Apache Parquet

Apache Parquet是Hadoop生态圈中的列式存储格式，可以兼容大多数计算框架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。Parquet的灵感来自于Dremel，其使用一种支持嵌套结构的存储格式，并使用列式存储的方式来提高查询性能，其开源实现是Apache Drill。

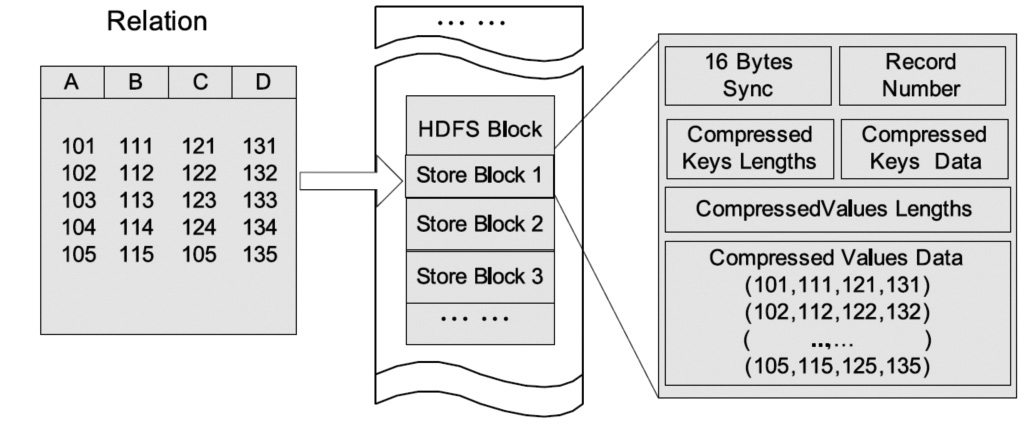
# Parquet与列存储

在基于Hadoop存储的系统中，数据存储格式决定了如何高效的存储、管理和使用数据，影响数据读写性能。在Hadoop的运行环境中，数据的存储格式要考虑以下因素：

* 数据加载速度
* 数据查询处理
* 数据存储空间利用率
* 高强度的动态负载模式

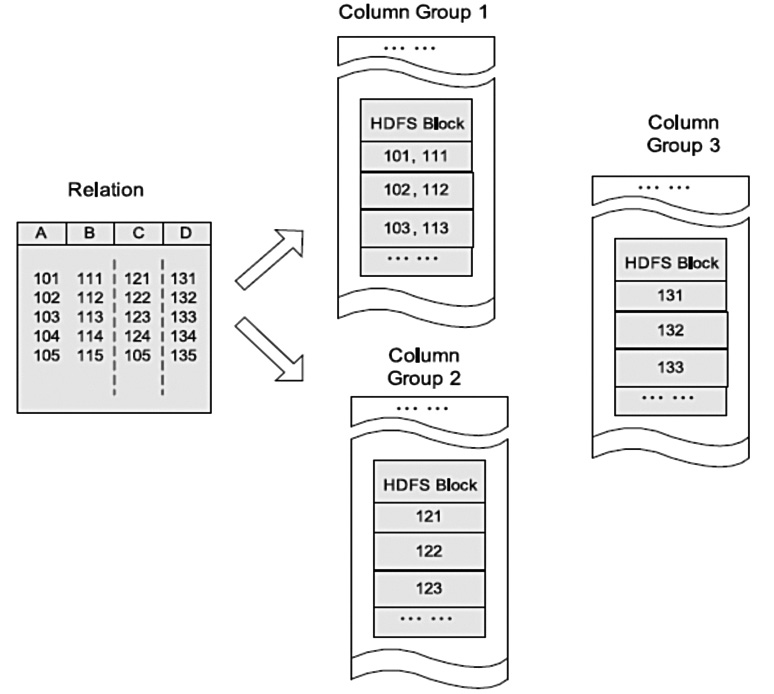
目前在Hadoop中，主要将数据的存储分成行存储和列存储两种文件格式

1. 行存储，以数据行为单位将数据聚簇在一起，其保存相同记录的所有域都在同一个集群节点，如下图：



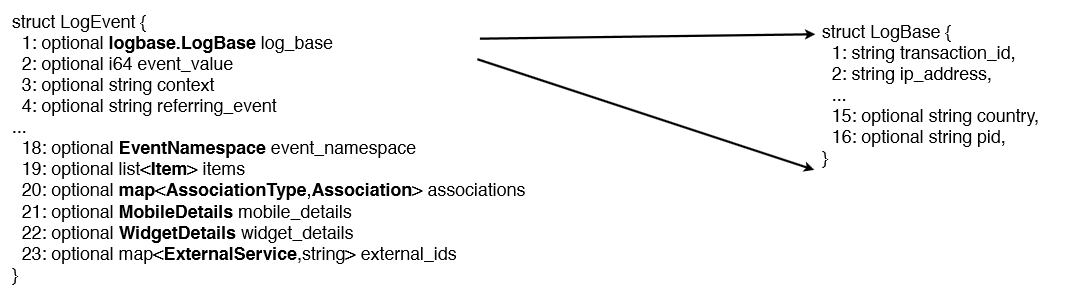
但是查询少数列式，不能直接定位到列而跳过不需要的列。

1. 列存储，以列数据集合为单位，一行记录会被拆分成多列，写入时追加到对应的列的末尾处，如下图：



在面向列的文件存储结构中，列A和列B存储在同一个列簇，而列C和列D分别存储在单独的列簇，这种结构使得在查询时能够直接读取所需的列而避免不必要列的读取，对相似的数据可以有一个更好的压缩比。可以充分利用列存储的特点支持向量计算，获取更好的扫描性能。

Parquet是一个自描述语言无关的列存储，最初的设计动机是为了存储嵌套数据，将Avro Thrift及Protobuf等格式数据转换成列数据来存储。在大数据环境下，通常的数据来源是服务端的埋点数据，将程序中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，这样在查询的时候不需要额外的解析便能获得想要的结果，如下图：



目前Hadoop主流的查询引擎都支持更丰富的数据类型，如Spark SQL、Impala都原生支持Struct、Map、Array这样的复杂数据类型。因此Parquet这种原生支持嵌套数据类型的存储格式非常有必要。

# Parquet概念

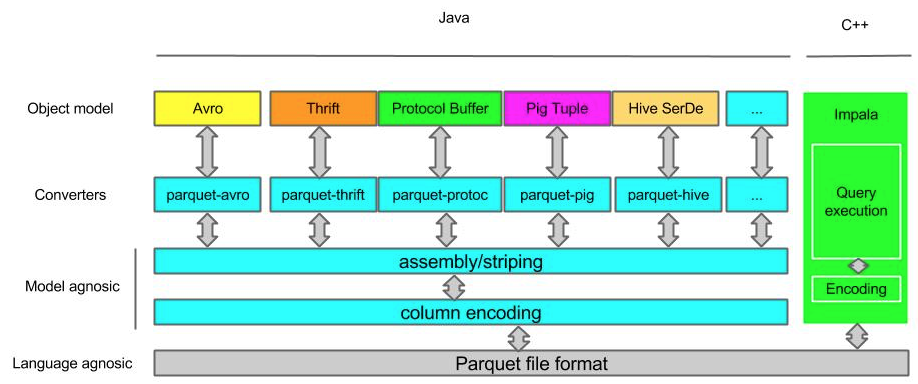
Parquet是一种存储格式，和语言及平台无关，不会和任何一种数据处理框架绑定，目前能和Parquet适配的组件包括如下：

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

Apache Parquet有以下几个子项目组成：

* Parquet-format，由Java实现，定义了所有Parquet元数据对象，元数据使用Apache Thrift进行序列化并存储在Parquent文件尾部
* Parquet-mr，包含多个子模块，包括实现Parquet文件读写功能，并提供和其他组件的适配工具，例如Hadoop Input/Output Formats,Hive Serde（目前Hive已经自带Parquent）,Pig Loaders等
* Parquet-compatibility，不同编程语言之间的读写文件的测试代码
* Parquet-cpp，用于读写Parquet文件的C++库
* Parquet-rs，读写Parquet文件的Rust库

各个组件的层次以及从上到下的交互的方式如下图所示：



1. 数据存储层，定义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
2. 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
3. 对象模型层定义了如何读取Parquet文件的内容，这一次转换包括Avro、Thrift和PB等序列化格式，Hive Serde等的适配

## **数据模型**

Parquet文件是以二进制的方式存储的，文件中包含该文件的数据和元数据，因此Parquet文件是自解析的

1. **常见术语**

* Block，在HDFS上最小的副本单位，会把Block存储在本地的一个文件并且由Hadoop维护分散在不同机器上的副本，通常一个Block的大小是256M或者512M
* File，HDFS上的文件，包括数据和原数据，数据分散存储在多个Block上
* Row Group，按照行将数据逻辑上划分为多个单元，每单元包含一定的行数，在一个HDFS文件中至少存储一个行组，Parquet读写的时候将整个行组缓存在内存中。
* Column Chunk，列块中的值都是相同类型的数据，相同列块连续存储在行组文件文件中
* Page，每个列块划分成多页，一个页是最小的编码单位，在同一个列块中不同的页可能使用不同的编码方式

在并行执行过程中，MapReduce读取Row Group或者文件，IO读取的是列块数据，编码单位是Page。

1. **数据模型Schema**

Parquet支持嵌套的数据类型，类似于Protocol Buffers，

* 每个数据模型的Schema包含多个字段
* 每个字段又可以包含多个字段
* 字段包含三个属性：重复数、数据类型和字段名
* 重复数三种：required(1次)，repeated(出现0或者多次), optional(0次或者1次)。
* 字段数据类型：group(复杂类型)，primitive(基本类型)

下面是AddressBook的数据模型Schema

*message AddressBook {*

*required string owner;*

*repeated string ownerPhoneNumbers;*

*repeated group contacts {*

*required string name;*

*optional string phoneNumber;*

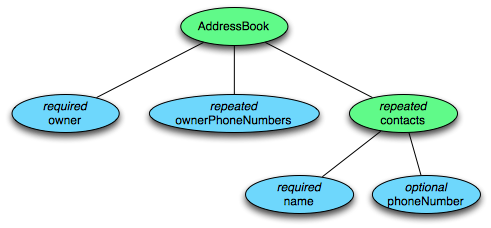
*}*

*}*

每条记录表示一个人的AddressBook：

* 有且只有一个Owner
* 每个Owner可以有0个或者多个ownerPhoneNumbers
* 每个Owner可以有0个或者多个contracts
* 每个contract，有且只有一个名字，PhoneNumber可有可无

树结构如下：



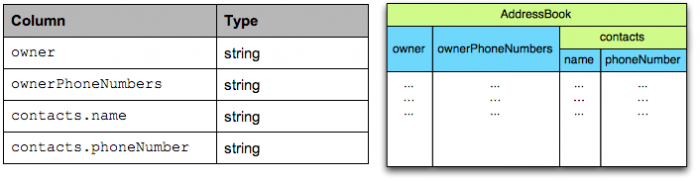
从图中可以看到Schema中所有基本类型字段都是叶子节点，一共由4个叶子节点，如果转换成扁平式的关系模型，该表包含4个列。

在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。由于记录中某一列可能出现0次或者多次，需要表示出哪些列的值构成一条完整的记录。

由于一条记录中可能存在比较深的嵌套关系，为记录维护树状结构占用较大的存储空间，因此使用Stripig/Assembly高效的对嵌套数据格式的压缩算法。

## **文件结构**

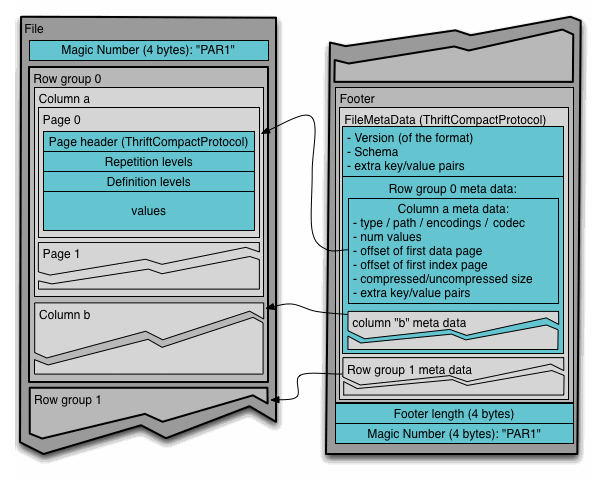
Parquet文件以二进制方式存储的，不能直接读取和修改，在文件中包括该文件的数据和元数据以实现自解析。在Parquet格式存储中，Schema的叶子节点实际对应多少个Column，上图中实际上是4个Column，如下图所示：



前面已经介绍了Parquet文件的存储概念，下面是切分的模式

* 所有的数据被水平切分成Row Group，在Row Group中包含对应区间内所有列的column chunk，其是数据读写的缓存单元，设置较大的Group Goup可以增加并行度。
* Column Chunk负责存储某列的数据，这些数据包括：Repetition Levels，Definition Level和values
* Column Chunk由Page组成，Page是压缩和编码的单元，对数据模型来说是透明的。
* 一个Parquet文件的最后是Footer，存储文件的元数据信息和统计信息

对于AddressBook Schema为例，其形成的文件格式如下图所示：



上图展示了一个Parquet文件的内容

* 一个文件可以存储多个行组
* 文件的首位是该文件的Magic Code，用于校验是否是一个Parquet文件
* Footer Length，文件元数据的大小，通过该值和文件长度可以计算元数据的便宜量
* FileMetadata，文件元数据包括一个行组的元数据信息和该文件存储数据的Schema信息
* 页元数据，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段，每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还不支持索引页。

在执行MR任务时可能存在多个Mapper任务的输入是同一个Parquet文件的情况，通过InputSplit标示出处理文件范围，如果多个InputSplit跨越Row Group，其保证Row Group被一个Mapper任务处理。其文件的内容物理结构如下所示：

*4-byte magic number "PAR1"*

*<Column 1 Chunk 1 + Column Metadata>*

*<Column 2 Chunk 1 + Column Metadata>*

*<Column N Chunk 1 + Column Metadata>*

*...*

*<Column N Chunk 2 + Column Metadata>*

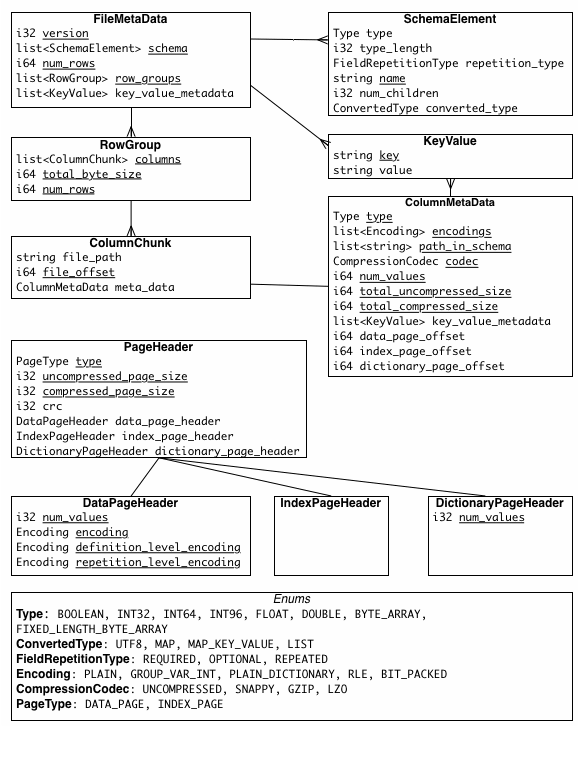
*<Column N Chunk M + Column Metadata>*

*File Metadata*

*4-byte length in bytes of file metadata*

*4-byte magic number "PAR1"*

在Parquet文件中，有三种类型的元数据：FileMetadata,Column Metadata和Page Header Metadata，使用Thrift协议来描述：



## **Striping/Assembly算法**

上面介绍了Parquet的数据模型，在Document中有多required列，而且一条记录中的数据分散在不同列中，如果组合不同的列值组成一条记录由Striping/Assembly算法决定。在PageMetadata中还存储了两个变量：Repetition Level(R)和Definition Level(D)，这两个变量用于序列化和反序列化嵌套数据类型。Striping/Assembly算法中每个值包含三部分：

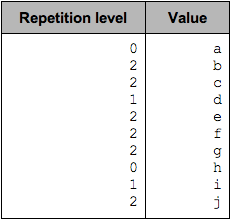
1. Definition Level

在嵌套数据类型中有些field是空的，没有定义。如果field是定义的，那么它的所有父节点都是被定义的。从根节点可以遍历，当某一个field路径上的节点开始为空，那么记录当前深度作为这个field的Definition Level。当DefinitionLevel为field的最大Level时，表示这个field是有数据的。对于required类型的field必须是有定义的，所以是不需要的。

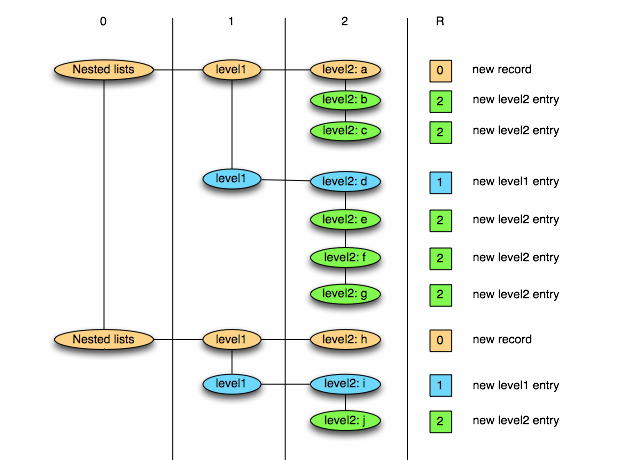
1）Repetition Levels

记录该field值是在哪个深度上是重复的，只有repeated类型的field需要Repetition level，optional和required类型不需要。Repetition Level = 0表示开始一个新的record，在关系型数据中repetion level总是0，在读取的时候根据该值推导出哪一层需要创建新节点。

下面是一个Value需要记录的Repeated Level值：

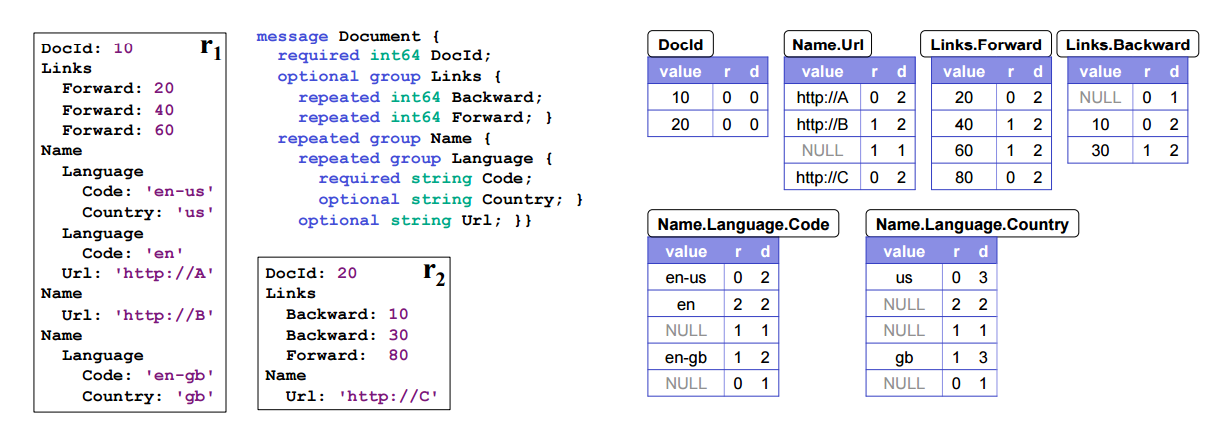


在读取的时候，顺序的读取每一个值，然后根据它的repeated level创建对象，当读取value=a时repeated level=0，表示需要创建一个新的根节点(新记录)，value=b时repeated level=2，表示需要创建一个新的level2节点，value=d时repeated level=1，表示需要创建一个新的level1节点，当所有列读取完成之后可以创建一条新的记录。本例中当读取文件构建每条记录的结果如下：



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。

下面是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，



1. DockId，对于r1，DocId=10，由于它记录的开始并且是已定义，所以R=0,D=0，同样r2中DocId=20，R=0,D=0
2. Links.Forwords，r1中，未定义但是Links是已定义，并且是该记录下的第一个值,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)。
3. Name.Url，r1中它有三个值，分别为url1=’[http://A](http://a/" \t "https://blog.csdn.net/yu616568/article/details/_blank)‘，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](http://b/" \t "https://blog.csdn.net/yu616568/article/details/_blank)‘，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](http://c/" \t "https://blog.csdn.net/yu616568/article/details/_blank)‘，R=0，D=2.
4. Name.Languages.Code，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使用示例

1）添加Pom依赖

*<dependencies>*

*<dependency>*

*<groupId>org.apache.parquet</groupId>*

*<artifactId>parquet-hadoop</artifactId>*

*<version>1.9.0</version>*

*</dependency>*

*<dependency>*

*<groupId>com.google.guava</groupId>*

*<artifactId>guava</artifactId>*

*<version>11.0.2</version>*

*</dependency>*

*<dependency>*

*<groupId>org.apache.hadoop</groupId>*

*<artifactId>hadoop-client</artifactId>*

*<version>2.8.1</version>*

*</dependency>*

*<dependency>*

*<groupId>org.apache.avro</groupId>*

*<artifactId>avro</artifactId>*

*<version>1.8.2</version>*

*</dependency>*

*<dependency>*

*<groupId>org.apache.parquet</groupId>*

*<artifactId>parquet-avro</artifactId>*

*<version>1.8.1</version>*

*</dependency>*

*</dependencies>*

2）定义Java POJO类

*public class Pojo {*

*private long id;*

*private String data;*

*public Pojo() {}*

*public Pojo(long id, String data) {*

*this.id = id;*

*this.data = data;*

*}*

*//get set toString ignore*

*}*

3）定义Parquet测试类

*public class ReadParquetPojo {*

*public static void main(String[] args) throws Exception {*

*Collection<Pojo> data = Lists.newArrayList(*

*new Pojo(1,"airport"),*

*new Pojo(2,"penguin")*

*);*

*Configuration conf = new Configuration();*

*Path dataFile = new Path("/tmp/data.gz.parquet");*

*//Write Pojos to a Parquet File*

*try (ParquetWriter<Pojo> writer = AvroParquetWriter.<Pojo>builder(dataFile)*

*.withSchema(ReflectData.AllowNull.get().getSchema(Pojo.class)) // generate nullable fields*

*.withDataModel(ReflectData.get())*

*.withConf(conf)*

*.withCompressionCodec(GZIP)*

*.withWriteMode(OVERWRITE)*

*.build()) {*

*for (Pojo pojo : data) {*

*writer.write(pojo);*

*}*

*}*

*//Read Pojos from a Parquet FIle*

*try (ParquetReader<Pojo> reader = AvroParquetReader.<Pojo>builder(dataFile)*

*.withDataModel(new ReflectData(Pojo.class.getClassLoader()))*

*.disableCompatibility() // always use this (since this is a new project)*

*.withConf(conf)*

*.build()) {*

*Pojo pojo;*

*while ((pojo = reader.read()) != null) {*

*System.out.println("All records: " + pojo);*

*}*

*}*

*//Read Pojos, but filter down to just penguins*

*FilterPredicate predicate = eq(binaryColumn("data"), Binary.fromString("penguin"));*

*try (ParquetReader<Pojo> reader = AvroParquetReader.<Pojo>builder(dataFile)*

*.withDataModel(new ReflectData(Pojo.class.getClassLoader()))*

*.disableCompatibility() // always use this (since this is a new project)*

*.withFilter(FilterCompat.get(predicate))*

*.withConf(conf)*

*.build()) {*

*Pojo pojo;*

*while ((pojo = reader.read()) != null) {*

*System.out.println("Just Penguins: " + pojo);*

*}*

*}*

*}*

*}*

1. 执行后，在HDFS目录下创建data.gz.parquet文件

输出如下：

*All records: Pojo{id=1, data=airport}*

*All records: Pojo{id=2, data=penguin}*

*Just Penguins: Pojo{id=2, data=penguin}*

http://www.infoq.com/cn/articles/in-depth-analysis-of-parquet-column-storage-format

http://parquet.apache.org/documentation/latest/

http://flyingdutchman.iteye.com/blog/1871025

https://blog.csdn.net/yu616568/article/details/50993491