# 第四章 存储机制

我们知道，DBMS 的作用就是管理各种数据库，提供对数据的定义，存储，控制功能。在关系型数据库中，存储的主要数据对象就是表（关系）。而多年来，CPU 和内存进步与传统存储器进步的差距被成倍拉大：摩尔定律指出，CPU、内存和网络按照摩尔定律每两年可以达到一次性能的翻倍，而传统的存储器并非如此。15K RPM 硬盘在十年前面市，而目前还没有看到 30K 甚至 20K RPM 的硬盘。传统的机械硬盘成为制约计算机发展的瓶颈，这一点在数据库领域也是如此。在装载数据库文件的时候计算机需要消耗 IO 资源，尤其在数据量更大时消耗更大，在将数据写入硬盘时也会发生同样的事情，这将带来很大的延迟。在解决这个问题上可以有两种思路，一种是采用固态硬盘（SSD）代替机械硬盘或者使用内存数据库。固态硬盘的优势在于没有机械装置，不会产生旋转和寻道延迟；而内存数据库将数据放在内存中直接操作数据。这种技术将传统数据库的读写瓶颈移除，也能极大提高数据库的性能；加快查询响应的另一种思路是针对不同的应用场景，选择不同的数据组织方式来提高数据读写能力，如列存储[1]等。列存储顾名思义，与传统的行存储相对，在存储介质中是按列连续存储而非按行连续存储，相同的字段被组织在一起，这有利于减少 IO 以及对数据的高效压缩。

关于行列存储的不同，研究[3]做了一系列的比较,就列存储而言，数据库系统只需要对于一个给定的查询只从磁盘中读出有限的

几列，不相关的属性不用读入内存。在数据仓库的环境中，一个典型的查询涉及到大量数据的聚合操作，列存储可以提供显而易见的优势。列存储可以消耗 CPU 周期达到节省磁盘带宽的目的。使用面对列的压缩算法，将数据以压缩的形式保存可以极大地提高数据库的性能。压缩算法在列存储中可以达到极高的压缩率，这是因为列存储有着更低的熵值。例如，存储着用户手机号码的字段值之间有着更高的相似度。如果这些字段值是有序的，那么压缩的效率将会更高。

除了压缩的技术之外，列存储中最常用的另外两种技术包括延迟物化[4]和块迭代。在大多数查询中需要涉及超过一个以上的字段，因此数据库需要将两个字段进行类似于 join 的操作。延迟物化的思想在于尽可以地推迟物化的时间，这样的好处在于避免在物化中涉及到很多不必要的列，保持列存数据库的优势。块迭代即每次将相同列的属性值按照块进行传送，在列存储中，这样做同样可以避免不必要的字段传送。另外，由于每个 block 都只有一列，只需要一次函数操作就可以得到需要的值（在行存储中可能需要 1-2 次函数操作)。

研究[3]同时指出，列存储也不是总是可以达到很高的性能，在 OLTP 的环境中列存储的效率较低，所有目前的数据库有些采用行列混合存储的模式，Greenplum 就是这样的数据库。其他的列式数据库包括 Sybase IQ，infobright、infiniDB、GBase 8a，ParAccel，Sand/DNA Analytics 和 Vertica。

## 4.1 数据组织方式

数据管理系统功能的本质是数据的存储和查询，换言之，数据是 DBMS 系统操作的对象。关系数据模型是上个世纪 80 年代是研究的重点，也在之后的很长时间成为主流的数据模型。但是在同时，关系模型的局限也渐渐显露出来，特别是随着 web2.0 的发展，关系模型暴漏出表达不够灵活，扩展性差，数据读写速度慢的缺点。因此，键值模型、文档模型和其他的模型逐渐被提出以弥补关系模型的不足。

按照不同的关系模型，科学界和工业界实现了不同的系统。在本节中，我们比较了数款常用的数据库系统，这些数据库系统包括：

- 分布式关系数据库 Greenplum 和 BigTable
- 分布式文档存储系统 MongoDB
- 分布式图形数据库 Neo4j
- 分布式文件存储系统 HDFS

在接下来的部分，我们将详细说明这些系统是如何在逻辑和物理上组织数据。

### 4.1.1 Greenplum

Greenplum[5]是 Pivotal 公司推出的分布式关系数据库，采用 MPP 架构，主要面对 OLAP 业务。在功能上，Greenplum 支持传统的数据库技术，如索引、日志、锁等。除此之外，Greenplum 还支持分区表、地理信息索引、json 格式数据等。

传统的关系型数据库主要采取行存储模式，这对于 OLTP 型事务是便捷的。行存储逻辑上如下表所示，表的每一行代表一个元组，按照关系型数据库的这种天然属性把数据按行来组织在一起。行存储模式下同一个元组的不同属性在存储上是连续的。如果要存储一个元组的某个属性值，首先读取元组的所有数据，然后进行投影操作，从整个元组中分离出需要的属性值。

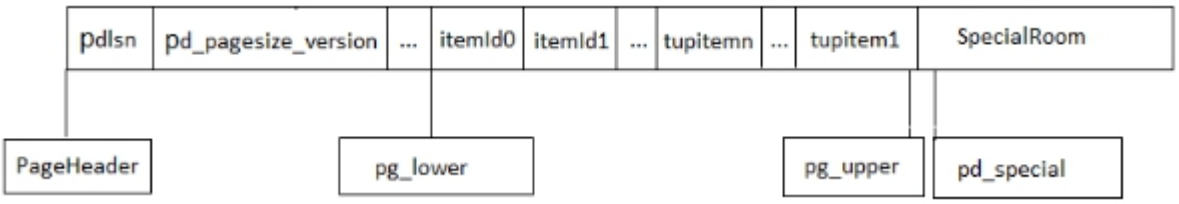
| Name | Album | Type | Singer | Comments |
|---|---|---|---|---|
| 后来的我们 | 五月天 | 流行 | 五月天 | 72842 |
| Lighters | Marshall Carter | Rope | Bruno Mars Royce Da | 25427 |
| 十年 | 黑白灰 | 伤感 | 陈奕迅 | 56822 |
| 缘分 | 最红 | 怀旧/影视原声 | 张国荣/梅艳芳 | 410 |

除了行存储，Green plum 还支持列存储模式，使用于 OLAP 场景。列存储模式下数据按照属性列来进行组织，如下表所示。表的不同元组的同种属性在存储上是连续的，不同的列独立存储。在 Greenplum 中，每一个属性列在操作系统中都是以单个文件的形式存放，根据实际需要读取对应的属性列。有关行列存储模式的进一步对比可以阅读 4.2 节的内容。目前的列式数据库包括：Sybase IQ，infobright、infiniDB、GBase 8a，ParAccel，Sand/DNA Analytics 和 Vertica 等。

| Name | 后来的我们 | Lighters | 十年 | 缘分 |
|---|---|---|---|---|
| Album | 五月天 | Marshall Carter | 黑白灰 | 最红 |
| Type | 流行 | Rope | 伤感 | 怀旧/影视原声 |

| Singer | 五月天 | Bruno Mars<br>Royce Da | 陈奕迅 | 张国荣/<br>梅艳芳 |
|---|---|---|---|---|
| Comments | 72842 | 25427 | 56822 | 410 |

Greenplum 的物理存储模式如下图所示：一个文件页空间被逻辑地分割成三部分：页描述区（Page Header），元组数据空间（Tuple Item Space），以及特殊空间（Special Space）。页描述区记载了页的使用情况，包括元组数据空间和特殊空间的起始位置等。元组空间是实际记录元组数据的地方。每个被记录的元组称为一项，每项由描述 id 和元组数据构成，项描述 id 描述了元组存储位置、大小以及一些状态标志。项描述 id 和项数据分别在元组数据空间的两头往中间靠拢存放。




### 4.1.2 BigTable

BigTable[6]是谷歌开发的分布式存储系统，可以用来管理半结构化数据。具有较强的扩展性，可以存储 PB 级的数据。谷歌网页的索引、Google Earth 以及 Google 金融的数据都使用到了 Big Table 技术。

从功能上看，BigTable 很像一个数据库，实现了很多数据库的策略，例如增删查改等。但 BigTable 提供了另一种组织数据的思路：

- 数据通过行和列名进行索引，这些名字可以是任意的字符串；
- 所有数据的类型都是无解释的字符串；

- 支持从内存或者硬盘中读取数据；

| Row Key | Time Stamp | Column Comments | Column Lable | | Column Singer |
|---|---|---|---|---|---|
| | | | 怀旧 | 影视原声 | |
| 缘份 | T5 | 410 | true | true | 张国荣 |
| | T4 | 360 | true | true | 张国荣 |
| | T3 | 310 | true | true | 张国荣 |
| | T2 | 240 | true | true | 张国荣 |
| | T1 | 210 | true | true | 张国荣 |

BigTable 的具体数据组织方式如上图。

BigTable 采用行键（row key）、列键（column key）和时间戳（timestamp）对图进行索引：

- 一个表中的行键，是任意的字符串。对每个行键下所包含的数据的读或写都是一个原子操作，（类似于行级别锁），以维护 BigTable 内数据的一致性。由于 BigTable 的存储是分布式的，行键还是数据划分的依据。
- 列键，也可以称为列簇，是一组相关属性的集合，通常都属于同一个数据类型。列键采用下面的语法命名：family:qualifier。例如上表中的 Lable：怀旧，Lable:影视原声都属于同一个列簇，他们都表示了“缘分”这首歌曲的标签属性。在 BigTable 中，访问控制以及磁盘和内存审计是在列家族层面上进行的。
- 时间戳用来区分相同数据的不同版本，是一个 64 位的整型数。时间戳必须是唯一的，以避免数据的冲突。上表的 T1-T5 代表 5 个不同的时间戳“缘分”这首歌曲的信息，可以看出它们在

不同的时间戳的评论数（Comments）列存在不同。在物理存储上，相同数据的不同版本根据时间戳进行降序存储。这样的好处是最新的版会被最先读取。

### 4.1.3 MongoDB

MongoDB[7]是一个基于文档的数据库系统，有着灵活的存储模式。这种模式在 MongoDB 中被称为集合（Collections）。每一个集合都是一个数据库，类似于 RDBMS 中的表的概念；每个集合都可以有多个文档(Documents)，类似于 RDBMS 的一行。然而不同于 RDBMS 的是集合的模式并非预先定义好或者说是一种松散的模式。

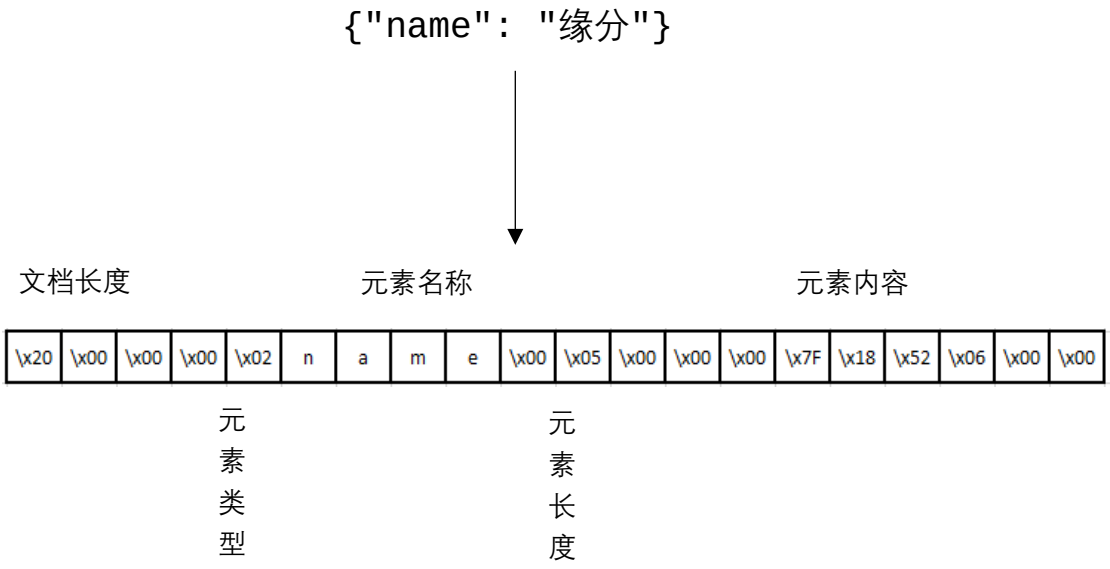
MongoDB 使用二进制 json（Binary JSON，BSON）存储数据。BSON 格式的数据(文档)存储有尺寸限制，最大为 16M。因此对于较大的数据，MongoDB 使用 GridFS 来辅助管理。GridFS 使用两个集合（collection）存储文件：files 集合用于存储文件的元数据；chunks 集合用于存储文件内容的二进制数据。

- **BSON 格式**

BSON 这种格式是专门为 MongoDB 而开发的，类似 json 的一种二进制格式。这种格式不一定比 json 存储的文件小，其优点是解释快。另外 BSON 增加了一些数据类型，如 Date.

以歌曲的名称信息为例，一个具体的 BSON 示例如下：前面的四个字节表示文档的长度，采用小端码表示，表示文档的长度是 20 个

字节。第 5 个字节（\x02）代表元素的类型，表示 string 类型。接下来的字节分表表示元素的名字、元素类型的长度以及元素的内容。最后一个字节（\x00）表示文档的结尾。

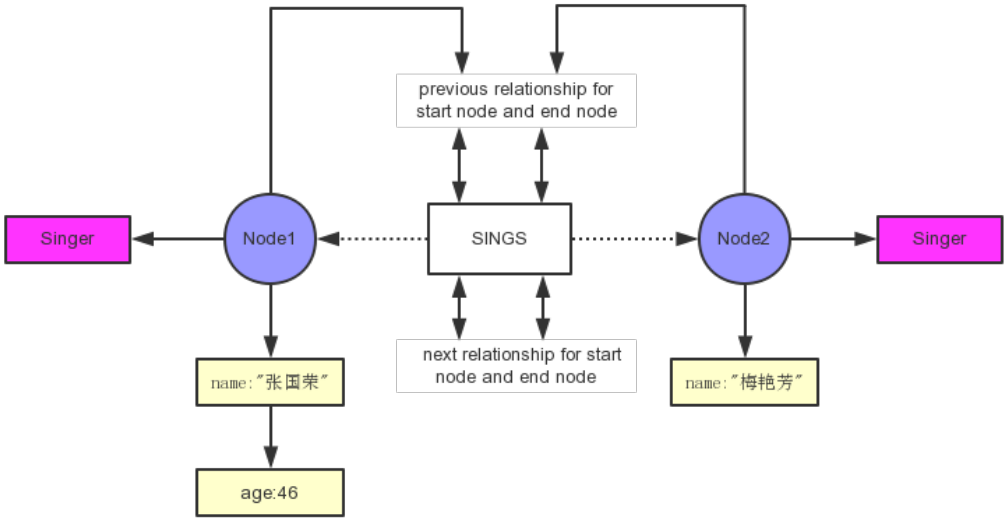



MongoDB 内部有预分配空间的机制，每个预分配的文件都用 0 进行填充，由于有了这个机制,MongoDB 始终保持额外的空间和空余的数据文件。

### 4.1.4 Neo4j

Neo4j[8]是一个高性能的 NOSQL 图形数据库，它将结构化数据存储在网络上而不是表中。下图描绘了 Neo4j 的存储模型。图中的顶点（Node)包含自身所拥有的属性结构（Property）和关系结构（Relationship ），这些结构都以链表的形式存储，Node 分别保存着这些链表的头部指针。属性链表是一条单链表，为了读取节点的属性，

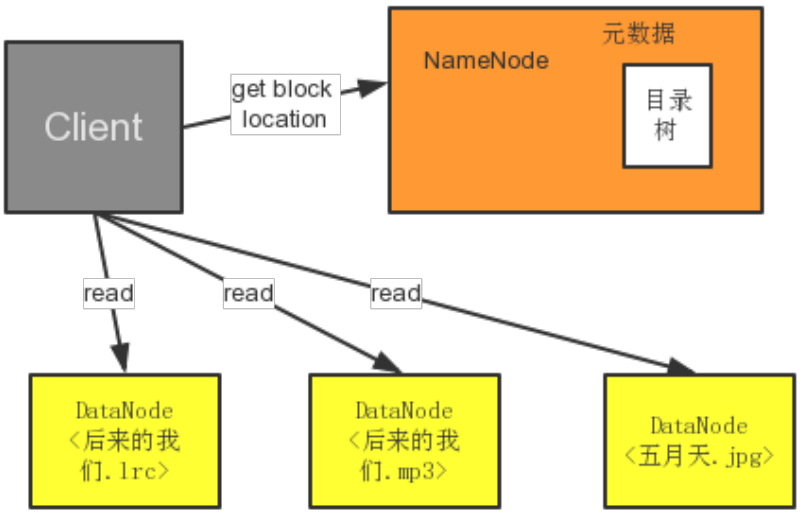
我们从指向属性链表的头部指针出发依次向后读取；而关系链表是一条双向循环链表，同样遍历关系链表也能找到顶点所有的关系。




在 Neo4j 中,主要有 4 类节点，属性，关系等文件是以数组作为核心存储结构，每条记录都是定长的；同时对节点，属性，关系等类型的每个数据项都会分配一个唯一的 ID，在存储时以该 ID 为数组的下标。这样，在访问时通过其 ID 作为下标，实现快速定位。所以在图遍历等操作时，可以实现 free-index。这些类的具体结构可以参看下图。inUse 是标志位，标志该条记录是否正在使用（是否可以进行回收）。nextRelID 和 nextPropId 对应着前面提到的链表结构。Label 是节点的标签信息； Extra 是标志位。例如用来识别密集连接顶点（densely connected nodes）。同样的，在关系结构中，同样有 inUse 的标志位。跟随其后的是和关系有关联的两个定点的 ID。其次还保存着关系类型的指针、两个端点在关系链表中的上一个/下一个关系的指针，以及当前关系是否位于关系链（relation chain）头部的一个标志位。

### 4.1.5 HDFS

HDFS[9]为 Mater/Slave 架构，一个 HDFS 集群有一个元数据节点 Namenode 和若干数据节点 Datanode 组成。Namenode 为 HDFS 主节点，负责管理文件系统的命名空间以及客户端对分布式文件系统的访问。Datanode 为从节点，负责管理该节点上的数据存储。




在 HDFS 上的文件实际上被分成了一个或多个连续的数据块（数据块大小通常为 64MB），为确保系统的可靠性，每个数据块通常有若干个副本（副本数通常为 3）分别存储在集群中的一个 Datanode 上。Namenode 负责保存和管理 HDFS 的元数据，执行文件系统的命名空间操作，如打开、关闭或重命名一个文件或目录，确定数据块到具体 Datanode 的映射。Datanode 负责处理文件系统对该节点上的数据的读写请求，在 Namenode 的统一调度下进行数据的创建、删除、更新和复制。

## 4.2 Greenplum 中的数据存储

### 4.2.1 行列存储方案

- **行列表的创建**

Greenplum 除了支持传统数据库中的行存储外，还支持列存储。默的存储方式是行存储，使用普通的 sql 语句创建的表就是行表。

想要创建列存储，首先必须保证该表也要是 appendonly 表。创建一个列存储的表只需要在创建表的时候，在 with 子句中加入 appendonly=true，orientation=column 即可。格式如下：

```
CREATE TABLE name
(…)
WITH (appendonly=true,orientation=column)
DISTRIBUTED BY(o_orderkey);
```

- **行列存储方案的选择**

Greenplum 同时支持行存储和列存储两种方案，但各有其优劣。一般来说，行存储的写入是一次完成。如果这种写入建立在操作系统的文件系统上，可以保证写入过程的成功或者失败，数据的完整性因此可以确定。列存储由于需要把一行记录拆分成单列保存，写入次数明显比行存储多，再加上磁头需要在盘片上移动和定位花费的时间，

实际时间消耗会更大。所以，行存储在写入上占有很大的优势。

还有数据修改,这实际也是一次写入过程。不同的是，数据修改是对磁盘上的记录做删除标记。行存储是在指定位置写入一次，列存储是将磁盘定位到多个列上分别写入，这个过程仍是行存储的列数倍。所以，数据修改也是以行存储占优。

数据读取时，行存储通常将一行数据完全读出，如果只需要其中几列数据的情况，就会存在冗余列，出于缩短处理时间的考量，消除冗余列的过程通常是在内存中进行的。列存储每次读取的数据是集合的一段或者全部，如果读取多列时，就需要移动磁头，再次定位到下一列的位置继续读取。

其次是对于两种存储的数据分布。由于列存储的每一列数据类型是同质的，不存在二义性问题。比如说某列数据类型为整型(int)，那么它的数据集合一定是整型数据。这种情况使数据解析变得十分容易。相比之下，行存储则要复杂得多，因为在一行记录中保存了多种类型的数据，数据解析需要在多种数据类型之间频繁转换，这个操作很消耗 CPU，增加了解析的时间。所以，列存储的解析过程更有利于分析大数据。

另外，记录的字段可分为查询常用字段和非常用字段，如果将非常用字段放在廉价设备上，可降低存储成本。但现有的行存储技术特点决定了常用字段和不常用字段在磁盘上是连续存储的，占用了同样的存储资源。这不利于资源设备的合理分配。列式数据库的出现在一定程度上弥补了行式数据库的以上不足。

列存储在查询应用时也会存在一些问题。例如，如果查询应用的需求是根据某一列的值进行查询计算，得到相关度最高的前若干条记录编号，并返回这若干条记录的某些指定列(字段)的值。列式数据库在实现这一需求时，因为记录是按列存储的，所以在根据某列的值查询并得到相关记录编号后，并不能立即得到记录的其他指定列的值，而是还需要先查找这些指定列，然后再从这些指定列中查找记录，读取出记录的对应值。这一查找过程降低了查询的性能。

最后，相比于行存储，列存储在磁盘中的每个 Page 仅仅存储来自单列的值，而不是整行的值。因此，压缩算法会更加高效，因为它们能够作用于同类型的数据。

用户在实际使用中，需要考虑自己的具体场景的使用，灵活地选择不同的行列存储方案。下面是对实际的查询中的行表和列表查询占用的时间的一个对比，表格的第一列是在不同条件语句下的查询，gpload 是 Greenplum 提供的加载外部文件的一种方式；第二列和第三列分别是行表和列表的查询时间统计：

| | | |
|---|---|---|
| gpload | 203.21 seconds | 28.93 seconds |
| select orderkey | 582.372 ms | 276.224 ms |
| Select * | 556.327 ms | 610.419 ms |
| select count(1) | 942.440 ms | 710.149 ms |

表 4-3 行列表查询效率排列

从表 4-3 中可以看出，列表查询单个字段的代价小于行表，这种情况在更宽的表之下会表现的更加突出。另外要注意实际查询时间受到机器性能的影响。

### 4.2.2 appendonly 表和压缩技术

Greenplum 中的表分为两种：堆表和 appendonly 表。堆表就是使用正常的建表语句建立起来的表结构。而我们在前面一节使用 with 字句声明 appendonly=true 建立起来的表就是 appendonly 表（AOT）。

而在 with 子句中使用 compresslevel 和 compresstype 声明的表就属于压缩表，压缩表的前提是该表必须是 appendonly 表。

如果一个表声明为 Appendonly 表，那么只能对该表进行 insert 和 truncate 操作而不能 update 或者 delete。简单来说，appendonly 表只能追加数据而不能更改数据。

在 4.3 以上版本的 Greenplum 中，讲 Appendonly 表改成 Appendonly-Optimized 表。用户可以对 AOT 中的元组进行 update 和 delete。数据库会维护一张 visitmap 表来标记 AOT 中每个元组的可见性，删除一条记录会将该记录标记为不可见，而对表的更新跟普通的堆表一样，实际上分解成删除和插入两个动作。所以当经过一段时间的更新之后，需要用 Vacuum 命令对其空间进行回收。就实际而言，AOT 的特性仍然没有改变。

在 Greenplum 中，压缩表的压缩算法主要是 zlib，分成 1-9 个 level，level 越高压缩率越高，但压缩和解压需要的时间也会增加。常

用的压缩级别是 compresslevel=5。需要注意 gpdb 企业版支持 quicklz 和 rle 压缩算法，但开源版本不支持。

压缩表使用于不需要对表进行更新和删除操作，用 truncate 和 insert 就可进行正常的数据分析的应用场景；也不能经常对表进行 alter 操作，对 appendonly 进行增加字段和修改字段类型比普通堆表慢得多；此外，appendonly 不支持唯一索引，访问表基本上是全表扫描。

- **与 AOT 有关的系统表**

orders_zlib 是一张 AOT 表，事实上它也是一张压缩表（建表语句见下面）。我们查询一下在 Greenplum 中的系统表中它的信息：

```
testdb=# select oid from pg_class where relname = 'orders_zlib';
  oid
-------
 24869
(1 row)
```

首先查到它的 oid 是 24869, 然后在系统表中找到跟它有关的表：

```
testdb=# select oid::regclass name from pg_class where relname = 'orders_zlib' or relname like '%24869%';
                name
```

```
---------------------------------
 orders_zlib
 pg_aoseg.pg_aoseg_24869
 pg_aoseg.pg_aoseg_24869_index
 pg_aoseg.pg_aovisimap_24869
 pg_aoseg.pg_aovisimap_24869_index
```

当然，如果是一张列压缩表的话，查询出来的结果可能略有不同：

```
postgres=# select oid::regclass from pg_class where oid = 16675 or relname like '%16675%';
                        oid
---------------------------------
 orders_column_zlib1
 pg_aoseg.pg_aocsseg_16675
 pg_aoseg.pg_aocsseg_16675_index
 pg_aoseg.pg_aovisimap_16675
 pg_aoseg.pg_aovisimap_16675_index
 pg_aoseg.pg_aoblkdir_16675
 pg_aoseg.pg_aoblkdir_16675_index
```

其中，pg_aocsseg 是列表的压缩信息，pg_aoblkdir 是列表在磁盘上的分布信息。

可以看到，除了 orders_zlib 这张表本身，还有 4 张表和它有关。其中有两张是索引，先不做分析。我们先看 pg_aoseg.pg_aoseg_248

69、pg_aoseg.pg_aovisimap_24869 这两张表，pg_aovisimap 中的信息,这里面存储着 AOT 元组的可见性信息。我们查看一下 pg_aoseg.pg_aoseg_24869 的表结构：

```
testdb=# \d pg_aoseg.pg_aoseg_24869
    ?o? "pg_aoseg.pg_aoseg_24869"
            Column      |          Type
  ----------------------------+-------------------------
  -
   Segno                 | integer
   Eof                   | double precision
   tupcount              | double precision
   varblockcount         | double precision
   eofuncompressed       | double precision
   modcount              | bigint
   state                 | smallint
```

表中的字段的含义如下：

| **字段名** | **描述** |
|---|---|
| Segno | File segment 的编号，可能有多行记录，通过 segmo 标识 |
| Eof | 压缩后的文件大小 |
| **字段名** | **描述** |

| | |
|---|---|
| tupcount | 表的元组量 |
| varblockcount | 表占用的块数 |
| eofuncompressed | 未压缩时的表大小 |
| modcount | 被修改的次数 |
| state | 状态量，在 Greenplum 中共有三种状态 |

由于 Greenplum 中表数据都存储在 segment 节点中，可以利用 gp_dist_random（）查询到 segment 中的相关信息：

```
testdb=# select eof,tupcount,eofuncompressed from
                gp_dist_random('pg_aoseg.pg_aoseg_82298');
    eof      | tupcount | eofuncompressed
----------+----------+-----------------
 30433416 |   749995 |        96819984
 30426264 |   749994 |        96805168
```

从这里可以看出，压缩率大致在 sum（eofuncompressed）/ sum(eofuncompressed)=3.18。

- **压缩表的创建**

压缩表和列表一样必须是 appendonly 表，创建一个压缩表的表只需要在创建表的时候，在 with 子句中加入 appendonly=true，compresslevel=*，compresstype=*即可。Level 和 type 可以同时声明也可以只声明其中的一个。默认的压缩类型为 zlib，level 为 1。整个建表

语句如下：

```
CREATE TABLE name
(…)
WITH(appendonly=true, compresslevel=5)
DISTRIBUTED BY(row_name);
```

一张普通的堆表就是利用正常语句建立起来的表，下面是对一组测试数据分别建立一张非压缩表和压缩表，比较两者在存储中所占的空间。

| 类型 | 普通堆表 | 压缩表 |
| --- | --- | --- |
| 占用磁盘 | 218.071040M | 60.863248M |

表 4-4 压缩表占用空间比较

可以看出，压缩率在 3-4 之间。Greenplum 中提供了一个函数——get_ao_compression_ratio()，可以查询压缩表的压缩率：

```
testdb=# select * from get_ao_compression_ratio('order_zlib');
 get_ao_compression_ratio
--------------------------
                     3.18
```

在 Greenplum 中，AOT 的表的相关信息保存在 pg_appendonly 中，相关的字段描述如下：

| **字段名** | **描述** |
| --- | --- |

| | |
|---|---|
| relid | 对应该表的 oid |
| blocksize | 块大小，8kb~2MB，默认为 32KB |
| safefswritesize | 安全写入数据的块大小 |
| compresslevel | 压缩率 |
| majorversion | 最大允许的版本 |
| minorversion | 最小允许的版本 |
| checksum | 是否有校验，跟 gp_appendonly_verify_block_checksum 参数的设置有关 |
| compresstype | 压缩的类型 |
| **字段名** | **描述** |
| columnstore | 是否是列存储 |
| segrelid | pg_aoseg 表的 oid |
| segidxid | pg_aoseg 表索引的 oid |
| blkdirrelid | pg_aoblkdir 表的 oid |
| blkdiridxid | pg_aoblkdir 表索引的 oid |
| version | MemTuples and block layout 的版本信息 |
| visimaprelid | pg_aovisimap 表的 oid |
| visimapidxid | pg_aovisimap 表索引的 oid |

注意：本节列出的表字段跟 Greenplum 的版本有关。

- **Greenplum 压缩表的支持**

目前版本的 Greenplum 支持 zlib，quicklz，rle 三种压缩方式。最新版本的压缩支持可以在 pg_compression 表中查询得到。

- RLE：RLE 算法又叫行程编码算法，是最早出现也是最简单的无损压缩算法。核心思想是将一个有序列表转化为一个三元组（A, B, C）的形式。然后把这些统计之后的数据存储起来。由于统计之后的数据较原始数据占用空间较小，从而实现数据的压缩。
- Quicklz：

  //to do
- ZLIB：ZLIB 使用的 DEFLATE 算法主要包括 RL77 和哈夫曼编码两个步骤。RL77 算法在压缩过程中在内存中会维护一个可以跟随压缩进程滑动的窗口作为术语字典，比起传统的方法，滑动窗口带来的好处就是匹配算法的时间消耗更短。根据构建好的字典，哈夫曼（Huffman）编码算法构造和维护一棵哈夫曼树，每一棵树的左节点编码为 0，右节点编码为 1。然后从根节点遍历得到叶子节点的编码，并使用编码代替原始的数据存储到磁盘中。

## 4.3 实验：向 gp 中添加新的压缩算法

### 4.3.1 Greenplum 创建压缩表及数据的压缩过程分析

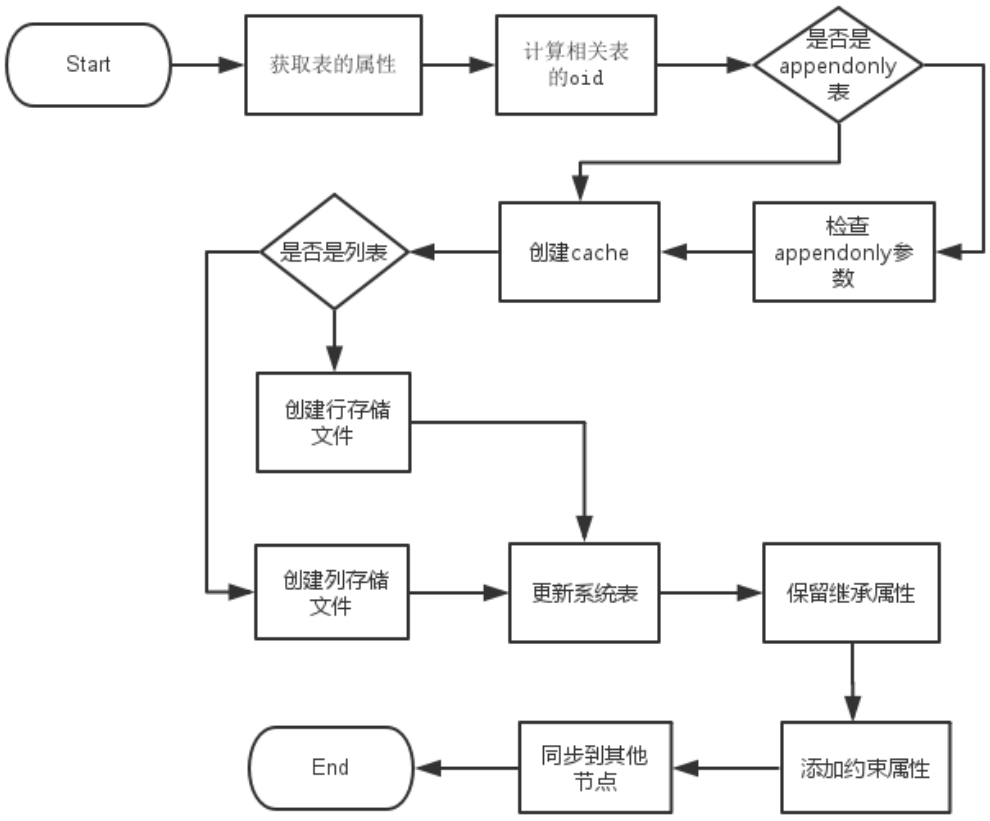



图 4-5 gp 创建表的流程

图 4-5 演示了 Greenplum 是如何创建一张表的。在用户的 create 语句提交之后，表的相关属性在后端被 CreateStmt 类型的变量来保存。这个变量在被初始化之后通过地址传参的方式被传递到 DefineRelation（）函数中。这个函数首先会检查要创建的表的模式和表空间，判断是否有效。然后计算与该表相关的表的 oid 的值（如我们上面说到的 aovisimap 表等），保存在如果该表只是一张普通的表，aovisimap 等表的 oid 值会标记为无效。在准备工作做好之后，

通过调用 heap_create_with_catalog（）函数在内存及本地物理存储介质（磁盘）上创建一张表，然后更新系统表（pg_class,pg_type,pg_appendonly）。然后给表添加继承下来的属性（分布策略等），定义的约束。最后调用 cdbdisp_dispatchUtilityStatement（）把查询树同步到 segment 节点上。

在这个过程中，表的压缩方法以及压缩 level 都是保存在 CreateStmt 类型的变量中，在 heap_create_with_catalog 中被保存到 pg_appendonly 表中，这个表在上一节中有过介绍。在这里可以看到，创建一张 AOT 表和创建一张普通堆表的过程类似，调用的都是同一个函数。只是在创建的过程中会进行两次参数检查，第一次检查要创建的表是否为一张 AOT 表，如果是，则检查 with 语句中的值是否正确。第二次检查要创建的表是否是一张列表，如果是，则物理文件以列存储的形式存储到磁盘中。

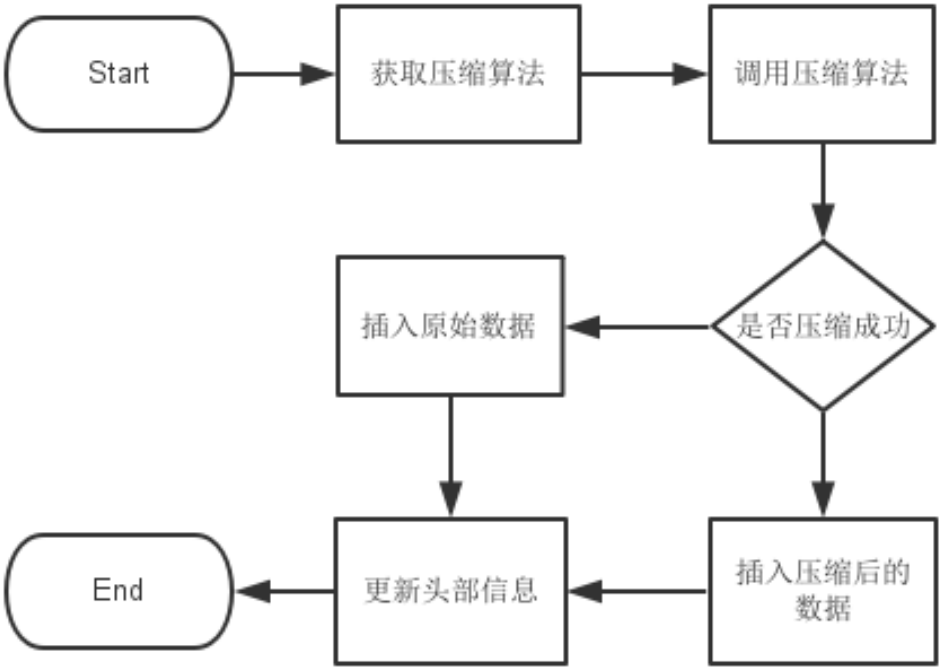



图 4-6 gp 压缩数据的流程

而当我们进行 insert 操作是，如果我们操作的是一张压缩表，那么就需要对用户的插入数据进行压缩。压缩的流程如图 4-6。数据的压缩函数 AppendOnlyStorageWrite_CompressAppend()在很多场景下都会被调用，而不仅仅是 insert 操作中，这里以 insert 为例。该函数的输入是一个 storageWrite 类型的变量，记录了压缩表的相关属性（压缩类型，level 等）以及用户的插入数据。通过 storageWrite 中的压缩算法对插入的数据进行压缩，在这里用到了类似于 java 的反射机制来寻找对应的压缩算法。在压缩之后并不会理解将数据 copy 到表中，而是比较压缩之后的数据跟原始数据的 size，只有 size 减小才被看做压缩成功并将压缩后的数据插入到表中，否则插入原始数据。然后更新 page 的头部信息，如 checksum 等。

### 4.3.2 snappy 压缩算法

Snappy 算法是谷歌开发，目前已被开源的一个数据压缩算法。Snappy 算法不追求最大压缩率，旨在提高数据的压缩速度。Snappy 特地为 64 位 x86 处理器做了优化，作为一个压缩库，它能利用单颗 Intel Corei7 处理器内核处理至少每秒 250MB~500MB 的数据流。

Snappy 的前身是 zippy。Snappy 压缩算法被谷歌应用到自家的很多内部项目中，包括 bigTable，MapReduc 和 RPC。Google 在这个库里面对压缩速度做了很多优化，作为代价，snappy 并没有考虑压缩大小以及兼容性的问题。

另外在健壮性方面，snappy 算法即使遇到损坏或恶意的输入也不会崩溃，并且被 Google 应用到 PB 级的数据压缩。

Snappy 有多重语言的版本，包括 C，C++，Java，Pytho 等。Snappy 采用新 BSD 协议开源。

### 4.3.4 修改源码（以 snappy 为示例）

在 3.4.1 中我们知道，在创建表的过程中会对 AOT 的参数进行检查，其中就包括了检查用户指定的压缩类型是否正确，因此我们需要修改一些代码来使得我们通过 snappy 算法建立的压缩表通过验证。在表的插入阶段，由于需要调用指定的压缩算法，所以我们也要插入新的压缩算法代码。最后，还需要在系统表中插入相关记录及修改 makefile 文件。

第一步：修改创建表时的验证

用户指定的压缩类型保存在一个数组变量中。所以第一步我们需要在该变量的成员中添加 snappy。该数组定义在 pg_compression.c/compresstype_is_valid()函数的 valid_comptypes 数组中。该函数会检查用户指定的压缩类型是否有效，如果无效，会提示“unknown compress type”

此外，还会对 level 进行检查。检查的代码在 reloptions.c/validateAppendOnlyRelOptions()，我们需要做一下修改，在 if 语句添加加粗部分代码:

```
if (comptype &&
    (pg_strcasecmp(comptype, "quicklz") == 0 ||
     pg_strcasecmp(comptype, "zlib") == 0 ||
     pg_strcasecmp(comptype, "rle_type") == 0||
     pg_strcasecmp(comptype, "snappy") == 0))
```

第二步：添加对 snappy 库的调用代码

snappy 是一个开源的代码，我们可以在官网上下载到代码然后进行编译安装。然后在 pg_compression.c 添加对 snappy 库的调用。

首先需要添加 snappy_state 的结构体（见附录 1），该结构体存储了 snappy 算法的相关信息。

之后，添加以下 5 个函数（具体实现见附录 1）

snappy_c_constructor

snappy_c_destructor

snappy_c_compress

snappy_c_decompress

snappy_c_validator

第三步：修改系统表

在创建一个新的数据库集群时，Greenplum 会使用 BKI 文件完成其工作的一部分。下面是 PostgreSQL 用户文档对于 BKI 文件的介绍：

“后端接口（BKI）文件是一些用特殊语言写的脚本，这些脚本是 PostgreSQL 后端能够理解的，以特殊的 "bootstrap"（引导）模式运行，这种模式允许在不存在系统表的零初始条件下执行数据库函数，而普通的 SQL 命令要求系统表必须存在。 因此 BKI 文件可以用于在第一时间创建数据库系统。 （可能除此以外也没有其它用处。）”

--Postgresq 中文手册

因此，我们需要修改相关的文件以达到系统初始化时在系统表 pg_compression 和 pg_proc 添加 snappy 的记录。

① 修改 include/pg_compression.h 文件

添加 snappy 算法在 pg_compression 表中的记录。

② 修改 include/gp_proc.h 文件

添加 snappy 算法的调用函数在 pg_proc 表中的记录。

在 initdb 期间，会检查压缩算法是否存在以及对应的函数是否存在。如果不进行这步插入，会在 gpinitsystem 阶段报找不到对应函数的错误。

第四步：修改 makefile

在 backend/makefile 文件中，添加如下记录，在编译过程中动态链接到 snappy 库中：

LIBS := $(LIBS) –lsnappy

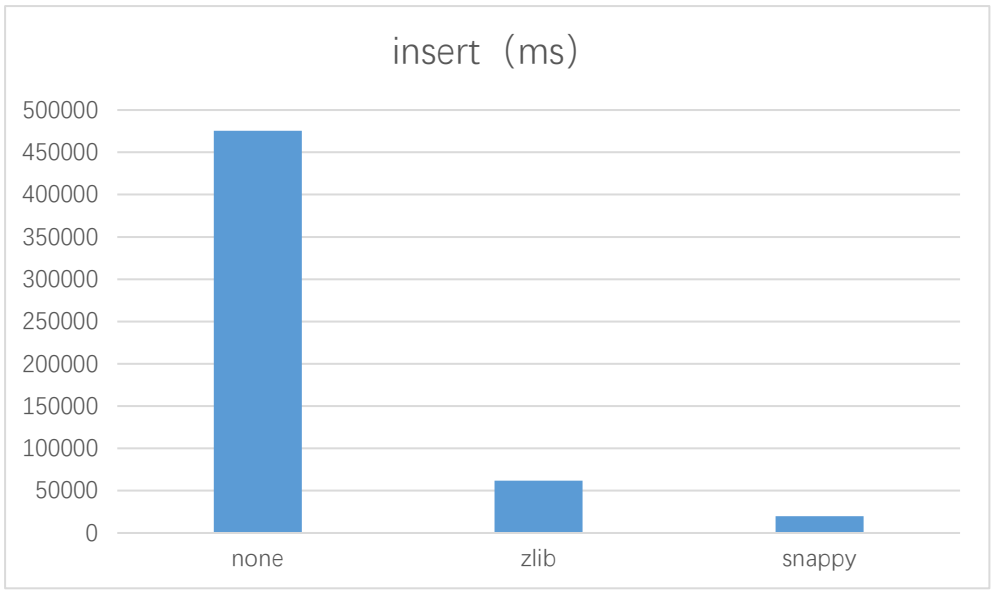
第五步 :修改系统变量, 把 snappy 的安装目录添加到系统变量中。

在/etc/ld.so.conf 中添加 snappy 的安装目录，然后执行命令 ldconfig 使其生效。

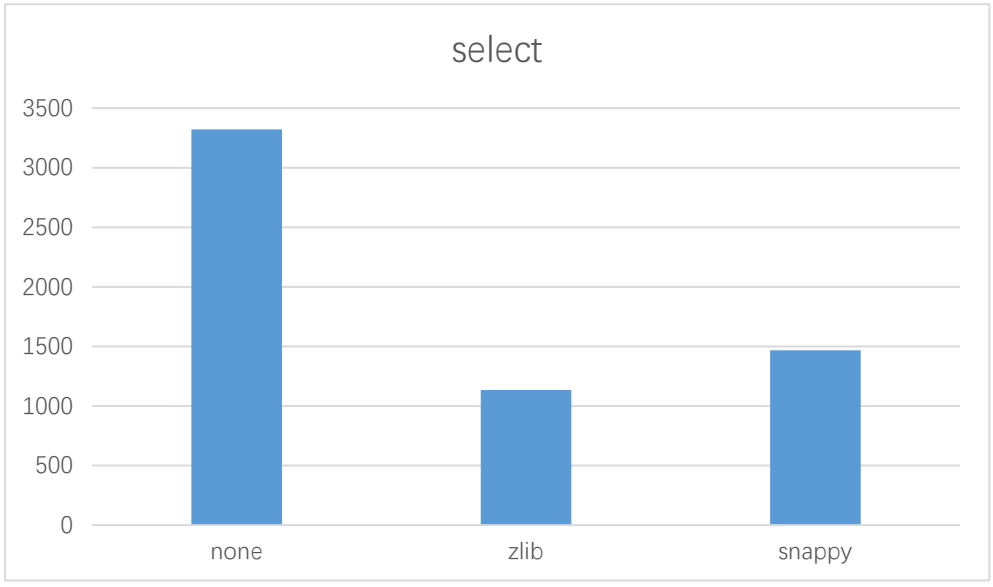
添加 lzo 进 gp 中的过程类似，只需要修改结构体使得 compress_fn 和 decompress_fn 和 lzo 提供的接口的函数参数一致即可。

### 4.3.5 压缩效率比较

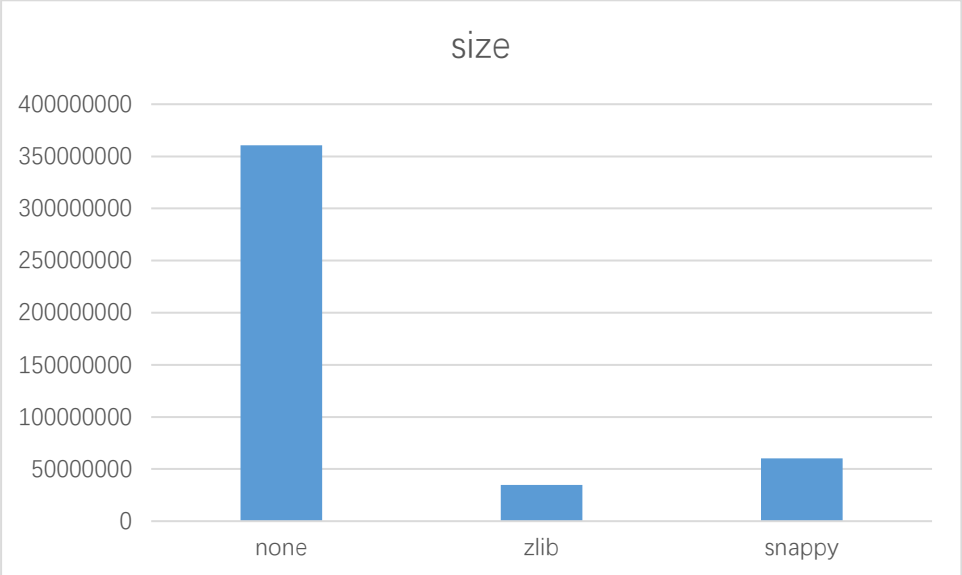
插入了 10000000 条数据，由于数据是整形按照 id 递归插入的，比较规律，可以看到压缩率很高。Snappy 的压缩速度比 zlib 快。




对插入的数据进行 select 操作，可以看到 zlib 的解压缩速度比 snappy 快。




比较三者的大小，可以看出 zlib 的存储占用比 zlib 多了一倍左右。




## 4.4 存储机制实验导引

### 4.4.1 存储机制实验实例

在本部分的实验中，我们会建立三张表：一张行表，一张列表，一张压缩表。通过比较行表和列表的查询时间以及普通行表与压缩表的占用的空间，分析它们的优缺点。由于压缩表常常和列表一起使用，所以在这里建立的压缩表也是一张列表。

1）建立一张行表

```
testdb=# create table ORDERS_ROW (O_ORDERKEY int,O_CUSTKEY int,O_ORDERSTATUS cha
r(1),O_TOTALPRICE double precision,O_ORDERDATE DATE,O_ORDERPRIORITY CHAR(15),O_C
LERK CHAR(15),O_SHIPPRIORITY INTEGER,O_COMMENT VARCHAR(78)) with(appendonly=true
,orientation = column) Distributed by(O_ORDERKEY);
CREATE TABLE
```

2） 建立一张列表

```
testdb=# create table ORDERS_COL (O_ORDERKEY int,O_CUSTKEY int,O_ORDERSTATUS cha
r(1),O_TOTALPRICE double precision,O_ORDERDATE DATE,O_ORDERPRIORITY CHAR(15),O_C
LERK CHAR(15),O_SHIPPRIORITY INTEGER,O_COMMENT VARCHAR(78)) with(appendonly=true
,orientation = column) Distributed by(O_ORDERKEY);
CREATE TABLE
```

3） 建立一张压缩表，压缩等级为 1

```
testdb=# create table ORDERS_COMP (O_ORDERKEY int,O_CUSTKEY int,O_ORDERSTATUS ch
ar(1),O_TOTALPRICE double precision,O_ORDERDATE DATE,O_ORDERPRIORITY CHAR(15),O_
CLERK CHAR(15),O_SHIPPRIORITY INTEGER,O_COMMENT VARCHAR(78)) with(appendonly=tru
e,orientation = column,compresslevel=1) Distributed by(O_ORDERKEY);
CREATE TABLE
```

4）利用 gpload 进行数据导入

使用 gpload 导入数据需要首先建立 yml 文件，本实验建立的文件内容如下：

```
[gpadmin@localhost ~]$ more gpload.yml
VERSION: 1.0.0.1
DATABASE: testdb
USER: gpadmin
HOST: 127.0.0.1
PORT: 5432
GPLOAD:
  INPUT:
    - SOURCE:
        LOCAL_HOSTNAME:
          - mdw
        PORT: 55555
        FILE:
          - /home/gpadmin/Desktop/neworder1.tbl
    - COLUMNS:
        - o_orderkey: integer
        - o_custkey: integer
        - o_orderstatus: character(1)
        - o_totalprice: double precision
        - o_orderdate: date
        - o_orderpriority: character(15)
        - o_clerk: character(15)
        - o_shippriority: integer
        - o_comment: character varying(78)

    - FORMAT: text
    - DELIMITER: '|'
    - ERROR_LIMIT: 25
  OUTPUT:
    - TABLE: orders_row
    - MODE: INSERT
```

我们通过 vim 命令分别建立三个配置文件，分别对应不同的表。三张文件的基本结构都类似，只是倒数第二行的 TABLE:***需要被替换成不同的表的名字。然后开始导入：

```
[gpadmin@localhost ~]$ gpload -f gpload.yml
2016-08-02 10:49:28|INFO|gpload session started 2016-08-02 10:49:28
2016-08-02 10:49:28|INFO|setting schema 'public' for table 'orders_row'
2016-08-02 10:49:28|INFO|started gpfdist -p 55555 -P 55556 -f "/home/gpadmin/Desktop/neworder1.tb
l" -t 30
2016-08-02 10:51:02|INFO|running time: 94.46 seconds
2016-08-02 10:51:03|INFO|rows Inserted          = 1499989
2016-08-02 10:51:03|INFO|rows Updated           = 0
2016-08-02 10:51:03|INFO|data formatting errors = 0
2016-08-02 10:51:03|INFO|gpload succeeded
```

导入行表数据

```
[gpadmin@localhost ~]$ gpload -f gpload_col.yml
2016-08-02 10:54:14|INFO|gpload session started 2016-08-02 10:54:14
2016-08-02 10:54:14|INFO|setting schema 'public' for table 'orders_col'
2016-08-02 10:54:14|INFO|started gpfdist -p 55555 -P 55556 -f "/home/gpadmin/Desktop/neworder1.tb
l" -t 30
2016-08-02 10:55:19|INFO|running time: 65.31 seconds
2016-08-02 10:55:20|INFO|rows Inserted          = 1499989
2016-08-02 10:55:20|INFO|rows Updated           = 0
2016-08-02 10:55:20|INFO|data formatting errors = 0
2016-08-02 10:55:20|INFO|gpload succeeded
```

导入列表数据

```
[gpadmin@localhost ~]$ gpload -f gpload_comp.yml
2016-08-02 10:57:35|INFO|gpload session started 2016-08-02 10:57:35
2016-08-02 10:57:35|INFO|setting schema 'public' for table 'orders_comp'
2016-08-02 10:57:35|INFO|started gpfdist -p 55555 -P 55556 -f "/home/gpadmin/Desktop/neworder1.tb
l" -t 30
2016-08-02 10:57:59|INFO|running time: 24.29 seconds
2016-08-02 10:58:00|INFO|rows Inserted          = 1499989
2016-08-02 10:58:00|INFO|rows Updated           = 0
2016-08-02 10:58:00|INFO|data formatting errors = 0
2016-08-02 10:58:00|INFO|gpload succeeded
```

导入压缩表数据

4） 全表查询时间统计

```
testdb=# select * from orders_row where o_orderkey = 1;
 o_orderkey | o_custkey | o_orderstatus | o_totalprice | o_orderdate | o_orderpr
------------+-----------+---------------+--------------+-------------+----------
          1 |     36901 | O             |    173665.47 | 1996-01-02  | 5-LOW
(1 row)

Time: 10576.181 ms
testdb=# select * from orders_col where o_orderkey = 1;
 o_orderkey | o_custkey | o_orderstatus | o_totalprice | o_orderdate | o_orderpr
------------+-----------+---------------+--------------+-------------+----------
          1 |     36901 | O             |    173665.47 | 1996-01-02  | 5-LOW
(1 row)

Time: 11491.899 ms
testdb=# select * from orders_comp where o_orderkey = 1;
 o_orderkey | o_custkey | o_orderstatus | o_totalprice | o_orderdate | o_orderpr
------------+-----------+---------------+--------------+-------------+----------
          1 |     36901 | O             |    173665.47 | 1996-01-02  | 5-LOW
(1 row)

Time: 3734.633 ms
```

5） 单属性列查询时间统计

```
testdb=# select o_custkey from orders_row where o_orderkey = 1;
 o_custkey
-----------
     36901
(1 row)

Time: 685.269 ms
testdb=# select o_custkey from orders_col where o_orderkey = 1;
 o_custkey
-----------
     36901
(1 row)

Time: 254.772 ms
testdb=# select o_custkey from orders_comp where o_orderkey = 1;
 o_custkey
-----------
     36901
(1 row)
```

6） 占用空间统计

```
testdb=# select pg_relation_size('orders_row');
 pg_relation_size
------------------
        161478024
(1 row)

testdb=# select pg_relation_size('orders_col');
 pg_relation_size
------------------
        161478088
(1 row)

testdb=# select pg_relation_size('orders_comp');
 pg_relation_size
------------------
         45293056
(1 row)
```

可以看出，行表在导入的效率很低；在单属性的查询上，列表和压缩表的效率远大于行表；而压缩表的占用空间（以 Byte 为单位）也小于非压缩表。

### 4.4.2 存储机制实验要求

1、在 Greenplum 对相同数据采用不同的存储方式验证两者的特性；（数据集最好大一点效果明显）

2、在 Greenplum 中比较压缩表与非压缩表在存储空间和读取性能的差异；

3、可以给某一属性列添加索引，比较在各种存储方式下索引对性能的影响。

## 4.5 参考文献

[1] Stonebraker M, Abadi D J, Batkin A, et al. C-store: a column-oriented DBMS[C]// International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September. DBLP, 2005:553-564.

[2] Lamb A, Fuller M, Varadarajan R, et al. The Vertica Analytic Database: C-Store 7 Years Later[J]. Computer Science, 2012, 5(12).

[3] Abadi D J, Madden S R, Hachem N. Column-stores vs. row-stores:how different are they really?[C]// ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, Bc, Canada, June. DBLP, 2008:967-980.

[4] Abadi D J, Myers D S, Dewitt D J, et al. Materialization Strategies in a Column-Oriented DBMS[C]// IEEE, International Conference on Data Engineering. IEEE, 2007:466-475.

[5]https://greenplum.org/

[6]Chang F, Dean J, Ghemawat S, et al. Bigtable:A Distributed Storage System for Structured Data[J]. Acm Transactions on

Computer Systems, 2008, 26(2):1-26.

[7]https://www.mongodb.com/

[8]Robinson I, Webber J, Eifrem E. Graph Databases[M]. O'Reilly Media, Inc. 2013:152-170.

[9]http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html