

《查询优化》

实验报告



学院： 计算机学院（国家示范性软件学院）

班级： 2019211308 2019211308 2019211308

姓名： 顾天阳 曾世茂 庞仕泽

学号： 2019211539 2019211532 2019211509

目录

2.1 执行计划的查看与分析	3
2.2 观察视图查询，with 临时视图查询的执行计划..	5
2.3 优化 sql 语句.....	7
2.3.1 复合索引做前缀.....	7
2.3.2 多表连接操作，在连接属性上建立索引	9
2.3.3 索引对小表查询的搜索.....	12
2.3.4 查询条件中函数对索引的影响.....	14
2.3.5 多表嵌入式 SQL 查询	15
2.3.6 where 查询条件中复合查询条件 OR 对索引的影响	16
2.3.7 聚集索引中的索引设计.....	18
2.3.8 select 子句中是否有 distinct 的区别.....	19
2.3.9 union 和 union all 的区别.....	20
2.3.10 from 中存在多余的关系表，即查询非最简化	20
遇到的问题及解决	21
实验总结	21

2.1 执行计划的查看与分析

1. 编写 sql 语句

查询 tbATUDData 中 PCI 小于 100 且在 tbATUC2I 中干扰强度小于 3 的主小区的切换目标小区。

```
select    "CellID","N_SECTOR_ID" from "tbATUDData","tbATUC2I","tbAdjCell"
WHERE "PCI" <100 and "CellID" = "SECTOR_ID" and "RATIO_ALL" <3 and  "CellID"
="S_SECTOR_ID";
```

查询 tbHandOver 中小区类型为“优化区”且设备厂家为华为的切换源小区和切换成功率。

```
SELECT "SCell","HOSUCCRATE" FROM "tbHandOver","tbOptCell","tbCell_2"
WHERE  "tbHandOver"."SCell" = "tbOptCell"."SECTOR_ID" and "tbHandOver"."SCell" =
"tbCell_2"."SECTOR_ID"
and "CELL_TYPE" = '优化区' and "VENDOR" = '华为'
```

2. 查看查询计划

查询 1:

1 EXPLAIN
2 select "CellID","N_SECTOR_ID" from "tbATUDData","tbATUC2I","tbAdjCell" WHERE "CellID" = "SECTOR_ID" and "CellID" = "S_SECTOR_ID" and "PCI" <100 and "RATIO_ALL" <3 ;

SQL执行记录 消息 结果 以下EXPLAIN select "CellID","N_SECTOR_ID" from "tbATUDData","tbATUC2I","tbAdjCell" 的执行结果集

以下是EXPLAIN select "CellID","N_SECTOR_ID" from "tbATUDData","tbATUC2I","tbAdjCell" 的执行结果集: 该表不可编辑

	QUERY PLAN
1	Hash Join (cost=5032.99..1200231.42 rows=108843563 width=18)
2	Hash Cond: (("tbATUDData"."CellID")::text = ("tbATUC2I"."SECTOR_ID")::text)
3	-> Seq Scan on "tbATUDData" (cost=0.00..13674.12 rows=74826 width=9)
4	Filter: ("PCI" < 100)
5	-> Hash (cost=3510.12..3510.12 rows=121830 width=27)
6	-> Hash Join (cost=36.83..3510.12 rows=121830 width=27)
7	Hash Cond: (("tbAdjCell"."S_SECTOR_ID")::text = ("tbATUC2I"."SECTOR_ID")::text)
8	-> Seq Scan on "tbAdjCell" (cost=0.00..1213.13 rows=69713 width=18)
9	-> Hash (cost=26.34..26.34 rows=839 width=9)
10	-> Seq Scan on "tbATUC2I" (cost=0.00..26.34 rows=839 width=9)
11	Filter: ("RATIO ALL" < 3:double precision)

解释:

第一层: seq scan on tbATUDData , 扫描检查的条件为 PCI<100 , 将符合条件的数据从 buffer 或磁盘中读出给上层运算, 预计执行时间为 0-13674ms

第二层: seq scan on tbATUC2I , 扫描检查的条件为 RATIO_ALL<3, 将符合条件的数据从 buffer 或磁盘中读出给上层运算, 预计执行时间为 0-26ms, 预计输出行数为 839 行

第三层: hash, 将第二层的结果计算 hash 值, 为后续 hash join 做准备

第四层: hash join, 将 tbAdjCell 与 tbATUC2I 连接在一起, 连接条件为

"tbAdjCell"."S_SECTOR_ID" = "tbATUC21"."SECTOR_ID" ， ， 预计执行时间为 36-3510ms
第五层: hash ， 将第四层的结果计算 hash 值，为后续 hash join 做准备
第六层: hash join ， 将第五层与第一层的结果连接，连接条件为("tbATUData"."CellID")
= ("tbATUC21"."SECTOR_ID")，预计时间为 5032.99-1200231ms 预计输出 108843563 行

查询 2:

执行SQL(F8) 格式化(F9) 执行计划(F6) 我的SQL

SQL提示

```
1 EXPLAIN
2 SELECT "SCell", "HOSUCCRATE" FROM "tbHandOver", "tbOptcell", "tbCell_2"
3 WHERE "tbHandOver"."SCell" = "tbOptcell"."SECTOR_ID" and "tbHandOver"."SCell" = "tbCell_2"."SECTOR_ID"
4 and "CELL_TYPE" = '优化区' and "VENDOR" = '华为';
```

SQL执行记录 消息 结果集1 X

以下是EXPLAIN SELECT "SCell", "HOSUCCRATE" FROM "tbHandOver", "tbOptcell", "tbCell_2" 的执行... 该表不可编辑。

	QUERY PLAN
1	Hash Join (cost=329.80..573.02 rows=7334 width=13)
2	Hash Cond: (("tbHandOver"."SCell")::text = (tbOptcell."SECTOR_ID")::text)
3	-> Seq Scan on "tbHandOver" (cost=0.00..142.36 rows=7336 width=13)
4	-> Hash (cost=325.76..325.76 rows=323 width=18)
5	-> Hash Join (cost=16.09..325.76 rows=323 width=18)
6	Hash Cond: ((tbCell_2."SECTOR_ID")::text = (tbOptcell."SECTOR_ID")::text)
7	-> Seq Scan on tbCell_2 (cost=0.00..285.00 rows=5504 width=9)
8	Filter: (("VENDOR")::text = '华为')::text)
9	-> Hash (cost=12.05..12.05 rows=323 width=9)
10	-> Seq Scan on tbOptcell (cost=0.00..12.05 rows=323 width=9)
11	Filter: (("CELL_TYPE")::text = '优化区')::text)

解释

第一层: seq scan on tbOptcell ， 扫描条件四 cell type = “优化区”。将符合条件的数据从磁盘或 buffer 中读出

第二层: hash，将第一层结果求 hash ， 为后续连接做准备

第三层 seq scan on tbCell ， 扫描条件为 vendor 为 “华为”，将符合条件的数据从磁盘或 buffer 中读出

第四层: hash join ，将第二三层的结果连接在一起，连接条件为 (tbCell_2."SECTOR_ID")=(tbOptcell."SECTOR_ID")

第五层: hash ， 将第四层的结果求 hash，为后续连接做准备

第六层 : seq scan on tbhandover ， 将所有数据从磁盘或 buffer 中读出

第七层: hash join，将第五层与第六层的结果进行连接输出，连接条件为 ("tbHandOver"."SCell") = (tbOptcell."SECTOR_ID")，总共花费时间预计为 329-573ms，输出 7334 行

2.2 观察视图查询，with 临时视图查询的执行计划

1. 在 tbcel_2l 上建立一个视图

```
1 create view tbcell_view as
2 select * from tbcell_2
```

SQL执行记录

消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】

```
create view tbcell_view as
select * from tbcell_2
执行成功，耗时：[15ms.]
```

2. 查询并查看执行计划

- (1) 使用视图进行查询

```
EXPLAIN ANALYZE
SELECT "SECTOR_ID"
from tbcell_view
```

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】

```
EXPLAIN ANALYZE
SELECT "SECTOR_ID"
from tbcell_view
执行成功，当前返回：[2]行，耗时：[13ms.]
```

执行结果：

以下是EXPLAIN ANALYZE SELECT "SECTOR_ID" from tbcell_view的执行结果集

该表不可编辑

复制行 复制列 列设置

	QUERY PLAN
1	Seq Scan on tbcell_2 (cost=0.00..272.04 rows=5504 width=9) (actual time=0.022..1.290 rows=5504 loops=1)
2	Total runtime: 1.676 ms

(2) 使用 with 临时视图进行查询

```
-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL: (1)】
EXPLAIN ANALYZE

with tempview as(
  select * from tbcell_2
)

SELECT "SECTOR_ID" from tempview;
执行成功，当前返回: [4]行，耗时: [14ms.]
```

以下是EXPLAIN ANALYZE with tempview as(select * from tbcell_2) SELECT "SECTOR_ID"...的执行结果集

该表不可编辑

复制行 复制列 列设置

	QUERY PLAN
1	CTE Scan on tempview (cost=272.04..382.12 rows=5504 width=82) (actual time=0.034..3.468 rows=5504 loops=1)
2	CTE tempview
3	-> Seq Scan on tbcell_2 (cost=0.00..272.04 rows=5504 width=126) (actual time=0.019..1.055 rows=5504 loops=1)
4	Total runtime: 4.214 ms

(3) 不使用视图进行查询

1 EXPLAIN ANALYZE
2 SELECT "SECTOR_ID" from tbcell_2;
3

SQL执行记录 消息 结果集1 x

以下是EXPLAIN ANALYZE SELECT "SECTOR_ID" from tbcell_2的执行结果集

该表不可编辑

复制行 复制列 列设置

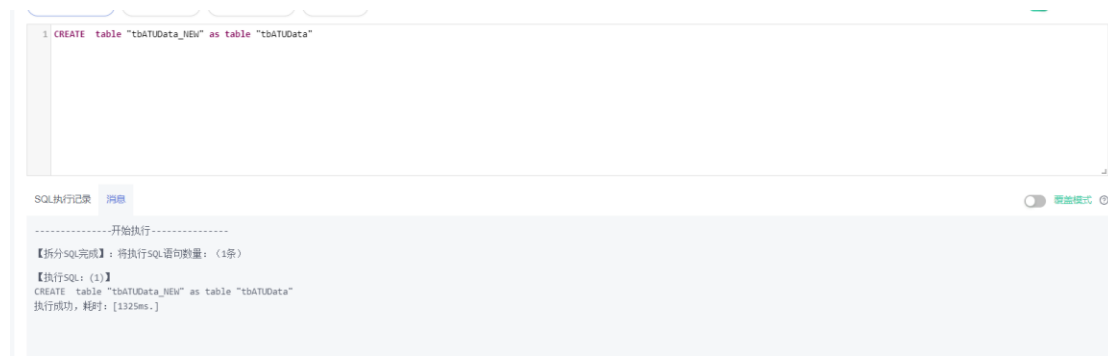
	QUERY PLAN
1	Seq Scan on tbcell_2 (cost=0.00..272.04 rows=5504 width=9) (actual time=0.021..1.292 rows=5504 loops=1)
2	Total runtime: 1.688 ms

分析：通过上述实验可知，建立与不建立视图在查询时间上影响不大，但使用 with 进行查询时，会将查询分成两层，第一层首先完成了 with 内的查询，第二层再从第一层结果中进行查询。所以时间稍微较长一些。建立视图后，可以直接从视图中进行查询，所以速度略快一些。所以当经常需要进行相关查询时可以建立视图，而少用到的视图可以使用 with 建立临时视图

2.3 优化 sql 语句

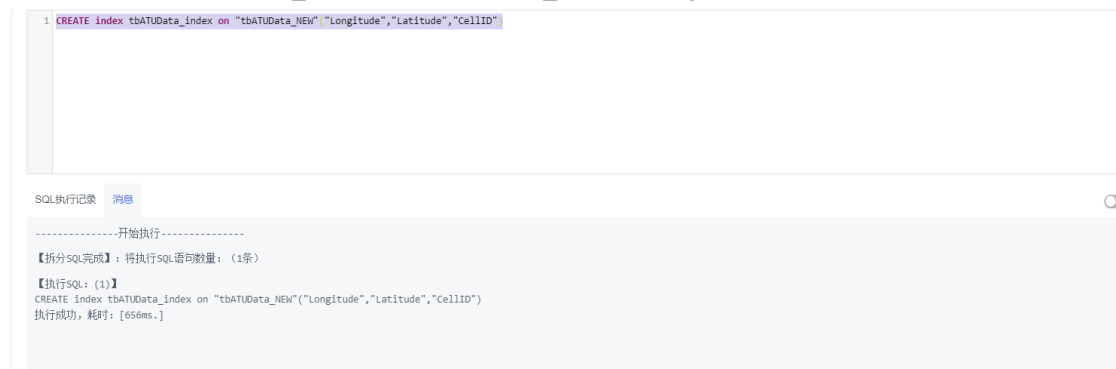
2.3.1 复合索引做前缀

1. 创建 tbATUDData 的备份表（去除索引以及其他约束）



2. 在 tbATUDData_NEW 创建组合索引

CREATE index tbATUDData_index on "tbATUDData_NEW"("Longitude","Latitude","CellID")



3. 最左前缀访问 atudate

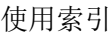
Sql:

EXPLAIN ANALYZE

SELECT * from "tbATUDData"

WHERE "Longitude" = 112.82765

结果



结论：符合最左前缀规则的查询可以利用索引大大提高搜索速度

3. 组合索引中除最左前缀索引外的其他索引与无索引的区别

Sql:

```
EXPLAIN ANALYZE
SELECT * from "tbATUData"
WHERE "Latitude" = 33.75314
```

不满足最左前缀规则的查询


```
1 EXPLAIN ANALYZE
2 SELECT * from "tbATUData_NEW"
3 WHERE "Latitude" = 33.75314
```

SQL执行记录 消息 结果集1 X

以下是EXPLAIN ANALYZE SELECT * from "tbATUData_NEW" WHERE "Latitude" = 33.75314的执行结果... 该表不可编辑。

	QUERY PLAN
1	[Bypass]
2	Index Scan using tbatudata_index on "tbATUData_NEW" (cost=0.00..12869.70 rows=155 width=212) (actual time=24.579..24.579 rows=0 loops=1)
3	Index Cond: ("Latitude" = 33.75314000000001919::double precision)
4	Total runtime: 24.646 ms

不使用索引

```
1 EXPLAIN ANALYZE
2 SELECT * from "tbATUData"
3 WHERE "Latitude" = 33.75314
```

SQL执行记录 消息 结果集1 X

以下是EXPLAIN ANALYZE SELECT * from "tbATUData" WHERE "Latitude" = 33.75314的执行结果... 该表不可编辑。

	QUERY PLAN
1	Seq Scan on "tbATUData" (cost=0.00..13674.12 rows=154 width=212) (actual time=76.958..76.958 rows=0 loops=1)
2	Filter: ("Latitude" = 33.75314000000001919::double precision)
3	Rows Removed by Filter: 394890
4	Total runtime: 77.007 ms

分析：尽管不满足最左前缀，但该索引仍然能加快搜索速度，但提高程度没有满足最左前缀的搜索高。故搜索速度由快到慢分别为：符合最左前缀规则的搜索，不符合最左前缀规则的搜索，不使用索引的搜索

2.3.2 多表连接操作，在连接属性上建立索引

1. 删除上个实验的索引

SQL提示 全屏模式

```
1 drop index tbATUData_index
```

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】

drop index tbATUData_index

执行成功，耗时：[15ms.]

2. 在 CellID 和 NCell_ID_1 上分别创建索引

```
1 create index "tbATUData_index1" on "tbATUData_NEW" ("CellID")
```

SQL执行记录

消息

-----开始执行-----
【拆分SQL完成】：将执行SQL语句数量：（1条）
【执行SQL：（1）】
create index "tbATUData_index1" on "tbATUData_NEW" ("CellID")
执行成功，耗时：[524ms.]

SQL提示

全屏模式

执行SQL(F8)

格式化(F9)

执行计划(F6)

我的SQL

SQL提示

全屏模式

```
1 create index "tbATUData_index2" on "tbATUData_NEW" ("NCell_ID_1")
```

SQL执行记录

消息

-----开始执行-----
【拆分SQL完成】：将执行SQL语句数量：（1条）
【执行SQL：（1）】
create index "tbATUData_index2" on "tbATUData_NEW" ("NCell_ID_1")
执行成功，耗时：[499ms.]

SQL提示

全屏模式

3. 比较有索引与无索引的两条语句的执行情况

(1) 不使用 CellID 索引

sql:

EXPLAIN ANALYZE

select DISTINCT "CellID","S_EARFCN"

from "tbATUData" ,"tbAdjCell"

where "CellID"='253903-0' AND "CellID"="S_SECTOR_ID";

以下是EXPLAIN ANALYZE select DISTINCT "CellID","S_EARFCN" from "tbATUData" ,"tbAdjCell" 的执行情况... 该表不可编辑。

复制行

复制列

列设置

QUERY PLAN	
1	HashAggregate (cost=18172.75..18172.76 rows=1 width=13) (actual time=241.522..241.523 rows=1 loops=1)
2	Group By Key: "tbATUData"."CellID", "tbAdjCell"."S_EARFCN"
3	-> Nested Loop (cost=0.00..17283.91 rows=177768 width=13) (actual time=1.200..165.981 rows=364240 loops=1)
4	-> Seq Scan on "tbATUData" (cost=0.00..13674.12 rows=1646 width=9) (actual time=0.011..02.682 rows=2320 loops=1)
5	Filter: (("CellID")::text = '253903-0')::text)
6	Rows Removed by Filter: 392570
7	-> Materialize (cost=0.00..1387.95 rows=108 width=13) (actual time=1.333..34.026 rows=364240 loops=2320)
8	-> Seq Scan on "tbAdjCell" (cost=0.00..1387.41 rows=108 width=13) (actual time=1.184..9.964 rows=157 loops=1)
9	Filter: (("S_SECTOR_ID")::text = '253903-0')::text)
10	Rows Removed by Filter: 69556
11	Total runtime: 241.628 ms

刷新

单行详情

(2) 使用 CellID 索引

EXPLAIN ANALYZE

select DISTINCT "CellID","S_EARFCN"

from "tbATUData_NEW" ,"tbAdjCell"

where "CellID"='253903-0' AND "CellID"="S_SECTOR_ID";

执行SQL(F8)

格式化(F9)

执行计划(F6)

我的SQL

SQL提示 ⓘ 全屏模式

```
1 EXPLAIN ANALYZE
2 select DISTINCT "CellID","S_EARFCN"
3 from "tbATUData_NEW" ,"tbAdjCell"
4 where "CellID"='253903-0' AND "CellID"="S_SECTOR_ID";
```

SQL执行记录 消息 结果集1 x

以下是EXPLAIN ANALYZE select DISTINCT "CellID","S_EARFCN" from "tbATUData_NEW" ,"tb. 的执行... ⓘ 该表不可编辑。

复制行

复制列

列设置

	QUERY PLAN
2	Group By Key: "tbATUData_NEW"."CellID", "tbAdjCell"."S_EARFCN"
3	-> Nested Loop (cost=20.27..7752.02 rows=176256 width=13) (actual time=1.548..84.364 rows=364240 loops=1)
4	-> Bitmap Heap Scan on "tbATUData_NEW" (cost=20.27..4161.14 rows=1632 width=9) (actual time=0.331..0.891 rows=2320 loops=1)
5	Recheck Cond: (("CellID")::text = '253903-0'::text)
6	-> Bitmap Index Scan on "tbATUData_index1" (cost=0.00..19.06 rows=1632 width=0) (actual time=0.302..0.302 rows=2320 loops=1)
7	Index Cond: (("CellID")::text = '253903-0'::text)
8	-> Materialize (cost=0.00..1387.95 rows=108 width=13) (actual time=1.373..33.699 rows=364240 loops=2320)
9	-> Seq Scan on "tbAdjCell" (cost=0.00..1387.41 rows=108 width=13) (actual time=1.288..9.927 rows=157 loops=1)
10	Filter: (("S_SECTOR_ID")::text = '253903-0'::text)
11	Rows Removed by Filter: 69556
12	Total runtime: 159.822 ms

(3) 不使用 NCell_ID_1 索引

EXPLAIN ANALYZE

select DISTINCT "NCell_ID_1","S_SECTOR_ID"

from "tbATUData" ,"tbAdjCell"

where "NCell_ID_1"= '259778-2' AND "NCell_ID_1"="N_SECTOR_ID";

SQL执行记录 消息 结果集1 x

以下是EXPLAIN ANALYZE select DISTINCT "NCell_ID_1","S_SECTOR_ID" from "tbATUData" ... 的执行... ⓘ 该表不可编辑。

复制行

复制列

列设置

	QUERY PLAN
1	HashAggregate (cost=20073.74..20073.75 rows=1 width=18) (actual time=305.206..305.227 rows=101 loops=1)
2	Group By Key: "tbATUData"."NCell_ID_1", "tbAdjCell"."S_SECTOR_ID"
3	-> Nested Loop (cost=0.00..18641.74 rows=286400 width=18) (actual time=0.033..194.675 rows=448404 loops=1)
4	-> Seq Scan on "tbATUData" (cost=0.00..13674.12 rows=3580 width=9) (actual time=0.012..90.249 rows=3476 loops=1)
5	Filter: (("NCell_ID_1")::text = '259778-2'::text)
6	Rows Removed by Filter: 391414
7	-> Materialize (cost=0.00..1387.81 rows=80 width=18) (actual time=0.276..41.748 rows=448404 loops=3476)
8	-> Seq Scan on "tbAdjCell" (cost=0.00..1387.41 rows=80 width=18) (actual time=0.017..11.007 rows=129 loops=1)
9	Filter: (("N_SECTOR_ID")::text = '259778-2'::text)
10	Rows Removed by Filter: 69584
11	Total runtime: 305.319 ms

刷新

单行详情

(4) 使用 NCell_ID_1 索引

EXPLAIN ANALYZE

select DISTINCT "NCell_ID_1","S_SECTOR_ID"

from "tbATUData_NEW" ,"tbAdjCell"

where "NCell_ID_1"= '259778-2' AND "NCell_ID_1"="N_SECTOR_ID";

SQL执行记录 消息 结果集1 X	
以下是EXPLAIN ANALYZE select DISTINCT "NCell_ID_1","S_SECTOR_ID" from "tbATUData_NE_的执... 该表不可编辑。 复制行 复制列 列设置	
	QUERY PLAN
2	Group By Key: "tbATUData_NEH"."NCell_ID_1", "tbAdjCell"."S_SECTOR_ID"
3	-> Nested Loop (cost=41.25..11481.85 rows=275920 width=18) (actual time=0.472..103.876 rows=448404 loops=1)
4	-> Bitmap Heap Scan on "tbATUData_NEW" (cost=41.25..6645.24 rows=3449 width=9) (actual time=0.445..1.354 rows=3476 loops=1)
5	Recheck Cond: (("NCell_ID_1")::text = '259778-2')::text)
6	-> Bitmap Index Scan on "tbATUData_index2" (cost=0.00..40.38 rows=3449 width=0) (actual time=0.427..0.427 rows=3476 loops=1)
7	Index Cond: (("NCell_ID_1")::text = '259778-2')::text)
8	-> Materialize (cost=0.00..1387.81 rows=80 width=18) (actual time=0.255..40.412 rows=448404 loops=3476)
9	-> Seq Scan on "tbAdjCell" (cost=0.00..1387.41 rows=80 width=18) (actual time=0.021..10.726 rows=129 loops=1)
10	Filter: (("S_SECTOR_ID")::text = '259778-2')::text)
11	Rows Removed by Filter: 69584
12	Total runtime: 212.849 ms

结果分析：尽管没有对两个键建立组合索引，但是有索引的搜索速度仍然快于没有索引的搜索

2.3.3 索引对小表查询的搜索

1. 对 tbOptCell 建立一个副本

CREATE table "tboptcell_new" as table "tboptcell"

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】

CREATE table "tboptcell_new" as table "tboptcell"

执行成功，耗时：[18ms.]

2. 在副本上创建索引

```
1 create index "tbOptCell_index" on "tboptcell_new" ("SECTOR_ID")
```

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】

create index "tbOptCell_index" on "tboptcell_new" ("SECTOR_ID")

执行成功，耗时：[15ms.]

3. 在 tbOptCell 上查询，观察是否用到索引：

EXPLAIN

SELECT "SECTOR_ID","CELL_TYPE" from "tboptcell"

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】

EXPLAIN

SELECT "SECTOR_ID","CELL_TYPE" from "tboptcell" where "SECTOR_ID" = '15114-128'

执行成功，当前返回：[2]行，耗时：[6ms.]

以下是EXPLAIN SELECT "SECTOR_ID","CELL_TYPE" from "tboptcell"的执行结果集

该表不可编辑。

复制行

复制列

列设置

	QUERY PLAN
1	Seq Scan on tboptcell (cost=0.00..10.64 rows=564 width=19)

可见使用的是顺序扫描，故没有使用索引

4. 在新表上查询：

1 EXPLAIN

2 SELECT "SECTOR_ID","CELL_TYPE" from "tboptcell_new"

SQL执行记录 消息 结果集1 X

以下是EXPLAIN SELECT "SECTOR_ID","CELL_TYPE" from "tboptcell_new"的执行结果集

该表不可编辑。

	QUERY PLAN
1	Seq Scan on tboptcell_new (cost=0.00..10.64 rows=564 width=19)

可见仍为顺序扫描，没有用到索引
若使用了 where 条件：则将使用索引

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】

EXPLAIN

SELECT "SECTOR_ID","CELL_TYPE" from "tboptcell_new" where "SECTOR_ID" = '15114-128'

执行成功，当前返回：[3]行，耗时：[6ms.]

1 EXPLAIN

2 SELECT "SECTOR_ID","CELL_TYPE" from "tboptcell_new" where "SECTOR_ID" = '15114-128'

SQL执行记录 消息 结果集1 X

以下是EXPLAIN SELECT "SECTOR_ID","CELL_TYPE" from "tboptcell_new" where "SECTOR_ID" = '15114-128'的执行结果集

该表不可编辑。

复制行 复制列 列设置

	QUERY PLAN
1	[Bypass]
2	Index Scan using "tboptcell_index" on tboptcell_new (cost=0.00..2.28 rows=1 width=19)
3	Index Cond: (("SECTOR_ID")::text = '15114-128'::text)

5. 强制使用索引查询 tbOptCell_new

(1) 禁止顺序扫描

1 set enable_seqscan = off

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】

set enable_seqscan = off

执行成功，耗时：[6ms.]

(2) 再次在 tbOptCell 上查询

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】

EXPLAIN

SELECT "SECTOR_ID","CELL_TYPE" from "tboptcell_new" where "SECTOR_ID" = '15114-128'

执行成功，当前返回：[3]行，耗时：[7ms.]

以下是EXPLAIN SELECT "SECTOR_ID","CELL_TYPE" from "tboptcell_new" where "SECTOR_ID" = '15114-128'的执行结果集

该表不可编辑。

复制行 复制列 列设置

	QUERY PLAN
1	[Bypass]
2	Index Scan using "tbOptCell_index" on tboptcell_new (cost=0.00..2.28 rows=1 width=19)
3	Index Cond: (("SECTOR_ID")::text = '15114-128'::text)

结果：在小表上强制使用索引反而会导致查询速度减慢。如果没有 `where` 语句，不管开不开顺序扫描，数据库都将默认顺序扫描

2.3.4 查询条件中函数对索引的影响

1. 删除原有的索引

1 drop index "tbATUData_index1";
2 drop index "tbATUData_index2";

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（2条）

【执行SQL：（1）】
drop index "tbATUData_index1";
执行成功，耗时：[12ms.]

【执行SQL：（2）】
drop index "tbATUData_index2";
执行成功，耗时：[12ms.]

2. 建立新的索引

1 CREATE index "tbATUData_index1" on "tbATUData_NEW" ("Longitude");
2 CREATE index "tbATUData_index2" on "tbATUData_NEW"("Latitude");
3

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（2条）

【执行SQL：（1）】
CREATE index "tbATUData_index1" on "tbATUData_NEW" ("Longitude");
执行成功，耗时：[401ms.]

【执行SQL：（2）】
CREATE index "tbATUData_index2" on "tbATUData_NEW"("Latitude");
执行成功，耗时：[405ms.]

3. 使用函数进行查询

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】
explain
select distinct B."CellID"
FROM "tbATUData_NEW" as A,"tbATUData_NEW" as B
where A."CellID"='253903-0'
and A."Longitude"=112.8284
and ABS(A."Latitude"-B."Latitude")<=0.001;
执行成功，当前返回：[8]行，耗时：[11ms.]

	QUERY PLAN
1	HashAggregate (cost=20157.00..20158.85 rows=105 width=9)
2	Group By Key: b."CellID"
3	-> Nested Loop (cost=0.00..19828.72 rows=131630 width=9)
4	Join Filter: (abs((a."Latitude" - b."Latitude")) <= .0010000000000000002::double precision)
5	-> Index Scan using "tbATUData_index1" on "tbATUData_NEW" a (cost=0.00..232.25 rows=1 width=8)
6	Index Cond: ("Longitude" = 112.82840000000000202::double precision)
7	Filter: (("CellID")::text = '253903-0'::text)
8	-> Seq Scan on "tbATUData_NEW" b (cost=0.00..12685.90 rows=394890 width=17)

4. 不使用函数进行查询

```
1 explain
2 select distinct B."CellID"
3 FROM "tbATUData_NEW" as A,"tbATUData_NEW" as B
4 where A."CellID"='253903-0'
5 and A."Longitude"=112.8284
6 and B."Latitude"<=0.001+A."Latitude"
7 and B."Latitude">=A."Latitude"-0.001
```

SQL执行记录 消息 结果集1 X

-----开始执行-----
【拆分SQL完成】：将执行SQL语句数量：（1条）
【执行SQL：（1）】
explain
select distinct B."CellID"
FROM "tbATUData_NEW" as A,"tbATUData_NEW" as B
where A."CellID"='253903-0'
and A."Longitude"=112.8284
and B."Latitude"<=0.001+A."Latitude"
and B."Latitude">=A."Latitude"-0.001
执行成功，当前返回：[10]行，耗时：[7ms.]

SQL执行记录 消息 结果集1 X

以下是explain select distinct B."CellID" FROM "tbATUData_NEW" as A,"tbATUData_NEW" ..的执行结果集。 该表不可编辑。

	QUERY PLAN
1	HashAggregate (cost=11072.82..11073.87 rows=105 width=9)
2	Group By Key: b."CellID"
3	-> Nested Loop (cost=677.57..10963.13 rows=43877 width=9)
4	-> Index Scan using "tbATUData_index1" on "tbATUData_NEW" a (cost=0.00..232.25 rows=1 width=8)
5	Index Cond: ("Longitude" = 112.828400000000000000000000000000::double precision)
6	Filter: (("CellID")::text = '253903-0'::text)
7	-> Bitmap Heap Scan on "tbATUData_NEW" b (cost=677.57..10292.11 rows=43877 width=17)
8	Recheck Cond: (("Latitude" <= (.00100000000000000000000000000000::double precision + a."Latitude")) AND ("Latitude" >= (a."Latitude" - .00100000000000000000000000000000::double precision)))
9	-> Bitmap Index Scan on "tbATUData_index2" (cost=0.00..666.60 rows=43877 width=0)
10	Index Cond: (("Latitude" <= (.00100000000000000000000000000000::double precision + a."Latitude")) AND ("Latitude" >= (a."Latitude" - .00100000000000000000000000000000::double precision)))

结果分析：第一个查询只在 A.longitude 条件中使用到了 longitude 索引，而没有使用到 latitude 索引。而第二个查询两个索引都使用到了，故第二个索引的速度会更快。而两个查询实际上是等价的，故当存在函数查询的时候，可以转换成等价的不存在函数的查询，以便使用索引以加快查询速度。

2.3.5 多表嵌入式 SQL 查询

1. 使用嵌套方式进行查询

```
explain analyze
select A."PCI"
from "tbPCIAssignment" as A
where A."SECTOR_ID" IN(
select B."SECTOR_ID"
from "tboptcell" as B
where B."CELL_TYPE"='优化区'
);
```

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）
【执行SQL：（1）】
explain analyze
select A."PCI"
from "tbPCIAssignment" as A
where A."SECTOR_ID" IN(
select B."SECTOR_ID"
from "tboptcell" as B
where B."CELL_TYPE"='优化区'
);
执行成功，当前返回：[9]行，耗时：[22ms.]

以下是explain analyze select A."PCI" from "tbPCIAssignment" as A where A."SECTOR_ID" = B."SECTOR_ID" 的执行结果集 🔗 该表不可编辑。 复制行 复制列 列设置

	QUERY PLAN
1	Hash Right Semi Join (cost=11.21..27.00 rows=246 width=4) (actual time=1.555..1.763 rows=276 loops=1)
2	Hash Cond: ((b."SECTOR_ID")::text = (a."SECTOR_ID")::text)
3	-> Seq Scan on tboptcell b (cost=0.00..12.05 rows=323 width=9) (actual time=0.013..0.120 rows=323 loops=1)
4	Filter: (("CELL_TYPE")::text = '优化区')::text)
5	Rows Removed by Filter: 241
6	-> Hash (cost=7.76..7.76 rows=276 width=13) (actual time=1.348..1.348 rows=276 loops=1)
7	Buckets: 32768 Batches: 1 Memory Usage: 13kB
8	-> Seq Scan on "tbPCIAssignment" a (cost=0.00..7.76 rows=276 width=13) (actual time=0.770..1.270 rows=276 loops=1)
9	Total runtime: 1.894 ms

刷新 进行详情

2. 通过连接的方式进行查询

```
explain analyze
select A."PCI"
from "tbPCIAssignment" as A,"tboptcell" as B
where B."CELL_TYPE"='优化区'
AND A."SECTOR_ID"=B."SECTOR_ID";
```

```
-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】
explain analyze
select A."PCI"
from "tbPCIAssignment" as A,"tboptcell" as B
where B."CELL_TYPE"='优化区'
AND A."SECTOR_ID"=B."SECTOR_ID";
执行成功，当前返回：[9]行，耗时：[9ms.]
```

以下是explain analyze select A."PCI" from "tbPCIAssignment" as A,"tboptcell" as B w... 的执行结果集 🔗 该表不可编辑。 复制行 复制列 列设置

	QUERY PLAN
1	Hash Join (cost=11.21..27.70 rows=323 width=4) (actual time=0.303..0.490 rows=276 loops=1)
2	Hash Cond: ((b."SECTOR_ID")::text = (a."SECTOR_ID")::text)
3	-> Seq Scan on tboptcell b (cost=0.00..12.05 rows=323 width=9) (actual time=0.014..0.129 rows=323 loops=1)
4	Filter: (("CELL_TYPE")::text = '优化区')::text)
5	Rows Removed by Filter: 241
6	-> Hash (cost=7.76..7.76 rows=276 width=13) (actual time=0.125..0.125 rows=276 loops=1)
7	Buckets: 32768 Batches: 1 Memory Usage: 13kB
8	-> Seq Scan on "tbPCIAssignment" a (cost=0.00..7.76 rows=276 width=13) (actual time=0.007..0.074 rows=276 loops=1)
9	Total runtime: 0.590 ms

实验结论：无论是从理论上还是从执行计划上看，两者都是等价的，与书上说的嵌套内查询会被调用多次不同，这应该是数据库进行查询优化后的结果，从执行计划上来看，两者均为分别经过筛选出优化区，并计算 hash 值后再连接在一起。

2.3.6 where 查询条件中复合查询条件 OR 对索引的影响

1. 首先删除 “tbATUData_new” 上原有的索引


```
1 drop index "tbATUData_index1";
2 drop index "tbATUData_index2";
3
```

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（2条）

【执行SQL：（1）】
drop index "tbATUData_index1";
执行成功，耗时：[12ms.]

【执行SQL：（2）】
drop index "tbATUData_index2";
执行成功，耗时：[10ms.]

2. 在 PCI 上创建索引

```
1 CREATE index tbATUData_index on "tbATUData_NEW" ("PCI")
```

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】
CREATE index tbATUData_index on "tbATUData_NEW" ("PCI")
执行成功，耗时：[350ms.]

3. 查看包含 A or B 类型的 sql 语句的查询计划

执行SQL(F8) 格式化(F9) 执行计划(F6) 我的SQL v SQL提示 全屏

```
1 explain
2 select *
3 from "tbATUData_NEW"
4 where "TAC"=14419 AND ("PCI"=166 OR "RSRP"=-96.81);
```

SQL执行记录 消息 结果集1 x 设置

以下是explain select * from "tbATUData_NEW" where "TAC"=14419 AND ("PCI"=166 OR "RSRP"=-96.81)的执行结果... 该表不可编辑。

	QUERY PLAN
1	Seq Scan on "tbATUData_NEW" (cost=0.00..15647.58 rows=1956 width=212)
2	Filter: (((("TAC" = 14419) AND ((("PCI" = 166) OR ("RSRP" = (-96.8100000000000023)::double precision))))

可见该查询的查询计划很简单，直接顺序扫描所有的数据并选出符合要求的数据，没有使用到索引

4. 查看等价的 Unionsql 语句的查询计划

```
1 explain
2 (select *
3 from "tbATUData_NEW"
4 where "TAC"=14419 AND "PCI"=166)
5 UNION
6 (select *
7 from "tbATUData_NEW"
8 where "TAC"=14419 AND "RSRP"=-96.81);
```

以下是explain (select * from "tbATUData_NEW" where "TAC"=14419 AND "PCI"=166) UNION...的执行结果。 该表不可编辑。

	QUERY PLAN
1	HashAggregate (cost=19009.00..19028.04 rows=1904 width=212)
2	Group By Key: bupt2019ds37."tbATUData_NEW".seq, bupt2019ds37."tbATUData_NEW".FileName, bupt2019ds37."tbATUData_NEW".Time, bupt2019ds37."tbATUData_NEW".PCI
3	-> Append (cost=18.05..18842.40 rows=1904 width=212)
4	-> Bitmap Heap Scan on "tbATUData_NEW" (cost=18.05..4163.01 rows=1585 width=212)
5	Recheck Cond: ("PCI" = 166)
6	Filter: ("TAC" = 14419)
7	-> Bitmap Index Scan on tbatudata_index (cost=0.00..17.68 rows=1632 width=0)
8	Index Cond: ("PCI" = 166)
9	-> Seq Scan on "tbATUData_NEW" (cost=0.00..14660.35 rows=399 width=212)
10	Filter: (("TAC" = 14419) AND ("RSP" = (-96.810000000000023)::double precision))

结果分析： 可见虽然这个查询使用到了索引，但由于需要分开扫描多次数据，导致在第一个 select 语句的预计时间就已经远超过了顺序扫描的时间，而最终 union 操作需要的时间代价更多，所以在查询中应该避免 union 操作

2.3.7 聚集索引中的索引设计

1. 删除原有的索引，在 longitude 以及 latitude 上分别重新建立索引。

```
1 drop index tbatudata_index;
2 CREATE index tbatudata_index1 on "tbATUData_NEW" ("Longitude");
3 CREATE index tbatudata_index2 on "tbATUData_NEW" ("Latitude");
```

SQL执行记录 消息

```
-----开始执行-----
【拆分SQL完成】：将执行SQL语句数量：（3条）
【执行SQL：（1）】
drop index tbatudata_index;
执行成功，耗时：[11ms.]
【执行SQL：（2）】
CREATE index tbatudata_index1 on "tbATUData_NEW" ("Longitude");
执行成功，耗时：[373ms.]
【执行SQL：（3）】
CREATE index tbatudata_index2 on "tbATUData_NEW" ("Latitude");
执行成功，耗时：[489ms.]
```

2. 查询无聚集索引运算的执行计划

```
explain
select "Latitude",avg("Longitude") as avg_Longitude
from "tbATUData"
group by "Latitude";
```

以下是explain select "Latitude",avg("Longitude") as avg_Longitude from "tbATUData" 的执行结果集。 该表不可编辑。

	QUERY PLAN
1	HashAggregate (cost=14661.35..14685.31 rows=1917 width=48)
2	Group By Key: "Latitude"
3	-> Seq Scan on "tbATUData" (cost=0.00..12686.90 rows=394890 width=16)

3. 查询有聚集索引的执行计划

```
explain
select "Latitude",avg("Longitude") as avg_Longitude
from "tbATUData_NEW"
group by "Latitude";
```

以下是explain select "Latitude",avg("Longitude") as avg_Longitude from "tbATUData_NEW" 的执行结果集。 该表不可编辑。

	QUERY PLAN
1	HashAggregate (cost=14660.35..14684.23 rows=1910 width=48)
2	Group By Key: "Latitude"
3	-> Seq Scan on "tbATUData_NEW" (cost=0.00..12685.90 rows=394890 width=16)

由结果看出，虽然建立了索引，但数据库仍采用了顺序查询，故两者的预计时间基本相同。

4. 强制使用索引

```
set enable_seqscan = off;
explain
select "Latitude",avg("Longitude") as avg_Longitude
from "tbATUDData_NEW"
group by "Latitude";
```



SQL 执行记录 消息 结果集 1 X 覆盖模式 ⓘ

以下是explain select "Latitude",avg("Longitude") as avg_Longitude from "tbATUDData_NEW" 的执行结果集 ⓘ 该表不可编辑。

	QUERY PLAN
1	GroupAggregate (cost=0.00..46874.19 rows=1910 width=48)
2	Group By Key: "Latitude"
3	-> Index Scan using tbatudata_index2 on "tbATUDData_NEW" (cost=0.00..44875.86 rows=394890 width=16)

由执行计划可见，强制开启索引后，顺序扫描变成了使用索引扫描，导致了执行效率更加低下。

2.3.8 select 子句中有无 distinct 的区别

1. 无 distinct

```
explain analyze
select "SECTOR_ID"
from "tboptcell"
where "EARFCN"=38400
and "CELL_TYPE"='保护带';
```



SQL 执行记录 消息 结果集 1 X 覆盖模式 ⓘ

以下是explain analyze select "SECTOR_ID" from "tboptcell" where "EARFCN"=38400 and "CELL_TYPE"='保护带' 的执行结果集 ⓘ 该表不可编辑。

	QUERY PLAN
1	Seq Scan on tboptcell (cost=0.00..13.46 rows=207 width=9) (actual time=0.070..0.128 rows=209 loops=1)
2	Filter: (("EARFCN" = 38400) AND (("CELL_TYPE")::text = '保护带')::text))
3	Rows Removed by Filter: 355
4	Total runtime: 0.175 ms

2. 有 distinct

```
explain analyze
select DISTINCT "SECTOR_ID"
from "tboptcell"
where "EARFCN"=38400
and "CELL_TYPE"='保护带';
```



SQL 执行记录 消息 结果集 1 X 覆盖模式 ⓘ

以下是explain analyze select DISTINCT "SECTOR_ID" from "tboptcell" where "EARFCN"=38400 and "CELL_TYPE"='保护带' 的执行结果集 ⓘ 该表不可编辑。

	QUERY PLAN
1	HashAggregate (cost=13.98..16.05 rows=207 width=9) (actual time=0.272..0.303 rows=209 loops=1)
2	Group By Key: "SECTOR_ID"
3	-> Seq Scan on tboptcell (cost=0.00..13.46 rows=207 width=9) (actual time=0.105..0.197 rows=209 loops=1)
4	Filter: (("EARFCN" = 38400) AND (("CELL_TYPE")::text = '保护带')::text))
5	Rows Removed by Filter: 355
6	Total runtime: 0.391 ms

有两次执行计划的结果可以看出，如果不添加 **distinct**，则数据库直接顺序扫描一遍数据就执行完毕了。而添加 **distinct** 后，数据库在扫描完成后还需执行 **group by** 语句，导致运行时间增加。但由于 **SECTOR_ID** 本身就是主键，没有重复的值，所以不应该添加

distinct 徒增运行时间

2.3.9 union 和 union all 的区别

1. 使用 union 进行查询

```
explain analyze
select "S_SECTOR_ID" from "tbAdjCell"
union
select "S_SECTOR_ID" from "tbSecAdjCell" ;
```

以下是explain analyze select "S_SECTOR_ID" from "tbAdjCell" union select "S_SECTOR_ID" from "tbSecAdjCell" 的执行结果集 [🔗 该表不可编辑。](#)

	QUERY PLAN
1	HashAggregate (cost=3556.97..4763.18 rows=128621 width=9) (actual time=65.738..66.016 rows=546 loops=1)
2	Group By Key: "tbAdjCell"."S_SECTOR_ID"
3	-> Append (cost=0.00..3255.42 rows=128621 width=9) (actual time=0.712..43.501 rows=128621 loops=1)
4	-> Seq Scan on "tbAdjCell" (cost=0.00..1213.13 rows=69713 width=9) (actual time=0.711..17.957 rows=69713 loops=1)
5	-> Seq Scan on "tbSecAdjCell" (cost=0.00..836.08 rows=58908 width=9) (actual time=0.745..12.451 rows=58908 loops=1)
6	Total runtime: 66.270 ms

基本执行顺序为分别顺序扫描，然后通过 group by 去除重复的数据

2. 使用 union all 进行查询

```
explain analyze
select "S_SECTOR_ID" from "tbAdjCell"
union ALL
select "S_SECTOR_ID" from "tbSecAdjCell" ;
```

以下是explain analyze select "S_SECTOR_ID" from "tbAdjCell" union ALL select "S_SECTOR_ID" from "tbSecAdjCell" 的执行结果集 [🔗 该表不可编辑。](#)

	QUERY PLAN
1	Result (cost=0.00..2049.21 rows=128621 width=9) (actual time=0.011..51.665 rows=128621 loops=1)
2	-> Append (cost=0.00..2049.21 rows=128621 width=9) (actual time=0.010..36.802 rows=128621 loops=1)
3	-> Seq Scan on "tbAdjCell" (cost=0.00..1213.13 rows=69713 width=9) (actual time=0.010..12.464 rows=69713 loops=1)
4	-> Seq Scan on "tbSecAdjCell" (cost=0.00..836.08 rows=58908 width=9) (actual time=0.014..10.321 rows=58908 loops=1)
5	Total runtime: 59.058 ms

由执行计划可知，union all 不会执行 group by 操作，所以不会完成去重，当确定数据没有重复的或没有数据唯一的要求的画，可以使用 union all 提高运行速度。

2.3.10 from 中存在多余的关系表，即查询非最简化

1. 查询语句最简化

```
explain analyze
select distinct "tboptcell" ."EARFCN"
from "tbPCIAssignment", "tboptcell"
where "tbPCIAssignment" ."SECTOR_ID"="tboptcell" ."SECTOR_ID";
```

SQL执行记录 消息 结果集 1 x [🔗 该表不可编辑。](#)

	QUERY PLAN
1	HashAggregate (cost=27.42..27.49 rows=7 width=4) (actual time=0.570..0.570 rows=1 loops=1)
2	Group By Key: tboptcell."EARFCN"
3	-> Hash Join (cost=11.21..26.73 rows=276 width=4) (actual time=0.317..0.520 rows=276 loops=1)
4	Hash Cond: ((tboptcell."SECTOR_ID")::text = ("tbPCIAssignment"."SECTOR_ID")::text)
5	-> Seq Scan on tboptcell (cost=0.00..10.64 rows=564 width=13) (actual time=0.009..0.009 rows=564 loops=1)
6	-> Hash (cost=7.76..7.76 rows=276 width=9) (actual time=0.120..0.120 rows=276 loops=1)
7	Buckets: 32768 Batches: 1 Memory Usage: 12kB
8	-> Seq Scan on "tbPCIAssignment" (cost=0.00..7.76 rows=276 width=9) (actual time=0.007..0.007 rows=276 loops=1)
9	Total runtime: 0.687 ms

2. 查询语句存在多余的关系表

```
explain analyze
select distinct "tboptcell"."EARFCN"
from "tbPCIAssignment", "tboptcell", "tbccl"
where "tbPCIAssignment"."SECTOR_ID" = "tboptcell"."SECTOR_ID"
and "tbccl"."SECTOR_ID" = "tboptcell"."SECTOR_ID"
```

以下是explain analyze select distinct "tboptcell"."EARFCN" from "tbPCIAssignment", "tbccl" 的执行结果

	QUERY PLAN
1	HashAggregate (cost=248.46..248.53 rows=7 width=4) (actual time=6.067..6.067 rows=1 loops=1)
2	Group By Key: tboptcell."EARFCN"
3	-> Nested Loop (cost=11.21..247.77 rows=276 width=4) (actual time=2.562..5.955 rows=276 loops=1)
4	Join Filter: (("tbPCIAssignment"."SECTOR_ID")::text = (tbccl."SECTOR_ID")::text)
5	-> Hash Join (cost=11.21..26.73 rows=276 width=22) (actual time=0.370..0.798 rows=276 loops=1)
6	Hash Cond: ((tboptcell."SECTOR_ID")::text = (tbPCIAssignment"."SECTOR_ID")::text)
7	-> Seq Scan on tboptcell (cost=0.00..10.64 rows=564 width=13) (actual time=0.012..0.149 rows=564 loops=1)
8	-> Hash (cost=7.76..7.76 rows=276 width=9) (actual time=0.181..0.181 rows=276 loops=1)
9	Buckets: 32768 Batches: 1 Memory Usage: 12kB
10	-> Seq Scan on "tbPCIAssignment" (cost=0.00..7.76 rows=276 width=9) (actual time=0.008..0.109 rows=276 loops=1)
11	-> Index Only Scan using tbccl_pkkey on tbccl (cost=0.00..0.79 rows=1 width=9) (actual time=4.743..4.851 rows=276 loops=276)

易知两者的查询结果应该是一样的。因为 **tbccl** 是多余的表，最终查询中既没有出现 **tbccl** 中的字段，也没利用到 **tbccl** 中的信息，但由于 **tbccl** 多执行了一次连接操作，所以导致最终运行时间大幅度增加。

遇到的问题及解决

问题一：在 2.3.3 中，禁止顺序扫描后，输入 sql 语句

```
Select "SECTOR_ID","CELL_TYPE"from "tboptcell_new"
```

执行计划仍显示为顺序扫描

问题解决：若 **where** 子句中没有查询条件的话，数据库默认为顺序查询。故想要体现出小表即使有索引可能也不用应该将 sql 改为

```
Select "SECTOR_ID","CELL_TYPE"from "tboptcell_new" where "SECTOR_ID" = 'xxx'
```

实验总结

这一次实验的实验大部分都比较简单，均为简单的基本优化以及编写 sql 语句时的基本注意事项，如 **distinct** 会导致数据排序引起查询速度变慢都是上课介绍过的内容。但第一次接触到了 **explain** 语句，通过这个语句可以将 sql 语句转换成执行计划，能让我更加深刻地了解 sql 语句是怎么执行的、怎么程式化地，也让我更好地明白哪里才是 sql 语句中最耗时间地地方，哪里是应该需要优化地地方。通过这一次的实验，我进一步加深熟悉了 **explain** 的用法，对我的帮助很大。