# OpenStore设计思想

## 系统目标和开发模式的颠覆

引言：当前软件开发的系统，数据存储占据着相当重要的部分。从几十年前一统天下的关系数据库存储数据，到如今大数据、人工智能大背景之下的文档型数据库（以mongodb为首）、KV数据库（以redis为首）、列式存储（hbase，rcfile文件等）、图形存储（neo4j、cayley等）。每一个数据存储方式，都直接决定了软件系统的设计架构、开发工具、开发模式、甚至直接影响了程序猿们的“开发习俗”（注：用“习俗”而不用“习惯”，一字之差，却能真实表达这个事实，习惯代表了个个人的常态，而习俗代表了一个团体，一个社区的约定俗成，虽然这个约定俗成在现在看来，并非最好或最正确）。

而在这些底层存储机制之上，又不断涌现出各种开发框架和平台，例如：关系数据库的各类数据库管理系统：sqlserver系列\mysql等；graphsql图形计算平台等。这些工具的推广和成型，直接导致了开发者，既享受了这些工具和平台带来的关于“数据存储“方面的操作便捷性，统一性。又不得不陷入到这些平台工具的”习俗“之中。

### 传统 vs 当下

关于数据存储，正如引言中谈到。我们可以在这里将“传统“缩小一定的范围，将其只限定于”企业应用“范围。对企业应用来说，”传统“的数据存储机制，毫无疑问，当属《关系型数据库》，没有之一。这除了历史原因决定外，还是由“关系数据库”本身的特性所决定的：

1. 对企业应用来说，大部分的业务数据都习惯性的按照“表格”方式组织，企业中每个部门，每个员工在执行日常事务时，都习惯了这种描述数据，存储数据的方式。
2. 大部分的企业应用中的“表格”式数据之间，业务上都是有“关联”的，并且这种关联，在企业以及企业员工的“思维定式”中，都是自动具有“强制性关联“的。例如：购物车id一定关联着购物车中的商品id
3. 传统技术的“教育“，在设计企业应用时，为了数据所谓的”完整性“、”一致性“等这些数据”约束“，关系数据库将这些概念植入了设计和开发人员的骨髓，数据的schema模式、事务、一对多，一对一等概念已经成为了”数据设计“的”基本常识“
4. 由于DBMS管理系统的普及和强大，以及操作方式SQL语言的统一便捷，使得开发企业应用数据存储部分的开发人员，不必费时费力去思考特定环境下“数据”的各种存储、查询等特性。直接一条简单的sql语句，即能完成相对比较复杂的企业应用中数据的新增、搜索、排序等操作

而当下，随着大数据和人工智能的流行，企业应用中的各种功能、业务、逻辑也面临着各种挑战和改变。同时，导致了企业应用中的《数据存储》部分，也有了翻天覆地的变化，主要表现在如下几方面：

1. 企业组织机构的复杂以及业务的延伸，导致企业每天产生的数据量呈几何级数增长，这导致了之前关系数据库中很难得出现的分库、分表、集群等“小场景“应用功能提上了日程。而关系库或关系库DBMS中的分库、分表、集群等操作和配置，又是及其复杂的
2. 由于当前互联网时代的快速发展，导致企业要面临的业务随时需要灵活的频繁变化，这导致了企业应用中的各种“数据“，并非一定是要模式化的，结构化的。【无模式】、【无类型】的由用户可以灵活定义的”数据“变得原来越受欢迎
3. 企业应用中所面临的现代“数据“，并非一切都是”扁平结构“的二维表所能表达的。或者说，用扁平结构的两维表有时候很难清晰简单的表达企业应用中涉及的数据，例如：分层次的企业组织机构和各类角色数据、企业客户的网络关系分布图等这些数据，用两维表来表达这种数据，【表结构】以及【表之间的关联】将不得不设计的非常复杂，从而导致从用户的“视角”根本无法直观的理解
4. 在当前互联网快速响应的整体风格下，企业应用中对数据的管理，有时并非要求的非常严格。也就是说，传统关系数据库要求的各种范式、事务、强一致性、外键约束等被越来越多的“弱化”！用户提交的一条评论，管理员一定要实时的查看到么？售后服务人员删除的一条数据，查询页面立马就要消失么？如果订单提交失败，预付款是否需要立即回滚？删除了订单，订单项中的商品如果存在，就一定非法么？当前的实践证明，关系数据库提出的严格数据范式，太理论化了。现实世界中的数据，应该将关系弱化、应该将一定范围的错误包容，才能为灵活的企业业务提供强有力的支持！

### 数据存储的开发模式

当要制作一个企业应用时（这里只描述数据部分），传统的设计方法如下：

* 需求分析人员分析出系统业务所涉及的数据结构，将客户方提出的无模式、非结构化的“业务数据”模式化、结构化
  + 1. 将客户直接的“业务概念”归类，例如：商品、订单、库存
    2. 将归类的数据分解成“组成字段“：例如，商品中包括名称、SKU、品牌等
    3. 组织归类数据之间的关系：例如，一个订单中包括多个商品
* 开发（设计）人员，根据需求分析员提供的“结构化“数据，开始设计”表结构“
  + 1. 将归类概念设计成“表“
    2. 将组成字段，设计成字段，并设定字段的类型，例如SKU为整数、名称为64位字符串
    3. 为了表达表之间的关系，在表字段中添加额外的【关系型】字段：外键，例如品牌id，订单id等【这些字段对业务方毫无意义，但对系统运行有着至关重要的意义】
    4. 为了企业应用上的一些特殊逻辑，添加额外的字段（我们称其为：计算字段）。例如：为了表达数据的“被删除”行为，额外添加：isDeleted字段、为了表达一些算法而添加的一些字段等
    5. 为了建立一些表之间的关系，纯粹添加一些毫无意义的“中间表“，这些表纯粹只为提供一些映射关系，没有任何业务意义
    6. 为了配合关系库的“约束”，设置一些字段的“数据约束”，例如：用户评论不能超过255个字符、商品订单id不能为null等
    7. 有时，为了sql语句查询的方便，以及降低sql语句语法的复杂度，特殊设计一些字段来降低sql风险。例如left join，right join，子查询优化等

传统“数据开发模式“，带来的”习俗“的好处是不言而喻的，但弊端随着时代的变化也越来越明显：

1. 业务数据和物理数据是脱离的。在用户视角，根本无法理解他的“业务数据”和数据库中表、字段、关系、约束有什么对应关系。
2. 一个完整的“业务数据“，纯粹为了系统和”习俗“，而被人为分解成了多张表进行了存储，而这些分散的表，又是通过”业务无法理解“的方式，关联在一起。例如，一个订单可能被分解成了三四张表进行了存储，而这三四张表通过用户不理解的各种主外键id关联
3. 数据的“主体内容”和“附加属性“都被添加到了一起，不仅占用了空间还降低了查询性能。例如：一个订单，有其主体内容，而订单的”创建者“、”修改者“、”创建时间“等，不过是订单的”附加属性“而已。但表结构中，也必须设计字段存储这些附加属性
4. 单纯从实现上来说，表结构是紧密存储结构而并非稀疏存储结构。因此，一张表无法简单稀疏化的分布在各个不同的节点上

基于以上的讨论，openstore如何改变传统的开发模式呢？openstore主要针对以下几点进行改变：

* 一个“业务概念“对应一个FileFamily文件族，一个FileFamily文件族代表着【业务上相同的概念】。例如：业务上，“采购单据”都是相同的概念，因此，他应该被存储在Family为PurchseOrder的一个或多个File中（也就是说，一个Family可以由一个文件存储，也可以有多个文件存储）。一个“文件族”和业务上的某一个“概念”是一对一对等的，例如：企业应用有如下概念：采购单、出库单、入库单、售后服务单、配件申请单、外呼任务单、核销单….那么openstore中就有7个FileFamily
* 每一个FileFamily都需要指定一个Format文件格式，文件格式就是在一个文件系统中的文件类型，如：Excel、Word、PDF、txt….。Openstore通常支持普通人员能够理解的常用文件（比如excel，pdf等），也支持专业开发能够理解的文件（json、avro序列化文件），同时也支持大数据环境下的各类文件格式（如：hadoop mapfile、sequenceFile、RCFile，ORCFile、hbasefile等），自定义文件格式的解析器，也可以随时加入openstore进行扩展。
* 每一个“业务概念“中的具体某一项，都对应一个File文件。

例如：

1. 某个具体的1号采购单，就是一个采购单.0001.xls文件
2. 名称为：XXXXX外呼任务单，就是一个”外呼任务.orc.it”文件（orc文件中的一条记录）

* Openstore中File都是逻辑上的概念，也就是说，一个FileFamaily只是一个逻辑上的File集合，一个File只是一个逻辑上的“文件“，实际一个File数据存储在物理上是否是一个实际存储盘上的一个或多个物理文件，通常可以自定义调整，或者根据每种Family的Format默认决定；如：orc文件格式，就只有一个文件。

总结一下，如果使用openstore来设计企业应用数据存储，可以按照如下整体流程进行：

找出业务概念【如：采购单】

使用openstore生成对应的FileFamily

具体生成细节，后续章节描述

选择FileFamily文件格式

普通：excel/pdf/txt…

专业：json/avro…

大数据mapfile/sequenceFile/orcfile

定义File各组成部分

实际创建File，向File输入数据

就像你在C盘上新建了一个《采购单001.xls excel文件，随后的输入数据，就是在此excel文件中编辑采购单各个组成部分

## 数据存储的常见场景

### “编辑”型应用 OR “查看”型的应用

任何一个企业应用，无论功能多么复杂，都能够划分为两个部分，如图所示：

人机交互（通常为app界面、web界面、桌面界面）

与其他系统的调用接口

系统本身后台处理（各类job、逻辑运算等）

对右半部分来说，不存在所谓：“编辑”操作多还是“查询”操作多的问题！只存在如何将这两类操作进行分离的问题。所以，我们重点分析上图中的左半部分。

#### 人机交互部分

通常来说，一个企业应用的业务虽然复杂，但仔细分析我们发现，其很多关于数据操作的部分，还是有迹可循的。例如：

* 如果你正在使用一个word类型的文字处理应用，是“编辑”的时机多？还是“查看”的时机多？

显然，这种类型的应用，“编辑”操作**远远大于**“查看”操作（至少，在一篇文档还没有成为“历史”或“归档”状态时）。有小伙伴可能会问：不对啊，我的word文档，“编辑“时也意味着”查看“啊，这两种操作应该是一样多才对！

真的是这样么？NO！让我们深入分析一下就知道了：

当你准备“编辑“一份word文档时（数据），你首先要打开这份“数据”。此时，对Word应用来说，它只需要**从“存储”上Read读取一次**，将文档读入到应用的缓冲中，就可以查看文档所有内容了。在此之后，无论你是翻页，滚动，都没有任何【读取】操作。而当你开始敲打键盘，上一秒插入一张图片，下一秒打入一个文字，然后按下crtl+s时，Word应用就会**立即调用Write写入操作。**

所以，从上面的分析，大家可以看到。这里我们所阐述的“编辑“操作远远大于”查看“操作，是**针对应用系统本身来说的，而并非人机交互中的UI界面。对于这种类型的应用，你通常只需要打开一次，却需要写入多次**

* 一个web论坛，发帖，回帖，是“编辑”操作多？还是“查看”操作多？

严格说来，论坛类型的web应用，不太属于“企业应用”的范畴，但与企业应用在【数据的存储操作】上也有许多相似之处。

小伙伴们都经常逛论坛，发帖回帖，在问答社区里提出问题回答问题。那么这种类型的应用，小伙伴很快会回答我：当然是“编辑”操作多啊！当在线人数非常多时，很多人发帖，回帖。这里，我们就需要对“编辑”操作，做一个小小的区分了：

1. 发帖：当你在一个社区发帖时，通常你是打开一个web编辑界面。此时，传统做法是，只要你没有将帖子编辑完成，最后点击“提交”按钮，论坛的web应用本身都不会发生“写”操作。当然，现代的web应用，都有一定的【临时保存，定时保存】功能，但即使有定时保存功能，也是在“提交”的那一刻，才会发生“写入”
2. 回帖：回帖也和发帖类似。但【回帖】操作，已经和“帖子“本身分离了开来，回帖，只是在另一个”数据区域“中**追加新内容**，并没有对”帖子原有内容“进行修改！也没有对其他帖子的内容进行“修改”操作，这样的【写入】操作，和word文字处理应用中的“编辑“是有很大的区分的，它仅仅只是Append追加内容而已，写操作，好比一种“流”式的写入，应用系统并没有发生随机的”写“操作！
3. 无论发帖还是回帖，都不可能存在对同一个“数据“例如帖子，或者回帖，进行多人同时【编辑】的操作！因此，“写入”操作，仅仅是单一在“提交”那个相对孤立的点发生的操作而已，这个操作，相对“某一个数据”本身而言并不频繁

下面，我们再来看看这种类型的应用的“读“操作：

1. 当你打开主页，查看论坛首页时，系统会至少发生一次“读取”。读取所有主题的帖子列表。此时发生的“读取”操作，都是【搜索】类、【聚合】类的读取操作。也就是说，此时的读取，并非读取整个大数据集的所有信息，而仅仅是读取一个文件、以及文件内容中的索引类信息即可
2. 当你点击“下一页“或者滚动时，系统还会发生【视窗】类的“搜索”读取操作，此读取操作通常是锁定一个大数据集的 “视窗”范围视图，将其读取。同样，读取的也是索引类的概要信息
3. 当点击某个帖子看帖时，系统将读取整个数据集中某一项的整体信息，以及相关联的其他项信息。整体信息本身将一次性读取，关联信息分批次读取

通过以上分析，大家可以看到，这种论坛型的应用，【读取】操作至少是>=【写入】操作的。并且，这种类型的应用，很少是随机“写入”，都是【追加】型，流式的“写入“

* 企业办公、工单型应用：这种类型的企业应用，其实与《论坛》类型非常相似。例如：办公应用，员工打开、人员请假、采购单据审批、销售单据审批等，原始提交的单据，一般在后续企业流程处理中不会修改。但这里和《论坛》型应用，又有一个区别

企业应用，往往涉及一系列流程处理，在流程处理过程中，可能会将上一步流程驳回，重新编辑。例如“采购申请单”中的某些项不符合要求，被下一步驳回，重新开始编辑。此时，的确发生【随机】形式的写操作。但这种写操作，因为本身数据量不多，也可以【整体块】覆盖的方式进行，或者为了记录版本，而仅仅发生新版本的追加“写入”操作。

#### 调用接口部分

企业应用提供的对外接口，绝大多数都是“读”接口，这其实有处于安全性的考虑。即使需要发生“写”操作，也会尽量避免发生随机修改，而仅仅是追加性修改！

【可能有很多人并不这么认为…..】

### Write的种类

对任何一个企业应用来说，写入数据是最基本的操作。无论是工单型企业应用中的创建工单、还是审批时添加的审批、或者是添加一个呼叫记录等，都需要将数据“写入”。注意：这里我们尽量避免说“写入数据库”！就是因为传统企业应用，或传统开发人员太容易把【数据写入】绑定到“写入数据库”。

实际上，【写入】操作，是多种多样得。通过第一节得分析，我们可以看出，【写入】操作，主要分为以下几种：

* 随机写入：虽然这种写入数据得方式很常见，但理论和实践都证明，严格的“随机写入“是不存在的！或者说，要实现”随机“写入，也可以用其他各种技术手段实现。
* Appen追加写入：实际上，这种写入方式，是最常见的“写入数据”的方式！也是对现实世界最真实的抽象，你对系统的任何修改，其实并非是修改，而仅仅是产生了一次新的【事件】，这个事件是在原有数据上增加了一个新版本的数据而已。大部分企业应用的write操作，其实都可以使用这种Append的方式解决。并且，目前CQRS应用开发方法，在数据层次上，从某种意义上来说，也是应用这种数据写入的管理方式。

#### 应用层的Write与实际存储层的Write的不协调

从企业应用层的角度，显然，随即写入是最希望得到的特性。可以将数据随便写入到任意位置，原位置有数据则覆盖，没有数据则追加，不是很自然的需求么！是的，这是应用的自然需求，也是必须满足的。但实际存储层，通过以上分析，大家知道，真正的随机写入，实际并不存在的。【现代各种流行的大数据计算平台以及数据库系统都基本不支持随机写入（hadoop，各种时间序列库等）】

实际上，怎么将应用层的随机写入和存储层的追加写入协调起来，HBase已经为我们提供了一种非常好的解决方案，看下图：



实际上，就是将上层引用的“随机写入”需求，先限制在“内存的LSM数据结构中”，然后达到一定阀值之后，flush到磁盘【Appen方式写入】，满足某种条件，compact合并，将相同数据，不同版本删除历史即可！

应用

内存LSM等结构

随机

写入

持久层

定期合并

Append写入

### 写入数据时的各种问题

以上两节，只是探讨了企业应用中write数据时的类型问题，但并没有讨论企业应用中，围绕着write数据，还涉及到的其他各种问题。

实际上，企业存储服务中，写入数据，还面临着以下几种相关联的问题：

* 写入数据的原子性如何切分

企业应用中，通常写入的数据都不是“流式”的字节数据，而大部分都是属于“结构化”或者半结构化的数据。例如：带有各种字段的一个订单、用json格式描述的商品信息等。那么，当写入一个订单数据时（包括很多订单条目）【原子性】的数据是订单？还是订单条目？亦或是条目中的某个字段？

通常来说，大部分的企业应用，都是保证“一条Record”记录原子写入即可。那么，怎么定义：一条Record记录？这通常来说，应该是由企业应用的“业务”来定义的，而非技术来定义。例如：一个应用认为，一个订单就是它的【一条记录】，一个订单数据的写入，必须是原子的；而另一个应用则认为，一个订单中的订单条目是它的一条记录，订单可以不用原子写入，而订单条目必须原子写入。

Openstore中每一种FileFamily的Format都提供了其【原子性】记录的界定范围，例如：一个excel的format，其每个单元格字段是【原子性】的；一个RCFile文件，其一行数据是原子的；一个Json文档格式，每个document是原子写入的；每一个应用，应该根据自己的业务，来决定采用何种原子性的Format

* 数据的备份如何保障

通常的企业应用，由于程序猿习俗的关系，一说到备份，就是“手动”或“自动”将数据库或表自动copy一份。但实际上，openstore中的“数据备份”是实时的（至少是准实时的），并且是在上层应用write的同时，透明进行备份的。从结构上来说，openstore至少必须满足如下的备份拓扑结构：

Master节点

Partition节点

Partition节点

Copy

数据读取

数据写入

宕机选举

同步

Openstore启动时候，根据在集群的region的范围内，通过选举算法，将region中的机器选举出master节点。应用的上层数据通常都是写入到Master节点中，openstore通过copy算法，将数据准实时的copy到partition节点中；通常，读取数据都发生在partion节点上。如果master节点宕机（或者partion宕机，后续讨论）则通过选举算法，将某个partition节点选举为mater。

Openstore的这种结构，天然保证读写是分离的，同时，任何一个节点宕机，也不会导致数据的丢失（丢失率随着集群大小而定）；

当然，没有任何一种结构是完美的，这种结构，也会有很多问题，比如选举算法的复杂性，copy时，如果保证事务以及数据的锁等。

* 数据的metadata

在企业应用的“用户层次”，只要将想要保存的数据存储为openstore的一个File即可。但实际上，一个文件File，除了本身的数据内容外，还具有一些描述File本身的所谓的“元数据“metadata。例如：文件的大小、文件的创建者、文件的创建时间、文件的版本、文件的最后修改时间、文件的md5哈西值等。这些meta数据，是在写入数据时，部分自动由系统创建（例如：文件大小、文件创建者），部分由用户自定义添加的。

请注意，很多企业应用开发者，经常混淆数据的metadata和attribute的区别。从概念上讲，数据的metadata和数据的attribute没有什么不同，但从存储格式和区域上来说，他们有着本质的不同：

1. 数据的metadata从类型上来说，只支持字符、整数、浮点、等简单基本类型，不支持复杂的嵌套类型的数据结构；而数据的attribute不同，它可以支持各种复杂的、嵌套的自定义数据结构
2. 如果数据是一个【链接型数据】（软、硬链接），则metadata和指向的目标的metadata是相同的，但attribute却可以不同
3. 从存储分布来说，metadata通常都集中存储在meta中心服务器上，而attribute通常可以分布存储

* 数据的版本和日志

通常的企业应用，保存的数据都是数据的“最终状态“，因为之前的数据都被所谓的update覆盖了。但通过之前的分析，真正的update是不存在的。因此，在write【旧数据】时，实际上，是在原来的数据上增加了一个新版本。这就导致了对于很多大数据场景的应用，实际上，谁可以记录数据所有版本的。并且有了版本，数据还能够作一定的回滚。当然，对于完整的企业应用，回滚比较复杂，还不仅仅涉及到数据本身，还涉及相关的其他数据的版本回滚。但单纯从一条数据记录来说，有版本记录，就意味着这条数据记录是完全可以版本回滚的。

在集群化的数据存储服务中，数据日志起到重要的作用。这里所说的数据日志，并非应用业务上的日志，而是系统对数据本身操作的日志，关系数据库使用日志完成回滚功能，HBase等使用日志完成HFile的合并等功能

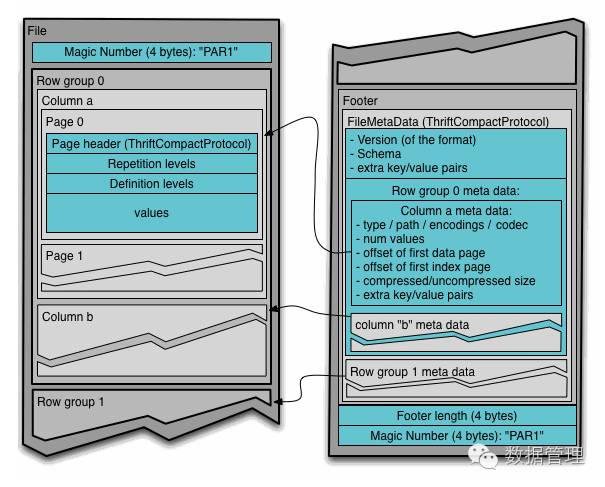
## 数据搜索

### 格式决定了搜索方式和效率

一个完整的数据存储系统，除了要write/read外，还必须有强有力的搜索支持。通常来说，搜索分为两部分：

1. 搜索的算法机制
2. 提供给用户的交互接口

其中搜索算法，都是由文件格式（File的Format）决定的。什么样的文件格式，文件内数据的存储方式，直接决定了数据搜索的机制和算法。例如：Parquet文件格式，就是一种列式存储格式，比较适合嵌套型数据结构的离线历史数据的存储和搜索，其存储格式如下所示：

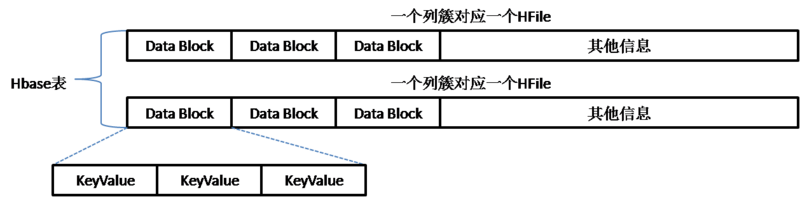
openstore中每一种文件格式，都有其自身的存储特点，从而决定了其搜索特点以及使用的场景，下面将针对目前所有传统以及大数据平台下的各类文件的搜索场景和特点做一个简略分析：

##### Hbase系

HBase是建立在Hadoop计算平台上的存储系统，支持实时的查询、随机写等操作。HBase从上层应用角度，和关系库一样，将数据划分到“表”中，但和关系库不一样的是，他是一种列式存储文件，而非行存储。他的表包含的概念有rowkey、列簇、列限定符、版本(timestamp)和值。如图：

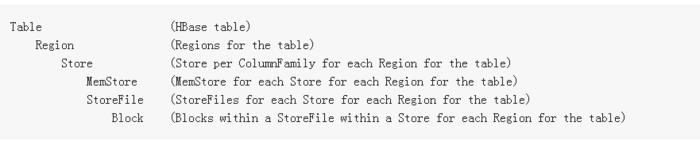


Hbase表中的每一个列簇会对应一个实际的文件，每个文件有若干个DataBlock(数据块默认64k)，DataBlock是HBase中数据存储的最小单元，DataBlock中以KeyValue的方式存储用户数据（KeyValue后面有timestamp，图中未标注），其他信息主要包含索引、元数据等信息，在此不做深入探讨。每个KeyValue都由4个部分构成，分别为key length，value length，key和value。其中key的结构相对复杂，包括rowkey、列、KeyType等信息，而value值对应具体列值的二进制数据。为了便于查询，对key做了一个简单的倒排索引，直接使用了java的ConcurrentSkipListMap。

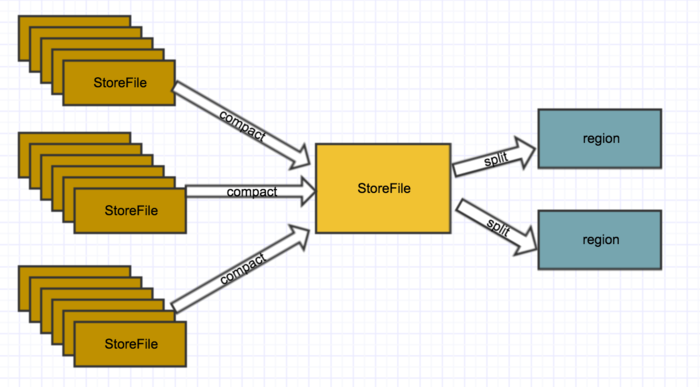


Hbase会根据Rowkey的范围不同，分散到不同的Region，Region又按照列簇分为不同的Store，每个Store实际上又包括StoreFile(对应Hfile)和MemStore，然后由RegionServer管理不同的Region，RegionServer即对应具体的进程，分散不同的机器，提供分布式的存储和查询。查询时，首先获取meta表（一种特殊的Region）所在的RegionServer，通过meta表查找表rowkey相对应的Region和RegionServer信息，最后连接数据所在的RegionServer，查找到相应的数据

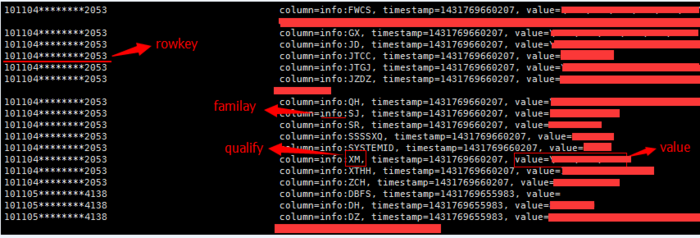
由此可以看到：HRegion和HStore是HBase存储的核心逻辑概念。他们之间的层次关系如下：



Table会根据rowkey分散到每个HRegion中，而每个HRegion又将每个列族分别存储为每个HStore进行存储，HStore又由两部分组成：MemStore和StoreFile。MemStore是一个写缓冲区，用户写入的数据首先会被放入MemStore中，当MemStore满了就会flush成一个StoreFile。达到一定数量的StoreFile后，会触发compact操作，将多个StoreFile合并成一个更大的StoreFile，在合并的过程中会进行版本的合并和数据的删除。当单个StoreFile大到一个阈值时(跟Region的maxSize相关)，就会进行split操作。将Region分裂成两个Region，父的Region会下线，新分裂的两个Region会被HMaster分配到相应的HRegionServer上。compact和split示意图：



StoreFile底层使用HFile存储实际的数据,HFile是Hbase的底层存储，所有数据以固定的数据格式存储在HDFS的特定目录下。我们来看看HFile的结构：



可以看到，HFile基本格式为： rowkey：timestamp:columm\_family:coloum:value.

逻辑上的一行数据被拆成了两行数据进行存储，同时存储的数据大大的膨胀了主要是冗余了rowkey timestamp 和coloumFamily字段，通常数据会膨胀2倍以上，具体大小视具体的数据。

同时，正是因为HFile的分裂和合并机制，满足了前面章节中描述的【数据写】机制：实际的随机写是不存在的，通过追加写，可以满足上层的随机写需求。

Hbase的这种结构，特别适合根据rowkey做单值查询，不适合scan的场景，因为大部分Scan的情况基本上需要扫描所有数据，性能会非常差。虽然也有扩展的Hbase二级索引方案，但基本上都是通过协处理器，需要另外建立一份rowkey的对应关系，Scan的时候先通过二级索引查找rowkey，然后在根据rowkey查找相应的数据。

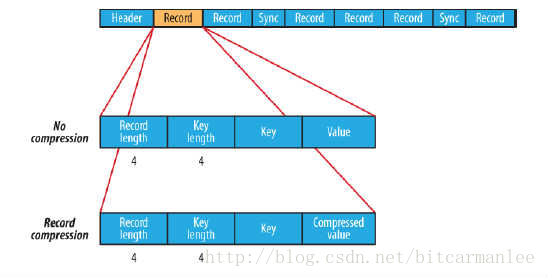
##### HDFS系

Hadoop系下的文件格式，是最丰富的。因为业界已经将hadoop作为大数据计算平台的事实标准了，而hadoop的文件系统HDFS，几乎大部分存储都是基于HDFS，HBase也不例外。下面我们就一一来看看HDFS之上的各种文件格式

###### SequenceFile

SequenceFile是Hadoop API 提供的一种二进制文件，它将数据以<key,value>的形式序列化到文件中。这种二进制文件内部使用Hadoop 的标准的Writable 接口实现序列化和反序列化。

在存储结构上，SequenceFile主要由一个Header后跟多条Record组成，Header主要包含了Key classname，value classname，存储压缩算法，用户自定义元数据等信息，此外，还包含了一些同步标识，用于快速定位到记录的边界。每条Record以键值对的方式进行存储，用来表示它的字符数组可以一次解析成：记录的长度、Key的长度、Key值和value值，并且Value值的结构取决于该记录是否被压缩



对于顺序文件，大家很容易理解。也有很多资料可以参考。但这里主要说明SequenceFile的几个有趣特性：

* 顺序的存储Key/Value

通常key为字符串，value可以任意的数据。他们按照顺序（可能压缩称块）存放在一个文件中。并且，只能append追加。但需要说明的是，普通的SequenceFile是没有索引的。因此，无法直接搜索指定的数据。

* Sync同步点

顺序文件中，为了方便的定位，引入了同步点。SequenceFile文件中，每2k字节的记录边界上就会插入一个同步点。有了同步点，在顺序文件中搜索给定位置有两种方法。第一种是调用**seek**()方法，该方法将读指针指向文件中指定的位置。例如，可以按如下方式搜査记录边界：

·· reader.seek(359);

assertThat(reader.next(key, value), is(true));

assertThat(((IntWritable) key).get(), is(95));

但如果给定位置不是记录边界，调用**next**()方法时就会出错！

第二种方法通过同步点査找记录边界。SequenceFile.Reader对象的 sync(long position)方法可以将读取位置定位到position之后的下一个同步点。如果position之后没有同步了，那么当前读取位置将指向文 件末尾。这样，我们对数据流中的任意位置调用sync()方法(例如非记录 边界)而且可以重新定位到下一个同步点并继续向后读取：

|  |  |
| --- | --- |
| 1  2  3  4 | reader.sync(360);  assertThat(reader.getPosition(), is(2021L));  assertThat(reader.next(key, value), is(true));  assertThat(((IntWritable) key).get(), is(59)); |

* 顺序文件中数据的搜索

通过对顺序文件特性的分析，我们可以看到，原始的顺序文件比较适合如下场景：

1. 日志型数据
2. 固定不变，只追加数据的场景
3. 只对全部数据进行分析计算的场景，而不是查找数据的场景

那么，如果要在顺序文件中搜索数据，应该如何做到呢？就只能通过外部记录key的位置信息，然后通过同步点定位了！这就催生了：带索引的顺序文件----MapFile的出现【当然，简单场景也可以自己外部记录key位置】

###### MapFile文件

MapFile是SequenceFile的变种，在SequenceFile中加入索引并排序后就是MapFile。索引作为一个单独的文件存储，一般每个128个记录存储一个索引。索引可以被载入内存，用于快速查找。存放数据的文件根据Key定义的顺序排列。   
MapFile的记录必须按照顺序写入，否则抛出IOException。

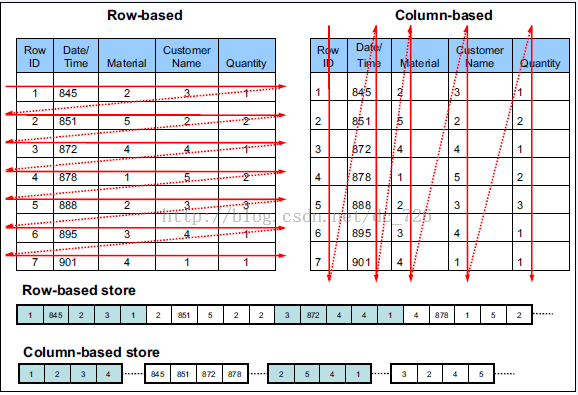
MapFile的衍生类型：

* SetFile:特殊的MapFile，用于存储一序列Writable类型的Key。Key按照顺序写入。
* ArrayFile：Key为整数，代表在数组中的位置，value为Writable类型。
* BloomMapFile：针对MapFile的get()方法，使用动态Bloom过滤器进行优化。过滤器保存在内存中，只有带key值存在的时候，才会调用常规的get()方法，真正进行读操作。

numbers.map是一个目录，包含data和index两个文件。index记录了key和它们在data文件的offset。这些key并不是所有的key，而是一种采样的key，默认值是128间隔取一个key。

###### ORCFile文件

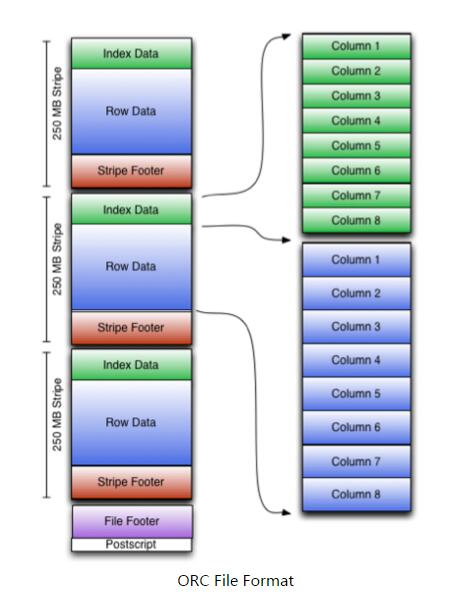
ORC文件从RCFile中衍化而来。是典型的列式存储结构。



从图中可以看到：

* 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。

ORCFile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。它具有如下结构：



1. 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。
2. 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。
3. 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。
4. 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。
5. 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。
6. 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。

* 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：

1. 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
2. DATA：该列的中属于当前stripe的成员值。
3. LENGTH：每一个成员的长度，这个是针对string类型的列才有的。
4. DICTIONARY\_DATA：对string类型数据编码之后字典的内容。
5. SECONDARY：存储Decimal、timestamp类型的小数或者纳秒数等。
6. 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。

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 Foobar (

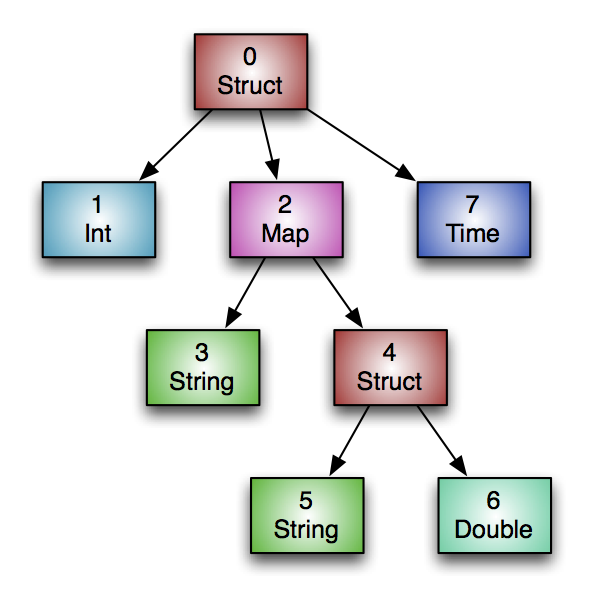
myInt int,

myMap map<string, struct<myString: string,myDouble: double>>,

myTime timestamp

);

Orcfile文件的schema如下：



下面，我们来仔细分析一下ORCFile文件中的各个部分的格式：

* 首先，我们简单介绍以下ORCFile中的**Index索引部分**

Orcfile文件中的索引有三个层次

1. 文件级：它记录了整个文件中每列的统计信息
2. Strip级：记录每个strip中每列的统计信息
3. 行级：每个strip中每10000行的每列的统计信息

文件和strip级的index索引，都是记录在file footer中。因此，他们很容易访问到以便决定是否需要读取文件的其他部分。行级索引则记录了每一个行组的统计信息以及每个行组的偏移位置。

Index中列的统计信息包括以下内容：

* Value的数目
* 是否为null值
* 大部分原始类型的最大、最小值以及数字类型的sum求和值
* Bloom 过滤器，以便进行选择过滤

所有的index中的信息，都是用于orcfile文件读取中的搜索参数的，例如：如果需要查询一个age > 100的pepole，则表达式：“age > 100”将会导致文件、strip、行组中age超过100的数据被读取

* 再来看看orcfile的**FileTail文件尾**

在第二章节已经描述过，大部分文件系统不存在实际的update、delete数据的，都是一次性写入的。HDFS也不例外，因此，orcfile将顶层索引信息存在了文件尾部。【当然，现实需求中不可能没有update、delete；所以，需要增加额外的机制对上层应用支持，hive中增加了orcfile的basefile 和delta增量文件的方式向上层支持更新、删除数据的操作】

Orcfile文件的文件尾由三部分组成：

* 文件元数据metadata
* File Footer
* Postscript

FileTail文件尾的信息都是使用protobuf协议记录的，下面我们就依次来看看这三部分的具体内容：

* **Postscript**

postscript提供了必要的信息来解释文件的其他部分，包括FileFooter的长度、元数据区、文件版本、以及压缩方式（zlib等）。Postscript是不压缩的，且为文件最后几个字节，postscript中存储的版本是能够保证hive正确读取文件的一个最低版本，按照major主版本和minor次版本的顺序存储：例如，[0,12]。

orcfile的读取过程，首先是从最后部分的postscript读取的。先读取16K的部分，文件最后一个字节包含了序列化后的postscript长度，它必须小于256。一旦postscript被解析成功，则压缩的序列化的footer长度就能够被解压缩以及被解析。Postscript结构定义如下：

message PostScript {

//FileFooter区的长度

optional uint64 footerLength = 1;

// orcfile文件的压缩方法

optional CompressionKind compression = 2;

// 每一个压缩块的最大大小

optional uint64 compressionBlockSize = 3;

// 写入器的版本

repeated uint32 version = 4 [packed = true];

// 元数据区的长度

optional uint64 metadataLength = 5;

// 固定字符串 "ORC"

optional string magic = 8000;

}

enum CompressionKind {

NONE = 0;

ZLIB = 1;

SNAPPY = 2;

LZO = 3;

LZ4 = 4;

ZSTD = 5;

}

* **FileFooter**

Footer区包括orcfile文件的内容布局、行总数、类型schema以及列的统计信息等。文件都分为三个部分：Header、Body、Tail。

Header包含“ORC”固定字节，以便支持工具来识别文件的类型。

Body包含行和索引，而Tail则给出了在此区域中描述的文件层级信息

Footer的结构很简单：

message Footer {

// 文件头的字节长度 (总是固定3)

optional uint64 headerLength = 1;

// 文件body内容的字节长度

optional uint64 contentLength = 2;

// strips的信息

repeated StripeInformation stripes = 3;

// orcfile中每列的类型schema

repeated Type types = 4;

// 添加的用户元数据

repeated UserMetadataItem metadata = 5;

// 文件中的行总数

optional uint64 numberOfRows = 6;

// 文件统计信息

repeated ColumnStatistics statistics = 7;

// 在每一个index实体中的最大行数

optional uint32 rowIndexStride = 8;

}

* **StripeInformation**

文件的body部分被切分为多个stripe。每个stripe都是自包含的，且读取时，仅仅只需要联合使用postscript、footer和自身的字节信息即可读取。每个stripe只包含行实体。因此，行从来不会跨越stripes。Stripes由三部分组成：stripe内行的index集合、数据区、strip footer。Index和数据区都是按照列来存储的，因此读取时，只需要读取请求的列即可。Strip的定义如下：

message StripeInformation {

// 此stripe的起始位置

optional uint64 offset = 1;

// index索引区的长度

optional uint64 indexLength = 2;

// 数据区的长度

optional uint64 dataLength = 3;

// stripe footer的长度

optional uint64 footerLength = 4;

// stripe中行的数目

optional uint64 numberOfRows = 5;

}

* **Type schema**

Orcfile文件中的行都具有相同的schema类型定义。Schema的定义在前面已经介绍过，这里不再描述。

类型树会被平展成一个预排序的列表，其中每个类型都分配好下一个id。显然，root的id永远为0，复合类型是具有子类型命名的域，这些域包含子类型id列表。Type schema定义如下：

message Type {

enum Kind {

BOOLEAN = 0;

BYTE = 1;

SHORT = 2;

INT = 3;

LONG = 4;

FLOAT = 5;

DOUBLE = 6;

STRING = 7;

BINARY = 8;

TIMESTAMP = 9;

LIST = 10;

MAP = 11;

STRUCT = 12;

UNION = 13;

DECIMAL = 14;

DATE = 15;

VARCHAR = 16;

CHAR = 17;

}

// 此schema的类型

required Kind kind = 1;

// 列表、map、结构体、union的子类型ids

repeated uint32 subtypes = 2 [packed=true];

// 结构体中的域名称列表

repeated string fieldNames = 3;

// 字符或可变字符的最大长度

optional uint32 maximumLength = 4;

// decimal 的整数位数和小数位数

optional uint32 precision = 5;

optional uint32 scale = 6;

}

* **ColumnStatistics列统计**

列统计信息的目标，是对于每一列，写入器都能够根据类型以及相关字段来记录count等信息。

例如：对于大多数原始类型，都记录了此列中的最小值和最大值；

对于数字类型，额外记录了sum信息；

Hive 1.1.0开始，还记录了在行组的hasNull标记的设置下，是否有null值；hasNull标记用于orc谓词下推来更好的执行null查询

**ColumnStatistics的结构如下：**

message ColumnStatistics {

// 值的数目

optional uint64 numberOfValues = 1;

// 对任意的列，根据类型，以下至多有一个值

optional IntegerStatistics intStatistics = 2;

optional DoubleStatistics doubleStatistics = 3;

optional StringStatistics stringStatistics = 4;

optional BucketStatistics bucketStatistics = 5;

optional DecimalStatistics decimalStatistics = 6;

optional DateStatistics dateStatistics = 7;

optional BinaryStatistics binaryStatistics = 8;

optional TimestampStatistics timestampStatistics = 9;

optional bool hasNull = 10;

}

整数的结构，包括最大、最小、sum总和（如果总和超过lang，则将不被记录）

message IntegerStatistics {

optional sint64 minimum = 1;

optional sint64 maximum = 2;

optional sint64 sum = 3;

}

浮点、小数、字符串类似：

message DoubleStatistics {

optional double minimum = 1;

optional double maximum = 2;

optional double sum = 3;

}

message DecimalStatistics {

optional string minimum = 1;

optional string maximum = 2;

optional string sum = 3;

}

message StringStatistics {

optional string minimum = 1;

optional string maximum = 2;

//字符串的sum存储所有字符串的总长度

optional sint64 sum = 3;

}

布尔值包括真假的统计

message BucketStatistics {

repeated uint64 count = 1 [packed=true];

}

日期记录自从2015/1/1号以来的最小、最大天数：

message DateStatistics {

optional sint32 minimum = 1;

optional sint32 maximum = 2;

}

时间戳记录自从2015/1/1号以来的最大、最小毫秒数

message TimestampStatistics {

optional sint64 minimum = 1;

optional sint64 maximum = 2;

}

二进制存储列中所有blob的字节数

message BinaryStatistics {

optional sint64 sum = 1;

}

* **UserMetadataItem**

用户可以向ORC文件中添加任意的key/value对，key/value的内容完全由应用定义，但key的类型必须是string，value的内容必须是binary。应用应该确保key是独一无二的，通常可以使用一个业务领域或组织唯一前缀来确保唯一性。

message UserMetadataItem {

required string name = 1;

required bytes value = 2;

}

* **文件元数据**

File metadata区，包含了在stripe粒度级别下的列统计信息。这些统计数据，可以让输入split，从而可以基于预先谓词下推来评估每个stripe

message Metadata {

repeated StripeStatistics stripeStats = 1;

}

message StripeStatistics {

//stripes级别下的列统计数据

repeated ColumnStatistics colStats = 1;

}

关于FileTail文件尾的结构，我们就介绍到这里，下面我们重点来看看ORCFile中核心的stripe的结构：

* **Stripe**

Orcfile文件的body由一系列的stripe组成，一个stripe通常都大于200M。他们互相独立，且通常由不同任务处理。

在orcfile中，每一列都使用streams的方式存储。例如：一个整数列使用两个stream来描述。PRESENT stream描述整数的每个bit，而DATA stream记录整数的value值。而二进制数据，则使用三个stream来描述：PRESENT/DATA/LENGTH，stream的定义如下：

message Stream {

enum Kind {

// 决定下一个值是否为非null的boolean stream

PRESENT = 0;

// 主data stream

DATA = 1;

// 用于变量的长度数目的每个值的长度

LENGTH = 2;

// blob字典

DICTIONARY\\_DATA = 3;

// 用于存储字典中每个值的实例数目，hive 0.11中取消

DICTIONARY\_COUNT = 4;

// 次数据stream

SECONDARY = 5;

// 用于定位特定行组的index索引

ROW\_INDEX = 6;

// 用于ORC-101规范之前的原始bloom过滤器

BLOOM\_FILTER = 7;

// 使用utf8组成的bloom过滤器

BLOOM\_FILTER\_UTF8 = 8;

}

required Kind kind = 1;

// 列 id

optional uint32 column = 2;

// 文件中列的字节长度

optional uint64 length = 3;

}

Stream中存储的内容细节，由列的类型和内容编码决定

* StripeFooter

Stripe页脚包含每列的编码以及包含他们位置的stream目录，定义如下：

message StripeFooter {

// 每个stream的位置

repeated Stream streams = 1;

// 每个列的编码

repeated ColumnEncoding columns = 2;

}

依赖于类型的几个选项，列编码分为直接的和基于字典的类别，定义如下：

message ColumnEncoding {

enum Kind {

// 使用RLE V1版本的直接定位到stream的编码

DIRECT = 0;

// 使用RLE V1版本的具有唯一值的字典进行编码

DICTIONARY = 1;

// 使用RLE V2版本的直接编码

DIRECT\\_V2 = 2;

// 使用RLE V2版本的字典编码

DICTIONARY\\_V2 = 3;

}

required Kind kind = 1;

// 对于字典编码，记录字典的大小

optional uint32 dictionarySize = 2;

}

下面我们就来看看，各种类型的列编码：

* Smallint、Int、BigInt列

如果不包含PRESENT Stream，则Data Stream中的所有值都将被包括；否则如果PRESENT Stream中具有false的bit位，则Data Stream中就不包含相应的值。

* Float与double列

使用IEEE 754浮点编码。Float使用4个字节，double使用8个字节

* String、char、varchar列

字符串可以选择使用直接编码或者字典编码；当列中由许多不同值时，直接编码是首选。所有的编码，PRESENT Stream决定值是否为null。Java orc写入器在第一个行组（10000行）之后自动选择编码。

直接编码的UTF8字节，保存在DATA Stream中，而每个值的长度写入到LENGTH Stream中。例如：如果值为(“内华达”、“加州”); DATA Stream为“NevadaCalifornia”，LENGTH Stream为[6、10]。

字典编码，则在DICTIONARY\_DATA中只存储不同的值，在LENGTH Stream中存储每个不同值的长度，而DATA Stream则存储每个不同值在字典中的位置。例如：[“Nevada”, “California”, “Nevada”, “California”, and “Florida”]，则DICTIONARY\_DATA为California｜Florida｜Nevada，（“｜“只是为了让读者方便明白字典编码的分组而手动添加，实际并不存在），LENGTH Stream为(10、7、6)，10为字典中第一个分组长度，7为第二个分组长度….;DATA Stream为：[2,0, 2, 0, 1], 2标示第一个值为字典数组中索引为2的组，0标示第二个值为字典数组中索引为0的组……

* Boolean，TinyInt小整数列

布尔和tinyint列非常简单，直接使用DATA Stream编码值即可

* 二进制列

DATA stream 记录二进制字符内容，LENGTH stream记录字节数，PRESENT Stream是可选的

其他格式的列，请参考规范文档，这里不做过多介绍。

下面我们来看看Strip中的行组索引：

* **Row Group Index**

行组索引，是对每一个原始列的ROW\_INDEX Stream组成。行组是由writer写入器控制的，默认是10000行。每个RowIndexEntry对象都标识了列在Stream中的位置以及行组的统计信息。Index Stream在stripes的最前部，因为通常他们不需要被读取，他们通常在谓词下推或者reader需要定位到特定行时才被加载。下面看看RowIndexEntry的结构定义：

message RowIndexEntry {

//每个列中行的位置

repeated uint64 positions = 1 [packed=true];

optional ColumnStatistics statistics = 2;

}

message RowIndex {

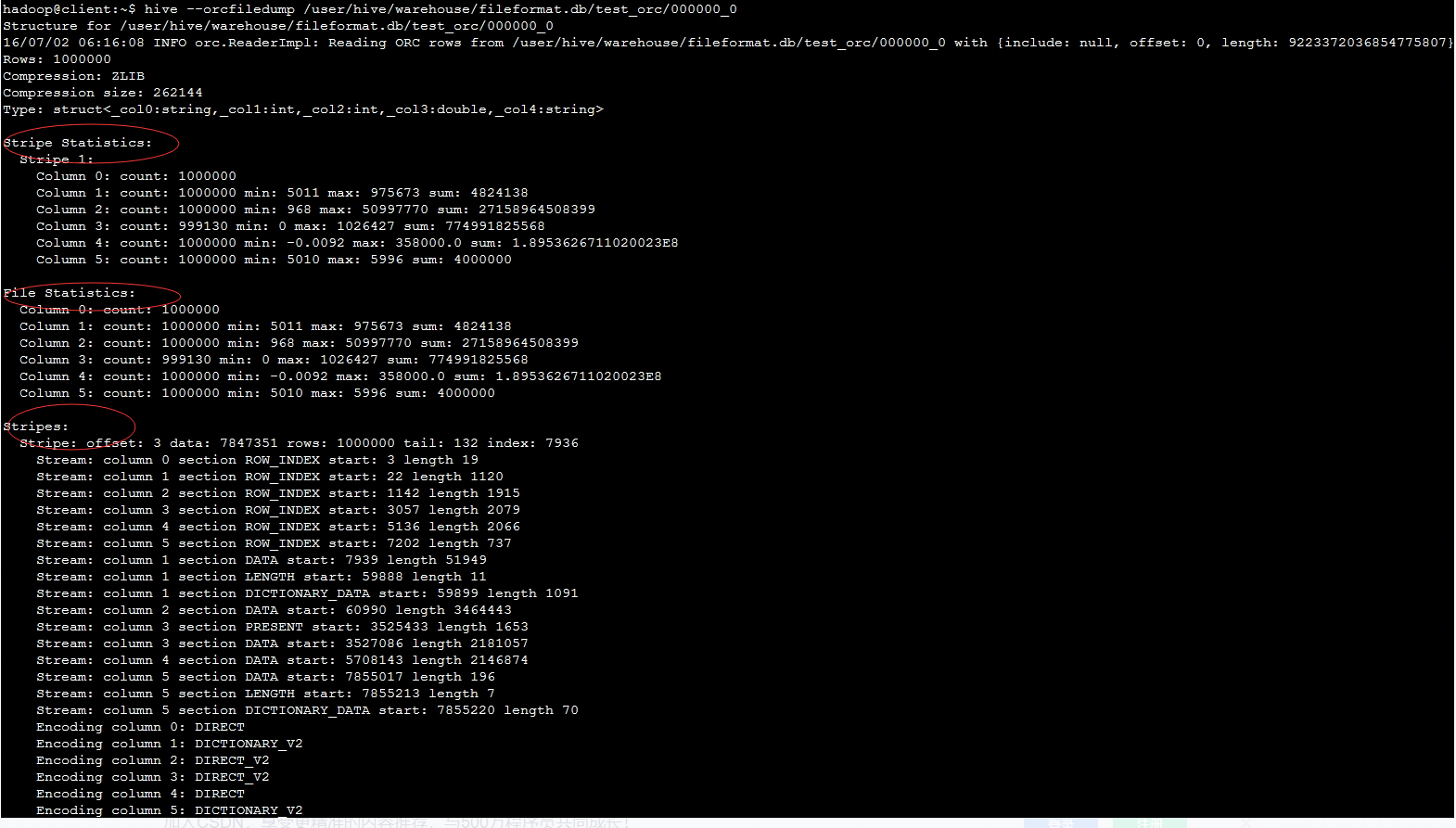
repeated RowIndexEntry entry = 1;

}

为了记录位置，每个stream都需要一个数字序列。对于未压缩的流，位置是RLE开始运行时到消费的字节的偏移量。而在压缩的Stream中，是压缩块的起始位置，后面跟着一个解压缩后需要消费得字节数，以及RLE中最终消费得值得数目。

对于具有多个流得列，每个流中得位置序列是连接在一起的。而对于字典，因为是随机访问的，没有字典的位置记录，因此即是读取部分stripe，也需要把整个字典都读取。

介绍完了ORC 最重要的Stripe的核心结构，下面我们来看看完整的stripe信息，如下图所示：



第三段则为stripes的详情，包括了各Stripe的offset、strip中总记录行数等Stripe层次的统计信息。从图中可以看到：每个字段的ROW\_INDEX以及DATA信息是保存在一块连续空间中的，这块文件从offset=3开始。

假设图中的数据结构如下：

| **字段** | **类型** |
| --- | --- |
| category\_id | string |
| product\_id | int |
| brand\_id | int |
| price | double |
| category\_id\_2 | string |

从图中可以看出，Stripe的行组Index区的数据记录如下：

| **起始位置** | **字段** |
| --- | --- |
| 3……21 | STRUCT |
| 22……1141 | category\_id |
| 1142……3056 | product\_id |
| 3057……5135 | brand\_id |
| 5136……7201 | price |
| 7202……7938 | category\_id\_2 |

RowData数据区的记录如下：

| **起始位置** | **字段** | **描述** |
| --- | --- | --- |
| 7939……59887 | category\_id | 字段对应词条int流 |
| 59888……59898 | category\_id | 词条长度int流 |
| 59899……60989 | category\_id | 字典词条数据 |
| 60990……3525432 | product\_id | 实际数据int流 |
| 3525433……3527085 | brand\_id | 标识IF NULL的byte流 |
| 3527086……5708142 | brand\_id | 实际数据int流 |
| 5708143……7855016 | price | double类型 |
| 7855017……7855212 | category\_id\_2 | 字段对应词条int流 |
| 7855213……7855219 | category\_id\_2 | 词条长度int流 |
| 7855220……7855289 | category\_id\_2 | 字典词条数据 |

最后一部分，记录了每个字段的编码方式，例如字典编码等。根据这些编码方式，以及PRESENT/DATA/LENGTH Stream流的数据，即可计算出每个stripe中行组中存储的每个字段的值！

###### Parquet文件

上一节仔细分析了ORCFile文件，ORCFile文件可以说展现了比较强劲的性能和优点，许多计算引擎都支持ORC文件，例如hive、impala、spark、Presto等。下面我们来看看另外一个列式存储的格式：parquet

首先，设计parquet文件格式的动机是：希望在hadoop生态系统上创建一个高压缩率，高效率的，可以有效描述任何项目中任意数据格式的列式存储系统。Parquet专用于描述复杂的嵌套数据结构，它使用google 的dremel系统论文中描述的record shredding and assembly algorithm算法来描述复杂的嵌套数据模型。Pqrquet可以被在hadoop生态系统中的任何数据处理框架使用，而不需要和任何其他库、包、框架依赖。

Parquet项目分为几个模块：

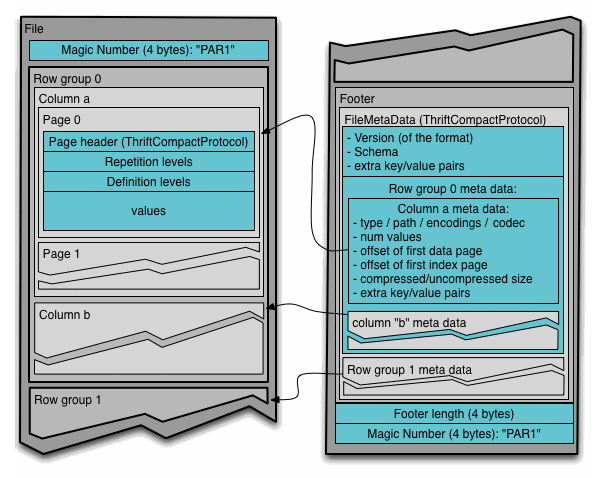
* Parquet-format:包含parquet文件的格式规范以及thrift定义的读取parquet文件的元数据定义
* Parquet-mr:包含多个子模块，用于读写嵌套的、列式存储的数据流的核心组件，并将这些核心组件映射到parquet格式。并提供hadoop上的Input/Output Formats, Pig loaders，以及其他与parquet交互的其他java工具
* Parquet-mr:提供读写parquet的c++库
* Parquet-compatibility： 兼容性测试项目，用于验证在不同语言中能够读写其他语言中的parquet文件
* **Parquet中的概念**
* 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逻辑单元，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，所以每一个行组的大小是由内存大小决定的。
* 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。

从层级上看，一个parquet文件由一个或多个行组组成，一个行组至少包含每一列的一个列块，一个列块包含一个或多个page页

* **Parquet文件格式与thrift定义**
* 4-byte magic number "PAR1"
* <Column 1 Chunk 1 + Column Metadata>
* <Column 2 Chunk 1 + Column Metadata>
* ...
* <Column N Chunk 1 + Column Metadata>
* <Column 1 Chunk 2 + Column Metadata>
* <Column 2 Chunk 2 + Column Metadata>
* ...
* <Column N Chunk 2 + Column Metadata>
* ...
* <Column 1 Chunk M + Column Metadata>
* <Column 2 Chunk M + Column Metadata>
* ...
* <Column N Chunk M + Column Metadata>
* File Metadata
* 4-byte length in bytes of file metadata
* 4-byte magic number "PAR1"

上例中，N列被划分成M个行组。文件元数据，包含所有列元数据的开始位置，元数据中的更多细节，可以在thrift文件中找到。读取器首先读取文件元数据找到需要读取的所有列块，然后列块被顺序的读取。下面是parquet文件的详细结构图：



下面，我们来仔细分析parquet文件中的各种结构：

* **File Footer**

我们可以看到，parquet文件和orc文件一样，都是在文件末尾存储各类元数据信息。之所以都如此，就是因为在hadoop系的各种列式、高性能存储中，都有一个假设：数据只会追加，而不会改变！甚至parquet都假定数据一次性写入后，都不会任何变化！这就给数据的分析，以及文件存储格式的定义带来了无限的好处。

从图中可以看到，Footer由几个部分组成：

* + **Footer Length**

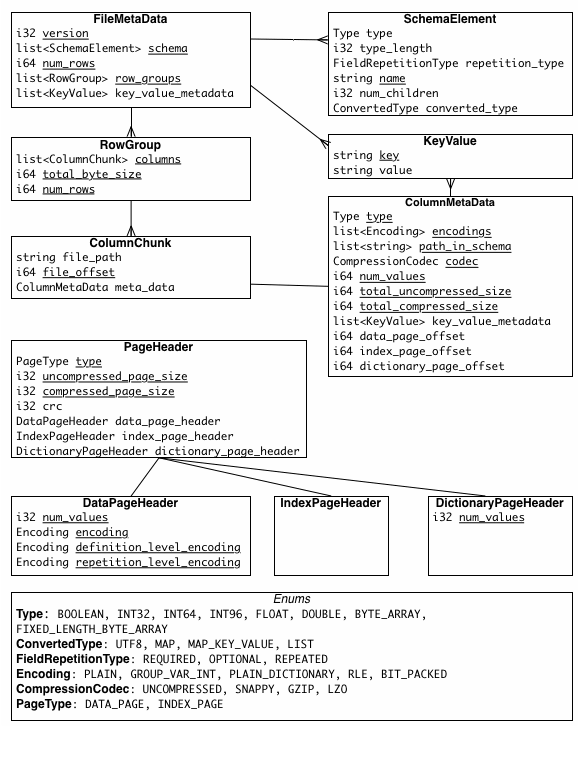
存储了文件元数据的大小，通过该值和文件长度可以计算出元数据的偏移量。

* + **FileMetadata**

文件的元数据中首先包括当前文件的Schema信息、额外自定义的key-value数据、以及每一个行组的元数据信息

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。

FileMetadata结构如图所示：



* **Primivite Types类型**

文件格式支持的类型应该尽可能最小，主要要考虑磁盘存储的类型影响。例如：parquet不显式的支持16位整数，而采用更高效的32位编码的整数。这可以显著的降低reader和writer的实现难度。Parquet支持以下类型：

* BOOLEAN: Bit Packed, LSB first
* INT32: 4 bytes 小端存储
* INT64: 8 bytes 小端存储
* INT96: 12 bytes 小端存储
* FLOAT: 4 bytes IEEE 小端存储
* DOUBLE: 8 bytes IEEE 小端存储
* BYTE\_ARRAY: 4 bytes 任意字节的数组
* FIXED\_LEN\_BYTE\_ARRAY: 纯字符字节
* **逻辑类型Parquet Logical Type Definitions**

逻辑类型通过指定如何解释primivite类型，用于扩展parquet存储的类型，使得parquet支持复杂的数据类型。通过这种方式，可以保持parquet最小的primivite类型以及丰富的可重用编码。例如：string字符类型存储为ByteArray，但使用UTF8编码注释解释。这些注释（Annotion）在文件的元数据中，存储为一个ConvertedType

##### Lucence系列（ElasticSearch为代表）