MySQL提高阶段

1 事务和隔离级别

1.1 事务的基本概念

MySQL中的事务是一系列SQL操作的集合，这些操作要么全部成功执行，要么全部失败回滚。事务是数据库管理系统提供的一种机制，用于确保数据库的完整性、一致性和持久性。事务具有以下特性：

- **原子性（Atomicity）**：事务是一个原子操作单元，不可再分割。这意味着事务中的所有操作要么全部成功执行，要么全部失败回滚，即使在系统发生故障时也是如此。
- **一致性（Consistency）**：事务的执行使数据库从一个一致性状态转移到另一个一致性状态。在事务开始之前和事务结束之后，数据库都必须保持一致性。
- **隔离性（Isolation）**：事务的执行不受其他事务的干扰。即使在同一时间有多个事务同时运行，每个事务也必须与其他事务隔离，以防止数据损坏或不一致。
- **持久性（Durability）**：一旦事务提交成功，对数据库的改变将是永久性的，即使系统发生故障也不会丢失。

上面的四个性质一般可以缩写为ACID。值得注意的是，一致性和其他三个性质并不在同一个层次上，具体来说就是：一致性是上层业务追求的目标，而原子性、隔离性和持久性是事务本身的属性。原子性、一致性和隔离性一起合作从而实现一致性。

下面我们通过一个例子来说明上述性质：

当涉及到银行转账时，事务的性质尤为重要，因为涉及到资金的安全和准确性。假设有一个银行系统，其中有两个账户：账户A和账户B。现在，用户要进行一笔转账操作，从账户A向账户B转移一定金额的资金。

首先，业务目标是一致性：在转账之前和之后，系统必须保持一致性。也就是说，无论转账是否成功，银行系统都必须确保账户A和账户B的总余额保持不变。如果转账成功，账户A的余额减少了100元，而账户B的余额增加了100元，使得总资金仍然保持不变。

那么事务是如何保证执行过程中总资金不变的呢？

1. 原子性：假设用户要转移100元。在一个事务中，银行系统会首先从账户A中扣除100元，然后将这笔资金存入账户B。在这个过程中，要么这两个操作都成功执行，要么都失败。如果任何一个操作失败，例如因为网络故障或系统崩溃，资金将不会丢失，因为整个操作将被回滚，账户A和账户B的余额将保持不变。
2. 隔离性：当一个用户正在进行转账操作时，其他用户可能也同时进行转账操作或查询账户余额。隔离性要求这些并发操作之间不会相互干扰。在事务中，转账操作应该独立于其他操作，直到事务完成。这意味着在转账过程中，其他用户不应该看到账户余额的不一致或不正确的状态。
3. 持久性：一旦转账操作成功提交，银行系统必须保证这笔资金转移是永久性的。即使系统发生故障或重新启动，转账后的余额变化也必须得到保留，不能丢失。

通过使用事务，银行系统可以确保转账操作的安全性和准确性，从而保护用户的资金和数据。

1.2 在MySQL当中使用事务

下面是在MySQL当中和事务相关的几个命令。

命令名称	描述
START TRANSACTION/BEGIN	用于启动一个新的事务。在启动事务之后，后续的SQL操作将被视为一个原子操作序列，直到显式地提交或回滚事务为止。
COMMIT	用于提交事务。当事务中的所有操作都成功完成并且需要将更改永久保存到数据库时，应使用COMMIT命令。提交事务会使得事务中的所有操作生效。
ROLLBACK	用于回滚事务。当在事务中发生错误或需要取消之前的更改时，可以使用ROLLBACK命令。回滚将撤销事务中的所有操作，并将数据库恢复到事务开始之前的状态。
SAVEPOINT	用于创建一个保存点（Savepoint），使得可以在事务中的任何位置进行部分回滚。SAVEPOINT命令允许将事务分割成更小的逻辑单元，并在需要时回滚到指定的保存点。
RELEASE SAVEPOINT	用于释放一个保存点，表示不再需要回滚到该保存点。RELEASE SAVEPOINT命令允许释放事务中已经创建的保存点，以节省资源。
ROLLBACK TO SAVEPOINT	用于回滚到指定的保存点。当需要在事务中的某个位置进行部分回滚时，可以使用ROLLBACK TO SAVEPOINT命令将事务回滚到指定的保存点。

接下来，我们通过SQL命令来演示上述例子。

```
# 构建表格并插入数据
CREATE TABLE bank_accounts (
    id INT AUTO_INCREMENT PRIMARY KEY,
    account_number VARCHAR(20) UNIQUE,
    balance DECIMAL(10, 2)
);

INSERT INTO bank_accounts (account_number, balance) VALUES
('1001', 1000.00),
('1002', 2000.00);

# 开始第一个事务
BEGIN; # 或者START TRANSACTION
# 随便执行几个DML
select * from bank_accounts;
update bank_accounts set balance = 1100 where id = 1;
# 在update之后，在本窗口和另一个窗口看到的表内容是不一样的
# 撤销修改并终止
ROLLBACK;

# 开始第二个事务
BEGIN; # 或者START TRANSACTION
# 执行几个DML
update bank_accounts set balance = 1100 where id = 1;
# 创建一个保存点
SAVEPOINT point1;
# 执行一个错误的语句
update bank_accounts set balance = 1900 where id = 1;
# 回到保存点
```

```
ROLLBACK TO point1;
# 执行一个正确的语句
update bank_accounts set balance = 1900 where id = 2;
# 提交事务
COMMIT;
# 之后本窗口和其他窗口看到的内容就一致了
```

> 如果系统变量autocommit的数值为1，那么每一句DML默认就是一个事务。如果系统变量autocommit的数值为0，那么执行DML默认在一个事务当中，需要commit才能结束事务。

2 并发问题和隔离级别

在之前的例子当中，我们了解到事务具有隔离性：一个事务在执行过程中，在另外的窗口是无法读到表的数据变化的。这种隔离性的实现就意味数据库底层需要处理多个用户的并发访问带来的问题。那接下来我们就要解决这两个问题：

- 并发会带来什么样的问题？所有的并发异常都是不可接受的吗？
- 隔离程度会存在不同的级别吗？隔离级别越高越好吗？

2.1 并发产生的四个问题

脏写（Dirty Write）：脏写是指在并发环境下，一个事务修改了另一个事务尚未提交的数据，然后另一个事务又修改了相同的数据并提交，导致数据变得不一致或无效。

```
T1          T2
----------|-----------
begin     |
read(A)   |
A=A+100   |
          |begin
          |read(A)
          |A=A+100
          |write(A) #假设没有隔离性
          |commit
write(A)  |
commit    |

T1事务在对A加了100之后,数据没有到数据库当中,T2读到的是原来的1000,随后T2 T1 相继写入都写入的使用1100,相当于T1的写入没有发生,违反了最终的一致性.
```

脏读（Dirty Read）： 当一个事务读取了另一个事务尚未提交的数据时，就发生了脏读。如果该事务后来回滚，则读取的数据实际上是无效的或不一致的。这可能会导致系统中出现不正确的结果。

```
T1          T2
----------|-----------
begin     |
read(A)   |
A=A-100   |
write(A)  |
          |begin
          |read(A) #假设只隔离了写,可以读取未提交的数据
          |read(B) #假设只隔离了写,可以读取未提交的数据
          |sum=A+B
          |commit
```

```
read(B)   |
B=B+100   |
write(B)  |
commit    |
```

T1将数据A从1000变为了900，并且写入到数据库中，但是T2去读的时候，A已经变成了900，然后B没有变化，接下来将A与B的总和写到数据库中并且提交了，对于T2而言数据的总和在执行前后是不一致的，就违背了事务的一致性但是对于T1而言，数据的综合还是2000，就是正常的。

不可重复读（Non-repeatable Read）： 不可重复读是指在一个事务内多次读取**同一行**数据，但由于其他事务的更新操作，每次读取的数据都不一致。这可能会导致在同一事务内对相同数据进行多次读取时出现不一致的结果。

```
T1         T2
----------|-----------
          |begin
          |read(A) #假设隔离了写,也只能读取已经提交的数据
          |生成报表1
read(A)   |
A=A+100   |
write(A)  |
commit    |
          |read(A) #假设隔离了写,也只能读取已经提交的数据
          |生成报表2
          |commit
```

幻读（Phantom Read）： 幻读是指在一个事务内多次执行相同的查询，但由于其他事务的插入或删除操作，每次执行的结果集都不一致。这可能会导致在同一事务内执行相同查询时，结果集中出现新增或删除的行。

```
T1         T2
----------|-----------
          |begin
          |read(所有) #假设隔离了写,以前读过的行都上了锁
          |生成报表1 A-->B-->C
insert(X) |
commit    |
          |read(所有) #假设隔离了写,以前读过的行都上了锁
          |生成报表2 A-->B-->C-->X
          |commit
```

2.2 隔离级别

为了不同程度地解决并发带来的问题，数据库提出了隔离级别的概念：

1. **读未提交（Read Uncommitted）**：
    - 在这个隔离级别下，一个事务可以读取到其他事务尚未提交的数据。这是最低的隔离级别，可能导致脏读、不可重复读和幻读等并发问题。
2. **读已提交（Read Committed）**：
    - 在这个隔离级别下，一个事务只能读取到其他事务已经提交的数据。这可以避免脏读，但是仍然可能发生不可重复读和幻读的问题。

3. **可重复读（Repeatable Read）**：
    - 在这个隔离级别下，一个事务在同一个事务中多次读取相同的数据，得到的结果是一致的。这可以避免脏读和不可重复读，但是仍然可能发生幻读的问题。
4. **串行化（Serializable）**：
    - 在这个隔离级别下，事务是串行执行的，每个事务都像是在独立的系统中执行一样。这可以避免脏读、不可重复读和幻读等所有并发问题，但是会降低系统的并发性能。

从上往下的隔离级别等级越来越高，等级越高，隔离效果就越好，但是消耗的系统资源也越多，用户可以根据实际情况进行设定。这些隔离级别可以不同程度地解决之前并发带来的问题。如下表所示：

	脏写	脏读	不可重复读	幻读
读未提交 READ UNCOMMITTED	×	√	√	√
读已提交 READ COMMITTED	×	×	√	√
可重复读 REPEATABLE READ	×	×	×	√
串行化 SERIALIZABLE	×	×	×	×

我们可以在MySQL当中使用命令调整隔离级别。

```
SELECT @@transaction_isolation; # 5.7也可以用tx_isolation
+-------------------------+
| @@transaction_isolation |
+-------------------------+
| REPEATABLE-READ         |
+-------------------------+

#这会将当前会话的事务隔离级别设置为读未提交（Read Uncommitted）。
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

我们可以演示一个脏读的例子：

```
# 窗口1更改了隔离级别
1>SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
# 窗口2也更改了隔离级别
2>SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
# 窗口1select
1>begin;
1>select * from bank_accounts;
# 窗口2修改数据
2>begin;
2>update bank_accounts set balance = 1000 where id = 1;
# 窗口1再select 发现数据变了
1>select * from bank_accounts;
# 窗口1 update的时候会引发阻塞 （ctrl+c可以中止）
1>update bank_accounts set balance = 900 where id = 1;
# 直到窗口2commit以后才可以update
```

将隔离级别调整成读已提交：

```
#窗口1更改了隔离级别，
```

```
1>SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
# 窗口2也更改了隔离级别
2>SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
# 窗口1读取数据
1>begin;
1>select * from bank_accounts;
# 窗口2修改数据
2>begin;
2>update bank_accounts set balance = 1000 where id = 1;
# 窗口1再select 发现数据不变
1>select * from bank_accounts;
# 窗口2commit
2>commit;
# 窗口1再select 发现数据变了
1>select * from bank_accounts;
```

将隔离级别调整成可重复读：

```
#窗口1更改了隔离级别，
1>SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
# 窗口2也更改了隔离级别
2>SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
# 窗口1读取数据
1>begin;
1>select * from bank_accounts;
# 窗口2修改数据
2>begin;
2>update bank_accounts set balance = 1000 where id = 1;
# 窗口1再select 发现数据不变
1>select * from bank_accounts;
# 窗口2commit
2>commit;
# 窗口1再select 发现数据还是不变
1>select * from bank_accounts;
# 窗口1commit之后，数据才真的变了
1>commit;
1>select * from bank_accounts;
```

MySQL当中，如果使用InnoDB存储配合REPEATABLE READ隔离级别，可以一定程度地避免幻读问题。

```
#窗口1更改了隔离级别，
1>SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
# 窗口2也更改了隔离级别
2>SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
# 窗口1读取数据
1>begin;
1>select * from bank_accounts;
# 窗口2插入一行新数据，并commit
2>begin;
2>insert into bank_accounts values (3,1003,1000);
2>commit;
# 窗口1再select 发现数据还是不变 所以select是没有幻读的
1>select * from bank_accounts;
# 但是读取最新的数据貌似有幻读
```

```
1>select * from bank_accounts for update;
# 但是insert into依然无法插入成功
1>insert into bank_accounts values (3,1003,1000);
ERROR 1062 (23000): Duplicate entry '3' for key 'bank_accounts.PRIMARY'
```

在可重复读（Repeatable Read）隔离级别下，InnoDB 存储引擎通过多版本并发控制（MVCC）来解决幻读问题。MVCC 是一种并发控制机制，允许事务在读取数据时看到一个一致性的快照，从而避免了某些幻读问题。

具体来说，在可重复读隔离级别下，当一个事务执行查询操作时，InnoDB 会对查询的每一行数据都创建一个快照（snapshot），该快照反映了事务启动时数据库中数据的状态。这意味着在同一事务内，即使其他事务对数据进行了修改或插入操作，事务也只会看到快照中的数据，而不会看到其他事务后来所做的修改或插入。

当其他事务对数据进行修改或插入时，InnoDB 会为新的数据行创建一个新的版本，并保留旧版本的数据。因此，在可重复读隔离级别下，一个事务在执行查询时会始终看到它启动时的数据快照，这样就避免了幻读问题。

3 索引

3.1 索引的作用

首先，我们先来看一个使用索引来优化查找的示例：

```
# 我们事先准备好一个100000行内容的表
mysql> select count(*) from large_table;
+----------+
| count(*) |
+----------+
|  1000000 |
+----------+
1 row in set (0.05 sec)

mysql> desc large_table;
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| id    | int         | NO   | PRI | NULL    | auto_increment |
| col1  | varchar(255) | YES  |     | NULL    |                |
| col2  | int         | YES  |     | NULL    |                |
| col3  | date        | YES  |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
4 rows in set (0.01 sec)

# 按主键查询
mysql> select * from large_table where id = 30000;
+-------+------------+------+------------+
| id    | col1       | col2 | col3       |
+-------+------------+------+------------+
| 30000 | Value958354 |  894 | 2020-07-27 |
+-------+------------+------+------------+
1 row in set (0.00 sec)
# 不按主键查询
mysql> select * from large_table where col1 = 'Value958354';
+--------+------------+------+------------+
```

```
| id     | col1        | col2 | col3       |
+--------+-------------+------+------------+
|  30000 | Value958354 |  894 | 2020-07-27 |
| 661608 | Value958354 | 7099 | 2020-09-03 |
+--------+-------------+------+------------+
2 rows in set (0.26 sec)
```

从上面的例子当中可以看出，按照主键查找和按照非主键查找的时间存在很大的差异，甚至相差了几个数量级。

为什么查找主键字段和其他有如此大的速度差异呢？在这里，索引就发挥了至关重要的作用。

去过图书馆的同学应该会知道图书馆的检索系统。图书馆为图书准备了检索目录，包括书名、书号、对应的位置信息，包括在哪个区、哪个书架、哪一层。我们可以通过书名或书号，快速获知书的位置，拿到需要的书。

MySQL 中的索引，就相当于图书馆的检索目录，它是帮助 MySQL 系统快速检索数据的一种存储结构。我们可以在索引中按照查询条件，检索索引字段的值，然后快速定位数据记录的位置，这样就不需要遍历整个数据表了。而且，数据表中的字段越多，表中数据记录越多，速度提升越是明显。

3.2 如何选择索引的数据结构?

在MySQL中，索引的数据结构不是唯一的，通常会根据不同的需求选择不同的数据结构。下面我们来了解一下各种常见的数据结构的优缺点：

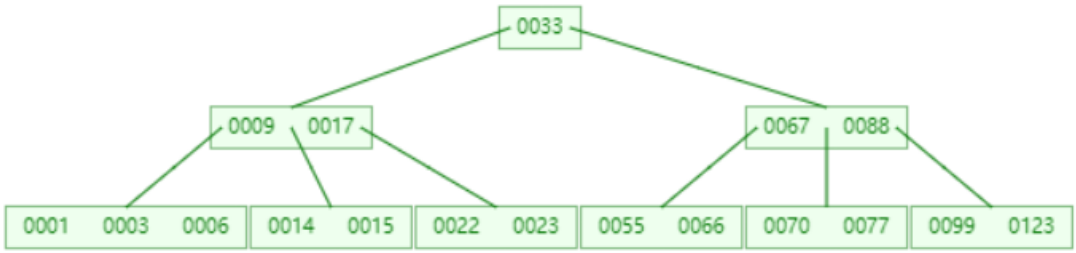
1. 哈希索引：优点是对于等值查询（比如 in、= 运算符）非常快，性能稳定且可预测。适用于对唯一性要求高的列，能够快速查找唯一值。缺点是不支持范围查询和排序，只能用于等值查询。哈希碰撞可能会影响查询性能，特别是在哈希表中有很多冲突的情况下。
2. 二叉排序树：优点是支持范围查询，缺点是在输入数据特殊的情况下有可能会退化成链表，使用红黑树取代之可以避免该问题。另外就是，虽然查找的时间复杂度很低，但是树的深度太深，导致访问磁盘的次数过多，检索速度慢。
3. 多叉排序树，B树：一方面，B树是排序树，支持范围查询；另一方面，B树是多叉树，在每一个节点可以存储多份数据，所以树的深度比红黑树低得多。在B树中时，每个节点中既存储索引信息，又存储数据域。

演示动画：[B-Tree Visualization (usfca.edu)]

测试数据：

```
3 15 17 22 88 33 123 9 6 14 1 23 66 55 99 77 67 70
```

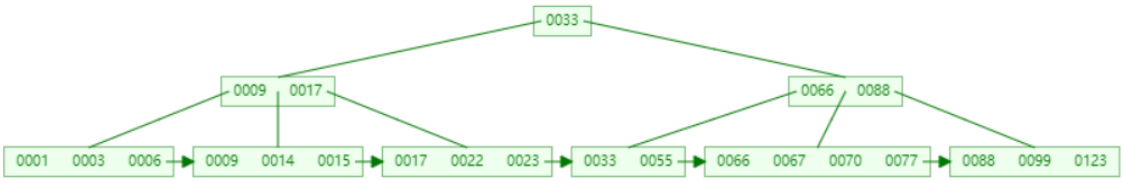
最终B树的形状：



4. B+树：在B树的基础上，为了降低树的深度，可以进一步进行优化。在B+树中，所有的数据域都存储在叶子节点中，而非叶子节点只存储索引信息。假设存储空间的一页的大小是固定的，那么每个节点越小，单页可以存储节点数就越多，由于非叶子节点的只存储索引，所以相较B树，一页能够保存的非叶子节点会更多。另外，B+树的叶子节点通过指针连接成一个有序链表，便于范围查询。

演示动画：[B+ Tree Visualization (usfca.edu)]

最终B+树的形状：



综上，数据库索引最常用的底层数据结构就是B+树。

3.3 MySQL索引的分类

在MySQL当中，可以根据索引的不同性质来对索引分类。

索引的性质分类

我们可以根据列的性质对索引进行分类。在主键上的索引称为**主键索引**，在唯一键上的索引称为**唯一索引**，其他的索引称为**常规索引**。值得特别注意的是：给表设定主键约束或者唯一性约束的时候，MySQL会自动创建主键索引或唯一性索引。这也是建议在创建表的时候，一定要定义主键的原因之一。

下面的例子创建了一个表并且自动创建了主键索引和唯一索引：

```
mysql> create table test (id int primary key, col1 int unique key, col3 int, col4
varchar(30));
Query OK, 0 rows affected (0.02 sec)

mysql> show index from test\G
*************************** 1. row ***************************
        Table: test
   Non_unique: 0 #标识非重复
     Key_name: PRIMARY
 Seq_in_index: 1 #索引中列的顺序
  Column_name: id
    Collation: A #排序规则
  Cardinality: 0 #不同值的数量
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_comment:
      Visible: YES
   Expression: NULL
*************************** 2. row ***************************
        Table: test
   Non_unique: 0
     Key_name: col1
 Seq_in_index: 1
  Column_name: col1
```

```
    Collation: A
  Cardinality: 0
     Sub_part: NULL
       Packed: NULL
         Null: YES
   Index_type: BTREE
      Comment:
Index_comment:
      Visible: YES
   Expression: NULL
```

下面是常规索引创建和删除的SQL语句：

```
#创建表的时候设置常规索引
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    ...
    INDEX index_name (column1, column2)
);
# 建好表以后再新增索引
CREATE INDEX index_name ON table_name (column1, column2);
ALTER TABLE table_name ADD INDEX index_name (column1, column2);
# 删除索引
DROP INDEX index_name ON table_name;
ALTER TABLE table_name DROP INDEX index_name;
```

列的数量分类

根据关注列的数量不同可以对索引进行分类。以单字段内容作为索引的排序查找的依据称为**单字段索引**；多个字段组合在一起作为索引的排序查找的依据称为**组合索引**。

在实际工作中，有时会遇到比较复杂的数据表，这种表包括的字段比较多，经常需要通过不同的字段筛选数据，特别是数据表中包含多个层级信息。我们经常要把这些层次信息作为筛选条件，来进行查询。这个时候单字段的索引往往不容易发挥出索引的最大功效，可以使用组合索引。

组合索引的多个字段是有序的，遵循左对齐的原则。因此，筛选的条件也要遵循从左向右的原则，如果中断，那么，断点后面的条件就没有办法利用索引了。

```
create table test (id int primary key, col1 int,col2 varchar(30), col3 int, col4
varchar(30), index myindex(col3,col2,col1));
# 在访问索引时，先定位col3，再定位col2，最后定位col1
mysql>  explain select * from test where col3 = 1 and col2 = 'def' and col1 = 2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: test
   partitions: NULL
         type: ref
possible_keys: myindex
          key: myindex
      key_len: 133
          ref: const,const,const
         rows: 1
     filtered: 100.00
```

```
        Extra: NULL
1 row in set, 1 warning (0.00 sec)

mysql> explain select * from test where col2 = 'def' and col1 = 2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: test
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 3
     filtered: 33.33
        Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

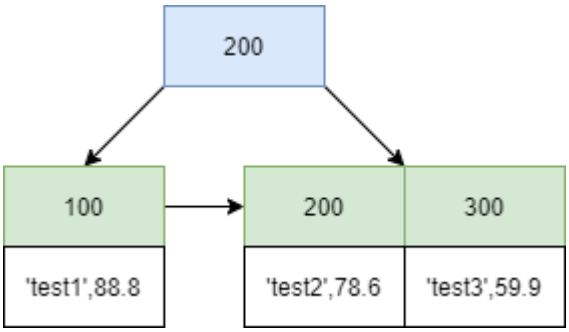
存储性质分类

我们可以根据索引的存储性质对索引进行分类。在InnoDB引擎当中，表的索引和表的内容有可能会放在一起。**聚簇索引**是一种特殊类型的索引，它与表的物理顺序一致存储在一起。在聚簇索引中，索引的叶子节点存储的是整行数据，而不是指向数据所在位置的指针。因此，聚簇索引可以加速范围查询和排序操作，但如果表中的数据经常被更新，可能会导致数据的物理重组和碎片的产生。在MySQL中，若使用InnoDB存储引擎且表存在主键，那么表的主键索引就是一种聚簇索引。

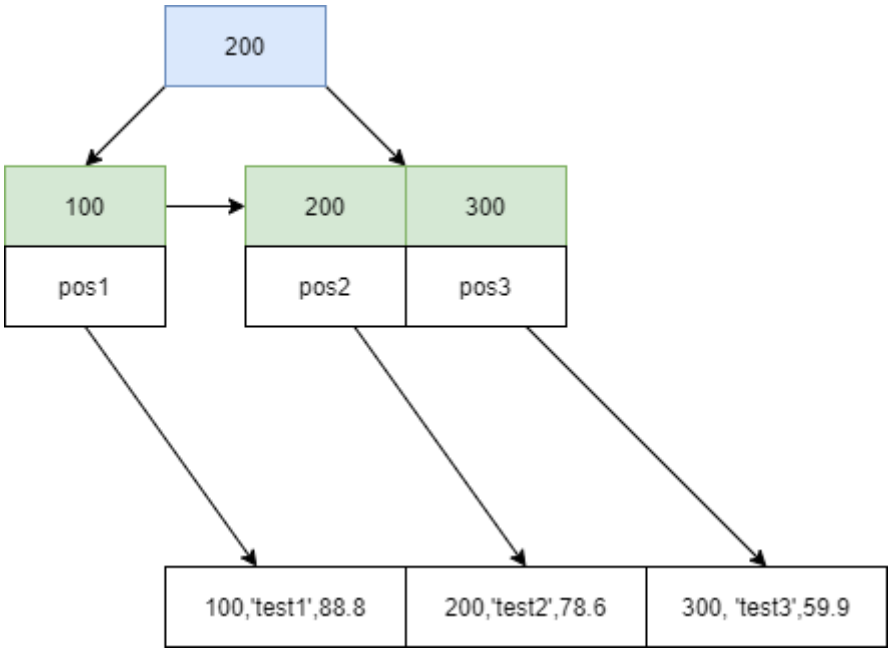
而**非聚簇索引**（或称为次要索引）将索引存储在一个地方，而实际数据存储在另一个地方。索引中的条目包含指向实际数据行的指针。

下面是聚簇索引和非聚簇索引的示意图。

聚簇索引：



非聚簇索引：



因为在InnoDB引擎中，主键优先是聚簇索引，所以使用主键就会存在下面的要求：

- 合理设计表的主键是非常重要的。推荐使用递增的整数作为主键，以避免数据的频繁移动和碎片化。
- 避免更新主键。对于频繁更新主键的表，可能会引起聚簇索引的频繁重组，导致性能下降。

> **覆盖索引**是指一个索引包含了查询所需的所有列，即索引可以“覆盖”查询的需求，不需要回表到数据页中获取额外的信息。通过使用覆盖索引，可以减少磁盘I/O和内存开销，提高查询的性能。
>
> 如果表的主键是聚簇索引，并且查询中涉及到了主键列，那么这个主键索引就可以作为覆盖索引来使用。因为聚簇索引的叶子节点存储的是整行数据，所以当查询需要的列包含在聚簇索引中时，可以直接从索引中获取数据，而不需要**回表**到数据页中获取额外的信息，从而实现了覆盖查询。

3.4 索引的优缺点

索引在数据库中起着至关重要的作用，它们可以加速数据的检索和查询操作，提高数据库的性能。索引存在很多优点：

1. **提高查询性能**：通过使用索引，数据库系统可以更快速地定位和访问数据，减少了数据的扫描量，从而加快了查询的执行速度。
2. **加速排序操作**：索引可以提供有序访问数据的能力，因此在执行排序操作时，可以减少排序的时间复杂度，提高排序的效率。
3. **支持唯一性约束**：通过在列上创建唯一索引，可以确保列中的值唯一，从而保证数据的完整性和一致性。
4. **优化连接操作**：对于连接操作（如JOIN），如果连接字段上存在索引，可以大大减少连接的时间复杂度，提高连接操作的性能。

有些情况下索引也有所不足：

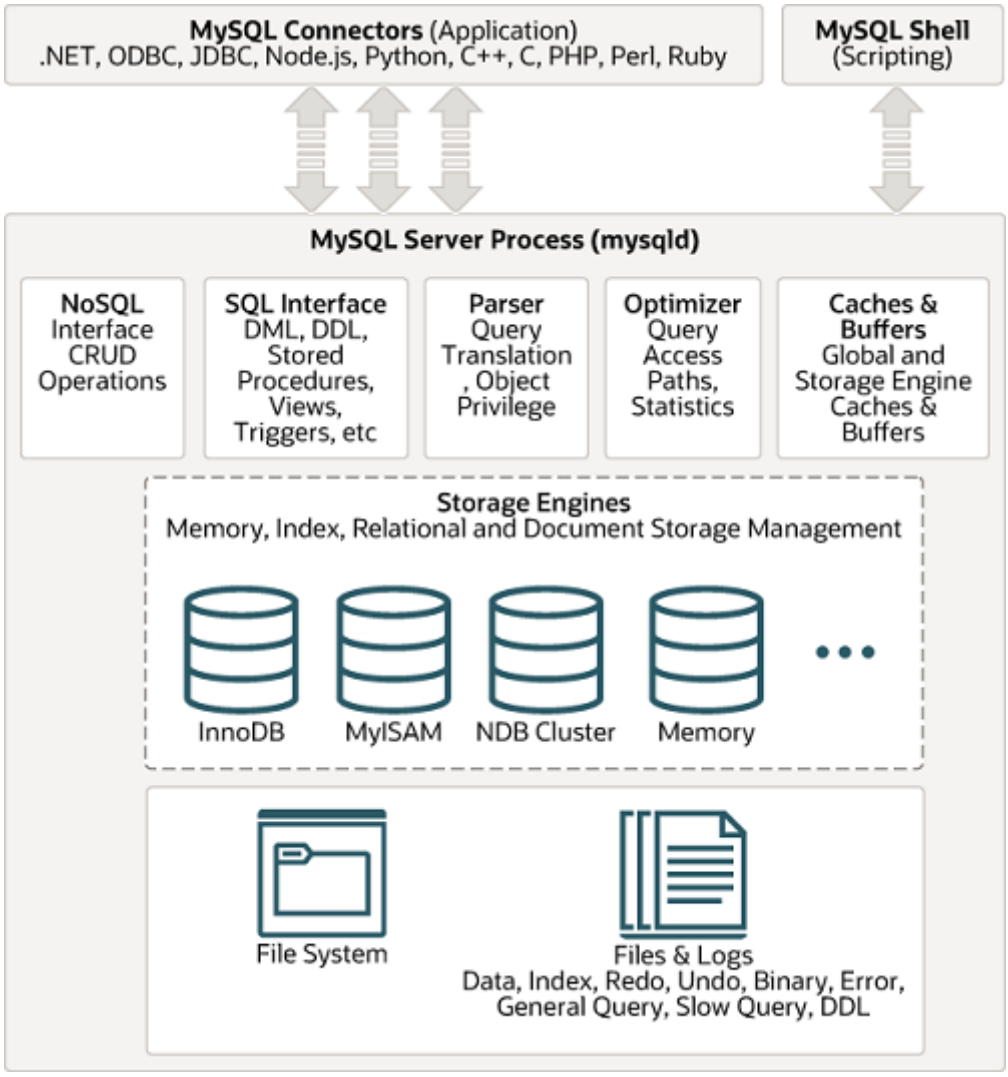
1. **占用存储空间**：索引需要占用额外的存储空间来存储索引数据结构，特别是对于大型表或复合索引，可能会占用大量的存储空间。
2. **增加写操作的开销**：对于插入、更新和删除操作，数据库系统需要维护索引的一致性，这可能会增加写操作的开销，降低写操作的性能。
3. **需要定期维护**：随着数据的插入、更新和删除，索引会变得不连续并产生碎片，需要定期进行索引的重建或优化，以维护索引的性能。

综上所述，索引在提高查询性能和数据完整性方面具有重要作用，但在使用时需要权衡其优缺点，并根据具体的业务需求和数据库特性进行合理的索引设计和管理。

4 存储引擎

4.1 MySQL的逻辑结构

在讲解MySQL的存储引擎之前，我们首先需要了解一下MySQL的逻辑结构：



可以发现存储位于SQL解析、优化程序的下层，文件系统的上层，所以存储引擎的基本功能就很明确了——存储引擎支持了最基础的增、删、查、改。

4.2 MySQL支持的存储引擎

由前文可知，存储引擎是数据库管理系统中的一个组件，它负责将数据存储到磁盘上，并提供对数据的读取、写入和管理功能。存储引擎定义了数据在磁盘上的存储格式、访问方法和支持的特性，不同的存储引擎具有不同的特点和适用场景。

MySQL支持多种存储引擎，每种引擎都具有其独特的特性、优势和适用场景。MySQL中常见的存储引擎有InnoDB、MyISAM和Memory。

```
# show engines命令可以列出MySQL支持的所有存储引擎
mysql> show engines\G
*************************** 1. row ***************************
      Engine: ARCHIVE
     Support: YES
```

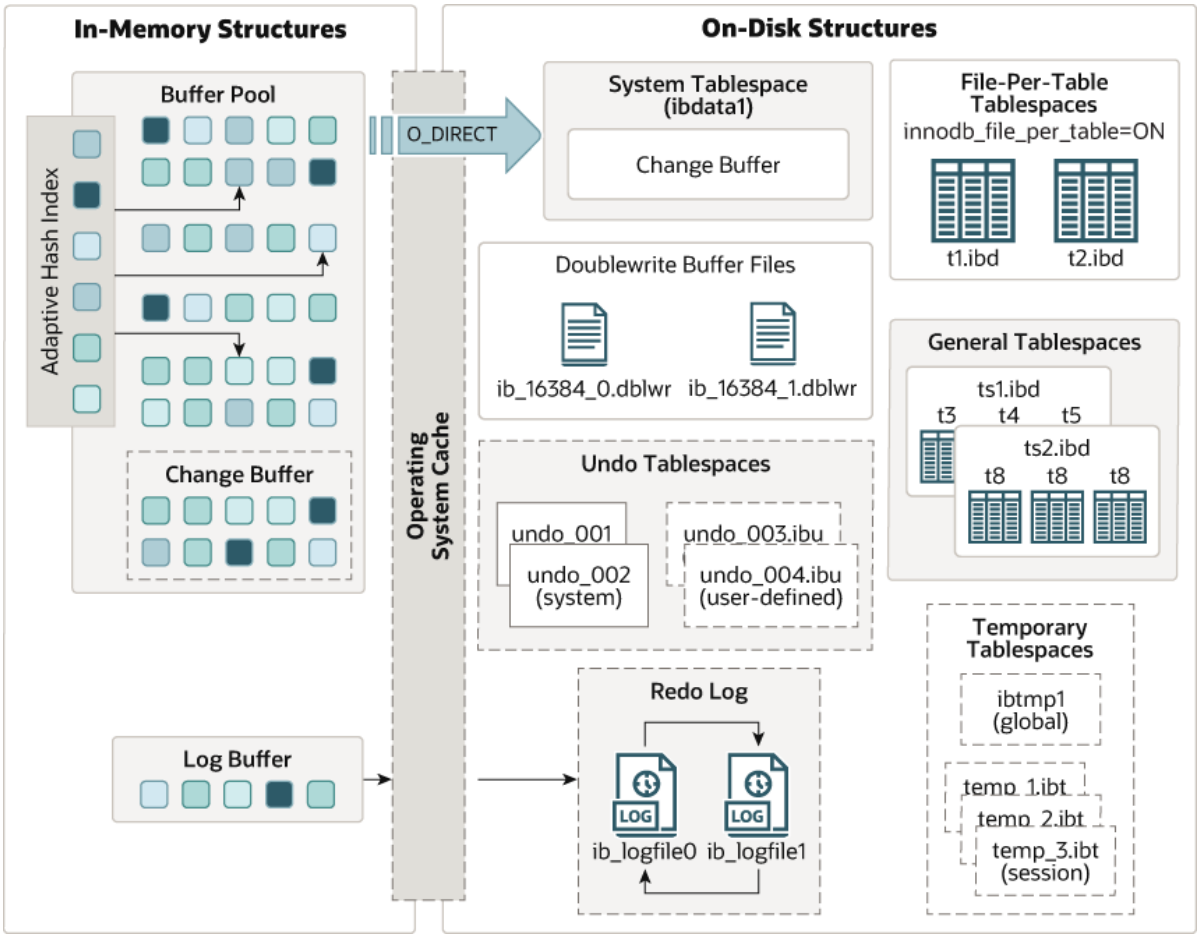
```
     Comment: Archive storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 2. row ***************************
      Engine: BLACKHOLE
     Support: YES
     Comment: /dev/null storage engine (anything you write to it disappears)
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 3. row ***************************
      Engine: MRG_MYISAM
     Support: YES
     Comment: Collection of identical MyISAM tables
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 4. row ***************************
      Engine: FEDERATED
     Support: NO
     Comment: Federated MySQL storage engine
Transactions: NULL
          XA: NULL
  Savepoints: NULL
*************************** 5. row ***************************
      Engine: MyISAM
     Support: YES
     Comment: MyISAM storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 6. row ***************************
      Engine: PERFORMANCE_SCHEMA
     Support: YES
     Comment: Performance Schema
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 7. row ***************************
      Engine: InnoDB
     Support: DEFAULT
     Comment: Supports transactions, row-level locking, and foreign keys
Transactions: YES
          XA: YES
  Savepoints: YES
*************************** 8. row ***************************
      Engine: MEMORY
     Support: YES
     Comment: Hash based, stored in memory, useful for temporary tables
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 9. row ***************************
      Engine: CSV
     Support: YES
```

```
      Comment: CSV storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
```

4.3 InnoDB的特性

InnoDB是MySQL 5.5被引入作为默认的存储引擎的。InnoDB支持事务、行级锁、外键约束、自动增长主键和MVCC等更多功能，以及具有高并发性和高性能，适用于需要事务支持和高并发读写的应用场景。

InnoDB的逻辑结构如下所示：



InnoDB支持行级锁

在一些业务场景中，比如正在对某张表或者某个数据库去做备份时，我们往往需要限制一些并发行为；在事务的不同隔离级别中，我们对不同的读写行为也会有着不同的限制。想要对并发行为进行控制就依赖于存储引擎底层一种非常重要的数据结构——锁。

存储引擎当中的锁有很多种，有些锁（比如表级锁、全局锁、行级锁）可以开放给用户使用，有些锁（比如意向锁、间隔锁、Next-Key锁等等）不能给用户直接使用，但是在实现各种隔离级别中发挥了重要作用。

在InnoDB引擎中，锁的最低粒度可以做到行级锁。行锁就是针对数据表中行记录的锁，分为读(S 共享)锁和写(X 排斥)锁。在很多隔离级别当中，行级锁发挥了重要作用：

```
# 比如在串行化隔离级别情况下，由于事务导致锁竞争
show status like 'innodb_row_lock%';
+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
| Innodb_row_lock_current_waits | 0     |
| Innodb_row_lock_time          | 26128 |
| Innodb_row_lock_time_avg      | 26128 |
| Innodb_row_lock_time_max      | 26128 |
| Innodb_row_lock_waits         | 1     |
+-------------------------------+-------+
```

用户也可以在较低隔离级别下的情况自行使用行锁：

```
# 上读锁的语法
# SELECT ... LOCK IN SHARE MODE;
1>SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; # 读未提交
2>SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; # 读未提交
1>begin; # 开启事务
1>select * from test1 where id = 1 lock in share mode; # 上读锁
2>begin; # 开启事务
2>select * from test1 where id = 1 lock in share mode; # 再上读锁不阻塞
2>insert into test1 values (2,'123','456'); # 插入不阻塞
2>update test1 set col1 = '234' where id = 2; # 写入另一行也不阻塞
2>update test1 set col1 = '234' where id = 1; # 写入同一行被阻塞了
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction

# 上写锁的语法
# SELECT ... FOR UPDATE;
1>SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; # 读未提交
2>SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; # 读未提交
1>begin; # 开启事务
1>select * from test1 where id = 1 for update; #上写锁
2>begin;
2>select * from test1 where id = 1 for update; # 再上写锁阻塞
ERROR 1317 (70100): Query execution was interrupted
2>select * from test1 where id = 1 lock in share mode; # 上读锁也阻塞
ERROR 1317 (70100): Query execution was interrupted
2>update test1 set col2 = 'def' where id = 1; #写入也阻塞
ERROR 1317 (70100): Query execution was interrupted
2>select * from test1 where id = 2 for update;# 其他行没有任何问题
```

由于InnoDB引擎支持事务操作，所以用户一般不会使用表级锁。表级锁一般只在MyISAM引擎上使用。

> 间隙锁
>
> 间隙锁（Gap Lock）是MySQL InnoDB存储引擎的一种特殊类型的锁，用于锁定索引范围而不是特定的行。它的主要作用是防止其他事务在当前事务读取索引范围内的数据时插入新的数据，从而防止幻读的发生。间隙锁通常在一些特定的事务隔离级别下使用，例如可重复读（REPEATABLE READ）。在这个隔离级别下，InnoDB存储引擎会在SELECT语句执行时自动在扫描范围内的索引上加上间隙锁，以防止其他事务在这个范围内插入新的数据。
>
> 具体来说，间隙锁在索引范围内的两个索引键之间的间隙上进行锁定，而不是实际的数据行。当一个事务获取了一个间隙锁时，其他事务就无法在该间隙中插入新的数据。这样可以确保其他事务在当前事务读取索引范围内的数据时，不会受到新插入数据的影响，从而避免了幻读的发生。

需要注意的是，间隙锁只对插入操作起作用，而不影响已存在的数据行的更新或删除操作。因此，它主要用于防止幻读的发生，而不是防止其他事务修改已存在的数据行。

```
#准备一个带有主键的表
mysql> create table test1 (id int primary key, col1 int, col2 varchar(30));
mysql> select * from test1;
+----+------+------+
| id | col1 | col2 |
+----+------+------+
|  3 |    1 | aaa  |
|  7 |    2 | bbb  |
| 11 |    3 | ccc  |
| 16 |    4 | ddd  |
+----+------+------+
4 rows in set (0.00 sec)
#修改一下隔离级别
1>SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
2>SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
1>begin;
1>select * from test1 where id > 5 and id < 9 for update;
# 查看系统的锁情况
mysql> select THREAD_ID,LOCK_TYPE,LOCK_MODE,LOCK_DATA from
performance_schema.data_locks;
+-----------+-----------+-----------+-----------+
| THREAD_ID | LOCK_TYPE | LOCK_MODE | LOCK_DATA |
+-----------+-----------+-----------+-----------+
|       132 | TABLE     | IX        | NULL      |
|       132 | RECORD    | X         | 7         |
|       132 | RECORD    | X,GAP     | 11        |
+-----------+-----------+-----------+-----------+

2> begin;
2> insert into test1 values(12,4,'ddd');
Query OK, 1 row affected (0.00 sec)
2> insert into test1 values(2,5,'eee');
Query OK, 1 row affected (0.00 sec)
2> insert into test1 values(10,5,'eee');
^C^C -- query aborted
ERROR 1317 (70100): Query execution was interrupted
2> rollback;

2> begin;
Query OK, 0 rows affected (0.00 sec)
2> insert into test1 values(6,5,'eee');
^C^C -- query aborted
ERROR 1317 (70100): Query execution was interrupted
2> rollback;
```

InnoDB支持外键约束

外键约束是关系数据库中的一种约束，用于确保数据的参照完整性。外键约束定义了表与表之间的关系，其中一个表的列（参照表）引用了另一个表的主键列（被参照表）的值作为外键，参照表的外键列中的值只能从被参照表的关联列中已存在的值当中选取。

```
create table hero (hid int primary key, name varchar(30));
insert into hero values(1,'刘备'),(3,'张飞'),(5,'赵云');
create table weapon (wid int primary key, wname varchar(30), hid int, constraint
foreign key (hid) references hero(hid));

mysql> insert into weapon values (1,'丈八蛇矛',3);
Query OK, 1 row affected (0.00 sec)

mysql> insert into weapon values (2,'青龙偃月刀',2);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint
fails (`improve`.`weapon`, CONSTRAINT `weapon_ibfk_1` FOREIGN KEY (`hid`)
REFERENCES `hero` (`hid`))

mysql> delete from hero where hid = 3;
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key
constraint fails (`improve`.`weapon`, CONSTRAINT `weapon_ibfk_1` FOREIGN KEY
(`hid`) REFERENCES `hero` (`hid`))
```

InnoDB的On-disk结构

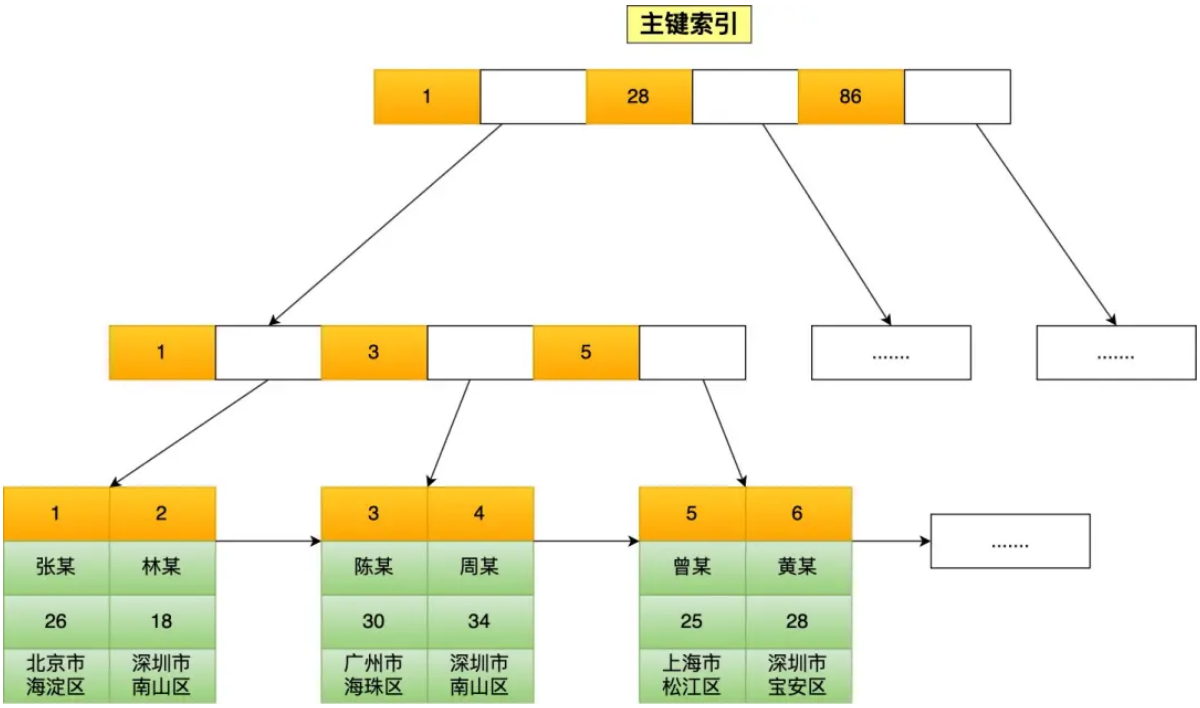
每当数据存储和修改的时候，InnoDB存储引擎最终都会在操作系统的文件系统中将数据和元信息以文件的方式存储起来。

```
# 该目录下存储所有的相关文件
sudo cd /var/lib/mysql

# 在另一个窗口执行事务，并往test1表当中写入数据，但还没有提交

# 可以获取最近被修改的数据
root@ubuntu:/var/lib/mysql# find . -mmin -10

./#ib_16384_0.dblwr
./undo_002
# undo log（回滚日志）：是 Innodb 存储引擎层生成的日志，实现了事务中的原子性，主要用于事务回滚和 MVCC。
./ibdata1
# 检查点文件（ibdata文件），用于存储数据页的改变和缓冲池中的数据。
./#innodb_redo/#ib_redo128
# redo log（重做日志）：是 Innodb 存储引擎层生成的日志，实现了事务中的持久性，主要用于掉电等故障恢复；
./improve/test1.ibd
# InnoDB存储引擎会创建一个或多个数据文件（通常以.ibd为扩展名），用于存储表的数据。这些数据文件包含了表中的实际数据、索引和其他元数据。
```

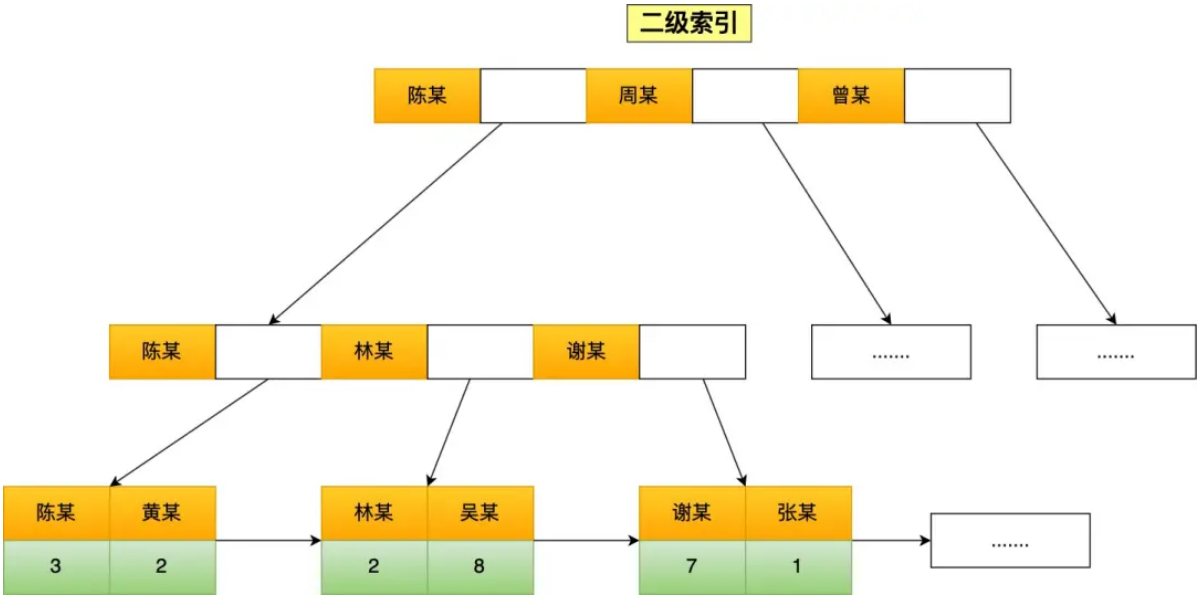
InnoDB优先使用聚簇索引

InnoDB存储引擎会优先使用聚簇索引，也就是说如果某张表存在主键索引，在默认情况下，主键索引就是聚簇索引，B+树的数据域就是表的行内容，后续添加的其他索引是非聚簇索引，也叫二级索引，B+树的数据域存储了主键值。

主键索引的示意图：



二级索引的示意图：



4.4 MyISAM的特性

MyISAM是旧版本（5.5版本一起）的MySQL的默认存储引擎。相较于InnoDB，MyISAM在很多特性上功能会比较缺失：

1. **表级锁定**：MyISAM 使用表级锁定来管理并发访问。这意味着当一个连接对表进行读写操作时，其他连接需要等待直到第一个连接释放锁定。这可能导致并发性能下降，特别是在高并发环境下。
2. **不支持事务**：MyISAM 引擎不支持事务，因此它不适合需要 ACID（原子性、一致性、隔离性和持久性）特性的应用程序，例如金融系统或者需要数据完整性的系统。
3. **不支持外键约束**：MyISAM 引擎不支持外键约束，这意味着数据库不会强制执行关系完整性，开发人员需要手动确保数据的一致性。
4. **支持全文索引**：MyISAM 支持全文索引，这使得对文本字段进行高效的全文搜索成为可能。这对于需要进行全文搜索的应用程序非常有用。（新版本的InnoDB也支持全文索引）
5. **较高的读取性能**：在读取密集型的应用场景下，MyISAM 通常比 InnoDB 表现得更好，这是因为它的表级锁定和更低的内存消耗。

下面是创建一个MyISAM表格并且使用表锁来控制并发访问的例子：

```
# 创建 MyISAM 表格
CREATE TABLE my_table (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50),
    age INT
) ENGINE=MyISAM;

# 使用表级锁定控制并发
# 在开始事务前手动获取表锁
1>LOCK TABLES my_table WRITE; # 1上写锁
2>LOCK TABLES my_table WRITE; # 2上写锁阻塞
^C^C -- query aborted
ERROR 1317 (70100): Query execution was interrupted
# LOCK TABLES my_table READ; # 上读锁
# 在事务中执行对表格的操作
1>INSERT INTO my_table (name, age) VALUES ('John', 30);
1>INSERT INTO my_table (name, age) VALUES ('Alice', 25);
# 释放表锁
1>UNLOCK TABLES;

2>LOCK TABLES my_table READ;
Query OK, 0 rows affected (0.00 sec)

2>SELECT * FROM my_table; #OK
+----+-------+------+
| id | name  | age  |
+----+-------+------+
|  1 | John  |   30 |
|  2 | Alice |   25 |
+----+-------+------+

2>INSERT INTO my_table (name, age) VALUES ('Bob', 20); #报错
ERROR 1099 (HY000): Table 'my_table' was locked with a READ lock and can't be
updated
```

接下来可以观察一下MyISAM存储引擎中会修改哪些磁盘文件：

```
root@ubuntu:/var/lib/mysql# find . -mmin -10
./improve/my_table_1338.sdi
./improve/my_table.MYI #索引文件
./improve/my_table.MYD #数据文件
```

4.4 MEMORY存储引擎的特性

Memory 存储引擎是 MySQL 数据库中的一种特殊存储引擎，它具有以下特点：

1. **数据存储在内存中**：与其他存储引擎不同，Memory 存储引擎将表数据存储在内存中而不是磁盘上。这意味着对于读取密集型的操作，Memory 存储引擎可以提供更快的访问速度，因为内存的读写速度通常比磁盘快得多。
2. **适用于缓存**：由于数据存储在内存中，Memory 存储引擎通常用于缓存数据，特别是在需要频繁访问临时数据或者需要快速查询结果的场景下。

3. **不支持事务**：与 MyISAM 类似，Memory 存储引擎也不支持事务。这意味着它不适合需要 ACID 特性的应用程序，因为在 Memory 存储引擎下，数据不会持久化到磁盘，一旦数据库重启或服务关闭，所有数据都会丢失。
4. **支持哈希索引**：Memory 存储引擎使用哈希索引而不是 B 树索引。这使得在某些特定情况下，例如对于等值查询，Memory 存储引擎可以提供更快的检索速度。
5. **表结构不支持TEXT和BLOB字段**：Memory 存储引擎不支持存储大型二进制对象（BLOB）和文本字段（TEXT）。因此，表结构中不能包含这些类型的字段。

```
create table test2 (id int primary key, col1 varchar(20), col2 int) engine =
memory;

insert into test2 values (1,'abc',1); # 插入表中的内容在断电以后会消失

mysql> select * from test2;
+----+------+------+
| id | col1 | col2 |
+----+------+------+
|  1 | abc  |    1 |
+----+------+------+
1 row in set (0.00 sec)
# 在另一个窗口重启服务端 sudo service mysql stop 再 sudo service mysql start
mysql> select * from test2;
ERROR 2013 (HY000): Lost connection to MySQL server during query
No connection. Trying to reconnect...
Connection id:    8
Current database: improve

Empty set (0.05 sec)
```

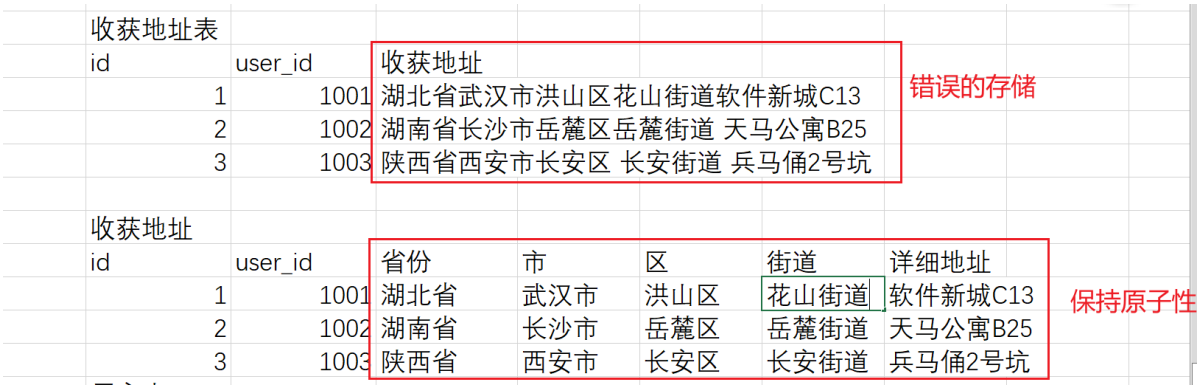
	InnoDB	MyISAM	MEMORY
存储限制	64TB	256TB	RAM
外键支持	√	×	×
哈希索引	√	×	√
锁的粒度	行	表	表
事务	√	×	×
MVCC	√	×	×

*数据库设计三大范式

在业务系统构建的过程中，用户应该怎么样设计数据库才好？在多年的数据库设计经验的基础上，很多大神总结了一些使用数据库的经验，并且最终提炼成了规则——数据库的**范式**就是这样的规则。范式用于设计关系数据库表结构，以减少数据冗余和提高数据存储效率。范式通常分为不同的级别，每个级别都有一组规则，用于确保数据库表的结构在存储数据时不会出现冗余、不一致或者不合理的情况。

*第一范式

确保表中的每个字段都是**原子性**的，即所有的字段都是基本数据字段，不可进一步拆分。这意味着一个字段中不应该包含多个值或者重复的值。



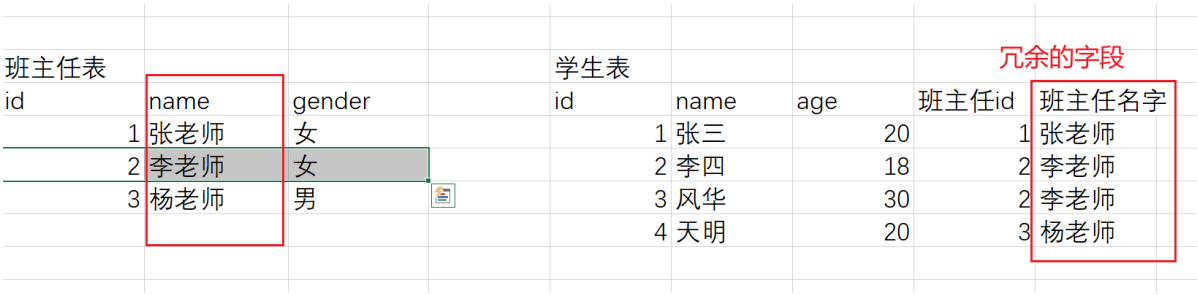
*第二范式

第二范式要求，在满足第一范式的基础上，还要满足数据表里的每一条数据记录，都是可唯一标识的。而且所有字段，都必须完全依赖主键，不能只依赖主键的一部分。也就是记录需要具有**唯一性**。

```
create table test1(
    id int primary key auto_increment,

)
```

*第三范式

第三范式要求数据表在满足第二范式的基础上，不能包含那些可以由非主键字段派生出来的字段，或者说，不能存在依赖于非主键字段的字段。简而言之，数据不要**冗余**。



在上表中，班主任名字重复存储出现冗余。冗余的缺点是：一方面，数据重复存储了，需要占用更多的磁盘空间，另一方面，如果要去修改某个老师的名字，那么需要在多个地方进行修改，增加了数据的维护成本。但是冗余也存在着好处：根据学生去查班主任的名字变得更简单了，查询效率变高了。

综上可知，冗余数据会使数据的维护成本增加，但是可以在某些场景中，方便数据的查询。那么在我们以后的工作中，要不要冗余数据呢？假如数据的查询需求远大于增删改的需求，那么可以考虑冗余数据；否则，不应该冗余数据。这种冗余数据的做法叫**反范式化设计**。如果你想查的更快，而且你不是特别在意这些磁盘空间，增删改的次数比较少，可以考虑冗余数据。

5 执行计划

5.1 执行计划和EXPLAIN

很多程序员在开发肯定遇到过这样的情况：写的 SQL 语句执行起来特别慢，要等好久才出结果，或者是干脆就“死”在那里，一点反应也没有。一旦遇到这种问题，我们就要考虑进行优化了。只要是开发过数据库应用的程序员肯定会有这样的体会：让应用运行起来不难，但是要运行得又快又好，就没那么不容易了，这很考验我们的内功。而要想提高应用的运行效率，就必须掌握优化查询的方法。

MySQL 的查询分析语句虽然不能直接优化查询，但是却可以用来让用户了解 SQL 语句的执行计划，以便分析查询效率低下的原因，进而有针对性地进行优化。

```
explain select * from large_table where id = 10000\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: large_table
   partitions: NULL
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
     filtered: 100.00
        Extra: NULL
1 row in set, 1 warning (0.00 sec)

explain select * from large_table where col1 = 'Value424931'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: large_table
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 996744
     filtered: 10.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

我们先来简单了解一下表格中各列的含义：

1. id：是一个查询序列号。
2. table：表示与查询结果相关的表的名称。
3. partition：表示查询访问的分区。
4. key：表示优化器最终决定使用的索引是什么。
5. key_len：表示优化器选择的索引字段按字节计算的长度。如果没有使用索引，这个值就是空。

6. ref：表示哪个字段或者常量被用来与索引字段比对，以读取表中的记录。如果这个值是“func”，就表示用函数的值与索引字段进行比对。
7. rows：表示为了得到查询结果，必须扫描多少行记录。
8. filtered：表示查询筛选出的记录占全部表记录数的百分比。
9. possible_key：表示 MySQL 可以通过哪些索引找到查询的结果记录。如果这里的值是空，就说明没有合适的索引可用。你可以通过查看 WHERE 条件语句中使用的字段，来决定是否可以通过创建索引提高查询的效率
10. Extra：表示 MySQL 执行查询中的附加信息。
11. type：表示表是如何连接的。

接下来我们来了解一些重点列的详细信息。

5.2 select_type

类型	描述
SIMPLE	简单的SELECT查询
PRIMARY	假如是一个复杂查询，最外层就称为PRIMARY
UNION	UNION语句中的第二、三...个SELECT
UNION RESULT	整合所有的UNION的查询结果

5.3 type

type列说明了表示表是如何连接的：

类型	描述符
system	由于表结构的特殊设置，表中只有一行。system是const的特例
const	表中最多只能有一个匹配项。当where中的条件都是唯一键/主键和常量作对比时，会启用const连接。
eq_ref	在生成结果表的过程中，针对前面生成的临时表，每次只会从这个临时表中读取一行数据。
ref	在生成结果表的过程中，针对前面生成的临时表，每次会从临时表根据索引来读取数据，作等值查找
range	使用索引作范围查找
index	使用索引遍历所有行
ALL	直接遍历所有行

从性能来说，system > const > eq_ref > ref > range > index > ALL。

一般来说，我们希望应用当中所有的查询方式都应该是eq_ref以上的级别。

下面来补充一个示例：

```
mysql> desc hero;
```

```
+-------+------------+------+-----+---------+-------+
| Field | Type       | Null | Key | Default | Extra |
+-------+------------+------+-----+---------+-------+
| hid   | int        | NO   | PRI | NULL    |       |
| name  | varchar(30) | YES  |     | NULL    |       |
+-------+------------+------+-----+---------+-------+
2 rows in set (0.00 sec)

mysql> desc weapon;
+-------+------------+------+-----+---------+-------+
| Field | Type       | Null | Key | Default | Extra |
+-------+------------+------+-----+---------+-------+
| wid   | int        | NO   | PRI | NULL    |       |
| wname | varchar(30) | YES  |     | NULL    |       |
| hid   | int        | YES  | MUL | NULL    |       |
+-------+------------+------+-----+---------+-------+
3 rows in set (0.00 sec)

explain select * from hero \G
type: ALL
explain select * from hero where hid > 1 \G
type: range
explain select * from hero where hid = 1 \G
type: const
explain select * from hero,weapon where hero.hid = weapon.hid\G
*************************** 1. row ***************************
           id: 1
         type: ALL
*************************** 2. row ***************************
           id: 1
         type: ref

explain select * from weapon,hero where weapon.hid = hero.hid\G
*************************** 1. row ***************************
           id: 1
         type: ALL
*************************** 2. row ***************************
           id: 1
         type: eq_ref
```

5.4 extra

在MySQL执行计划的EXPLAIN结果中，extra列中包含了有关查询执行的额外信息。以下是extra的常见取值及其含义：

extra	含义
Using index	表示查询使用了覆盖索引，即只使用了索引而不需要访问表的实际行数据
Using temporary	表示MySQL需要创建一个临时表来处理查询，通常是因为涉及到了复杂的排序操作或者包含了GROUP BY操作。
Using filesort	表示MySQL需要对结果进行排序操作，并且可能需要使用临时文件来存储排序中间结果。

extra	含义
Using where	表示MySQL在执行查询时应用了WHERE条件。

5.5 索引失效

首先，我们需要先回顾一下explain的结果当中和索引相关的内容：

- possible_keys：可能用到的索引。
- key：实际用到的索引
- key_len：使用索引的最大可能长度。

在使用where设置查询条件时，如果对索引采用一些特殊的操作，往往可能会导致索引失效——这就意味着数据库查询无法有效地使用现有索引来加速查询，而导致性能下降或者需要进行全表扫描的情况。

先来看看对一个拥有整型主键，大表格的读取操作：

```
mysql> explain select * from large_table where id = 10000\G
*************************** 1. row ***************************
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4

# 如果表达式对索引列做了运算，可能会导致索引失效
mysql> explain select * from large_table where id + 10000 = 20000\G
*************************** 1. row ***************************
possible_keys: NULL
          key: NULL
      key_len: NULL
```

接下来来看一个拥有字符串主键的表格的读取操作：

```
desc test3;
+-------+----------+------+-----+---------+-------+
| Field | Type     | Null | Key | Default | Extra |
+-------+----------+------+-----+---------+-------+
| id    | char(30) | NO   | PRI | NULL    |       |
| col1  | int      | YES  |     | NULL    |       |
| col2  | int      | YES  | MUL | NULL    |       |
+-------+----------+------+-----+---------+-------+
mysql> select * from test3;
+-----+------+------+
| id  | col1 | col2 |
+-----+------+------+
| xxx |    6 |    1 |
| aab |    1 |    3 |
| aza |    7 |    3 |
| abc |    2 |    4 |
| bac |    5 |    6 |
| abb |    3 |    7 |
+-----+------+------+
```

```
explain select * from test3 where id = 'aab'\G
*************************** 1. row ***************************
possible_keys: PRIMARY
          key: PRIMARY

explain select * from test3 where col2 = 6 and col1 = 5\G
*************************** 1. row ***************************
possible_keys: myidx
          key: myidx

# 使用了<>导致了主键索引失效
mysql> explain select * from test3 where id <> 'aab'\G
*************************** 1. row ***************************
possible_keys: PRIMARY
          key: myidx

#使用or也导致索引失效了
mysql> explain select * from test3 where id < 'aab' or id > 'add'\G
*************************** 1. row ***************************
possible_keys: PRIMARY
          key: myidx

#使用union效果更好
explain select * from test3 where id < 'aab' union select * from  test3 where  id
> 'add'\G
*************************** 1. row ***************************
possible_keys: PRIMARY,myidx
          key: PRIMARY

*************************** 2. row ***************************
possible_keys: PRIMARY,myidx
          key: PRIMARY

insert into test3 values('123',1,2);

# 尽量避免出现数值到字符串的自动转换，这会导致索引失效
explain select * from test3 where id = 123\G
*************************** 1. row ***************************
possible_keys: PRIMARY
          key: myidx

explain select * from test3 where id = '123'\G
*************************** 1. row ***************************
possible_keys: PRIMARY
          key: PRIMARY
```