实验报告 5：Index 创建以及性能比较

课程名称：数据库系统实践	**年级**：2018 级	**上机实践成绩**：
指导教师：赵慧	**姓名**：陈俊潼	
上机实践名称：SQL 实践	**学号**：10185101210	**上机实践日期**：2020.5.28
上机实践编号：5	**组号**：第 12 小组	**上机实践时间**：10:00-11:30

一、目的

- 理解索引，正确使用索引

二、内容与设计思想

- 在大数据量的情况下，针对不同类型的查询建立不同类型的索引，并通过实验验证建立索引之后查询性能的差异。
- 掌握索引适用的条件，掌握不同类型的索引的适用范围。

三、使用环境

macOS 10.15.3
MySQL 8.0.19
Navicat Premium 15.0.12

四、实验过程

该部分具有大量代码，为了便于排版和方便阅读，我使用了 Markdown 编写。见后续页面。

五、总结

经过本次实验，我了解到了索引

六、附录

无。

Database Lab 5: Index

10185101210 陈俊潼

配置信息

项	值
系统版本	macOS 10.5.4
MySQL 版本	8.0.19
计算机内存	16 GB
计算机处理器	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz
SSD 读取速度（单测）	2.25 GB / s

建立数据

首先使用下列语句新建表：

```
CREATE TABLE personHealthInfo (
  pNo varchar(18) NOT NULL,
  pName varchar(20),
  location varchar(20),
  healthStatus varchar(20))
```

接着新建运行过程：

```
CREATE PROCEDURE generateData(num int)
BEGIN
    declare pno varchar(18);
    declare pname varchar(20);
    declare location varchar(20);
    declare healthStatus varchar(20);
    declare counter   bigint;
    declare location_set varchar(200);
    declare health_set varchar(200);
    set location_set = 'Shanghai,Beijing,Guangzhou';
    set health_set = 'Health,Uncertain,Diagnosis,Cure';
    set counter=1;
    while counter<num do
        set pno=ltrim(cast(floor(rand(counter)*1000000000) as char));
        set pname=concat('p',pno);
        set location= substring_index(substring_index(location_set, ',', floor(1 + (rand() * 3))),',',-1);
        set healthStatus = substring_index(substring_index(health_set, ',', floor(1 + (rand() * 4))),',',-1);
```

```
        insert into personHealthInfo(pNo,pName,location, healthStatus)
 values(pno,pname,location, healthStatus);
        set counter=counter+1;
    end while;
END;
```

然后调用该过程可以创建不同数量的数据。尝试运行：

```
CALL generateData(1000000)
```

可以看到，创建 1000000 条数据花费的时间为 316.96 秒。

sql	message
CALL generateData()	OK, Time: 315.969000s

查询结果

通过查阅 MySQL 的官方文档：

15.6.2.1 Clustered and Secondary Indexes

Every `InnoDB` table has a special index called the clustered index where the data for the rows is stored. Typically, the clustered index is synonymous with the primary key. To get the best performance from queries, inserts, and other database operations, you must understand how `InnoDB` uses the clustered index to optimize the most common lookup and DML operations for each table.

- When you define a `PRIMARY KEY` on your table, `InnoDB` uses it as the clustered index. Define a primary key for each table that you create. If there is no logical unique and non-null column or set of columns, add a new auto-increment column, whose values are filled in automatically.
- If you do not define a `PRIMARY KEY` for your table, MySQL locates the first `UNIQUE` index where all the key columns are `NOT NULL` and `InnoDB` uses it as the clustered index.
- If the table has no `PRIMARY KEY` or suitable `UNIQUE` index, `InnoDB` internally generates a hidden clustered index named `GEN_CLUST_INDEX` on a synthetic column containing row ID values. The rows are ordered by the ID that `InnoDB` assigns to the rows in such a table. The row ID is a 6-byte field that increases monotonically as new rows are inserted. Thus, the rows ordered by the row ID are physically in insertion order.

在 MySQL 中无法使用 `CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX index_name ON table_name(column_name)` 的方式直接显式指定创建聚簇索引索引或者费聚簇索引索引。而是在定义了一个主键后，MySQL 会自动把这个主键作为聚簇索引。所以在接下来的实验中，将先不建立索引进行查询语句统计时间，再为 `pNo` 建立一个主键，进行余下的操作。

值得一提的是，在未建立主键之前，MySQL 会自动查询到第一个 `UNIQUE` 的索引，并基于索引列建立聚簇索引，因此在建立非聚簇索引的时候，我将不会指定 `UNIQUE` 字段。

于是编写 SQL 脚本如下：

```
DELETE FROM personHealthInfo;
CALL generateData(5000);

-- No index, equivelent
SELECT * FROM personHealthInfo WHERE pNo = 406135974;
-- No index, range
SELECT * FROM personHealthInfo WHERE pNo < 406135974;

CREATE INDEX index1 ON personHealthInfo(pNo);
```

```sql
-- Non-cluster index on pNo, equivalent
SELECT * FROM personHealthInfo WHERE pNo = 406135974;
-- Non-cluster index on pNo, range
SELECT * FROM personHealthInfo WHERE pNo < 406135974;
DROP INDEX index1 on personHealthInfo;

ALTER TABLE personHealthInfo ADD PRIMARY KEY (pNo);
-- Cluster index on pNo, equivalent
SELECT * FROM personHealthInfo WHERE pNo = 406135974;
-- Cluster index on pNo, range
SELECT * FROM personHealthInfo WHERE pNo < 406135974;

CREATE INDEX index1 ON personHealthInfo(location);
-- Cluster on pNo & Non-cluster on location
SELECT * FROM personHealthInfo WHERE location = 'Shanghai';
DROP INDEX index1 on personHealthInfo;

CREATE INDEX index1 ON personHealthInfo(healthStatus);
-- Cluster on pNo & Non-cluster on health
SELECT healthStatus, count(*) FROM personHealthInfo WHERE location = 'Shanghai'
GROUP BY healthStatus;
DROP INDEX index1 on personHealthInfo;

ALTER TABLE personHealthInfo DROP PRIMARY KEY;
```

例如，使用 5000 条数据可以得到来自 Navicat 的一下 Message，得到对应语句的执行时间：

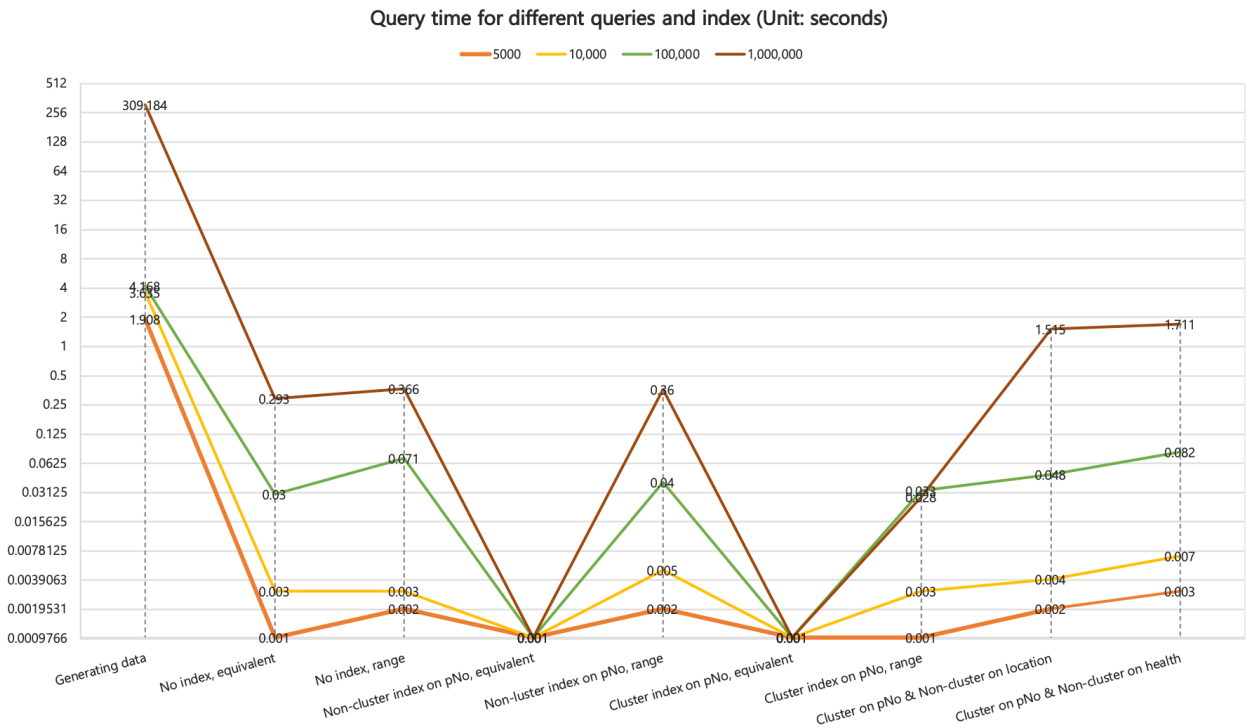
Message | Result 1 | Result 2 | Result 3 | Result 4 | Result 5 | Result 6 | Result 7 | Result 8 | Profile | Status

sql	message
DELETE FROM personHealthInfo	Affected rows: 99999, Time: 0.501000s
CALL generateData(1000000)	OK, Time: 309.184000s
-- No index, equivelent	OK, Time: 0.293000s
-- No index, range	OK, Time: 0.366000s
CREATE INDEX index1 ON personHealthInfo(pNo)	OK, Time: 1.286000s
-- Non-cluster index on pNo, equivalent	OK, Time: 0.000000s
-- Non-cluster index on pNo, range	OK, Time: 0.360000s
DROP INDEX index1 on personHealthInfo	OK, Time: 0.012000s
ALTER TABLE personHealthInfo ADD PRIMARY KEY (pNo)	OK, Time: 2.710000s
-- Cluster index on pNo, equivalent	OK, Time: 0.000000s
-- Cluster index on pNo, range	OK, Time: 0.280000s
CREATE INDEX index1 ON personHealthInfo(location)	OK, Time: 2.069000s
-- Cluster on pNo & Non-cluster on location	OK, Time: 1.515000s
DROP INDEX index1 on personHealthInfo	OK, Time: 0.009000s
CREATE INDEX index1 ON personHealthInfo(healthStatus)	OK, Time: 1.907000s
-- Cluster on pNo & Non-cluster on health	OK, Time: 1.711000s
DROP INDEX index1 on personHealthInfo	OK, Time: 0.006000s
ALTER TABLE personHealthInfo DROP PRIMARY KEY	OK, Time: 3.437000s

于是分别以 5,000 条，10,000 条，100,000 条， 1,000,000 条数据进行三次实验，可以得到以下结果，单位均为秒：

Subject (Time unit: seconds)	5000	10,000	100,000	1,000,000
Generating data	1.908	3.635	4.168	309.184
No index, equivalent	0.001	0.003	0.03	0.293
No index, range	0.002	0.003	0.071	0.366
Non-cluster index on pNo, equivalent	0.001	0.001	0.001	0.000
Non-luster index on pNo, range	0.002	0.005	0.040	0.360
Cluster index on pNo, equivalent	0.000	0.000	0.000	0.000
Cluster index on pNo, range	0.001	0.003	0.033	0.028
Cluster on pNo & Non-cluster on location	0.002	0.004	0.048	1.515
Cluster on pNo & Non-cluster on health	0.003	0.007	0.082	1.711

对结果作图表如下（纵轴为指数坐标轴）：



以上结果全部通过在 LOCALHOST 上运行的 MySQL 服务器得到，避免了网络传输的不稳定性对结果造成的影响。

可以看见，在建立了非聚簇索引之后，等值搜索的速度立即下降到接近于 0。而对于范围搜索，在中型和大型的数据量下，还是可以看见有轻微的下降结果。

进一步地，对 `pNo` 建立聚簇索引之后，可以发现范围搜索的速度相比非聚类的速度又下降了许多，100 万条数据的情况下从 0.36 秒下降到来了 0.028 秒。

而对于最后两条二级索引，经过试验，第一次查询会经过较长时间（如图、表所示），用时 1.5 秒。不过由于数据库的缓存机制，在第二次查询的时候，速度就可以提升到 0.026 秒左右。为了对比该查询不经过索引的查询时间，我重新运行了一次语句，但只对 `pNO` 建立聚簇索引而不对 `healthStatus` 或 `location` 建立索引。可以得到最后两条语句的查询时间分别为：

sql	message
-- Cluster on pNo & Non-cluster on location	OK, Time: 0.401000s
-- Cluster on pNo & Non-cluster on health	OK, Time: 0.470000s

发现运行时间反倒比之前的更短了。经过探索，我认为原因在于如果对 `healthStatus` 或 `location` 建立了非聚簇索引，那么数据库在查询到涉及这两个字段的查询时就会从非聚簇索引中查找到所有满足条件的子节点。然而，非聚簇索引的子节点在磁盘中并不是顺序存储的，大量的随机访问会产生很多不必要的磁盘读取时间。相应的，如果不对 `healthStatus` 或 `location` 建立非聚簇索引，那么数据库会从 `pNo` 的聚簇索引依次遍历并判断是否满足查询要求。由于聚簇索引是顺序读取，磁盘的读取速度将会大大提高，从而得到更快的查询速度。

更多思考

索引创建的规则

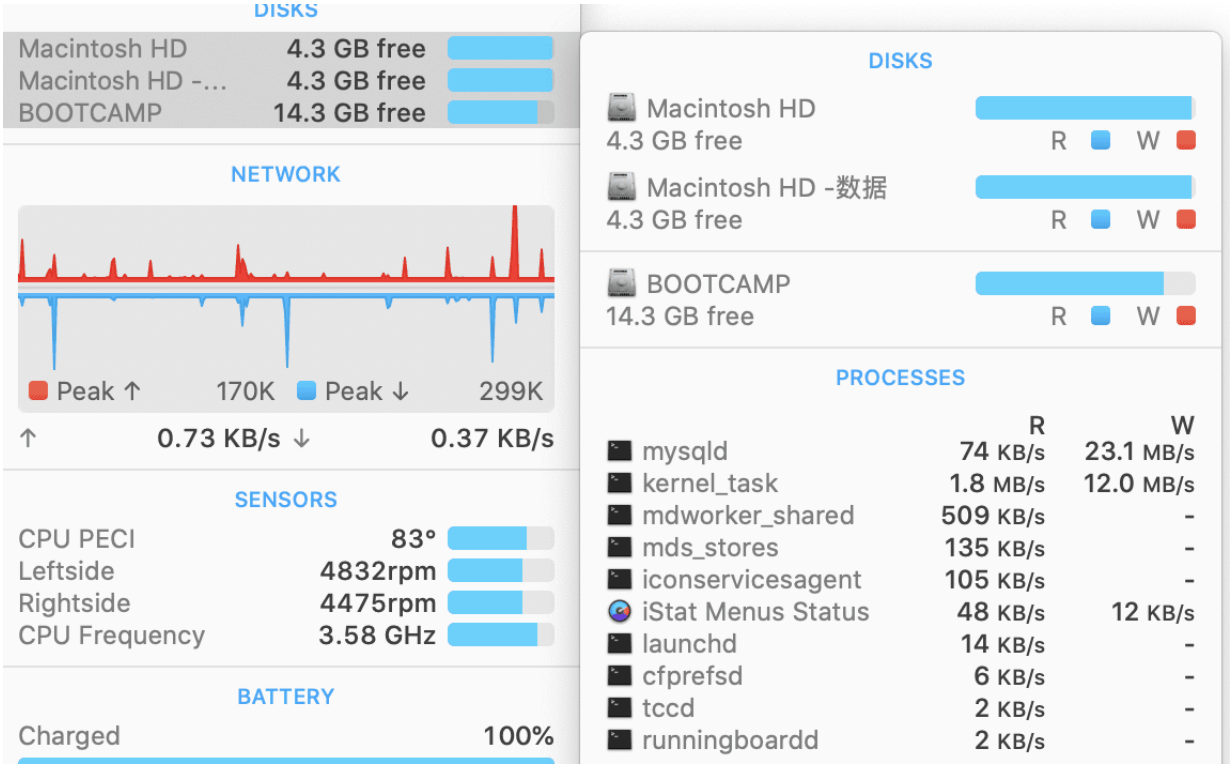
1. MySQL 默认会为第一个 `UNIQUE` 的非空索引栏创建聚簇索引。
2. 通常对于数据量较大的情况，创建一个用于查找的索引是必须的。这样可以有效地减少查找时间，提高查找效率。
3. 如果已经有了一个聚簇索引，要谨慎地选择第二索引。除非对第二索引的查询需求非常大，否则不推荐在数据库中建立第二索引，这样有可能反而会降低查询性能。

自增主键上的聚簇索引

这样的聚簇索引是有意义的。聚簇索引决定了数据项在存盘中的存储顺序。对自增主键进行聚簇索引，意味着一系列的主键将会是序列存储的，这对查询和更新操作都非常有利，可以使用二分查找等高效率的方式维护数据。

连续操作的性能要求

在创建一百万条数据时，计算机甚至会出现假死的情况。检查系统运行情况：



除了发现 CPU 温度急剧上升外，也能看到 MySQL 占用了大量 CPU 资源并不断写入磁盘和读取磁盘数据。

实验统计数据及图表源文件：

https://billc.oss-cn-shanghai.aliyuncs.com/file/2020-05-28-LAB5.xlsx