顺序的主键什么时候会造成更坏的结果？

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争用。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，则可能需要考虑重新设计表或者应用，或者更改 innodb_autoinc_lock_mode 配置。如果你的服务器版本还不支持 innodb_autoinc_lock_mode 参数，可以升级到新版本的 InnoDB，可能对这种场景会工作得更好。

5.3.6 覆盖索引

通常大家都会根据查询的 WHERE 条件来创建合适的索引，不过这只是索引优化的一个方面。设计优秀的索引应该考虑到整个查询，而不单单是 WHERE 条件部分。索引确实是一种查找数据的高效方式，但是 MySQL 也可以使用索引来直接获取列的数据，这样就不再需要读取数据行。如果索引的叶子节点中已经包含要查询的数据，那么还有什么必要再回表查询呢？如果一个索引包含（或者说覆盖）所有需要查询的字段的值，我们就称之为“覆盖索引”。

178

覆盖索引是非常有用的工具，能够极大地提高性能。考虑一下如果查询只需要扫描索引而无须回表，会带来多少好处：

- 索引条目通常远小于数据行大小，所以如果只需要读取索引，那 MySQL 就会极大地减少数据访问量。这对缓存的负载非常重要，因为这种情况下响应时间大部分花费在数据拷贝上。覆盖索引对于 I/O 密集型的应用也有帮助，因为索引比数据更小，更容易全部放入内存中（这对于 MyISAM 尤其正确，因为 MyISAM 能压缩索引以变得更小）。
- 因为索引是按照列值顺序存储的（至少在单个页内是如此），所以对于 I/O 密集型的范围查询会比随机从磁盘读取每一行数据的 I/O 要少得多。对于某些存储引擎，例如 MyISAM 和 Percona XtraDB，甚至可以通过 OPTIMIZE 命令使得索引完全顺序排列，这让简单的范围查询能使用完全顺序的索引访问。
- 一些存储引擎如 MyISAM 在内存中只缓存索引，数据则依赖于操作系统来缓存，因此要访问数据需要一次系统调用。这可能会导致严重的性能问题，尤其是那些系统调用占了数据访问中的最大开销的场景。
- 由于 InnoDB 的聚簇索引，覆盖索引对 InnoDB 表特别有用。InnoDB 的二级索引在叶子节点中保存了行的主键值，所以如果二级主键能够覆盖查询，则可以避免对主键索引的二次查询。

在所有这些场景中，在索引中满足查询的成本一般比查询行要小得多。

不是所有类型的索引都可以成为覆盖索引。覆盖索引必须要存储索引列的值，而哈希索引、空间索引和全文索引等都不存储索引列的值，所以 MySQL 只能使用 B-Tree 索引做覆盖索引。另外，不同的存储引擎实现覆盖索引的方式也不同，而且不是所有的引擎都支持覆盖索引（在写作本书时，Memory 存储引擎就不支持覆盖索引）。

当发起一个被索引覆盖的查询（也叫做索引覆盖查询）时，在 EXPLAIN 的 Extra 列可以看到“Using index”的信息[注12]。例如，表 sakila.inventory 有一个多列索引 (store_id, film_id)。MySQL 如果只需访问这两列，就可以使用这个索引做覆盖索引，如下所示：

```
mysql> EXPLAIN SELECT store_id, film_id FROM sakila.inventory\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: inventory
         type: index
possible_keys: NULL
          key: idx_store_id_film_id
      key_len: 3
          ref: NULL
         rows: 4673
        Extra: Using index
```

179

索引覆盖查询还有很多陷阱可能会导致无法实现优化。MySQL 查询优化器会在执行查询前判断是否有一个索引能进行覆盖。假设索引覆盖了 WHERE 条件中的字段，但不是整个查询涉及的字段。如果条件为假（false），MySQL 5.5 和更早的版本也总是会回表获取数据行，尽管并不需要这一行且最终会被过滤掉。

来看看为什么会发生这样的情况，以及如何重写查询以解决该问题。从下面的查询开始：

```
mysql> EXPLAIN SELECT * FROM products WHERE actor='SEAN CARREY'
    -> AND title like '%APOLLO%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: products
         type: ref
possible_keys: ACTOR,IX_PROD_ACTOR
          key: ACTOR
      key_len: 52
          ref: const
         rows: 10
        Extra: Using where
```

注 12：很容易把 Extra 列的“Using index”和 type 列的“index”搞混淆。其实这两者完全不同，type 列和覆盖索引毫无关系；它只是表示这个查询访问数据的方式，或者说是 MySQL 查找行的方式。MySQL 手册中称之为连接方式（join type）。

这里索引无法覆盖该查询，有两个原因：

- 没有任何索引能够覆盖这个查询。因为查询从表中选择了所有的列，而没有任何索引覆盖了所有的列。不过，理论上 MySQL 还有一个捷径可以利用：WHERE 条件中的列是有索引可以覆盖的，因此 MySQL 可以使用该索引找到对应的 actor 并检查 title 是否匹配，过滤之后再读取需要的数据行。
- MySQL 不能在索引中执行 LIKE 操作。这是底层存储引擎 API 的限制，MySQL 5.5 和更早的版本中只允许在索引中做简单比较操作（例如等于、不等于以及大于）。MySQL 能在索引中做最左前缀匹配的 LIKE 比较，因为该操作可以转换为简单的比较操作，但是如果是通配符开头的 LIKE 查询，存储引擎就无法做比较匹配。这种情况下，MySQL 服务器只能提取数据行的值而不是索引值来做比较。

也有办法可以解决上面说的两个问题，需要重写查询并巧妙地设计索引。先将索引扩展至覆盖三个数据列（artist, title, prod_id），然后按如下方式重写查询：

180

```
mysql> EXPLAIN SELECT *
    -> FROM products
    ->    JOIN (
    ->       SELECT prod_id
    ->       FROM products
    ->       WHERE actor='SEAN CARREY' AND title LIKE '%APOLLO%'
    ->    ) AS t1 ON (t1.prod_id=products.prod_id)\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: <derived2>
                ...omitted...
*************************** 2. row ***************************
           id: 1
  select_type: PRIMARY
        table: products
                ...omitted...
*************************** 3. row ***************************
           id: 2
  select_type: DERIVED
        table: products
         type: ref
possible_keys: ACTOR,ACTOR_2,IX_PROD_ACTOR
          key: ACTOR_2
      key_len: 52
          ref:
         rows: 11
        Extra: Using where; Using index
```

我们把这种方式叫做延迟关联（deferred join），因为延迟了对列的访问。在查询的第一阶段 MySQL 可以使用覆盖索引，在 FROM 子句的子查询中找到匹配的 prod_id，然后根

据这些 prod_id 值在外层查询匹配获取需要的所有列值。虽然无法使用索引覆盖整个查询，但总算比完全无法利用索引覆盖的好。

这样优化的效果取决于 WHERE 条件匹配返回的行数。假设这个 products 表有 100 万行，我们来看一下上面两个查询在三个不同的数据集上的表现，每个数据集都包含 100 万行：

1. 第一个数据集，Sean Carrey 出演了 30 000 部作品，其中有 20 000 部的标题中包含了 Apollo。
2. 第二个数据集，Sean Carrey 出演了 30 000 部作品，其中 40 部的标题中包含了 Apollo。
3. 第三个数据集，Sean Carrey 出演了 50 部作品，其中 10 部的标题中包含了 Apollo。

使用上面的三种数据集来测试两种不同的查询，得到的结果如表 5-2 所示。

181 表5-2：索引覆盖查询和非覆盖查询的测试结果

数据集	原查询	优化后的查询
示例 1	每秒 5 次查询	每秒 5 次查询
示例 2	每秒 7 次查询	每秒 35 次查询
示例 3	每秒 2 400 次查询	每秒 2 000 次查询

下面是对结果的分析：

- 在示例 1 中，查询返回了一个很大的结果集，因此看不到优化的效果。大部分时间都花在读取和发送数据上了。
- 在示例 2 中，经过索引过滤，尤其是第二个条件过滤后只返回了很少的结果集，优化的效果非常明显：在这个数据集上性能提高了 5 倍，优化后的查询的效率主要得益于只需要读取 40 行完整数据行，而不是原查询中需要的 30 000 行。
- 在示例 3 中，显示了子查询效率反而下降的情况。因为索引过滤时符合第一个条件的结果集已经很小，所以子查询带来的成本反而比从表中直接提取完整行更高。

在大多数存储引擎中，覆盖索引只能覆盖那些只访问索引中部分列的查询。不过，可以更进一步优化 InnoDB。回想一下，InnoDB 的二级索引的叶子节点都包含了主键的值，这意味着 InnoDB 的二级索引可以有效地利用这些“额外”的主键列来覆盖查询。

例如，sakila.actor 使用 InnoDB 存储引擎，并在 last_name 字段有二级索引，虽然该索引的列不包括主键 actor_id，但也能够用于对 actor_id 做覆盖查询：

```
mysql> EXPLAIN SELECT actor_id, last_name
    -> FROM sakila.actor WHERE last_name = 'HOPPER'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: ref
possible_keys: idx_actor_last_name
          key: idx_actor_last_name
      key_len: 137
          ref: const
         rows: 2
        Extra: Using where; Using index
```

未来 MySQL 版本的改进

182

上面提到的很多限制都是由于存储引擎 API 设计所导致的，目前的 API 设计不允许 MySQL 将过滤条件传到存储引擎层。如果 MySQL 在后续版本能够做到这一点，则可以把查询发送到数据上，而不是像现在这样只能把数据从存储引擎拉到服务器层，再根据查询条件过滤。在本书写作之际，MySQL 5.6 版本（未正式发布）包含了在存储引擎 API 上所做的一个重要的改进，其被称为“索引条件推送（index condition pushdown）”。这个特性将大大改善现在的查询执行方式，如此一来上面介绍的很多技巧也就不再需要了。

5.3.7 使用索引扫描来做排序

MySQL 有两种方式可以生成有序的结果：通过排序操作；或者按索引顺序扫描[注13]；如果 EXPLAIN 出来的 type 列的值为“index”，则说明 MySQL 使用了索引扫描来做排序（不要和 Extra 列的“Using index”搞混淆了）。

扫描索引本身是很快的，因为只需要从一条索引记录移动到紧接着的下一条记录。但如果索引不能覆盖查询所需的全部列，那就不得不每扫描一条索引记录就都回表查询一次对应的行。这基本上都是随机 I/O，因此按索引顺序读取数据的速度通常要比顺序地全表扫描慢，尤其是在 I/O 密集型的工作负载时。

MySQL 可以使用同一个索引既满足排序，又用于查找行。因此，如果可能，设计索引时应该尽可能地同时满足这两种任务，这样是最好的。

只有当索引的列顺序和 ORDER BY 子句的顺序完全一致，并且所有列的排序方向（倒序

注 13：MySQL 有两种排序算法，更多细节可以阅读第 7 章。