hadoop数据格式&&压缩方式

数据存储方式	压缩方式
textfile	
SequenceFile	NONE, RECORD, BLOCK。 Record压缩率低，一般建议使用BLOCK压缩
Orc	
PARQUET	
AVRO	
RCFILE	

textFile

普通的文本格式文件，还有其他文本格式如json，xml等

缺点：

无压缩，磁盘开销大：
可结合Gzip、Bzip2使用（系统自动检查，执行查询时自动解压），但使用这种方式，hive不会对数据进行切分，从而无法对数据进行并行操作。

解析不便：对文本文件的解析开销一般会比二进制格式高几十倍以上，尤其是XML 和JSON，它们的解析开销比Textfile 还要大

不具备类型和模式：比如销售金额、利润这类数值数据或者日期时间类型的数据，字符串长短不一，使用mr处理时无法排序，往往需要转换为有模式的二进制文件。

SequenceFile

SequenceFile文件是Hadoop用来存储**二进制形式**的**key-value**对而设计的一种平面文件(Flat File)。
这种二进制文件直接将<key, value>对序列化到文件中。
SequenceFile文件不保证其存储的key-value数据是按照key的某个顺序存储的，同时不支持append操作。

优点：

1)支持压缩，且可定制为基于Record或Block压缩（Block级压缩性能较优）
2)本地化任务支持：因为文件可以被切分，因此MapReduce任务时数据的本地化情况应该是非常好的。
3)难度低：因为是Hadoop框架提供的API，业务逻辑侧的修改比较简单。

缺点：

是需要一个合并文件的过程，且合并后的文件将不方便查看。

适用场景：

小文件合并。将小文件文件名作为key，文件内容作为value序列化到大文件中。
指定为hive存储格式。
如果外部表使用，需要先把数据导入textfile表，再把该表的数据导入到sequencefile的表（store as seq...）。

hive中使用注意：Hive 中的SequenceFile 继承自Hadoop API 的SequenceFile，不过它的key为空，使用value 存放实际的值， 这样是为了避免MR 在运行map 阶段的排序过程。如果你用Java API 编写SequenceFile，并让Hive 读取的话，请确保使用value字段存放数据，否则你需要自定义读取这种SequenceFile 的InputFormat class 和OutputFormat class。

底层原理：

在SequenceFile文件中，每一个key-value被看做是一条记录(**Record**)，因此基于Record的压缩策略，SequenceFile文件支持三种压缩类型(SequenceFile.CompressionType):

NONE: 对records不进行压缩;

RECORD: 仅压缩每一个record中的**value值;**

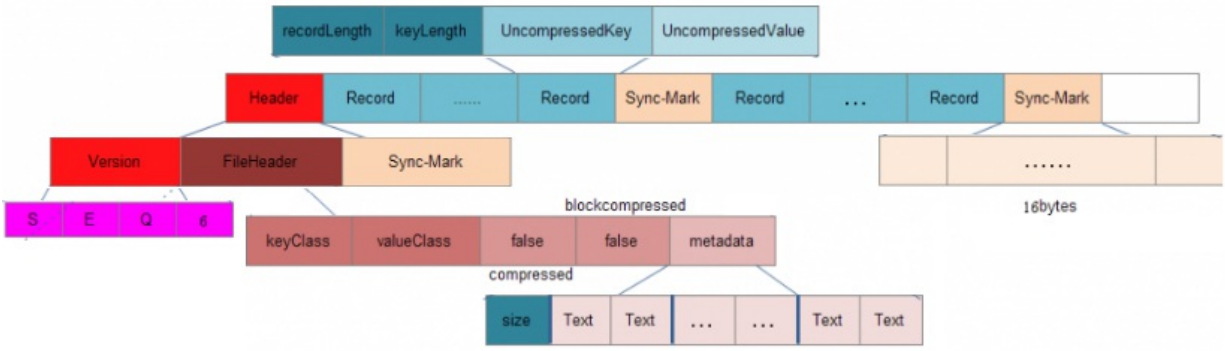
BLOCK: 将一个block中的所有records压缩在一起;

所以，有对应的三个writer类来对应压缩类型写入数据。

其中，第三种block的大小compressionBlockSize默认值为1000000，可通过配置参数io.seqfile.compress.blocksize指定。

sequenceFile的格式（根据压缩情况分为三种）

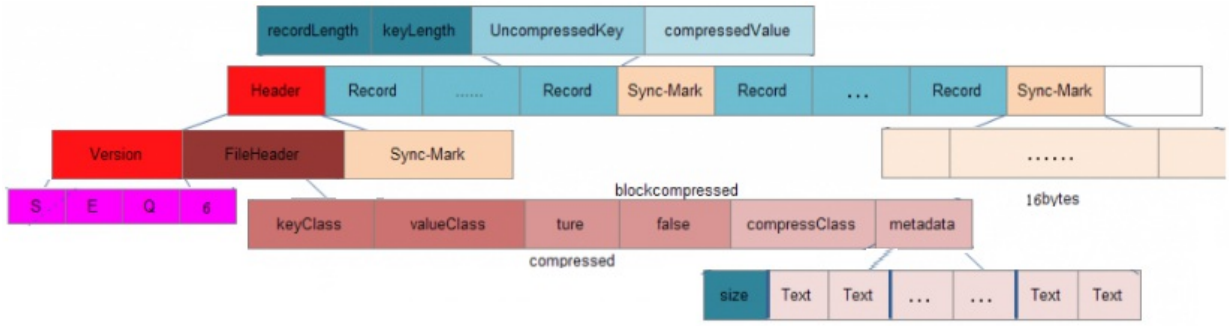
无压缩格式：



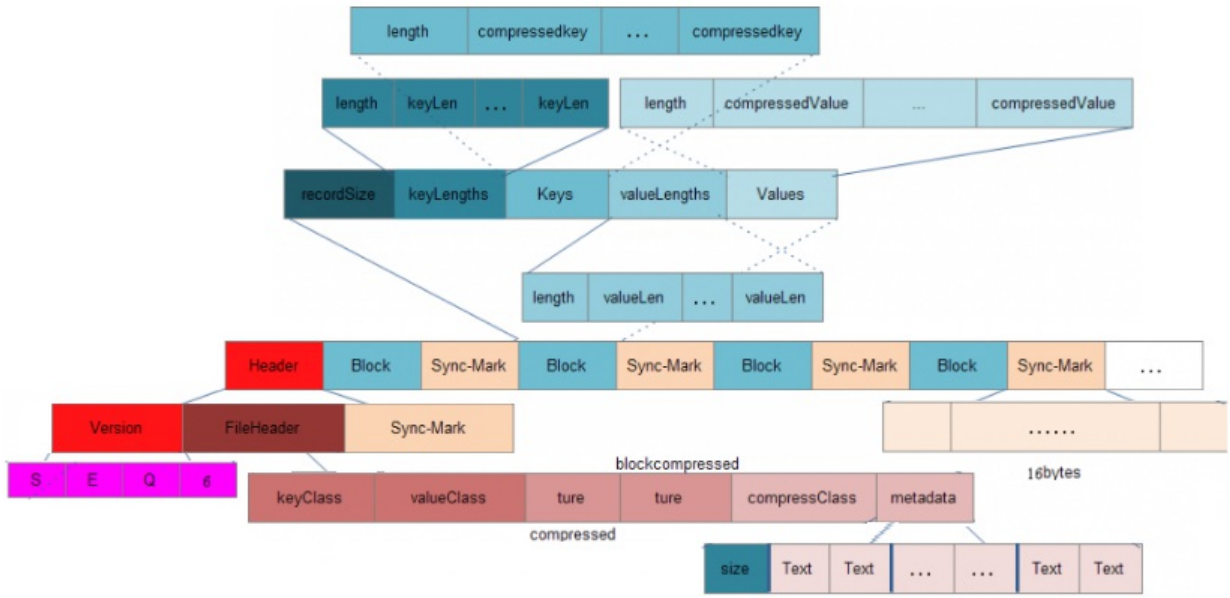
分为表头Header，Record块，达到一定大小（2000byte）就会写一个sync

其中表头Header包括：版本信息，文件头和同步sysnc，其他文件头包括，key和value的Class，是否压缩以及是否块压缩，和元数据。
Record包括：记录的长度、Key的长度、Key值和Value值

Record压缩格式：



Block压缩格式：



RcFile

RCFILE是一种行列存储相结合的存储方式。首先，其将数据按行分块，保证同一个record在一个块上，避免读一个记录需要读取多个block。其次，块数据列式存储，有利于数据压缩和快速的列存取。

RCFile（Record Columnar File）存储结构遵循的是“先水平划分，再垂直划分”的设计理念

RCFile存储的表是水平划分的，分为多个行组，每个行组再被垂直划分，以便每列单独存储；

RCFile在每个行组中利用一个列维度的数据压缩，并提供一种Lazy解压（decompression）技术来在查询执行时避免不必要的列解压；

RCFile支持弹性的行组大小，行组大小需要权衡数据压缩性能和查询性能两方面。

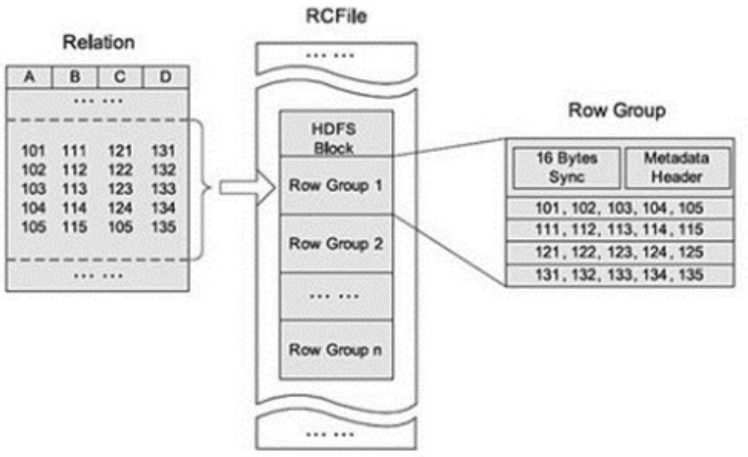
优点：结合了行存储和列存储的优点

1，RCFile保证同一行的数据位于同一节点，因此元组重构的开销很低；

2，像列存储一样，RCFile能够利用列维度的数据压缩，并且能跳过不必要的列读取。

首先，RCFile具备相当于行存储的数据加载速度和负载适应能力；其次，RCFile的读优化可以在扫描表格时避免不必要的列读取，测试显示在多数情况下，它比其他结构拥有更好的性能；再次，RCFile使用列维度的压缩，因此能够有效提升存储空间利用率。

比较：相比TEXTFILE和SEQUENCEFILE，RCFILE由于列式存储方式，数据加载时性能消耗较大，但是具有较好的压缩比和查询响应。数据仓库的特点是一次写入、多次读取，因此，整体来看，RCFILE相比其余两种格式具有较明显的优势。



数据格式

RCFile基于HDFS架构，表格占用多个HDFS块。

每个HDFS块中，RCFile以行组为基本单位来组织记录。也就是说，存储在一个HDFS块中的所有记录被划分为多个行组。对于一张表，所有行组大小都相同。一个HDFS块会有一个或多个行组。

一个行组包括三个部分。第一部分是行组头部的同步标识，主要用于分隔HDFS块中的两个连续行组；第二部分是行组的元数据头部，用于存储行组单元的信息，包括行组中的记录数、每个列的字节数、列中每个域的字节数；第三部分是表格数据段，即实际的列存储数据。在该部分中，同一列的所有域顺序存储。从图可以看出，首先存储了列A的所有域，然后存储列B的所有域等。

压缩方式

RCFile的每个行组中，元数据头部和表格数据段分别进行压缩。

对于所有元数据头部，RCFile使用RLE（Run Length Encoding）算法来压缩数据。由于同一列中所有域的长度值都顺序存储在该部分，RLE算法能够找到重复值的长序列，尤其对于固定的域长度。

表格数据段不会作为整个单元来压缩；相反每个列被独立压缩，使用Gzip压缩算法（RCFile对表格数据的所有列使用同样的压缩算法）。RCFile使用重量级的Gzip压缩算法，是为了获得较好的压缩比，而不使用RLE算法的原因在于此时列数据非排序。此外，由于Lazy压缩策略，当处理一个行组时，RCFile不需要解压所有列。因此，相对较高的Gzip解压开销可以减少。

数据追加

RCFile不支持任意方式的数据写操作，仅提供一种追加接口，类似于hdfs的写文件。

RCFile为每列创建并维护一个内存column holder，当记录追加时，所有域被分发，每个域追加到其对应的column holder。此外，RCFile在元数据头部中记录每个域对应的元数据。

RCFile提供两个参数控制在刷写到磁盘之前，内存中缓存多少记录。一个参数是记录数的限制，另一个是内存缓存的大小限制。

RCFile首先压缩元数据头部并写到磁盘，然后分别压缩每个column holder，并将压缩后的column holder刷写到底层文件系统中的一个

行组中。

数据读取和Lazy解压

在MapReduce框架中，mapper将顺序处理HDFS块中的每个行组。当处理一个行组时，RCFile无需全部读取行组的全部内容到内存。相反，它仅仅读元数据头部和给定查询需要的列。因此，它可以跳过不必要的列以获得列存储的I/O优势。例如，表tbl(c1, c2, c3, c4)有4个列，做一次查询“SELECT c1 FROM tbl WHERE c4 = 1”，对每个行组，RCFile仅仅读取c1和c4列的内容。在元数据头部和需要的列数据加载到内存中后，它们需要解压。元数据头部总会解压并在内存中维护直到RCFile处理下一个行组。然而，RCFile不会解压所有加载的列，相反，它使用一种Lazy解压技术。

Lazy解压意味着列将不会在内存解压，直到RCFile决定列中数据真正对查询执行有用。由于查询使用各种WHERE条件，Lazy解压非常有用。如果一个WHERE条件不能被行组中的所有记录满足，那么RCFile将不会解压WHERE条件中不满足的列。例如，在上述查询中，所有行组中的列c4都解压了。然而，对于一个行组，如果列c4中没有值为1的域，那么就无需解压列c1。

行组大小

I/O性能是RCFile关注的重点，因此RCFile需要行组够大并且大小可变。行组大小和下面几个因素相关。

行组大的话，数据压缩效率会比行组小时更有效。根据对Facebook日常应用的观察，当行组大小达到一个阈值后，增加行组大小并不能进一步增加Gzip算法下的压缩比。

行组变大能够提升数据压缩效率并减少存储量。因此，如果对缩减存储空间方面有强烈需求，则不建议选择使用小行组。需要注意的是，当行组的大小超过4MB，数据的压缩比将趋于一致。

尽管行组变大有助于减少表格的存储规模，但是可能会损害数据的读性能，因为这样减少了Lazy解压带来的性能提升。而且行组变大会占用更多的内存，这会影响并发执行的其他MapReduce作业。考虑到存储空间和查询效率两个方面，Facebook选择4MB作为默认的行组大小，当然也允许用户自行选择参数进行配置。

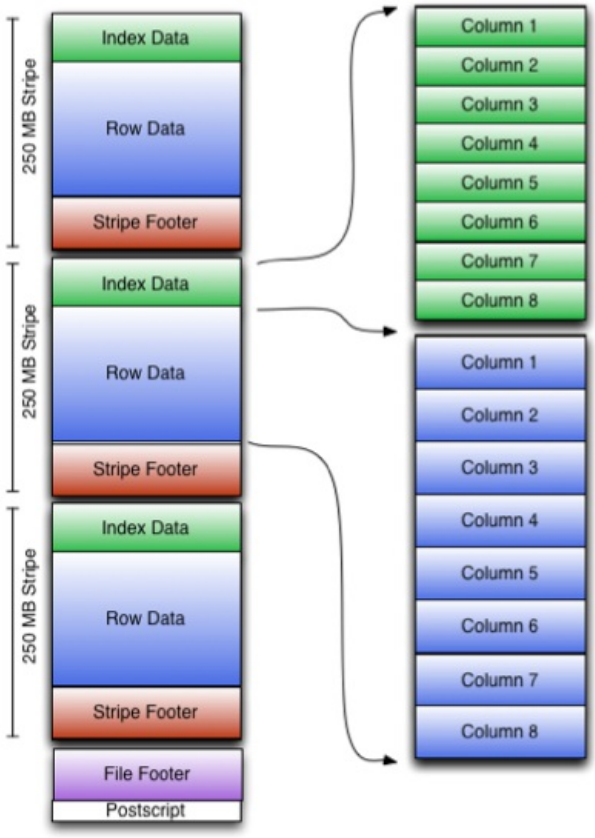
OrcFile Optimized Row Columnar file

ORC（OptimizedRC File）存储源自于RC（RecordColumnar File）这种存储格式，RC是一种列式存储引擎，对schema演化（修改schema需要重新生成数据）支持较差，而ORC是对RC改进，但它仍对schema演化支持较差，主要是在**压缩编码**，**查询性能**方面做了优化。

RC/ORC最初是在Hive中得到使用，最后发展势头不错，独立成一个单独的项目。Hive 1.x版本对事务和update操作的支持，便是基于ORC实现的（其他存储格式暂不支持）。ORC发展到今天，已经具备一些非常高级的feature，比如支持update操作，支持ACID，支持struct，array复杂类型。

优点：

- 每个task只输出单个文件，这样可以减少NameNode的负载；
- 支持各种复杂的数据类型，比如： datetime, decimal, 以及一些复杂类型(struct, list, map, and union)；
- 在文件中存储了一些轻量级的索引数据；
- 基于数据类型的块模式压缩：
  - a、integer类型的列用行程长度编码(run-length encoding)
  - String类型的列用字典编码(dictionary encoding)；
- 用多个互相独立的RecordReaders并行读相同的文件；
- 无需扫描markers就可以分割文件；
- 绑定读写所需要的内存；
- metadata的存储是用 Protocol Buffers的，所以它支持添加和删除一些列。



每个Orc文件由1个或多个stripe组成，每个stripe250MB大小，Stripe实际相当于之前的rcfile里的RowGroup概念，不过大小由4MB->250MB，这样能提升顺序读的吞吐率。

每个文件包括：**Stripe，File Footer,Postscript**

每个Stripe由三部分组成，分别是Index Data,Row Data,Stripe Footer：

Stripe.Index Data：一个轻量级的index，默认是每隔1W行做一个索引。通过行索引，可以在stripe快速读取的过程中跳过很多行。所以默认最大可以跳过1万行。这里做的索引应该只是记录某行的各字段在Row Data中的offset，它可以跳到正确的压缩块位置。还包括每个Column的max和min值，具体没细看代码。

Stripe.Row Data：存的是具体的数据，和RCfile一样，先取部分行，然后对这些行按列进行存储。与RCfile不同的地方在于每个列进行了编码，分成多个Stream来存储。

Stripe.Stripe Footer：存的是各个Stream的类型，长度等信息，Row data在表扫描的时候会用到

因为可以通过过滤预测跳过很多行，因而可以在表的 secondary keys 进行排序，从而可以大幅减少执行时间。比如你的表的主分区是交易日期，那么你可以对次分区（state、zip code以及last name）进行排序。

File Footer：存的是每个Stripe的行数，每个Column的数据类型信息等，还包含了列级别的一些聚合的结果，比如：count, min, max, and sum。

PostScript：记录了整个文件的压缩类型以及FileFooter的长度信息等。在读取文件时，会seek到文件尾部读PostScript，从里面解析到File Footer长度，再读FileFooter，从里面解析到各个Stripe信息，再读各个Stripe，即**从后往前读**。

Orc编码格式：

字典编码：用于String类型的字段

Run-Length编码：用于int，long，short等类型的编码

Bit编码：可以用于各种数据类型

字典编码：对于String类型的每个字段分别保存一个字典，记录每个值在字典中的位置，保存字典的数据结构采用一棵红黑树。对于每个String字段，最终会有三个输出Stream，分别是StringOuptut(记录字典中的值)，LengthOutput(记录每个字典值的长度)，RowOutput(记录字段在字典中的位置)。

思考1：为什么要用红黑树？

因为红黑树无论是插入，删除，查找的性能都比较平均，都是O(logN)，而且是平衡查找树，最坏情况也不会退化成O(N)

思考2：其实一般存储时还会使用LZO之类的压缩，它们本身就是一种字典压缩，为什么Orc里面要自己做字典压缩？

因为LZO之类的压缩窗口一般比较小（LZO默认是64KB），而Orc的字典压缩是以整个字段为范围来压缩的，压缩率会更好。

Run-Length编码：Run-Length能够对等差数列（完全相等也属于等差数列）进行压缩，该等差数列需要满足以下两个条件：

1，至少包含3个元素；2，差值在-128~127之间（因为差值用1Byte来表示）

对于不满足等差数列的数字，Run-Length编码也能存储，但是没有压缩效果，Run-Length的具体存储如下：

第一个Byte是Control Byte，取值在-128~127之间，其中-1~-128代表后面存储着1~128个不满足等差数列的数字，0~127代表后面存储着3~130个等差数列的数字；

如果Control Byte>=0，则后面跟着一个Byte存储差值，否则不存储该Byte；

如果Control Byte>=0，则后面跟着等差数列的第一个数，否则跟着-Control Byte个数字。

例子：

原始数字：12,12,12,12,12,10,7,13

经过Run-Length的数字：2,0,12,-3,10,7,13

Bit编码：对所有类型的字段都可以采用Bit编码来表示该值是否为null。在写任何类型字段之前，先判断该字段值是够为null，如果为null则bit值存为0，否则存为1，对于为null的字段在实际编码时不需要存储了。经过Bit编码之后，可以对于8个bit组成一个Byte，再对其进行Run-Length编码。

Orc对于hive的复杂类型array,map,list等，将其降维成基本类型来存储

ORcfile相对于RCfile做了哪些改进，从Orc作者的ppt里截了张图，分别解释下各行：

	RC File	Trevni	ORC File
Hive Type Model	N	N	Y
Separate complex columns	N	Y	Y
Splits found quickly	N	Y	Y
Default column group size	4MB	64MB*	250MB
Files per a bucket	1	> 1	1
Store min, max, sum, count	N	N	Y
Versioned metadata	N	Y	Y
Run length data encoding	N	N	Y
Store strings in dictionary	N	N	Y
Store row count	N	Y	Y
Skip compressed blocks	N	N	Y
Store internal indexes	N	N	Y

Hive type model:RCfile在底层存储时不保存类型，都当做Byte流来存储
Separtor complex columns:Orc将复杂类型拆开存储
Default Column group size：不用解释了
Store min，max，count，sum：存了这些便于快速地skip掉一个stripe
Run-Length Data-coding：整数类型做Run-Length变长编码
Store Strings in dictionary：String类型做字典编码
Store Row Count：每个Stripe会存储行数
Skip Compressed blocks:可以直接skip掉压缩过的block
Store internal indexes:存储了一个轻量级的index

AVRO

参考：Avro简介

Avro是一个基于二进制数据传输高性能的中间件。在Hadoop的其他项目中例如HBase(Ref)和Hive(Ref)的Client端与服务端的数据传输也采用了这个工具。Avro是一个数据序列化的系统。Avro 可以将数据结构或对象转化成便于存储或传输的格式。Avro设计之初就用来支持数据密集型应用，适合于远程或本地大规模数据的存储和交换。

Avro的数据格式总是以易于处理的形式存储数据结构与数据。Avro可以在运行时使用这些定义以通用的方式向应用程序呈现数据，而不是需要代码生成。

代码生成在Avro中是可选的。它在一些编程语言有时使用特定的数据结构，对应于经常序列化的数据类型是非常好用的。但是在像Pig和Hive这样的脚本系统中，代码生成将是一种负担，所以Avro不需要它。

存储全部的数据结构定义和数据的另外一个优势是允许数据被更快更简洁的写入。Protocol Buffere 为数据添加注解，因此即使定义和数据不完全匹配，数据仍有可能被处理。然而这些注释使得数据更大和更慢的被处理。Avro不需要这些注释，使得Avro数据比其他序列化系统更小和更快地处理。

特点

Ø 丰富的数据结构类型；

Ø 快速可压缩的二进制数据形式，对数据二进制序列化后可以节约数据存储空间和网络传输带宽；

Ø 存储持久数据的文件容器；

Ø 可以实现远程过程调用RPC；

Ø 简单的动态语言结合功能。

技术要领

1，数据类型

数据类型标准化的意义：不同系统对相同的数据能够正确解析；有利于数据序列化/反序列化。

简单类型：null,boolean,int ,long ,float,double,bytes,string

复杂数据类型：Records，Enums，Arrays，Maps，Fixed，Unions

2,序列化/反序列化

Avro指定两种数据序列化编码方式：binary encoding 和Json encoding。使用二进制编码会高效序列化，并且序列化后得到的结果会比较小；而JSON一般用于调试系统或是基于WEB的应用。

3、 模式Schema

Schema通过JSON对象表示。Schema定义了简单数据类型和复杂数据类型，其中复杂数据类型包含不同属性。通过各种数据类型用户可以自定义丰富的数据结构。

Schema由下列JSON对象之一定义：

1. JSON字符串：命名

2. JSON对象：{“type”：“typeName” …attributes…}

3. JSON数组：Avro中Union的定义

eg：一个二位点可以被定义为Avro record:

```
{
    "type": "record", "name": "Point",
     "fields":
     [
          {"name": "x", "type": "int"},
          {"name": "y", "type": "int"},
     ]
}
```

每个实例被序列化为两个整数，没有额外的记录和注释。整数使用可变长度的zig-zag 编码写入。因此，小的负数和整数能够被写仅仅需要两个字节，100个点仅仅需要200个字节。

除了Record和数字类型，Avro还支持数组、map、枚举、可变和固定长度的二进制字节数据和字符串。它还能定义一个容器文件格式，为了能够为MapReduce和其他计算框架提供支持。

4、 排序

Avro为数据定义了一个标准的排列顺序。比较在很多时候是经常被使用到的对象之间的操作，标准定义可以进行方便有效的比较和排序。同时标准的定义可以方便对Avro的二进制编码数据直接进行排序而不需要反序列化。

只有当数据项包含相同的Schema的时候，数据之间的比较才有意义。数据的比较按照Schema深度优先，从左至右的顺序递归的进行。找到第一个不匹配即可终止比较。

5、 对象容器文件

Avro定义了一个简单的对象容器文件格式。一个文件对应一个模式，所有存储在文件中的对象都是根据模式写入的。对象按照块进行存储，块可以采用压缩的方式存储。为了在进行mapreduce处理的时候有效的切分文件，在块之间采用了同步记号。一个文件可以包含任意用户定义的元数据。

一个文件由两部分组成：文件头和一个或者多个文件数据块。

Parquet

Apache Parquet 最初的设计动机是存储嵌套式数据，比如Protocolbuffer，thrift，json等，将这类数据存储成列式格式，以方便对其高效压缩和编码，且使用更少的IO操作取出需要的数据，这也是Parquet相比于ORC的优势，它能够透明地将Protobuf和thrift类型的数据进行列式存储

parquet数据格式的优势？？
列式存储，使用时只需要读取需要的列，支持向量运算，能获得更好的扫描行。
压缩编码可以降低磁盘存储空间，由于同一列的数据类型是一样的，可以使用不同的压缩编码。
可以跳过不符合条件的数据，只读取需要的数据，降低IO的数据量
适配多种计算框架，查询引擎（hive,impala,pig等），计算框架（mapreduce,spark等），数据模型（avro,thrift,json等）
缺点：
比如它不支持update操作（数据写成后不可修改），不支持ACID等。

Parquet适配多种计算框架

Parquet是语言无关的，而且不与任何一种数据处理框架绑定在一起，适配多种语言和组件，能够与Parquet配合的组件有：

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

那么Parquet是如何与这些组件协作的呢？这个可以通过图2来说明。数据从内存到Parquet文件或者反过来的过程主要由以下三个部分组成：

1, 存储格式(storage format)

parquet-format项目定义了Parquet内部的数据类型、存储格式等。

2, 对象模型转换器(object model converters)

这部分功能由parquet-mr项目来实现，主要完成外部对象模型与Parquet内部数据类型的映射。

3, 对象模型(object models)

对象模型可以简单理解为内存中的数据表示，Avro, Thrift, Protocol Buffers, Hive SerDe, Pig Tuple, Spark SQL InternalRow等这些都是对象模型。Parquet也提供了一个example object model 帮助大家理解。

例如parquet-mr项目里的parquet-pig项目就是负责把内存中的Pig Tuple序列化并按列存储成Parquet格式，以及反过来把Parquet文件的数据反序列化成Pig Tuple。

这里需要注意的是Avro, Thrift, Protocol Buffers都有他们自己的存储格式，但是Parquet并没有使用他们，而是使用了自己在parquet-format项目里定义的存储格式。所以如果你的应用使用了Avro等对象模型，这些数据序列化到磁盘还是使用的parquet-mr定义的转换器把他们转换成Parquet自己的存储格式。

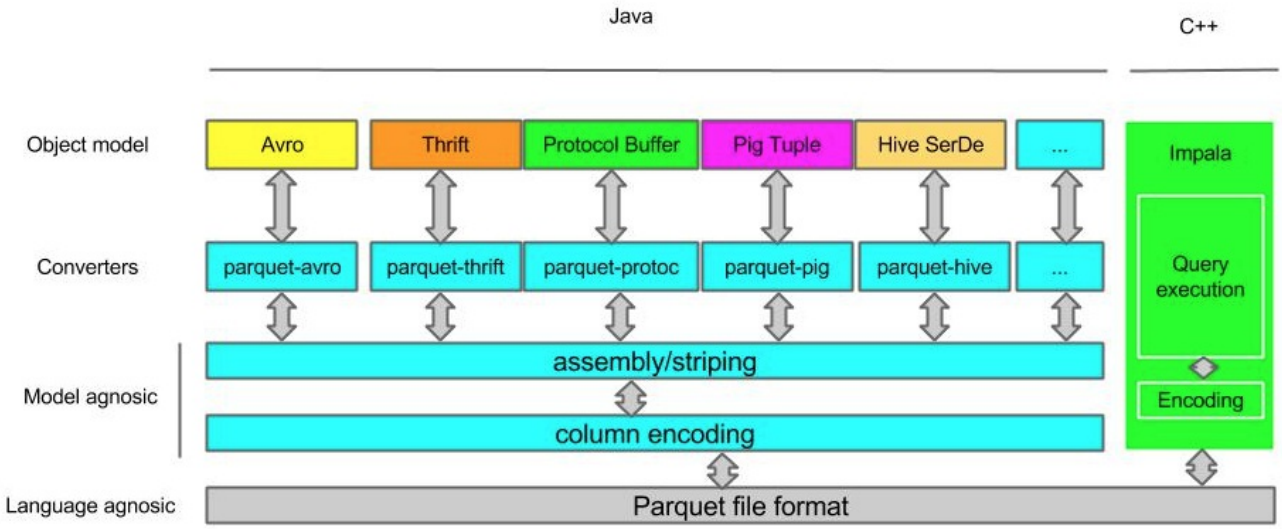


图2 Parquet项目的结构

parquet压缩比例

说明：原始日志大小为214G左右，120+字段

采用csv（非压缩模式）几乎没有压缩。

采用parquet 非压缩模式、gzip、snappy格式压缩后分别为17.4G、8.0G、11G，达到的压缩比分别是：12、27、19。

若我们在hdfs上存储3份，压缩比仍达到4、9、6倍

怎么理解压缩比？？
比如压缩比为12，就是说压缩后的1G相当于压缩前的12G，也就是源文件是压缩后文件的几倍。

parquet参考：深入分析Parquet列式存储格式

压缩格式
压缩后的文件可分割，支持mapper并发读取
hive中的使用，就是先把数据导入textfile表，再把该表的数据导入到sequencefile的表（store as seq...）。

LZO
gz
bzip
gzip
snappy

数据做压缩和解压缩会增加CPU的开销，但可以最大程度的减少文件所需的磁盘空间和网络I/O的开销，所以最好对那些I/O密集型的作业使用数据压缩，cpu密集型，使用压缩反而会降低性能。

而Hive中间结果是map输出传给reduce，所以应该使用低cpu开销和高压缩效率，一般最好使用snappy。

hadoop中支持的压缩格式

DEFLATEorg.apache.hadoop.io.compress.DefaultCodec
gzip org.apache.hadoop.io.compress.GzipCodec
bzip org.apache.hadoop.io.compress.BZip2Codec
Snappy org.apache.hadoop.io.compress.SnappyCodec
LZO:
org.apache.hadoop.io.compress.LzopCodec或者com.hadoop.compression.lzo.LzopCodec;
org.apache.hadoop.io.compress.LzoCodec或者com.hadoop.compression.lzo.LzoCodec;

注意：
(1)org.apache.hadoop.io.compress.LzoCodec和com.hadoop.compression.lzo.LzoCodec功能一样，都是源码包中带的，返回都是lzo_deflate文件
(2)有两种压缩编码可用，即LzoCodec和LzopCodec，区别是：
1)LzoCodec比LzopCodec更快， LzopCodec为了兼容LZOP程序添加了如 bytes signature, header等信息
2)LzoCodec作为Reduce输出，结果文件扩展名为".lzo_deflate"，无法被lzop读取；
而使用LzopCodec作为Reduce输出，生成扩展名为".lzo"的文件，可被lzop读取
3)LzoCodec结果（.lzo_deflate文件)不能由lzo index job的"DistributedLzoIndexer"创建index；且".lzo_deflate"文件不能作为MapReduce输入（不识别，除非自编inputformat）。而所有这些".LZO"文件都支持
综上所述，应该map输出的中间结果使用LzoCodec，reduce输出用 LzopCodec

文件存储和读取的优化。比如对于一些case而言，如果只需要某几列，使用rcfile和parquet这样的格式会大大减少文件读取成本。再有就是存储文件到S3上或者HDFS上，可以根据情况选择更合适的格式，比如压缩率更高的格式。另外，特别是对于shuffle特别多的情况，考虑留下一定量的额外内存给操作系统作为操作系统的buffer cache，比如总共50G的内存，JVM最多分配到40G多一点。

补充：行列存储的比较：

行存储结构

基于Hadoop系统行存储结构优点在于快速数据加载和动态负载的高适应能力，因为行存储保证了相同记录的所有域都在同一个集群节点，即同一个HDFS块。不过，行存储的缺点也是显而易见的，例如它不能支持快速查询处理，因为当查询仅仅针对多列表中的少数几列时，它不能跳过不必要的列读取；此外，由于混合着不同数据值的列，行存储不易获得一个极高的压缩比，即空间利用率不易大幅提高。尽管通过熵编码和利用列相关性能够获得一个较好的压缩比，但是复杂数据存储实现会导致解压开销增大。

列存储结构

A和B存储在同一Column Group，而C和D分别存储在单独的Column Group。（见rcfile部分的图）

查询时列存储能够避免读不必要的列，并且压缩一个列中的相似数据能够达到较高的压缩比。

然而，由于元组重构开销较高，它并不能提供基于Hadoop系统的快速查询处理。

列存储不能保证同一记录所有域都存储在同一集群节点。记录重构将导致通过集群节点网络的大量数据传输。

尽管预先分组后，多个列在一起能够减少开销，但是对于高度动态负载模式，它并不具备很好的适应性。

参考：
hive 压缩全解读(hive表存储格式以及外部表直接加载压缩格式数据)；HADOOP存储数据压缩方案对比（LZO,gz，ORC）