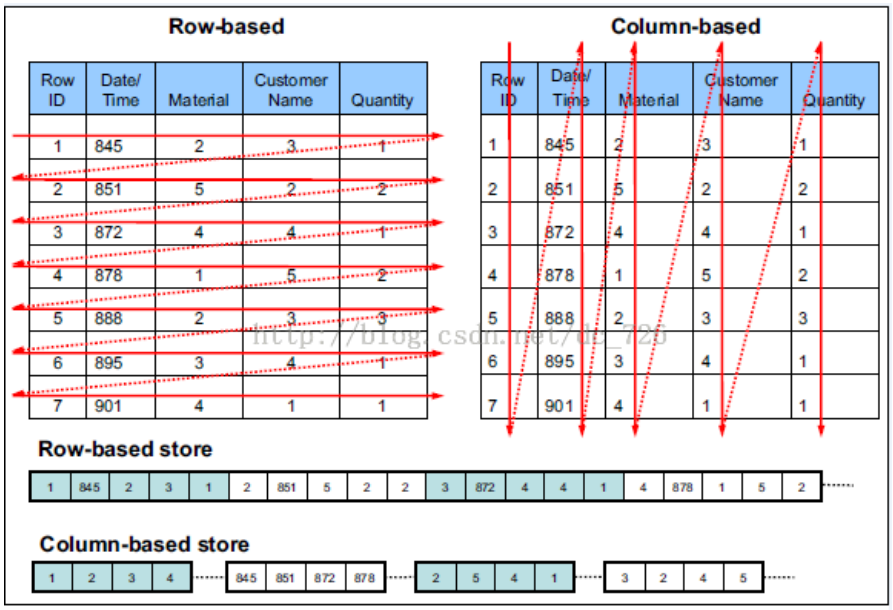


图1 行式存储VS列式存储

嵌套数据格式

通常我们使用关系数据库存储结构化数据，而关系数据库支持的数据模型都是扁平式的，而遇到诸如List、Map和自定义Struct的时候就需要用户自己解析，但是在大数据环境下，数据的来源多种多样，例如埋点数据，很可能需要把程序中的某些对象内容作为输出的一部分，而每一个对象都可能是嵌套的，所以如果能够原生的支持这种数据，查询的时候就不需要额外的解析便能获得想要的结果。例如在Twitter，他们一个典型的日志对象（一条记录）有87个字段，其中嵌套了7层，如下图。

```
struct LogEvent {
  1: optional logbase.LogBase log_base
  2: optional i64 event_value
  3: optional string context
  4: optional string referring_event
...
  18: optional EventNamespace event_namespace
  19: optional list<Item> items
  20: optional map<AssociationType,Association> associations
  21: optional MobileDetails mobile_details
  22: optional WidgetDetails widget_details
  23: optional map<ExternalService,string> external_ids
}
```

```
struct LogBase {
  1: string transaction_id,
  2: string ip_address,
  ...
  15: optional string country,
  16: optional string pid,
}
```

图2 嵌套数据模型

随着嵌套格式的数据的需求日益增加，目前Hadoop生态圈中主流的查询引擎都支持更丰富的数据类型，例如Hive、SparkSQL、Impala等都原生的支持诸如struct、map、array这样的复杂数据类型，这样促使各种存储格式都需要支持嵌套数据格式。

Parquet存储格式

Apache Parquet是Hadoop生态圈中一种新型列式存储格式，它可以兼容Hadoop生态圈中大多数计算框架(Mapreduce、Spark等)，被多种查询引擎支持（Hive、Impala、Drill等），并且它是语言和平台无关的。Parquet最初是由Twitter和Cloudera合作开发完成并开源，2015年5月从Apache的孵化器里毕业成为Apache顶级项目。

Parquet最初的灵感来自Google于2010年发表的Dremel论文，文中介绍了一种支持嵌套结构的存储格式，并且使用了列式存储的方式提升查询性能，在Dremel论文中还介绍了Google如何使用这种存储格式实现并行查询的，如果对此感兴趣可以参考论文和开源实现Drill。

数据模型

Parquet支持嵌套的数据模型，类似于Protocol Buffers，每一个数据模型的schema包含多个字段，每一个字段有三个属性：重复次数、数据类型和字段名，重复次数可以是以下三种：required(只出现1次)，repeated(出现0次或多次)，optional(出现0次或1次)。每一个字段的数据类型可以分成两种：group(复杂类型)和primitive(基本类型)。例如Dremel中提供的Document的schema示例，它的定义如下：

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward;
  }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
     }
    optional string Url;
  }
}
```

可以把这个Schema转换成树状结构，根节点可以理解为repeated类型，如图3。

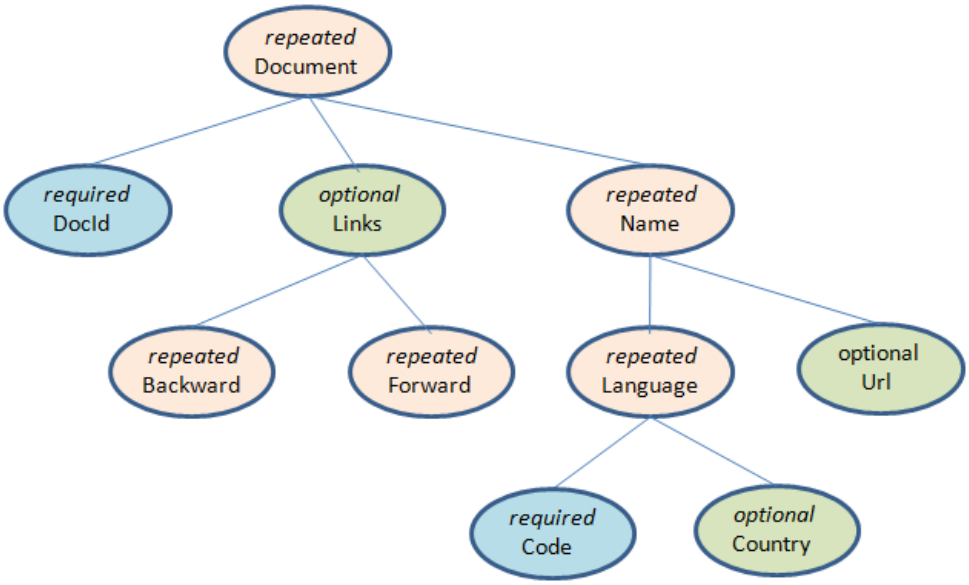


图3 Parquet的schema结构

可以看出在Schema中所有的基本类型字段都是叶子节点，在这个Schema中一共存在6个叶子节点，如果把这样的Schema转换成扁平式的关系模型，就可以理解为该表包含六个列。Parquet中没有Map、Array这样的复杂数据结构，但是可以通过repeated和group组合来实现的。由于一条记录中某一列可能出现零次或者多次，需要标示出哪些列的值构成一条完整的记录。这是由Striping/Assembly算法实现的。

由于Parquet支持的数据模型比较松散，可能一条记录中存在比较深的嵌套关系，如果为每一条记录都维护一个类似的树状结可能会占用较大的存储空间，因此Dremel论文中提出了一种高效的对于嵌套数据格式的压缩算法：Striping/Assembly算法。它的原理是每一个记录中的每一个成员值有三

部分组成：Value、Repetition level和Definition level。value记录了该成员的原始值，可以根据特定类型的压缩算法进行压缩，两个level值用于记录该值在整个记录中的位置。对于repeated类型的列，Repetition level值记录了当前值属于哪一条记录以及它处于该记录的什么位置；对于repeated和optional类型的列，可能一条记录中某一列是没有值的，假设我们不记录这样的值就会导致本该属于下一条记录的值被当做当前记录的一部分，从而造成数据的错误，因此对于这种情况需要一个占位符标示这种情况。

通过Striping/Assembly算法，parquet可以使用较少的存储空间表示复杂的嵌套格式，并且通常Repetition level和Definition level都是较小的整数值，可以通过RLE算法对其进行压缩，进一步降低存储空间。

文件结构

Parquet文件是以二进制方式存储的，是不可以直接读取和修改的，Parquet文件是自解析的，文件中包括该文件的数据和元数据。在HDFS文件系统和Parquet文件中存在如下几个概念：

- HDFS块(Block)：它是HDFS上的最小的副本单位，HDFS会把一个Block存储在本地的一个文件并且维护分散在不同的机器上的多个副本，通常情况下一个Block的大小为256M、512M等。
- HDFS文件(File)：一个HDFS的文件，包括数据和元数据，数据分散存储在多个Block中。
- 行组(Row Group)：按照行将数据物理上划分为多个单元，每一个行组包含一定的行数，在一个HDFS文件中至少存储一个行组，Parquet读写的时候会将整个行组缓存在内存中，所以如果每一个行组的大小是由内存大的小决定的。
- 列块(Column Chunk)：在一个行组中每一列保存在一个列块中，行组中的所有列连续的存储在这个行组文件中。不同的列块可能使用不同的算法进行压缩。
- 页(Page)：每一个列块划分为多个页，一个页是最小的编码的单位，在同一个列块的不同页可能使用不同的编码方式。

通常情况下，在存储Parquet数据的时候会按照HDFS的Block大小设置行组的大小，由于一般情况下每一个Mapper任务处理数据的最小单位是一个Block，这样可以把每一个行组由一个Mapper任务处理，增大任务执行并行度。Parquet文件的格式如下图所示。

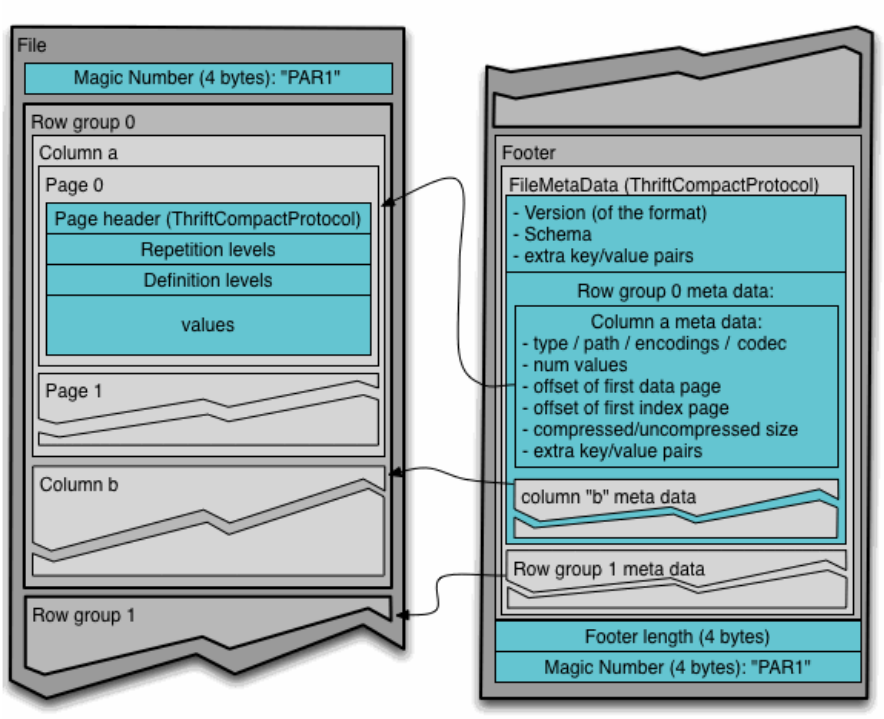


图4 Parquet文件结构

上图展示了一个Parquet文件的结构，一个文件中可以存储多个行组，文件的首位都是该文件的Magic Code，用于校验它是否是一个Parquet文件，Footer length存储了文件元数据的大小，通过该值和文件长度可以计算出元数据的偏移量，文件的元数据中包括每一个行组的元数据信息和当前文件的Schema信息。除了文件中每一个行组的元数据，每一页的开始都会存储该页的元数据，在Parquet中，有三种类型的页：数据页、字典页和索引页。数据页用于存储当前行组中该列的值，字典页存储该列值的编码字典，每一个列块中最多包含一个字典页，索引页用来存储当前行组下该列的索引，目前Parquet中还不支持索引页，但是在后面的版本中增加。

数据访问

说到列式存储的优势，Project下推是无疑最突出的，它意味着在获取表中原始数据时只需要扫描查询中需要的列，由于每一列的所有值都是连续存储的，避免扫描整个表文件内容。

在Parquet中原生就支持Project下推，执行查询的时候可以通过Configuration传递需要读取的列的信息，这些列必须是Schema的子集，Parquet每次会扫描一个Row Group的数据，然后一次性得将该Row Group里所有需要的列的Cloumn Chunk都读取到内存中，每次读取一个Row Group的数据能够大大降低随机读的次数，除此之外，Parquet在读取的时候会考虑列是否连续，如果某些需要的列是存储位置是连续的，那么一次读操作就可以把多个列的数据读取到内存。

在数据访问的过程中，Parquet还可以利用每一个row group生成的统计信息进行谓词下推，这部分信息包括该Column Chunk的最大值、最小值和空值个数。通过这些统计值和该列的过滤条件可以判断该Row Group是否需要扫描。另外Parquet未来还会增加诸如Bloom Filter和Index等优化数据，更加有效的完成谓词下推。

ORC文件格式

ORC文件格式是一种Hadoop生态圈中的列式存储格式，它的产生早在2013年初，最初产生自Apache Hive，用于降低Hadoop数据存储空间和加速Hive查询速度。和Parquet类似，它并不是一个单纯的列式存储格式，仍然是首先根据行组分割整个表，在每一个行组内进行按列存储。ORC文件是自描述的，它的元数据使用Protocol Buffers序列化，并且文件中的数据尽可能的压缩以降低存储空间的消耗，目前也被Spark SQL、Presto等查询引擎支持，但是Impala对于ORC目前没有支持，仍然使用Parquet作为主要的列式存储格式。2015年ORC项目被Apache项目基金会提升为Apache顶级项目。

数据模型

和Parquet不同，ORC原生是不支持嵌套数据格式的，而是通过对复杂数据类型特殊处理的方式实现嵌套格式的支持，例如对于如下的hive表：

```
CREATE TABLE `orcStructTable`(
  `name` string,
  `course` struct<course:string,score:int>,
  `score` map<string,int>,
  `work_locations` array<string>)
```

ORC格式会将其转换成如下的树状结构：

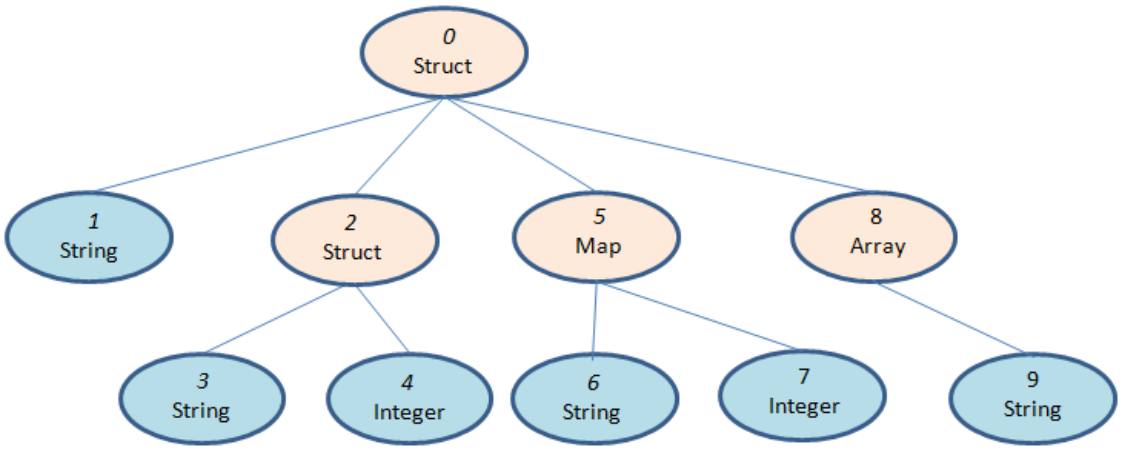


图5 ORC的schema结构

在ORC的结构中这个schema包含10个column，其中包含了复杂类型列和原始类型的列，前者包括LIST、STRUCT、MAP和UNION类型，后者包括BOOLEAN、整数、浮点数、字符串类型等，其中STRUCT的孩子节点包括它的成员变量，可能有多个孩子节点，MAP有两个孩子节点，分别为key和value，LIST包含一个孩子节点，类型为该LIST的成员类型，UNION一般不怎么用得到。每一个Schema树的根节点为一个Struct类型，所有的column按照树的中序遍历顺序编号。

ORC只需要存储schema树中叶子节点的值，而中间的非叶子节点只是做一层代理，它们只需要负责孩子节点值得读取，只有真正的叶子节点才会读取数据，然后交由父节点封装成对应的数据结构返回。

文件结构

和Parquet类似，ORC文件也是以二进制方式存储的，所以是不可以直接读取，ORC文件也是自解析的，它包含许多的元数据，这些元数据都是同构ProtoBuffer进行序列化的。ORC的文件结构入图6，其中涉及到如下的概念：

- ORC文件：保存在文件系统上的普通二进制文件，一个ORC文件中可以包含多个stripe，每一个stripe包含多条记录，这些记录按照列进行独立存

储，对应到Parquet中的row group的概念。

- 文件级元数据：包括文件的描述信息PostScript、文件meta信息（包括整个文件的统计信息）、所有stripe的信息和文件schema信息。
- stripe：一组行形成一个stripe，每次读取文件是以行组为单位的，一般为HDFS的块大小，保存了每一列的索引和数据。
- stripe元数据：保存stripe的位置、每一个列的在该stripe的统计信息以及所有的stream类型和位置。
- row group：索引的最小单位，一个stripe中包含多个row group，默认为10000个值组成。
- stream：一个stream表示文件中一段有效的数据，包括索引和数据两类。索引stream保存每一个row group的位置和统计信息，数据stream包括多种类型的数据，具体需要哪几种是由该列类型和编码方式决定。

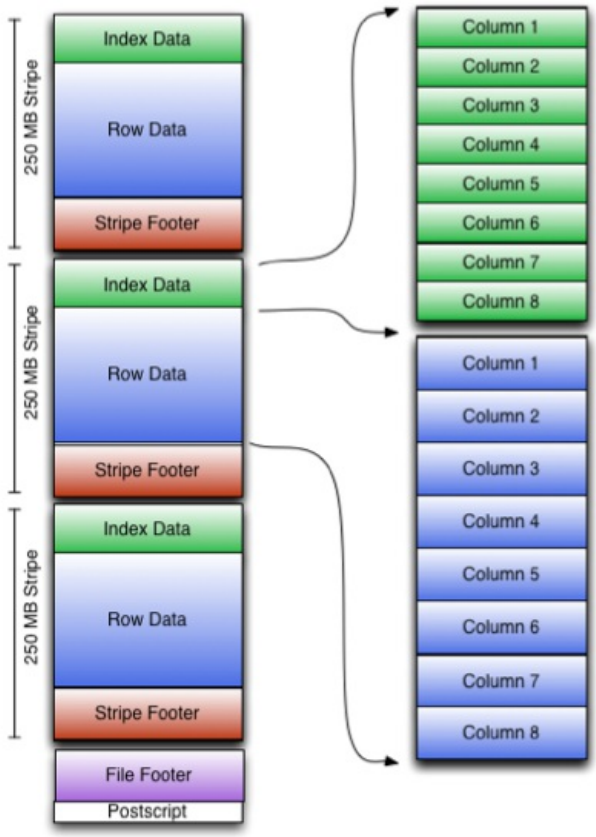


图6 ORC文件结构

在ORC文件中保存了三个层级的统计信息，分别为文件级别、stripe级别和row group级别的，他们都可以用来根据Search ARGuments（谓词下推条件）判断是否可以跳过某些数据，在统计信息中都包含成员数和是否有null值，并且对于不同类型的数据设置一些特定的统计信息。

数据访问

读取ORC文件是从尾部开始的，第一次读取16KB的大小，尽可能的将Postscript和Footer数据都读入内存。文件的最后一个字节保存着PostScript的长度，它的长度不会超过256字节，PostScript中保存着整个文件的元数据信息，它包括文件的压缩格式、文件内部每一个压缩块的最大长度(每次分配内存的大小)、Footer长度，以及一些版本信息。在Postscript和Footer之间存储着整个文件的统计信息(上图中未画出)，这部分的统计信息包括每一个stripe中每一列的信息，主要统计成员数、最大值、最小值、是否有空值等。

接下来读取文件的Footer信息，它包含了每一个stripe的长度和偏移量，该文件的schema信息(将schema树按照schema中的编号保存在数组中)、整个文件的统计信息以及每一个row group的行数。

处理stripe时首先从Footer中获取每一个stripe的其实位置和长度、每一个stripe的Footer数据(元数据，记录了index和data的的长度)，整个striper被分为index和data两部分，stripe内部是按照row group进行分块的(每一个row group中多少条记录在文件的Footer中存储)，row group内部按列存储。每一个row group由多个stream保存数据和索引信息。每一个stream的数据会根据该列的类型使用特定的压缩算法保存。在ORC中存在如下几种stream类型：

- PRESENT：每一个成员值在这个stream中保持一位(bit)用于标示该值是否为NULL，通过它可以只记录部位NULL的值
- DATA：该列的中属于当前stripe的成员值。
- LENGTH：每一个成员的长度，这个是针对string类型的列才有的。
- DICTIONARY_DATA：对string类型数据编码之后字典的内容。
- SECONDARY：存储Decimal、timestamp类型的小数或者纳秒数等。

- ROW_INDEX：保存stripe中每一个row group的统计信息和每一个row group起始位置信息。

在初始化阶段获取全部的元数据之后，可以通过includes数组指定需要读取的列编号，它是一个boolean数组，如果不指定则读取全部的列，还可以通过传递SearchArgument参数指定过滤条件，根据元数据首先读取每一个stripe中的index信息，然后根据index中统计信息以及SearchArgument参数确定需要读取的row group编号，再根据includes数据决定需要从这些row group中读取的列，通过这两层的过滤需要读取的数据只是整个stripe多个小段的区间，然后ORC会尽可能合并多个离散的区间尽可能的减少I/O次数。然后再根据index中保存的下一个row group的位置信息调至该stripe中第一个需要读取的row group中。

由于ORC中使用了更加精确的索引信息，使得在读取数据时可以指定从任意一行开始读取，更细粒度的统计信息使得读取ORC文件跳过整个row group，ORC默认会对任何一块数据和索引信息使用ZLIB压缩，因此ORC文件占用的存储空间也更小，这点在后面的测试对比中也有所印证。

在新版本的ORC中也加入了对Bloom Filter的支持，它可以进一步提升谓词下推的效率，在Hive 1.2.0版本以后也加入了对此的支持。

性能测试

为了对比测试两种存储格式，我选择使用TPC-DS数据集并且对它进行改造以生成宽表、嵌套和多层嵌套的数据。使用最常用的Hive作为SQL引擎进行测试。

测试环境

- Hadoop集群：物理测试集群，四台DataNode/NodeManager机器，每个机器32core+128GB，测试时使用整个集群的资源。
- Hive：Hive 1.2.1版本，使用hiveserver2启动，本机MySql作为元数据库，jdbc方式提交查询SQL
- 数据集：100GB TPC-DS数据集，选取其中的Store_Sales为事实表的模型作为测试数据
- 查询SQL：选择TPC-DS中涉及到上述模型的10条SQL并对其进行改造。

测试场景和结果

整个测试设置了四种场景，每一种场景下对比测试数据占用的存储空间的大小和相同查询执行消耗的时间对比，除了场景一基于原始的TPC-DS数据集外，其余的数据都需要进行数据导入，同时对比这几个场景的数据导入时间。

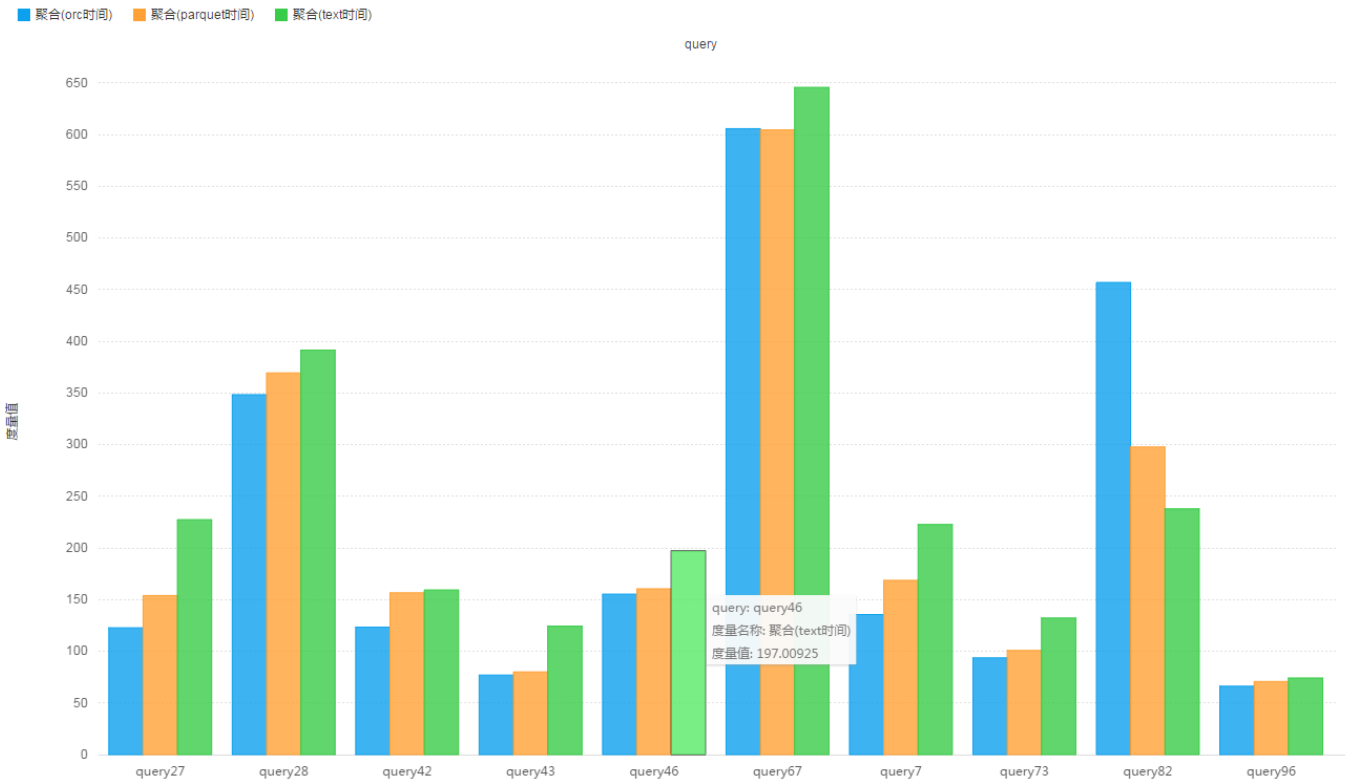
场景一：一个事实表、多个维度表，复杂的join查询。

基于原始的TPC-DS数据集。

Store_Sales表记录数：287,997,024，表大小为：

- 原始Text格式，未压缩 : 38.1 G
- ORC格式，默认压缩（ZLIB）,一共1800+个分区 : 11.5 G
- Parquet格式，默认压缩（Snappy），一共1800+个分区 ： 14.8 G

查询测试结果：



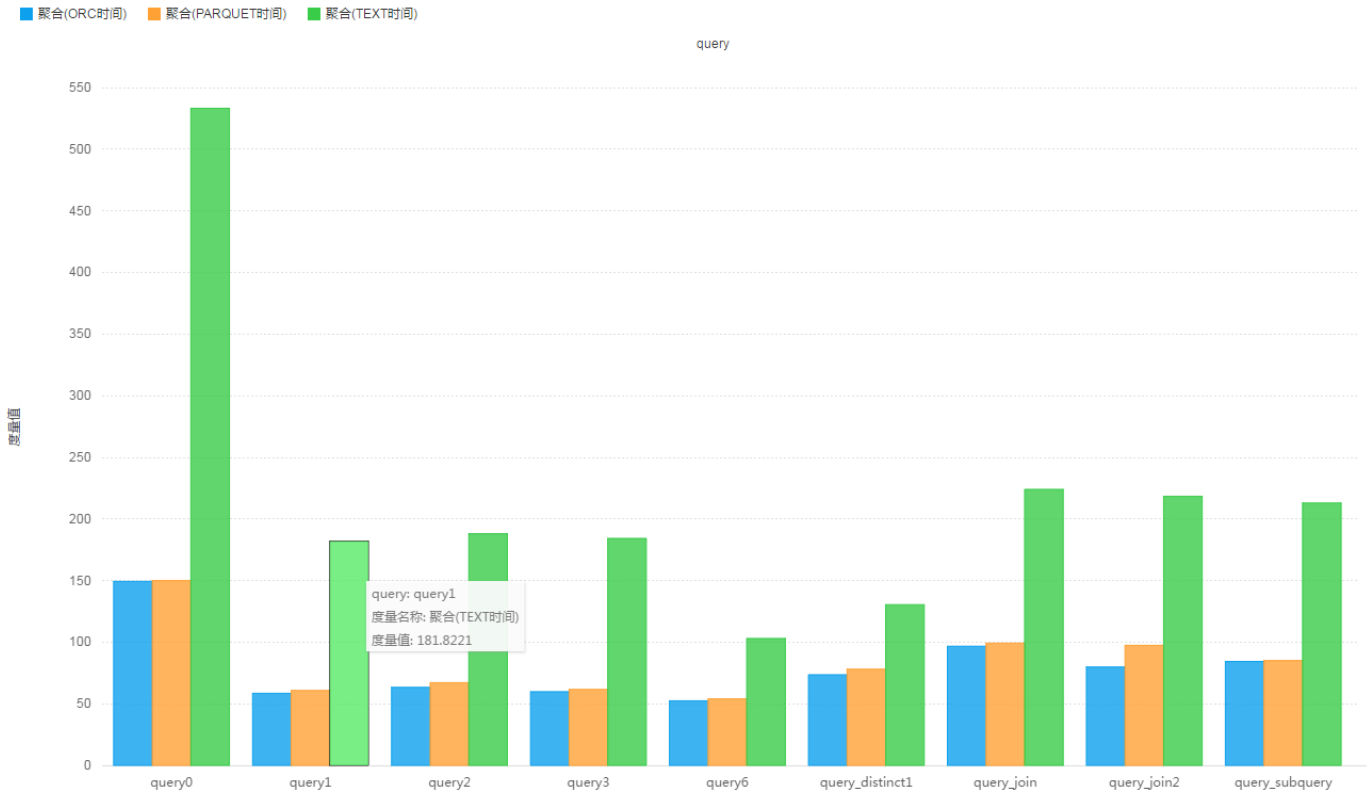
场景二：维度表和事实表join之后生成的宽表，只在一个表上做查询。

整个测试设置了四种场景，每一种场景下对比测试数据占用的存储空间的大小和相同查询执行消耗的时间对比，除了场景一基于原始的TPC-DS数据集外，其余的数据都需要进行数据导入，同时对比这几个场景的数据导入时间。选取数据模型中的store*sales, household*demographics, customer*address, datedim, store表生成一个扁平式宽表(storesales*wide_table)，基于这个表执行查询，由于场景一种选择的query大多数不能完全match到这个宽表，所以对场景1中的SQL进行部分改造。

store*sales*wide_table表记录数：263,704,266，表大小为：

- 原始Text格式，未压缩： 149.0 G
- ORC格式，默认压缩： 10.6 G
- PARQUET格式，默认压缩： 12.5 G

查询测试结果：



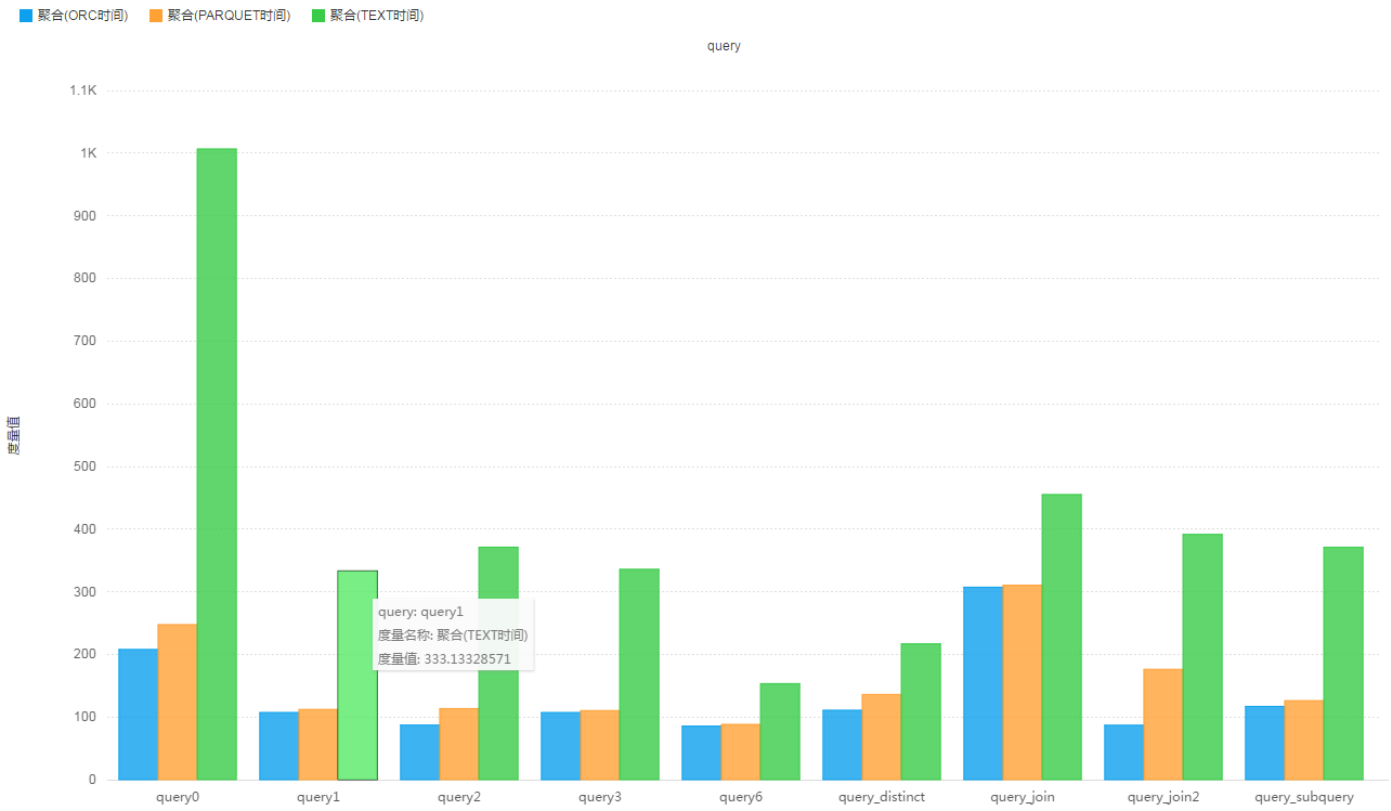
场景三：复杂的数据结构组成的宽表，struct、list、map等（1层）

整个测试设置了四种场景，每一种场景下对比测试数据占用的存储空间的大小和相同查询执行消耗的时间对比，除了场景一基于原始的TPC-DS数据集外，其余的数据都需要进行数据导入，同时对比这几个场景的数据导入时间。在场景二的基础上，将维度表（除了store*sales表）转换成一个struct或者map对象，源store*sales表中的字段保持不变。生成有一层嵌套的新表（store*saleswidetable*one_nested），使用的查询逻辑相同。

store*sales*wide_table*one*nested表记录数：263,704,266，表大小为：

- 原始Text格式，未压缩： 245.3 G
- ORC格式，默认压缩： 10.9 G 比store_sales表还小？
- PARQUET格式，默认压缩： 29.8 G

查询测试结果：



场景四：复杂的数据结构，多层嵌套。（3层）

整个测试设置了四种场景，每一种场景下对比测试数据占用的存储空间的大小和相同查询执行消耗的时间对比，除了场景一基于原始的TPC-DS数据集外，其余的数据都需要进行数据导入，同时对比这几个场景的数据导入时间。在场景三的基础上，将部分维度表的struct内的字段再转换成struct或者map对象，只存在struct中嵌套map的情况，最深的嵌套为三层。生成一个多层嵌套的新表（store*sales*wide_table*more*nested），使用的查询逻辑相同。

该场景中只涉及一个多层嵌套的宽表，没有任何分区字段，store*sales*wide_table*more*nested表记录数：263,704,266，表大小为：

- 原始Text格式，未压缩： 222.7 G
- ORC格式，默认压缩： 10.9 G 比store_sales表还小？
- PARQUET格式，默认压缩： 23.1 G 比一层嵌套表store*sales*wide_table*one*nested要小？

查询测试结果：

结果分析

从上述测试结果来看，星状模型对于数据分析场景并不是很合适，多个表的join会大大拖慢查询速度，并且不能很好的利用列式存储带来的性能提升，在使用宽表的情况下，列式存储的性能提升明显，ORC文件格式在存储空间上要远优于Text格式，较之于PARQUET格式有一倍的存储空间提升，在导数据（insert into table select 这样的方式）方面ORC格式也要优于PARQUET，在最终的查询性能上可以看到，无论是无嵌套的扁平式宽表，或是一层嵌套表，还是多层嵌套的宽表，两者的查询性能相差不多，较之于Text格式有2到3倍左右的提升。

另外，通过对比场景二和场景三的测试结果，可以发现扁平式的表结构要比嵌套式结构的查询性能有所提升，所以如果选择使用大宽表，则设计宽表的时候尽可能的将表设计的扁平化，减少嵌套数据。

通过这三种文件存储格式的测试对比，ORC文件存储格式无论是在空间存储、导数据速度还是查询速度上表现的都较好一些，并且ORC可以一定程度上支持ACID操作，社区的发展目前也是Hive中比较提倡使用的一种列式存储格式，另外，本次测试主要针对的是Hive引擎，所以不排除存在Hive与ORC的敏感度比PARQUET要高的可能性。

总结

本文主要从数据模型、文件格式和数据访问流程等几个方面详细介绍了Hadoop生态圈中的两种列式存储格式——Parquet和ORC，并通过大数据量的测试对两者的存储和查询性能进行了对比。对于大数据场景下的数据分析需求，使用这两种存储格式总会带来存储和性能上的提升，但是在实际使用时还需要针对实际的数据进行选择。另外由于不同开源产品可能对不同的存储格式有特定的优化，所以选择时还需要考虑查询引擎的因素。