新一代列式存储格式Parquet

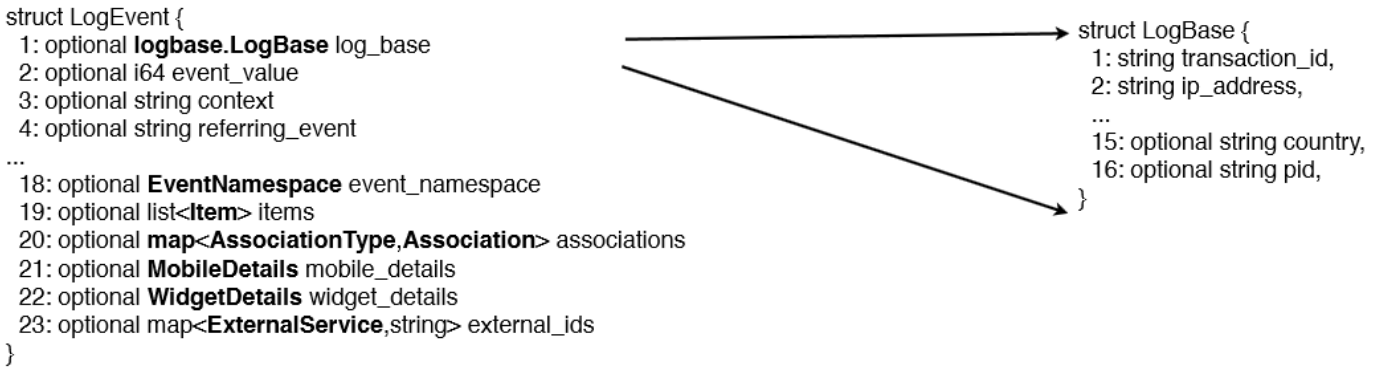
Apache Parquet是Hadoop生态圈中一种新型列式存储格式，它可以兼容Hadoop生态圈中大多数计算框架(Hadoop、Spark等)，被多种查询引擎支持（Hive、Impala、Drill等），并且它是语言和平台无关的。Parquet最初是由Twitter和Cloudera(由于Impala的缘故)合作开发完成并开源，2015年5月从Apache的孵化器里毕业成为Apache顶级项目，最新的版本是1.8.1。

Parquet是什么

Parquet的灵感来自于2010年Google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Apache Drill。

嵌套数据模型

在接触大数据之前，我们简单的将数据划分为结构化数据和非结构化数据，通常我们使用关系数据库存储结构化数据，而关系数据库中使用数据模型都是扁平式的，遇到诸如List、Map和自定义Struct的时候就需要用户在应用层解析。但是在大数据环境下，通常数据的来源是服务端的埋点数据，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，这样在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，在他们的生产环境中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:



另外，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，这样也就使得诸如Parquet这种原生支持嵌套数据类型的存储格式也变得至关重要，性能也会更好。

列式存储

列式存储，顾名思义就是按照列进行存储数据，把某一列的数据连续的存储，每一行中的不同列的值离散分布。列式存储技术并不新鲜，在关系数据库中都已经在使用，尤其是在针对OLAP场景下的数据存储，由于OLAP场景下的数据大部分情况下都是批量导入，基本上不需要支持单条记录的增删改操作，而查询的时候大多数都是只使用部分列进行过滤、聚合，对少数列进行计算（基本不需要select * from xx之类的查询）。列式存储可以大大提升这类查询的性能，较之于行是存储，列式存储能够带来这些优化：

- 由于每一列中的数据类型相同，所以可以针对不同类型的列使用不同的编码和压缩方式，这样可以大大降低数据存储空间。
- 读取数据的时候可以把映射(Project)下推，只需要读取需要的列，这样可以大大减少每次查询的I/O数据量，更甚至可以支持谓词下推，跳过不满足条件的列。
- 由于每一列的数据类型相同，可以使用更加适合CPU pipeline的编码方式，减小CPU的缓存失效。

Parquet的组成

Parquet仅仅是一种存储格式，它是语言、平台无关的，并且不需要和任何一种数据处理框架绑定，目前能够和Parquet适配的组件包括下面这些，可以看出基本上通常使用的查询引擎和计算框架都已适配，并且可以很方便的将其它序列化工具生成的数据转换成Parquet格式。

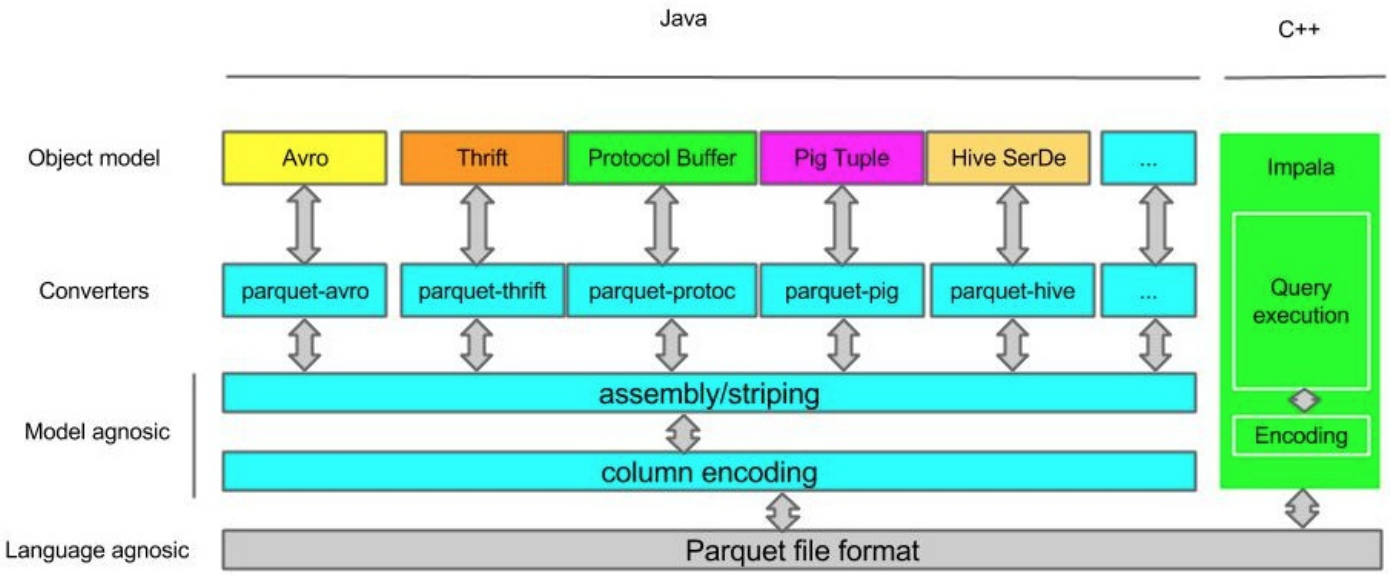
- 查询引擎: Hive, Impala, Pig, Presto, Drill, Tajo, HAWQ, IBM Big SQL
- 计算框架: MapReduce, Spark, Cascading, Crunch, Scalding, Kite
- 数据模型: Avro, Thrift, Protocol Buffers, POJOs

项目组成

Parquet项目由以下几个子项目组成:

- parquet-format项目由java实现，它定义了所有Parquet元数据对象，Parquet的元数据是使用Apache Thrift进行序列化并存储在Parquet文件的尾部。
- parquet-format项目由java实现，它包括多个模块，包括实现了读写Parquet文件的功能，并且提供一些和其它组件适配的工具，例如Hadoop Input/Output Formats、Hive Serde(目前Hive已经自带Parquet了)、Pig loaders等。
- parquet-compatibility项目，包含不同编程语言之间(JAVA和C/C++)读写文件的测试代码。
- parquet-cpp项目，它是用于用于读写Parquet文件的C++库。

下图展示了Parquet各个组件的层次以及从上到下交互的方式。



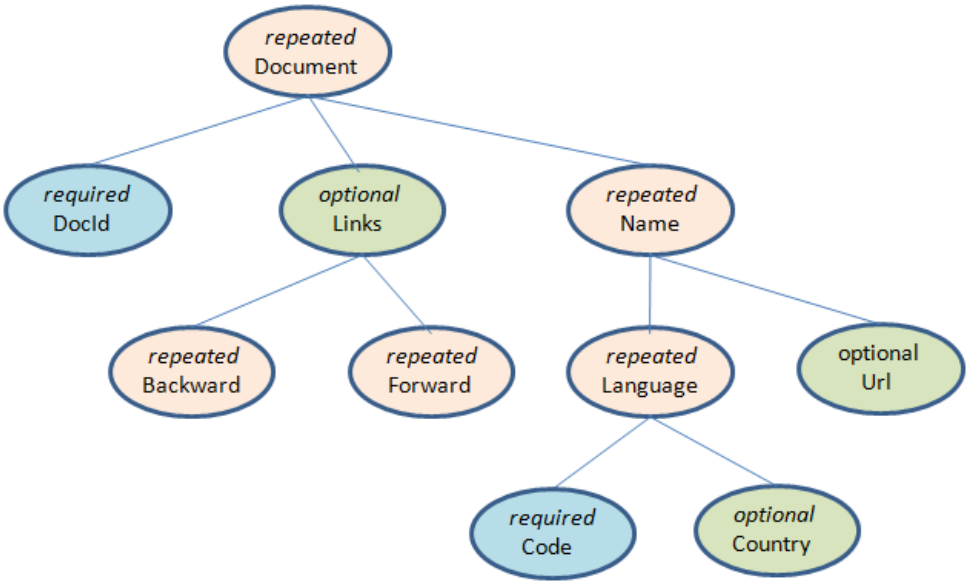
- 数据存储层定义了Parquet的文件格式，其中元数据在parquet-format中定义，包括Parquet原始类型定义、Page类型、编码类型、压缩类型等等。
- 对象转换层完成其他对象模型与Parquet内部数据模型的映射和转换，Parquet的编码方式使用的是striping and assembly算法。
- 对象模型层定义了如何读取Parquet文件的内容，这一层转换包括Avro、Thrift、PB等序列化格式、Hive serde等的适配。并且为了帮助大家理解和使用，Parquet提供了org.apache.parquet.example包实现了java对象和Parquet文件的转换。

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，每一个字段又可以包含多个字段，每一个字段有三个属性：重复数、数据类型和字段名，重复数可以是以下三种：required(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，如下图:



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组合来实现这样的需求。在这个包含6个字段的表中有以下几个字段和每一条记录中它们可能出现的次数：

```
DocId                int64    只能出现一次
Links.Backward       int64    可能出现任意多次，但是如果出现0次则需要使用NULL标识
Links.Forward        int64    同上
Name.Language.Code   string   同上
Name.Language.Country string   同上
Name.Url             string   同上
```

由于在一个表中可能存在出现任意多次的列，对于这些列需要标示出现多次或者等于NULL的情况，它是由Striping/Assembly算法实现的。

Striping/Assembly算法

上文介绍了Parquet的数据模型，在Document中存在多个非required列，由于Parquet一条记录的数据分散的存储在不同的列中，如何组合不同的列值组成一条记录是由Striping/Assembly算法决定的，在该算法中列的每一个值都包含三部分：value、repetition level和definition level。

Repetition Levels

为了支持repeated类型的节点，在写入的时候该值等于它和前面的值在哪一层节点是不共享的。在读取的时候根据该值可以推导出哪一层上需要创建一个新的节点，例如对于这样的一个schema和两条记录。

```
message nested {
     repeated group leve1 {
          repeated string leve2;
     }
}
r1:[[a,b,c,] , [d,e,f,g]]
r2:[[h] , [i,j]]
```

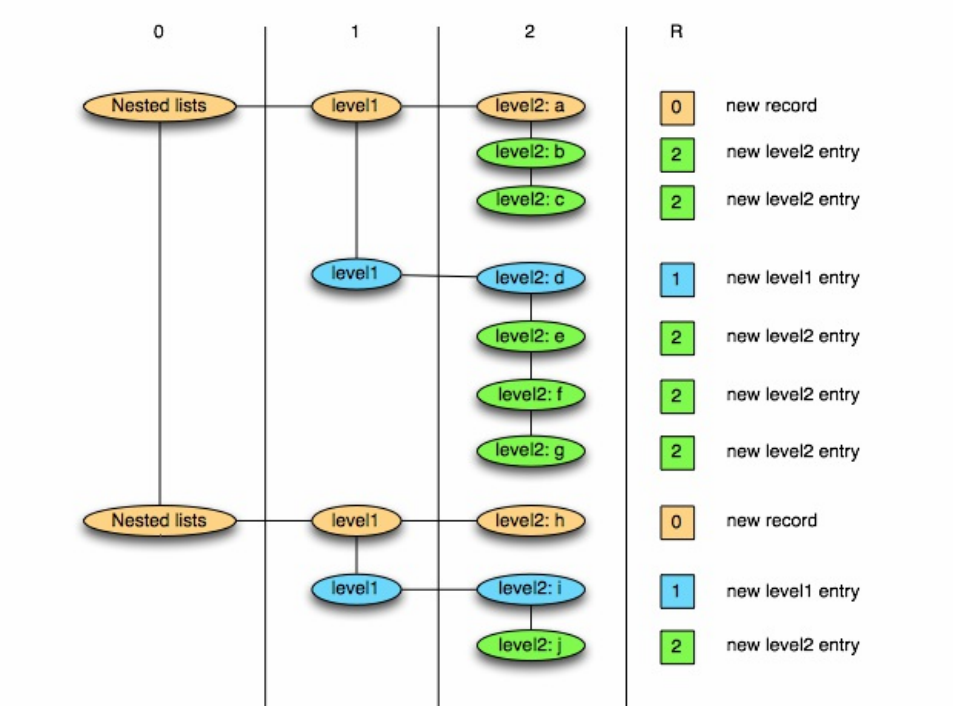
计算repetition level值的过程如下：

- value=a是一条记录的开始，和前面的值(已经没有值了)在根节点(第0层)上是不共享的，所以repeated level=0.
- value=b它和前面的值共享了level1这个节点，但是level2这个节点上是不共享的，所以repeated level=2.
- 同理value=c, repeated level=2.
- value=d和前面的值共享了根节点(属于相同记录)，但是在level1这个节点上是不共享的，所以repeated level=1.
- value=h和前面的值不属于同一条记录，也就是不共享任何节点，所以repeated level=0.

根据以上的分析每一个value需要记录的repeated level值如下：

Repetition level	Value
0	a
2	b
2	c
1	d
2	e
2	f
2	g
0	h
1	i
2	j

在读取的时候，顺序的读取每一个值，然后根据它的repeated level创建对象，当读取value=a时repeated level=0，表示需要创建一个新的根节点(新记录)，value=b时repeated level=2，表示需要创建一个新的level2节点，value=d时repeated level=1，表示需要创建一个新的level1节点，当所有列读取完成之后可以创建一条新的记录。本例中当读取文件构建每条记录的结果如下：



可以看出repeated level=0表示一条记录的开始，并且repeated level的值只是针对路径上的repeated类型的节点，因此在计算该值的时候可以忽略非repeated类型的节点，在写入的时候将其理解为该节点和路径上的哪一个repeated节点是不共享的，读取的时候将其理解为需要在哪一层创建一个新的repeated节点，这样的话每一列最大的repeated level值就等于路径上的repeated节点的个数（不包括根节点）。减小repeated level的好处能够使得在存储使用更加紧凑的编码方式，节省存储空间。

Definition Levels

有了repeated level我们就可以构造出一个记录了，为什么还需要definition levels呢？由于repeated和optional类型的存在，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。

definition level的值仅仅对于空值是有效的，表示在该值的路径上第几层开始是未定义的，对于非空的值它是没有意义的，因为非空值在叶子节点是定义的，所有的父节点也肯定是定义的，因此它总是等于该列最大的definition levels。例如下面的schema。

```
message ExampleDefinitionLevel {
  optional group a {
    optional group b {
      optional string c;
    }
  }
}
```

它包含一个列a.b.c，这个列的的每一个节点都是optional类型的，当c被定义时a和b肯定都是已定义的，当c未定义时我们就需要标示出在从哪一层开始

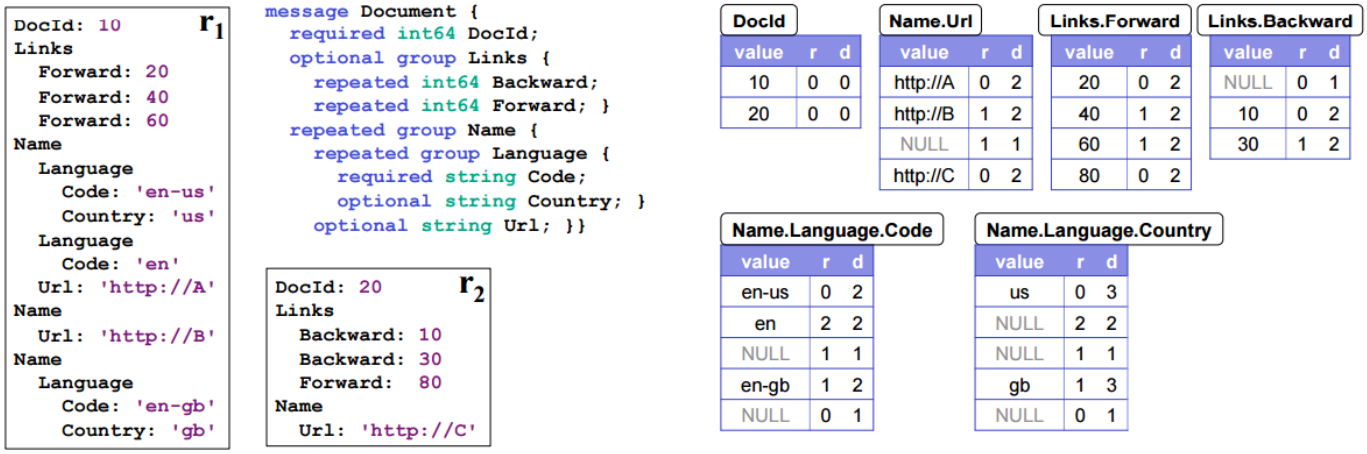
时未定义的，如下面的值：

Value	Definition Level
a: null	0
a: { b: null }	1
a: { b: { c: null } }	2
a: { b: { c: "foo" } }	3 (actually defined)

由于definition level只需要考虑未定义的值，而对于repeated类型的节点，只要父节点是已定义的，该节点就必须定义（例如Document中的DocId，每一条记录都该列都必须有值，同样对于Language节点，只要它定义了Code必须有值），所以计算definition level的值时可以忽略路径上的required节点，这样可以减小definition level的最大值，优化存储。

一个完整的例子

本节我们使用Dremel论文中给的Document示例和给定的两个值r1和r2展示计算repeated level和definition level的过程，这里把未定义的值记录为NULL，使用R表示repeated level，D表示definition level。



首先看DocuId这一列，对于r1，DocId=10，由于它是记录的开始并且是已定义的，所以R=0，D=0，同样r2中的DocId=20，R=0，D=0。

对于Links.Forward这一列，在r1中，它是未定义的但是Links是已定义的，并且是该记录中的第一个值，所以R=0，D=1，在r1中该列有两个值，value1=10，R=0(记录中该列的第一个值)，D=2(该列的最大definition level)。

对于Name.Url这一列，r1中它有三个值，分别为url1='http://A'，它是r1中该列的第一个值并且是定义的，所以R=0，D=2；value2='http://B'，和上一个值value1在Name这一层是不相同的，所以R=1，D=2；value3=NULL，和上一个值value2在Name这一层是不相同的，所以R=1，但它是未定义的，而Name这一层是定义的，所以D=1。r2中该列只有一个值value3='http://C'，R=0，D=2.

最后看一下Name.Language.Code这一列，r1中有4个值，value1='en-us'，它是r1中的第一个值并且是已定义的，所以R=0，D=2(由于Code是required类型，这一列repeated level的最大值等于2)；value2='en'，它和value1在Language这个节点是不共享的，所以R=2，D=2；value3=NULL，它是未定义的，但是它和前一个值在Name这个节点是不共享的，在Name这个节点是已定义的，所以R=1，D=1；value4='en-gb'，它和前一个值在Name这一层不共享，所以R=1，D=2。在r2中该列有一个值，它是未定义的，但是Name这一层是已定义的，所以R=0，D=1.

Parquet文件格式

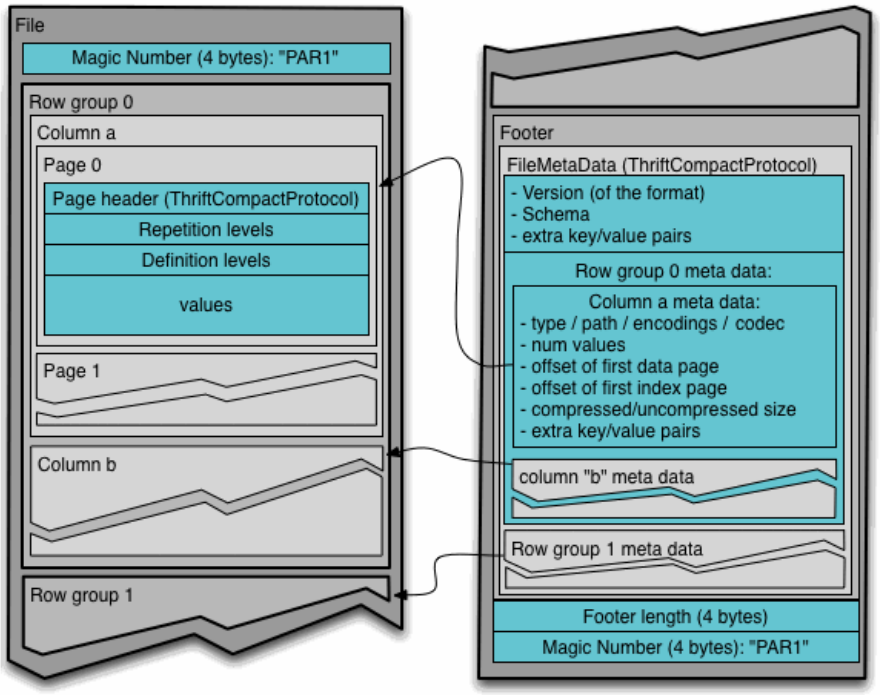
Parquet文件是以二进制方式存储的，所以是不可以直接读取的，文件中包括该文件的数据和元数据，因此Parquet格式文件是自解析的。在HDFS文件系统和Parquet文件中存在如下几个概念。

- 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，例如记录占用空间比较小的Schema可以在每一个行组中存储更多的行。

- 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。一个列块中的值都是相同类型的，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。

文件格式

通常情况下，在存储Parquet数据的时候会按照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。



上图展示了一个Parquet文件的内容，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该文件存储数据的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。

在执行MR任务的时候可能存在多个Mapper任务的输入是同一个Parquet文件的情况，每一个Mapper通过InputSplit标示处理的文件范围，如果多个InputSplit跨越了一个Row Group，Parquet能够保证一个Row Group只会被一个Mapper任务处理。

映射下推(Project PushDown)

说到列式存储的优势，映射下推是最突出的，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，所以分区取出每一列的所有值就可以实现TableScan算子，而避免扫描整个表文件内容。

在Parquet中原生就支持映射下推，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，映射每次会扫描一个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。

谓词下推(Predicate PushDown)

在数据库之类的查询系统中最常用的优化手段就是谓词下推了，通过将一些过滤条件尽可能的在最底层执行可以减少每一层交互的数据量，从而提升性能，例如"select count(1) from A Join B on A.id = B.id where A.a > 10 and B.b < 100"SQL查询中，在处理Join操作之前需要首先对A和B执行TableScan操作，然后再进行Join，再执行过滤，最后计算聚合函数返回，但是如果把过滤条件A.a > 10和B.b < 100分别移到A表的TableScan和B表的TableScan的时候执行，可以大大降低Join操作的输入数据。

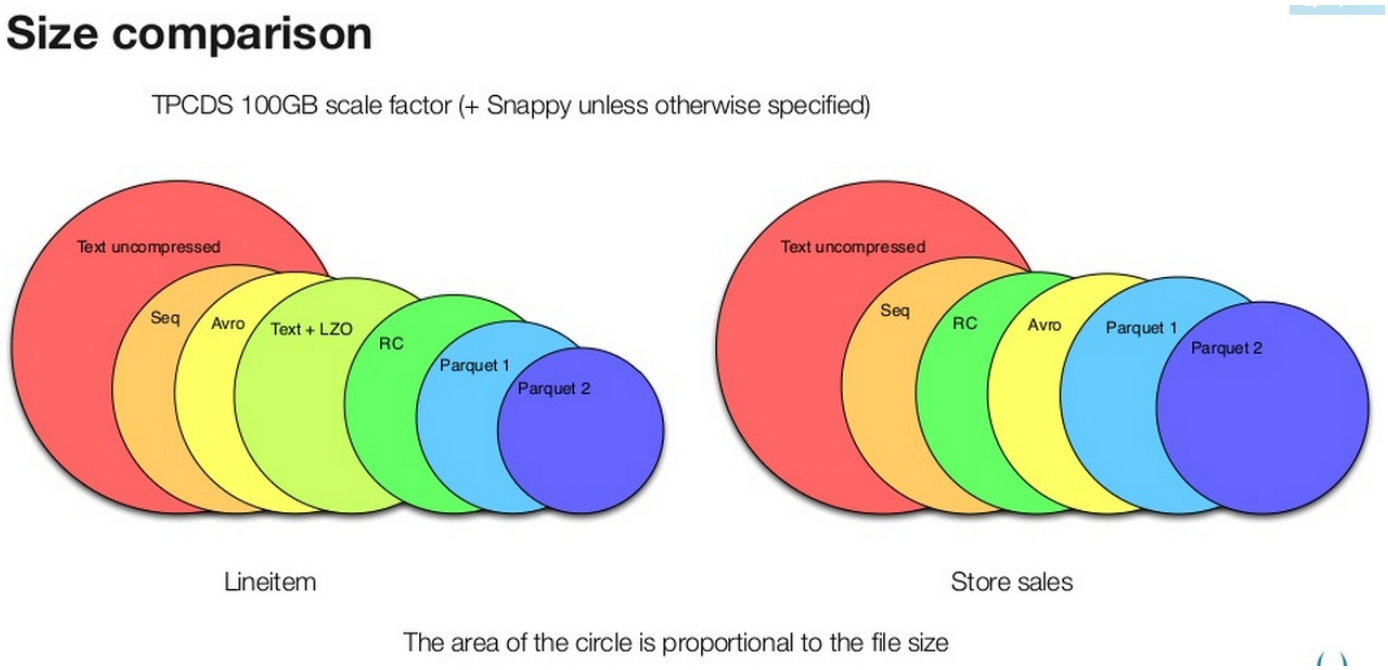
无论是行式存储还是列式存储，都可以在将过滤条件在读取一条记录之后执行以判断该记录是否需要返回给调用者，在Parquet做了更进一步的优化，优化的方法时对每一个Row Group的每一个Column Chunk在存储的时候都计算对应的统计信息，包括该Column Chunk的最大值、最小值和空值个

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。

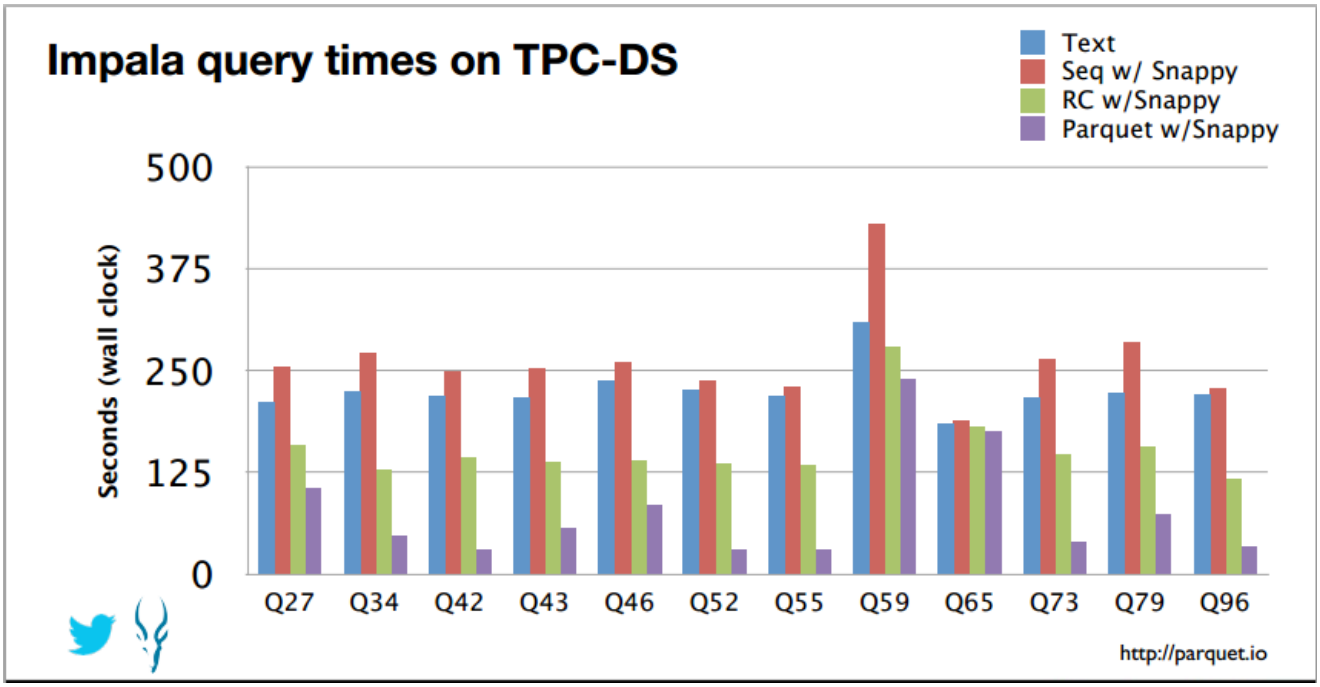
在使用Parquet的时候可以通过如下两种策略提升查询性能：1、类似于关系数据库的主键，对需要频繁过滤的列设置为有序的，这样在导入数据的时候会根据该列的顺序存储数据，这样可以最大化的利用最大值、最小值实现谓词下推。2、减小行组大小和页大小，这样增加跳过整个行组的可能性，但是此时需要权衡由于压缩和编码效率下降带来的I/O负载。

性能

相比传统的行式存储，Hadoop生态圈近年来也涌现出诸如RC、ORC、Parquet的列式存储格式，它们的性能优势主要体现在两个方面：1、更高的压缩比，由于相同类型的数据更容易针对不同类型的列使用高效的编码和压缩方式。2、更小的I/O操作，由于映射下推和谓词下推的使用，可以减少一大部分不必要的数据扫描，尤其是表结构比较庞大的时候更加明显，由此也能够带来更好的查询性能。

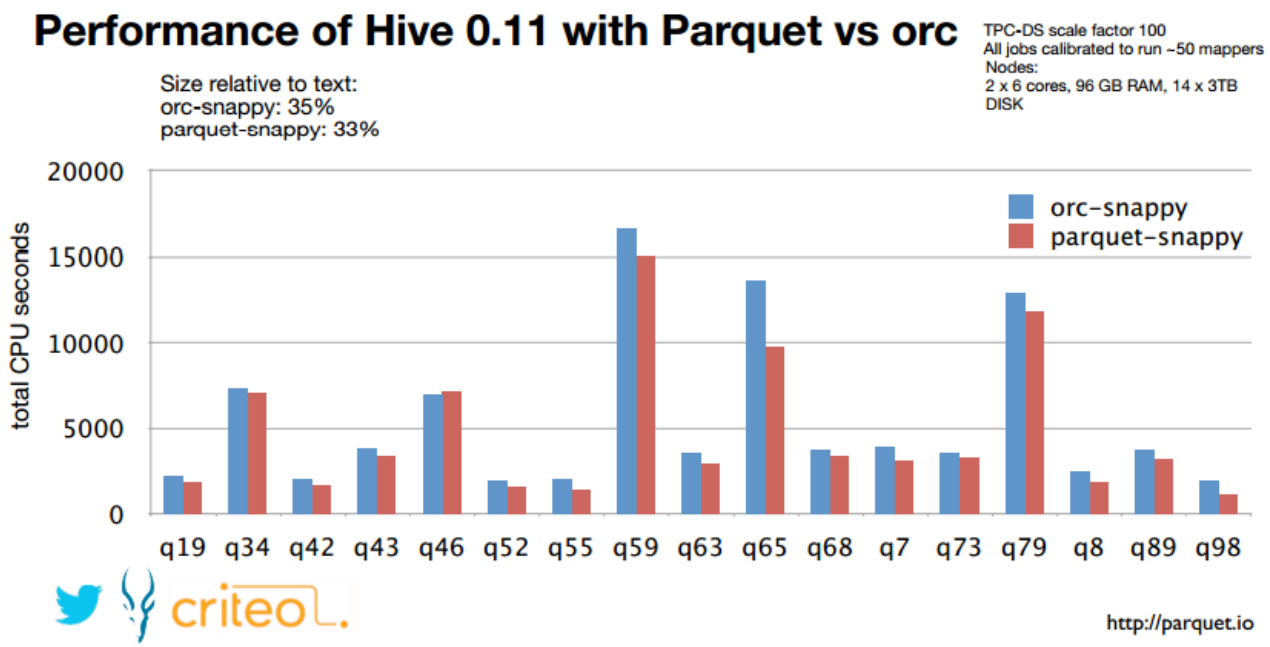


上图是展示了使用不同格式存储TPC-H和TPC-DS数据集中两个表数据的文件大小对比，可以看出Parquet较之于其他的二进制文件存储格式能够更有效的利用存储空间，而新版本的Parquet(2.0版本)使用了更加高效的页存储方式，进一步的提升存储空间。



上图展示了Twitter在Impala中使用不同格式文件执行TPC-DS基准测试的结果，测试结果可以看出Parquet较之于其他的行式存储格式有较明显的性能

提升。



上图展示了criteo公司在Hive中使用ORC和Parquet两种列式存储格式执行TPC-DS基准测试的结果，测试结果可以看出在数据存储方面，两种存储格式在都是用snappy压缩的情况下量中存储格式占用的空间相差并不大，查询的结果显示Parquet格式稍好于ORC格式，两者在功能上也都有优缺点，Parquet原生支持嵌套式数据结构，而ORC对此支持的较差，这种复杂的Schema查询也相对较差；而Parquet不支持数据的修改和ACID，但是ORC对此提供支持，但是在OLAP环境下很少会对单条数据修改，更多的则是批量导入。

项目发展

自从2012年由Twitter和Cloudera共同研发Parquet开始，该项目一直处于高速发展之中，并且在项目之初就将其贡献给开源社区，2013年，Criteo公司加入开发并且向Hive社区提交了向hive集成Parquet的patch(HIVE-5783)，在Hive 0.13版本之后正式加入了Parquet的支持；之后越来越多的查询引擎对此进行支持，也进一步带动了Parquet的发展。

目前Parquet正处于向2.0版本迈进的阶段，在新的版本中实现了新的Page存储格式，针对不同的类型优化编码算法，另外丰富了支持的原始类型，增加了Decimal、Timestamp等类型的支持，增加更加丰富的统计信息，例如Bloon Filter，能够尽可能得将谓词下推在元数据层完成。

总结

本文介绍了一种支持嵌套数据模型对的列式存储系统Parquet，作为大数据系统中OLAP查询的优化方案，它已经被多种查询引擎原生支持，并且部分高性能引擎将其作为默认的文件存储格式。通过数据编码和压缩，以及映射下推和谓词下推功能，Parquet的性能也较之其它文件格式有所提升，可以预见，随着数据模型的丰富和Ad hoc查询的需求，Parquet将会被更广泛的使用。

参考

1. Dremel: Interactive Analysis of Web-Scale Datasets
2. Dremel made simple with Parquet
3. Parquet: Columnar storage for the people
4. Efficient Data Storage for Analytics with Apache Parquet 2.0
5. 深入分析Parquet列式存储格式
6. Apache Parquet Document