文件系统设计说明

我们所设计的分布式文件系统在架构上与GFS有许多相似之处，但我们从上层应用的实际需求出发，对文件系统的设计做了一些改进。在本文档中，我们将采取自顶向下的叙述方式。首先从向上层应用提供的模型出发，分析应用的需求，并基于应用需求提出改进的文件模型。随后，从系统顶层架构入手，描述应用API的宏观流程，并分析系统中各个角色的具体实现是如何支撑应用API的实现的。最后，简要描述一些我们在设计过程中曾经调研但由于时间问题未能实现的问题，以作为未来的改进方向。

文件模型

在设计我们的文件系统时，我们并非采取构建专有系统的思路，亦即提供一种通用的文件系统API，而上层的应用需要修改自身的设计以适应底层文件系统所提供的API；相反，我们采取构建领域系统的方法，即从上层应用的需求入手，分析并提取上层应用的需求与现有文件系统API的语义不匹配之处，并提出新的API和系统设计，以更好地支持上层应用的开发。

上层应用的特征

在gDoc部分，我们选择构建一个多人合作表格编辑应用。在对几种电子表格应用进行研究后，我们观察到，**传统文件系统提供的API与文档编辑应用的需求存在巨大的差距。**要理解这一点，首先就需要理解现有的电子表格应用是如何实现将表格持久化存储为文件的。我们调研了两种电子表格文件格式，包括我们实际采用的前端组件Luckysheet的文件格式，以及Microsoft Excel所采用的Spreadsheet ML文件格式。

Luckysheet [1]

Luckysheet接受的文件格式是基于JSON的文本文件格式。每一个表格可以表示为下面的JSON对象：

```
{
    "name": "Cell", //工作表名称
    "color": "", //工作表颜色
    "index": 0, //工作表索引
    "status": 1, //激活状态
    ...
    "row": 36, //行数
    "column": 18, //列数
    "defaultRowHeight": 19, //自定义行高
    "defaultColWidth": 73, //自定义列宽
    "celldata": [], //初始化使用的单元格数据
    ...
},
```

出于篇幅考虑，我们略去了部分对象属性。总而言之，该对象可以分为两大部分，即**表格元数据**和**单元格数据**。前者主要由一系列JSON键值对组成，而后者是一个数组，包含一系列JSON对象，每个对象表示一个单元格的数据。单元格数据对象如下所示：

```
{
    "r": 0,
    "c": 0,
    "v": {
        ct: {fa: "General", t: "g"},
        m:"value1",
        v:"value1"
    }
}
```

一个单元格数据对象由三项组成。“r”表示单元格所在的行，“c”表示单元格所在的列，“v”是一个表示该单元格中的具体数据以及格式信息等的JSON对象。由于实际表格大部分都是离散的，对于空单元格，Luckysheet并不会保存它们的数据，只有当用户实际修改一个单元格时，Luckysheet才会将被修改单元格的数据加入到数组中，或修改数组中已有的单元格对象。

对于文件系统的设计，我们可以得出三个关键的观察：

- **单元格顺序是自包含的。**换言之，Luckysheet并不依赖单元格对象在数组中的顺序，来决定单元格在表格中的位置，而是通过单元格对象自身的数据来决定。
- **JSON对象是无序的。**JSON对象中，不同键值对的相对顺序，并不影响JSON的解析。
- **存在大量就地修改与文件中部插入操作。**在修改一个已有的单元格时，Luckysheet将会期望就地应用对文件的修改。而新增一个单元格时，Luckysheet会在单元格数组的末尾进行插入。然而，单元格数组的末尾并非文件的末尾，而是文件的中部。

Spreadsheet ML [2]

Microsoft Excel所采用的Spreadsheet ML文件格式是一种基于XML的文本文件格式。其与表格相关的XML如下所示：

```
<worksheet . . .>
   . . .
   <cols>
      <col min="1" max="1" width="26.140625" customWidth="1"/>
      . . .
   </cols>
   <sheetData>
      <row r="1">
         <c r="A1" s="1" t="s">
            <v>0</v>
            . . .
         </c>
      </row>
      . . .
   </sheetData>
   . . .
   <mergeCells count="1">
      <mergeCell ref="B12:J16"/>
   </mergeCells>
   <pageMargins . . ./>
   <pageSetup . . ./>
   <tableParts ccount="1">
      <tableParts count="1">
   </tablePart r:id="rId2"/>
</worksheet>
```

该文件格式同样可以被划分为**表格元数据**和**单元格数据**两部分。单元格数据主要由`<sheetData>`标记表示，其子元素为一系列`<row>`标记，每个`<row>`标记表示一行的数据。后者的子元素为一系列`<c>`标记，每个`<c>`标记表示一个单元格的数据。在新增和修改单元格时，Spreadsheet ML的需求与Luckysheet类似。

不难看出，Spreadsheet ML的文件格式同样符合Luckysheet文件格式的**单元格顺序自包含**，**元数据标记无序**与**存在大量就地修改与文件中部插入操作**三个影响文件系统设计的关键特性。

传统文件系统的问题

我们所调研的两种表格文件格式均包含大量的就地修改和中部插入操作，不仅如此，文档（docx）或幻灯片格式也需求大量的类似操作。而传统文件系统所提供的基于Offset（偏移量）的API无法很好地支撑这些应用的需求。根本原因在于，传统文件系统所提供的文件抽象是一段**线性**且**连续**的字节流，这是与它们向上提供的基于Offset的API相适应的。而相反地，在上面的分析中我们可以看到，文档文件格式对于整个文件的连续性并没有要求，或者说它们只要求**某个最小单元**的连续性（例如一个单元格对象中的数据不能乱序），而整个文件的数据并不需要是连续的。而这些文件在修改时执行的就地修改与中部插入操作，从传统文件系统的角度而言，正是破坏了文件的连续性。例如，在就地修改一个单元格时，若新的单元格对象的长度大于原对象的长度，则必须在原本的文件位置“插入”一段新的字节；中部插入操作更是显然。正因如此，作为上层应用，为了适应传统文件系统提供的连续性抽象，就必须采取一些相对低效的方式，例如首先将插入位置之后的字节全部

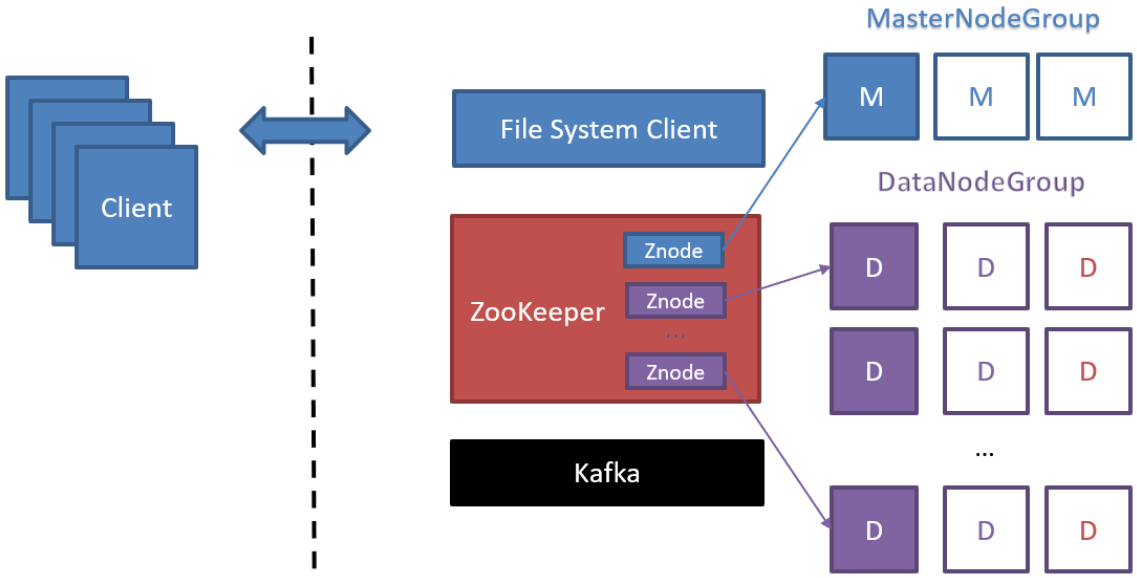
读入内存，然后首先写入需要插入的数据，再将原有数据写回。另一种方式是首先将文件完整读入内存，并在内存中应用修改，定期将修改后的文件完整刷回磁盘。这两种方式虽然能够适应底层的传统文件系统，但却都引入了额外的IO开销，造成性能降低。

SheetFile

我们根据上述的几点关键观察，提出了我们的文件系统所提供的文件抽象：SheetFile。顾名思义，这是一种专门为电子表格应用的文件操作需求所设计的文件抽象方式。相较于传统文件抽象，SheetFile放弃了文件连续性要求。它并不为应用提供一个线性连续的文件字节流抽象，而是提供更高层的表格抽象。在目前的各种电子表格应用中，单元格都是基本元素，因此，在SheetFile的抽象中，应用不再需要手动计算Offset，并基于Offset修改文件，取而代之的是基于单元格的API。亦即，应用将可以基于行号和列号，直接对一张抽象表格中的某个特定单元格进行读写操作。不过，单元格中的具体数据，对于文件系统而言仍然是不透明的，即应用可以在单元格中存储任何数据，文件系统并不限制某个单元格中的数据格式。行号和列号都是32位无符号整数。另外，由于几乎不可能存在4294967295x4294967295表格，因此应用需要使用一个特殊的单元格，即行列号分别为4294967295（即32位无符号最大值）的单元格来存储表格的元数据。

系统架构

系统架构图



系统整体架构如上图所示。与GFS类似，系统中同样存在MasterNode（存储文件系统元数据的控制节点，上图中以蓝色的M表示）以及DataNode（用于实际存储Chunk的数据节点，上图中以浅黄色的D表示）。出于容错的考虑，每种节点都不是单点的。多个MasterNode组成一个集群，该集群内，有一个节点为Primary节点，以实心方框表示，剩余节点为Secondary节点，以空心方框表示。Primary节点的地址记录在Zookeeper的特殊Znode中，客户端可以通过该Znode了解当前的Primary节点地址。它对外提供RPC服务，响应来自客户端的请求，而Secondary作为Primary的备份，不对外提供服务。当Primary节点崩溃时，从Secondary节点中选举产生新的Primary节点。

DataNode的容错方式与MasterNode类似，均采用Primary-Secondary备份机制，区别在于每个DataNode从属于一个组（DataNodeGroup），同组内的DataNodes互相进行备份。而每个组存储文件系统数据的一个子集，而并非像MasterNode那样只有一个集群存储全部的文件系统元数据。

我们使用Zookeeper进行选举，使用Kafka存储和传递日志。

顶层API

我们的文件系统对应用提供下列顶层API：

名称	语义
CreateSheet	创建一个表格文件
OpenSheet	打开一个表格文件
DeleteSheet	**标记删除**一个表格文件
ResumeSheet	**恢复**一个已被**标记删除**的表格文件
ReadSheet	读取一个表格文件的全部数据
ReadCell	读取一个单元格的全部数据
WriteCell	将数据写入一个单元格

MasterNode设计说明

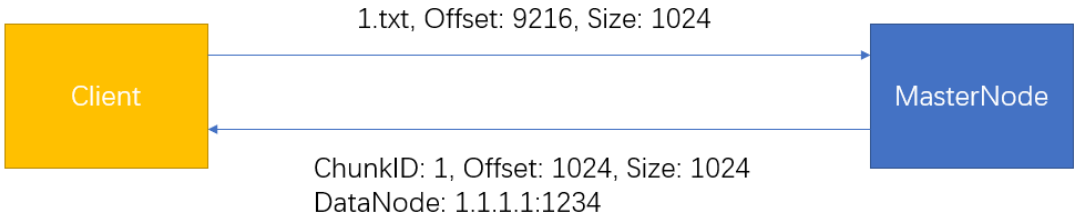
文件系统元数据

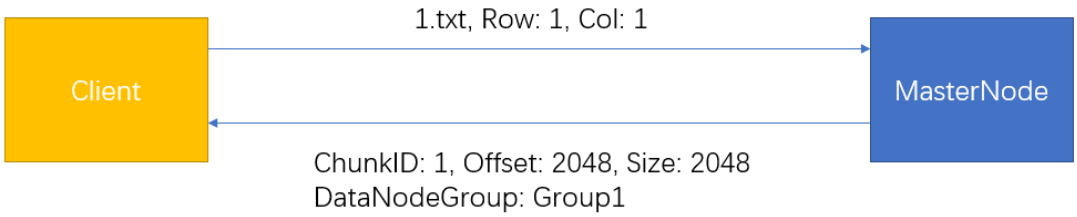
MasterNode作为控制节点，主要存储文件系统的元数据。文件系统客户端首先通过RPC与MasterNode进行交互，获取必要的控制信息，随后再与某个DataNodeGroup（中的当前Primary节点）进行数据面交互。我们的文件系统的元数据主要包括文件元数据和顶级目录元数据。

文件元数据

和GFS等分布式文件系统类似，我们的文件系统中，文件的数据也被划分为多个Chunk，分布在不同的数据节点上。具体而言，在我们的系统中，每个Chunk由一个DataNodeGroup负责保存。一个Chunk在我们的系统中大小固定为8KiB。既然文件由一系列Chunk组成，那么文件的元数据自然包括组成该文件的所有Chunk的信息。

在GFS等系统中，文件的元数据需要提供从文件Offset到具体Chunk以及Chunk内Offset的转换，这可以通过将Chunk元数据存储在类似B+树的查找树中来实现，其键为文件Offset。而在我们的设计中，上层应用并不通过Offset来访问文件，而是通过单元格的行列号。因此我们的文件元数据也需要包含类似前述B+树的索引结构，提供从单元格到该单元格所在Chunk的Chunk内Offset的映射。文件系统客户端通过查询MasterNode，得到自己需要读写的单元格所在的Chunk信息，然后和负责存储该Chunk的DataNodeGroup通信，基于Chunk内Offset获取该单元格数据，如下图所示，上方展示了GFS等文件系统的情况，而下方展示了我们的文件系统的情况。





然而，一个单元格的数据大小并不是固定的，而是取决于应用中用户在这个单元格实际输入的数据多少，相反地，Chunk的大小却是固定的，如果出现一个单元格的数据需要存储于多个Chunk中怎么办？对于这一问题，我们目前采取了一种简单的策略，即限制单个单元格的最大大小为2KiB，而每个Chunk中包含4个固定的slots，即预留4个大小各为2KiB的空间供4个单元格进行存储。这并不是一种较好的设计，除了会限制用户的使用方式外，更大的问题是它会降低存储空间的利用率。但出于时间原因，我们无法实现更复杂的设计，在本文档的下一步工作一节中，我们列出了一些其他的设计，可供读者参考。

顶级目录元数据

出于易用性的考虑，文件系统需要通过文件名来组织系统中现有的文件，对外提供基于文件名访问某个特定文件的API。在现有的文件系统实现中，文件名到某个文件的映射通常由目录结构来维护，我们的文件系统也采用了类似的设计。但由于时间关系，我们只采用了一个顶级目录，而不支持多级目录，所有文件均存储于顶级目录下。每个文件表示为顶级目录中的一个条目（Entry），条目中除了记录文件名到文件元数据集合的映射外，还有一个标记删除位，该位置为1时则无法通过文件名访问该文件。我们的删除操作就是通过修改对应条目中的该位实现的。

元数据存储

Zookeeper的问题[3]

某个作为Secondary的MasterNode若最终要能成为Primary，则必须保证在它成为Primary时，它所保存的元数据与先前的Primary节点基本相同（可能最后的少数更改有所丢失），而这与元数据的存储方式息息相关。一种自然的想法是，Zookeeper为它的客户端提供了一种分布式、强顺序、强一致性的控制数据存储方式，因此可以将文件系统的元数据存储于Zookeeper中。这样，Primary和Secondary节点都不需要做任何额外的处理，只需直接从Zookeeper读写数据即可。然而，此种设计有严重的问题。关键在于，Zookeeper并非一种用于存储大量数据的分布式数据库，而只是一个“Coordination Service”，因此它提供的保证是较强的操作顺序以及一致性。这种保证的代价有三：

- Zookeeper的写入性能较差，只适合读取远多于写入的场景，而这与大部分分布式系统中的关键控制信息的需求一致。然而，文件系统的元数据会随着文件写入操作频繁变更，并非读多写少的场景。
- Zookeeper的Znode中无法存储过多的数据，官方指出每个Znode的数据量应当在几KiB这一数量级
- Zookeeper虽然提供了树状数据模型，但并不支持类似平衡查找树的操作，无法支持文件系统元数据的查找需要

综上，我们并不使用Zookeeper存储元数据，仅使用它进行选举操作。

内存缓存

我们最终的设计是放弃一个由Master集群中所有MasterNode共享的全局元数据存储，改为每个MasterNode均维护一份存储于本地内存和文件系统中的元数据拷贝，各个节点之间的元数据一致性通过日志同步来保证（详见下文）。在正常情况下，MasterNode将所有元数据缓存于内存中。**对元数据的读写均由内存缓存直接响应**，仅定期通过Checkpoint机制将内存缓存持久化存储到本地文件系统中（详见下文）。在节点崩溃重启后，节点首先加载Checkpoint的数据，然后读取并应用Checkpoint之后的日志，以重建内存缓存，尽可能恢复崩溃前内存缓存中的数据（详见下文）。

持久化存储

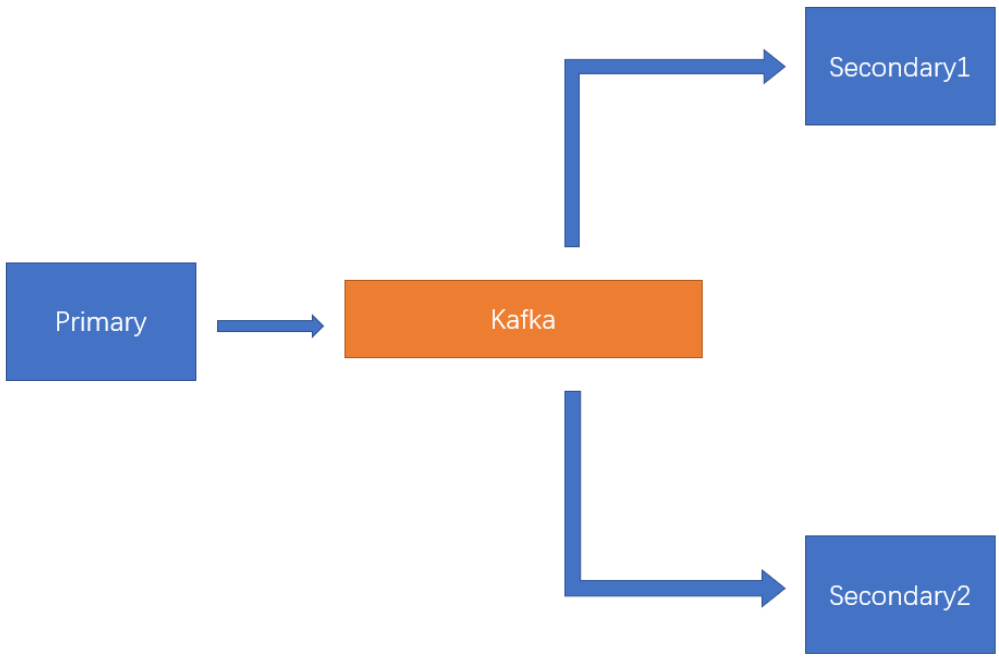
单元格索引等元数据需要通过磁盘B+树的方式进行索引。然而，一个高性能的磁盘B+树，通常也是一个单机存储引擎的核心部分。我们并没有时间从零开始编写一个高性能磁盘B+树，也很少有这方面的现成开源库。因此，在这里我们采取了一种变通的方式。Sqlite是一种以库的方式集成到其他应用中的SQL数据库实现，且其核心数据结构正是磁盘B+树，因此我们通过将sqlite集成到MasterNode中来使用其磁盘B+树存储功能。

具体而言，在每次Checkpoint过程中，我们将单元格索引、Chunk以及顶级目录条目数据持久化保存到本地sqlite数据库。其中，Chunk和目录条目使用一张sqlite数据表保存。对于单元格索引，每个表格文件有一张单独的索引数据表，与该文件有关的所有单元格索引数据均存储于该表中。从Sqlite实现的角度考虑，这相当于为每个表格文件建立了一棵单独的磁盘B+树。

容错

基于主从日志同步的容错

我们的容错保证基于Primary节点和Secondary节点间的日志同步来完成，如下图所示：



Primary对外响应RPC请求。在处理一个RPC请求时，若该请求需要涉及到修改元数据，则Primary首先将对元数据的修改形成日志，并推送（Push）到Kafka中，随后再对自己所拥有的元数据副本应用修改（内存缓存），并返回响应结果。Secondary节点不对外提供服务，而是不断从Kafka中拉取（Pull）日志，并将每条日志中记录的操作都应用到自己的元数据副本中。最终，当一个Secondary节点成为Primary时，只要它完整应用了所有的日志，则它将拥有和Primary崩溃前基本一致（除了尚未发送到Kafka的日志）的元数据副本。

选举

根据Zookeeper Recipes[4]，我们自行实现了其中描述的选举逻辑并进行了封装。该选举算法类似于同步原语中的Ticket Lock，主要基于选举请求抵达Zookeeper的顺序来决定当前的Primary节点。每个参与选举的节点都要向Zookeeper发出一个选举请求，第一个成功完成请求的节点即被选举为Primary。当Primary崩溃后，Zookeeper将自动唤醒该节点之后的第一个节点作为新的Primary节点。此外，每个集群的选举还使用一个特殊的确认Znode。当新的Primary已准备好对外界提供服务时，将自身的监听地址写入该Znode中。文件系统客户端可以通过不断读取该确认Znode来获取Primary节点的最新地址。

日志条目

对于读元数据请求，MasterNode并不需要记录日志。但对于写元数据的请求，MasterNode需要在对自己的元数据副本应用修改之前，先将对元数据的更改形成日志并推送到Kafka中。只有当Kafka成功写入日志并返回后，MasterNode才能在自己的副本上应用修改并返回。

目前需要修改元数据的操作主要包括下列操作：

- 创建文件
- 写入单元格
- 标记删除文件
- 恢复文件

在我们的实现中，创建文件同时涉及到目录条目、单元格索引以及Chunk三种元数据的修改（因为在创建文件时我们将会预先创建前述用于存储表格元数据的特殊单元格）。写入单元格涉及单元格索引以及Chunk两种元数据的修改，后两个操作只涉及到目录条目一种元数据的修改。对于同时修改多项元数据的操作，文件系统领域的常见做法（如jbd2）是提供一种类似事务的日志API，每条日志只记录一种元数据的修改，多种元数据的修改分为多条日志，这些日志共同组成一个事务。若该事务最终成功提交，则事务中的所有元数据修改有效，否则这些修改均无效。在我们的系统中，由于元数据的种类以及单次写入操作所涉及的元数据量都较小，我们采取在一条日志条目中同时记录三者的更改的设计，若某个更改操作不需要修改某种元数据，则相应选项留空即可。在实现上，我们使用protocol buffer定义了日志条目的二进制格式，如下所示：

```
message MasterEntry {
    oneof _Cell {
        Empty e1 = 1;
        CellEntry cell = 2;
    }
    oneof _Chunk {
        Empty e2 = 3;
        ChunkEntry chunk = 4;
    }
    oneof _FileMap {
        Empty e3 = 5;
        FileMapEntry map_entry = 6;
    }
}
```

若某一项元数据未被修改，可以使用一个特殊的空消息`Empty`替代。除了上述的普通日志外，我们的日志基础支持库还实现了一个特殊的预定义日志条目Checkpoint，以用于在主从节点之间同步进行Checkpoint操作，将内存中的缓存数据刷入磁盘。此外，数据库中还会保存当前的Checkpoint信息，主要是下次重启时从Kafka中的哪一位置开始消费日志。

剩下需要考虑的一种情况是，如果Primary节点已经将Checkpoint日志写入Kafka，但在自身尚未完全将内存缓存写入磁盘时便崩溃；或一个Secondary收到Checkpoint日志后，开始进行Checkpoint操作，在操作未完成前便崩溃，是否会引入额外的问题？在我们的设计中不存在这样的情况，这是因为我们的日志条目记录的是操作的最终结果，而非操作过程，因此应用日志条目是幂等的，同一条目可以被多次应用，只要保证按顺序应用日志即可。上述两种情况等价于Checkpoint操作没有成功，下次启动时它们还是从上一个Checkpoint开始重新应用日志条目，并不会带来新的问题。

Kafka日志存储

我们选用Kafka来在Primary和Secondary节点之间传输日志，是因为Kafka有如下几大优势：

- 支持Pub/Sub模型，即Primary发布（Publish）日志，Secondary订阅（Subscribe）日志
- 支持持久化存储消息
- 由于消息被持久化，在Kafka中消息被消费一次后并不会被移除，从而消息可以被反复多次消费，并且每个消费者可以拥有不一样的消费进度。

以上的几项特征使得Kafka非常适合用于进行commit log的同步（但也存在一些问题，详见“下一步工作”一节）。在实现上，每个集群（Master集群，以及每个DataNodeGroup）均使用一个独立的Kafka Topic，每个Topic仅包含一个partition，从而保证该Topic中的消息一定是完全顺序的。每个Primary节点调用Kafka Producer API，不断将新的日志条目写入该topic中，而每个Secondary各自实例化一个Kafka Consumer，从该topic中不断拉取日志。由于不同Secondary节点的Consumer并不共享，因此它们具有独立的消费进度，互不干扰。

选举与日志

本节结合选举和日志两种功能描述Primary和Secondary两种角色的日志操作，以及Secondary转换为Primary时的流程。

每个节点启动时，都试图将自己作为Secondary节点运行。首先读取自身的sqlite数据库中的Checkpoint记录，将自己的Kafka Consumer的位置设为Checkpoint中保存的位置。对于一个新的节点，数据库不存在，则它将从Kafka的起始位置消费日志。之后，节点从数据库中加载全部文件系统元数据到内存缓存中，并恢复内存数据结构。此后，节点参与选举，有如下两种可能：

- 该节点被选举为Primary，则它开始立即消费Checkpoint之后的所有日志，直到Kafka中不存在剩余的日志后，将自己的地址写入前述的确认Znode，开始对外提供服务
- 该节点确认自己为Secondary，则它开始反复从Kafka中拉取日志并应用（但并不要求该节点的消费进度始终紧跟Primary的生产进度）。

若当前Primary节点崩溃，则将会有某一个Secondary节点被选举为Primary。但后者**并不能立即确认选举结果**，因为Secondary的日志消费逻辑并不保证该节点在被选举为Primary时已经应用了所有已有的日志，从而获得一个与崩溃前的Primary基本一致的内存元数据副本。因此，在确认选举结果前，新的Primary节点也需要立即进行快速日志消费，直到不存在剩余的未消费日志，才能确认自己的选举结果。

Datanode设计说明

RPC

Datanode主要负责Chunk级别的读写操作，提供的RPC有ReadChunk，WriteChunk和DeleteChunk三种。

文件系统的Client会调用ReadChunk，WriteChunk来进行Chunk级别的读写，而在文件彻底删除不再需要从回收站恢复之后，Master会调用DeleteChunk来永久删除该Chunk文件。

持久化格式

Datanode以chunk文件为粒度在磁盘上进行持久化，存储文件命名为`chunk_{#chunk_id}`,内部存储的数据为:

```
#data_1_offset: #data1
...
#data_n_offse:  #datan
#MAX_DATA_OFFSET: version_{#version_id}
```

我们认为现代操作系统的Page cache已经可以很好的对数据进行缓存操作，手动维护内存的缓存在我们对文件的存储方式并没有对比操作系统更好的认识的时候是没有必要的，所以在这里所有的文件读写操作我们都直接利用操作系统的文件接口。

多备份与容错

Page Cache并不能保证文件落盘的问题我们的项目使用记Log的方式来进行保证。
Datanode需要记录Write和Delete时候的修改，即在client调用WriteChunk和DeleteChunk RPC时记Log。不同Datanode组对应的Log列表会记录在不同的Kafka Topic组来加以区分。具体内容在

日志设计

日志条目

WriteChunk RPC需要记录的东西包括:

- `datanode id` -- datanode组的ID（ZK中的ID，所有对应的主从datanode都需要读取这一log记录到自己的持久化磁盘中）
- `version` -- 当前文件的version号
- `chunk id`, `offset`, `size` -- 实际需要写入的data的chunk的id、写入起始offset、数据size
- `data(zip form)` -- 数据的zip压缩格式
- `checksum of data` -- data的checksum

Log的具体结构为:

```
|<- write ->|<- version ->|<- chunkid ->|<-  off  ->|<-  size ->|<-  cks  ->|<- data ->|
|<- 64bit ->|<-  64bit  ->|<-  64bit  ->|<- 64bit ->|<- 64bit ->|<- 32bit ->|
```

Delete Chunk RPC需要记录的东西包括:

- `chunk id`: 需要删除的chunk id

Log的具体结构为:

```
|<- write ->|<- version ->|<- chunkid ->|<-  off  ->|<-  size ->|<-  cks  ->|<- data ->|
|<- 64bit ->|<-  64bit  ->|<-  64bit  ->|<- 64bit ->|<- 64bit ->|<- 32bit ->|
```

这里的datanode id会作为多个topic组来加以区分。

会在log中记录data的压缩格式数据来节省在kafka中存储的内容的大小，同时还会记录checksum来在持久化的时候和本地磁盘中的数据进行比较，从而快速比较得出该log是否已经做完的判断，避免解压缩

Datanode的管理采用主从结构，使用ZooKeeper的节点来记录Primary Datanode的地址，同时通过ZooKeeper创建节点的机制可以进行Leader的选举，新Primary Datanode的地址就会记录在该记录节点中。Client在发现某些RPC失效后只需要去重新询问这一记录节点，再次尝试连接，直到最后获得了正确的Datanode的地址即可。

Primary Datanode会将日志记录到Kafka中，Secondary Datanode会去Log中阅读数据来和Primary Datanode进行同步

与GFS的比较

1. 我们这里Chunk的粒度较小。因此如果在master维护所有的chunk地址数据会有较大的开销，同时Chunk向Server发送数据也涉及到很多数据的网络传输。所以我们这里将datanode改为了机器为粒度的备份，牺牲了一定的系统伸缩性和资源均匀利用率，但考虑到我们当前的应用场景有大量的写入操作，需要很多元数据的更新，现在的设计极大的减少了这些开销
2. GFS采用了同步数据写入，即Primary Datanode需要等到所有Secondary Datanodes的回复才返回客户端。而我们的文件系统采用异步的数据更新，虽然不能做到即时从任意副本读到最新的数据，但考虑到我们系统只有在1.第一次打开文件进行全局的读取时 2.写入或修改某一单元格，需要将新的数据转发到其他前端时，才会涉及到读操作，所以异步的写入是合理的，可以整体的提高读写性能。

下一步工作

由于项目时间所限，在本项目中，许多地方我们并没有采取最完美的设计。事实上，很多设计问题我们在前期研究时均已考虑并进行了一定的设计，但因为时间和设计复杂度原因，我们并未能将它们实现。我们在此简要描述这些设计，以供读者参考。

Append-only修改

如果将文件的定义规定为“线性连续的字节流”，那么严格意义上来说，我们所实现的系统并不能称之为文件系统。如前所述，传统文件系统所提供的抽象从根本上不适合现有应用的需求，对此，一种解决方法是放弃使用文件系统的一致性来保证上层应用的一致性，使用数据库等替代方案；另一种方法就是提出新的文件系统抽象，这也是我们的动力所在。然而，如果底层的文件系统需要固守传统的文件抽象，那么应当如何设计文件的修改操作，使得文件修改的开销尽量降低呢？这并非是不可能的。

现有文件格式的修改操作需求的根本问题在于，它们要求文件系统能提供就地的修改支持，而这使用传统的POSIX操作并不能高效地完成（Linux提供了非标准系统调用fallocate可以处理这一问题）。那么，上层应用若要适应底层文件系统，最根本的解决办法是定义新的修改操作方法，使得就地修改变为append-only的修改，后者在几乎所有的文件系统中都能得到极为高效的支持。

在这一方面，应用可以借鉴许多数据库系统的设计，将文件数据分为索引和值两部分。索引部分只保留一个到值部分的指针，这个指针可以做成固定大小的（例如一个8bytes的整数），从而修改某一个指针的指向不会带来任何问题，因为修改前后，被修改部分的数据大小不变，不涉及插入操作。值部分使用一个Value Log存储值，即该部分采用连续的日志形式进行组织，添加一个新的值，只涉及到在Value Log末尾进行append，而不需要修改先前的日志条目。修改一个值的做法是在Value Log末尾添加一个新的值，并修改原有的指针指向新值的日志位置即可。这一设计的主要问题是随着文件的不断写入，Value Log的碎片化会变得较为严重，需要额外实现对Value Log的清理工作，另外对于原有的文件格式可能需要较大的修改，因而我们最终没有实现。

GC与更好的分配策略

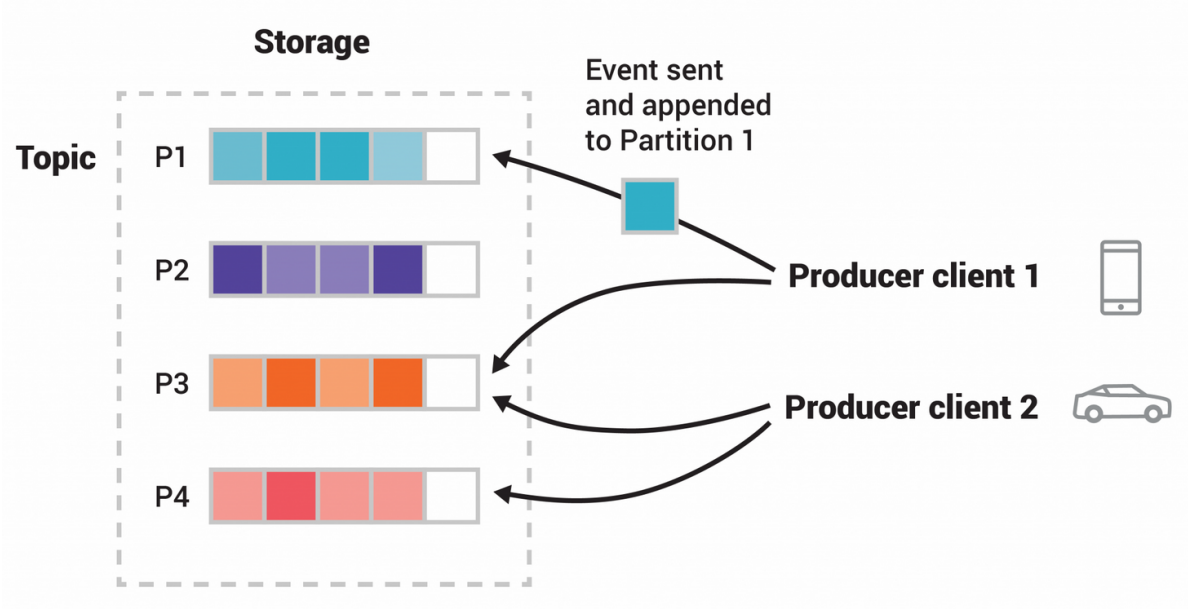
目前的文件系统设计中，对于文件的删除我们采用的是简单的标记删除策略。理论上，当一个文件被标记删除，且在一段时间内仍未被恢复时，该文件所使用的Chunk应当被真正地从系统中删除，以释放所使用的存储空间。又如，当一个Chunk所存储的4个单元格的数据都被用户删除时，该Chunk也应当被彻底删除。上面两种情况都属于垃圾回收（GC）的范畴。出于节省空间使用的考虑，文件系统中应当运行定时监控任务，监控文件系统中的垃圾并进行回收。但由于GC的实现可能需要阻塞文件系统中的所有节点，且还需要额外的日志处理，我们暂未实现。

此外，每当创建一个先前不存在的单元格时，我们需要为这个新的单元格分配一个Chunk。显然，我们应当优先利用还有空余“插槽”的Chunk，而非创建新的Chunk。在目前的实现中，我们只是简单地记住上一个还有空余的Chunk，然而这一策略的利用率并不理想。因为若一个Chunk先被写入了4个单元格，随后它所包含的单元格又被删除，就会重新将其变为有空余的Chunk。目前的实现中，我们无法感知到这一点，从而不会利用这样的Chunk。更合理的设计是增加新的DeleteCell API，并将有空余的Chunk以Free list的形式进行组织，而非只记住一个空余的Chunk。

每当创建了一个新的Chunk，我们需要将其分配到某一个DataNode组中进行存储。目前MasterNode在分配时，只使用简单的Round Robin策略，而并不考虑DataNodeGroup的节点数量、容量、负载等信息。更合理的分配策略应当进一步考虑这些信息。

替换Kafka

如前所述，Kafka提供的几个关键特性非常适合在Primary和Secondary之间传递日志。但Kafka也有一些我们所不需要的额外设计，导致降低了日志传递的效率。根本问题在于Kafka的Partition策略。Kafka的Partition策略如下图所示：



Kafka的数据模型中，每一条消息都由三个部分组成：

- Key，该部分并非用于唯一性的保证，可能有大量不同的消息使用相同的Key
- Value
- 时间戳

Kafka的Key主要用于进行partition，而非保证消息的唯一性。如上图所示，每个Topic会被划分为多个partition，划分的依据正是Key的值，若两条消息拥有不同的Key，则它们可能进入不同的partition中。而**Kafka只保证同一个partition内的消息对于所有消费者的顺序一致**，如果一个节点需要同时从多个partition消费消息，则它们的顺序没有任何保证。事实上，Kafka Consumer在创建时，必须指定一个它所需要消费的

partition，不存在同时消费所有partition的选项。然而，在我们的设计中，我们正是需要一个全局有序的，但又可以被反复独立消费的类队列模型，因此我们只能使用一个partition，以确保所有消息都是有序的。这种设计虽然满足了应用的需求，但却严重降低了性能，因为Kafka高性能的一大保证，正是消息的partition。

一个专门的commit log服务不应当使用Key作为partition的依据，而应当使用时间或者消息位置进行partition，即每个partition保存一个连续的子区间。目前的各种时序数据库（如influxdb）就是采用这样的partition策略，但它们是基于时间进行划分，而非基于消息位置，而后者才是我们所需要的。如果时间充裕，我们可能需要自行开发一个日志传输服务。

读写分离

在我们的设计中，Secondary节点只是作为单纯的备份存在，增加Secondary节点可以提升系统的可靠性，但不能提升系统的性能，系统的性能始终受制于Primary单点。事实上，在许多采用Primary-Secondary容错机制的分布式系统中，更常见的策略是Primary与Secondary进行读写分离，即一个Primary节点响应写请求，其余所有Secondary节点均可以响应读请求。这种设计的主要复杂性在于，在我们原来的设计中，Secondary不需要服务外界，因此它的消费进度可以落后于生产进度，而且不同Secondary节点的消费进度可以不一致。但若采用读写分离的设计，则需要保证不同Secondary节点的一致性，这就要求我们不能使用消息中间件进行消息传递，而需要使用类似2PC的方式，增加了实现的复杂性。

快照

由于Kafka为我们的系统持久化存储日志，因此理论上而言，我们的系统可以随时支持新的节点的加入，只需要从头开始依次应用所有的日志即可。然而显然地，当系统已经运行了一段时间，再从头开始重放日志是非常低效的。更好的办法是使用一个快照服务器。Primary或某个Secondary节点可以将自己完成Checkpoint后的数据库或数据文件目录存储到快照服务器中，而新加入的节点可以直接从快照服务器取得一份快照，从而直接获取某一个相对较新的Checkpoint版本，避免从头开始应用日志。

系统级OT

目前，在合作编辑应用中，用于解决多人同时编辑可能产生的冲突的主流方案是OT（Operational Transformation）算法。它的思想是不通过锁等机制阻碍冲突操作的产生，而是允许互相冲突的操作，并通过一系列预先定义的语义，将其合并为正确的最终结果。显然，由于不需要开销巨大的锁机制，OT算法能提供更好的性能和用户体验。我们对OT算法进行了一些研究，并且注意到，Luckysheet的JSON文件格式与Spreadsheet ML的XML文件格式本质上都是一种树状结构。在研究过程中，我们找到了Claudia等的一篇工作：*Tree-based model algorithm for maintaining consistency in real-time collaborative editing systems*[5]，他们提出了一种将树状文件结构与成熟的线性结构OT算法相结合的方法，如下图所示：

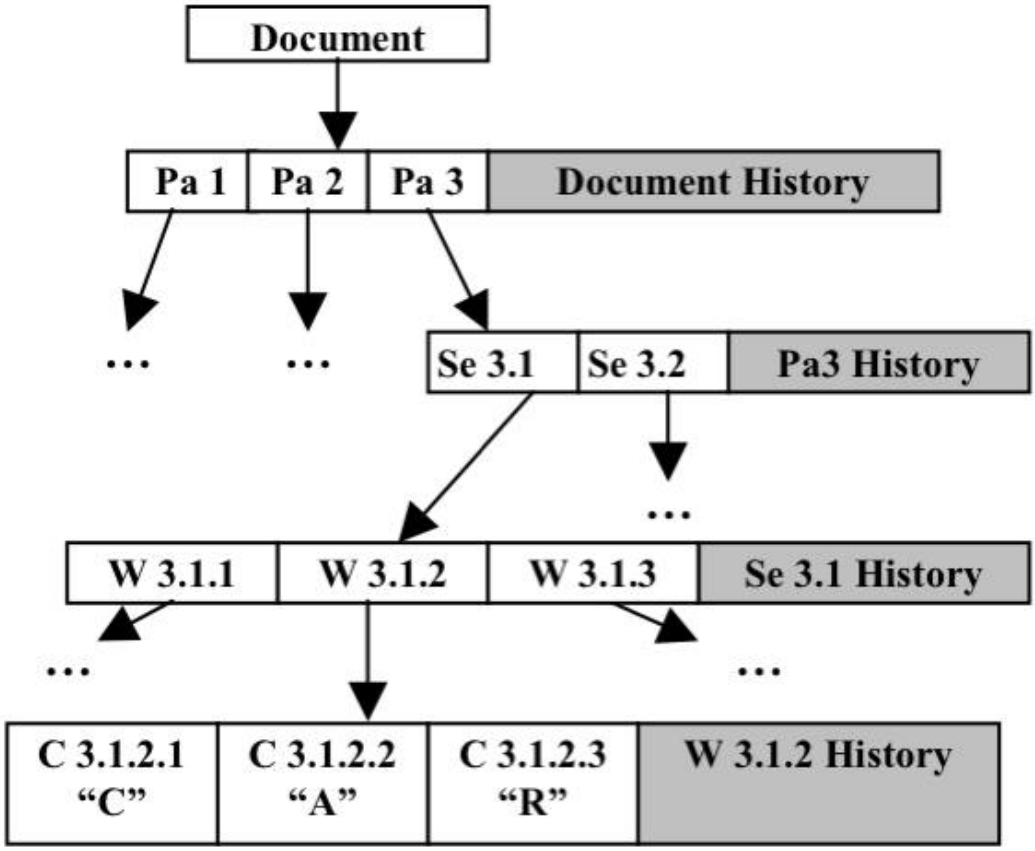


Fig. 2. Example of structure of a document

我们注意到，树状结构的每一层仍然是一个线性结构。因此，我们在每一层内部仍然可以应用一种传统的基于线性结构的OT算法（例如GOT），而在层与层之间，则需要使用他们在本文中提出的新的算法。

受该工作的启发，我们认为，从系统的角度，可以考虑为文件提供一个树结构的抽象，而非一个单层线性抽象，允许应用根据实际需求定义树的具体结构，以及在每一层中所使用的OT算法。而我们的系统则负责在该树结构中层次化地进行OT运算。但由于OT算法的语义较为复杂，且OT的语义高度依赖于具体应用，并没有较为统一的开源实现，我们出于时间问题未能实现。

---

1. https://mengshukeji.github.io/LuckysheetDocs/zh/guide/sheet.html#%E5%88%9D%E5%A7%8B%E5%8C%96%E9%85%8D%E7%BD%AE ↩

2. http://officeopenxml.com/SScontentOverview.php ↩

3. https://zookeeper.apache.org/doc/r3.7.0/zookeeperOver.html ↩

4. https://zookeeper.apache.org/doc/current/recipes.html#sc_leaderElection ↩

5. Ignat, Claudia, and Moira C. Norrie. "Tree-based model algorithm for maintaining consistency in real-time collaborative editing systems." *Fourth International Workshop on Collaborative Editing, CSCW 2002, IEEE Distributed Systems online*. 2002. ↩