第07章_InnoDB数据存储结构

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网： http://www.atguigu.com

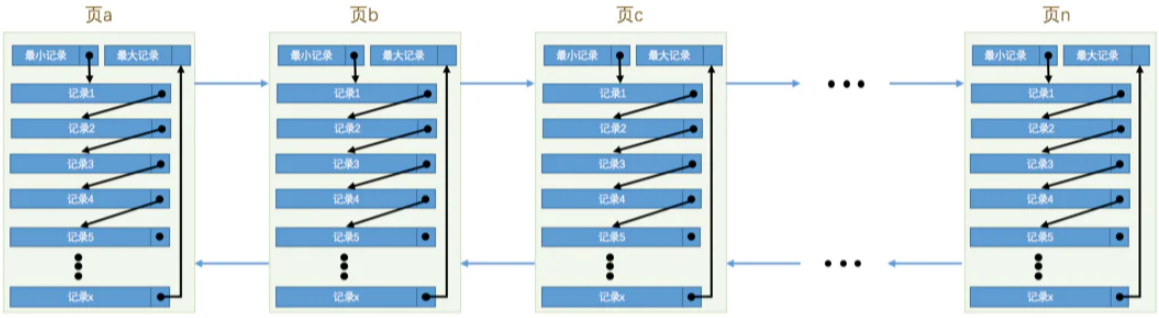
1. 数据库的存储结构：页

1.1 磁盘与内存交互基本单位：页

InnoDB 将数据划分为若干个页，InnoDB中页的大小默认为 **16KB** 。

以 页 作为磁盘和内存之间交互的 基本单位 ，也就是一次最少从磁盘中读取16KB的内容到内存中，一次最少把内存中的16KB内容刷新到磁盘中。也就是说，**在数据库中，不论读一行，还是读多行，都是将这些行所在的页进行加载。也就是说，数据库管理存储空间的基本单位是页（Page），数据库 I/O 操作的最小单位是页。** 一个页中可以存储多个行记录。

> 记录是按照行来存储的，但是数据库的读取并不以行为单位，否则一次读取（也就是一次 I/O 操作）只能处理一行数据，效率会非常低。



1.2 页结构概述

页a、页b、页c ... 页n 这些页可以 不在物理结构上相连 ，只要通过 双向链表 相关联即可。每个数据页中的记录会按照主键值从小到大的顺序组成一个 单向链表 ，每个数据页都会为存储在它里边的记录生成一个 页目录 ，在通过主键查找某条记录的时候可以在页目录中 使用二分法 快速定位到对应的槽，然后再遍历该槽对应分组中的记录即可快速找到指定的记录。

1.3 页的大小

不同的数据库管理系统（简称DBMS ）的页大小不同。比如在 MySQL 的 InnoDB 存储引擎中，默认页的大小是 16KB ，我们可以通过下面的命令来进行查看：

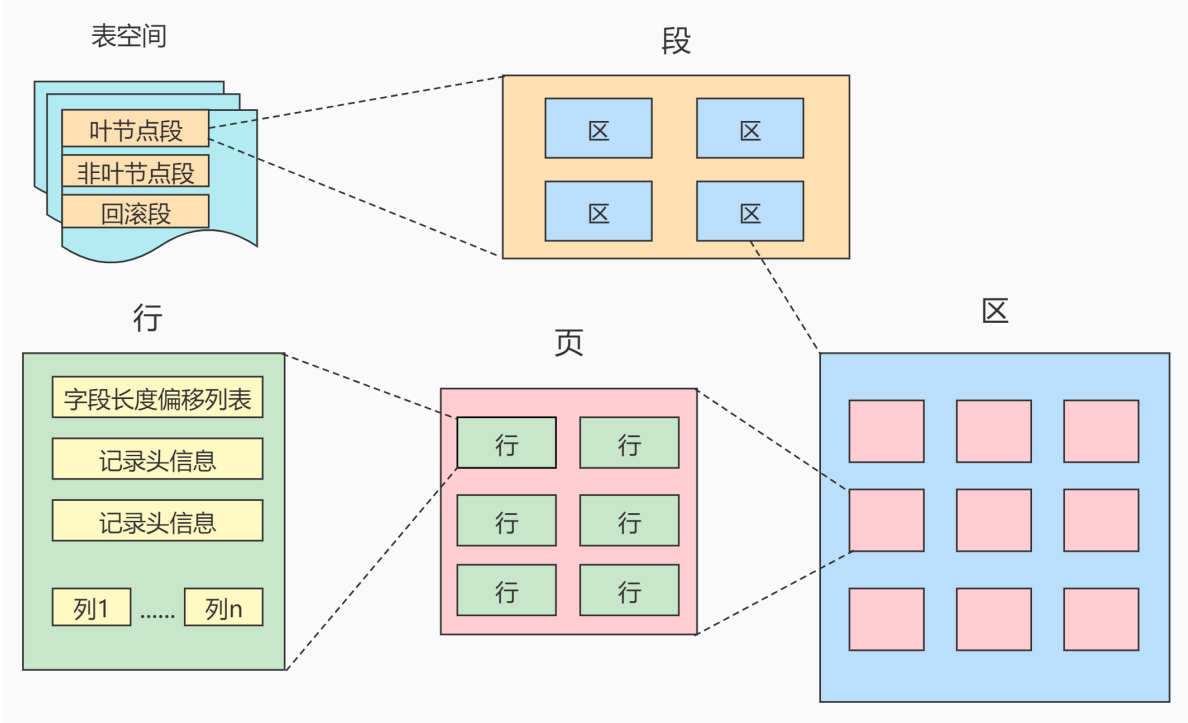
```
mysql> show variables like '%innodb_page_size%';
```

```
mysql> show variables like '%innodb_page_size%';
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| innodb_page_size | 16384 |
+------------------+-------+
1 row in set, 1 warning (0.00 sec)
```

SQL Server 中页的大小为 `8KB`，而在 Oracle 中我们用术语“`块`”（Block）来代表“页”，Oralce 支持的块大小为 2KB，4KB，8KB，16KB，32KB 和 64KB。

1.4 页的上层结构

另外在数据库中，还存在着区（Extent）、段（Segment）和表空间（Tablespace）的概念。行、页、区、段、表空间的关系如下图所示：

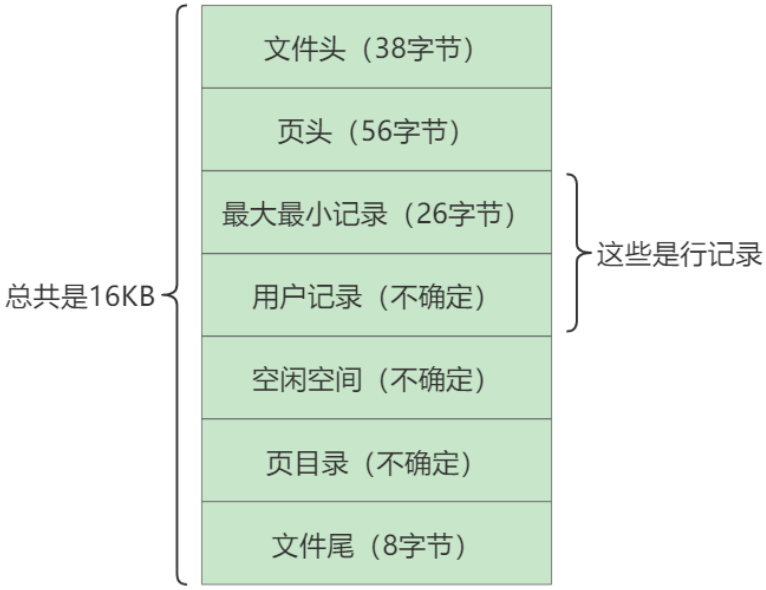


2. 页的内部结构

页如果按类型划分的话，常见的有`数据页（保存 B+ 树节点）`、`系统页`、`Undo 页`和`事务数据页`等。数据页是我们最常使用的页。

数据页的`16KB`大小的存储空间被划分为七个部分，分别是文件头（File Header）、页头（Page Header）、最大最小记录（Infimum+supremum）、用户记录（User Records）、空闲空间（Free Space）、页目录（Page Directory）和文件尾（File Tailer）。

页结构的示意图如下所示：



这 7 个部分作用分别如下，我们简单梳理如下表所示：

名称	占用大小	说明
File Header	38字节	文件头，描述页的信息
Page Header	56字节	页头，页的状态信息
Infimum+Supremum	26字节	最大和最小记录，这是两个虚拟的行记录
User Records	不确定	用户记录，存储行记录内容
Free Space	不确定	空闲记录，页中还没有被使用的空间
Page Directory	不确定	页目录，存储用户记录的相对位置
File Trailer	8字节	文件尾，校验页是否完整

我们可以把这 7 个结构分成 3 个部分。

大家直接参照课件《第07章_InnoDB数据存储结构.mmap》学习即可。

3. InnoDB行格式（或记录格式）

我们平时的数据以行为单位来向表中插入数据，这些记录在磁盘上的存放方式也被称为 `行格式` 或者 `记录格式` 。InnoDB存储引擎设计了4种不同类型的 `行格式` ，分别是 `Compact` 、 `Redundant` 、 `Dynamic` 和 `Compressed` 行格式。查看MySQL8的默认行格式：

```
mysql> SELECT @@innodb_default_row_format;
+-----------------------------+
| @@innodb_default_row_format |
+-----------------------------+
| dynamic                     |
+-----------------------------+
1 row in set (0.00 sec)
```

也可以使用如下语法查看具体表使用的行格式：

```
SHOW TABLE STATUS like '表名'\G
```

大家直接参照课件《第07章_InnoDB数据存储结构.mmap》学习即可。

4. 区、段与碎片区

4.1 为什么要有区?

B+ 树的每一层中的页都会形成一个双向链表，如果是以 页为单位 来分配存储空间的话，双向链表相邻的两个页之间的 物理位置可能离得非常远 。我们介绍B+树索引的适用场景的时候特别提到范围查询只需要定位到最左边的记录和最右边的记录，然后沿着双向链表一直扫描就可以了，而如果链表中相邻的两个页物理位置离得非常远，就是所谓的 随机I/O 。再一次强调，磁盘的速度和内存的速度差了好几个数量级， 随机I/O是非常慢 的，所以我们应该尽量让链表中相邻的页的物理位置也相邻，这样进行范围查询的时候才可以使用所谓的 顺序I/O 。

引入 区 的概念，一个区就是在物理位置上连续的 64个页 。因为 InnoDB 中的页大小默认是 16KB，所以一个区的大小是 64*16KB= 1MB 。在表中 数据量大 的时候，为某个索引分配空间的时候就不再按照页为单位分配了，而是按照 区为单位分配 ，甚至在表中的数据特别多的时候，可以一次性分配多个连续的区。虽然可能造成 一点点空间的浪费 （数据不足以填充满整个区），但是从性能角度看，可以消除很多的随机I/O， 功大于过 ！

4.2 为什么要有段?

对于范围查询，其实是对B+树叶子节点中的记录进行顺序扫描，而如果不区分叶子节点和非叶子节点，统统把节点代表的页面放到申请到的区中的话，进行范围扫描的效果就大打折扣了。所以 InnoDB 对 B+ 树的 叶子节点 和 非叶子节点 进行了区别对待，也就是说叶子节点有自己独有的区，非叶子节点也有自己独有的区。存放叶子节点的区的集合就算是一个 段 （ segment ），存放非叶子节点的区的集合也算是一个段。也就是说一个索引会生成2个段，一个 叶子节点段 ，一个 非叶子节点段 。

除了索引的叶子节点段和非叶子节点段之外，InnoDB中还有为存储一些特殊的数据而定义的段，比如回滚段。所以，常见的段有 数据段 、 索引段 、 回滚段 。数据段即为B+树的叶子节点，索引段即为B+树的非叶子节点。

在InnoDB存储引擎中，对段的管理都是由引擎自身所完成，DBA不能也没有必要对其进行控制。这从一定程度上简化了DBA对于段的管理。

段其实不对应表空间中某一个连续的物理区域，而是一个逻辑上的概念，由若干个零散的页面以及一些完整的区组成。

4.3 为什么要有碎片区?

默认情况下，一个使用InnoDB存储引擎的表只有一个聚簇索引，一个索引会生成2个段，而段是以区为单位申请存储空间的，一个区默认占用1M（64 * 16Kb = 1024Kb）存储空间，所以**默认情况下一个只存了几条记录的小表也需要2M的存储空间么?** 以后每次添加一个索引都要多申请2M的存储空间么？这对于存储记录比较少的表简直是天大的浪费。这个问题的症结在于到现在为止我们介绍的区都是非常 纯粹 的，也就是一个区被整个分配给某一个段，或者说区中的所有页面都是为了存储同一个段的数据而存在的，即使段的数据填不满区中所有的页面，那余下的页面也不能挪作他用。

为了考虑以完整的区为单位分配给某个段对于 数据量较小 的表太浪费存储空间的这种情况，InnoDB提出了一个 碎片（fragment）区 的概念。在一个碎片区中，并不是所有的页都是为了存储同一个段的数据而存在的，而是碎片区中的页可以用于不同的目的，比如有些页用于段A，有些页用于段B，有些页甚至哪个段都不属于。 碎片区直属于表空间 ，并不属于任何一个段。

4.4 区的分类

区大体上可以分为4种类型：

- 空闲的区(FREE)：现在还没有用到这个区中的任何页面。
- 有剩余空间的碎片区(FREE_FRAG)：表示碎片区中还有可用的页面。
- 没有剩余空间的碎片区(FULL_FRAG)：表示碎片区中的所有页面都被使用，没有空闲页面。
- 附属于某个段的区(FSEG)：每一个索引都可以分为叶子节点段和非叶子节点段。

处于FREE、FREE_FRAG以及FULL_FRAG这三种状态的区都是独立的，直属于表空间。而处于FSEG状态的区是附属于某个段的。

> 如果把表空间比作是一个集团军，段就相当于师，区就相当于团。一般的团都是隶属于某个师的，就像是处于FSEG的区全都隶属于某个段，而处于FREE、FREE_FRAG以及FULL_FRAG这三种状态的区却直接隶属于表空间，就像独立团直接听命于军部一样。

5. 表空间

表空间可以看做是InnoDB存储引擎逻辑结构的最高层，所有的数据都存放在表空间中。

表空间是一个逻辑容器，表空间存储的对象是段，在一个表空间中可以有一个或多个段，但是一个段只能属于一个表空间。表空间数据库由一个或多个表空间组成，表空间从管理上可以划分为系统表空间（System tablespace）、独立表空间（File-per-table tablespace）、撤销表空间（Undo Tablespace）和临时表空间（Temporary Tablespace）等。

5.1 独立表空间

独立表空间，即每张表有一个独立的表空间，也就是数据和索引信息都会保存在自己的表空间中。独立的表空间（即：单表）可以在不同的数据库之间进行迁移。

空间可以回收（DROP TABLE 操作可自动回收表空间；其他情况，表空间不能自己回收）。如果对于统计分析或是日志表，删除大量数据后可以通过：`alter table TableName engine=innodb;` 回收不用的空间。对于使用独立表空间的表，不管怎么删除，表空间的碎片不会太严重的影响性能，而且还有机会处理。

独立表空间结构

独立表空间由段、区、页组成。前面已经讲解过了。

真实表空间对应的文件大小

我们到数据目录里看，会发现一个新建的表对应的.ibd文件只占用了96K，才6个页面大小（MySQL5.7中），这是因为一开始表空间占用的空间很小，因为表里边都没有数据。不过别忘了这些.ibd文件是自扩展的，随着表中数据的增多，表空间对应的文件也逐渐增大。

查看 InnoDB 的表空间类型：

```
mysql > show variables like 'innodb_file_per_table';
```

```
mysql> show variables like 'innodb_file_per_table';
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| innodb_file_per_table | ON    |
+-----------------------+-------+
1 row in set, 1 warning (0.01 sec)
```

你能看到 innodb_file_per_table=ON，这就意味着每张表都会单独保存为一个 `.ibd` 文件。

5.2 系统表空间

系统表空间的结构和独立表空间基本类似，只不过由于整个MySQL进程只有一个系统表空间，在系统表空间中会额外记录一些有关整个系统信息的页面，这部分是独立表空间中没有的。

InnoDB数据字典

每当我们向一个表中插入一条记录的时候， `MySQL校验过程` 如下：

先要校验一下插入语句对应的表存不存在，插入的列和表中的列是否符合，如果语法没有问题的话，还需要知道该表的聚簇索引和所有二级索引对应的根页面是哪个表空间的哪个页面，然后把记录插入对应索引的B+树中。所以说，MySQL除了保存着我们插入的用户数据之外，还需要保存许多额外的信息，比方说：

- 某个表属于哪个表空间，表里边有多少列
- 表对应的每一个列的类型是什么
- 该表有多少索引，每个索引对应哪几个字段，该索引对应的根页面在哪个表空间的哪个页面
- 该表有哪些外键，外键对应哪个表的哪些列
- 某个表空间对应文件系统上文件路径是什么
- ...

上述这些数据并不是我们使用 `INSERT` 语句插入的用户数据，实际上是为了更好的管理我们这些用户数据而不得已引入的一些额外数据，这些数据也称为 `元数据` 。InnoDB存储引擎特意定义了一些列的 `内部系统表` （internal system table）来记录这些这些元数据：

表名	描述
`SYS_TABLES`	整个InnoDB存储引擎中所有的表的信息
`SYS_COLUMNS`	整个InnoDB存储引擎中所有的列的信息
`SYS_INDEXES`	整个InnoDB存储引擎中所有的索引的信息
`SYS_FIELDS`	整个InnoDB存储引擎中所有的索引对应的列的信息
SYS_FOREIGN	整个InnoDB存储引擎中所有的外键的信息
SYS_FOREIGN_COLS	整个InnoDB存储引擎中所有的外键对应列的信息
SYS_TABLESPACES	整个InnoDB存储引擎中所有的表空间信息
SYS_DATAFILES	整个InnoDB存储引擎中所有的表空间对应文件系统的文件路径信息
SYS_VIRTUAL	整个InnoDB存储引擎中所有的虚拟生成列的信息

这些系统表也被称为 `数据字典` ，它们都是以 `B+` 树的形式保存在系统表空间的某些页面中，其中 `SYS_TABLES` 、 `SYS_COLUMNS` 、 `SYS_INDEXES` 、 `SYS_FIELDS` 这四个表尤其重要，称之为基本系统表（basic system tables），我们先看看这4个表的结构：

SYS_TABLES表结构

列名	描述
NAME	表的名称。主键
ID	InnoDB存储引擎中每个表都有一个唯一的ID。(二级索引)
N_COLS	该表拥有列的个数
TYPE	表的类型，记录了一些文件格式、行格式、压缩等信息
MIX_ID	已过时，忽略
MIX_LEN	表的一些额外的属性
CLUSTER_ID	未使用，忽略
SPACE	该表所属表空间的ID

SYS_COLUMNS表结构

列名	描述
TABLE_ID	该列所属表对应的ID。（与 POS 一起构成联合主键）
POS	该列在表中是第几列
NAME	该列的名称
MTYPE	main data type，主数据类型，就是那堆INT、CHAR、VARCHAR、FLOAT、DOUBLE之类的东东
PRTYPE	precise type，精确数据类型，就是修饰主数据类型的那堆东东，比如是否允许NULL值，是否允许负数啥的
LEN	该列最多占用存储空间的字节数
PREC	该列的精度，不过这列貌似都没有使用，默认值都是0

SYS_INDEXES表结构

列名	描述
TABLE_ID	该索引所属表对应的ID。（与 ID 一起构成联合主键）
ID	InnoDB存储引擎中每个索引都有一个唯一的ID
NAME	该索引的名称
N_FIELDS	该索引包含列的个数
TYPE	该索引的类型，比如聚簇索引、唯一索引、更改缓冲区的索引、全文索引、普通的二级索引等等各种类型
SPACE	该索引根页面所在的表空间ID
PAGE_NO	该索引根页面所在的页面号
MERGE_THRESHOLD	如果页面中的记录被删除到某个比例，就把该页面和相邻页面合并，这个值就是这个比例

SYS_FIELDS表结构

列名	描述
INDEX_ID	该索引列所属的索引的ID。（与 POS 一起构成联合主键）
POS	该索引列在某个索引中是第几列
COL_NAME	该索引列的名称

注意：用户是 不能直接访问 InnoDB的这些内部系统表，除非你直接去解析系统表空间对应文件系统上的文件。不过考虑到查看这些表的内容可能有助于大家分析问题，所以在系统数据库 information_schema 中提供了一些以 innodb_sys 开头的表：

```
mysql> USE information_schema;
Database changed

mysql> SHOW TABLES LIKE 'innodb_sys%';
+-----------------------------------------+
| Tables_in_information_schema (innodb_sys%) |
+-----------------------------------------+
| INNODB_SYS_DATAFILES                    |
| INNODB_SYS_VIRTUAL                      |
| INNODB_SYS_INDEXES                      |
| INNODB_SYS_TABLES                       |
| INNODB_SYS_FIELDS                       |
| INNODB_SYS_TABLESPACES                  |
| INNODB_SYS_FOREIGN_COLS                 |
| INNODB_SYS_COLUMNS                      |
| INNODB_SYS_FOREIGN                      |
| INNODB_SYS_TABLESTATS                   |
+-----------------------------------------+
10 rows in set (0.00 sec)
```

在 `information_schema` 数据库中的这些以 `INNODB_SYS` 开头的表并不是真正的内部系统表（内部系统表就是我们上边以 `SYS` 开头的那些表），而是在存储引擎启动时读取这些以 `SYS` 开头的系统表，然后填充到这些以 `INNODB_SYS` 开头的表中。以 `INNODB_SYS` 开头的表和以 `SYS` 开头的表中的字段并不完全一样，但供大家参考已经足矣。