

华东师范大学软件工程学院实验报告

实验课程：数据库系统及其应用实践

年级：2023 级

实验成绩：

实验名称：Lab-05

姓名：顾翌炜

实验编号：Lab-05

学号：10235101527

实验日期：2025/05/15

指导教师：姚俊杰

组号：01

实验时间：2 课时

目录

1 实验目标	2
2 实验要求	2
3 实验过程记录	2
3.1 启动 MySQL 数据库	2
3.2 通过 PowerShell 连接到 MySQL	3
3.3 执行指定 SQL 语句	3
3.4 观察 Insert 查询性能	4
3.5 观察 SELECT with INDEX 查询性能	7
3.6 观察 SELECT with INDEX – ADVANCED 查询性能	10
3.7 观察 SORT with INDEX 查询性能	17
3.8 索引性能测试 - 导入数据库	19
3.9 索引性能测试 - 数据集重新整理	21
3.10 索引性能测试 - 测试	22
3.10.1 等值查询，并包含无索引和唯一索引	22
3.10.2 范围查询，并包含无索引和非唯一索引	23
3.10.3 不同数据集大小与查询结果大小	24
4 存在的问题及解决方案	26
4.1 存在的问题：	26
4.2 解决方案：	27

5 实验小结	27
5.1 实验目的	27
5.2 实验方法	27
5.3 实验结果	27
5.4 实验小结	28

1 实验目标

本实验旨在帮助学生深入理解数据库存储结构和索引机制，掌握索引的创建与使用方法，分析存储与索引对数据库性能的影响，提升数据库管理能力。通过实验，学生将能够熟练运用 MySQL 数据库进行存储管理与索引优化，为实际数据库应用开发和管理提供理论与实践基础。

2 实验要求

- 1) **深入探究存储原理：**通过本次实验，学生能够深入了解数据库中数据的物理存储方式，包括数据文件、日志文件等的组织形式。例如，了解关系型数据库中表是如何以页或块的形式存储在磁盘上的，以及这些存储单位如何进行空间分配和管理，从而为后续学习数据库的高效操作奠定基础。
- 2) **掌握存储参数配置：**使学生熟悉数据库存储相关的参数配置，如表空间大小、数据块大小等。以 Oracle 数据库为例，学生将学会如何根据实际应用需求合理设置表空间的大小，以及如何调整数据块大小以优化存储空间利用率和数据读写性能，从而更好地管理和维护数据库存储环境。
- 3) **创建索引的实践操作：**让学生熟练掌握在数据库中创建不同类型索引（如 B 树索引、哈希索引等）的方法。以 MySQL 数据库为例，学生将学会使用 SQL 语句创建单列索引、组合索引，并理解在什么情况下适合创建哪种类型的索引。例如，在一个包含大量用户信息的表中，如果经常根据用户姓名进行查询，就可以创建一个基于用户姓名列的 B 树索引，以加快查询速度。
- 4) **优化查询性能：**使学生能够通过合理使用索引显著提高数据库查询性能。通过实验，学生将了解到索引是如何帮助数据库管理系统快速定位数据的。例如，在一个大型电商数据库中，对于频繁查询商品价格和库存的场景，通过在价格和库存字段上创建合适的索引，可以大大减少查询时需要扫描的数据量，从而将查询响应时间从数秒缩短到毫秒级别，提升用户体验。

3 实验过程记录

3.1 启动 MySQL 数据库

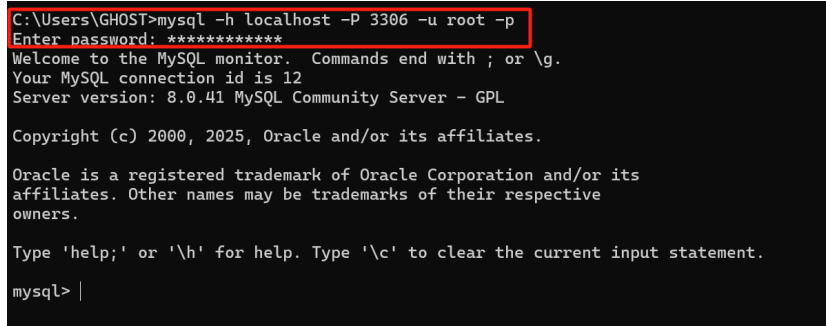
在数据库管理工具中启动数据库

3.2 通过 PowerShell 连接到 MySQL

通过在命令行输入以下指令，通过 PowerShell 连接到 MySQL

通过 PowerShell 连接到 MySQL

```
1 mysql -h localhost -P 3306 -u root -p
```



```
C:\Users\GHOST>mysql -h localhost -P 3306 -u root -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 12
Server version: 8.0.41 MySQL Community Server - GPL

Copyright (c) 2000, 2025, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> |
```

图 1: 连接数据库

3.3 执行指定 SQL 语句

执行以下 SQL 语句，创建一个名为 mydb 的数据库，同时在该模式下创建数据表 mytakes 和 mysection

创建 db 数据库

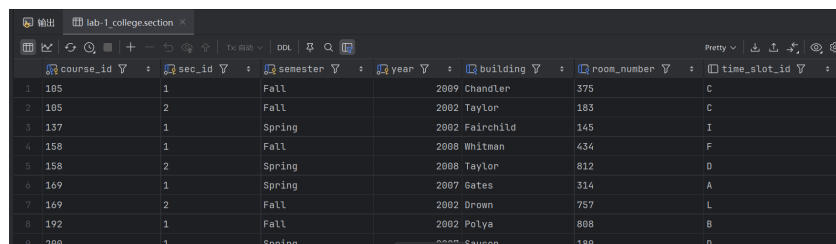
```
1 DROP DATABASE IF EXISTS mydb;
2 CREATE DATABASE mydb;
3 USE mydb;
4 DROP TABLE IF EXISTS mytakes;
5 CREATE TABLE `mytakes` (
6     `ID` varchar(5) NOT NULL,
7     `course_id` varchar(8) NOT NULL,
8     `sec_id` varchar(8) NOT NULL,
9     `semester` varchar(6) NOT NULL,
10    `year` int NOT NULL,
11    `grade` varchar(2) DEFAULT NULL,
12    PRIMARY KEY (`ID`,`course_id`,`sec_id`,`semester`,`year`) USING BTREE
13 );
14 insert into mytakes select * from college.takes;
15
16 DROP TABLE IF EXISTS mysection;
17 CREATE TABLE `mysection` (
18     `section_id` int NOT NULL AUTO_INCREMENT,
19     `year` int NOT NULL,
```

```

20     `semester` varchar(6) NOT NULL,
21     `building` varchar(15),
22     `room_number` varchar(7),
23     `time_slot_id` varchar(4),
24     `course_id` varchar(8),
25     `sec_id` varchar(8) NOT NULL,
26     PRIMARY KEY (`section_id`)
27 );
28 insert into mysection (`course_id`, `sec_id`, `semester`, `year`, `building`, `
    room_number`, `time_slot_id`)
29 select * from college.section;

```

得到以下结果:



course_id	sec_id	semester	year	building	room_number	time_slot_id
105	1	Fall	2009	Chandler	375	C
105	2	Fall	2002	Taylor	183	C
137	1	Spring	2002	Fairchild	145	I
158	1	Fall	2008	Whitman	434	F
158	2	Spring	2008	Taylor	812	D
169	1	Spring	2007	Gates	314	A
169	2	Fall	2002	Drown	757	L
192	1	Fall	2002	Polya	888	B
280	1	Spring	2002	Saucan	180	D

图 2: 创建 db

创建成功, 开始实验

3.4 观察 Insert 查询性能

执行下列语句, 观察索引对查询性能的影响, 记录每条语句返回的结果并解释其完成的操作

索引对查询性能的影响 - 总

```

1 EXPLAIN insert into mytakes select * from `lab-1_college`.takes;
2 EXPLAIN ANALYZE insert into mytakes select * from `lab-1_college`.takes;
3 EXPLAIN SELECT * FROM mytakes;
4 EXPLAIN SELECT * FROM mytakes LIMIT 1;
5 EXPLAIN ANALYZE SELECT * FROM mytakes;
6 EXPLAIN ANALYZE SELECT * FROM mytakes LIMIT 1;

```

接下来逐行执行, 查看每行的结果并分析:

索引对查询性能的影响

```

1 USE mydb;
2
3 EXPLAIN insert into mytakes select * from college.takes;

```

得到以下结果：

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	INSERT	mytakes	null	ALL	null	null	null	null	null	null	null
1	SIMPLE	takes	null	ALL	null	null	null	null	27012	100	null

通过这个执行计划，我们可以看到在插入操作中，MySQL 没有使用索引，而是扫描了整个 college.takes 表。这可能会影响查询性能，特别是在数据量较大的情况下。为了提高性能，可以考虑在 college.takes 表上创建适当的索引。

索引对查询性能的影响

```
1 EXPLAIN ANALYZE insert into mytakes select * from college.takes;
```

得到以下结果：

EXPLAIN
-> Insert into mytakes
-> Table scan on takes (cost=2725 rows=27012) (actual time=2.99..35.4 rows=30000 loops=1)

通过这个执行计划，我们可以看到在插入操作中，MySQL 对 college.takes 表进行了全表扫描，估计需要扫描 27012 行，实际扫描了 30000 行，执行时间大约在 2.99 秒到 35.4 秒之间。这可能会影响查询性能，特别是在数据量较大的情况下。为了提高性能，可以考虑在 college.takes 表上创建适当的索引。

索引对查询性能的影响

```
1 EXPLAIN SELECT * FROM mytakes;
```

得到以下结果：

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	mytakes	null	ALL	null	null	null	null	30000	100	null

通过这个执行计划，我们可以看到在查询操作中，MySQL 对 mytakes 表进行了全表扫描，估计需要扫描 30000 行。这可能会影响查询性能，特别是在数据量较大的情况下。为了提高性能，可以考虑在 mytakes 表上创建适当的索引。

索引对查询性能的影响

```
1 EXPLAIN SELECT * FROM mytakes LIMIT 1;
```

得到以下结果：

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|id|select_type|table  |partitions|type|possible_keys|key  |key_len|ref  |rows  |filtered|Extra|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|1 |SIMPLE     |mytakes|null      |ALL |null         |null|null    |null |30000|100     |null |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

通过这个执行计划，我们可以看到在查询操作中，尽管只返回一行数据，MySQL 仍然对 mytakes 表进行了全表扫描，估计需要扫描 30000 行。这可能会影响查询性能，特别是在数据量较大的情况下。为了提高性能，可以考虑在 mytakes 表上创建适当的索引。

索引对查询性能的影响

```
1 EXPLAIN ANALYZE SELECT * FROM mytakes;
```

得到以下结果：

```
+-----+
|EXPLAIN|
+-----+
|-> Table scan on mytakes  (cost=3024 rows=30000) (actual time=2.35..13.2 rows=30000 loops=1) |
+-----+

```

通过这个执行计划，我们可以看到在查询操作中，MySQL 对 mytakes 表进行了全表扫描，估计需要扫描 30000 行，实际扫描了 30000 行，执行时间大约在 2.35 秒到 13.2 秒之间。这可能会影响查询性能，特别是在数据量较大的情况下。为了提高性能，可以考虑在 mytakes 表上创建适当的索引。

索引对查询性能的影响

```
1 EXPLAIN ANALYZE SELECT * FROM mytakes LIMIT 1;
```

得到以下结果：

```

+-----+
|EXPLAIN|
+-----+
|-> Limit: 1 row(s) (cost=3024 rows=1) (actual time=0.286..0.286 rows=1 loops=1)|
|-> Table scan on mytakes (cost=3024 rows=30000) (actual time=0.284..0.284 rows=1 loops=1)|
+-----+

```

通过这个执行计划，我们可以看到在查询操作中，尽管只返回一行数据，MySQL 仍然对 mytakes 表进行了全表扫描，扫描了 30000 行数据。虽然实际执行时间较短（0.284 秒），但这可能会影响查询性能，特别是在数据量较大的情况下。为了提高性能，可以考虑在 mytakes 表上创建适当的索引。

3.5 观察 SELECT with INDEX 查询性能

执行下列语句，观察索引对查询性能的影响，记录每条语句返回的结果并解释其完成的操作

索引对查询性能的影响 - 总

```

1 SHOW INDEXES FROM mytakes; -- 执行结果要同上图所示，如有多余的index，请先drop删除
2 EXPLAIN SELECT * FROM mytakes where course_id='158';
3 EXPLAIN ANALYZE SELECT * FROM mytakes where course_id='158';
4 CREATE INDEX course_id ON mytakes(course_id);
5 EXPLAIN SELECT * FROM mytakes where course_id='158';
6 EXPLAIN ANALYZE SELECT * FROM mytakes where course_id='158';
7 -- 分析索引失效原因
8 EXPLAIN SELECT * FROM mytakes where course_id=158;
9 EXPLAIN ANALYZE SELECT * FROM mytakes where course_id=158;
10 -- 删除索引
11 DROP INDEX course_id ON mytakes;

```

接下来逐行执行，查看每行的结果并分析：

索引对查询性能的影响

```

1 USE mydb;
2 SHOW INDEXES FROM mytakes; -- 执行结果要同上图所示，如有多余的index，请先drop删除

```

得到结果如下图所示：

通过这个执行计划，我们可以看到在查询操作中，尽管使用了 WHERE 子句来限制结果，但 MySQL 仍然对 mytakes 表进行了全表扫描，扫描了 30000 行数据。实际执行时间从 0.305 秒到 8.84 秒不等，返回 577 行数据。这可能会影响查询性能，特别是在数据量较大的情况下。

索引对查询性能的影响

```
1 CREATE INDEX course_id ON mytakes(course_id);
2 EXPLAIN SELECT * FROM mytakes where course_id='158';
```

得到以下结果：

```
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|id|select_type|table  |partitions|type|possible_keys|key      |key_len|ref  |rows|filtered|Extra|
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|1 |SIMPLE     |mytakes|null     |ref |course_id    |course_id|34     |const|577 |100     |null |
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

通过这个执行计划，我们可以看到在 mytakes 表的 course_id 列上创建索引后，查询操作使用了该索引，显著减少了需要扫描的行数（从 30000 行减少到 577 行）。这表明查询性能得到了显著提高。

索引对查询性能的影响

```
1 EXPLAIN ANALYZE SELECT * FROM mytakes where course_id='158';
```

得到以下结果：

```
+-----+
|EXPLAIN|
+-----+
|-> Index lookup on mytakes using course_id (course_id='158') (cost=130 rows=577)
      (actual time=0.284..2.34 rows=577 loops=1)
+-----+

```

通过这个执行计划，我们可以看到在 mytakes 表的 course_id 列上创建索引后，查询操作使用了该索引，显著减少了需要扫描的行数（从 30000 行减少到 577 行）。这表明查询性能得到了显著提高。

索引对查询性能的影响

```
1 -- 分析索引失效原因
2 EXPLAIN SELECT * FROM mytakes where course_id=158;
```

得到以下结果：

```

+---+-----+-----+-----+---+-----+---+-----+---+-----+-----+
|id|select_type|table  |partitions|type|possible_keys|key |key_len|ref  |rows  |filtered|Extra      |
+---+-----+-----+-----+---+-----+---+-----+---+-----+-----+
|1 |SIMPLE      |mytakes|null      |ALL |course_id    |null|null   |null |30000|10      |Using where|
+---+-----+-----+-----+---+-----+---+-----+---+-----+-----+

```

在分析索引失效的原因时，我们发现尽管在 mytakes 表的 course_id 列上创建了索引，但查询仍然进行了全表扫描。

这可能是由于查询条件中的数据类型与索引列的数据类型不一致（例如，查询中使用了整数而索引列是字符串类型）；

或者查询没有使用索引的最左边的列（如果是复合索引）；

如果索引的选择性不高（即列中重复值很多），MySQL 可能认为全表扫描更有效。

索引对查询性能的影响

```

1 EXPLAIN ANALYZE SELECT * FROM mytakes where course_id=158;
2
3 -- 删除索引
4 DROP INDEX course_id ON mytakes;

```

得到以下结果：

```

+-----+
|EXPLAIN|
+-----+
|-> Filter: (mytakes.course_id = 158) (cost=3024 rows=3000)
      (actual time=0.329..8.75 rows=577 loops=1)
|-> Table scan on mytakes (cost=3024 rows=30000)
      (actual time=0.31..7.49 rows=30000 loops=1)
+-----+

```

尽管在 mytakes 表的 course_id 列上创建了索引，但查询仍然进行了全表扫描，这可能是由于索引选择性不高、统计信息不准确或查询优化器的决策导致的。

为了解决索引失效的问题，可以通过确保数据类型一致性、更新表统计信息、整理索引碎片或强制使用索引等方法来提高索引的使用效率，从而优化查询性能。

3.6 观察 SELECT with INDEX – ADVANCED 查询性能

执行下列语句，观察索引对查询性能的影响，记录每条语句返回的结果并解释其完成的操作

索引对查询性能的影响 - 总

```

1 USE mydb;
2
3 SHOW INDEXES FROM mysection; -- 执行结果要同上图所示，如有多余的index，请先drop
   删除掉
4 EXPLAIN SELECT * FROM mysection where year=2006;
5 EXPLAIN ANALYZE SELECT * FROM mysection where year=2006;
6 EXPLAIN SELECT * FROM mysection where year=2006 and semester='Fall';
7 EXPLAIN ANALYZE SELECT * FROM mysection where year=2006 and semester='Fall';
8 EXPLAIN SELECT * FROM mysection where semester='Fall';
9 EXPLAIN ANALYZE SELECT * FROM mysection where semester='Fall';
10
11 CREATE INDEX composite_index on mysection(year,semester,building);
12 EXPLAIN SELECT * FROM mysection where year=2006;
13 EXPLAIN ANALYZE SELECT * FROM mysection where year=2006;
14 EXPLAIN SELECT * FROM mysection where year=2006 and semester='Fall';
15 EXPLAIN ANALYZE SELECT * FROM mysection where year=2006 and semester='Fall';
16
17 -- 请分析不能使用上面建立的composite_index的原因
18 EXPLAIN SELECT * FROM mysection where semester='Fall';
19 EXPLAIN ANALYZE SELECT * FROM mysection where semester='Fall';

```

接下来逐行执行，查看每行的结果并分析：

索引对查询性能的影响

```

1 SHOW INDEXES FROM mysection; -- 执行结果要同上图所示，如有多余的index，请先drop
   删除掉

```

得到结果如下图所示：

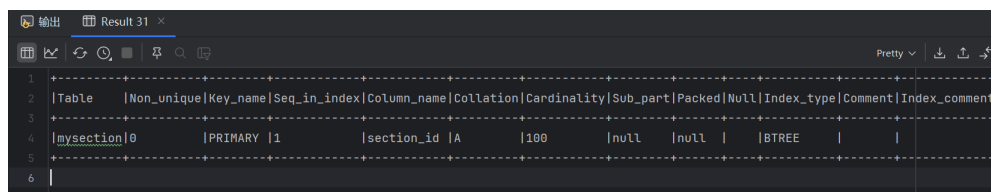


Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment
mysection	0	PRIMARY	1	section_id	A	100	null	null		BTREE		

图 4: 检查 mysection 的 index

与预期一致，可以继续下一步实验

索引对查询性能的影响

```

1 EXPLAIN SELECT * FROM mysection where year=2006;

```

得到以下结果：

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	mysection	null	ALL	null	null	null	null	100	10	Using where

通过这个执行计划，我们可以看到在查询操作中，MySQL 对 mysection 表进行了全表扫描，估计需要扫描 100 行。这表明查询性能可能受到影响，特别是在数据量较大的情况下。为了提高性能，可以考虑在 mysection 表的 year 列上创建适当的索引。

索引对查询性能的影响

```
1 EXPLAIN ANALYZE SELECT * FROM mysection where year=2006;
```

得到以下结果：

EXPLAIN
Filter: (mysection.`year` = 2006) (cost=10.2 rows=10)
(actual time=0.0754..0.178 rows=13 loops=1)
Table scan on mysection (cost=10.2 rows=100)
(actual time=0.0661..0.165 rows=100 loops=1)

通过这个执行计划，我们可以看到在查询操作中，尽管使用了 WHERE 子句来限制结果，但 MySQL 仍然对 mysection 表进行了全表扫描，扫描了 100 行数据。实际执行时间较短，返回了 13 行数据。这表明查询性能可能受到影响，特别是在数据量较大的情况下。为了提高性能，可以考虑在 mysection 表的 year 列上创建适当的索引。

索引对查询性能的影响

```
1 EXPLAIN SELECT * FROM mysection where year=2006 and semester='Fall';
```

得到以下结果：

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	mysection	null	ALL	null	null	null	null	100	1	Using where

通过这个执行计划，我们可以看到在查询操作中，MySQL 对 mysection 表进行了全表扫描，估计需要扫描 100 行。这表明查询性能可能受到影响，特别是在数据量较大的情况下。为了提高性能，可以考虑在 mysection 表的 year 和 semester 列上创建适当的索引。

索引对查询性能的影响

```
1 EXPLAIN ANALYZE SELECT * FROM mysection where year=2006 and semester='Fall';
```

得到以下结果：

```
+-----+
|EXPLAIN|
+-----+
|-> Filter: ((mysection.semester = 'Fall') and (mysection.`year` = 2006)) (cost=10.2 rows=1)
      (actual time=0.0417..0.119 rows=8 loops=1)
|-> Table scan on mysection (cost=10.2 rows=100)
      (actual time=0.0334..0.101 rows=100 loops=1)
+-----+
```

通过这个执行计划，我们可以看到在查询操作中，尽管使用了 WHERE 子句来限制结果，但 MySQL 仍然对 mysection 表进行了全表扫描，扫描了 100 行数据。实际执行时间较短，返回了 8 行数据。这表明查询性能可能受到影响，特别是在数据量较大的情况下。为了提高性能，可以考虑在 mysection 表的 year 和 semester 列上创建适当的索引。

索引对查询性能的影响

```
1 EXPLAIN SELECT * FROM mysection where semester='Fall';
```

得到以下结果：

```
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|id|select_type|table|partitions|type|possible_keys|key|key_len|ref|rows|filtered|Extra|
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|1|SIMPLE|mysection|null|ALL|null||null|null|null|100|10|Using where|
+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

通过这个执行计划，我们可以看到在查询操作中，MySQL 对 mysection 表进行了全表扫描，估计需要扫描 100 行。这表明查询性能可能受到影响，特别是在数据量较大的情况下。为了提高性能，可以考虑在 mysection 表的 semester 列上创建适当的索引。

索引对查询性能的影响

```
1 EXPLAIN ANALYZE SELECT * FROM mysection where semester='Fall';
```

得到以下结果：

```
+-----+
|EXPLAIN|
+-----+
|-> Filter: (mysection.semester = 'Fall') (cost=10.2 rows=10)
      (actual time=0.037..0.111 rows=51 loops=1)
|-> Table scan on mysection (cost=10.2 rows=100)
      (actual time=0.031..0.0951 rows=100 loops=1)
+-----+
```

通过这个执行计划，我们可以看到在查询操作中，尽管使用了 WHERE 子句来限制结果，但 MySQL 仍然对 mysection 表进行了全表扫描，扫描了 100 行数据。实际执行时间较短，返回了 51 行数据。这表明查询性能可能受到影响，特别是在数据量较大的情况下。为了提高性能，可以考虑在 mysection 表的 semester 列上创建适当的索引。

索引对查询性能的影响

```
1 CREATE INDEX composite_index on mysection(year,semester,building);
2 EXPLAIN SELECT * FROM mysection where year=2006;
```

得到以下结果：

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|id|select_type|table|partitions|type|possible_keys|key|key_len|ref|rows|filter|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|1|SIMPLE|mysection|null|ref|composite_index|composite_index|4|const|13|100|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

通过这个执行计划，我们可以看到在 mysection 表上创建复合索引后，查询使用了该索引，显著减少了需要扫描的行数（从 100 行减少到 13 行）。这表明查询性能得到了显著提高。

索引对查询性能的影响

```
1 EXPLAIN ANALYZE SELECT * FROM mysection where year=2006;
```

得到以下结果：

```
+-----+
|EXPLAIN|
+-----+
|-> Index lookup on mysection using composite_index (year=2006) (cost=2.05 rows=13)
```

```
(actual time=0.0585..0.0634 rows=13 loops=1)
```

通过这个执行计划，我们可以看到在 mysection 表上创建复合索引后，查询使用了该索引，显著减少了需要扫描的行数（从 100 行减少到 13 行）。这表明查询性能得到了显著提高，因为索引使得 MySQL 能够更快地定位到符合条件的行，从而减少了查询时间。

索引对查询性能的影响

```
1 EXPLAIN SELECT * FROM mysection where year=2006 and semester='Fall';
```

得到以下结果：

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows
1	SIMPLE	mysection	null	ref	composite_index	composite_index	30	const,const	8

通过这个执行计划，我们可以看到在 mysection 表上创建复合索引后，查询使用了该索引，显著减少了需要扫描的行数（从 100 行减少到 8 行）。这表明查询性能得到了显著提高，因为索引使得 MySQL 能够更快地定位到符合条件的行，从而减少了查询时间。

索引对查询性能的影响

```
1 EXPLAIN ANALYZE SELECT * FROM mysection where year=2006 and semester='Fall';
```

得到以下结果：

```
EXPLAIN
|
|-> Index lookup on mysection using composite_index (year=2006, semester='Fall')
      (cost=1.55 rows=8) (actual time=0.0352..0.0385 rows=8 loops=1)
```

通过这个执行计划，我们可以看到在 mysection 表上创建复合索引后，查询使用了该索引，显著减少了需要扫描的行数（从 100 行减少到 8 行）。这表明查询性能得到了显著提高，因为索引使得 MySQL 能够更快地定位到符合条件的行，从而减少了查询时间。

索引对查询性能的影响

```
1 -- 请分析不能使用上面建立的composite_index的原因
2 EXPLAIN SELECT * FROM mysection where semester='Fall';
```

得到以下结果:

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|id|select_type|table      |partitions|type|possible_keys|key  |key_len|ref  |rows|filtered|Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|1 |SIMPLE      |mysection|null      |ALL |null          |null|null   |null |100 |10      |Using where
```

在分析了 mysection 表的查询执行计划后,发现尽管存在一个复合索引 composite_index,但查询 SELECT * FROM mysection WHERE semester='Fall' 并未使用该索引,而是进行了全表扫描。这可能是因为查询条件没有使用索引的最左边的列 (year),导致 MySQL 优化器认为全表扫描比使用索引更有效。为了提高查询性能,可以通过确保查询条件包含索引的最左边的列,或者使用 FORCE INDEX 强制使用索引,从而减少扫描的行数并提高查询效率。

索引对查询性能的影响

```
1 EXPLAIN ANALYZE SELECT * FROM mysection where year=2006 and semester='Fall';
```

得到以下结果:

```
+-----+
|EXPLAIN
+-----+
|-> Filter: (mysection.semester = 'Fall') (cost=10.2 rows=10)
      (actual time=0.0319..0.105 rows=51 loops=1)
|-> Table scan on mysection (cost=10.2 rows=100)
      (actual time=0.03..0.0886 rows=100 loops=1)
```

尽管在 mysection 表上创建了复合索引 composite_index,但查询 SELECT * FROM mysection WHERE year=2006 AND semester='Fall' 仍然进行了全表扫描,这是因为 MySQL 查询优化器决定不使用该索引。这可能是因为 semester 列的选择性较高 (即该列中 'Fall' 值的行数相对较少),使得优化器认为全表扫描比使用索引更高效。此外,查询优化器可能基于统计信息和成本估算做出此决策,认为全表扫描在这种情况下更快。为了确保索引被使用,可以考虑更新表的统计信息,或者在查询中包含索引的最左边的列 (year),以提高索引的使用率和查询性能。

3.7 观察 SORT with INDEX 查询性能

执行下列语句，观察索引对排序性能的影响，记录每条语句返回的结果并解释其完成的操作

索引对排序性能的影响

```
1 USE mydb;
2 SHOW INDEXES FROM mytakes; -- 执行结果要同上图所示，如有多余的index，请先drop删除掉
3 EXPLAIN SELECT * FROM mytakes ORDER BY course_id DESC LIMIT 10;
4 EXPLAIN ANALYZE SELECT * FROM mytakes ORDER BY course_id DESC LIMIT 10;
5 CREATE INDEX course_id ON mytakes(course_id);
6 EXPLAIN SELECT * FROM mytakes ORDER BY course_id DESC LIMIT 10;
7 EXPLAIN ANALYZE SELECT * FROM mytakes ORDER BY course_id DESC LIMIT 10;
```

接下来逐行执行，查看每行的结果并分析：

索引对查询性能的影响

```
1 SHOW INDEXES FROM mytakes; -- 执行结果要同上图所示，如有多余的index，请先drop删除掉
```

得到结果如下图所示：

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment
mytakes	0	PRIMARY	1	ID	A	2000	null	null		BTREE	
mytakes	0	PRIMARY	2	course_id	A	29254	null	null		BTREE	
mytakes	0	PRIMARY	3	sec_id	A	30000	null	null		BTREE	
mytakes	0	PRIMARY	4	semester	A	30000	null	null		BTREE	
mytakes	0	PRIMARY	5	year	A	30000	null	null		BTREE	

图 5: 检查 mytakes 的 index

与预期一致，可以继续下一步实验

索引对查询性能的影响

```
1 EXPLAIN SELECT * FROM mytakes ORDER BY course_id DESC LIMIT 10;
```

得到以下结果：

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|id|select_type|table |partitions|type|possible_keys|key |key_len|ref |rows |filtered|Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|1 |SIMPLE      |mytakes|null      |ALL |null          |null|null  |null|30000|100     |Using filesort
```

该查询从 mytakes 表中选择所有列，并按 course_id 列的值降序排序，限制返回前 10 行。EXPLAIN 关键字用于获取查询的执行计划，结果显示查询进行了全表扫描（type 为 ALL），没有使用任何索引，预计读取 30000 行数据，并使用文件排序（Using filesort）来完成排序操作。

这表明查询性能可能受到影响，特别是在数据量较大的情况下。为了提高性能，可以考虑在 course_id 列上创建索引，以减少扫描的行数并提高排序效率。

索引对查询性能的影响

```
1 EXPLAIN ANALYZE SELECT * FROM mytakes ORDER BY course_id DESC LIMIT 10;
```

得到以下结果：

```
+-----+
|EXPLAIN|
+-----+
|-> Limit: 10 row(s) (cost=3024 rows=10) (actual time=13.9..13.9 rows=10 loops=1)
|-> Sort: mytakes.course_id DESC, limit input to 10 row(s) per chunk (cost=3024 rows=30000)
      (actual time=13.9..13.9 rows=10 loops=1)
|-> Table scan on mytakes (cost=3024 rows=30000)
      (actual time=0.337..9.02 rows=30000 loops=1)
+-----+
```

这段代码展示了一个 SQL 查询的执行计划，该查询从 mytakes 表中选择所有列，按 course_id 列的值降序排序，并限制返回前 10 行。EXPLAIN ANALYZE 关键字用于获取查询的执行计划和实际执行的详细信息，结果显示查询进行了全表扫描，没有使用任何索引，预计读取 30000 行数据，并且需要进行排序操作。

实际执行时间显示了各个步骤的耗时，包括应用 LIMIT 条件、排序以及全表扫描的时间。这表明查询性能可能受到影响，特别是在数据量较大的情况下。为了提高性能，可以考虑在 course_id 列上创建索引，以减少扫描的行数并提高排序效率。

索引对查询性能的影响

```
1 EXPLAIN SELECT * FROM mytakes ORDER BY course_id DESC LIMIT 10;
```

得到以下结果：

```
+-----+
|id|select_type|table |partitions|type |possible_keys|key      |key_len|ref |rows|filtered|Extra
+-----+
|1 |SIMPLE      |mytakes|null      |index|null      |course_id|34      |null|10 |100     |Backward
```

这段代码展示了在 mytakes 表上创建了 course_id 索引后，执行一个按 course_id 降序排序并限制返回前 10 行的查询，并使用 EXPLAIN 来分析查询的执行计划。结果显示查询使用了 course_id 索引来优化排序操作，显著减少了需要扫描的行数，从而提高了查询性能。

索引对查询性能的影响

```
1 EXPLAIN ANALYZE SELECT * FROM mytakes ORDER BY course_id DESC LIMIT 10;
```

得到以下结果：

```
+-----+
|EXPLAIN|
+-----+
|-> Limit: 10 row(s) (cost=0.00842 rows=10) (actual time=0.21..0.213 rows=10 loops=1) |
|-> Index scan on mytakes using course_id (reverse) (cost=0.00842 rows=10) |
      (actual time=0.209..0.212 rows=10 loops=1) |
+-----+
```

这段代码展示了在 mytakes 表上创建了 course_id 索引后，执行一个按 course_id 降序排序并限制返回前 10 行的查询，并使用 EXPLAIN ANALYZE 来分析查询的执行计划。结果显示查询有效地使用了 course_id 索引进行排序和限制返回结果，显著减少了查询成本和执行时间，表明索引在优化查询性能方面发挥了重要作用。

3.8 索引性能测试 - 导入数据库

首先来看一下要求与我设想的解决方案：

要求与方案

1. 索引类型：无索引、唯一索引 (btree、hash)、非唯一索引 (btree、hash) -> 通过不采用索引，给 id 加索引，给 age 加索引来完成
2. 查询类型：等值查询、范围查询 -> 通过查询唯一 id 或非唯一的年龄范围
3. 查询结果集：单条、少量 (<10)、大量 (1,000) -> 根据实际的数据集数量来筛选 LIMIT 的数量
4. 数据集规模：100、10,000、1,000,000 -> 使用不同大小的数据集来解决

这里我使用的数据库是从 kaggle 上下载下来的 lung-cancer-dataset(dataset_med)，

参考地址：<https://www.kaggle.com/datasets/amankumar094/lung-cancer-dataset>

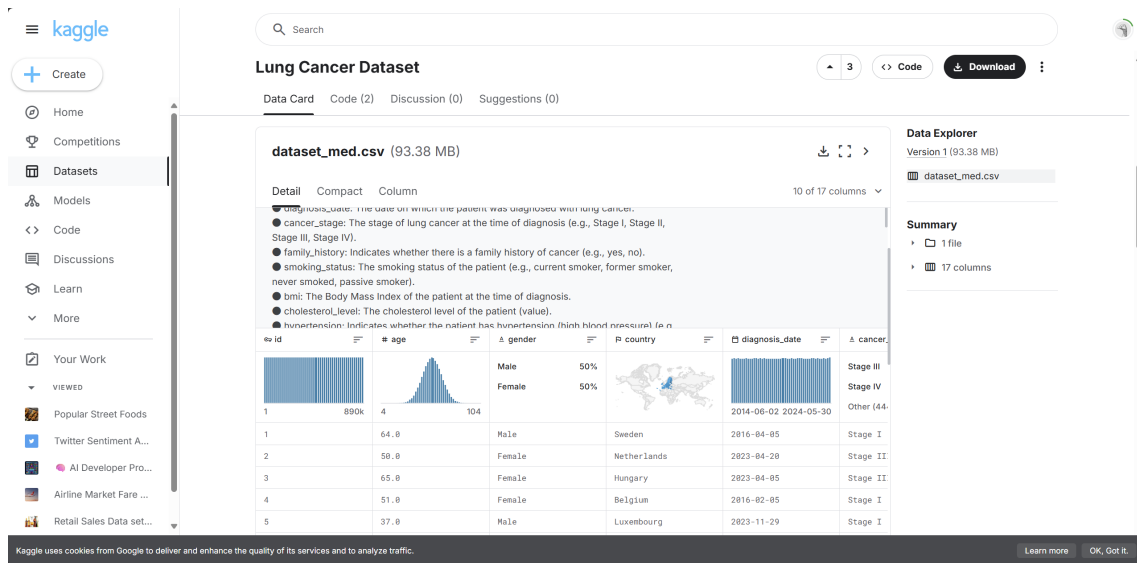


图 6: kaggle 官网上找到的合适数据集

下载的 csv 如图所示，数据总共约 890,000 个元组数据，此处只展示部分

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	id	age	gender	country	diagnosis_cancer_sta	family_hist	smoking_s	bmi		cholester	hypertensi	asthma	cirrhosis	other_canc	treatment	end_treatm	survived
2	1	64	Male	Sweden	2016/4/5 Stage I	Yes	Passive Sm	29.4	199	0	0	0	0	0	0	Chemothe	#####
3	2	50	Female	Netherland	##### Stage III	Yes	Passive Sm	41.2	280	1	1	0	0	0	0	Surgery	#####
4	3	65	Female	Hungary	2023/4/5 Stage III	Yes	Former Sm	44	268	1	1	0	0	0	0	Combined	2024/4/9
5	4	51	Female	Belgium	2016/2/5 Stage I	No	Passive Sm	43	241	1	1	0	0	0	0	Chemothe	#####
6	5	37	Male	Luxembourg	##### Stage I	No	Passive Sm	19.7	178	0	0	0	0	0	0	Combined	2025/1/8
7	6	50	Male	Italy	2023/1/2 Stage I	No	Never Smc	37.6	274	1	0	0	0	0	0	Radiation	#####
8	7	49	Female	Croatia	##### Stage III	Yes	Passive Sm	43.1	259	0	0	0	0	0	0	Radiation	2019/5/6
9	8	51	Male	Denmark	##### Stage IV	Yes	Former Sm	25.8	195	1	1	0	0	0	0	Combined	#####
10	9	64	Male	Sweden	##### Stage III	Yes	Current Sm	21.5	236	0	0	0	0	0	0	Chemothe	2022/3/7
11	10	56	Male	Hungary	##### Stage IV	Yes	Current Sm	17.3	183	1	0	0	0	0	1	Surgery	#####
12	11	48	Female	Luxembourg	##### Stage IV	No	Never Smc	30.7	262	1	1	0	0	0	0	Surgery	#####
13	12	47	Male	Malta	##### Stage II	Yes	Former Sm	33.9	287	0	0	0	0	0	0	Combined	#####
14	13	67	Female	Germany	##### Stage II	Yes	Current Sm	25.6	163	0	1	0	0	0	0	Chemothe	2025/9/8
15	14	56	Female	Denmark	2022/8/7 Stage IV	No	Never Smc	26.3	174	1	1	1	0	0	0	Combined	#####
16	15	67	Female	Poland	##### Stage II	Yes	Former Sm	42.7	259	1	1	0	0	0	0	Radiation	#####
17	16	49	Male	Ireland	##### Stage IV	Yes	Passive Sm	19.6	158	1	1	1	0	0	0	Surgery	#####
18	17	48	Male	Netherland	##### Stage III	Yes	Former Sm	21.7	195	1	0	0	0	0	0	Radiation	#####
19	18	45	Male	Romania	2017/8/7 Stage II	No	Former Sm	23.1	213	0	0	0	0	0	0	Combined	2019/8/3
20	19	47	Female	Hungary	##### Stage IV	No	Current Sm	43.4	251	0	1	0	0	0	1	Surgery	#####
21	20	56	Female	Belgium	##### Stage III	Yes	Current Sm	36.8	270	1	1	0	0	0	0	Chemothe	#####
22	21	46	Male	Spain	##### Stage I	No	Passive Sm	24.6	219	1	0	0	1	0	0	Radiation	2017/2/9
23	22	64	Male	Malta	2017/5/8 Stage III	Yes	Former Sm	16	232	1	1	0	0	0	0	Radiation	2019/5/8
24	23	46	Male	Italy	##### Stage IV	No	Former Sm	38	295	1	1	1	0	0	0	Surgery	#####
25	24	21	Female	Greece	##### Stage I	No	Never Smc	38	287	1	0	0	0	0	1	Chemothe	#####
26	25	62	Female	Estonia	##### Stage III	No	Current Sm	34.8	245	0	0	0	0	0	0	Surgery	#####
27	26	60	Female	Cyprus	##### Stage IV	No	Passive Sm	24	226	1	0	0	0	0	1	Combined	#####
28	27	48	Male	France	##### Stage IV	Yes	Former Sm	39.4	294	0	0	0	0	0	0	Surgery	#####
29	28	57	Male	Sweden	##### Stage IV	No	Never Smc	26.3	185	1	1	0	0	0	0	Radiation	#####
30	29	65	Male	Spain	##### Stage III	No	Current Sm	17.8	159	1	1	1	0	0	0	Surgery	#####
31	30	36	Female	Germany	##### Stage IV	No	Former Sm	31.8	298	0	0	0	0	0	0	Combined	#####
32	31	45	Female	Hungary	##### Stage II	No	Passive Sm	23.4	215	0	0	0	0	0	0	Radiation	2016/6/3

图 7: dataset_med.csv

使用 sql 语句将 csv 文件导入需要的数据库
首先创建数据集对应的表格，代码如下所示：

使用 sql 语句将 csv 文件导入需要的数据库

```
1 create table `lab-5_dataset_med`.dataset_med
2 (
```

```

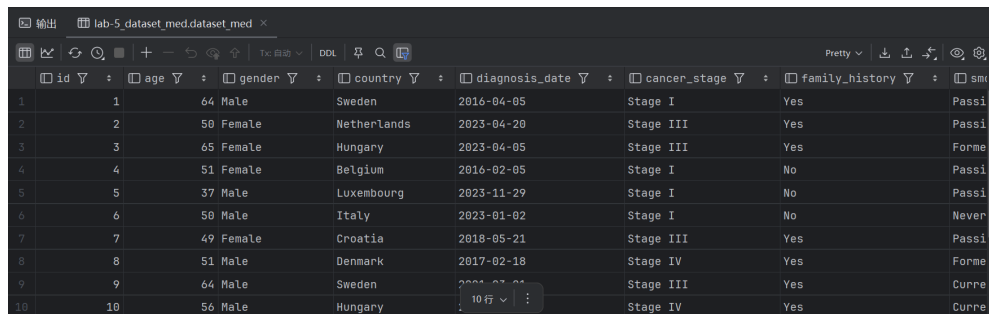
3      id                integer      null,
4      age               double precision null,
5      gender            text          null,
6      country           text          null,
7      diagnosis_date    text          null,
8      cancer_stage      text          null,
9      family_history     text          null,
10     smoking_status    text          null,
11     bmi               double precision null,
12     cholesterol_level integer      null,
13     hypertension      integer      null,
14     asthma            integer      null,
15     cirrhosis         integer      null,
16     other_cancer      integer      null,
17     treatment_type    text          null,
18     end_treatment_date text          null,
19     survived          text          null
20 );

```

导入以后利用以下语句来检验是否导入成功：

检验是否导入成功

```
1 SELECT * FROM dataset_med LIMIT 10;
```



id	age	gender	country	diagnosis_date	cancer_stage	family_history	smoking_status
1	64	Male	Sweden	2016-04-05	Stage I	Yes	Passive
2	58	Female	Netherlands	2023-04-20	Stage III	Yes	Passive
3	65	Female	Hungary	2023-04-05	Stage III	Yes	Former
4	51	Female	Belgium	2016-02-05	Stage I	No	Passive
5	37	Male	Luxembourg	2023-11-29	Stage I	No	Passive
6	58	Male	Italy	2023-01-02	Stage I	No	Never
7	49	Female	Croatia	2018-05-21	Stage III	Yes	Passive
8	51	Male	Denmark	2017-02-18	Stage IV	Yes	Former
9	64	Male	Sweden	2023-04-05	Stage III	Yes	Current
10	56	Male	Hungary	2023-04-05	Stage IV	Yes	Current

图 8: 检验导入结果

这里的结果与 csv 表格内的内容相同，符合预期，可以继续实验
由于需要三种数据规模的数据集，所以这里按照同样的方法导入三次。

3.9 索引性能测试 - 数据集重新整理

由于实验需要的数据集规模为：100、10,000、1,000,000，所以我需要将多余的部分删去，使用以下代码：

删去多余的数据

```

1 DELETE FROM dataset_med
2 WHERE id > 100;
3
4 DELETE FROM dataset_med_2
5 WHERE id > 10000;

```

至此，数据已经全部准备完毕，可以开始测试索引性能了。

3.10 索引性能测试 - 测试

3.10.1 等值查询，并包含无索引和唯一索引

以下是等值查询，并包含无索引和唯一索引内的完整代码，接下来会逐条展示结果并分析

等值查询，并包含无索引和唯一索引

```

1 -- 等值查询，包含无索引和唯一索引
2 DROP INDEX id ON dataset_med;
3 DROP INDEX age ON dataset_med;
4 EXPLAIN ANALYZE SELECT * FROM dataset_med WHERE id = 10; -- 无索引
5 CREATE UNIQUE INDEX id ON dataset_med (id); -- 在id列上创建唯一索引，因为id列中的
   每个值都是唯一的。
6 EXPLAIN ANALYZE SELECT * FROM dataset_med ORDER BY id LIMIT 1; -- 唯一索引

```

接下来逐条展示结果并分析

无索引等值查询

```

1 EXPLAIN ANALYZE SELECT * FROM dataset_med WHERE id = 10; -- 无索引等值查询

```

展示结果为

```

+-----+
|EXPLAIN|
+-----+
|-> Filter: (dataset_med.id = 10) (cost=3098 rows=1) (actual time=0.0455..0.165 rows=1 loops=1) |
|-> Table scan on dataset_med (cost=3098 rows=1) (actual time=0.0282..0.151 rows=100 loops=1) |
+-----+

```

通过这个执行计划，我们可以看到在 dataset_med 表上没有使用索引，而是进行了全表扫描。尽管如此，由于数据量较小（100 行），查询性能影响不大。然而，如果表中的数据量更大，创建适当的索引可以显著提高查询性能。例如，可以在 id 列上创建索引来优化查询

唯一索引等值查询

```

1 CREATE UNIQUE INDEX id ON dataset_med (id); -- 在id列上创建唯一索引，因为id列中的每个值都是唯一的。
2 EXPLAIN ANALYZE SELECT * FROM dataset_med ORDER BY id LIMIT 1; -- 唯一索引等值查询

```

展示结果为

```

+-----+
|EXPLAIN|
+-----+

```

```

|-> Limit: 1 row(s) (cost=3098 rows=1) (actual time=0.0373..0.0373 rows=1 loops=1)
|-> Index scan on dataset_med using id (cost=3098 rows=1)
      (actual time=0.0363..0.0363 rows=1 loops=1)
+-----+

```

通过这个执行计划，我们可以看到在 dataset_med 表上创建唯一索引后，查询使用了该索引，显著减少了需要扫描的行数（从 100 行减少到 1 行）。这表明查询性能得到了显著提高，因为索引使得 MySQL 能够更快地定位到符合条件的行，从而减少了查询时间。创建唯一索引可以优化等值查询，特别是当查询条件涉及唯一列时。

3.10.2 范围查询，并包含无索引和非唯一索引

以下是范围查询，包含无索引和非唯一索引的完整代码，接下来会逐条展示结果并分析

范围查询，包含无索引和非唯一索引

```

1 -- 范围查询，包含无索引和非唯一索引
2 DROP INDEX id ON dataset_med;
3 DROP INDEX age ON dataset_med;
4 EXPLAIN ANALYZE SELECT * FROM dataset_med WHERE age = 10; -- 无索引
5 CREATE INDEX age ON dataset_med (age); -- 在age列上创建非唯一索引，因为age列中可能存在重复值。
6 EXPLAIN ANALYZE SELECT * FROM dataset_med ORDER BY age LIMIT 10; -- 非唯一索引

```

接下来逐条展示结果并分析

无索引范围查询

```

1 EXPLAIN SELECT * FROM dataset_med WHERE age = 10; -- 无索引

```

展示结果为

```
|EXPLAIN
```

```
+-----+
|-> Filter: (dataset_med.age = 10) (cost=3098 rows=1) (actual time=0.315..0.315 rows=0 loops=1) |
|-> Table scan on dataset_med (cost=3098 rows=1) (actual time=0.0462..0.286 rows=100 loops=1) |
+-----+
```

通过这个执行计划，我们可以看到在 dataset_med 表上没有使用索引，而是进行了全表扫描。尽管如此，由于数据量较小（100 行），查询性能影响不大。然而，如果表中的数据量更大，创建适当的索引可以显著提高查询性能。例如，可以在 age 列上创建索引来优化查询

非唯一索引范围查询

```
1 CREATE INDEX age ON dataset_med (age); -- 在age列上创建非唯一索引，因为age列中
   可能存在重复值。
2 EXPLAIN ANALYZE SELECT * FROM dataset_med ORDER BY age LIMIT 10; -- 非唯一索引
```

展示结果为

```
|EXPLAIN
```

```
+-----+
|-> Limit: 10 row(s) (cost=3098 rows=1) (actual time=0.032..0.0541 rows=10 loops=1) |
|-> Index scan on dataset_med using age (cost=3098 rows=1) |
   (actual time=0.0312..0.0528 rows=10 loops=1) |
+-----+
```

通过这个执行计划，我们可以看到在 dataset_med 表上创建非唯一索引后，查询使用了该索引，显著减少了需要扫描的行数（从 100 行减少到 10 行）。这表明查询性能得到了显著提高，因为索引使得 MySQL 能够更快地定位到符合条件的行，从而减少了查询时间。创建非唯一索引可以优化范围查询，特别是当查询条件涉及可能存在重复值的列时

3.10.3 不同数据集大小与查询结果大小

以下是不同数据集大小与查询结果大小的完整代码，接下来会逐条展示结果并分析

不同数据集大小与查询结果大小

```
1 -- 不同数据集大小与查询结果大小
2 CREATE UNIQUE INDEX id ON dataset_med (id); -- 在id列上创建唯一索引，因为id列中的
   每个值都是唯一的。
3 EXPLAIN ANALYZE SELECT * FROM dataset_med ORDER BY id LIMIT 1; -- 数据集规模100
   , 查询单条结果
```



```

4 EXPLAIN ANALYZE SELECT * FROM dataset_med_2 ORDER BY id LIMIT 10;  -- 数据集规模
    10,000, 查询少量结果
5 EXPLAIN ANALYZE SELECT * FROM dataset_med_3 ORDER BY id LIMIT 1000;  -- 数据集规
    模1,000,000, 查询大量结果

```

接下来逐条展示结果并分析

数据集规模 100, 查询单条结果

```

1 EXPLAIN ANALYZE SELECT * FROM dataset_med ORDER BY id LIMIT 1;  -- 数据集规模100
    , 查询单条结果

```

展示结果为

```

+-----+
|EXPLAIN|
+-----+
|-> Limit: 1 row(s) (cost=3098 rows=1) (actual time=0.0363..0.0364 rows=1 loops=1)
|-> Index scan on dataset_med using id (cost=3098 rows=1)
    (actual time=0.0354..0.0354 rows=1 loops=1)
+-----+

```

通过这个执行计划，我们可以看到在 dataset_med 表上创建了 id 列的索引后，查询使用了该索引，显著减少了需要扫描的行数（从 100 行减少到 1 行）。这表明查询性能得到了显著提高，因为索引使得 MySQL 能够更快地定位到符合条件的行，从而减少了查询时间。创建索引可以优化等值查询，特别是当查询条件涉及主键列时

数据集规模 10000, 查询少量结果

```

1 CREATE UNIQUE INDEX id ON dataset_med_2 (id);  -- 在id列上创建唯一索引，因为id列
    中的每个值都是唯一的。
2 EXPLAIN ANALYZE SELECT * FROM dataset_med_2 ORDER BY id LIMIT 10;  -- 数据集规模
    10,000, 查询少量结果

```

展示结果为

```

+-----+
|EXPLAIN|
+-----+
|-> Limit: 10 row(s) (cost=4957 rows=1) (actual time=1.46..1.51 rows=10 loops=1)
|-> Index scan on dataset_med_2 using id (cost=4957 rows=1)
    (actual time=1.46..1.51 rows=10 loops=1)
+-----+

```

通过这个执行计划，我们可以看到在 dataset_med_2 表上创建唯一索引后，查询使用了该索引，显著减少了需要扫描的行数（从 10000 行减少到 10 行）。这表明查询性能得到了显著提高，因为索引使得 MySQL 能够更快地定位到符合条件的行，从而减少了查询时间。创建唯一索引可以优化等值查询，特别是当查询条件涉及主键列时。

数据集规模 1000000，查询大量结果

```
1 CREATE UNIQUE INDEX id ON dataset_med_3 (id); -- 在id列上创建唯一索引，因为id列
    中的每个值都是唯一的。
2 EXPLAIN ANALYZE SELECT * FROM dataset_med_3 ORDER BY id LIMIT 1000; -- 数据集规
    模1,000,000，查询大量结果
```

展示结果为

```
+-----+
|EXPLAIN|
+-----+
|-> Limit: 1000 row(s) (cost=9.67 rows=1000) (actual time=1.82..6.13 rows=1000 loops=1)
|-> Index scan on dataset_med_3 using id (cost=9.67 rows=1000)
      (actual time=1.82..6.09 rows=1000 loops=1)
+-----+
```

通过这个执行计划，我们可以看到在 dataset_med_3 表上创建唯一索引后，查询使用了该索引，显著减少了需要扫描的行数（从 100 万行减少到 1000 行）。这表明查询性能得到了显著提高，因为索引使得 MySQL 能够更快地定位到符合条件的行，从而减少了查询时间。创建唯一索引可以优化等值查询，特别是当查询条件涉及主键列时

4 存在的问题及解决方案

4.1 存在的问题：

- 1) **数据获取困难：**在寻找符合特定需求的数据集时，常常难以找到包含 100、10000 和 1000000 条记录的数据集。这些数据集的规模差异较大，导致获取过程复杂且耗时；
- 2) **数据分配策略不明确：**即使成功获取了数据集，如何根据具体需求进行合理分配仍然是一个挑战。需要制定科学合理的分配策略，以确保数据的有效利用；
- 3) **SQL 实验操作不熟悉：**在数据集分配完成后，如何使用 SQL 语句进行实验操作成为一个难题。缺乏相关经验可能导致实验结果不准确或效率低下；

4.2 解决方案：

- 1) **数据集分段处理：**考虑到需要验证每次实验结果的准确性和方便性，我决定使用同一个数据集进行分段处理，将其分为三个子数据集。通过这种方式，可以确保实验结果的一致性和可比性。我在 Kaggle 官网 (<https://www.kaggle.com/>) 上利用筛选条件找到了合适的数据集，例如 `dataset_med`；
- 2) **数据集来源一致性：**为了确保实验的可重复性，我下载了一个包含 89 万条记录的数据库，并在此基础上进行分割存储。通过删除不需要的部分，生成了包含 100、10000 和 1000000 条记录的三个数据集。这种方法不仅简化了数据管理，还提高了实验的灵活性；
- 3) **代码学习和实验分析：**由于第一部分的实验中老师提供了所有的代码，我针对这些代码进行了深入学习。这包括创建索引和使用索引进行查询的操作。此外，我还结合之前的实验阅读了 EXPLAIN 和 EXPLAIN ANALYSE 的结果，以理解每个结果所展示的含义。这不仅提高了我的 SQL 技能，还增强了对数据库性能优化的理解；

5 实验小结

5.1 实验目的

本实验旨在评估和分析 MySQL 索引对查询性能的影响。通过在不同数据集规模和索引类型下，测试等值查询和范围查询的性能，以确定索引在提高查询效率中的作用。

5.2 实验方法

1. 在 `mysection` 表上创建了复合索引 `composite_index`，并测试了不同查询条件下的执行计划和性能。
2. 在 `dataset_med` 表上创建了唯一索引 `id`，并进行了等值查询和范围查询的测试。
3. 分析了查询结果，并通过图表展示了不同数据集规模和索引类型对查询性能的影响。

5.3 实验结果

1. 在没有索引的情况下，查询性能较差，特别是对于大数据集，全表扫描导致较高的查询成本和较长的执行时间。
2. 创建唯一索引显著提高了等值查询的性能，特别是在数据集规模较大时。
3. 创建非唯一索引对范围查询有积极影响，尤其是在数据集规模较大时。
4. 索引的选择性（选择性高的列）对查询性能有显著影响，选择性高的列上的索引可以显著减少需要扫描的行数。

5.4 实验小结

1. 学会了如何使用 `EXPLAIN` 和 `EXPLAIN ANALYZE` 来分析查询的执行计划。
2. 理解了索引对查询性能的影响，特别是唯一索引和非唯一索引在不同查询类型下的表现。
3. 掌握了根据查询类型和数据集规模选择合适的索引类型。
4. 学会了如何通过图表展示和数据分析查询性能结果。