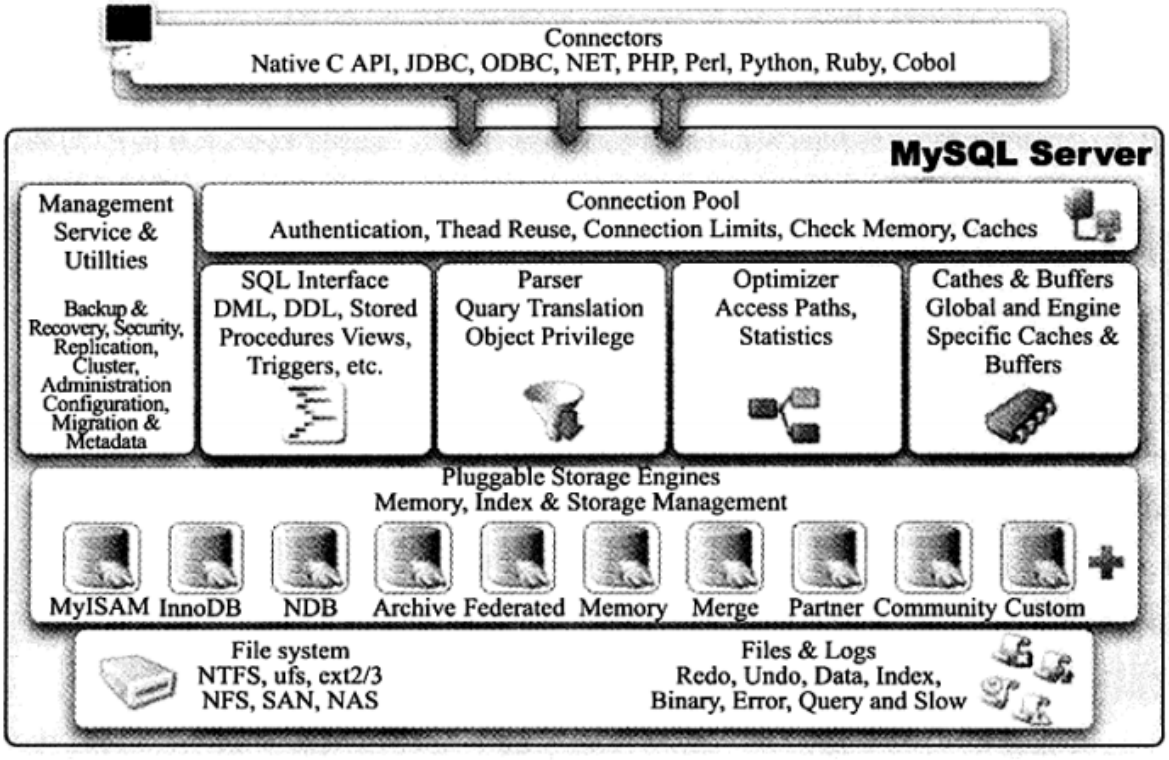
补充：进入MySQL命令行

```
[root@izuf6i3w0v8mmmqlv77eanz ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED
    STATUS              PORTS                                  NAMES
9c45085572ae        mysql               "docker-entrypoint..."   5 months ago
     Up 18 hours         0.0.0.0:3306->3306/tcp, 33060/tcp   baka_mysql
[root@izuf6i3w0v8mmmqlv77eanz ~]# docker exec -it 9c45085572ae bash
root@9c45085572ae:/# mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 11
Server version: 8.0.15 MySQL Community Server - GPL

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

1. MySQL体系结构和存储引擎

1. MySQL被设计成一个单进程多线程架构的数据库，MySQL数据库实例在系统上的表现就是一个进程
2. 当启动实例时，读取配置文件，根据配置文件的参数来启动数据库实例；若没有，按编译时的默认参数设置启动
3. MySQL体系结构



结构	说明
Connectors	不同语言中与SQL的交互
Management Services & Utilities	管理服务和工具组件，例如备份恢复、MySQL复制、集群等
Connection Pool	连接池组件，管理缓冲用户连接、用户名、密码、权限校验、线程处理等需要缓存的需求
SQL Interface	SQL接口组件，接收用户的SQL命令，并且返回用户需要查询的结果
Parser	查询分析器组件，SQL命令传递到解析器的时候会被解析器验证和解析
Optimizer	优化器组件，对查询进行优化
Cache & Buffer	缓冲组件，查询缓存，如果查询缓存有命中的查询结果，查询语句就可以直接去查询缓存中取数据，MySQL 8被取消
Engine	插件式存储引擎，是MySQL中具体与文件打交道的子系统，是MySQL中最具特色的一个地方
File System	物理文件

4. InnoDB存储引擎

   存储引擎是基于表的，而不是数据库

   - 支持事务：行锁设计、支持外键
   - 将数据放到一个逻辑的表空间中，ibd文件
   - 通过多版本并发控制（MVCC）来获得高并发性，并且实现了SQL的4个标准隔离级别，默认为可重复读
   - 提供插入缓存、二次写、自适应哈希索引、预读等高性能和高可用的功能
   - 采用聚集的方式存储表中数据，按主键的顺序进行存放。没有指定主键，会为每一行生成一个6字节的ROWID，并以此作为主键

5. MyISAM

   - 不支持事务、表锁设计、支持全文索引

2. InnoDB存储引擎

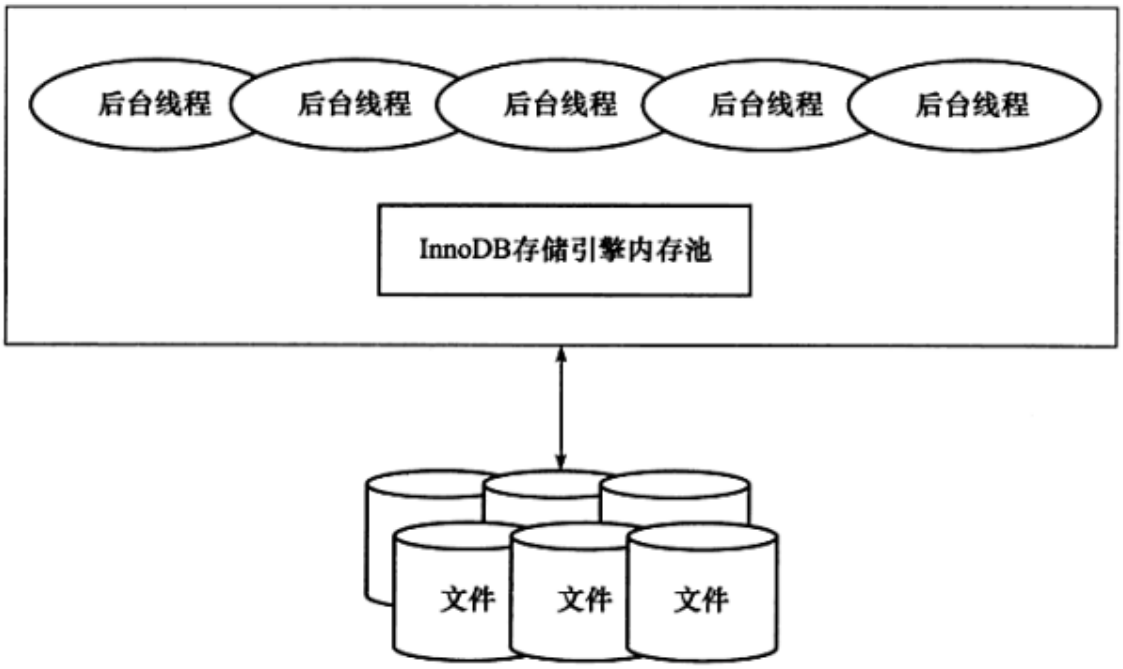


图 2-1　InnoDB 存储引擎体系架构

show engine innodb status;

注意：显示的不是当前的状态，而是过去某个时间范围内InnoDB存储引擎的状态。从下面的例子可以发现，`Per second averages calculated from the last 23 seconds`代表的信息为过去23秒内数据库状态。

```
mysql> show engine innodb status\G;
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
2019-09-11 04:31:08 7f4c7398a700 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 23 seconds
-----------------
BACKGROUND THREAD
-----------------
srv_master_thread loops: 10 srv_active, 0 srv_shutdown, 227 srv_idle
srv_master_thread log flush and writes: 237
----------
SEMAPHORES
----------
OS WAIT ARRAY INFO: reservation count 12
OS WAIT ARRAY INFO: signal count 12
Mutex spin waits 0, rounds 0, OS waits 0
RW-shared spins 12, rounds 360, OS waits 12
RW-excl spins 0, rounds 0, OS waits 0
Spin rounds per wait: 0.00 mutex, 30.00 RW-shared, 0.00 RW-excl
------------
TRANSACTIONS
------------
Trx id counter 3128
Purge done for trx's n:o < 3125 undo n:o < 0 state: running but idle
History list length 41
```

```
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 0, not started
MySQL thread id 1, OS thread handle 0x7f4c7398a700, query id 89 localhost root
init
show engine innodb status
---TRANSACTION 3107, not started
MySQL thread id 4, OS thread handle 0x7f4c73906700, query id 84 125.70.76.210
root
---TRANSACTION 3127, not started
MySQL thread id 3, OS thread handle 0x7f4c73948700, query id 86 125.70.76.210
root
--------
FILE I/O
--------
I/O thread 0 state: waiting for completed aio requests (insert buffer thread)
I/O thread 1 state: waiting for completed aio requests (log thread)
I/O thread 2 state: waiting for completed aio requests (read thread)
I/O thread 3 state: waiting for completed aio requests (read thread)
I/O thread 4 state: waiting for completed aio requests (read thread)
I/O thread 5 state: waiting for completed aio requests (read thread)
I/O thread 6 state: waiting for completed aio requests (write thread)
I/O thread 7 state: waiting for completed aio requests (write thread)
I/O thread 8 state: waiting for completed aio requests (write thread)
I/O thread 9 state: waiting for completed aio requests (write thread)
Pending normal aio reads: 0 [0, 0, 0, 0] , aio writes: 0 [0, 0, 0, 0] ,
 ibuf aio reads: 0, log i/o's: 0, sync i/o's: 0
Pending flushes (fsync) log: 0; buffer pool: 0
356 OS file reads, 88 OS file writes, 56 OS fsyncs
0.00 reads/s, 0 avg bytes/read, 0.00 writes/s, 0.00 fsyncs/s
-------------------------------------
INSERT BUFFER AND ADAPTIVE HASH INDEX
-------------------------------------
Ibuf: size 1, free list len 0, seg size 2, 0 merges
merged operations:
 insert 0, delete mark 0, delete 0
discarded operations:
 insert 0, delete mark 0, delete 0
Hash table size 276707, node heap has 0 buffer(s)
0.00 hash searches/s, 0.00 non-hash searches/s
---
LOG
---
Log sequence number 1886077
Log flushed up to   1886077
Pages flushed up to 1886077
Last checkpoint at  1886077
0 pending log writes, 0 pending chkp writes
29 log i/o's done, 0.00 log i/o's/second
----------------------
BUFFER POOL AND MEMORY
----------------------
Total memory allocated 137363456; in additional pool allocated 0
Dictionary memory allocated 108374
Buffer pool size   8192
Free buffers       7854
Database pages     338
Old database pages 0
Modified db pages  0
```

```
Pending reads 0
Pending writes: LRU 0, flush list 0, single page 0
Pages made young 0, not young 0
0.00 youngs/s, 0.00 non-youngs/s
Pages read 328, created 10, written 53
0.00 reads/s, 0.00 creates/s, 0.00 writes/s
No buffer pool page gets since the last printout
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
LRU len: 338, unzip_LRU len: 0
I/O sum[0]:cur[0], unzip sum[0]:cur[0]
--------------
ROW OPERATIONS
--------------
0 queries inside InnoDB, 0 queries in queue
0 read views open inside InnoDB
Main thread process no. 1, id 139966295516928, state: sleeping
Number of rows inserted 0, updated 11, deleted 0, read 67
0.00 inserts/s, 0.00 updates/s, 0.00 deletes/s, 0.00 reads/s
----------------------------
END OF INNODB MONITOR OUTPUT
============================

1 row in set (0.00 sec)
```

2.1 InnoDB体系结构

2.1.1 后台线程

后台线程的主要作用是负责刷新内存池中的数据，保证缓存池中的内存缓存的是最近的数据。

1. **Master Thread**

   是一个非常核心的后台线程，主要负责将缓冲池中的数据异步刷新到磁盘，保证数据的一致性，包括脏页的刷新、合并插入缓冲、UNDO页的回收等

2. **IO Thread**

   大量使用AIO（Async IO）来处理写IO请求（write、read、insert buffer和IO thread），这样可以极大提高数据库的性能。而IO Thread的工作主要是负责这些IO请求的回调（call back）处理。

   有4种类型的IO Thread：write、read、insert buffer和log IO thread

3. **Purge Thread**

   事务被提交后，其所使用的undo log(回滚日志)可能不再需要，因此需要Purge Thread来回收已经使用并分配的undo页。

   在 1.1版本之前，purge操作仅在InnoDB的Master Thread中完成。而从1.1开始，purge操作可以独立到单独的线程中进行。

4. **Page Cleaner Thread**

   在InnoDB 1.2.x版本引入。作用是将之前版本中脏页的刷新操作都放到单独的线程中完成，以减轻Master Thread的工作

2.1.2 内存

缓冲池

InnoDB存储引擎是基于磁盘存储的，并将其中的记录按照页的方式进行管理。由于CPU速度与磁盘速度之间的鸿沟，基于磁盘的系统通常使用缓存池技术来提供数据库的整体性能。

在数据库中进行读取页的操作，首先将从磁盘读到的页放在缓冲池中，这个过程称为将页“FIX”在缓冲池中。下一次再读相同的页时，首页判断该页是否在缓存池中。若在缓冲池中，称该页在缓冲池中被命中，直接读取该页。否则，读取磁盘上的页。

对于数据库中页的修改操作，首先修改在缓冲池中的页，然后再以一定的频率（checkpoint机制）刷新到磁盘上。

缓冲池的大小直接影响着数据库的整体性能。

通过命令show variables like 'innodb_buffer_pool_size'可以观察缓冲池的大小。
134217728/1024/1024 = 128M

```
mysql> show variables like 'innodb_buffer_pool_size';
+-------------------------+-----------+
| Variable_name           | Value     |
+-------------------------+-----------+
| innodb_buffer_pool_size | 134217728 |
+-------------------------+-----------+
1 row in set (0.00 sec)
```

缓冲池中缓存的数据页类型有：索引页、数据页、UNDO页、插入缓冲（insert buffer）、自适应哈希索引、InnoDB存储的锁信息、数据字典信息等。

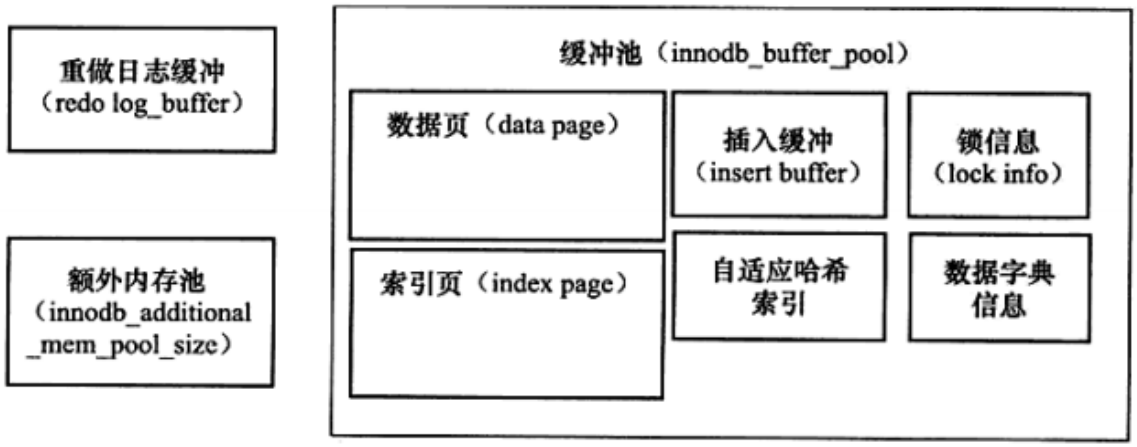


图 2-2　InnoDB 内存数据对象

从InnoDB 1.0.x版本开始，允许有多个缓冲池实例。每个页根据哈希值平均分配到不同缓冲池实例中。减少数据库内部的资源竞争，增加数据库的并发处理能力。

LRU List、Free List和Flush List

缓冲池是一个很大的内存区域，其中存放各种类型的页，那么如何对这么大的内存区域进行管理呢?

1. **LRU List：管理缓冲池中页的可用性**

   通常来说，数据库中的缓冲池是通过LRU（Lastest Recent Used，最近最少使用）算法来进行管理的。即最频繁使用的页在LRU列表的前端，而最少使用的页在LRU列表的尾端。当缓冲池不能存放新读取到的页时，将首先释放LRU列表中尾端的页。

   InnoDB中，对传统的LRU算法做了一些优化，在列表中加入了midpoint位置。新读取到的页，虽然是最新访问到的页，但并不直接放入到LRU列表的首部，而是放入到LRU列表的midpoint位置（默认在LRU列表长度的5/8处）。

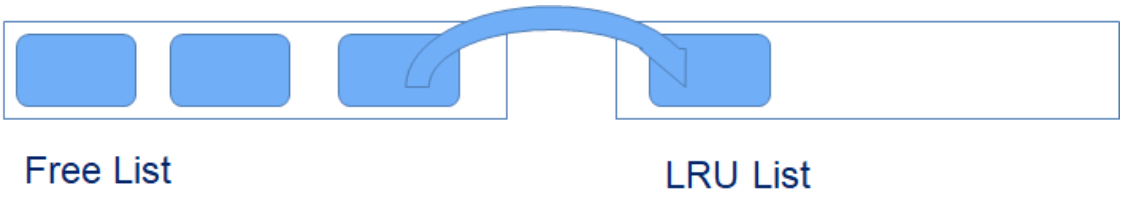
为什么不使用传统的LRU算法？因为有些操作仅仅是在这次操作中需要，并不是活跃数据，就会导致LRU列表中热点数据被刷出，从而影响缓冲池的效率。

为解决这个问题，引入了另外一个参数来进一步管理LRU列表：innodb_old_blocks_time，用于表示页读取到mid位置后需要等待多久才会被加入到LRU列表的热端。

```
mysql> show variables like 'innodb_old_blocks_time'\G;
*************************** 1. row ***************************
variable_name: innodb_old_blocks_time
        Value: 1000
1 row in set (0.01 sec)
```

2. **Free List**

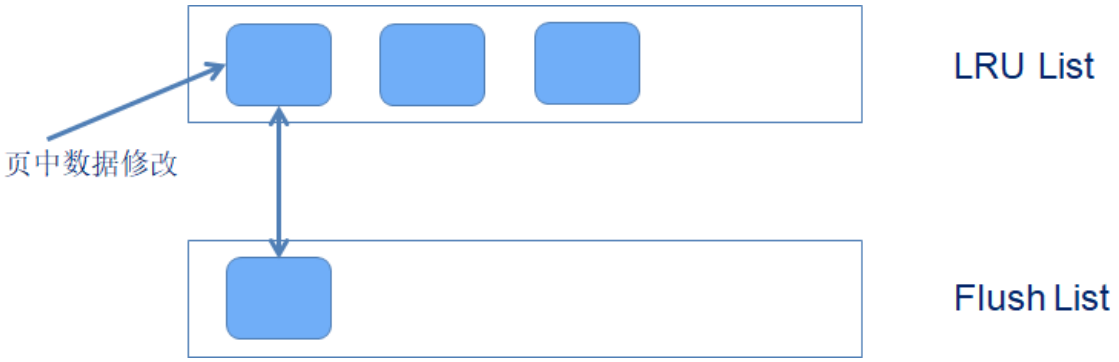
   当数据库刚启动时，LRU列表是空的，即没有任何的页。这时页都存放在Free列表中。当需要从缓冲池中分页时，首先从Free列表中查找是否有可用的空闲页，若有则将该页从Free列表中删除，放入LRU列表中。否则，根据LRU算法，淘汰LRU列表末尾的页，将该内存空间分配给新的页。



3. **Flush List：管理将页刷新回磁盘**

在LRU列表中的页被修改后，称该页为脏页（dirty page），即缓冲池中的页和磁盘上的页的数据产生了不一致。这时数据库会通过CHECKPOINT机制将脏页刷新回磁盘，而Flush列表中的页即为脏页列表。

脏页既存在LRU列表中，也存在于Flush列表中，二者互不影响。



重做日志缓冲

1. InnoDB存储引擎首先将重做日志信息先放入到这个缓冲区，然后按一定频率将其刷新到重做日志文件。
2. 该值可由参数innodb_log_buffer_size控制，默认为8MB，一般不需要设置的很大。
   8388608/1024/1024 = 8M

```
mysql> show variables like 'innodb_log_buffer_size'\G;
*************************** 1. row ***************************
Variable_name: innodb_log_buffer_size
        Value: 8388608
1 row in set (0.00 sec)
```

3. 刷新时机（三种）
    - Matster Thread每一秒将重做日志缓冲刷新到重做日志文件
    - 每个事务提交时会将重做日志缓冲刷新到重做日志文件
    - 当重做日志缓冲池剩余空间小于1/2时，重组日志缓冲刷新到重做日志文件

额外的内存池

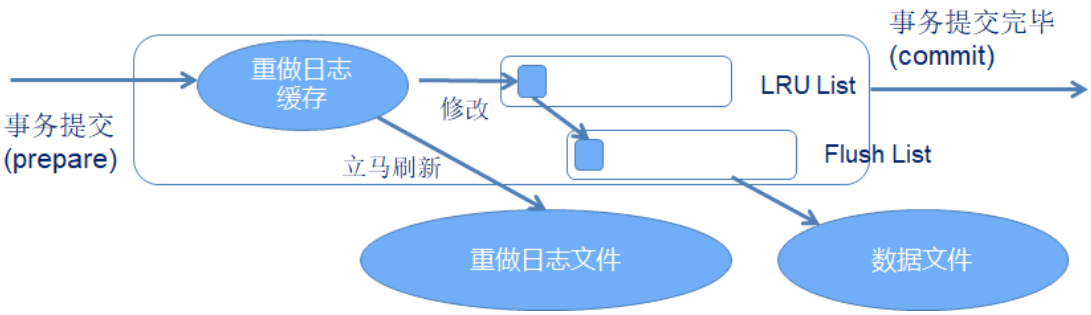
在InnoDB存储引擎中，对内存的管理是通过一种称为内存堆(heap)的方式进行的。在对一些数据结构本身的内存进行分配时，需要从额外的内存池中进行申请，当该区域的内存不够时，会从缓冲池中进行申请。例如，分配了缓冲池(innodb_buffer_pool)，但是每个缓冲池中的帧缓冲(frame buffer)还有对应的缓冲控制对象(buffer control block)，这些对象记录了一些诸如LRU、锁、等待等信息，而这个对象的内存需要从额外内存池中申请。

2.2 Checkpoint技术

1. **问题描述**

    页的操作首先都是在缓冲池中完成的。如果一条DML语句，如Update或Delete改变了页中的记录，那么此时页是脏的，即缓冲池中的页的版本要比磁盘的新。数据库需要将新版本的页从缓冲池刷新到磁盘。

    倘若每次一个页发生变化，就将新页的版本刷新到磁盘，那么这个开销是非常大的。若热点数据集中在某几个页中，那么数据库的性能将变得非常差。同时，如果在从缓冲池将页的新版本刷新到磁盘时发生了宕机，那么数据就不能恢复了。**为了避免发生数据丢失的问题，当前事务数据库系统普遍都采用了Write Ahead Log策略，即当事务提交时，先写重做日志，再修改页。**当由于发生宕机而导致数据丢失时，通过重做日志来完成数据的恢复。这也是事务ACID中D (Durability 持久性)的要求。



2. **Checkpoint（检查点）技术的目的是解决以下几个问题**
    - 缩短数据库的恢复时间

        当数据库发生宕机时，数据库不需要重做所有的日志，因为Checkpoint之前的页都已经刷新回磁盘。故数据库只需对Checkpoint后的重做日志进行恢复。

    - 缓冲池不够用时，将脏页刷新到磁盘

        当缓冲池不够用时，根据LRU算法会溢出最近最少使用的页，若此页为脏页，那么需要强制执行Checkpoint，将脏页也就是页的新版本刷回磁盘。

    - 重做日志不可用时，刷新脏页

对重做日志的设计都是循环使用的，并不是让其无限增大。数据库发生宕机时，若此时重做日志还需要使用，那么必须产生Checkpoint，将缓冲池中的页至少刷新到当前重做日志的位置。

3. **通过LSN（Log Sequence Number）来标记版本**
4. **触发时机**
    - Sharp Checkpoint

      数据库关闭时，将所有的脏页都刷新回磁盘，默认工作方式。
    - Fuzzy Checkpoint

      数据库运行时，使用Fuzzy Checkpoint只刷新一部分脏页，而不是刷新所有的脏页回磁盘。
5. **在InnoDB存储引擎中可能发生如下几种情况的Fuzzy Checkpoint**
    - **Master Thread Checkpoint**

      Master Thread差不多每十秒刷新一定比例的脏页回磁盘
    - **FLUSH_LRU_LIST Checkpoint**

      InnoDB需要保证LRU列表中有差不多100个空闲页(innodb_lru_scan_depth=1024)可供使用，如果没有则将LRU列表尾端的页移除，若这些页中有脏页，则需要进行Checkpoint
    - **Async/Sync Flush Checkpoint**

      发生在重做日志文件不可用(日志不能被覆写)的情况，这时需要强制将一些页刷新回磁盘，而此时脏页是从Flush_List中选取的
    - **Dirty Page too much Checkpoint**

      脏页的数量太多，导致InnoDB存储引擎强制进行Checkpoint；通过参数innodb_max_dirty_pages_pct控制，默认值为75，当缓冲池中脏页的数量占据75%时，强制进行Checkpoint。

2.3 Master Thread工作方式

Master Thread具有最高的线程优先级级别。其内部由多个循环（loop）组成：主循环（loop）、后台循环（background loop）、刷新循环（flush loop）、暂停循环（suspend loop）。Master Thread会根据数据库运行的状态在loop、background loop、flush loop和suspend loop中切换。

Loop被称为主循环，因为大多数操作都在这个循环中，其中有两大部分的操作——每秒操作和每10秒的操作。

1. **每秒操作**
    - 日志缓冲刷新到磁盘，即使这个事务还没有提交(总是);
    - 合并插人缓冲(可能);
    - 至多刷新100个InnoDB的缓冲池中的脏页到磁盘(可能);
    - 如果当前没有用户活动，则切换到background loop (可能)。
2. **每10秒操作**
    - 刷新100个脏页到磁盘(可能的情况下);
    - 合并至多5个插入缓冲(总是);
    - 将日志缓冲刷新到磁盘(总是);
    - 删除无用的Undo页(总是);
    - 刷新100个或者10个脏页到磁盘(总是)。

循环切换：Master Thread -> background loop（可能会跳转到 flush loop） -> flush loop -> suspend loop，将Master Thread挂起，等待事件的发生

2.4 InnoDB关键特性

InnoDB存储引擎的关键特性包括：

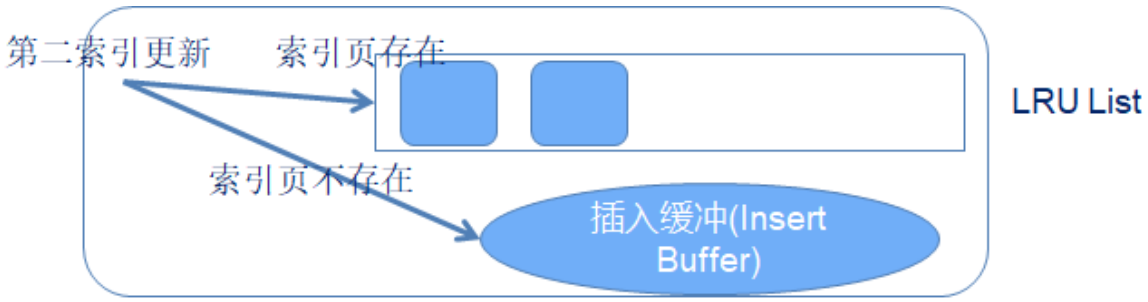
1. 插入缓冲（Insert Buffer）
2. 两次写（Double write）
3. 自适应哈希索引（Adaptive Hash Index）
4. 异步IO（Async IO）
5. 刷新邻接页（Flush Neighbor Page）

这些特性为InnoDB存储引擎带来更好的性能以及更高的可靠性

2.4.1 插入缓冲

非聚集索引的插入是离散的，需要随机读取磁盘，由于随机读取的存在而导致插入操作性能下降，引入插入缓冲是为了提高非聚集索引的插入性能。

对于非聚集索引的插入或更新操作，不是每一次直接插入到索引页中，而是先判断插入的非聚集索引页是否在缓冲池中，若在，则直接插入；若不在，则先放入一个Insert Buffer对象中。数据库这个非聚集索引已经插入到叶子节点，而实际并没有，只是存放在另外一个位置。然后再以一定的频率和情况进行Insert Buffer和辅助索引页子节点的merge(合并)操作，这时通常能将多个插入合并到一个操作中(因为在一个索引页中)。



使用Insert Buffer需要满足以下两个条件：

1. 索引是辅助索引
2. 索引不是唯一的

在InnoDB 1.0.x后对于DML的操作也可以进行缓冲，它们分别是：Insert Buffer、Delete Buffer、Purge Buffer。

数据结构实现：全局只有一颗Insert Buffer B+树，负责对所有的表的辅助索引进行Insert Buffer。而这颗B+树存放在共享表空间中，默认也就是ibdata1中。

2.4.2 二次写

提高数据页的可靠性

当数据库发生宕机时，在应用重做日志之前，用户需要一个页的副本，当写入失效发生（写了一部分，又宕机了）时，先通过页的副本来还原该页（因为原页可能已经损坏，对其重做是没有意义的），再进行重做，这就是double write。

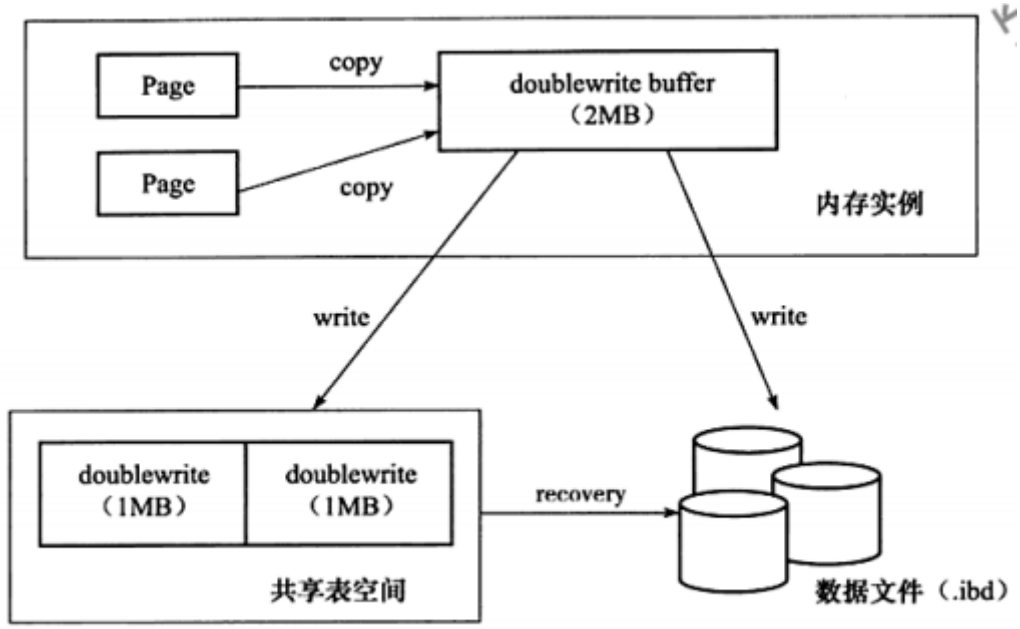


图 2-5　InnoDB 存储引擎 doublewrite 架构

2.4.3 自适应哈希索引

哈希(hash)是一种非常快的查找方法，在一般情况下这种查找的时间复杂度为0(1)，即一般仅需要一次查找就能定位数据。而B+树的查找次数，取决于B+树的高度，在生产环境中，B+树的高度一般为3~4层，故需要3~ 4次的查询。

InnoDB存储引擎会监控对表上各索引页的查询。如果观察到建立哈希索引可以带来速度提升，则建立哈希索引，称之为自适应哈希索引(Adaptive Hash Index, AHI)。AHI是通过缓冲池的B+树页构造而来，因此建立的速度很快，而且不需要对整张表构建哈希索引。InnoDB 存储引擎会自动根据访问的频率和模式来自动地为某些热点页建立哈希索引。

自适应哈希索引有一个要求，即对这个页的连续访问模式必须是一样的。比如联合索引(a, b)，如果交替 WHERE a = xx 和 WHERE a = xx and b = xx 是不行的。

2.4.4 异步IO

为了提高磁盘操作性能，当前数据库系统都采用异步IO（AIO）的方式来处理磁盘操作。InnoDB存储引擎也是如此。

2.4.5 刷新邻接页

当刷新一个脏页时，InnoDB存储引擎会检测该页所在区的所有页，如果是脏页，那么一起进行刷新。

3. 文件

MySQL数据库和InnoDB存储引擎表的各种类型文件如下：

1. **参数文件**：告诉MySQL实例启动时在哪里可以找到数据库文件，并且指定某些初始化参数。
2. **日志文件**：用来记录MySQL实例对某种条件作出响应时写入的文件，如错误日志文件、二进制文件、慢查询日志文件、查询日志文件等。
3. **socket文件**：当用UNIX域套接字方式进行连接时需要的文件。
4. **pid文件**：MySQL实例的进程ID文件。
5. **MySQL表结构文件**：用来存放MySQL表结构定义文件。
6. **存储引擎文件**：存储记录和索引等数据。

3.1 参数文件

MySQL数据库的参数文件是以**文本方式**进行存储的。用户可以直接通过一些常用的文本编辑器进行参数的修改。

可以把数据库参数看成是一个键/值对。如：innodb_buffer_pool_size=1G。可以通过show varibales查看数据库中的所有参数，也可以通过like 来过滤参数名。或者通过information_schema下的GLOBAL_VARIABLES视图来查找。

```
mysql> show variables like 'innodb_buffer%'\G;
*************************** 1. row ***************************
Variable_name: innodb_buffer_pool_dump_at_shutdown
        Value: OFF
*************************** 2. row ***************************
Variable_name: innodb_buffer_pool_dump_now
        Value: OFF
*************************** 3. row ***************************
Variable_name: innodb_buffer_pool_filename
        Value: ib_buffer_pool
*************************** 4. row ***************************
Variable_name: innodb_buffer_pool_instances
        Value: 8
*************************** 5. row ***************************
Variable_name: innodb_buffer_pool_load_abort
        Value: OFF
*************************** 6. row ***************************
Variable_name: innodb_buffer_pool_load_at_startup
        Value: OFF
*************************** 7. row ***************************
Variable_name: innodb_buffer_pool_load_now
        Value: OFF
*************************** 8. row ***************************
Variable_name: innodb_buffer_pool_size
        Value: 134217728
8 rows in set (0.00 sec)
```

MySQL数据库中的参数可以分为两类：

1. 动态（dynamic）参数：可以在MySQL实例运行中进行更改
2. 静态（static）参数：在整个实例生命周期都不能更改，就好像是只读的

可以通过SET命令对动态的参数值进行修改，语法如下：

```
SET [global | session] system_var_name = expr
SET [@@global. | @@session.] system_var_name = expr
```

- global：参数的修改是基于整个实例的生命周期
- session：参数的修改是基于当前会话

3.2 日志文件

日志文件记录了影响MySQL数据库的各种类型活动。常见的日志文件有：

1. 错误日志（error log）
2. 二进制日志（binlog）

3. 慢查询日志（show query log）
4. 查询日志（log）

这些日志文件可以帮助DBA对MySQL数据库的运行状态进行诊断，从而更好地进行数据库层面的优化。

3.2.1 错误日志

对MySQL的启动、运行、关闭过程进行了记录。该文件不仅记录了所有的错误信息，也记录了一些警告或正确的信息。可以通过show variables like 'log_error'来定位该文件。默认情况下错误文件的文件名为服务器的主机名。

```
mysql> show variables like 'log_error'\G;
*************************** 1. row ***************************
Variable_name: log_error
        Value:
1 row in set (0.00 sec)
```

3.2.2 慢查询日志

帮助定位可能存在问题的SQL语句，从而进行SQL语句层面的优化。例如，可以在MySQL启动时设一个阈值，将运行时间超过该值的所有SQL语句都记录到慢查询日志文件中。该阈值可以通过参数long_query_time来设置，默认值为10，代表10秒。 只会记录执行时间大于该阈值的SQL。

```
mysql>  show variables like 'long_query_time'\G;
*************************** 1. row ***************************
Variable_name: long_query_time
        Value: 10.000000
1 row in set (0.00 sec)
```

默认情况下，MySQL数据库并不启动慢查询日志，用户需要手工将这个参数设置为ON；

```
mysql> set global slow_query_log  = 'ON';
Query OK, 0 rows affected (0.02 sec)

mysql> show variables like 'slow_query_log'\G;
*************************** 1. row ***************************
Variable_name: slow_query_log
        Value: ON
1 row in set (0.00 sec)
```

MySQL 5.1开始可以将慢查询的日志记录放入一张表中，名为slow_log

```
mysql> show create table mysql.slow_log\G;
*************************** 1. row ***************************
       Table: slow_log
Create Table: CREATE TABLE `slow_log` (
  `start_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  `user_host` mediumtext NOT NULL,
  `query_time` time NOT NULL,
  `lock_time` time NOT NULL,
  `rows_sent` int(11) NOT NULL,
  `rows_examined` int(11) NOT NULL,
  `db` varchar(512) NOT NULL,
  `last_insert_id` int(11) NOT NULL,
```

```
  `insert_id` int(11) NOT NULL,
  `server_id` int(10) unsigned NOT NULL,
  `sql_text` mediumtext NOT NULL,
  `thread_id` bigint(21) unsigned NOT NULL
) ENGINE=CSV DEFAULT CHARSET=utf8 COMMENT='Slow log'
1 row in set (0.00 sec)
```

3.2.3 查询日志

记录了所有对MySQL数据库请求的信息，无论这些请求是否得到了正确的执行。默认文件名：主机名.log

从MySQL 5.1开始，可以将查询日志的记录放到mysql架构下的general_log表中。

```
mysql> show create table mysql.general_log\G;
*************************** 1. row ***************************
       Table: general_log
Create Table: CREATE TABLE `general_log` (
  `event_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  `user_host` mediumtext NOT NULL,
  `thread_id` bigint(21) unsigned NOT NULL,
  `server_id` int(10) unsigned NOT NULL,
  `command_type` varchar(64) NOT NULL,
  `argument` mediumtext NOT NULL
) ENGINE=CSV DEFAULT CHARSET=utf8 COMMENT='General log'
1 row in set (0.02 sec)
```

3.2.4 二进制日志

记录了对MySQL数据库执行更改的所有操作，但是不包括select和show这类操作，因为这类操作对数据本身并没有修改。此外，二进制日志还包括执行数据库更改操作的时间等其他额外信息。

默认不开启，需要手动修改配置文件参数来启动。

bin_log.index为二进制的索引文件，用来存储过往产生的二进制日志序号。如果超过二进制文件的最大值，则产生新的二进制文件，后缀名+1，并记录到.index文件。

二进制日志主要有以下几种作用：

1. **恢复**：某些数据的恢复需要二进制日志
2. **复制**：通过复制和执行二进制日志使一台远程的MySQL数据库（一般称为slave或standby）与一台MySQL数据库（一般称为master或primary）进行实时同步
3. **审计**：通过二进制日志中的信息来进行审计，判断是否有对数据进行注入的攻击

二进制日志的格式有三种，由binlog_format控制：STATEMENT、ROW和MIXED

3.3 套接字文件

在UNIX系统本地连接MySQL可以采用UNIX域套接字方式，这种方式需要一个套接字文件。套接字文件可由参数scoket控制。

```
mysql> show variables like 'socket'\G;
*************************** 1. row ***************************
Variable_name: socket
        Value: /var/run/mysqld/mysqld.sock
1 row in set (0.00 sec)
```

3.4 pid文件

当MySQL实例启动时，会将自己的进程ID写入一个文件中——该文件即为 pid文件。该文件可由参数 pid_file 控制，默认位于数据库目录下，文件名为主机名.pid。

```
mysql> show variables like 'pid_file'\G;
*************************** 1. row ***************************
Variable_name: pid_file
        Value: /var/run/mysqld/mysqld.pid
1 row in set (0.00 sec)
```

3.5 表结构定义文件

因为MySQL插件式存储引擎的体系结构的关系，MySQL 数据的存储是根据表进行的，每个表都会有与之对应的文件。但不论表采用何种存储引擎，MySQL都有一个以frm为后缀名的文件，这个文件记录了该表的表结构定义。

frm还用来存放视图的定义，如用户创建了一个v_a视图，那么对应地会产生一个v_a.frm 文件，用来记录视图的定义，该文件是文本文件，可以直接使用cat命令进行查看。

3.6 InnoDB存储引擎文件

每个表存储引擎有自己独有的文件，这些文件包括重做日志文件、表空间文件。

3.6.1 表空间文件

InnoDB采用将存储的数据按表空间(tablespace)进行存放的设计。在默认配置下会有一个初始大小为10MB，名为ibdata1的文件。该文件就是默认的表空间文件(tablespace file)，用户可以通过参数innodb data_ file_ path对其进行设置。

若设置了参数innodb_file_per_table，则用户可以将每个基于InnoDB存储引擎的表产生一个独立表空间。独立表空间的命名规则为：表名.ibd。这样就可以不用将所有的数据都存放于默认的表空间中。

这些单独的表空间文件仅存储该表的数据、索引和插入缓冲BITMAP等信息，其余信息还是存放在默认的表空间中。InnoDB存储引擎对于文件的存储方式如下：

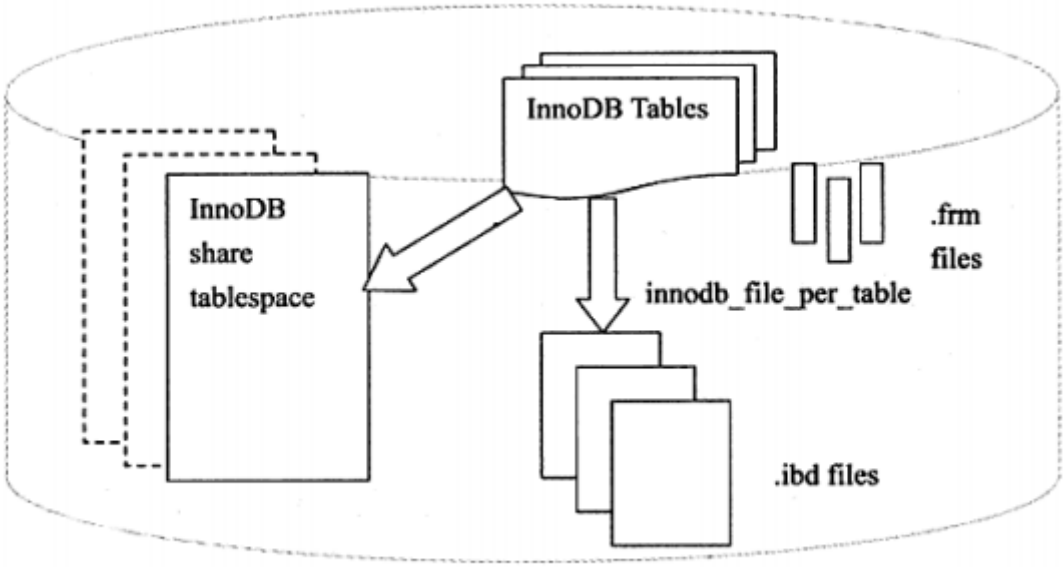


图 3-1　InnoDB 表存储引擎文件

3.6.2 重做日志文件

在默认情况下，在InnoDB存储引擎的数据目录下会有两个名为ib_logfile0和ib_logfile1的文件，采用**循环写入**的方式运行，它们记录了对于InnoDB存储引擎的事务日志。

当实例或介质失败(media failure)时，重做日志文件就能派上用场。例如，数据库由于所在主机掉电导致实例失败，InnoDB存储引擎会使用重做日志恢复到掉电前的时刻，以此来保证数据的完整性。

重做日志文件的大小设置对于InnoDB存储引擎的性能有着非常大的影响。太大，恢复可能需要很长的时间；太小，会频繁地发生async checkpoint，导致性能的抖动。

重做日志和二进制日志的区别

	二进制日志	**重做日志**
层级	记录所有与MySQL数据库有关的日志记录，包括InnoDB、MyISAM等	只记录InnoDB存储引擎本身的事务日志
记录内容	记录的是关于一个事务的具体操作内容	记录的是关于每个页的更改的物理情况
写入时间	仅在事务提交前进行提交，即只写磁盘一次，无论这时该事务多大	在事务进行的过程中，不断有重做日志条目被写入到重做日志文件中
写入方式	日志可以重加	日志是循环重复写的

重做日志写入过程如下：写入重做日志文件的操作不是直接写，而是先写入一个重做日志缓冲中，然后按照一定的条件顺序地写入日志文件。

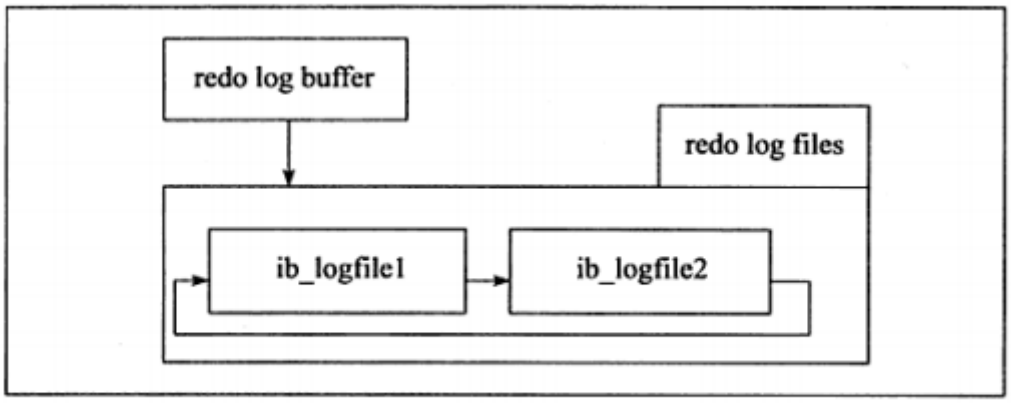


图 3-3　重做日志写入过程

4. 表

4.1 索引组织表

索引组织表：表都是根据主键顺序组织存放的。每张表都有一个主键，如果创建表时没有显式地定义主键，InnoDB首先会判断是否有非空的唯一索引，有，则该列为主键；否则自动创建一个6字节大小的指针_rowid。

4.2 InnoDB逻辑存储结构

所有数据都被逻辑地存放在一个空间（tablespace）中，称之为表空间。表空间又由段（segment）、区（extent）、页（page）组成。

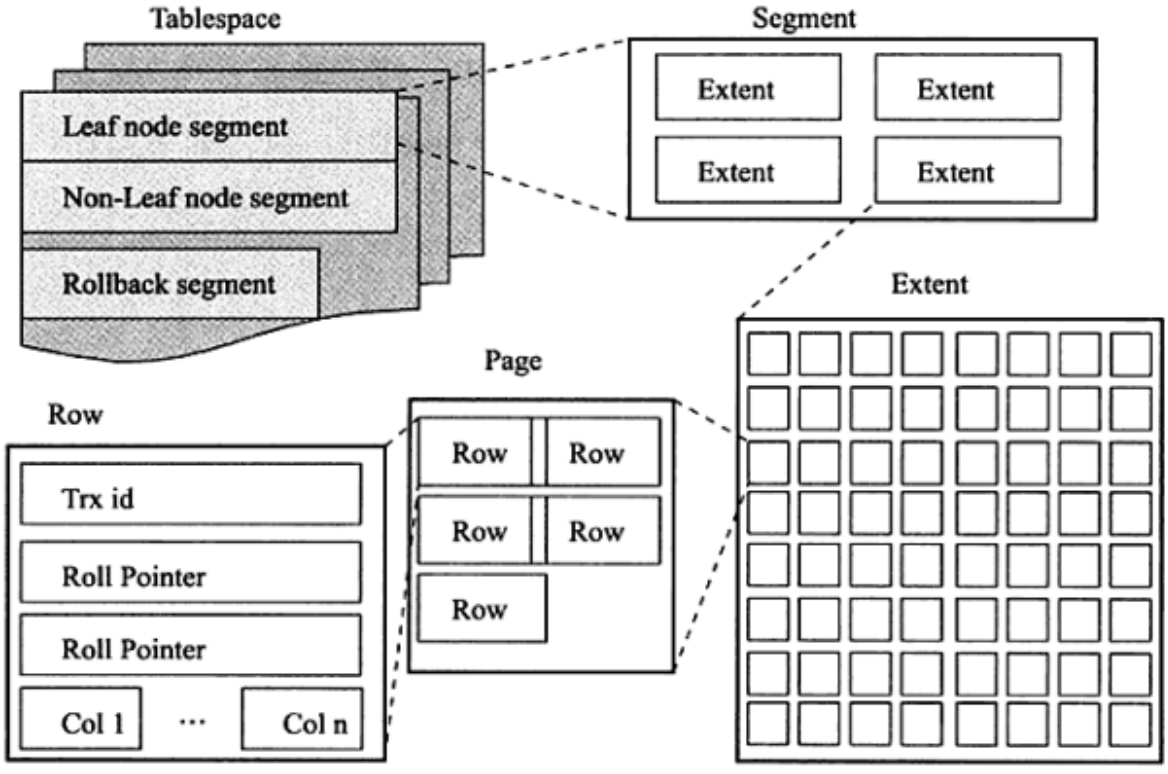


图 4-1　InnoDB 逻辑存储结构

4.2.1 表空间

所有数据都存放在表空间中，默认ibdata1，如果启用innodb_file_per_table，则每个表内的数据可以单独放到一个表空间（.ibd）内。注意，如果启用了innodb_file_per_table，每张表的表空间内存放的只是的数据、索引和插入缓冲BITMAP页，其他类的信息，如回滚日志、插入缓冲、索引页、系统事务信息、二次缓冲写等还是存放在原来的共享表空间内。

4.2.2 段

常见的段有数据段、索引段、回滚段等。

InnoDB存储引擎表是索引组织的，因此数据即索引、索引即数据。数据段即为B+树的叶子节点，索引即为B+树的非叶子节点。回滚段比较特殊，后面单独介绍。

4.2.3 区

由连续页组成的空间，在任何情况下每个区的大小都为1MB。为了保证区中页的连续性，InnoDB 存储引擎一次从磁盘申请4~5个区。在默认情况下，InnoDB存储引擎页的大小为16KB，即一个区中一共有64个连续的页。

启用innodb_file_per_table后，先使用32个页大小的碎片页来存放数据，在使用完这些页之后才是64个连续页的申请。目的：在开始时申请较小的空间，节约磁盘容量的开销。

4.2.4 页

磁盘管理的最小单位，默认16KB。

常见的页类型有：

- 数据页
- undo页
- 系统页
- 事务数据页
- 插入缓冲位图页
- 插入缓存空闲列表页
- 未压缩的二进制大对象页
- 压缩的二进制大对象页

4.2.5 行

InnoDB存储引擎是面向行的，数据是按照行进行存放的。每个页最多存放 `16KB/2-200=7992` 行记录。

4.3 行记录格式

通过 `SHOW TABLE STATUS like 'table_name'` 来查看当前表使用的行格式。

```
mysql> show table status like 'user'\G;
*************************** 1. row ***************************
           Name: user
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 30
 Avg_row_length: 546
    Data_length: 16384
Max_data_length: 0
   Index_length: 16384
      Data_free: 0
```

```
 Auto_increment: 1037
    Create_time: 2019-09-11 04:27:05
    Update_time: NULL
     Check_time: NULL
      Collation: utf8_bin
       Checksum: NULL
 Create_options: row_format=COMPACT
        Comment:
1 row in set (0.00 sec)
```

行记录格式有：Compact、Redundant、Compress、Dynamic

4.3.1 Compact行记录格式

MySQL 5.0引入，设计目标是高效地存储数据。一个页存放的行数据越多，其性能就越高。

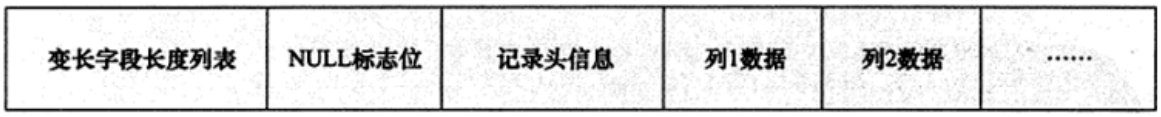


图 4-2　Compact 行记录的格式

1. 变成字段长度列表

   非NULL，记录字段的实际长度，按照列的顺序**逆序**放置，若列的长度小于255字节，用1字节表示；若大于255个字节，用2字节表示。变成字段的长度最大不能超过2字节，因为VARCHAR类型的最大长度限制为65535。

2. NULL标志位

   标识行数据是否有NULL值，有则用1表示

3. 记录头信息

表 4-1　Compact 记录头信息

名　　称	大小 (bit)	描　　述
()	1	未知
()	1	未知
deleted_flag	1	该行是否已被删除
min_rec_flag	1	为 1，如果该记录是预先被定义为最小的记录
n_owned	4	该记录拥有的记录数
heap_no	13	索引堆中该条记录的排序记录
record_type	3	记录类型，000 表示普通，001 表示 B+ 树节点指针，010 表示 Infimum，011 表示 Supremum，1xx 表示保留
next_record	16	页中下一条记录的相对位置
Total	40	

4. 列数据

   实际存储每个列的数据。

   注意：NULL不占该部分任何空间，即NULL除了占有NULL标志位，实际存储不占用任何空间。

5. 隐藏列

   事务ID列和回滚指针，如果没有定义主键，每行还会增加一个6字节的rowid列。

案例：

1. 创建表

```
mysql> CREATE TABLE mytest (
    -> t1 VARCHAR(10),
    -> t2 VARCHAR(10),
    -> t3 CHAR(10),
    -> t4 VARCHAR(10)
    ->) ENGINE=INNODB CHARSET=LATIN1 ROW_FORMAT=COMPACT;
Query OK, 0 rows affected (0.00 sec)
```

2. 插入数据

```
mysql> INSERT INTO mytest
    -> VALUES ('a','bb','bb','ccc');
Query OK, 1 row affected (0.01 sec)
```

3. 第一行数据

```
03 02 01/* 变长字段长度列表，逆序 */
00 /*NULL 标志位，第一行没有 NULL 值 */
00 00 10 00 2c /*Record Header，固定 5 字节长度 */
00 00 00 2b 68 00/*RowID InnoDB 自动创建，6 字节 */
00 00 00 00 06 05/*TransactionID*/
80 00 0000 32 01 10/*Roll Pointer*/
61/* 列 1 数据 'a'*/
62 62/* 列 2 数据 'bb'*/
62 62 20 20 20 20 20 20 20 20/* 列 3 数据 'bb' */
63 63 63/* 列 4 数据 'ccc'*/
```

固定字段CHAR字段未能在完全占有其长度空间时，会用0x20来进行填充。

4.3.2 行溢出数据

将一条记录中的某些数据存储在真正的数据页面之外。

VARCHAR最多可以存储65535字节，但是真的可以吗?

```
mysql> create table test(
    -> a varchar(65535)
    -> )charset=latin1 engine=innodb;
ERROR 1118 (42000): Row size too large. The maximum row size for the used table
type, not counting BLOBs, is 65535. This includes storage overhead, check the
manual. You have to change some columns to TEXT or BLOBs
```

说明有其他开销，实际能存放65532。

如果设置字符类型为UFT-8和GBK，结果如何?

行溢出数据的存储，数据页保存前768字节的前缀数据，之后是偏移量，指向行溢出页。

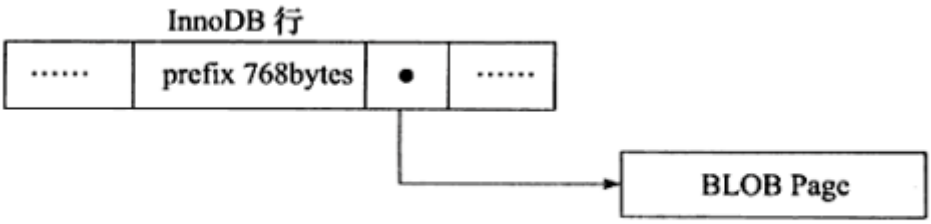


图 4-4　行溢出数据的存储

InnoDB 1.0.x版本开始引入了新的文件格式：Compressed和Dynamic，采用完全的行溢出方式，在数据页只存放20个字节的指针，实际存放的数据都在Off Page中。

图 4-5　Barracuda 文件格式的溢出行

4.3.4 CHAR的行结果存储

在多字节多字符的情况下，CHAR和VARCHAR的实际行存储基本是没有区别的。

4.4 InnoDB数据页结构

由七个部分组成，如下：

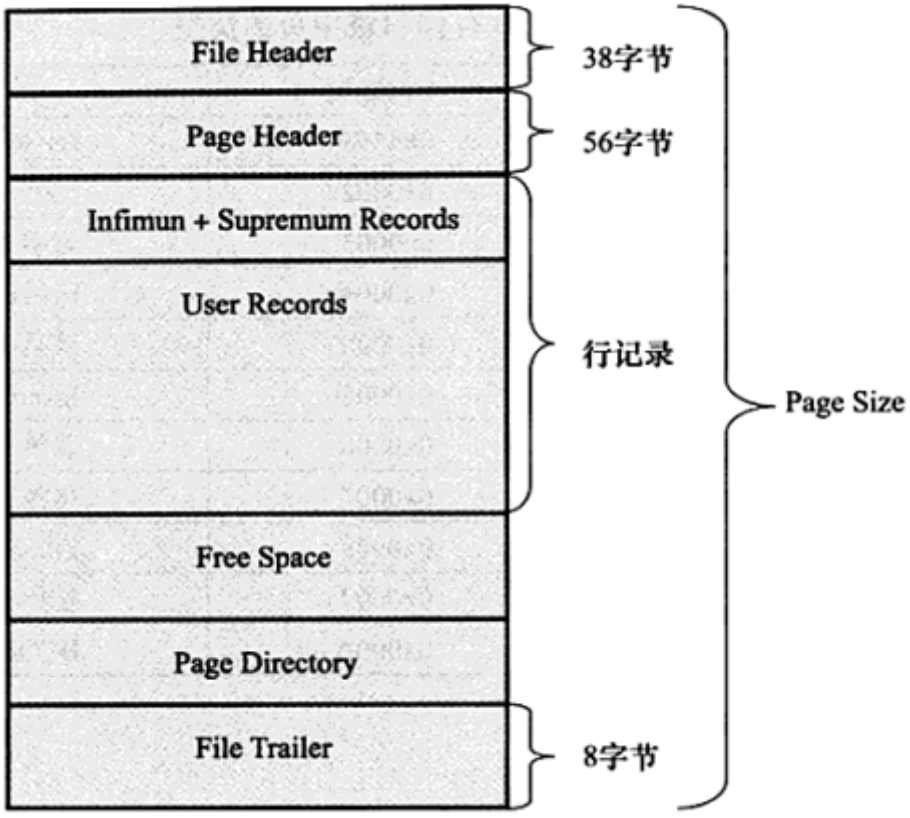


图 4-6　InnoDB 存储引擎数据页结构

名称	字节数	作用
File Header（文件头）	38	记录页的一些头信息
Page Header（页头）	56	记录数据页的状态信息
Infimum和Supremum Record	--	虚拟的行记录，用来限定记录的边界
User Record（用户记录，即行记录）	--	实际存储行记录内容
Free Space（空闲空间）	--	一条记录被删除后，该空间会被加入到空闲列表中
Page Directory（页目录）	--	存放记录的相对地址
File Trailer（文件结尾信息）	8	检测页是否已经完整地写入磁盘

4.4.1 Infimum和Supremum Record

虚拟的行记录，用来限定记录的边界。

Infimum记录是比该页中任何主键值都要小的值，Supremum指比任何可能大的值还要大的值。这两个值在页创建时被建立，并且在任何情况下不会被删除。

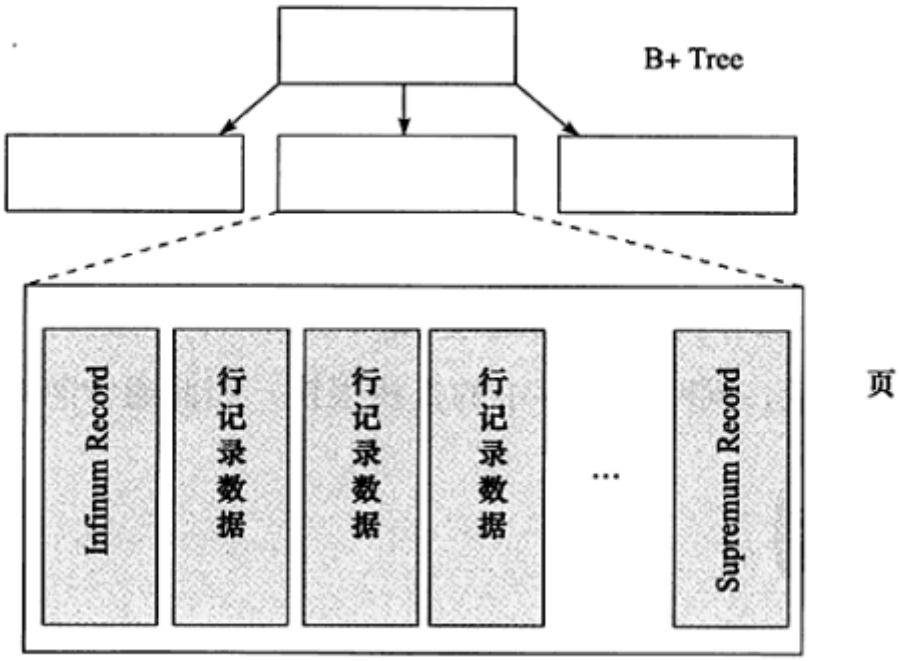
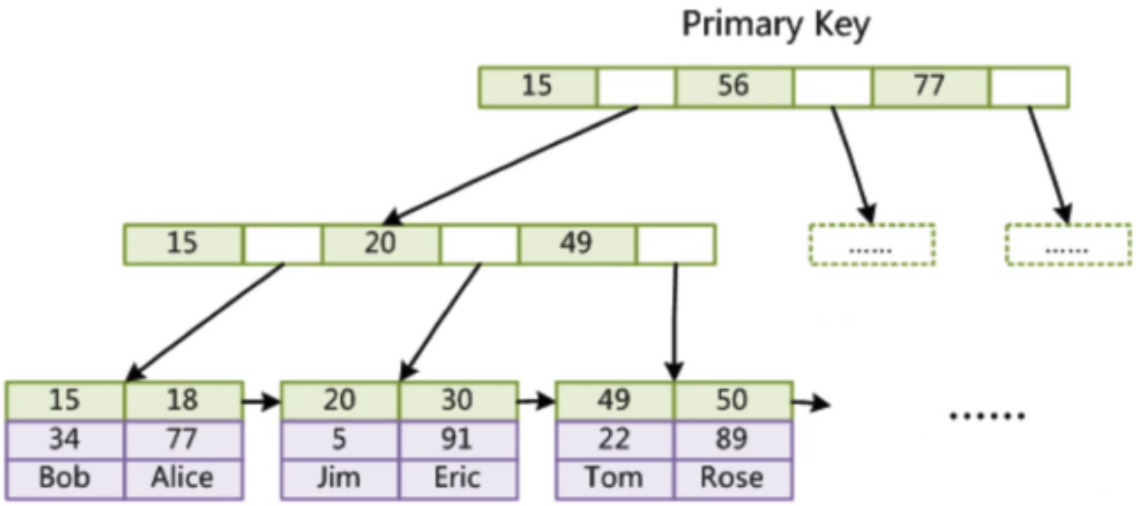


图 4-7　Infinum 和 Supremum Record

4.4.2 Page Directory

存放记录的相对位置（注意，这里存放的是页相对位置，而不是偏移量），有时候这些记录指针称为Slots(槽)或者目录槽(Directory Slots)。一个槽中可能有多个记录。由于Page Directory是稀疏目录，二叉查找的结果只是一个粗略的结果，因此必须通过record header中的next_record来继续查找相关记录。

如下图，15，18组成一个槽，找到槽之后，还需要通过指针找到记录。



5. 索引与算法

5.1 概述

InnoDB支持以下几种常见的索引：

- B+树索引
- 全文索引
- 哈希索引，自适应的，不能人为干预是否在一张表中生成哈希索引

注意：

B+树索引并不能找到一个给定键值的具体行，能找到的只是被查找数据所在的页。然后数据库通过把页读入内存，再在内存中进行查找，最后得到要查找的数据。

5.2 数据结构与算法

粗略的介绍一下，具体的请查看相关书籍。

5.2.1 二分查找法

用来查找一组**有序的**记录数组中的某一记录。

如有5、10、19、21、31、37、42、48、50、52这10个数，现要从这10个数中查找48这条记录，其查找过程如下：

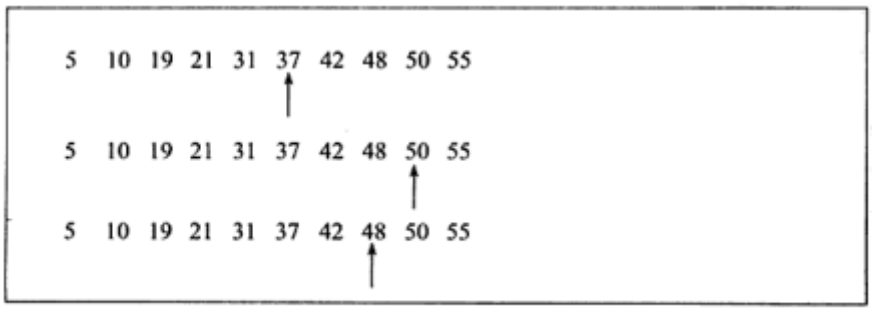


图 5-1　二分查找法

每页Page Directory中的槽是按照主键的顺序进行存放的，对应某一条具体记录的查询是通过对Page Directory进行二分查找得到的（槽前面的博客介绍过）。

5.2.2 二叉查找树和平衡二叉树

B+树是通过二叉查找树，再由平衡二叉树，B树演化而来的。

二叉查找树的特点：左子树的键值总是小于根的键值，右子树的键值总是大于根的键值，如图5-2。同时，二叉树可以任意构造，所以它可能会按照5-3的方式建立二叉树，这时该二叉树几乎退化成链表，查询效率就降低了。

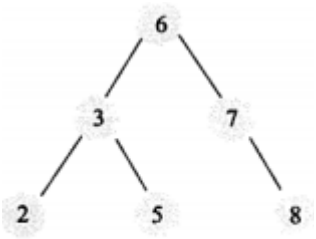
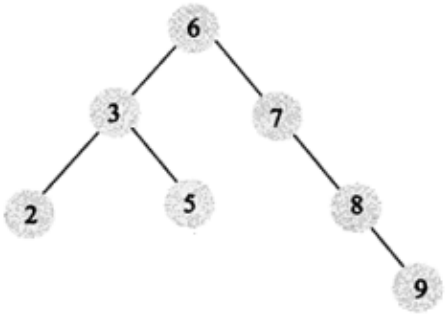


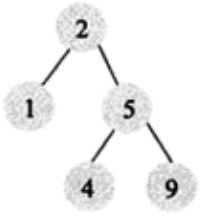
图 5-2　二叉查找树

图 5-3　效率较低的一棵二叉查找树

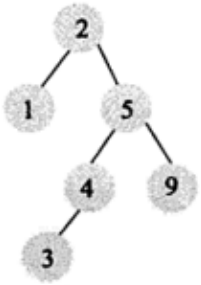
为解决这种情况，引入了平衡二叉树，即：首先满足二叉查找树的定义，其次必须满足任何节点的两个子树的高度最大差为1。要满足这个条件，就必须对二叉树进行左旋或者右旋操作来使树保持平衡，有些时候可能需要旋转多次，如图所示。



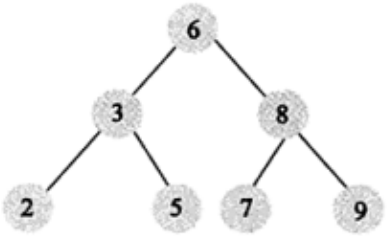
插入新值9



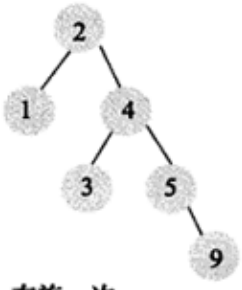
平衡二叉树



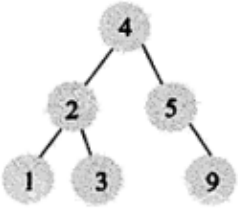
插入新键值3



左旋以保证平衡



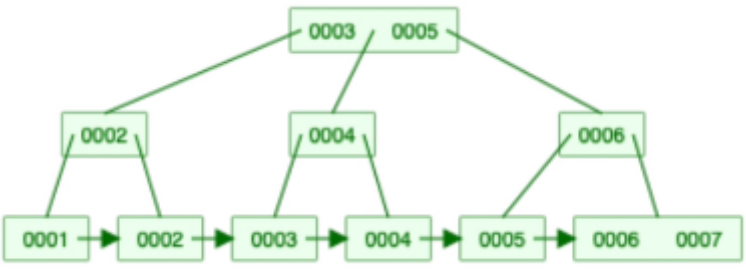
右旋一次



再左旋一次

5.2.3 B+树

首先来看一棵由数字1-7组成的B+树。



可以看到B+树有以下特点：

- 所有记录都在叶子节点上，并且是顺序存放的
- 非叶子节点冗余了部分叶子节点的数据
- 节点之间通过链表进行链接
- 一个节点可以存储多个值

B+树的插入和删除操作是通过对B+树的**拆分或者合并**来保持树的平衡，由于篇幅有限，这里不进行详细说明，感兴趣的可以查看《MySQL技术内幕 InnoDB存储引擎 第2版》。

5.3 B+树索引

B+树索引的本质就是B+树在数据库中的实现，在数据库中，B+树的高度都一般在2~4层，也就是说查找某一键值的行记录最多只需要2到4次IO。当前一般机械磁盘每秒至少可以做100次IO，2~4次的IO意味这查询时间只需0.02~0.04秒。

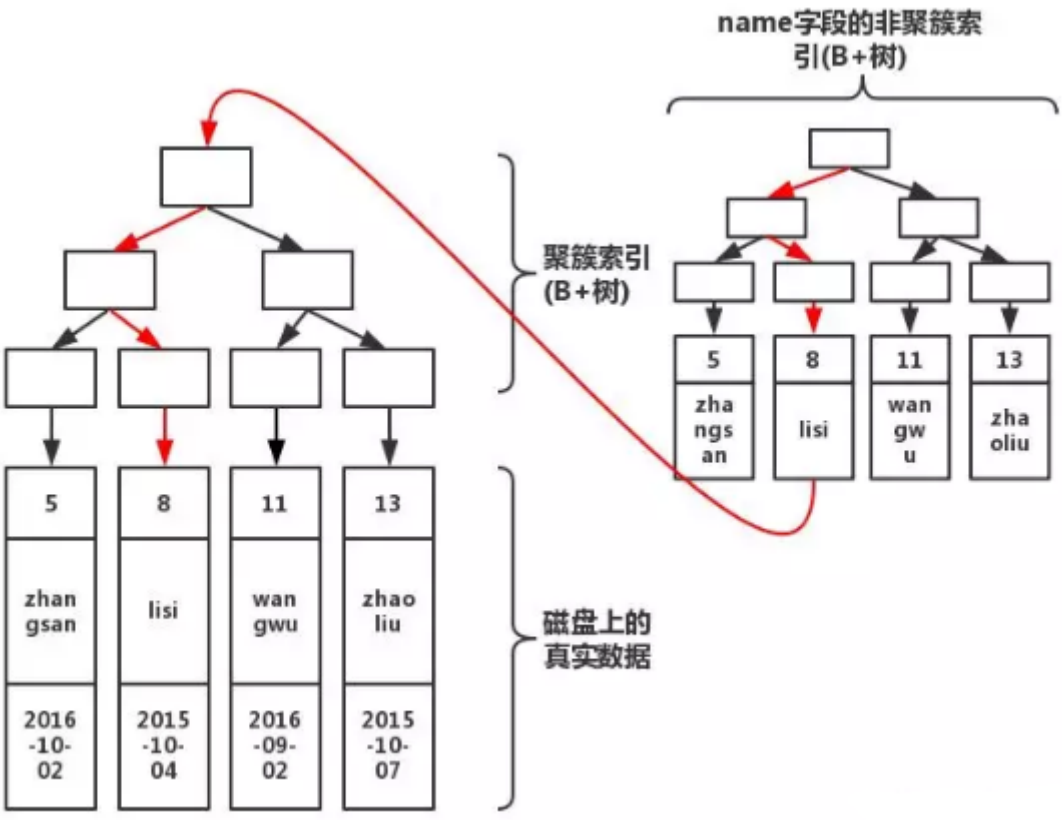
数据库中的B+树索引可以分为**聚集索引(clustered index)和辅助索引(secondary index)**，但是不管是聚集还是辅助的索引，其内部都是B+树，即高度平衡的，叶子节点存放着所有的数据。聚集索引与辅助索引不同的是，叶子节点存放的是否是一整行的信息。

5.3.1 聚集索引和非聚集索引

InnoDB存储引擎表是索引组织表，表中数据是按照主键顺序存放的。聚集索引就是按照每张表的主键构造一棵B+树，同时**叶子节点存放的即为整张表的行记录数据**，也将聚集索引的叶子节点称为数据页。每个数据页都通过一个双向链表来进行链接。

非聚集索引（也称辅助索引），其叶子节点并不包含行记录的全部数据，存放的仅是主键和索引键。创建的索引，如联合索引、唯一索引等，都属于非聚簇索引。

其关系如下图，图片来自于网络。



当通过辅助索引来查找数据时，innodb存储引擎会通过辅助索引叶子节点获得主键索引的主键，然后再通过主键索引找到完整的行记录。

例如在一棵高度为3的辅助索引树中查找数据，那需要对这颗辅助索引树进行3次IO找到指定主键，如果聚集索引树的高度同样为3，那么还需要对聚集索引树进行3次查找，最终找到一个完整的行数据所在的页，因此一共需要6次IO访问来得到最终的数据页。

5.3.2 B+树索引的使用

联合索引

联合索引指对表上的多个列进行索引。

首先创建一张表，并且索引列为(a, b)

```
create table t (
    a int,
    b int,
    primary key (a),
    key idx_a_b (a, b)
)engine=innodb;
```

向表中插入一些数据，先看一下联合索引内部的结构。可以看出数据按(a, b)的顺序进行了存放，a的顺序：1,1,2,2,3,3，b的顺序1,2,1,4,1,2，乍看是无序的，但当a值相同时，b是有序的。

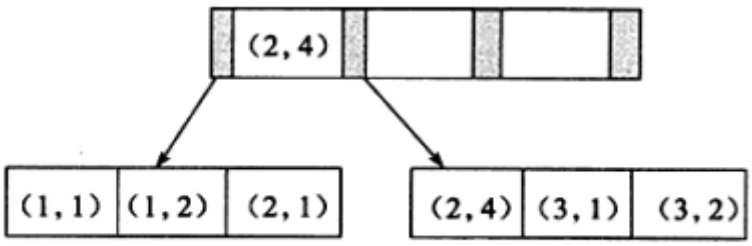


图 5-22　多个键值的 B+ 树

查询的时候，就涉及到了一个非常重要的知识点：**最佳左前缀**

例如查询：

1. `select * from t where a = 1 and b = 2`，a有序，b有序，显然可以使用(a, b)这个联合索引
2. `select * from t where a=1`，a有序，也可以使用(a, b)索引
3. `select * from t where b = 3`，b整体上是无序的，所以用不到(a, b)索引

注意：

因为我们查询的是所有字段，而辅助索引叶子节点仅存储了**索引值和主键值**，即`(primary key1, primary key2, ..., key1, key2, ...)`，这个时候需要通过主键id回表查询，补充字段。在某些情况下可以使用**覆盖索引**来避免回表查询。

覆盖索引

覆盖索引，即从辅助索引中就可以得到查询的记录，而不需要查询聚集索引中的记录。好处是覆盖索引不包含整行记录的所有信息，故其大小要远小于聚集索引，因此可以减少大量的IO操作。

覆盖索引叶子节点存放的数据为`(primary key1, primary key2, ..., key1, key2, ...)`，因此，以下查询都可以仅使用一次辅助联合索引来完成查询。

```
select key2 from table where key1 = xxx;
select primary key2, key2 from table where key1 = xxx;
select primary key1, key2 from table where key1 = xxx;
select primary key2, primary key2, key2 from table where key1 = xxx;
```

优化

1. 索引提示
   - **USE INDEX**，告诉优化器可以选择该索引，实际上优化器还是会根据自己的判断进行选择。如：`select * from t USE INDEX(a) where a = 1 and b = 2;`
   - **FORCE INDEX**，强制指定某个索引来完成查询。如：`select * from t FORCE INDEX(a) where a = 1 and b = 2;`
2. **Multi-Range Read（MRR）**，根据索引进行查询优化，减少磁盘的随机访问，并且将随机访问转化为较为顺序的数据访问，可适应于range，ref，eq_ref类型的访问。在查询辅助索引时，首先根据得到的查询结果，按照主键进行排序，并按照主键排序的顺序进行书签查找。
3. **Index Condition Pushdown（ICP）**，根据索引进行查询优化，当进行索引查询时，首先根据索引来查找记录，然后再根据where条件来过滤数据

执行顺序：from，on，outer join，where，group by，having，select，order by，limit

5.4 B+树索引的分裂

B+树索引页的分裂并不总是从页的中间开始，若插入是顺序的，从中间记录分裂会导致页空间的浪费。若插入是随机的，则取中间记录作为分裂点的记录。

如下图，分裂点为插入记录本身，向右分裂后仅插入记录本身，这在自增插入时是普遍存在的一种情况。

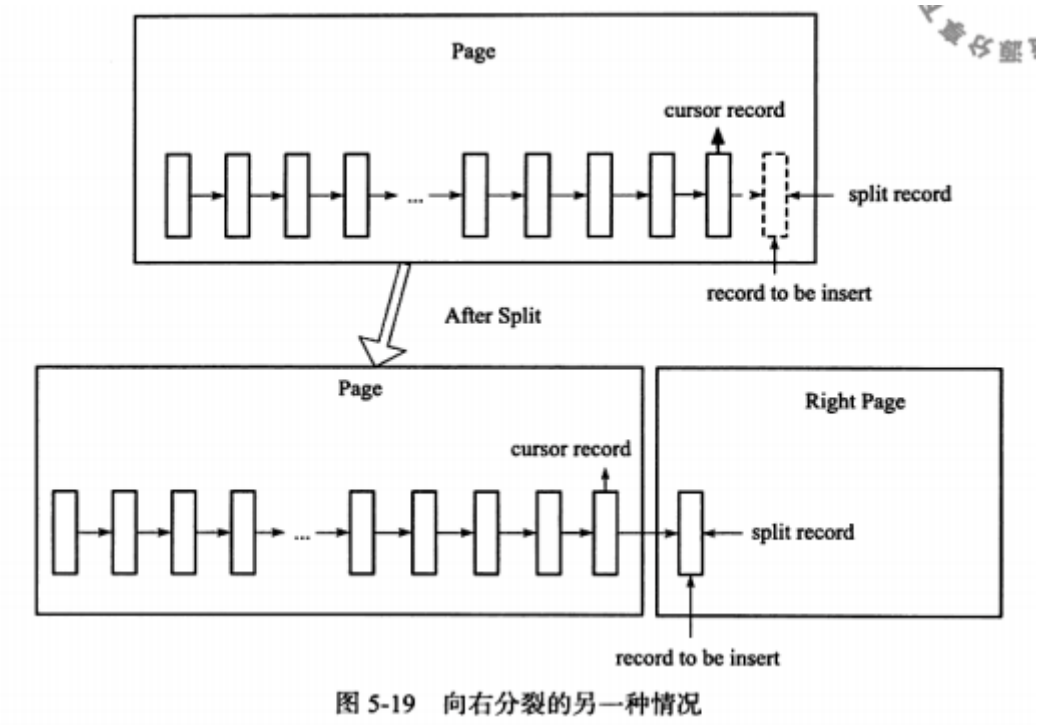


图 5-19　向右分裂的另一种情况

5.5 Cardinality值

还是之前的表t，通过`show index from t`来查看表中索引信息。

```
mysql> show index from t\G;
*************************** 1. row ***************************
        Table: t
   Non_unique: 0
     Key_name: PRIMARY
 Seq_in_index: 1
  Column_name: a
```

```
    Collation: A
  Cardinality: 0
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_comment:
*************************** 2. row ***************************
        Table: t
   Non_unique: 1
     Key_name: idx_a_b
 Seq_in_index: 1
  Column_name: a
    Collation: A
  Cardinality: 0
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_comment:
*************************** 3. row ***************************
        Table: t
   Non_unique: 1
     Key_name: idx_a_b
 Seq_in_index: 2
  Column_name: b
    Collation: A
  Cardinality: 0
     Sub_part: NULL
       Packed: NULL
         Null: YES
   Index_type: BTREE
      Comment:
Index_comment:
3 rows in set (0.00 sec)
```

Cardinality表示选择性，建立索引的前提是列中的数据都是高选择性的，因此该值非常关键，表示索引中不重复记录数量的**预估值**，在实际应用中，该值应尽可能地接近1（**有点问题，应该是值越大，选择性越高?**），如果非常小，那么用户需要考虑是否还有必要创建这个索引。

如何对Cardinality进行统计呢?

如果每次索引在发生操作时就对其进行Cardinality的统计，那么将会给数据库带来很大的负担。所以，数据库对于Cardinality的统计都是通过采样的方法来完成的。

采样过程如下:

1. 取得B+树索引中叶子节点的数量，记为A
2. 随机取得B+树索引中的8个叶子节点。统计每个页不同记录的个数，即为P1，P2，...，P8
3. 根据采样信息给出Cardinality的预估值：Cardinality=(P1+P2+...+P8)*A/8

Cardinality更新策略为:

1. 表中1/16的数据已发生过变化
2. 某一行发生变化的次数大于2,000,000

3. 当执行SQL语句ANALYZE TABLE、SHOW TABLE STATUS、SHOW INDEX以及访问INFORMATION_ SCHEMA架构下的表TABLES和STATISTICS时，会去更新Cardinality值。若表中的数据量非常大，并且表中存在多个辅助索引时，执行上述这些操作可能会非常慢。虽然用户可能并不希望去更新Cardinality值。

5.6 全文索引

全文检索(Full-Text Search)是将存储于数据库中的整本书或整篇文章中的任意内容信息查找出来的技术。它可以根据需要获得全文中有关章、节、段、句、词等信息，也可以进行各种统计和分析。

通过倒排索引来实现。

6. 锁 - 实现事务的隔离性

推荐看书，书中有很多例子，看起来也容易理解些。

6.1 什么是锁

锁是数据库区别于文件系统的一个关键特性。锁机制用于管理对共享资源的并发访问。例如：操作缓冲池中的LRU列表、删除、添加、移动LRU列表中的元素，为了保证一致性，必须有锁的介入。

InnoDB存储引擎锁的实现和Oracle数据库非常类似，提供一致性的非锁定读、行级锁支持。行级锁没有相关额外的开销，并可以同时得到并发性和一致性。

6.2 lock和latch

latch一般称为闩锁（轻量级的锁），因为其要求锁定的时间必须非常短。若持续的时间长，则应用的性能会非常差。在InnoDB存储引擎中，latch又可以分为mutex（互斥量）和rwlock（读写锁）。其目的是用来保证并发线程操作临界资源的正确性，并且通常没有死锁检测的机制。 lock的对象是事务，用来锁定的是数据库中的对象，如表、页、行。并且一般lock的对象仅在事务commit或rollback后进行释放（不同事务隔离级别释放的时间可能不同）。有死锁检测机制。

1. 查看latch信息：`show engine innodb mutex`

```
mysql> show engine innodb mutex\G;
*************************** 1. row ***************************
  Type: InnoDB
  Name: sum rwlock: buf0buf.cc:781
Status: waits=1
1 row in set (0.00 sec)
```

2. 查看lock信息：通过命令`SHOW ENGINE INNODB STATUS`及information_schema架构下的表`INNODB_TRX（显示当前运行的InnoDB事务）`、`INNODB_LOCKS（锁的情况）`、`INNODB_LOCK_WAITS（事务的等待信息）`来观察锁的信息。
3. lock与latch的比较

	lock	latch
对象	事务	线程
保护	数据库内容	内存数据结构
持续时间	整个事务过程	临界资源
模式	行锁、表锁、意向锁	读写锁、互斥量
死锁	通过 waits-for graph、time out 等机制进行死锁检测与处理	无死锁检测与处理机制。仅通过应用程序加锁的顺序（lock leveling）保证无死锁的情况发生
存在于	Lock Manager 的哈希表中	每个数据结构的对象中

6.3 InnoDB存储引擎中的锁

6.3.1 锁的类型

实现了两种标准的行级锁：

- 共享锁（S Lock），允许事务读一行数据
- 排他锁（X Lock），允许事务删除或者更新一行数据

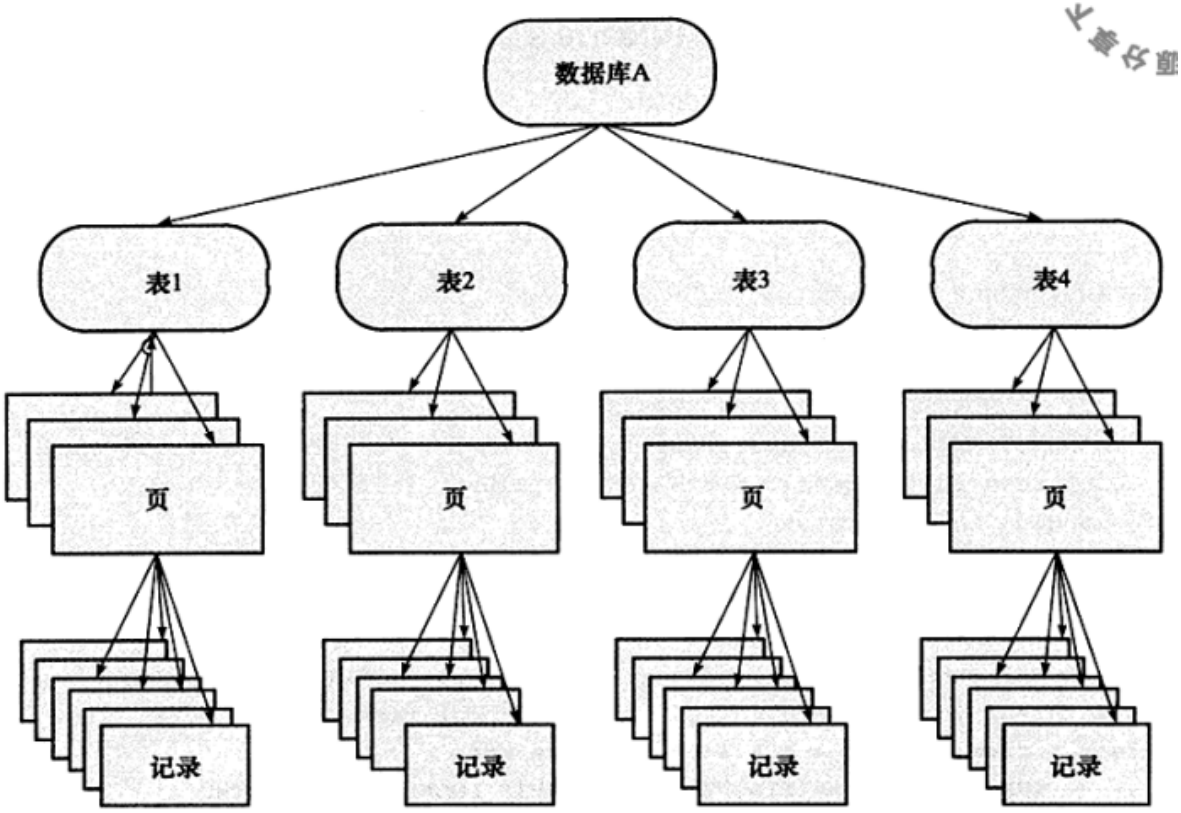
排他锁和共享锁的兼容性：

- X锁与任何锁都不兼容
- S锁仅和S锁兼容

例：如果事务T1已经获得了行r的共享锁，那么另外的事务T2可以立即获得行r的共享锁，若有其他事务T3想要获取行r的排他锁，则必须等待T1，T2释放行r上的共享锁。

InnoDB支持多粒度锁定，允许事务在行级上和表级上的锁同时存在，称之为意向锁。意向锁将锁定的对象分为多个层次，意味着事务希望在更细粒度上锁。

若将上锁的对象看成一棵树，那么对最下层的对象上锁，也就是对最细粒度的对象进行上锁，那么首先需要对粗粒度的对象上锁。如：需要对页上的记录r行上X锁，那么分别需要对数据库A、表、页上意向锁IX，最后对记录r上X锁。若其中任何一个部分导致等待，那么该操作需要等待粗粒度锁的完成。

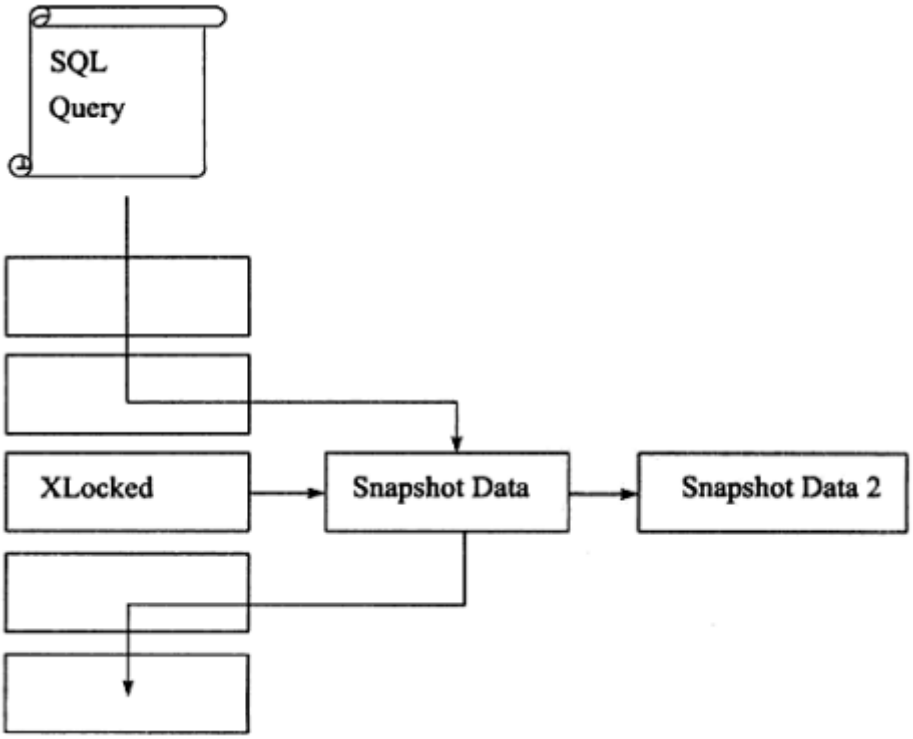


由于InnoDB存储引擎是行级别的锁，因此**意向锁不会阻塞除全表扫描以外的任何请求。**InnoDB存储引擎中锁的兼容性。

	IS	IX	S	X
IS	兼容	兼容	兼容	不兼容
IX	兼容	兼容	不兼容	不兼容
S	兼容	不兼容	兼容	不兼容
X	不兼容	不兼容	不兼容	不兼容

6.3.2 一致性非锁定读

一致性非锁定读指InnoDB存储引擎通过行多版本控制的方式来读取当前执行时间数据库中的行数据。如果读取的行正在执行DELETE或者UPDATE操作，这时读取操作就不会因此去等待行上的锁释放。相反地，会去读取行的一个快照数据。



快照数据是指行之前版本的数据，每行记录可能有多个版本，该实现是通过undo段（默认在表空间中）来完成的。undo用来在事务中回滚数据，因此快照数据本身没有额外的开销。此外，读取快照数据是不需要上锁的，因为没有事务需要对历史的数据进行修改操作。

MVCC：多版本并发控制

在不同的事务隔离级别下，读取的方式不同，并不是在每个事务隔离级别下都是采用非锁定的一致性读。**READ COMMITTED，总是读取被锁定行的最新一份快照数据。REPEATABLE READ，读取事务开始时的行数据版本。**

案例：在不同的事务隔离级别下，执行结果会不同。

时间	会话 A	会话 B
1	BEGIN	
2	SELECT * FROM parent WHERE id = 1;	
3		BEGIN
4		UPDATE parent SET id=3 WHERE id = 1;
5	SELECT * FROM parent WHERE id = 1;	
6		COMMIT;
7	SELECT * FROM parent WHERE id = 1;	
8	COMMIT	

6.3.3 一致性锁定读

在某些情况下，需要显式地对数据库加锁以保证数据逻辑的一致性，即使是对于SELECT的只读操作。对于SELECT语句支持两种一致性锁定读的操作：

- `SELECT ... FOR UPDATE`，对读取的行数据加一个X锁，其他事务不能对已锁定的行加上任何锁
- `SELECT ... LOCK IN SHARE MODE`，对读取的行记录加一个S锁，其他事务可以向被锁定的行加S锁，但如果加X锁，则会被阻塞

注意：**以上两个锁定语句，必须在事务中执行。**

6.3.4 自增长与锁

在InnoDB存储引擎的内存结构中，对每个自增长的计数器的表进行插入操作时，这个计数器会被初始化，执行如下语句来得到计数器的值：`SELECT MAX(auto_inc_col) FROM t FOR UPDATE`。

插入操作会依据这个自增长的计数器值加1赋予自增长列。这个实现方式称做**AUTO-INC Locking。**这种锁其实是采用一种特殊的表锁机制，为了提高插入的性能，锁不是在一个事务完成后才释放，而是在完成对自增长值插入的SQL语句后立即释放。存在一定的性能问题，第一，对于有自增长值的列的并发插入性能较差，事务必须等待前一个插入操作的完成；其次，对于insert ... select的大数据量的插入会影响插入的性能，因为另一个事务中的插入操作会被阻塞。

从MySQL 5.1.22开始，提供了一种**轻量级互斥量**的自增长实现机制，大大提高了自增长值插入的性能。

自增长值的列必须是索引的第一个列。

6.3.5 外键和锁

对于外键值的插入或更新操作，首先需要查询父表中的记录，即SELECT父表（使用的是一致性锁定读，即：`SELECT ... LOCK IN SHARE MODE`，即主动对父表加S锁。如果这时父表上已经加了X锁，子表上的操作会被阻塞）。

6.4 锁的算法

6.4.1 行锁的三种算法

有3种行锁算法，分别是：

- Record Lock：单个行记录上的锁，没有主键，会使用隐式的主键进行锁定
- Gap Lock：间隙锁，锁定一个范围，但不包含记录本身
- Next-Key Lock：Gap Lock + Record Lock，锁定一个范围，并且锁定记录本身，其设计目的是为了解决Phantom Problem（幻象问题）

用户可以通过以下两种方式来显式地关闭Gap Lock：

- 将事务的隔离级别设置为READ COMMITTED，读已提交会出现不可重复读和幻读
- 将参数inodb_ locks_ unsafe_ for_ binlog设置为1

在上述的配置下，除了外键约束和唯一性检查依然需要的Gap Lock，其余情况仅使用Record Lock进行锁定。

当查询的索引含有**唯一属性**时，会对Next-Key Lock进行优化，将其降级为Record Lock，即仅锁住索引本身，而不是一个范围。若唯一索引由多个列组成，而查询仅是查找多个唯一索引列中的其中一个，那么查询其实是range类型查询，而不是point类型查询，依然使用Next-Key Lock进行锁定。

案例讲解。

6.4.2 解决Phantom Problem

在默认的事务隔离级别下，即RETEATABLE READ下，采用Next-Key Locking机制来避免Phantom Problem问题。在READ COMMITTED下，仅使用Record Lock。

Phantom Problem是指在同一事务下，连续执行两次同样的SQL语句可能导致不同结果，第二次SQL语句可能返回之前不存在的行，即：当前事务能够看到其他事务的结果。

如：`SELECT * FROM t WHERE a > 2 FOR UPDATE`，在默认隔离级别下，是对(2, 正无穷)这个范围加X锁，因此任何对于这个范围的插入都是不被允许的，从而避免了Phantom Problem。

6.5 锁问题

锁会带来的问题：

- 脏读
- 不可重复读
- 幻读，使用gap lock解决
- 丢失更新

6.5.1 脏读

脏页跟脏数据是两个完全不同的概念。脏页指的是缓冲池中已经被修改的页，但是还没有刷新到磁盘中。脏数据是指事务对缓冲池中的行记录的修改，并且还没有提交。

脏读就是在不同的事务下，当前事务可以读到另外事务未提交的数据（脏数据）。

Time	会话 A	会话 B
1	SET @@tx_isolation='read-ncommitted';	
2		SET @@tx_isolation='read-ncommitted';
3		BEGIN;
4		mysql> SELECT * FROM t\G; ********* 1. row ********** a: 1 1 row in set (0.00 sec)
5	INSERT INTO t SELECT 2;	
6		mysql> SELECT * FROM t\G; ********* 1. row ********** a: 1 ********* 2. row ********** a: 2 2 row in set (0.00 sec)

6.5.2 不可重复读

不可重复读是指在一个事务内多次读取同一数据集合。在这个事务还没有结束时，另外一个事务也访问该同一数据集合， 并做了一些DML操作。因此，在第一个事务中的两次读数据之间，由于第二个事务的修改，那么第一个事务两次读到的数据可能是不一样的。这样就发生了在一个事务内两次读到的数据是不一样的情况，这种情况称为不可重复读。

不可重复读和脏读的区别是：脏读是读到未提交的数据，而不可重复读读到的却是已经提交的数据，但是其违反了数据库事务一致性的要求。

Time	会话 A	会话 B
1	SET@@tx_isolation='read-committed';	
2		SET @@tx_isolation='read-committed';
3	BEGIN	BEGIN
4	mysql>SELECT * FROM t ; ********* 1. row ********** a: 1 1 row in set (0.00 sec)	
5		INSERT INTO t SELECT 2;
6		COMMIT;
7	mysql>SELECT * FROM t ; ********* 1. row ********** a: 1 ********* 1. row ********** a: 2 2 row in set (0.00 sec)	

一般来说，不可重复读的问题是可以接受的，因为其读到的是已经提交的数据，本身不会带来很大的问题。

采用Next-Key Lock算法，避免不可重复读的现象。对于索引的扫描，不仅锁住扫描到的索引，而且还锁住这些索引覆盖的范围，因此在这个范围内的插入都是不允许的。对于update，例如：两个事务，在第一个事务中执行范围查询：`select * from t where t > 5 for update;`，在另一个事务中执行`update t set a = 1 where a > 5`，该事务会被阻塞，等待上一个事务提交。

6.5.3 丢失更新

一个事务的更新操作会被另一个事务的更新操作所覆盖，从而导致数据的不一致。例如：

1. 事务T1将行记录r更新为v1，但是事务T1并未提交
2. 与此同时，事务T2将行记录r更新为v2（阻塞），事务T2未提交
3. 事务T1提交
4. 事务T2提交

以上操作理论上不会导致丢失更新，但是如果是先查询，再在查询的基础上更新，就会产生逻辑意义的丢失更新问题，设想银行丢失更新的情况.....

解决：让事务在这种情况下的操作变成串行化，而不是并行操作，即读取时也加锁，`select ... for update`

6.6 阻塞

因为不同锁之间的兼容性关系，在有些时刻一个事务中的锁需要等待另一个事务中的锁释放它所占有的资源。

`innodb_lock_wait_timeout`用来控制等待的时间（默认50秒）；`innodb_rollback_on_timeout`用来设定是否在等待超时时对进行中的事务进行回滚操作（默认off，不回滚），InnoDB不会回滚大部分的错误异常，但死锁除外。

6.7 死锁

死锁是指在两个或两个以上的事务在执行过程中，因争夺资源而造成的一种相互等待的现象。若无外力作用下，事务都将无法推进下去。

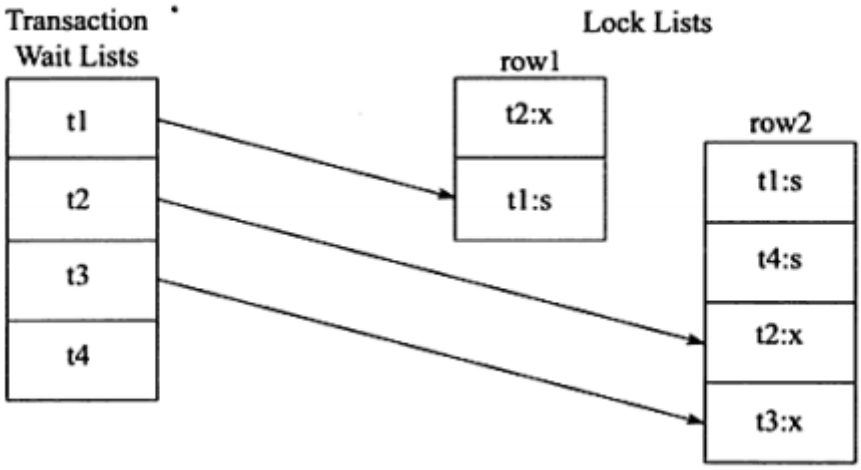
死锁检测：**超时机制+wait-for graph（等待图）**

wait-for graph要求数据库保存以下两种信息：

1. 锁的信息链表
2. 事务等待链表

通过上述链表可以构造出一张图，若图中存在回路，就代表存在死锁，资源间相互发生等待。

如图，存在t1和t2相互等待。



通常来说InnoDB存储引擎选择回滚undo量最小的事务。

死锁发生的机率是非常小的。

死锁示例，AB-BA死锁，1213错误，不需要对其进行回滚。

时间	会话 A	会话 B
1	BEGIN;	
2	mysql>SELECT * FROM t WHERE a = 1 FOR UPDATE; ********* 1. row ********** a: 1 1 row in set (0.00 sec)	BEGIN
3		mysql>SELECT * FROM t WHERE a = 2 FOR UPDATE; ********* 1. row ********** a: 2 1 row in set (0.00 sec)
4	mysql>SELECT * FROM t WHERE a = 2 FOR UPDATE; # 等待	
5		mysql>SELECT * FROM t WHERE a = 1 FOR UPDATE; ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

6.8 锁升级

锁升级是将当前锁的粒度降低，即：行锁升级为页锁，或者将页锁升级为表锁。若在数据库的设计中认为锁是一个稀缺资源，而且想避免锁的开销，那数据库中会频繁出现锁升级的现象。

InnoDB不存在锁升级问题。

7. 事务

7.1 概述

7.1.1 ACID

1. **原子性**：要么都做，要么都不做
2. **一致性**：数据库从一种一致性状态转变为下一种一致性状态
3. **隔离性**：每个读写事务的对象对其它事务的操作对象能相互分离，即：该事务提交前对其他事务都不可见
4. **持久性**：事务一旦提交，其结果就是永久性的

7.1.2 分类

1. **扁平事务**

最简单，所有操作都处于同一层次，由begin开始，由commit或者rollback结束，其间操作都是原子的

2. **带有保存点的扁平事务**

   允许在事务执行过程中回滚到同一事务中较早的一个状态

保存点：通知系统应该记住事务当前的状态，以便当之后发生错误时，事务能回到保存点当时的状态；使用 `SAVE WORK` 函数建立，通知系统记录当前的处理状态；保存点在事务内部单调递增

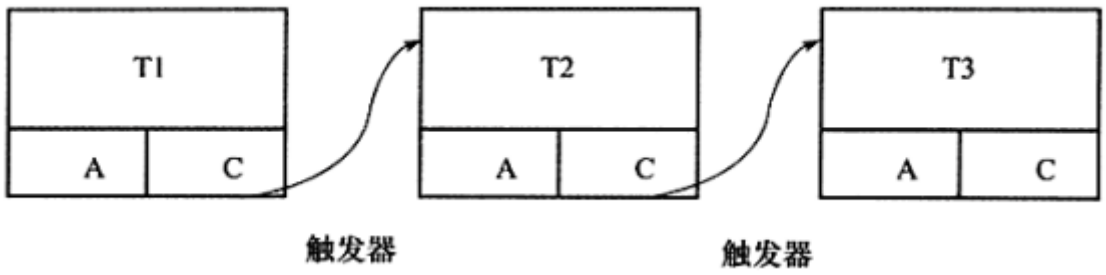
当系统发生崩溃时，所有的保存点都将消失。当进行恢复时，事务需要从最开始处重新执行，而不能从最近的一个保存点继续执行

案例：出门旅游

3. **链事务**

   可视为保存点模式的一种变种。

在提交一个事务时，释放不需要的数据对象，将必要的处理上下文隐式地传给下一个要执行的事务。注意，提交事务操作和开始下一个事务操作将合并为一个原子操作，也就是下一个事务将看到上一个事务的结果。

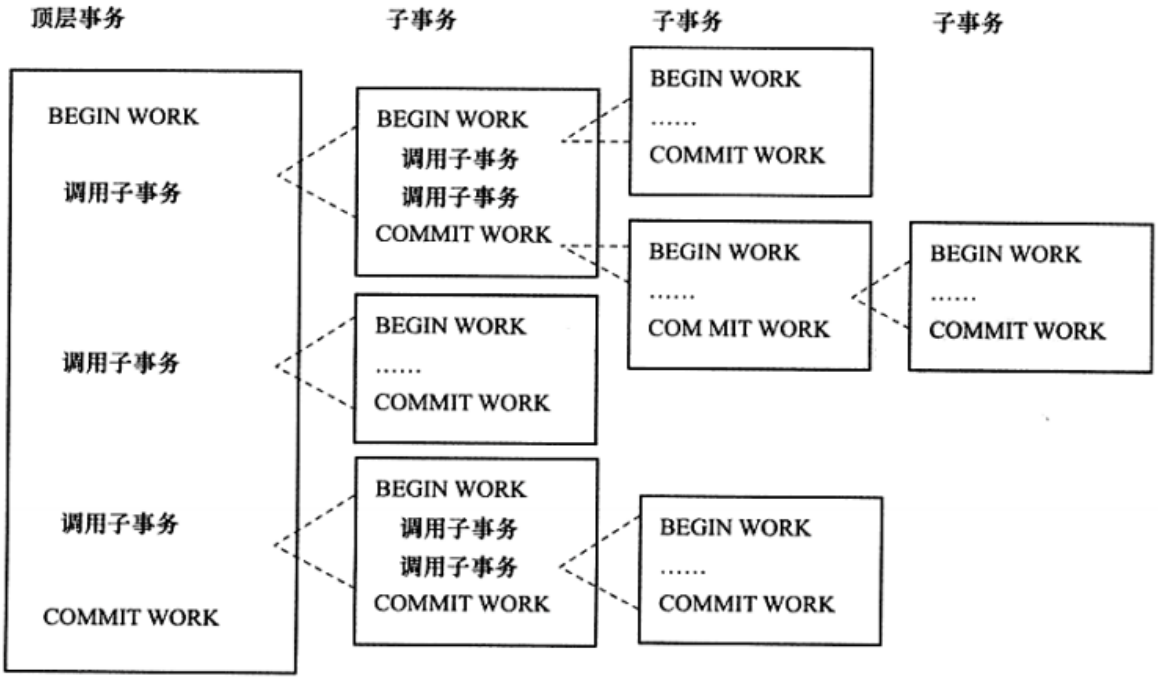


回滚仅限于当前事务，即只能恢复到最近的一个保存点

4. **嵌套事务**（不原生支持）

   是一个层次结构框架，由一个顶层事务控制着各个层次的事务

   所有子事务都在顶层事务提交后才真正的提交，任意事务的回滚都会引起它的所有子事务的回滚



5. **分布式事务**

   在分布式环境下运行的扁平事务，需要根据数据所在位置访问网络中的不同节点

   案例：银行转账，从一张卡转到另外一张卡

7.2 事务的实现

1. 隔离性由锁来实现
2. 原子性和持久性通过redo（重做日志）来实现
3. 一致性通过undo（回滚日志）来实现，事务回滚，MVCC功能

7.2.1 redo

基本概念

1. 重做日志缓存和重做日志文件
2. 顺序写
3. fsync操作：重做日志缓存 -> 文件系统缓存 -> 文件系统，因此，磁盘的性能决定了事务提交的性能，即数据库性能
4. 重做日志刷新到磁盘的策略，事务提交时、master thread每秒操作或者事务提交时，将重做日志写入文件系统缓存
5. 与二进制日志（**进行POINT IN TIME的恢复及主从复制的建立**）的区别

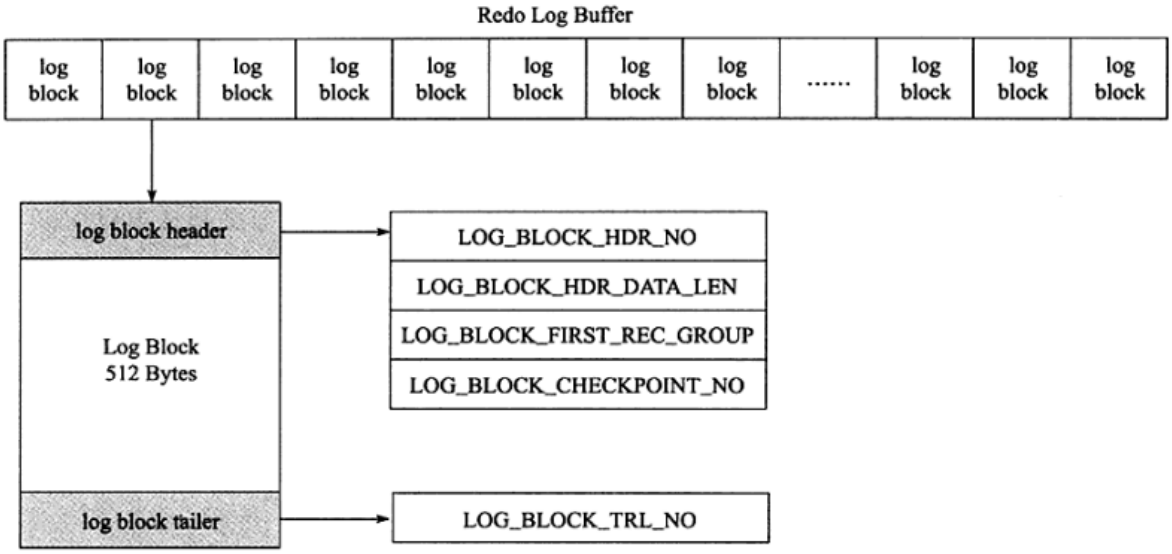
log block

重做日志缓存、重做日志文件都是以块（block）的方式进行保存的，称之为重做日志块（redo log block），每块的大小为512字节。

若一个页中产生的重做日志数量大于512字节，那么需要分割为多个重做日志块进行存储。

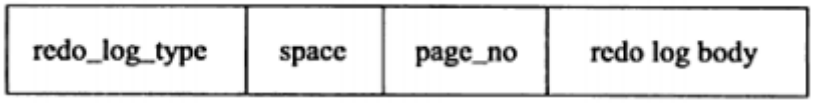
由于重做日志块的大小和磁盘扇区大小一致，都是512字节，因此重做日志的写入可以保证原子性，不需要doublewrite技术。

重做日志块的结构：日志块头（12字节）、日志（492字节）、日志块尾（8字节）



重做日志格式

重做日志格式是基于页的，虽然有着不同的重做日志格式（InnoDB 1.2 有51种），但是它们有着通用的头部格式。



通用的头部格式由以下3部分组成：

1. redo_log_type：重做日志的类型
2. space：表空间的ID
3. page_no：页的偏移量

LSN（Log Sequence Number）

日志序列号，占8个字节，并且单调递增，其表示的含义有：

1. 重做日志写入的**字节**的总量

LSN=1000；事务1写入100字节，LSN=1100；事务2写入200字节，LSN=1300

LSN存在于重做日志和每个页中，在每个页的头部，有一值FIL_PAGE_LSN，记录了该页的LSN，表示该页最后刷新时LSN的大小，可以用来判断页是否需要进行恢复操作，若在页中的LSN=1000，数据库启动时，LSN=1300，并且该事务已经提交，那么需要进行恢复操作，反之，则不需要。

通过`show engine innodb status`查看LSN的情况。

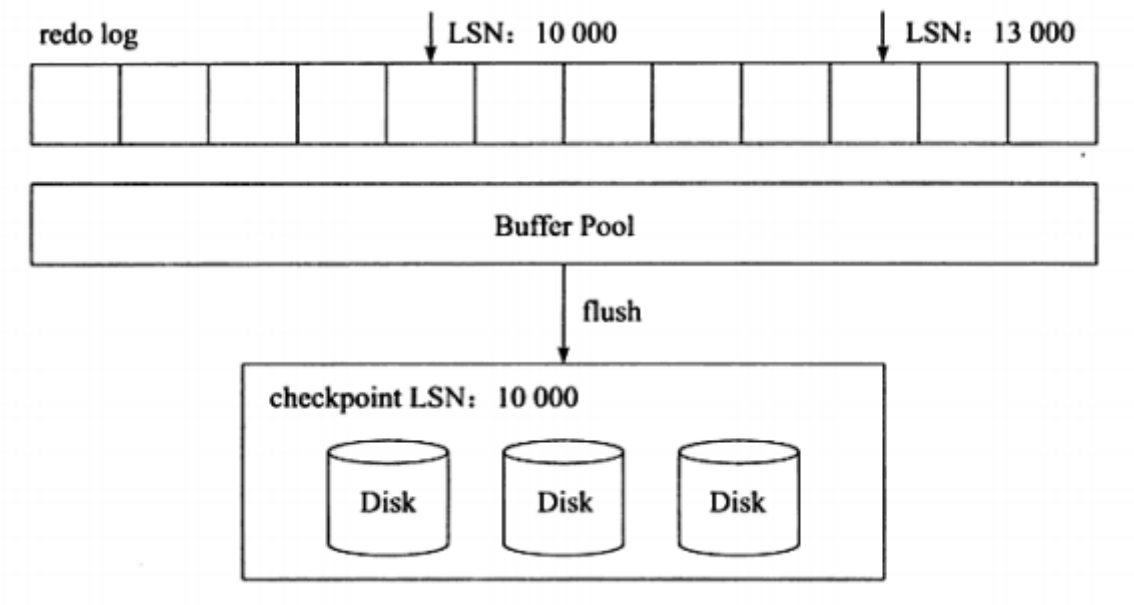
```
LOG
---
Log sequence number 381727521755    -- 当前的LSN
Log flushed up to   381727521755    -- 刷新到重做日志文件的LSN
Pages flushed up to 381727306352    -- 当前最旧的脏页数据对应的LSN
Last checkpoint at  381727304797    -- 刷新到磁盘的LSN
0 pending log flushes, 0 pending chkp writes
11035274 log i/o's done, 0.98 log i/o's/second
```

2. checkpoint的位置
3. 页的版本

恢复

重做日志记录的是物理日志，因此恢复速度比逻辑日志，如二进制日志，要快很多。

checkpoint机制表示已经刷新到磁盘页上的LSN，因此在恢复过程中仅需要恢复checkpoint开始的日志部分。如，当数据库在checkpoint的LSN为10,000发生宕机，恢复操作仅恢复LSN 10,000 ~ 13,000范围内的日志。



重做日志是幂等的，即：`f(f(x)) = f(x)`

案例：insert操作

7.2.2 undo

基本概念

可以被重用

作用：

1. **回滚**
2. **MVCC**

对数据库进行修改时，不仅会产生redo，还会产生一定量的undo，利用undo信息可以将数据回滚到修改之前的样子。

undo存放在数据库内部的一个特殊段（segment）中，这个段称为undo段，位于共享表空间中。

存储的是逻辑日志，即SQL语句。回滚时，做的是与之前相反的工作，对于每个insert，InnoDB存储引擎会执行一个delete；对于每个delete，会执行一个insert；对于每个update，会执行一个相反的update，将修改前的行放回去。

undo log会产生redo log，也就是undo log的产生会伴随着redo log的产生，这是因为undo log也需要持久性的保护。

存储管理

InnoDB存储引擎有rollback segment，每个回滚段中记录了1024个undo log segment，而在每个undo log segment段中进行undo页的申请。

InnoDB 1.1版本开始，支持最大128个rollback segment，支持同时在线事务的限制提高到了128*1024。

从InnoDB 1.2开始，可通过参数对rollback segment做进一步的设置。这些参数包括：

1. **innodb_undo_directory**：文件所在位置，可以存放在独立表空间中，默认为'.'，表示当前InnoDB存储引擎目录
2. **innodb_undo_logs**：rollback segment文件的个数
3. **innodb_undo_tablespaces**：构成rollback segment文件的数量

```
mysql> SHOW VARIABLES LIKE 'innodb_undo%' ;
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| innodb_undo_directory    | .     |
| innodb_undo_logs         | 128   |
| innodb_undo_tablespaces | 0     |
+--------------------------+-------+
3 rows in set (0.01 sec)
```

事务提交后并不能马上删除undo log及undo log所在的页。这是因为可能还有其他事务需要通过undo log来得到行记录之前的版本。故事务提交时将undo log放入一个链表中，是否可以最终删除undo log及undo log所在页由purge线程来判断。

格式

1. **insert undo log**

指在insert操作中产生的undo log。因为insert操作的记录，只对事务本身可见，对其他事务不可见，故该undo log可以在事务提交后直接删除，不需要进行purge操作。

2. **update undo log**

对delete和update操作产生的undo log，该undo log可能需要提供MVCC机制，因此不能在事务提交时就进行删除。

格式图

purge

用于最终完成delete和update操作。

group commit

磁盘fsync的性能是有限的，为了提高磁盘效率，当前数据库都提供了group commit的功能，即**一次fsync可以刷新确保多个事务日志被写入文件**。对于InnoDB存储引擎来说，事务提交时会进行两个阶段的操作：

1. 修改内存中事务对应的信息，并且将日志写入重做日志缓冲
2. 调用fsync确保日志都从重做日志缓冲写入磁盘

在InnoDB 1.2版本之前，在开启二进制日志后，group commit功能会失效，从而导致性能的下降。MySQL 5.6采用Binary Log Group Commit（BLGC）解决。

7.3 事务控制语句

1. start transaction|begin
2. commit
3. rollback

4. savepoint identifier
5. release savepoint identifier
6. rollback to [savepoint] identifier
7. set transaction

7.4 隐式提交的SQL语句

1. **DDL语句**：alter、create、drop、rename table、truncate table（不能被回滚）
2. **用来隐式地修改MySQL架构的操作**：create user、drop user、grant、rename user、revoke、set password
3. **管理语句**：analyze table、cache index、check table、load index into cache、optimize table、repair table

7.5 不好的事务习惯

默认配置下，MySQL数据库总是自动提交的

1. 在循环中提交
2. 使用自动提交
3. 使用自动回滚
4. 长事务，执行时间长的事务，可以转换为小批量的事务来进行处理

8 备份与恢复

MySQL数据库提供的大部分工具（如mysqldump、ibbackup、replication）都能很好地完成备份的工作，当然也可以通过第三方工具来完成，如xtrabacup、LVM快照备份等。

8.1 备份与恢复概述

根据备份的方法不同可以将备份分为：

1. **Hot Backup（热备）**：在数据库运行中直接备份，对正在运行的数据库没有任何影响
2. **Cold Backup（冷备）**：在数据库停止情况下，一般只需要复制相关的数据库物理文件即可
3. **Warn Backup（温备）**：在数据库运行中进行，但是会对当前数据库的操作有影响，如加一个全局读锁以保证备份数据的一致性

按照备份后文件的内容，备份文件又可以分为：

1. **逻辑备份**：备份出的文件是可读的，一般是文本文件。内容一般由一条条SQL语句，或者表内实际数据组成
2. **裸文件备份**：指复制数据库的物理文件，既可以在数据库运行中复制，也可以在数据库停止运行时直接的数据文件复制

按照备份数据库的内容来分，备份又可以分为：

1. **完全备份**：对数据库进行一个完整的备份
2. **增量备份**：在上次完全备份的基础上，对于更改的数据进行备份
3. **日志备份**：对MySQL数据库二进制日志的备份，通过对一个完全备份进行二进制日志的重做来完成数据库的point-in-time的恢复工作。数据库复制（replication）原理就是异步实时地将二进制日志重做传送并应用到从数据库

8.2 冷备

只需要备份MySQL数据的frm文件，共享表空间文件，独立表空间文件（*.ibd），重做日志文件。另外建议定期备份MySQL数据库的配置文件my.cnf，这样有利于恢复的操作。

8.3 逻辑备份

1. **mysqldump**：用来完成转存数据库的备份及不同数据库之间的移植，备份的文件就是导出的SQL语句，一般只需要执行这个文件就可以完成数据恢复
2. **select ... into outfile**：导出一张表的数据，通过load data infile来进行导入恢复

8.4 二进制日志备份与恢复

二进制日志非常关键，用户可以通过它完成point-in-time的恢复工作。MySQL数据库的replication同样需要二进制日志。在默认情况下并不启用二进制日志，要使用二进制日志首先必须启用它。

推荐的二进制日志的服务器配置：

```
log-bin = mysql-bin
sync_binlog = 1
innodb_support_xa = 1
```

8.5 热备

1. ibbackup
2. XtraBackup

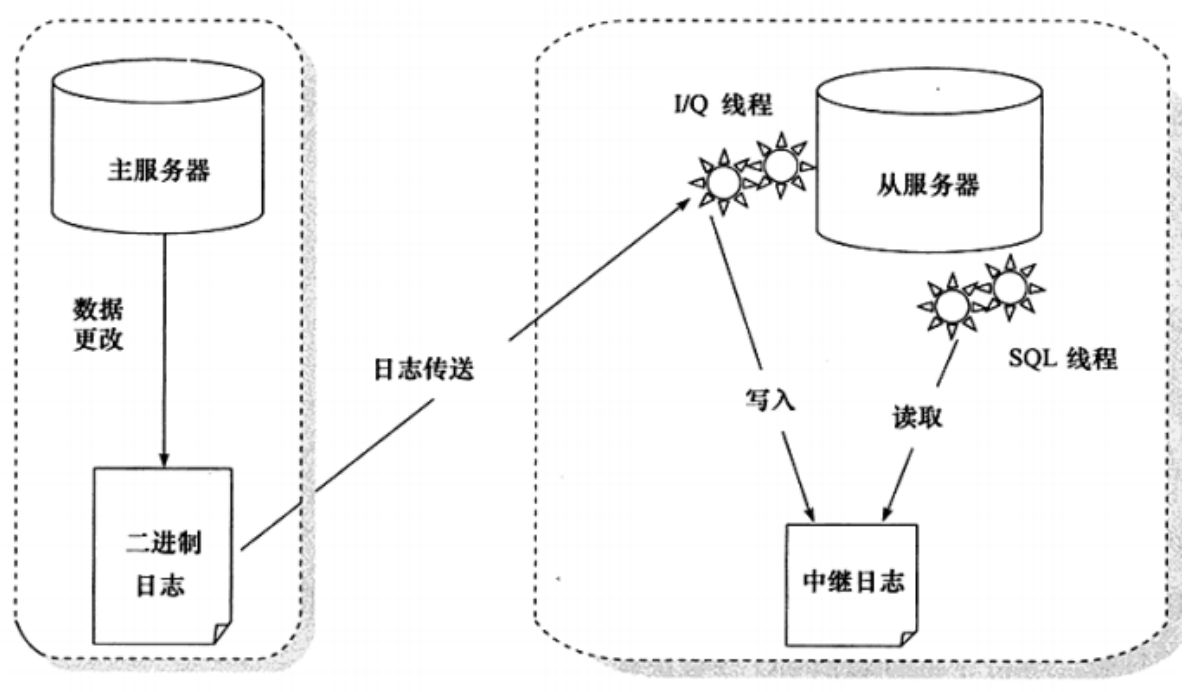
8.6 快照备份

通过文件系统支持的快照功能对数据库进行备份。备份的前提是所有数据库文件放在同一文件分区中，然后对该分区进行快照操作。

8.7 复制

复制(replication) 是MySQL数据库提供的一种高可用高性能的解决方案，一般用来建立大型的应用。总体来说，replication 的工作原理分为以下3个步骤:

1. 主服务器(master) 把数据更改记录到二进制日志(binlog) 中。
2. 从服务器(slave) 把主服务器的二进制日志复制到自己的中继日志(relay log)中。
3. 从服务器重做中继日志中的日志，把更改应用到自己的数据库上，以达到数据的最终一致性。



8.8 企业实现

采用 阿里云 **云数据库 RDS 版**

采用 **半同步** 方式

https://help.aliyun.com/document_detail/26183.html?spm=5176.2020520104.237.3.301b1450ZjO2lm