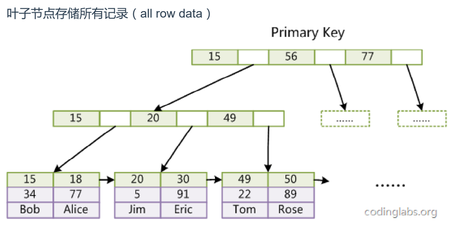
MySQL学习笔记（Day027：Secondary Index）

MySQL学习

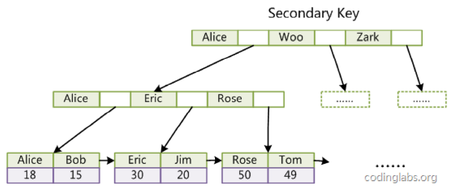
一. Secondary Index（二级索引）

1.1. Secondary Index 介绍

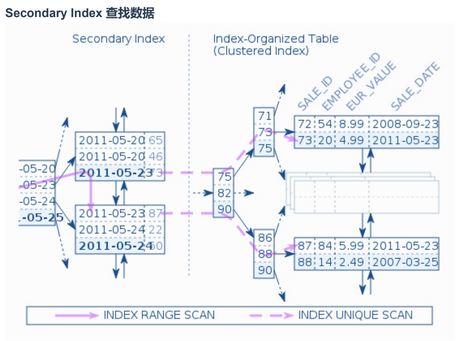
- **Clustered Index（聚集索引）**
  - 叶子节点存储所有记录（all row data）



- **Secondary Index（二级索引）**
  - 也可以称为 非聚集索引
  - 叶子节点存储的是 索引 和 主键 信息
  - 在找到索引后，得到对应的主键，再 回到聚集索引 中找主键对应的记录（row data）
    - Bookmark Lookup（书签查找）
    - 俗称 回表
      - 回表 不止 多 一次IO
      - 而是 多N次 IO（N=树的高度）



- **Secondary Index 查找数据**



1.2. Secondary Index 回表

```
create table userinfo (
    userid int not null auto_increment,
    username varchar(30),
    registdate datetime,
    email varchar(50),
    primary key(userid),
    unique key idx_username(username),
    key idx_registdate(registdate)
);
```

1. 假设查找 username 为Tom，先找二级索引 idx_username ，通过找到 key 为Tom，并得到对应的primary key：userid_a。
2. 得到了userid_a后，再去找聚集索引中userid_a的记录（row data）。
3. 上述一次通过 二级索引 得到 数据 （row data）的 查找过程 ，即为 回表 。
4. 上述过程都是MySQL自动帮你做的。

可以将上述的 userinfo 表进行人工拆分，从而进行 人工回表 ，拆分如下：

```
-- 表1 : 创建一个只有主键userid的表，将原来的二级索引 人工拆分 成独立的表
create table userinfo(
    userid int not null auto_increment,
    username varchar(30),
    registdate datetime,
    email varchar(50),
    primary key(userid)
);

-- 表2：idx_username表，将userid和username作为表的字段，并做一个复合主键 （对应原来的idx_username索引）
create table idx_username(
    userid int not null,
    username varchar(30),
    primary key(username, userid)
);

-- 表3：idx_registdate表，将userid和registdate作为表的字段，并做一个复合主键 （对应原来的idx_registdate 索引）
create table idx_registdate(
    userid int not null,
    registdate datetime,
    primary key(registdate, userid)
);

-- 表4：一致性约束表
create table idx_username_constraint(
    username varchar(30),
    primary key(username)
);

-- 插入数据，使用事物，要么全插，要么全不差
start transaction;
insert into userinfo values(1, 'Tom', '1990-01-01', 'tom@123.com');
insert into idx_username_constraint('Tom');
insert into idx_username(1, 'Tom');
insert into idx_registdate(1, '1990-01-01')
commit;
```

- 假设要查找TOM的 email ：
  1. 先查找 Tom 对应的 userid ，即找的是 idx_username表 （*对应之前就是在idx_username索引中找tom*）
  2. 得到 userid 后，再去 userinfo表 ，通过 userid 得到 email 字段的内容（*对对应之前就是在 聚集索引 中找userid的记录（row data）*）
  3. 上述两次查找就是 人工回表

> 拆表后，就需要开发自己去实现 回表 的逻辑；而开始的一张大表，则是MySQL自动实现该逻辑。

1.3. 堆表的二级索引

1. 在堆表中，是 没有聚集索引 的， 所有的索引都是二级索引 ；
2. 索引的 叶子节点 存放的是 key 和 指向堆中记录的指针 （*物理位置*）

1.4. 堆表和IOT表二级索引的对比

1. 堆表中的二级索引查找 不需要回表 ，且查找速度和 主键索引 一致，因为两者的 叶子节点 存放的都是 指向数据 的 指针 ；反之 IOT表 的的二级索引查找需要回表。
2. 堆表中某条记录（row data）发生更新且 无法原地更新 时，该记录（row data）的物理位置将发生改变；此时， 所有索引 中对该记录的 指针 都需要 更新 （*代价较大*）；反之，IOT表中的记录更新，且 主键没有更新 时， 二级索引 都 无需更新 （*通常来说主键是不更新的*）
   - 实际数据库设计中，堆表的数据无法原地更新时，且在一个 页内有剩余空间 时，原来数据的空间位置不会释放，而是使用指针指向新的数据空间位置，此时该记录对应的所有索引就无需更改了；
   - 如果 页内没有剩余空间 ，所有的索引还是要更新一遍；
3. IOT表页内是有序的，页与页之间也是有序的，做range查询很快。

1.5. index with included column（含列索引）

在上面给出的 userinfo 的例子中，如果要查找某个 用户的email ，需要回表，如何不回表进行查询呢？

1. 方案一 ：复合索引

```
-- 表结构
create table userinfo (
    userid int not null auto_increment,
    username varchar(30),
    registdate datetime,
    email varchar(50),
    primary key(userid),
    unique key idx_username(username, email),  -- 索引中有email，可以直接查到，不用回表
    key idx_registdate(registdate)
);

-- 查询
select email from userinfo where username='Tom';
```

> 该方案可以做到 只查一次 索引就可以得到用户的email，但是 复合索引 中username和email都要 排序
> 而 含列索引 的意思是索引中 只对username 进行排序，email是不排序的，只是带到索引中，方便查找。

2. 方案二：拆表

```
create table userinfo (
    userid int not null auto_increment,
    username varchar(30),
    registdate datetime,
    email varchar(50),
    primary key(userid),
    key idx_registdate(registdate)
);

create table idx_username_include_email (
    userid int not null,
    username varchar(30),
    email varchar(50),
    primary key(username, userid),
    unique key(username)
);

-- 两个表的数据一致性可以通过事物进行保证
```

> 通过拆表的方式，查找 idx_username_include_email 表，既可以通过 username 找到 email ，但是需要告诉研发，如果想要通过useranme得到email，查这张表速度更快，而不是查userinfo表

对于含有多个索引的IOT表，可以将索引拆成不同的表，进而提高查询速度
但是实际使用中，就这个例子而言，使用复合索引，代价也不会太大。

二. Multi-Range Read（MRR）

2.1. 回表的代价

```
mysql> alter table employees add index idx_date (hire_date);  -- 给 employees 增加一个索引
mysql> show create table employees\G
*************************** 1. row ***************************
       Table: employees
Create Table: CREATE TABLE `employees` (
  `emp_no` int(11) NOT NULL,
  `birth_date` date NOT NULL,
  `first_name` varchar(14) NOT NULL,
  `last_name` varchar(16) NOT NULL,
  `gender` enum('M','F') NOT NULL,
  `hire_date` date NOT NULL,
  PRIMARY KEY (`emp_no`),
  KEY `idx_date` (`hire_date`)  -- 新增的索引
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
1 row in set (0.00 sec)

-- 查询语句1
mysql> select * from employees where emp_no between 10000 and 20000; -- 主键查找1W条数据

-- 查询语句2
mysql> select * from employees where hire_date >= '1990-01-01' limit 10000; -- select * 操作，每次查找需要回表
```

1. 对于 查询语句1 ，假设一个页中有100条记录，则只需要100次IO；
2. 对于 查询语句2 ，此次查询中，假设 聚集索引 和 hire_date索引 （二级索引）的高度都是 3 ，且查找 1W 条（*假设不止1W条*），则需要查询的IO数为 (3+N)+3W
    - 3 为 第一次 找到 hire_date>=1990-01-01 所在的页（二级索引）的IO次数
    - N 为从第一次找到的页 往后 读页的IO次数（注意二级索引也是连续的， 不需要 从根再重新查找）
        - 所以 3+N 就是在 hire_date （二级索引）中读取IO的次数
    - 3W 为在IOT表中进行 回表 的次数
3. 在MySQL5.6之前，实际使用过程中，优化器可能会选择直接进行 扫表 ，而 不会 进行如此多的回表操作。

2.2. MRR 介绍

> **MRR**：针对 物理访问 ，随机转顺序，空间换时间。

1. 开辟一块 内存 空间作为cache
    - 默认为 32M ，注意是 线程级 的，不建议设置的很大；

```
mysql> show variables like "%read_rnd%";
+----------------------+----------+
| Variable_name        | Value    |
+----------------------+----------+
| read_rnd_buffer_size | 33554432 |  -- 32M
+----------------------+----------+
1 row in set (0.00 sec)
```

2. 将 需要回表 的 主键 放入上述的 内存 空间中（*空间换时间*），放满 后进行 排序 （*随机转顺序*）；
3. 将 排序 好数据（主键）一起进行回表操作，以提高性能；
    - 在 IO Bound 的SQL场景下，使用MRR比不使用MRR系能 提高 将近 10倍 （*磁盘性能越低越明显*）；
    - 如果数据都在内存中，MRR的帮助不大， 已经在内存 中了，不存在随机读的概念了（随机读主要针对物理访问）

> SSD 仍然需要开启该特性，多线程下的随机读确实很快，但是我们这里的操作是一条SQL语句，是 单线程 的，所以 顺序 的访问还是比 随机 访问要 更快 。

```
mysql> show variables like 'optimizer_switch'\G
*************************** 1. row ***************************
Variable_name: optimizer_switch
        Value: index_merge=on,index_merge_union=on,index_merge_sort_union=on,index_merge_intersection=on,engine_condition_pushdown=on,index_condition_pushdown=on,mrr=on,mrr_cost_based=on,block_nested_loop=on,batched_key_access=off,materialization=on,semijoin=on,loosescan=on,firstmatch=on,duplicateweedout=on,subquery_materialization_cost_based=on,use_index_extensions=on,condition_fanout_filter=on,derived_merge=on
1 row in set (0.00 sec)

-- 其中MRR默认是打开的 mrr=on，不建议关闭

mysql> explain select * from employees where hire_date >= '1990-01-01';
+----+-------------+-----------+------------+------+---------------+------+---------+------+--------+----------+-------------+
| id | select_type | table     | partitions | type | possible_keys | key  | key_len | ref  | rows   | filtered | Extra       |
+----+-------------+-----------+------------+------+---------------+------+---------+------+--------+----------+-------------+
|  1 | SIMPLE      | employees | NULL       | ALL  | idx_date      | NULL | NULL    | NULL | 298124 |    50.00 | Using where |
+----+-------------+-----------+------------+------+---------------+------+---------+------+--------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
-- 虽然mrr=on打开了，但是没有使用MRR


mysql> set optimizer_switch='mrr_cost_based=off'; -- 将该值off，不让MySQL对MRR进行成本计算（强制使用MRR）
Query OK, 0 rows affected (0.00 sec)

mysql> explain select * from employees where hire_date >= '1990-01-01';
+----+-------------+-----------+------------+-------+---------------+----------+---------+------+--------+----------+-----------------------------------+
| id | select_type | table     | partitions | type  | possible_keys | key      | key_len | ref  | rows   | filtered | Extra                             |
+----+-------------+-----------+------------+-------+---------------+----------+---------+------+--------+----------+-----------------------------------+
|  1 | SIMPLE      | employees | NULL       | range | idx_date      | idx_date | 3       | NULL | 149062 |   100.00 | Using index condition; Using MRR |
+----+-------------+-----------+------------+-------+---------------+----------+---------+------+--------+----------+-----------------------------------+
1 row in set, 1 warning (0.00 sec)
-- 使用了MRR
```

三. 求B+树的高度

如上图所示，每个页的 Page Header 中都包含一个 PAGE_LEVEL 的信息，表示该页所在B+树中的层数， 叶子节点 的PAGE_LEVEL为 0 。
所以树的 高度 就是 root页 的 PAGE_LEVEL + 1

3.3. PAGE_LEVEL

从一个页的 第64字节 开始读取，然后再读取 2个字节 ，就能得到 PAGE_LEVEL 的值

3.4. 获取root页

> 官方文档

```
mysql> use information_schema;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed

mysql> desc INNODB_SYS_INDEXES;
+----------------+---------------------+------+-----+---------+-------+
| Field          | Type                | Null | Key | Default | Extra |
+----------------+---------------------+------+-----+---------+-------+
| INDEX_ID       | bigint(21) unsigned | NO   |     | 0       |       |
| NAME           | varchar(193)        | NO   |     |         |       |
| TABLE_ID       | bigint(21) unsigned | NO   |     | 0       |       |
| TYPE           | int(11)             | NO   |     | 0       |       |
| N_FIELDS       | int(11)             | NO   |     | 0       |       |
| PAGE_NO        | int(11)             | NO   |     | 0       |       |
| SPACE          | int(11)             | NO   |     | 0       |       |
| MERGE_THRESHOLD | int(11)            | NO   |     | 0       |       |
+----------------+---------------------+------+-----+---------+-------+
8 rows in set (0.00 sec)

mysql> select * from INNODB_SYS_INDEXES where space<>0 limit 1\G
*************************** 1. row ***************************
       INDEX_ID: 18
           NAME: PRIMARY
       TABLE_ID: 16
           TYPE: 3
       N_FIELDS: 1
        PAGE_NO: 3  -- 根据官方文档，该字段就是B+树root页的PAGE_NO
          SPACE: 5
MERGE_THRESHOLD: 50
1 row in set (0.01 sec)

-- 没有table的name，只有ID

mysql> select b.name , a.name, index_id, type, a.space, a.PAGE_NO
    -> from INNODB_SYS_INDEXES as a, INNODB_SYS_TABLES as b
    -> where a.table_id = b.table_id
    -> and a.space <> 0 and b.name like "dbt3/%"; -- 做一次关联
+----------------------+-----------------------+----------+------+-------+---------+
| name                 | name                  | index_id | type | space | PAGE_NO |
+----------------------+-----------------------+----------+------+-------+---------+
| dbt3/customer        | PRIMARY               |       64 |    3 |    43 |       3 |
| dbt3/customer        | i_c_nationkey         |       65 |    0 |    43 |       4 |
| dbt3/lineitem        | PRIMARY               |       66 |    3 |    44 |       3 |
| dbt3/lineitem        | i_l_shipdate          |       67 |    0 |    44 |       4 |
| dbt3/lineitem        | i_l_suppkey_partkey   |       68 |    0 |    44 |       5 |
| dbt3/lineitem        | i_l_partkey           |       69 |    0 |    44 |       6 |
| dbt3/lineitem        | i_l_suppkey           |       70 |    0 |    44 |       7 |
| dbt3/lineitem        | i_l_receiptdate       |       71 |    0 |    44 |       8 |
| dbt3/lineitem        | i_l_orderkey          |       72 |    0 |    44 |       9 |
| dbt3/lineitem        | i_l_orderkey_quantity |       73 |    0 |    44 |      10 |
| dbt3/lineitem        | i_l_commitdate        |       74 |    0 |    44 |      11 |
| dbt3/nation          | PRIMARY               |       75 |    3 |    45 |       3 |
| dbt3/nation          | i_n_regionkey         |       76 |    0 |    45 |       4 |
| dbt3/orders          | PRIMARY               |       77 |    3 |    46 |       3 |
| dbt3/orders          | i_o_orderdate         |       78 |    0 |    46 |       4 |
| dbt3/orders          | i_o_custkey           |       79 |    0 |    46 |       5 |
| dbt3/part            | PRIMARY               |       80 |    3 |    47 |       3 |
| dbt3/partsupp        | PRIMARY               |       81 |    3 |    48 |       3 |
| dbt3/partsupp        | i_ps_partkey          |       82 |    0 |    48 |       4 |
| dbt3/partsupp        | i_ps_suppkey          |       83 |    0 |    48 |       5 |
| dbt3/region          | PRIMARY               |       84 |    3 |    49 |       3 |
| dbt3/supplier        | PRIMARY               |       85 |    3 |    50 |       3 |
| dbt3/supplier        | i_s_nationkey         |       86 |    0 |    50 |       4 |
| dbt3/time_statistics | GEN_CLUST_INDEX       |       87 |    1 |    51 |       3 |
+----------------------+-----------------------+----------+------+-------+---------+
24 rows in set (0.00 sec)
-- 聚集索引页的root页的PAGE_NO一般就是3
```

3.5. 读取PAGE_LEVEL

```
mysql> select count(*) from dbt3.lineitem;
+----------+
| count(*) |
+----------+
|  6001215 |
+----------+
1 row in set (5.68 sec)


shell> hexdump -h
hexdump: invalid option -- 'h'
hexdump: [-bcCdovx] [-e fmt] [-f fmt_file] [-n length] [-s skip] [file ...]
shell> hexdump  -s 24640 -n 2 -Cv lineitem.ibd
00006040  00 02                                           |..|
00006042
```

1. `24640 = 8192 * 3 + 64`
    - 其中 `8192` 是我的页大小
    - `root页` 的 `PAGE_NO` 为 `3` ，表示是 `第4个页` ，则需要 `跳过` 前面 `3个页` ，才能 `定位到root页` ，所以要 `*3`
    - 然后加上 `64` 个字节的偏移量，即可定位到 `PAGE_LEVEL`
2. `-n 2` 表示读取的字节数，这里读取 `2个字节` ，即可以读到 `PAGE_LEVEL`

根据上述 `hexdump` 的结果，root页中的 `PAGE_LEVEL` 为2，表示该索引的高度为 `3` （*从0开始计算*）