[译] [论文] Bigtable: A Distributed Storage System for Structured Data (OSDI 2006)

0.1 译者序

本文翻译自 2006 年 Google 的分布式存储经典论文：**Bigtable: A Distributed Storage System for Structured Data** (PDF)。

标题直译为：**《Bigtable：适用于结构化数据的分布式存储系统》**。

本文对排版做了一些调整，以更适合网页阅读。

翻译仅供个人学习交流。由于译者水平有限，本文不免存在遗漏或错误之处。如有疑问，请查阅原文。

---

1 摘要

Bigtable 是一个用于管理**结构化数据**（structured data）的**分布式存储系统**，设计可以扩展到非常大的规模：由几千个通用服务器（commodity servers）组成的 PB 级存储。

很多 Google 产品，包括 web index、Google Earth 和 Google Finance，都将数据存储在 Bigtable 中。不过，这些应用对 Bigtable 的要求有很大差异，不管是从数据大小（从URL 到网页到卫星图像）还是从延迟（从后台批量处理到实时数据服务）考虑。但是，Bigtable 仍然给这些产品提供了一个灵活、高性能的解决方案，它提供的简单数据模型可使**客户端动态控制**数据的**布局和格式**（layout and format）。

本文介绍 Bigtable 的设计与实现。

2 引言

在过去的两年半中，我们设计、实现并部署了一个称为 Bigtable 的分布式存储系统，用于管理 Google 的结构化数据。设计中 Bigtable 能可靠地扩展到 **PB 级数据，上千个节点**。现在已经实现了广泛的应用场景支持、可扩展性、高性能，以及高可用性等设计目标。

目前 Bigtable 已经被超过 60 个 Google 产品和项目所使用，其中包括 Google Analytics、Google Finance、Orkut、Personalized Search、Writely、以及 Google Earth。这些产品的使用场景差异很大，从面向吞吐的批处理任务，到延迟敏感的终端用户数据服务。不同产品使用的 Bigtable 集群配置差异也很大，有的集群只有几台节点，有的有几千台，存储几百 TB 的数据。

从某些方面看，Bigtable 像是一个数据库：它的很多实现策略（implementation strategies）确实和数据库类似。**并行数据库** [14]（Parallel databases）和**主存数据库** [13]（main-memory databases）已经在可扩展性和高性能方面取得了很大成功，（Bigtable 也关注这两方面，但除此之外，）Bigtable 提供的接口与它们不同。

Bigtable 不支持完整的关系型数据模型（full relational data model）；它提供给客户端的是一个**简单数据模型**（simple data model），支持**动态控制数据的布局和格式**（layout and format），并允许客户端推测**数据在底层存储中的 locality（本地性）特性**。数据使用**行名和列名**（row and column names）进行索引，这些名字可以是任意字符串（strings）。

Bigtable **不理解数据的内容**（将数据视为 uninterpreted strings），虽然很多字符串都是客户端将各种结构化和半结构化数据（structured and semi-structured data）序列化而来的。客户端可以通过精心**选择 schema 来控制数据的 locality**。schema 参数还可以让客户端动态控制数据是从内存还是磁盘读取（serve）。

3 数据模型

一个 Bigtable 就是一个**稀疏、分布式、持久的多维有序**映射表（map），数据通过**行键、列键和一个时间戳**进行索引，表中的每个数据项都是**不作理解的字节数组**。

> A Bigtable is a sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

映射：`(row:string, column:string, time:int64) -> string`

> **注解：**
>
> $$\text{BigTable: RowKey : string} \times \text{ColumKey : string} \times \text{Timestamp : int64} \rightarrow \text{string} \tag{1}$$

我们首先评估了类似 Bigtable 这样的系统有哪些潜在的使用场景，然后才确定了数据模型。举个具体例子，这个例子也影响了 Bigtable 的一些设计：我们想保存大量的网页和网页相关的元数据，这些数据会被不同的项目使用，这里将这张表称为 `Webtable`。

在 `Webtable` 中，我们用网页的 URL 作为行键，网页某些信息作为列键，将网页内容存储在 `contents:` 列，并记录抓取网页时对应的时间戳，最终存储布局如图 1 所示。

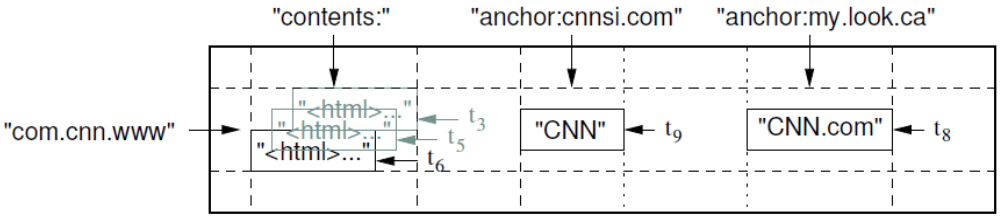


图 1 存储网页的 bigtable 的一个切片（slice）

- 行索引：`URL`
- `contents:` 列：存储页面内容（page content）
- `anchor:` 开头的列：存储引用了这个页面的 anchor（HTML 锚点）的文本（text of the anchors that reference this page）

图中可以看出，CNN 主页被 Sports Illustrated（`cnnsi.com`）和 MY-look 主页（`my.look.ca`）引用了，因此会有 `anchor:cnnsi.com` 和 `anchor:my.look.ca` 两列，其中每列一个版本；`contents:` 列有三个版本，时间戳分别为 `t3`、`t5` 和 `t6`。

3.1 行（Row）

行键（row key）可以是**任意字符串**（目前最大支持 64KB，大部分用户使用的 key 都在 10-100 字节之间）。

单行数据的读/写操作是原子的（不管该行有多少列参与读/写），这样的设计使得多个客户端并发更新同一行时，更容易推断系统的行为。

Bigtable 中的数据是根据**行键的词典顺序（lexicographic order）**组织的，并**动态地对行范围（row range）进行切分（partition）**。

> **注解：**
>
> $$\begin{aligned} & r_x \in \text{Rows} \\ & t_n = \{r_n, r_{n+1}, \cdots, r_{n+m}\} \in \text{Tablets} \\ & \text{Rows} = \{\underbrace{r_0, \cdots, r_k}_{t_1}, \underbrace{r_{k+1}, \cdots, r_l}_{t_2}, \cdots\} \end{aligned} \tag{2}$$
>
> 行Key（64 KB）是根据字典排序，根据范围切分为不同的 tablet
> Tablet 是负载均衡的单位

每个行范围称为一个 tablet，这是**请求分散和负载均衡的单位**（unit of distribution and load balancing）。因此，**读取一个较小的行范围（short row ranges）是很高效的**，通常情况下只需要和很少的几台机器通信。客户端可以利用这个特性，通过合理的选择行键来在访问数据时获得更好 locality。

举个例子，在 Webtable 中，将 URL 的 hostname 字段进行翻转，来自相同域（domain）的页面在存储时就会变成连续的行。例如 `maps.google.com/index.html` 页面在存储时行键就是 `com.google.maps/index.html`。**来自相同域的页面存储到连续的行**，会使那些针对主机和域的分析（host and domain analyses）非常高效。

3.2 列族（Column Families）

多个**列键**（column keys）可以组织成 **column families**（列族）。**column family 是访问控制（access control）的基本单位**。

一般来说，存储在**同一 column family 内的数据，其类型都是相同的**（我们会将同一 column family 内的数据压缩到一起）。

必须**先创建一个 column family**，才能向这个 column family 内的列写入数据；创建完成后，就可以在这个 family 内使用任何的列键。我们有意使得**每个 table 内的 column family 数量尽量少**（最多几百个），并且在随后的过程中 family 很少有变化。

但另一方面，每个 table 的**列数量并没有限制**。

列键的格式：`family:qualifier`，其中 `family` 必须为可打印的（printable）字符串，但 `qualifier`（修饰符）可以为任意字符串。例如，Webtable 中有一个 colum family 是**语言（language）**，用来标记每个网页分别是用什么语言写的。在这个 column family 中我们只用了一个列键，其中存储的是每种语言的 ID。Webtable 中的另一个 column family 是 anchor，在这个 family 中每一个列键都表示一个独立的 anchor，如图 1 所示，其中的修饰符（qualifier）是引用这个网页的 anchor名字，对应的数据项内容是链接的文本（link text）。

访问控制（access control）和磁盘及内存记账（accounting）都是在 column family 层做的。还是以 Webtable 为例，这种级别的控制可以使我们管理几种不同类型的应用：有的只添加新的基础数据进来，有的读取基础数据后创建衍生的 column family，有的只允许查看当前的数据（甚至可以根据保密程度只允许查看一部分 column family）。

> **注解：**

Row Key	Product Info:Name	Product Info:Description	Product Info:Category	Inventory:Quantity	Inventory:Supplier	Sales:Price	Sales:Discount
prod001	Smartphone X	High-end smartphone...	Electronics	100	TechSupplier Inc	999.99	0.1
prod002	Cotton T-Shirt	Comfortable cotton...	Apparel	500	FashionWholesale	19.99	0
prod003	Coffee Maker	Automatic drip...	Home Appliances	50	KitchenGoods Ltd	79.99	0.15

例如对上面，我们存在列key: `Product Info:Name` 、 `Product Info:Description` 、 `Product Info:Category` 、 `Inventory:Quantity` 、 `Inventory:Supplier`

而 `Product Info:Name` 、 `Product Info:Description` 、 `Product Info:Category` 构成 `Product Info` 这个 Column Family

$$\underbrace{\text{anchor}}_{\text{family}} : \underbrace{\text{google.com}}_{\text{qualify}} \tag{3}$$

访问控制和磁盘内存记账颗粒度在于 column family

3.3 时间戳

Bigtable 中的**每个数据都可以存储多个版本**，不同版本用时间戳索引。

时间戳是 64 位整数，可以由 Bigtable 指定，这种情况下就是毫秒（ms）级的真实时间戳；也可以由客户端应用指定。如果是应用指定，那为了避免冲突，应用必须保证时间戳的唯一性。

同一数据的不同版本以时间戳**降序**（decreasing timestamp order）的方式存储，这样首先读到的都是最新的版本。

为避免版本化数据的管理过于繁琐，我们提供了两个配置参数可以让 Bigtable 自动进行垃圾回收（GC）。客户端可以指定：

- 保留最后的 N 个版本
- 保留最近的某段时间内的版本（例如，只保留过去 7 天写入的版本）

在 Webtable 中，每个页面的时间戳是该页面被爬取时的时间，我们设置只保留最后的 3个版本。

注解：

`int64 timestamp`

根据时间降序（desc）。有几种保存手段：

- 保留最后 N 个版本
- 保留某段时间的版本（e.g., 过去 N 天）

4 API

注解：

BigTable API 提供了创建、删除 Table&Col Family

即只有增删查的能力。

Bigtable API 提供了创建、删除 table 和 column family 的功能。另外，它还提供了更改集群、table 和 column family 元数据的能力，例如访问控制权限。

客户端应用可以读/写 Bigtable 中的值，从指定行中查找值，或者对 table 内的一个数据子集进行遍历。

图 2 是向 Bigtable 写数据的一段 C++ 代码，使用了 `RowMutation` 抽象来执行一系列更新操作。为保持代码简洁，例子中去掉了一些无关的技术细节。

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

图 2 Writing to Bigtable

`Apply()` 向 Webtable 执行一次原子操作，其中包括：添加一个 anchor 到 `www.cnn.com`，删除另一个 anchor。

图 3 是另一个例子，使用一个 `Scanner` 抽象对一行内的所有 anchor 进行遍历。

```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
  printf("%s %s %lld %s\n",
         scanner.RowName(),
         stream->ColumnName(),
         stream->MicroTimestamp(),
         stream->Value());
}
```

图 3 Reading from Bigtable

客户端可以在多个 column family 上进行遍历，这里有几种限制 scan 产生的行、列和时间戳的机制。例如，可以指定以上 scan 只产生列键能匹配正则表达式 `anchor:*.cnn.com` 的 anchors，或者时间戳在最近 10 天内的 anchor。

Bigtable 还提供其他的一些特性，使得用户可以对数据进行更复杂的控制。

首先，提供了**单行事务**（single-row transaction），可以对单行内的数据执行原子的“读-修改-写”（read-modify-write）序列操作。但 Bigtable 目前并不支持通用的跨行事务（general transactions across row keys），虽然它提供了在客户端侧进行**跨行批量写**（batching writes across row keys）的接口。

第二，允许将 cell（table 中的一个格子）当整型计数器用。

最后，支持在服务端执行由客户端提供的脚本。脚本使用的是 Google 为数据处理开发的称为 Aawzall [28] 的语言。目前这套基于 Sawzall 的 API 不允许客户端脚本将数据回写到 Bigtable，但它们可以进行各种形式的数据变换、计算、求和等等。

Bigtable 可以和 MapReduce [12] 一起使用，后者是 Google 开发的一个大规模并行计算框架。我们写了一些封装函数，将 Bigtable 用作 MapReduce job 的输入源和输出目标。

5 外部系统依赖（Building Blocks）

Bigtable 构建在其他几个 Google 的基础设施之上。

- GFS
- SSTable
- Chubby

5.1 GFS

Bigtable 使用分布式文件系统 GFS（Google File System）[17] 存储日志和数据文件。

Bigtable 集群通常**和其他一些分布式应用共享一个服务器资源池**（pool of machines），而且 **Bigtable 进程经常和其他应用混跑在同一台机器上**。

Bigtable 依赖一个集群管理系统来调度任务、管理共享的机器上的资源、处理机器故障，以及监控机器状态。

5.2 SSTable

Bigtable **内部使用 Google 的 SSTable 格式存储数据**。

SSTable 是一个**持久化的、有序的、不可变的**映射表（map），其中的**键和值都可以是任意字节字符串**。它提供了按 key 查询和对指定的 key range 进行遍历的操作。

> An SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings.

在内部，每个 SSTable 都包含一系列的 **blocks**（通常每个 block 64KB，但这个参数可配置）。

block 用 **block index**（存储在 SSTable 的末尾）来定位，block index会在打开 SSTable 的时候加载到**内存**。

一次查询操作只需要一次磁盘寻址（disk seek）：首先在**内存**中通过二分查找（binary search）找到 block index，然后定位到 block 在**磁盘**中的位置，从**磁盘**读取相应的数据。另外，也可以**将整个 SSTable 映射到内存**，这样查询就完全不需要磁盘操作了。

5.3 Chuby

Bigtable 依赖 Chubby —— 一个高可用、持久的分布式锁服务（a highly-available and persistent distributed lock service）[8]。

一个 Chubby 服务由 **5 个活跃副本**（active replicas）组成，其中一个会被选举为 master，并负责处理请求。只有大多数副本都活着，并且互相之间可以通信时，这个服务才算活着（live）。

在遇到故障时，Chubby 使用 Paxos 算法 [9, 23] 保证副本之间的一致性。

Chubby 提供了一个包含**目录和小文件**的命名空间（namespace），**每个目录或文件都可以作为一个锁**，读或写一个文件是**原子的**。

Chubby 客户端库维护了一份这些文件的一致性缓存（consistent caching）。每个 Chubby 客户端都会和 Chubby 服务维持一个 session。当一个客户端的租约（lease）到期并且无法续约（renew）时，这个 session 就失效了。**session 失效后会失去它之前的锁和打开的文件句柄**（handle）。Chubby 客户端还可以在 Chubby 文件和目录上注册回调函数，当文件/目录有变化或者 session 过期时，就会收到通知。

Bigtable 使用 Chubby 完成很多不同类型的工作：

1. 保证任何时间最多只有一个 active master
2. 存储 Bigtable 数据的 bootstrap location（见 5.1）
3. tablet 服务发现和服务终止清理工作（见 5.2）
4. 存储 Bigtable schema 信息（每个 table 的 column family 信息）
5. 存储访问控制列表

如果 Chubby 服务不可用超过一段时间，Bigtable 也将变得不可用。我们近期对 14 个 Bigtable 集群（总共依赖 11 个 Chubby 集群）的测量显示，由于 Chubby 不可用（网络或 Chubby 本身问题引起的） 导致的 Bigtable 不可用时间（数据在 Bigtable 中但无法访问）百分比平均为 `0.0047%`，受影响最大的那个集群为 `0.0326%`。

6 实现

Bigtable 主要由三个组件构成：

1. 一个客户端库，会链接到每个客户端
2. 一个 master server
3. 多个 tablet server

可以根据系统负载动态地向集群添加或删除 tablet server。

master 负责：

1. 将 tablet 分配给 tablet server
2. 检测 tablet server 的过期（expiration）及新加（addition）事件
3. 平衡 tablet server 负载
4. 垃圾回收（GC）
5. 处理 schema 变动，例如 table 和 column family 的创建

每个 tablet server 管理一组 tablets（一般 10 ~ 1000 个）。tablet server 管理这些 tablet 的读写请求，并且当 tablet 太大时，**还负责对它们进行切分**（split）。

和很多单 master（single master）分布式存储系统一样 [17, 21]，**客户端数据不经过 master 节点：读写请求直接到 tablet server**。由于**客户端不依赖 master 就能确定 tablet 位置信息**，因此大部分客户端从来不和 master 通信。因此，实际中 master 节点的负载很低。

每个 Bigtable 集群会有很多张 table，每张 table 会有很多 tablets，每个 tablets 包含一个 row range（行键范围）内的全部数据。初始时每个 table 只包含一个 tablet。当 table 逐渐变大时，它会自动分裂成多个 tablets，**默认情况下每个 tablet 大约 100-200MB**。

注解：

$$\text{Bigtable} = \{\text{Library}, 1 \times \text{Master}, N \times \text{Tablet Server}\} \tag{4}$$

Master 将 Tablet 分配给 TS。检测 TS 的 Expire & Add 事件。平衡负载，GC，Schema变动。

TS 管理一组 TBLTs（10-1000个），当 TBLTs 过大进行拆分。

每个BT 集群有多张 Table，每张Table 有很多 Tablets

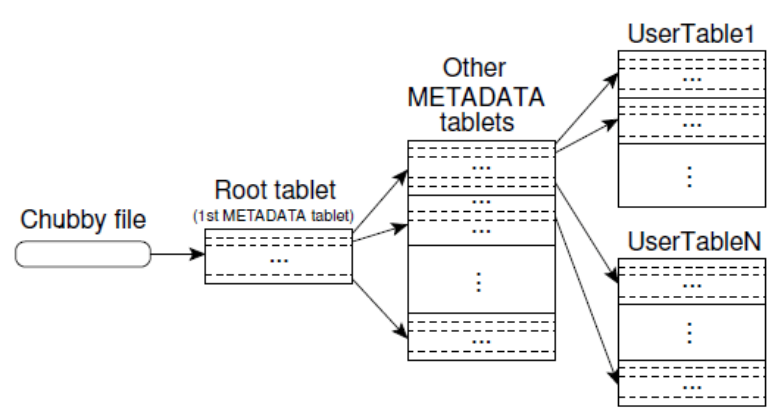
$$\begin{aligned} \text{Tablet Server} &= \text{Table}^N \\ \text{Table} &= \text{Tablet}^M \end{aligned} \tag{5}$$

最开始，Table 只有一个 Tablet（$M = 0$），Table变大会将 Tablet 分割成多个（$M > 0$）。每个Tablet 100-200M。

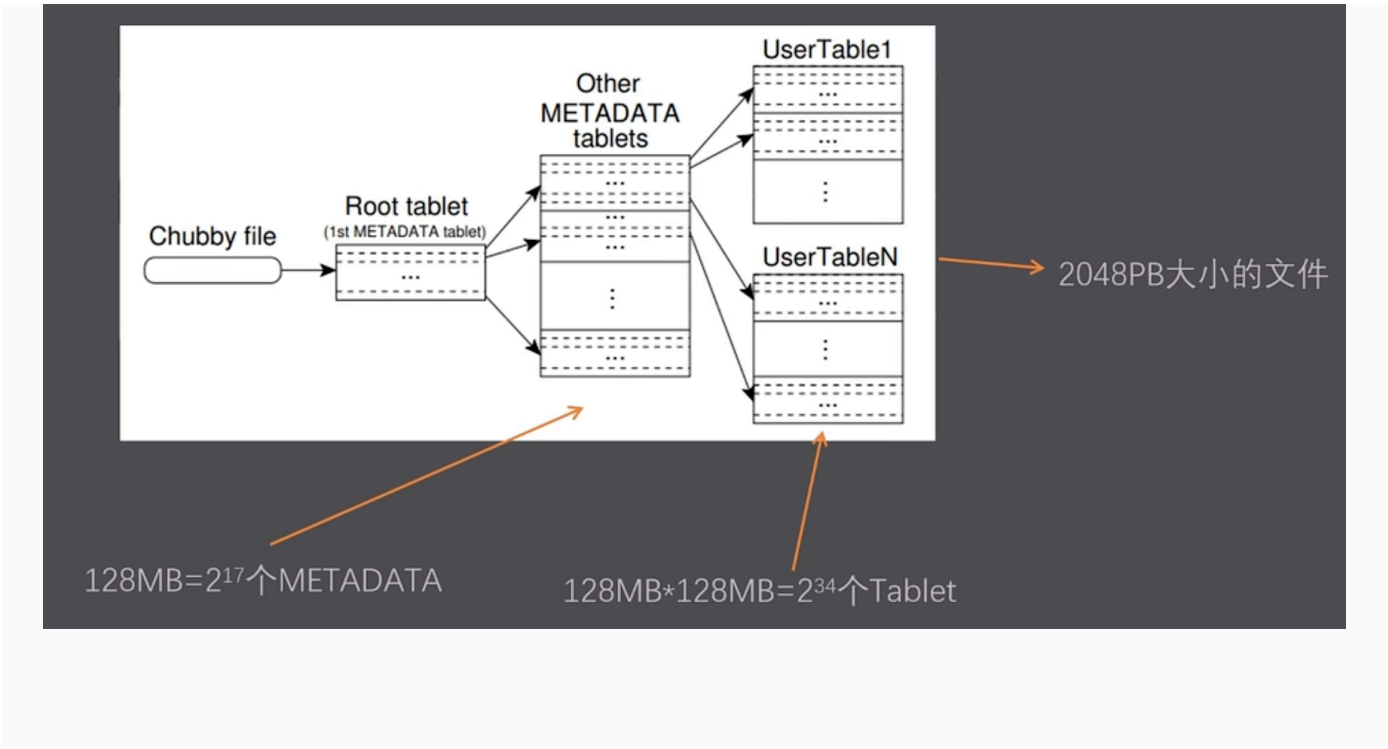
一个 Tablet 对应 1 个 GFS 中的文件（全局唯一文件名）
逻辑上一个 Tablet 是一个 LSM 树

GFS更支持追加写而不是改写。LSM的所有写操作都是追加写。

查找 Tablet 文件位置



Level 0 入口文件：存储与 Chubby，包含Root Tablet。Chubby起选举保证高可用作用。
Level 1 Root Tablet 包含一个 METADATA表，记录METADATA里所有Tablets的位置
Level 2 METADATA：每个 Tablets 包含一个 UserTablet的集合
Level 3 UserTables：由多个UserTable存储的最终数据项



6.1 Tablet 位置

6.1.1 服务端

我们使用一个和 B+ 树 [10] 类似的**三级结构**（three level hierarchy）来存储 tablet 位置信息，如图 4 所示。

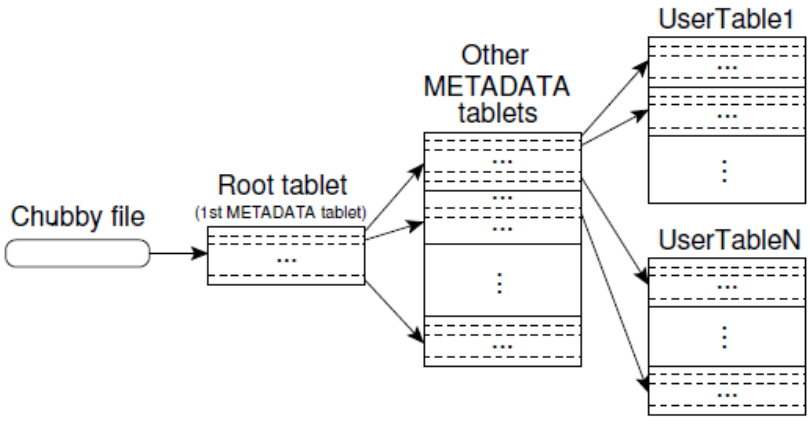


图 4 Tablet location hierarchy

- 第一级：**Chubby 中的一个文件**
- 第二级：**METADATA tables**（第一个 `METADATA` table 比较特殊，所以在图中单独画出，但它其实和其他 `METADATA` table 都属于第二级）
- 第三级：**User Tablets**

`METADATA` 是一个特殊的 tablet，其中的第一个 tablet 称为 **root tablet**。root tablet 和 `METADATA` 内其他 tablet 不同之处在于：它**永远不会分裂**（split），这样就可以**保证 tablet location 层级不会超过三层**。

三级间的关系：

- Chubby 中的文件保存了 root tablet 的位置

- Root tablet 保存了 `METADATA` table 内所有其他 table 的位置
- 每个 `METADATA` tablet（root tablet 除外）保存了一组 user tablet 的位置

`METADATA` table 存储 user tablet 位置信息的方式（假设 user table 名为 `UserTableX`）：

- value：`UserTableX` 的位置
- key（row key）：`UserTableX` 的 table ID 和它的最后一行的某种**编码**（encoding）

> The METADATA table stores the location of a tablet under a row key that is an encoding of the tablet's table identifier and its end row.

`METADATA` 的**每行数据在内存中大约占 1KB**。如果将 `METADATA` tablet 限制在 `128MB` 这样一个中等大小，这种三级位置方案就可以存储高达 `2^34` 个 tablets（`128MB` = `2^17 * 1KB`，即 `METADATA` table 可以指向 `2^17` 个 user table，每个user table 同样是 `128MB` 的话，就有 `2^17 * 2^17 = 2^34` 个 tablets，译者注）。如果每个 tablet 128 MB 大小，那总数据量就高达 `2^61` 字节（`128MB = 2^27 Byte`，`2^34 * 2^27 = 2^61`，即 **`2000PB`**）。

> With a modest limit of 128 MB METADATA tablets, our three-level location scheme is sufficient to address 234 tablets (or 2^61 bytes in 128 MB tablets).

6.1.2 客户端

客户端库会缓存 tablet 位置信息。
如果客户端不知道 tablet 的位置，或者发现缓存的位置信息不对，它就会去访问 table location 层级结构，逐层向上（recursively moves up）。

如果客户端的缓存是空的，位置算法需要三个网络往返（round trip），其中包括一次 Chubby 读取。如果客户端缓存过期了，位置算法需要最多六次网络往返，因为只会在 cache miss 的时候才会检测缓存是否过期（假设 `METADATA` tablets 移动不是非常频繁）。

虽然 tablet 位置放在内存，不需要 GFS 操作，但是，我们可以通过**客户端预取**（prefetch）的方式继续减少这里的开销：**每次从 `METADATA` table 读取的时候，都读取多个 tablet 的元数据**。

另外，我们还在 `METADATA` table 中存储了其他一些次要信息，包括每个 tablet 上的事件的日志（例如使用这个 tablet 的服务是何时启动的），这些信息对 debug 和性能分析很有用。

6.2 Tablet 分配

每个 tablet 每次只会分配给一个 tablet server。

master 会跟踪活着的 tablet server 以及当前 tablet 和 tablet server 的分配关系，其中包括哪些 tablet 是还没有被分配出去的。当一个 tablet 还没有分配出去，并且找到了一个有空闲资源的 tablet server，master 就会向这个 server **发送一个 tablet 加载请求**（load request），将这个 tablet 分配给它。

Bigtable **使用 Chubby 跟踪 tablet servers**。当一个 tablet server 启动后，它会**在特定的 Chubby 目录下创建和获取一个名字唯一的独占锁**（exclusive lock）。master 通过**监听这个目录**（the *servers directory*）来**发现 tablet servers**。

如果一个 tablet server **失去了这个独占锁**，例如由于网络分裂导致 Chubby session断了，那这个 server 会**停止服务这个 tablet**。（Chubby 提供了一种高效机制使得 tablet server 无需产生网络流量就可以判断它自己是否还拥有锁）。

tablet server 失去锁之后，如果锁文件还在，它会尝试重新去获取这个锁；如果锁文件不在了，tablet server 会**自杀（kill itself）**，因为它无法为这个 tablet 提供服务了。

tablet server 终止时（例如，由于集群管理系统将 tablet server 所在的机器移除集群）会将它持有的锁释放，这样 master 就可以及时将对应的 tablets 分配给其他 tablet server。

master 负责检测 tablet server 是否工作正常，以及及时重新分配 tablets。

为了检测 tablet server 是否正常工作，master 会定期地询问每个 tablet server 的锁的状态。如果一个 server 汇报说锁丢失了，或者如果 master 连续 N 次无法连接到这个 server，master 就会**尝试亲自去获取这个锁文件**。如果获取锁成功，说明 Chubby 是活着的，那 master 就可以确定：要么是 tablet server 挂了，要么是它无法连接到 Chubby，然后 master 就会删掉这个锁文件，以保证这个 tablet server 不会再为这个 tablet 提供服务。删除后，master 就将原来分配给这个 tablet server的 tablets 标记为未分配的（unassigned）。

为了保证 Bigtable 不受 master 和 Chubby 之间的网络问题的影响，master 会在它的 Chubby session 过期时自杀。但如前面所描述的，**master 挂掉不会影响 tablets 的分配**。

6.2.1 master 启动流程

当一个 master 被集群管理系统启动后，它必须先查看当前的 tablet 分配情况，然后才能去修改。

master 启动后所做的事情如下：

1. 从 Chubby 获取一个唯一的 `master` 锁，这样为了避免并发的 master 实例化（instantiation）
2. 扫描 Chubby 中的 `servers` 目录，查看当前有哪些活着的 server
3. 和每个活着的 tablet server 通信，查看（discover）当前分别给这些 tablet server 分配了哪些 tablets
4. 扫描 `METADATA` table，查看当前有哪些 tablets（全部 tablets 都在这里）；扫描过程中发现的还未被分配出去的 tablets，会添加到一个未分配 tables 集合，后面就可以被重新分配出去

6.2.2 难点

以上过程的一个难点是：**在扫描 `METADATA` table 之前，必须保证 `METADATA` tablets 自己已经被分配出去了**。

> One complication is that the scan of the METADATA table cannot happen until the METADATA tablets have been assigned.

因此，如果在步骤 3 中发现 root tablet 还没有被分配出去，那 master 就要先将它放到未分配 tablets 集合，然后去执行步骤 4。这样就保证了 root tablet 将会被分配出去。

6.2.3 tablet 分裂和分裂后的新 tablet 发现

因为 root tablet 包含了所有 `METADATA` tablet 的名字，因此 master 扫描 root tablet之后就知道了当前有哪些 tablets。

只有在发生以下情况时，当前的 tablets 集合才会有变化：

1. 创建或删除一个 table
2. 两个 tablets 合并成一个更大的，或者一个 tablet 分裂成两个小的

master 能够跟踪这些变化，因为除了 tablet 分裂之外，其他流程都是由 master 处理的。**tablet 分裂**比较特殊，因为它是**由 tablet server 发起**的。

tablet server 将新的 tablet 信息记录到 `METADATA` table，然后提交这次分裂。提交后，master 会收到通知。如果通知丢失（由于 tablet server 或 master 挂掉），master会在它下次要求一个 tablet server 加载 tablets 时发现。这个 tablet server 会将这次分裂信息通知给 master，因为它在 `METADATA` table 中发现的 tablets 项只覆盖 master 要求它加载的 tablets 的了一部分。

6.3 为 Tablet 提供服务（Tablet Serving）

Tablet 的持久状态存储在 GFS 中，如图 5 所示。

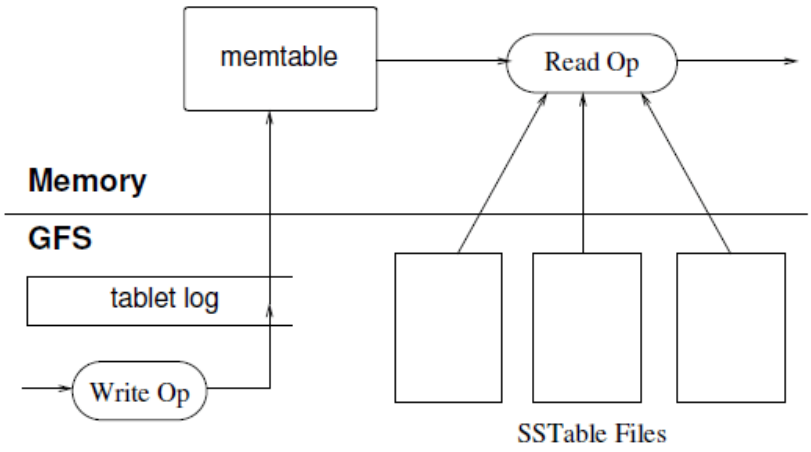


图 5 Reading from Bigtable

更新（update）会提交到一个 commit log 文件，其中保存了 redo 记录。最近的几次更新会存储在**内存中**一个称为 `sstable` 的有序缓冲区（sorted buffer）中；其他老一些的更新存储在 SSTable 中。

6.3.1 Tablet 恢复

恢复一个 tablet 时，tablet server 需要从 `METADATA` table 读取它的元数据。

这里的元数据包括：

1. 组成这个 tablet 的 SSTable 列表
2. 一系列 redo 点，指向 commit log 中 tablet 的数据

tablet server **将 SSTable 索引读到内存，然后应用 redo 点之后提交的所有更新**，就可以重建 memtable。

6.3.2 写操作

当一个写操作到达 tablet server 时，它会检查写操作是否格式正确（well-formed），以及发送者是否有权限执行这次操作。

鉴权的实现方式是：**从 Chubby 文件读取允许的写者列表**（writer list）（在绝大多数情况下，这次读都会命中 Chubby 客户端的缓存）。

一次合法的写操作会记录到 commit log。为了提高小文件写入的吞吐，我们使用了**批量提交**（group commit）技术 [13, 16]。**写操作被提交后，它的内容（数据）就会/才会插入到 memtable**。

6.3.3 读操作

一次读操作到达 tablet server 时，也会执行类似的格式检查和鉴权。

合法的读操作是在 SSTable 和 memtable 的合并视图上进行的（executed on a merged view of the sequence of SSTables and the memtable）。由于 SSTable 和 memtable 都是按词典顺序排序的，因此合并视图的创建很高效。

在 tablet 分裂或合并时，读或写操作仍然是可以进行的。

6.4 压缩（Compactions）

- minor compaction
- major compaction

随着写操作的增多，memtable 在不断变大。**memtable 超过一定大小时会被冻结**（frozen），然后创建一个新的 memtable 来接受写入，冻结的 memtable 会**转化成 SSTable 写入 GFS**，这称为 **minor compaction**。

minor compaction 有两个目的：

1. 减少 tablet server 占用的内存
2. tablet server 挂掉之后恢复时，减少从 commit log 读取的数据量

在 compaction 的过程中，读和写操作是可以正常进行的。

每次 minor compaction 都会创建一个新 SSTable，如果不加额外处理，后面的读操作可能就需要将多个 SSTable 进行合并才能读到需要的内容。

因此，我们在后台定期地执行一个 **merge compaction**，这样就可以保证文件（SSTable）数量保持在一个范围内。合并压缩读取若干个 SSTable 和 `memtable` 的内容，然后写到一个新的 SSTable。写入完成后，原来的 SSTable 和 memtable 的内容就可以删掉了。这种将多个 SSTable 重写成一个的 merge compaction 就称为 **major compaction**。

非 major compaction 产生的 SSTable 会包含特殊的删除信息（deletion entries），用于标记其中已经被删除的数据 —— 实际上这些数据还没有被真正删除，只是标记为已删除。而 **major compaction 产生的 SSTable 不会包含这些删除信息或者已删除的数据**（deletion information or deleted data）。

Bigtable 定期地遍历所有 tablets，执行 major compaction 操作。这使得 Bigtable 可以**及时回收已（被标记为）删除的数据占用的资源**，而且可以**保证已（被标记为）删除的数据及时从系统中消失**，这对于存储敏感数据的服务来说是很重要的。

7 改进（Refinements）

以上描述的实现需要一些改进才能满足我们的用户所需的高性能、可用性和可靠性。

本节将更深入地介绍几个实现部分，以此来展示这些需求。

7.1 Locality groups

客户端可以将多个 column family 组织到一个 locality group。每个 tablet 会**为每个 locality group 生成一个单独的 SSTable**。

将一般**不会一起访问的 column family 划分到不同的 locality group 会提升读性能**。例如，Webtable 中的页面元数据（例如语言和校验和）可以放到同一个 locality group，而将页面内容放到另一个 locality group：应用读取元数据的时候就不需要再读取整个页面内容。

此外，还可以**基于 locality group 维度对某些参数进行调优**。例如，可以声明一个 locality group 是**驻留内存的**（in-memory）。驻留内存的 locality group 对应的 SSTable 会被惰性加载到 tablet server 的内存。 一旦加载，这类 column family 的读操作就不再需要访问磁盘。这个特性对访问频繁的小文件非常有用：`METADATA` table 的 `location` column family 内部用的就是这种类型。

7.2 压缩（Compression）

客户端可以控制 SSTable 是否需要压缩，以及用什么格式压缩。

7.2.1 压缩的粒度和算法

压缩的**基本单位是 SSTable block**（大小可以由 locality group 的参数控制）。虽然 block 级别的压缩（相对于更大的数据级别）损失了一些压缩效率，但在只需读取部分内容时，我们不需要解压整个文件，从而**提高了读效率**。

我们的很多客户端都使用一种自定义的双通（two-pass）压缩算法：

1. 先使用 Bentley-McIlroy 算法 [6] 压缩大窗口内的长公共前缀（long common strings across a large window）
2. 再使用一个快速算法压缩 16KB 窗口内的重复字符串

在现代计算机上，这两个算法都非常快，**压缩速度可以达到 100~200 MB/s，解压可以达到 400~1000 MB/s**。

7.2.2 压缩的速度和效率

虽然相比于压缩效率我们更看重压缩速度，但令人惊奇的是，我们的双通压缩算法效率非常好。

例如，在 Webtable 中，我们存储了大量的页面进行了一次实验。实验中每个页面只存储了一个版本。结果显示，这个算法的压缩比达到了 10:1，而典型情况下 Gzip 压缩 HTML 页面只有 3:1 或 4:1 的效率。

这么高的压缩效率来自 Webtable 的行（row）组织方式：来自相同域名（host）的页面都存储在一起。这些页面有着很多类似内容（模板），非常适合 Bentley-McIlroy 算法。不止是 Webtable，很多应用都根据行名（row names）将相似的数据组织到一起进行存储，因此可以取得非常好的压缩比。如果数据是存储了多个版本而不是一个版本，那压缩比会更高。

7.3 读缓存

为了提高读性能，tablet server 使用了两级缓存：

- Scan Cache
  - 高层缓存
  - 存储 SSTable 返回给 tablet server 的 **key-value pair**

  - 适用于**频繁访问相同数据**的应用

- Block Cache
  - 低层缓存
  - 存储从 GFS 读取的 **SSTable blocks**
  - 适用于**连续访问相邻（相近）数据**的应用。例如顺序读，或者在热点行（hot row）中相同 locality group 内不同列的随机读

7.4 Bloom 过滤器

5.3 介绍过，一次读操作必须要对组成一个 tablet 状态的所有 SSTable 都进行读取。如果这些 SSTable 没有在内存，我们就要进行多次磁盘访问。我们允许客户端在一个特殊的locality group 内指定要**对 SSTable 创建 Bloom 过滤器** [7]，通过这种方式就可以减少这种磁盘访问。

Bloom 过滤器可以判断一个 SSTable 是否包含指定行/列对（row/column pair）对应的数据。对于特定的应用来说，给 tablet server 增加少量内存用于存储 Bloom 过滤器，就可以**极大地减少读操作的磁盘访问**。

我们的实际使用也显示，大部分对不存在的行或列的访问都无需涉及磁盘操作（在 Bloom过滤器这一层就判断不存在了，无需再查找磁盘）。

7.5 Commit-log 实现

7.5.1 每个 tablet 还是每个 tablet server 一个 log 文件

如果为每个 tablet 维护一个单独的 log 文件，那会导致底层 GFS 大量文件的并发写。考虑到 GFS 的具体实现，这些并发写进而会导致大量的磁盘访问，以完成不同物理文件的并发写入。另外，每个 tablet 一个 log 文件的设计还会降低组提交（group commit，批量提交）优化的有效性，因为每个组（group）都会很小。

因此，为了克服以上问题，我们为**每个 tablet server 维护一个 commit log**，将属于这个 tablet server 的不同的 tablet 操作都写入这同一个物理上的 log 文件 [18, 20]。

7.5.2 恢复过程变复杂

这种方式使得常规操作（normal operations）的性能得到了很大提升，但是，它使 tablet恢复过程变得复杂。

当一个 tablet server 挂掉后，它负责的那些 tablets 就会重新分配给其他（大量）的 tablet servers：通常情况下每个 tablet server 只会分到其中的一小部分。恢复一个 tablet 的状态时，新的 tablet server 需要从原 tablet server 的 commit log 里重新应用（reapply）这个 tablet 的修改（mutation）。然而，**这些 tablet 的 mutation 都混在同一个物理的 log 文件内**。

一种方式是每个新的 tablet server 都去读完整的 commit log，将自己需要的部分过滤出来。但是，如果有 100 个机器分到了 tablet 的话，这个 log 文件就要被读 100 次。

7.5.3 优化：两个写线程和两份 commit log

为了避免这种重复读，我们**将 commit log 内容**以 `(table; row name; log sequence number)` 为键（key）**进行排序。在排序后的 commit log 中，每个 tablet 的所有mutation 都是连续的**，因此可以实现高效的读取：**只需一次磁盘寻址加随后的顺序读**。为了加速排序过程，我们还将 commit log 分割成 64 MB 的段（segment），分散到多个 tablet server 上并发地进行排序。

这个排序过程是由 **master 协调（coordinate）、tablet server 触发**的：tablet server 向 master 汇报说需要从一些 commit log 中恢复一些 mutation。

写提交记录到 GFS 有时会遇到性能卡顿，这可能有多方面原因。例如，负责写操作的 GFS server 挂了，或者到三个指定的 GFS master 的网络发生了拥塞或过载。为了减少这些 GFS 导致的延迟抖动，**每个 tablet server 为 commit log 使用了两个写线程**：每个线程写到各自的 log 文件，但同时只会有一个线程是活跃的。如果当前的活跃线程写性能非常差，写操作就会切换到另一个线程，由这个新线程负责之后的写。

log 中的记录（entry）都有序列号，恢复的时候可以根据序列号过滤由于 log 切换导致的重复数据。

7.6 加速 tablet 恢复过程

如果 master 将一个 tablet 从一个 tablet server 移动到另一个，源 tablet server 会先对这个 tablet 进行一次 minor compaction。这会对 commit log 里还未压缩的状态进行一次压缩，减少恢复时需要读取的数据量。这次压缩完成后，源 tablet server 停止为这个 tablet 提供服务。

源 tablet server 在真正卸载（unload）这个 tablet 之前会再进行一次（通常非常快的）minor compaction，对第一次 minor compaction 到当前时刻内新进来的未压缩状态进行压缩。这次压缩做完之后，这个 tablet 就可以被其他的 tablet server 加载（load），而无需恢复任何 log 记录。

7.7 利用不可变性（Exploiting immutability）

除了 SSTable 缓存之外，Bigtable 系统其他一些部分也因 SSTable 的不可变性而得到简化。例如，从 SSTable 读取数据时，对文件系统的访问不需要任何同步。因此，对行的并发控制可以实现地非常高效。

读和写操作涉及的唯一可变数据结构是 memtable。为减少 memtable 的读竞争，我们将 memtable 的行（row）设计为**写时复制**（copy-on-write），这样读和写就可以并行进行。

因为 SSTable 是不可变的，所以**彻底删除数据**（permanently removing deleted data）的问题就变成了**对过期的 SSTable 进行垃圾回收**（garbage collecting obsolete SSTables）。

每个 tablet 的 SSTable 会注册到 `METADATA` table。master 会对过期的 SSTable 进行**“先标记后清除”**（mark-and-sweep）[25]，其中 `METADATA` table 记录了这些 SSTable 的对应的 tablet 的 root。

最后，**SSTable 的不可变性使得 tablet 分裂过程更快**。我们直接让子 tablet 共享父 tablet 的 SSTable，而不是为每个子 tablet 分别创建一个新的 SSTable。

8 性能评估

8.1 测试环境

我们在一套有 N 个 tablet server 的 Bigtable 集群进行测试，测量 N 变化时 Bigtable的性能和可扩展性。

每个 tablet server 使用 1GB 内存，写到由 1786 台节点组成的 GFS 集群，其中每个节点配备了两个 400GB 的 IDE 硬盘。

N 个客户端生成 Bigtable 负载用于测试（用和 tablet server 同样数量的客户端是为了保证客户端不会称为性能瓶颈）。

每个机器有两个双核 Opteron 2 GHz 处理器，足够的物理内存，以及一个 1Gbps 以太网链路。所有机器连接到一个**两级树状交换网络**（two-level tree-shaped switched network），网络根节点有 100-200 Gbps 的聚合带宽。所有机器都在同一个物理基础设施中，因此机器间的时延小于 1ms。

tablet server、master、测试用的客户端，以及 GFS server 都运行在相同的一组机器上。本实验是在一个正常使用中的集群上进行的，因此：

1. 每个机器都运行了一个 GFS server
2. 有的机器运行了一个 tablet server，或者一个客户端进程，或者其他与本实验无关的工作任务

8.2 性能指标

`R` 是测试中 Bigtable 的不重复行键（row key）数量。`R` 的选择使得每个基准测试中每个 tablet server 读或写大约 1GB 数据。

`sequential write`（顺序写）将行空间等分成 10N 份，通过一个中心调度器分配给 N 个客户端，每个客户端都是先拿到一份进行处理，完成后调度器会再分给它一份，这种动态分配可以减少客户端所在机器上的其他进程对实验的影响。每一个行键对应写一个字符串，字符串是随机生产的，因此无法压缩（uncompressible）。另外，不同行键对应的字符串是不同的，因此也是无法跨行压缩的。

`random write`（随机写）基准测试与顺序写类似，除了行键在写之前是对 `R` 取模的（row key was hashed modulo R），因此写操作可以在整个测试期间都均匀地分散到整个行空间。

`sequential read`（顺序读）生产行键的方式与顺序写类似，读的也是顺序写测试写入的数据。

`random read`（随机读）与随机写类似。

`scan`（扫描）和顺序读类似，但利用了 Bigtable 提供的**扫描给定行范围内的所有值**的 API。使用这个 API 可以减少 RPC 的次数，因为一次 RPC 就可以从 tablet server 取到大量的值。

`random read (mem)` 和顺序读类似，但测试数据的 locality group 标记为驻留内存型（in-memory），因此会从 tablet server 的内存而不是 GFS 读取。在这个测试中，我们将每个 tablet 的测试数据从 1GB 降到了 100MB，以充分保证它们能落到 tablet server 的内存中。

图 6 以两种视图展示了读/写 1000 字节的值到 Bigtable 时的性能。左侧是每个 tablet server 每秒的操作数；右侧是聚合之后的每秒操作数。

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

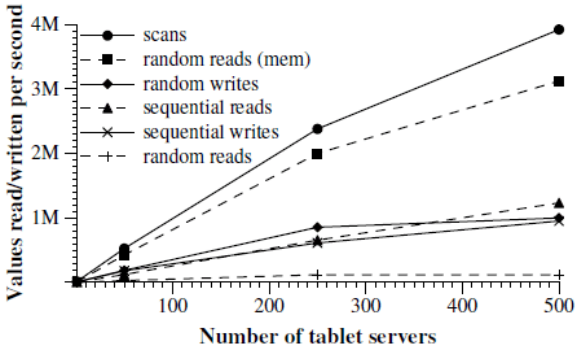


图 6 读/写 1000 字节的值到 Bigtable 时的性能

8.3 单 tablet-server 性能

首先看单个 tablet server 的性能。

随机读比其他的操作都要慢一个数量级甚至更多。

每次随机读都需要将 64KB 的 SSTable block 从 GFS 通过网络传输到 tablet server，而其中仅仅包含了一个 1000 字节的值。tablet server 每秒大约 1200 次读操作，折算约为 `75 MB/s` 从 GFS 读数据。考虑到网络栈、SSTable 解析、Bigtable 代码等开销，这个带宽足以使 tablet server 的 CPU 达到饱和了，也足以使机器的网络链路饱和了（75 MB/s = 600 Mbps，系统总共 1Gbps 带宽）。大部分这种访问类型的 Bigtable 应用会**将 block size 设置的更小**，一般设为 8 KB。

从内存的随机读会快很多，因为每个 1000 字节的读都是从 tablet server 的本地内存读取的，不需要从 GFS 访问 64KB 的 block。

随机和顺序写的性能都要比随机读好，因为每个 tablet server 会将所有写操作追加到同一个 commit log 然后执行**批量提交**（group commit），从而高效地写入到 GFS。**随机写和顺序写的性能并没有明显差异**，因为两种情况下，所有到 tablet server 的写最后都是到了同一个 commit log。

顺序读的性能远好于随机读，因为每个从 GFS **预取**（prefetch）的 64KB SSTable block 都存储到了 blcok 缓存，下一次 64 读请求就会用到。

扫描的性能更好，因为客户端的一次 RPC 请求就可以从 tablet server 拿到大量的值，因此 RPC 开销被平摊了。

8.4 扩展性（scaling）

当我们将系统中 tablet server 的数量从 1 增加到 500 时，聚合吞吐量（aggregate throughput）的增长非常明显，超过了 100 倍。例如，当 tablet server 数量增加到 500 倍时，`random read (mem)` 增长了几乎 300 倍。这是因为这个基准测试的性能瓶颈在 tablet server 的 CPU。

但是，**性能并没有线性增长**。对于大部分基准测试，在 tablet server 从 1 增加到 500 的过程中，单台 server 的吞吐量都有一个明显的下降（图 6 左边的表）。这个下降是**由不同 server 配置导致的负载不均衡引起的，大部分情况下是由于机器上的其他进程在竞争 CPU 和网络资源**。

我们的负载均衡算法就是想解决这个问题，但由于两个主要原因无法做到完美：

1. 减少 tablet 的移动会引起 rebalancing 的抖动（tablet 在移动的时候会有很短的一段时间不可用，一般在 1 秒以下）
2. 基准测试生成的负载会随着测试的进行而不断漂移（shifts around）

随机读基准测试的扩展性最差（server 增加 500 倍时，它的聚合吞吐量只增加了 100 倍）。前面解释过，造成这个问题的原因是对于每个 1000 字节的值，我们都需要通过网络传输一个 64KB 的 block。这个数据量使得我们与其他进程共享的 1Gbps 网络带宽达到饱和，因此随着机器数量的增加，每节点平均吞吐量（per-server throughput）下降非常明显。

9 真实应用

截至 2006 年 8 月，Google 总共运行着 388 个非测试的 Bigtable 集群，分布在不同的数据中心，加起来有 24,500 个 tablet server。

表 1 展示了这些集群中 tablet server 数量的大致分布：

表 1 Bigtable 集群中 tablet server 数量分布

# of tablet servers	# of clusters
0 .. 19	259
20 .. 49	47
50 .. 99	20
100 .. 499	50
> 500	12

其中一些集群是用于开发目的，因此会有较长时间的空闲状态。

我们挑选了 14 个活跃集群，总共包含 8069 个 tablet server，提供了如下聚合性能：

- 120 万次请求/秒（QPS）
- 741 MB/s RPC 入流量
- 16 GB/s RPC 出流量

图 2 给出了目前在用的几个 table 的一些数据。

表 2 生产环境 Bigtable 的一些数据

Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency-sensitive?
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
Google Earth	0.5	64%	8	7	2	33%	Yes
Google Earth	70	–	9	8	3	0%	No
Orkut	9	–	0.9	8	5	1%	Yes
Personalized Search	4	47%	6	93	11	5%	Yes

一些 table 存储的是给用户使用的数据，另外一些存储的是批处理用的数据。table 的大小、平均 cell 大小、内存中数据（served from memory）所占的比例、table schema的复杂度等等差异都很大。在本节接下来的内容中，我们将简要介绍产品团队是如何使用 Bigtable 的。

9.1 Google Analytics

Google Analytics (analytics.google.com) 是一个帮助网站管理员分析网站流量的服务。

它提供了很多聚合统计数据，例如每天的独立访问量和每个 URL 每天的访问量，以及网站跟踪报告，例如给定一组之前浏览了某个页面的用户，它可以给出实际发生了购买行为的用户比例。

为了实现这些功能，网络管理员需要在他们的网页上嵌入一段 JavaScript 代码。这样每当这个网页被访问时，这段程序就会被激活。它会记录很多的信息，例如用户 ID 以及页面信息，发送给 Google Analytics，Google Analytics 会对这些信息进行汇总，最后呈现给网站管理员。

这里简要介绍 Google Analytics 使用的两个 table。

原始点击（raw click）table（~200 TB）为每个用户维护了一个（数据）行。行名是网站名和 session 创建时间组成的一个元组（tuple）。这样的 schema 保证了访问网站的 session 按照时间顺序（chronologically）是连续的。这个 table 压缩到了原始大小的14%。

汇总（summary）table（~20TB）存储了每个网站的一些预定义的汇总。这个 table 是通过定期的 MapReduce 任务对原始点击表进行计算得到的。每个 MapReduce 任务会从原始点击表中提取最近的 session 数据，系统整体的吞吐受限于 GFS 的吞吐。这个表压缩到了原始大小的 29%。

9.2 Google Earth

Google 提供了地球高精度卫星图服务给用户，可以通过基于网页的 Google Maps 接口（maps.google.com）或客户端软件 Google Earth（earth.google.com）访问。这些产品允许用户在任何分辨率的卫星图上游走，停留、查看和标注。

这个系统使用了一个表来做数据预处理，另外很多表来服务客户端数据。预处理 pipeline使用一个表来存储原始图像。预处理过程会将图像进行清洗和合并（clean and consolidate），变成可以提供服务的数据。这个表存储了大约 70 TB 的数据，因此是放在磁盘上的。另外这些图像都已经高效地压缩过了，因此 Bigtable 的压缩是关闭的。表的每一行代表一个 geographic segment（地理位置）。行名的设计使得地理上相邻的 segment 在存储的时候也是相邻的。另外，这个表还包含一个 column family，用来跟踪每个 segment 的数据来源（sources of data for each segment）。这个 column family 有大量的列：基本上每个原始数据图像（raw data image）都有一列。因为每个 segment 都是用少量几张图像合成的，因此这个 column family 非常稀疏。

预处理 pipeline 强烈依赖 MapReduce 对 Bigtable 内的数据进行变换。部分 MapReduce job 进行时，系统整体可以达到每个 tablet server 1MB/s 以上的数据处理速度。

服务系统使用一个表来索引存储在 GFS 中的数据。这个表相对比较小（~500GB），但它必须保证每个数据中心每秒几万次请求（QPS）的负载下，仍然保持很低的延迟。因此，这个表同时分散到了几百个 tablet server 上进行处理，并且还包含了驻留内存的 column family。

9.3 Personalized Search

Personalized Search（个性化搜索）(www.google.com/psearch)是一个自选的服务，它会记录用户的搜索关键词和在各种 Google 服务上的点击，例如网页搜索、图像和新闻等等。用户可以通过浏览自己的搜索关键词和点击记录来查看他们的搜索历史，可以要求根据自己过去的 Google 使用习惯来向他们提供个性化搜索结果。

个性化搜索将用户数据存储到 Bigtable。每个用户有一个唯一的用户 ID，并根据这个 ID分配一个行名。所有的用户动作存储在另一个表，每种类型的动作会占用一个 column family（例如，有一个 column family 存储所有的网页查询）。每个数据用动作发生的时刻作为它在 Bigtable 中的时间戳。

个性化搜索利用 MapReduce 在 Bigtable 上进行运算，为每个用户生成一个 profile。这些 profile 就会用来做个性化的实时搜索。

个性化搜索的数据会在几个 Bigtable 之间做复制，以提高可用性，减少客户端距离导致的延迟。这个团队最初在 Bigtable 之上开发了自己的一套客户端侧复制机制，以保证所有副本的最终一致性。现在，复制子系统已经集成到服务端。

个性化搜索存储系统的设计允许其他团队在他们各自的列中添加用户级别的（per-user）信息，这个系统现在被很多 Google 其他产品在使用，存储他们自己的用户级别的（per-user）配置选项和设置。但在多个开发团队之间共享一个表会导致数量异常庞大的 column family。

为了帮助共享，我们给 Bigtable 添加了一个简单的配额（quota）机制，限制单一客户端在一个共享表中所占的存储大小。对于那些多个产品团队使用 Bigtable 存储用户级别信息的场景，这种机制提供了一定的隔离性。

10 从中所学（Lessons）

在设计、实现、维护和支持 Bigtable 的过程中，我们得到了很多有用的经验，也学习到了很多有趣的教训。

10.1 故障源远比你想象中多

首先我们认识到，大型分布式系统在很多方面的故障面前都很脆弱，不仅仅是很多分布式协议所假设的网络分裂和出错后停止服务（fail-stop failures）。例如，我们就遇到过如下场景引起的故障：

- 内存和网络损坏
- 很大的时钟偏差（clock skew）
- 机器死机（hung）
- 更复杂的和非对称的网络分裂
- 依赖的基础服务的 bug（例如 Chubby）
- GFS 配额溢出（overflow）
- 计划及非计划的硬件维护

随着对这一问题的了解的深入，我们开始修改各种的协议来应对这一问题。例如，我们给 RPC 机制添加了校验和。

另外，我们还去掉了系统的一个部分对另一部分的假设。例如，我们不再假设一次 Chubby 操作只会返回固定的几种错误。

10.2 避免过早添加使用场景不明确的新特性

我们得到的另一重要经验是：如果还不是非常清楚一个新特性将被如何使用，那就不要着急添加到系统中。

例如，我们最初有计划在 API 中支持广义事物模型（general-purpose transaction）。但因为当时没有迫切的使用场景，因此没有立即去实现。现在有了很多真实应用跑在 Bigtable 之后，我们审视了这些应用的真实需求，发现大部分应用其实只需要单行事务（single-row transaction）。

对于真的有分布式事务需求的人，我们发现他们最核心的需求其实是维护二级索引（secondary indices），因此我们计划通过添加一个特殊的机制来满足这个需求。这个机制没有分布式事务通用，但性能会更好（尤其是跨上百行以上的更新），而且对于乐观跨数据中心复制（optimistic cross-data-center replication）来说，和我们系统的集成会更好。

10.3 系统级监控非常重要

在日常支持 Bigtable 中学到的实际一课是：合理的系统级监控（例如监控 Bigtable 本身，以及使用 Bigtable 的客户端）非常重要。

例如，我们扩展了我们的 RPC 系统，可以记录重要动作的详细跟踪信息。这个特性帮助我们检测和解决了很多问题，包括：

1. tablet 数据结构上的锁竞争
2. 提交 Bigtable mutation 时 GFS 写很慢
3. `METADATA` tablets 不可用时访问 `METADATA` 表时被卡住（stuck）

监控的另一个例子是每个 Bigtable 集群都注册到了 Chubby。这使得我们可以跟踪所有的集群，看到集群有多大，各自运行的是什么版本，接收到的流量有多大，是否有异常的大延迟等等。

10.4 保持设计的简洁

我们学到的最重要经验是：简单设计带来的价值（the value of simple designs）。

考虑到我们的系统规模（10 万行代码，不包括测试），以及代码都会随着时间以难以意料的方式演进，我们发现代码和设计的简洁性对代码的维护和 debug 有着巨大的帮助。

> Given both the size of our system (about 100,000 lines of non-test code), as well as the fact that code evolves over time in unexpected ways, we have found that code and design clarity are of immense help in code maintenance and debugging.

一个例子是我们的 tablet server 成员（membership）协议。我们的第一版非常简单：master 定期向 tablet server 提供租约，如果一个 tablet server 的租约到期，它就自杀。不幸的是，这个协议在发生网络问题时可用性非常差，而且对 master 恢复时间也很敏感。

接下来我们重新设计了好几版这个协议，直到它令我们满意。但是，这时的协议已经变得过于复杂，而且依赖了一些很少被其他应用使用的 Chubby 特性。最后发现我们花了大量的时间来 debug 怪异的边界场景，不仅仅是 Bigtable 代码，还包括 Chubby 代码。

最终，我们放弃了这个版本，重新回到了一个新的更简单的协议，只依赖使用广泛的 Chubby特性。

11 相关工作

The Boxwood project [24] has components that overlap in some ways with Chubby, GFS, and Bigtable, since it provides for distributed agreement, locking, distributed chunk storage, and distributed B-tree storage. In each case where there is overlap, it appears that the Boxwood's component is targeted at a somewhat lower level than the corresponding Google service. The Boxwood project's goal is to provide infrastructure for building higher-level services such as file systems or databases, while the goal of Bigtable is to directly support client applications that wish to store data.

Many recent projects have tackled the problem of providing distributed storage or higher-level services over wide area networks, often at "Internet scale." This includes work on distributed hash tables that began with projects such as CAN [29], Chord [32], Tapestry [37], and Pastry [30]. These systems address concerns that do not arise for Bigtable, such as highly variable bandwidth, untrusted participants, or frequent reconfiguration; decentralized control and Byzantine fault tolerance are not Bigtable goals.

In terms of the distributed data storage model that one might provide to application developers, we believe the key-value pair model provided by distributed B-trees or distributed hash tables is too limiting. Key-value pairs are a useful building block, but they should not be the only building block one provides to developers. The model we chose is richer than simple key-value pairs, and supports sparse semi-structured data. Nonetheless, it is still simple enough that it lends itself to a very ef cient representation, and it is transparent enough (via locality groups) to allow our users to tune important behaviors of the system.

Several database vendors have developed parallel databases that can store large volumes of data. Oracle's Real Application Cluster database [27] uses shared disks to store data (Bigtable uses GFS) and a distributed lock manager (Bigtable uses Chubby). IBM's DB2 Parallel Edition [4] is based on a shared-nothing [33] architecture similar to Bigtable. Each DB2 server is responsible for a subset of the rows in a table which it stores in a local relational database. Both products provide a complete relational model with transactions.

Bigtable locality groups realize similar compression and disk read performance benets observed for other systems that organize data on disk using column-based rather than row-based storage, including C-Store [1, 34] and commercial products such as Sybase IQ [15, 36], SenSage [31], KDB+ [22], and the ColumnBM storage layer in MonetDB/X100 [38]. Another system that does vertical and horizontal data partioning into and achieves good data compression ratios is AT&T's Daytona database [19]. Locality groups do not support CPUcache- level optimizations, such as those described by Ailamaki [2].

The manner in which Bigtable uses memtables and SSTables to store updates to tablets is analogous to the way that the Log-Structured Merge Tree [26] stores updates to index data. In both systems, sorted data is buffered in memory before being written to disk, and reads must merge data from memory and disk.

C-Store and Bigtable share many characteristics: both systems use a shared-nothing architecture and have two different data structures, one for recent writes, and one for storing long-lived data, with a mechanism for moving data from one form to the other. The systems differ significantly in their API: C-Store behaves like a relational database, whereas Bigtable provides a lower level read and write interface and is designed to support many thousands of such operations per second per server. C-Store is also a "read-optimized relational DBMS", whereas Bigtable provides good performance on both read-intensive and write-intensive applications.

Bigtable's load balancer has to solve some of the same kinds of load and memory balancing problems faced by shared-nothing databases (e.g., [11, 35]). Our problem is somewhat simpler: (1) we do not consider the possibility of multiple copies of the same data, possibly in alternate forms due to views or indices; (2) we let the user tell us what data belongs in memory and what data should stay on disk, rather than trying to determine this dynamically; (3) we have no complex queries to execute or optimize.

12 总结

我们在 Google 设计了 Bigtable，一个存储**结构化数据**的分布式系统。

Bigtable 从 2005 年 4 月开始用于生产环境，而在此之前，我们花了大约 **7 个人年**（person-year）的时间在设计和实现上。到 2006 年 8 月，已经有超过 60 个项目在使用 Bigtable。

我们的用户很喜欢 Bigtable 提供的性能和高可用性，当集群面临的负载不断增加时，他们只需简单地向集群添加更多的节点就可以扩展 Bigtable 的容量。

考虑到 Bigtable 的接口不是太常规（unusual），一个有趣的问题就是，我们的用户需要花多长时间去适应 Bigtable。新用户有时不太确定如何使用 Bigtable 最合适，尤其是如果之前已经习惯了关系型数据库提供的广义事务。然后，很多 Google 产品成功地使用了 Bigtable 还是说明了，我们的设计在实际使用中还是非常不错的。

当前我们正在添加一些新的特性，例如支持 secondary indices，以及构建跨数据中心复制的、有多个 master 副本的 Bigtable。我们还在做的是将 Bigtable 作为一个服务提供给各产品组，以后每个组就不需要自己维护他们的集群。随着服务集群的扩展，我们将需要处理更多 Bigtable 内部的资源共享问题 [3, 5]。

最后，我们发现**构建我们自己的存储解决方案可以带来非常大的优势**。为 Bigtable 设计自己的数据模型已经给我们带来非常多的便利性。另外，我们**对 Bigtable 的实现，以及 Bigtable 所依赖的其他 Google 基础设施有足够的控制权**，因此任何一个地方有瓶颈了，我们都可以及时解决。

13 Acknowledgements

We thank the anonymous reviewers, David Nagle, and our shepherd Brad Calder, for their feedback on this paper. The Bigtable system has benefited greatly from the feedback of our many users within Google. In addition, we thank the following people for their contributions to Bigtable: Dan Aguayo, Sameer Ajmani, Zhifeng Chen, Bill Coughran, Mike Epstein, Healfdene Goguen, Robert Griesemer, Jeremy Hylton, Josh Hyman, Alex Khesin, Joanna Kulik, Alberto Lerner, Sherry Listgarten, Mike Maloney, Eduardo Pinheiro, Kathy Polizzi, Frank Yellin, and Arthur Zwiegincew.

14 参考文献

1. ABADI, et al. Integrating compression and execution in columnoriented database systems. SIGMOD 2006
2. AILAMAKI, et al. Weaving relations for cache performance. In The VLDB Journal 2001
3. BANGA, et al. Resource containers: A new facility for resource management in server systems. OSDI 1999
4. BARU, et al. DB2 parallel edition. IBM Systems Journal 1995
5. BAVIER, et al. Operating system support for planetary-scale network services. NSDI 2004
6. BENTLEY, et al. Data compression using long common strings. In Data Compression Conference 1999
7. BLOOM, et al. Space/time trade-offs in hash coding with allowable errors. CACM 1970
8. BURROWS, M. **The Chubby lock service for loosely coupled distributed systems**. OSDI 2006
9. CHANDRA, et al. Paxos made live An engineering perspective. PODC 2007
10. COMER, D. Ubiquitous B-tree. Computing Surveys, 1979
11. COPELAND, et al. Data placement in Bubba. SIGMOD (1988)
12. DEAN, et al. **MapReduce: Simplified data processing on large clusters**. OSDI 2004
13. DEWITT, et al. Implementation techniques for main memory database systems. SIGMOD 1984
14. DEWITT, et al. Parallel database systems: The future of high performance database systems. CACM, 1992
15. FRENCH, C. D. One size fits all database architectures do not work for DSS. SIGMOD 1995
16. GAWLICK, D., AND KINKADE, D. Varieties of concurrency control in IMS/VS fast path. Database Engineering Bulletin, 1985
17. GHEMAWAT, et al. **The Google file system**. SOSP, 2003
18. GRAY, J. Notes on database operating systems. In Operating Systems - An Advanced Course, 1978
19. GREER, R. Daytona and the fourth-generation language Cymbal. SIGMOD, 1999

20. HARTMAN, J. H., AND OUSTERHOUT, J. K. The Zebra striped network file system. SOSP, 1993

21. KX.COM. kx.com/products/database.php. Product page.

22. LAMPORT, L. The part-time parliament. ACM TOCS, 1998

23. MACCORMICK, et al. Boxwood: Abstractions as the foundation for storage infrastructure. OSDI 2004

24. MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine. CACM, 1960

25. O'NEIL, et al. **The log-structured merge-tree (LSM-tree)**. Acta Inf, 1996

26. ORACLE.COM. www.oracle.com/technology/products/- database/clustering/index.html. Product page.

27. PIKE, et al. Interpreting the data: Parallel analysis with Sawzall. Scientific Programming Journal, 2005

28. RATNASAMY, et al. A scalable content-addressable network. SIGCOMM 2001

29. ROWSTRON, et al: Scalable, distributed object location and routing for largescale peer-to-peer systems. Middleware 2001

30. SENSAGE.COM. sensage.com/products-sensage.htm. Product page.

31. STOICA, et al. Chord: A scalable peer-to-peer lookup service for Internet applications. SIGCOMM 2001

32. STONEBRAKER, M. The case for shared nothing. Database Engineering Bulletin, 1986

33. STONEBRAKER, et al. C-Store: A columnoriented DBMS. VLDB 2005

34. STONEBRAKER, et al. Mariposa: A new architecture for distributed data. the Tenth ICDE, 1994

35. SYBASE.COM. www.sybase.com/products/databaseservers/ sybaseiq. Product page.

36. ZHAO, et al. Tapestry: An infrastructure for fault-tolerant wide-area location and routing, 2001

37. ZUKOWSKI, et al. MonetDB/X100 - A DBMS in the CPU cache. IEEE Data Eng, 2005