第五章 MySQL 索引

索引在MySQL中也叫做 键(key)。是存储引擎用于快速找到记录的一种数据结构。

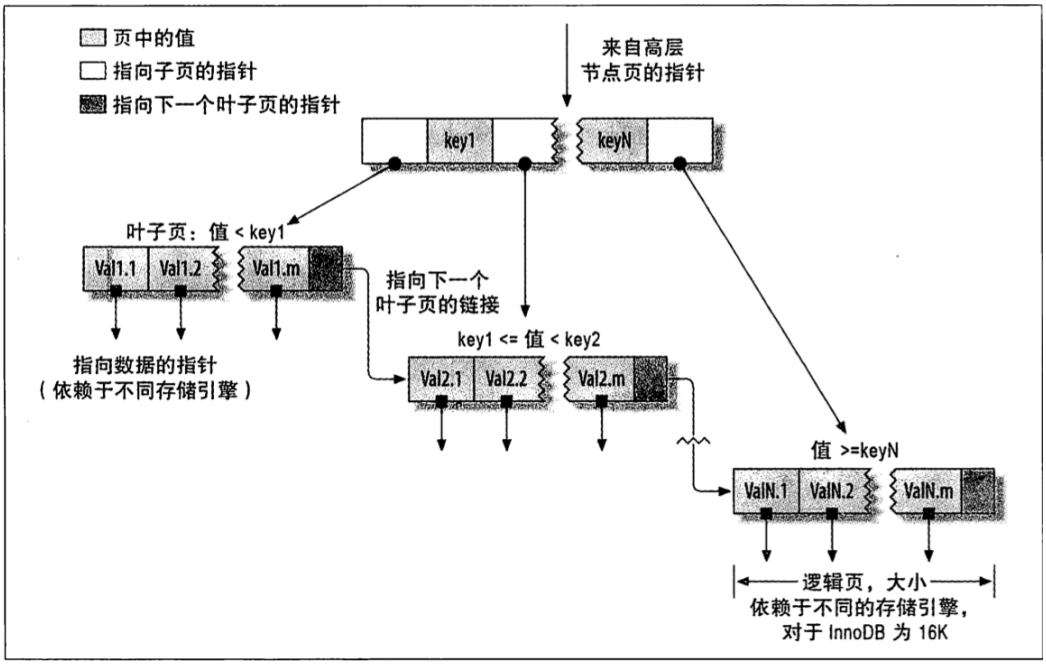
5.1 索引类型

索引有很多类型，可以为不同的场景提供不同的性能。在MySQL中索引是在*存储引擎层*而不是在服务器层实现的。因此并没有统一的标准:

- 不同的存储引擎的索引的工作方式不一样
- 不是所有的存储引擎都支持所有类型的索引
- 不同的存储引擎相同类型的索引的底层实现也不一定相同

5.1.1 B-Tree索引

如果没有特别指明类型，通常所说的索引就是指B-Tree索引，它使用B-Tree数据结构来存储数据[1]。不过底层的存储引擎也可能使用不同的数据结构。例如，NDB集群存储引擎内部实际使用的是T-Tree结构存储这种索引，不过名字用的仍然是BTREE，InnoDB使用的是B+Tree[2]。



B-Tree索引意味着所有的值都是顺序存储的，并且每一个叶子节点到根节点的距离相同。上图展示了InnoDB存储引擎的B-Tree索引的抽象表示(实际使用的是B+Tree结构)。

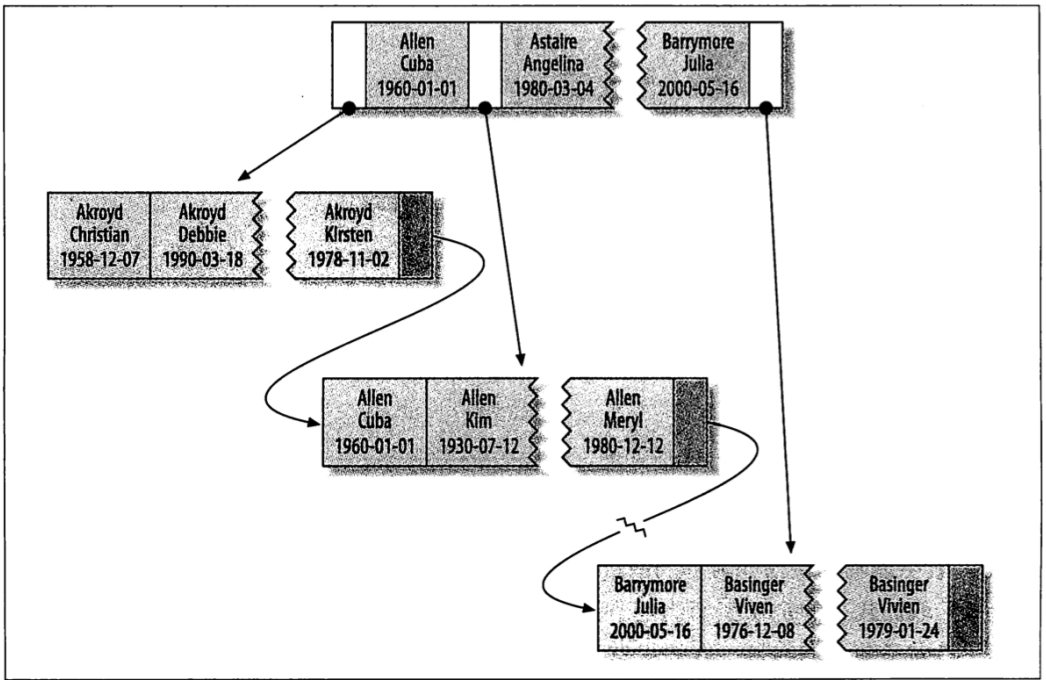
B+Tree的特点是内部节点只存储Key而没有data，叶子节点则没有指针。但是用于数据库索引时都进行了优化，叶子节点添加了指向下一个叶子节点的指针，提高区间查询效率。

B-Tree索引可以加快访问数据的速度，因为存储引擎无需进行全表扫描来获取所需要的数据，取而代之的是从根节点开始进行搜索。

B-Tree对索引列是顺序组织的，所以很适合查找范围数据。例如，在一个基于文本域的索引树上按字母顺序传递连续的值进行查找。假设有如下数据表：

```
CREATE TABLE People (
    last_name  varchar(50) not null,
    first_name varchar(50) not null,
    dob        date        not null,
    gender     enum('m', 'f')not null,
    key(last_name, first_name, dob)
    );
```

其数据示例如下图：



注意：索引对多个值进行排序的依据是**CREATE TABLE**语句中定义索引时列的

顺序。(参考最后一条记录)

可以使用B-Tree索引的查询类型

B-Tree索引适用于：

- 全键值
- 键值范围
- 键前缀查找(只适用于根据最左前缀的查找)

前面的数据表中的索引对如下类型的查询有效：

1. 全值匹配(和索引中所有列比较)
2. 匹配最左前缀 (只使用索引的第一列)
3. 匹配列前缀 (只匹配某一列的值的开头部分，只使用了索引的第一列)
4. 匹配范围值 (例如查找姓在Allen喝Barrymore之间的人， 只使用了索引的第一列)
5. 精确匹配某一列并范围匹配另外一列 (例如查找所有姓为Allen，名字以字母K开头的人，第一列全匹配，第二列范围匹配)
6. 只访问索引的查询 (B-Tree通常支持“只访问索引的查询”，即查询只需访问索引无须访问数据行)

因为索引树中的节点是有序的，因此除了按值查找之外，索引还可以用于查询中的**ORDER BY**操作。

B-Tree索引的限制

B-Tree索引也有其使用限制：

- 如果不是按照索引的最左列开始查找，则无法使用索引。例如上面的例子就无法用于查找名字为Bill的人，也无法查找某个特定生日的人，因为这两列都不是最左数据列。
- 不能跳过索引中的列。也就是说，上面例子中的索引无法用语查找姓为Smith并且出生于某个特定生日的人。即，如果不指明*first_name*则MySQL只能使用索引的第一列。
- 如果查询中有某个列的范围查询，则其右边所有列都无法使用索引优查找。例如：

```
SELECT * from People WHERE last_name='Smith' AND
first_name LIKE 'J%' AND dob ='1976-12-23';
```

这个查询只能使用索引的前两列，因为LIKE是一个范围条件。

由上面的示例可以明白，B-Tree索引的索引列顺序非常重要。所有的限制都和索引列顺序有关。因此，在优化性能的时候可能会需要使用相同的列但是顺序不同的索引来满足不同类型的查询。

5.1.2 哈希索引

哈希索引(Hash Index)基于哈希表实现，只有精确匹配索引所有列的查询才有效。在MySQL中只有**Memory引擎**显式支持哈希索引，这也是Memory引擎表的默认索引类型，Memory引擎同时也支持B-Tree索引。Memory引擎的哈希索引是支持**非唯一哈希索引**的，如果多个列的哈希索值相同，索引会以链表的方式存放多个记录的指针到同一个哈希条目中。

因为哈希索引自身只需存储对应的哈希值，所以索引的结构非常紧凑，这也让哈希索引的查找速度非常快，但是哈希索引也有其缺陷：

- 哈希索引只包含哈希值和行指针，不存储字段值，所以**不能使用索引中的值来避免读取行。**
- 哈希索引数据不是按照索引值顺序存储的，所以无法用于排序。
- 哈希索引不支持部分索引列匹配查找，因为哈希索引始终使用索引列的全部内容来计算哈希值。
- 哈希索引只支持等值比较查询，包括：=, IN(), <=>，也不支持任何范围查找，如WHERE price > 100。
- 可能会出现哈希冲突影响性能，因为出现哈希冲突时引擎需要遍历链表中的所有行指针，逐行比较。
- 如果哈希冲突很多，索引的维护代价会很高。因为当从表中删除一行时，存储引擎需要遍历对应哈希值的链表中的每一行并删除对应行的引用。

自适应哈希索引(Adaptive hash index)

InnoDB引擎中有个特殊功能，当InnoDB中某些索引值被频繁使用时，它会在内存中基于B-Tree索引自动创建一个哈希索引，这样就让B-Tree索引具有了哈希索引的一些特点。

创建自定义哈希索引

模拟InnoDB自动创建哈希索引的方式，用户可以自己创建假的哈希索引。也就是使用哈希值来进行索引查找，需要做的就是在WHERE子句中手动指定使用哈希函数。例如：

```
mysql> SELECT id FROM url WHERE url="http://www.mysql.com"
mysql> SELECT id FROM url WHERE url="http://www.mysql.com"
AND
    -> url_crc=CRC32("http://www.mysql.com")
```

只需删除以前在url上的索引而新增一个url_crc索引列就可以极大的提升速度。

5.1.3 空间数据索引

MyISAM表支持空间索引，可以用作地理位置数据存储。和B-Tree不同，这类索引无须前缀查询。

空间索引会从所有的维度来索引数据，查询时，可以有效的使用任意维度来组合查询。

5.1.4 全文索引

全文索引是一种特殊类型的索引，它查找的是文本中的关键词，而非直接比较索引中的值。

5.2 索引的优点

索引有三个优点：

1. 索引大大减少了服务器需要扫描的数据量
2. 索引可以帮助服务器避免排序和临时表
3. 索引可以将随机I/O变为顺序I/O (B-Tree索引讲相关的列值存储在一起，可以顺序读)

5.3 高性能的索引策略

5.3.1 独立的列

独立的列是指索引列不能是表达式的一部分，也不能是函数的参数。例如下面的查询就无法使用actor_id列的索引：

```
mysql> SELECT actor_id FROM sakila.actor WHERE actor_id + 1
 = 5;
```

我们应该养成简化WHERE条件的习惯，始终将索引列单独放置在比较符号的一侧。下面是另一个常见错误：

```
mysql> SELECT ... WHERE TO_DAYS(CURRENT_DATE) - TO_DAYS(dat
e_col) <= 10;
```

5.3.2 前缀索引和索引选择性

有时候需要索引很长的字符串列，这会让索引变得大而慢。一个策略是前面提到的模拟哈希索引，但是可能还不够。还可以通过索引开始的部分字符串，这样可以大大节约索引空间，从而提高索引效率。但是这样会降低索引的**选择性**[3]

一般情况下某个列前缀的选择性也是足够高的，足以满足查询性能要求。对于**BLOB**、**TEXT**或者很长的**VARCHAR**类型的列，必须使用前缀索引，因为MySQL不允许索引这些列的完整长度。

如何选择前缀长度

通过选择不同的前缀长度计算其与完整列长度的选择性的差值，找到最小的差值的那个长度即可。

```
mysql> SELECT COUNT(*) AS cnt, city FROM sakila.city_demo G
ROUP BY city
    -> ORDER BY cnt DESC LIMIT 10;
mysql> SELECT COUNT(*) AS cnt, LEFT(city, 3) AS pref FROM s
akila.city_demo
    -> GROUP BY pref ORDER by cnt DESC LIMIT 10;
mysql> SELECT COUNT(*) AS cnt, LEFT(city, 7) AS pref FROM s
akila.city_demo
    -> GROUP BY pref ORDER by cnt DESC LIMIT 10;
```

还有一种方法是计算完整列的选择性，并使前缀的选择性接近于完整列的选择性。

```
mysql> SELECT COUNT(DISTINCT city) / COUNT(*) FROM sakila.c
ity_demo;
+-------------------------------+
| COUNT(DISTINCT city)/COUNT(*) |
+-------------------------------+
|                        0.0312 |
+-------------------------------+
```

通常如果前缀的选择性接近0.0312就可以使用了。

```
mysql> SELECT COUNT(DISTINCT LEFT(city, 3))/COUNT(*) AS sel
3,
    ->     COUNT(DISTINCT LEFT(city, 4))/COUNT(*) AS sel4,
    ->     COUNT(DISTINCT LEFT(city, 5))/COUNT(*) AS sel5,
    ->     COUNT(DISTINCT LEFT(city, 6))/COUNT(*) AS sel6,
    ->     COUNT(DISTINCT LEFT(city, 7))/COUNT(*) AS sel7,
    -> FROM sakila.city_demo;
+--------+--------+--------+-------+--------+
| sel3   | sel4   | sel5   | sel6  | sel7   |
+--------+--------+--------+-------+--------+
| 0.0239 | 0.0293 | 0.0305 | 0.0309| 0.0310 |
+--------+--------+--------+-------+--------+
```

当长度达到7时，选择性的增幅已经很小。找到合适的长度之后，下面演示如何创建前缀索引。

```
mysql> ALTER TABLE sakila.city_demo ADD KEY (city(7));
```

前缀索引可以使索引更小、更快，但是也有缺点。MySQL无法使用前缀索引做ORDER BY和GROUP BY，也无法使用前缀索引做覆盖扫描。

5.3.3 多列索引

常见的错误理解：为每个列创建独立的索引，或者按照错误的顺序创建多列索引。

其实，在多个列上创建独立的单列索引大部分情况下并不能提高MySQL的查询

性能。MySQL5.0引入了一种叫做*索引合并(index merge)*的策略，一定程度上可以使用表上的多个单列索引来定位制定的行。更早版本的MySQL职能使用其中某一个列索引，然而这种情况下没有哪一个独立的单列索引是有效的。

可以通过参数*optimizer_switch*来关闭索引合并功能，也可以使用**IGNORE INDEX**提示让优化器忽略掉某些索引列。

5.3.4 选择合适的索引顺序

如何选择索引的列顺序有一个经验法则：将选择性最高的列放在索引最前列。但是这并不是绝对的，避免随机I/O和排序更为重要。

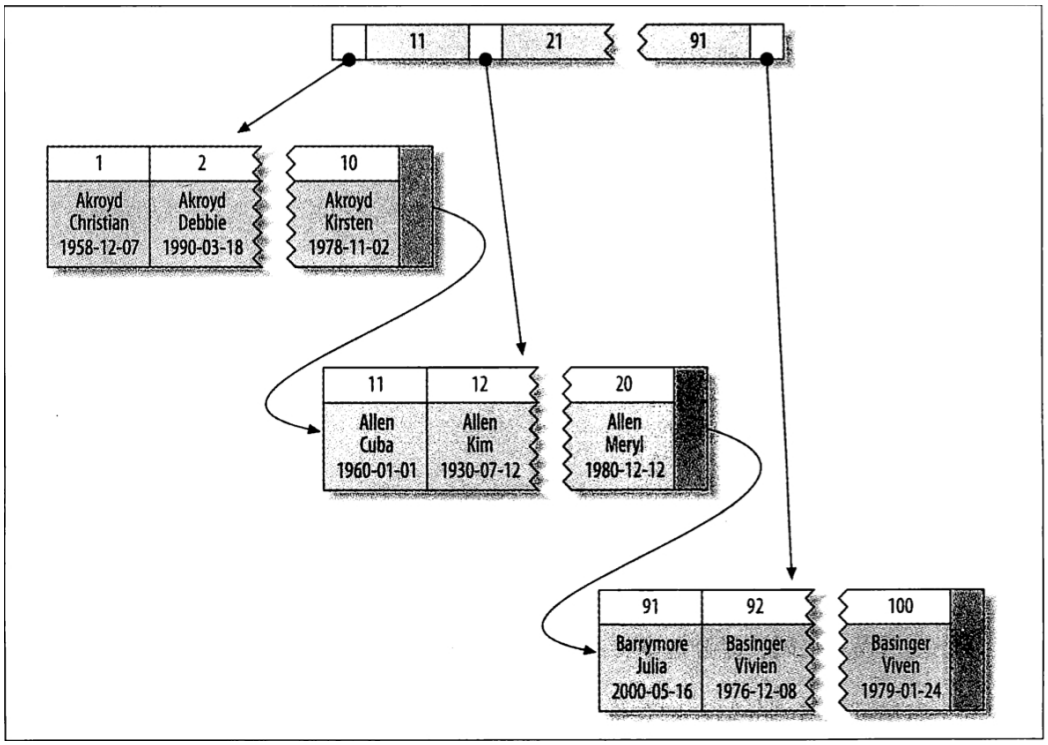
当不需要考虑排序和分组时，将选择性最高的列放前面通常效果很好。

5.3.5 聚簇索引

聚簇索引并不是一种单独的索引类型，而是一种数据存储方式。具体细节依赖于其实现方式，但InnoDB的聚簇索引实际上在同一个数据结构中存储了B-Tree索引和数据行。

当表有聚簇索引时，它的数据行实际上存储在索引的叶子页(leaf page)中。术语*聚簇*表示数据行和相邻的键值紧凑的存储在一起。 因为无法同时把数据行存放在两个不同的地方，所以一个表只能有一个聚簇索引。

因为存储引擎负责实现索引，因此不是所有的存储引擎都支持聚簇索引。下图展示了聚簇索引中的记录是如何存放的。叶子页包含了行的全部数据，但是节点页则只包含了索引列。图中的索引项是整数值。



InnoDB通过主键聚集数据，如果没有定义主键，InnoDB会选择一个唯一的非空索引代替。如果没有这样的索引，InnoDB会隐式定义一个主键来作为聚簇索引。

InnoDB只聚集在同一个页面中的记录，包含相邻键值的页面可能相距很远。

聚簇主键可能对性能有帮助，也可能导致严重的问题。

聚集数据的优点：

- 可以把相关数据保存在一起，减少磁盘读取次数
- 数据访问更快。聚簇索引将索引和数据保存在同一个B-Tree中，因此从聚簇索引中获取数据通常比在非聚簇索引中查找更快
- 使用覆盖索引扫描的查询可以直接使用页节点中的主键值

聚集数据的缺点：

- 聚簇索引最大限度的提高了I/O密集型应用的性能，但是如果数据都放在内存中则没有什么优势了。
- 插入速度严重依赖于插入顺序。按照主键的顺序插入是加载数据到InnoDB表中速度最快的方式。如果不是，那么在加载完成后最好使用**OPTIMIZE TABLE**命令重现组织一下表。

- 更新聚簇索引列的代价很高，因为会强制InnoDB将每个被更新的行移动到新的位置。
- 基于聚簇索引的表插入新行，或者主键被更新导致需要移动行的时候，可能面临*页分裂(page split)*问题。当插入一行到某个已满的页中时，存储引擎会将该页分裂成两个页面来容纳行，这就是一次页分裂操作，页分裂会导致表占用过多的磁盘空间。
- 聚簇索引可能导致全表扫描变慢，尤其行比较稀疏，或者由于页分裂导致数据存储不连续时。
- 二级索引(非聚簇索引)可能比想象的大，因为在二级索引中叶子节点包含了引用行的主键列。
- 二级索引的访问需要两次索引查找，而不是一次。

5.3.5.1 在InnoDB表中按主键顺序插入

如果正在使用的InnoDB表没有数据需要聚集，可以定义一个代理键(surrogate key)作为主键，这种主键的数据应该喝应用无关，最简单的方法是使用**AUTO_INCREAMENT**自增列。这样可以保证数据行是顺序写入，对于根据主键做关联操作的性能会更好。

最好避免随机的(不连续且值的分布范围非常大)聚簇索引，特别是对于I/O密集型的应用。例如，从性能角度考虑，用UUID来作为聚簇索引则会特别糟糕：使得聚簇索引的插入变得完全随机，这是最坏的情况，使得数据没有任何聚集特性。

Tip

顺序的主键什么时候回造成更坏的结果?
答： 对于高并发工作负载，在InnoDB中按主键顺序插入可能会造成明显的竞争。主键的上界会成为*热点*，因为所有的插入都发生在这里，所以并发插入可能导致间隙锁竞争。另一个热点可能是*AUTO_INCREMENT*锁机制，如果遇到这个问题可能需要重新设计表或者应用。

---

1. 很多存储引擎使用B + Tree来存储数据。↩
2. MyISAM使用前缀压缩技术使得索引更小，但是InnoDB则按照原数据格式进行存储。↩
3. 索引的选择性指，不重复的索引值(基数)和数据的记录总数(#T)的比值，范

围从1/#T到1之间。↩