医疗门诊预约系统项目优化报告

组员：潘峰立、吴侃真、杨弈骋、刘焯、毛彦凯

一、业务背景描述

随着应用软件用户负载的增加和越来越复杂的应用环境，操作系统的各项性能参数，数据库的使用效率、用户的响应速度、系统的安全运行等性能问题主键成为系统必须考虑的指标之一。性能测试以及优化通常通过自动化的测试工具模拟多种正常、峰值以及异常负载的条件来对系统的各项性能指标进行测试，用来检测系统是否达到用户提出的性能指标，即使发现系统中存在的瓶颈，最后起到优化系统的目的。随着需求不断增加，特别是复杂逻辑的需求，一旦出现高并发量是，也将可能导致数据库主机无法承载，因此数据库优化亟待解决。

本项目就是基于上一阶段的模拟医疗系统数据库设计进行的优化。本报告将展示我们小组成员利用课内外的各种手段对于上一阶段的 naive 数据库设计进行的性能分析、测试、并最终提升查询性能的过程。通过本次优化，模拟医疗系统数据库的设计将更加合理，查询更加高效，更加接近符合实际功用的高效数据库。

二、优化目标

本次优化将从我们为每个不同角色设计的功能入手，提升查询性能，进而提高用户的体验。还将从更宏观的表结构入手，在降低数据库的负载压力的同时，再次提升用户的查询效率。我们更加关心的是查询性能，对于管理员功能中的大量写入修改操作的优化并不关心。

工作负载主要为大量病人并发查询医生接诊安排、添加预约，医生查询病人预约信息，同时高效且合理地维护如当前预约人数、医生平均分等信息，优化目标也基于此。

具体优化对象有：

医生和病人的功能：

（1）get_appointment_infoBydoctor：查询某个医生某个时间段的所有预约信息

（2）get_appointment_num：获得一个医生接诊安排的实时预约人数

（3）get_doctor_schedule：查询某个科室某个时间段的医生接诊安排

（4）add_appointment：病人添加新预约

以及一些会影响性能的 trigger

（5）update_doctor_avg_grade：更新医生的评分。

三、优化方案概述

将预约信息 appointment、医生接诊安排 doctor_schedule、科室接诊计划 dept_schedule 分成过期、有效两组。

过期信息以分表的形式存在对应的 cold 表里，存储引擎为 MyISAM。

有效信息的存储引擎为 InnoDB，利用索引进行优化。

设置 event，每天将过期数据从有效表转移到过期表中，若数据超过 cold 表设定大小，进行分表。

将每个医生接诊安排的实时预约人数存在内存表中，存储引擎为 Heap

对医生的平均评分的维护由延迟更新实现，设置 event，每周对所有医生的平均评分进行一次更新

将 get_doctor_schedule 病人根据科室、时间段获取医生接诊安排，

get_appointment_infoByDoctor 医生根据日期、时间获取病人预约信息，
两个查询中的嵌套子查询改为连接的查询形式，但优化效果不明显。

四、优化前性能分析及测试过程

以下测试数据库数据量为

病人预约数 appointment 239 万条

医生接诊安排 doctor_schedule 24000 条

科室接诊计划 dept_schedule 22000 条

病人 patient 20000 个

1）如何获取某个医生接诊安排对应的实时预约人数

性能分析：

优化前为函数 get_appointment_num()

```
42  EXPLAIN
43  SELECT count(*)
44  FROM appointment
45  WHERE doctor_schedule_id=17746 AND state!=0;
```

在研究不同引擎的特点的时候，我们了解到，Innodb 在查询 count(*)的时候不像 MyIsam 一样有一个特殊的文件保存着该信息，需要每次都去全表查询一次，显然，这种高昂的代价不是我们想要的。

信息 结果 1 剖析 状态

id	select_type	table	partitions	type	possible_keys	key
1	SIMPLE	appointment	(Null)	ref	FK_appointment_an	FK_appointment_and_appointment_unit

（续表）

key_len	ref	rows	filtered	Extra
5	const	100	90.00	Using where

测试过程及测试结果：

Mysqlslap 的参数设置：iteration=10，concurrency=100；

测试数据

1	get_appointment_num("3392");
2	get_appointment_num("29594");
3	get_appointment_num("11396");
4	get_appointment_num("28071");
5	get_appointment_num("10130");
6	get_appointment_num("32119");
7	get_appointment_num("33476");
8	get_appointment_num("13579");
9	get_appointment_num("32878");
10	get_appointment_num("17746");

测试结果：

优化前：

通过去掉一个最高分和最低分来统计平均值（*表示去掉的数据）

标号	1	2	3	4	5	6	7	8	9	10	AVG
平均时间/s	0.078	0.192	0.176	0.296*	0.189	0.204	0.071*	0.192	0.220	0.189	0.18
最短时间/s	0.016	0.063*	0.015	0.015*	0.016	0.031	0.015	0.016	0.031	0.015	0.019
最长时间/s	0.125*	1.109	1.016	1.141	1.156	0.140	0.140	1.110	1.156*	1.109	0.87

需要说明的是，当只有一个用户（即 c=1）的时候，查询的时间很短，在 1ms 左右。而大量并发让全表查询的缺点充分暴露，高代价的积累最终导致了查询时间变慢，所以需要进行优化。

该函数会之后在 get_doctor_schedule（）里面被调用，进而拉低病人查询医生的速度。

2）get_doctor_schedule　病人根据科室、时间段获取医生接诊安排

性能分析：

优化前代码：

```
SELECT name,building,room_id,start_time,end_time,get_appointment_num(doctor_schedule_id) as current_appointment_num,max_appointment
from (((doctor_schedule NATURAL JOIN dept_schedule) NATURAL JOIN time_slot)NATURAL JOIN person)NATURAL JOIN room
WHERE date_t=aDay AND start_time=beginTime AND end_time=endTime AND is_open=1 AND dept_name=dept;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	time_slot	(Null)	ALL	PRIMARY	(Null)	(Null)	(Null)	10	10	Using where
1	SIMPLE	dept_sche	(Null)	ref	PRIMARY,FK_dept_	FK_dept_	63	const	704	1.25	Using where
1	SIMPLE	doctor_sc	(Null)	ref	FK_doctor_dept_sc	FK_doct	5	hosipita	14	10	Using where
1	SIMPLE	room	(Null)	eq_ref	PRIMARY	PRIMAR	34	hosipita	1	100	Using index
1	SIMPLE	person	(Null)	eq_ref	PRIMARY	PRIMAR	41	hosipita	1	100	(Null)

表中 id 相同，说明按从上到下的顺序，依次进行了 5 次的 select 的操作。

直接将所有有关表 natural join 起来，运行速度依赖于 mysql 内部优化机制，同时在每条查询结果中调用函数 get_appointment_num 获取当前预约人数，其中用到了聚集函数，效率非常低下。

测试过程及结果：

测试数据：

1	get_doctor_schedule("2020-04-01","08:00:00","09:00:00","surgery");
2	get_doctor_schedule("2020-04-15","10:00:00","11:00:00","anesthesiology");
3	get_doctor_schedule("2020-04-22","13:00:00","14:00:00","endocrinology");
4	get_doctor_schedule("2020-05-02","08:00:00","09:00:00","surgery");
5	get_doctor_schedule("2020-05-12","15:00:00","16:00:00","orthopedic");
6	get_doctor_schedule("2020-06-05","09:00:00","10:00:00","endocrinology");
7	get_doctor_schedule("2020-06-16","16:00:00","17:00:00","orthopedic");
8	get_doctor_schedule("2020-06-27","10:00:00","11:00:00","orthopedic");
9	get_doctor_schedule("2020-07-01","09:00:00","10:00:00","anesthesiology");
10	get_doctor_schedule("2020-07-19","14:00:00","15:00:00","surgery");

Mysqlslap 参数 -c 100 -i 10

测试结果：

优化前：

标号	1	2	3	4	5	6	7	8	9	10	AVG
平均时间/s	0.148	.0.047	0.150	0.040	0.039	0.150	0.048	1.359	1.414	0.143	0.388
最短时间/s	0.016	0.016	0.016	0.015	0.015	0.015	0.046	0.235	0.250	0.031	0.065
最长时间/s	1.047	0.063	1.047	0.062	0.063	1.047	0.062	1.844	1.813	1.047	0.809

3）get_appointment_infoByDoctor 医生根据日期、时间获取病人预约信息

性能分析：

```
-- 查询某个医生某个时间段的所有预约信息
DROP PROCEDURE IF EXISTS `get_appointment_infoByDoctor`;
delimiter ;;
CREATE PROCEDURE `get_appointment_infoByDoctor`(aDay date,beginTime time,endTime time,doctor_id VARCHAR(13))
BEGIN
    SELECT *
    FROM (appointment NATURAL JOIN patient)
    WHERE doctor_schedule_id IN (
        SELECT doctor_schedule_id
        FROM (doctor_schedule NATURAL JOIN dept_schedule) NATURAL JOIN time_slot
        WHERE doctor_schedule.person_id = doctor_id AND date_t=aDay AND start_time=beginTime AND end_time=endTime
    ) AND state = 1;

END
```

在优化前的版本中存在一个 IN 语句导致的嵌套子查询，我们查阅了教材之后，考虑可以从嵌套子查询的角度进行优化。

从上述讨论可知，复杂嵌套子查询的优化是一个困难的工作，许多优化器仅做少量的去除相关工作。只要有可能，最好避免使用复杂嵌套子查询，因为我们不能确信查询优化器能够成功地将它们转换成一种能够有效运算的形式。

Explain 性能分析如下：

id	select_type	table	partitions	type	possible_keys	key
1	SIMPLE	time_slot	(Null)	ALL	PRIMARY	(Null)
1	SIMPLE	doctor_schedule	(Null)	ref	PRIMARY,FK_doctor	FK_doctor_and_appointment_unit
1	SIMPLE	dept_schedule	(Null)	eq_ref	PRIMARY,FK_dept_ti	PRIMARY
1	SIMPLE	appointment	(Null)	ref	FK_appointment_an	FK_appointment_and_appointment_unit
1	SIMPLE	patient	(Null)	eq_ref	PRIMARY	PRIMARY

（续表）

key_len	ref	rows	filtered	Extra
(Null)	(Null)	10	10.00	Using where
42	const	22	100.00	Using where
4	before2.doctor_sche	1	5.00	Using where
5	before2.doctor_sche	99	10.00	Using where
41	before2.appointmer	1	100.00	(Null)

测试过程及测试结果：

先创建一个 view，里面包含了所有可能查询出结果的数据。从该 view 中抽取数据，反复调用该函数，多次测试求平均值。

Mysqlslap 的参数设置：iteration=10，concurrency=100；

```
#通过这个知道可能有数据的组合
CREATE VIEW doc_query AS
SELECT *
    FROM (doctor_schedule NATURAL JOIN dept_schedule) NATURAL JOIN time_slot;
call  get_appointment_infoByDoctor("2020-06-01","08:00:00","09:00:00","237");
```

测试数据：

优化前测试原数据

1	get_appointment_infoByDoctor("2020-06-01","08:00:00","09:00:00","237");
2	get_appointment_infoByDoctor("2020-06-15","10:00:00","11:00:00","541");
3	get_appointment_infoByDoctor("2020-06-26","11:00:00","12:00:00","59");
4	get_appointment_infoByDoctor("2020-06-26","14:00:00","15:00:00","387");
5	get_appointment_infoByDoctor("2020-06-25","09:00:00","10:00:00","370");
6	get_appointment_infoByDoctor("2020-06-23","11:00:00","12:00:00","26");
7	get_appointment_infoByDoctor("2020-06-19","09:00:00","10:00:00","880");
8	get_appointment_infoByDoctor("2020-06-11","12:00:00","13:00:00","276");
9	get_appointment_infoByDoctor("2020-06-25","13:00:00","14:00:00","461");
10	get_appointment_infoByDoctor("2020-06-18","09:00:00","10:00:00","106");

测试结果：

优化前：通过去掉一个最高分和最低分来统计平均值（*表示去掉的数据）

标号	1	2	3	4	5	6	7	8	9	10	AVG
平均时间/s	0.250	0.198	0.198*	0.229	0.228	0.253*	0.229	0.245	0.240	0.236	0.264
最短时间/s	0.047	0.032	0.031	0.031	0.047	0.046	0.031	0.015*	0.063*	0.046	0.039
最长时间/s	1.391*	1.140	1.141	1.109	1.141	1.156	1.125	1.157	1.078*	1.204	1.147

4）add_appointment 病人添加就诊记录

性能分析：

优化前在插入预约信息时，要调用聚集函数 count，取得该医生接诊安排的实时预约人数，判断是否超过最大预约人数，考虑是否能避免每次插入调用聚集函数的开销。

测试过程及测试结果：

测试数据

1	add_appointment(3245,"10001");
2	add_appointment(3186,"17770");
3	add_appointment(11458,"19368");
4	add_appointment(51683,"6335");
5	add_appointment(59702,"7884");
6	add_appointment(60212,"9279");
7	add_appointment(78479,"19370");
8	add_appointment(95341,"19366");
9	add_appointment(97059,"13442 ");
10	add_appointment(106094,"6131");

优化前：

标号	1	2	3	4	5	6	7	8	9	10	AVG
平均时间/s	0.025	0.124	0.014	0.018	0.023	0.015	0.015	0.018	0.119	0.117	0.0488
最	0.000	0.000	0.000	0.000	0.015	0.000	0.000	0.000	0.015	0.000	0.003

短时间/s											
最长时间/s	0.032	1.032	0.031	0.031	0.032	0.031	0.032	0.032	1.016	1.016	0.3285

5）update_doctor_avg_grade　根据就诊记录更新医生的平均评分

性能分析：

优化前代码：

```
SET new_appointment_id=NEW.appointment_id;
SELECT doctor_schedule.person_id INTO p_id
FROM appointment JOIN doctor_schedule USING(doctor_schedule_id)
WHERE appointment_id=new_appointment_id;

SELECT avg(grade) into new_avg_grade
FROM (clinic_entry NATURAL JOIN appointment) JOIN doctor_schedule USING(doctor_schedule_id)
WHERE doctor_schedule.person_id=p_id;
```

我们将其作为一个 trigger，每次病人评价完一个医生之后触发

代码分析：

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	appointment	(Null)	const	PRIMARY,FK_appo	PRIMAR	4	const	1	100	(Null)
1	SIMPLE	doctor_schedule	(Null)	const	PRIMARY	PRIMAR	4	const	1	100	(Null)

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	doctor_schedule	(Null)	index	PRIMARY,FK_doctc	FK_doct	42	(Null)	105075	10	Using whe
1	SIMPLE	appointment	(Null)	ref	PRIMARY,FK_appo	FK_appc	5	hosipita	7	100	Using ind
1	SIMPLE	clinic_entry	(Null)	ref	FK_appointment_a	FK_appc	4	hosipita	1	100	(Null)

进行 5 个表的 select 的操作，其中一个 doctor_schedule 涉及到的行非常的多，这对于每一个病人结束治疗这一十分常见操作之后的 trigger，有这样大的查询操作，效率非常低下。

测试过程及测试结果：

测试数据：person_id 为 appointment_id 通过第一个 select 得出的

1	appointment_id=1，person_id=30
2	appointment_id=100，person_id=689
3	appointment_id=200，person_id=415
4	appointment_id=300，person_id=261
5	appointment_id=400，person_id=277
6	appointment_id=500，person_id=267
7	appointment_id=600，person_id=50
8	appointment_id=700，person_id=971
9	appointment_id=800，person_id=284
10	appointment_id=900，person_id=416

Mysqlslap 参数 -c 100 -i 10

测试结果：

第一个 select

标号	1	2	3	4	5	6	7	8	9	10	AVG
平均时间/s	0.015	0.018	0.114	0.014	0.014	0.010	0.014	0.012	0.015	0.014	0.024
最短时间/s	0.000	0.015	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.0015
最长时间/s	0.032	0.031	1.016	0.031	0.016	0.016	0.032	0.016	0.031	0.016	0.1237

第二个 select

标号	1	2	3	4	5	6	7	8	9	10	AVG
平均时间/s	1.418	1.514	1.297	1.428	1.421	1.568	1.435	1.303	1.853	2.028	1.5265
最短时间/s	0.688	0.703	0.625	0.753	0.734	0.688	0.750	0.562	0.781	1.922	0.8206
最长时间/s	1.813	1.765	1.735	1.765	1.781	1.828	1.750	1.750	2.078	2.093	1.8358

五、优化方案及前后测试结果对比

以下测试数据库数据量为

病人预约数 appointment 239 万条

医生接诊安排 doctor_schedule 24000 条

科室接诊计划 dept_schedule 22000 条

病人 patient 20000 个

1）如何获取某个医生接诊安排对应的实时预约人数

优化方案：

为了保留这个功能，考虑到预约位次这个属性有与缓存相似的特性，访问较为频繁，且如果丢失不会产生影响，可以通过 appointment 计算得出，我们采取了另一种做法。

建立了一个属性为(doctor_schedule_id,people_count)的 heap 表，直接存储该条医生接诊安排，加快访问速度，并且再插入预约信息时进行维护。

```
/*==============================================================*/
/* Table: schedule_people_realtime                              */
/*==============================================================*/
create table schedule_people_realtime
(
   doctor_schedule_id  int not null,
   people_count int not null,
   primary key (doctor_schedule_id)
) engine = Heap;
```

测试过程及测试结果：

优化后：

由于优化后我们取消了该函数，所以优化效果请参见 get_doctor_schedule、add_appointment 的优化结果。

2）get_doctor_schedule　病人根据科室、时间段获取医生接诊安排

优化方案：

优化后代码：

```
DROP PROCEDURE IF EXISTS get_doctor_schedule;
DELIMITER //
CREATE PROCEDURE get_doctor_schedule(aDay date,beginTime time,endTime time,dept varchar(20))
BEGIN

    DECLARE cur_people_count INT;
    DECLARE cur_dept_schedule_id INT;

    SELECT dept_schedule_id INTO cur_dept_schedule_id
    FROM (dept_schedule NATURAL JOIN time_slot)
    WHERE dept_name = dept AND date_t = aDay AND start_time = beginTime AND end_time = endTime;

    SELECT *
    FROM (doctor_schedule NATURAL JOIN room) NATURAL JOIN doctor NATURAL JOIN schedule_people_realtime
    WHERE dept_schedule_id = cur_dept_schedule_id AND is_open = 1;

END
//
DELIMITER ;
GRANT EXECUTE ON PROCEDURE get_doctor_schedule TO patient;
```

1、在 dept_schedule 中添加组合索引(dept_name,date,time_slot_id)，与查询语句中 WHERE 根据科室、时间契合。

在 doctor_schedule 中添加索引（dept_schedule_id），同样与 WHERE 契合。

2、将嵌套子查询改为具有连接的查询，去除相关性。

3、去除了 doctor_schedule 表中数据的函数调用过程。这个函数的意义在于获取当前预约人数，得到若是病人添加预约后的位次。

为了保留这个功能，考虑到预约位次这个属性有与缓存相似的特性，访问较为频繁，

且如果丢失不会产生影响，可以通过 appointment 计算得出，我们采取了另一种做法。

建立了一个属性为(doctor_schedule_id,people_count)的 heap 表，直接存储该条医生接诊安排，加快访问速度。

测试过程及结果：

测试数据：

1	get_doctor_schedule("2020-04-01","08:00:00","09:00:00","surgery");
2	get_doctor_schedule("2020-04-15","10:00:00","11:00:00","anesthesiology");
3	get_doctor_schedule("2020-04-22","13:00:00","14:00:00","endocrinology");
4	get_doctor_schedule("2020-05-02","08:00:00","09:00:00","surgery");
5	get_doctor_schedule("2020-05-12","15:00:00","16:00:00","orthopedic");
6	get_doctor_schedule("2020-06-05","09:00:00","10:00:00","endocrinology");
7	get_doctor_schedule("2020-06-16","16:00:00","17:00:00","orthopedic");
8	get_doctor_schedule("2020-06-27","10:00:00","11:00:00","orthopedic");
9	get_doctor_schedule("2020-07-01","09:00:00","10:00:00","anesthesiology");
10	get_doctor_schedule("2020-07-19","14:00:00","15:00:00","surgery");

Mysqlslap 参数 -c 100 -i 10

测试结果：

优化后：

标号	1	2	3	4	5	6	7	8	9	10	AVG
平均时间/s	0.031	0.037	0.031	0.129	0.032	0.137	0.034	0.031	0.039	0.029	0.053
最短时间/s	0.016	0.015	0.015	0.000	0.000	0.015	0.016	0.000	0.016	0.000	0.0093
最长时间/s	0.047	0.047	0.047	1.031	0.047	1.047	0.047	0.047	0.047	0.047	0.2454

结果分析：

优化后平均时间明显降低

对优化后代码进行 explain 分析，

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_l	ref	rows	filtered	Extra
1	SIMPLE	time_slot	(Null)	ALL	PRIMARY	(Null)	(Null)	(Null)	10	10	Using whe
1	SIMPLE	dept_schedule	(Null)	ref	idx_time,FK_dept_timeslot	idx_time	108	const,cc	1	100	Using ind

id	select_type	table	type	possible_keys	key	key_l	ref	rows	filter	Extra
1	SIMPLE	doctor_schedule	ref	idx_dept_schedule_id,FK_docto	idx_dept_schedule_id	5	const	10	10	Using whe
1	SIMPLE	room	eq_ref	PRIMARY	PRIMARY	34	hospita	1	100	Using ind
1	SIMPLE	doctor	eq_ref	PRIMARY	PRIMARY	41	hospita	1	100	(Null)

发现在两次 SELECT 中均利用到了索引，使得在根据 dept_schedule_id 查询 doctor_schedule 的时候涉及到的行数从原来的 703 行减少到了 10，从而大大提高了查询效率。

由于原本的查询语句较为简单，mysql 的内部优化机制使得去除相关性的改进对运算速度几乎没有影响。

3）get_appointment_infoByDoctor 医生根据日期、时间获取病人预约信息

优化方案：

考虑将嵌套子查询改为连接的查询

```
-- 查询某个医生某个时间段的所有预约信息
DROP PROCEDURE IF EXISTS `get_appointment_infoByDoctor`;
delimiter ;;
CREATE PROCEDURE `get_appointment_infoByDoctor`(aDay date,beginTime time,endTime time,doctor_id VARCHAR(13))
BEGIN
    DECLARE cur_doctor_schedule_id INT;

    -- 找对应医生接诊安排
    SELECT doctor_schedule_id INTO cur_doctor_schedule_id
    FROM (doctor_schedule NATURAL JOIN dept_schedule) NATURAL JOIN time_slot
    WHERE date_t = aDay AND start_time = beginTime AND end_time = endTime AND person_id = doctor_id;

    -- 找对应预约安排
    SELECT *
    FROM (appointment NATURAL JOIN patient)
    WHERE doctor_schedule_id = cur_doctor_schedule_id AND state = 1;
END
```

此外，为 appointment 添加索引（doctor_id），与第二个查询中的 WHERE 匹配，加速 SELECT 操作

```
create table appointment
(
    appointment_id       int not null auto_increment,
    doctor_schedule_id   int,
    person_id            varchar(13) not null,
    precedence           int not null,
    state                int not null,
    primary key (appointment_id),
    index idx_doctor_schedule_id (doctor_schedule_id)
) engine = InnoDB, auto_increment = 1;
```

测试过程及测试结果：

先创建一个 view，里面包含了所有可能查询出结果的数据。从该 view 中抽取数据，反复调用该函数，多次测试求平均值。

Mysqlslap 的参数设置：iteration=10，concurrency=100；

```
#通过这个知道可能有数据的组合
CREATE VIEW doc_query AS
SELECT *
    FROM (doctor_schedule NATURAL JOIN dept_schedule) NATURAL JOIN time_slot;
call  get_appointment_infoByDoctor("2020-06-01","08:00:00","09:00:00","237");
```

测试数据：

优化后测试原数据

1	get_appointment_infoByDoctor("2020-06-08","08:00:00","09:00:00","391");
2	get_appointment_infoByDoctor("2020-06-23","08:00:00","09:00:00","81");
3	get_appointment_infoByDoctor("2020-06-10","09:00:00","10:00:00","980");
4	get_appointment_infoByDoctor("2020-06-02","10:00:00","11:00:00","375");
5	get_appointment_infoByDoctor("2020-06-04","11:00:00","12:00:00","116");
6	get_appointment_infoByDoctor("2020-06-08","12:00:00","13:00:00","254");
7	get_appointment_infoByDoctor("2020-06-02","13:00:00","14:00:00","192");
8	get_appointment_infoByDoctor("2020-06-26","14:00:00","15:00:00","832");
9	get_appointment_infoByDoctor("2020-06-22","14:00:00","15:00:00","650");
10	get_appointment_infoByDoctor("2020-06-08","15:00:00","16:00:00","530");

测试结果：

优化后：

标号	1	2	3	4	5	6	7	8	9	10	AVG
平均时间/s	0.093*	0.204	0.204	0.226	0.326	0.223	0.233	0.212	0.234	0.351*	0.233
最短时间/s	0.031	0.015*	0.047	0.062	0.047	0.031	0.063	0.047	0.047	0.078*	0.047
最长时间/s	0.141*	1.109	1.156	1.188	1.266	1.140	1.157	1.125	1.219	1.313*	1.17

结果分析及结论：

优化前后两者的测试时间相差无几。优化效果很不明显。我们一开始认为有以下可能原因：1）测试方法不正确。应该处理大量并发访问的情况。

2）优化过程不对。

3）数据量不够大导致结果不明显。

检查 Explain 之后我们发现了真正的原因。

优化前：

```
EXPLAIN
SELECT *
  FROM (appointment NATURAL JOIN patient)
  WHERE doctor_schedule_id IN (
    SELECT doctor_schedule_id
    FROM (doctor_schedule NATURAL JOIN dept_schedule) NATURAL JOIN time_slot
    WHERE doctor_schedule.person_id = "237" AND date_t="2020-06-01" AND start_time="08:00:00" AND end_time="09:00:00"
  ) AND state = 1;
```

id	select_type	table	partitions	type	possible_keys	key
1	SIMPLE	time_slot	(Null)	ALL	PRIMARY	(Null)
1	SIMPLE	doctor_schedule	(Null)	ref	PRIMARY,FK_doctor	FK_doctor_and_appointment_unit
1	SIMPLE	dept_schedule	(Null)	eq_ref	PRIMARY,FK_dept_ti	PRIMARY
1	SIMPLE	appointment	(Null)	ref	FK_appointment_an	FK_appointment_and_appointment_unit
1	SIMPLE	patient	(Null)	eq_ref	PRIMARY	PRIMARY

（续表）

key_len	ref	rows	filtered	Extra
(Null)	(Null)	10	10.00	Using where
42	const	22	100.00	Using where
4	before2.doctor_sche	1	5.00	Using where
5	before2.doctor_sche	99	10.00	Using where
41	before2.appointmer	1	100.00	(Null)

优化后：

```
EXPLAIN
  SELECT doctor_schedule_id
  FROM (doctor_schedule NATURAL JOIN dept_schedule) NATURAL JOIN time_slot
  WHERE date_t="2020-06-08" AND start_time="08:00:00" AND end_time = "09:00:00" AND person_id = "391";

EXPLAIN
  SELECT *
  FROM (appointment NATURAL JOIN patient)
  WHERE doctor_schedule_id = 13508 AND state = 1;
```

id	select_type	table	partitions	type	possible_keys	key
1	SIMPLE	time_slot	(Null)	ALL	PRIMARY	(Null)
1	SIMPLE	doctor_schedule	(Null)	ref	idx_dept_schedule_i	FK_doctor_and_appointment_unit
1	SIMPLE	dept_schedule	(Null)	eq_ref	PRIMARY,FK_dept_ti	PRIMARY

key_len	ref	rows	filtered	Extra
(Null)	(Null)	10	10.00	Using where
42	const	18	100.00	Using where
4	after2.doctor_sched	1	5.00	Using where

id	select_type	table	partitions	type	possible_keys	key
1	SIMPLE	appointment	(Null)	ref	idx_doctor_schedule	idx_doctor_schedule_id
1	SIMPLE	patient	(Null)	eq_ref	PRIMARY	PRIMARY

key_len	ref	rows	filtered	Extra
5	const	100	10.00	Using where
41	after2.appointment.	1	100.00	(Null)

由于 select 语句结构较为简单，消除嵌套子查询并没有很大效果。

此外，还可以发现在 key 那一列（代表着使用何种索引）。在优化后确实使用了我们创建的 doctor_schedule_id 索引，但是在优化前，自动使用了外键的索引 FK_appointment_and_appointment_unit。所以才导致了前后优化效果不明显。

最终测试说明，这个不是性能瓶颈所在。

5）add_appointment 病人添加就诊记录

优化方案：

借用 1）的优化思路，每次获取实时预约人数通过内存表 schedule_people_realtime，提高效率，并在插入完成后更新 schedule_people_realtime 中相应实时预约人数。

测试过程及测试结果：

测试数据

1	add_appointment(3245,"10001");
2	add_appointment(3186,"17770");
3	add_appointment(11458,"19368");
4	add_appointment(51683,"6335");
5	add_appointment(59702,"7884");
6	add_appointment(60212,"9279");
7	add_appointment(78479,"19370");
8	add_appointment(95341,"19366");
9	add_appointment(97059,"13442 ");
10	add_appointment(106094,"6131");

优化后：

标号	1	2	3	4	5	6	7	8	9	10	AVG
平均时间/s	0.018	0.018	0.014	0.020	0.022	0.043	0.018	0.014	0.017	0.023	0.0207
最短时间/s	0.000	0.000	0.000	0.015	0.016	0.125	0.000	0.000	0.000	0.015	0.0171
最长时间/s	0.032	0.047	0.016	0.032	0.047	0.141	0.047	0.031	0.032	0.047	0.0472

结果分析：

将聚集函数 count 替换为从内存表 schedule_people_realtime 中查询数据，插入速度约提升了一倍，达到了理想的效果。

5）update_doctor_avg_grade　根据就诊记录更新医生的平均评分

优化方案：

考虑医生的平均评分是一个较为模糊的数据，且对时效性的要求比较低，我们采用了延迟更新的策略，设置一个定时器，每周更新一次所有医生的评分。

```
DROP PROCEDURE IF EXISTS update_docotor_avg_grade;
delimiter //
CREATE PROCEDURE update_doctor_avg_grade()
BEGIN
    UPDATE doctor
    SET avg_grade = (
        SELECT avg(grade)
        FROM (doctor_schedule_cold NATURAL JOIN appointment_cold) NATURAL JOIN clinic_entry
        WHERE person_id = doctor.person_id
    )
    WHERE TRUE;
END
//
delimiter ;

SET GLOBAL event_scheduler = 1;

CREATE EVENT event_update_doctor
ON SCHEDULE EVERY 7 DAY STARTS '2020-04-01 23:00:00'
DO CALL update_doctor_avg_grade();
```

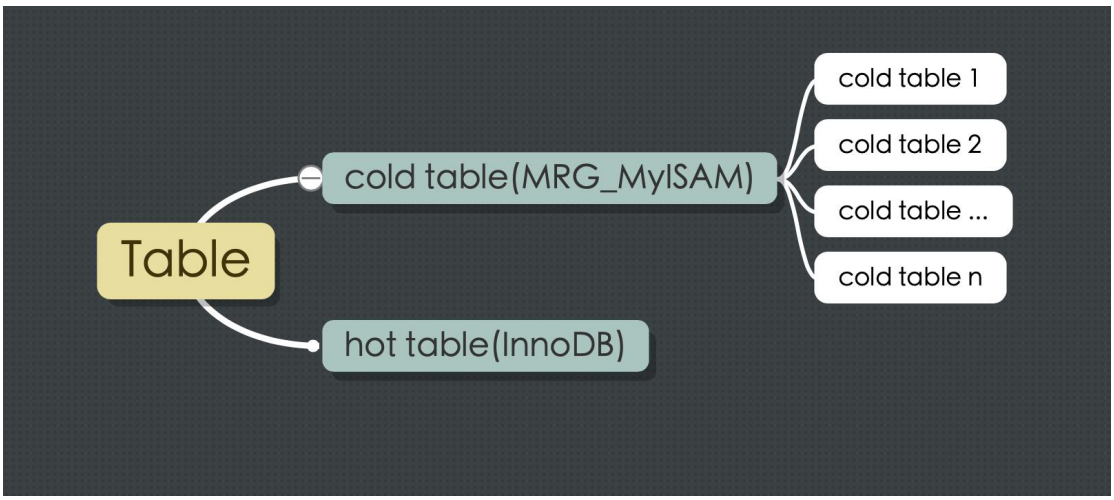
结果分析：

优化前一次插入就诊记录带来的 trigger 代价非常大，长达 1s，严重影响系统正常工作。在改为延迟更新后，每周只需对所有医生评分进行一次更新，更新频率大大降低，消除了插入就诊记录带来的瓶颈。

6）淘汰数据

优化方案：

考虑到 MyISAM 具有较高的读写性能，而 innoDB 支持事务与故障恢复的特点，我们决定对 appointment，doctor_schedule，dept_schedule 这三个表进行冷热数据的划分。考虑到预约、门诊安排等数据在过期之后就没有修改的意义了，我们将未过期的数据存在一个以 innoDB 为引擎的 hot table 中，而过期数据存储在以 MyISAM 为引擎的 cold table 中，并对其进行水平分表。下图为基本架构。



性能分析：

考虑到数据淘汰的操作不会在系统的峰值时期执行，对性能要求其实并不高。于是我们对其进行了简单测试。测试了 20 万数据量的 hot table 在一次 dump to cold table 操作后（根据我们组对单表查询性能与数据量的关系进行的测试，我们最终决定每一个 cold table 设置为 1 万数据的最大容量）。

测试结果：

	appointment	doctor_schedule	dept_schedule
平均时间 /s	3.368	5.464	4.698
最短时间 /s	2.879	3.290	3.116
最长时间 /s	4.035	6.311	5.796

结果分析：

该测试规模远大于医院一天实际会产生的预约数据，且在空闲时段该操作也可快速完成，故该结果达到预期。