

A Database for an Electronic Medical Record System

Student Clinic Module Project-2

Group Members

Abhishek Deshpande

Sonika Rajan

Suyash Nande

Requirement Specification

Student Clinic is the database project designed to facilitate student health record in a university. This system will be used by the doctor and clinic staff in the student health center of the university. It is developed to improve the clinic management and automates the workflow that happens in a student clinic. It will make it easier for the retrieval of history information of the student.

In this project, we have added few more tables to enhance the database and make it more efficient. We have added appointment table and receptionist table. When the student will visit on clinic, he will connect with the receptionist where they will be asked to provide their name, date of birth, contact number, address and type of problems. Student will make registration first. If the student never registered before, student information is collected and stored in the database. Now, the receptionist creates an appointment for the student and student is then assigned to the doctor, which may yield to diagnosis of the disease based on the problem category.

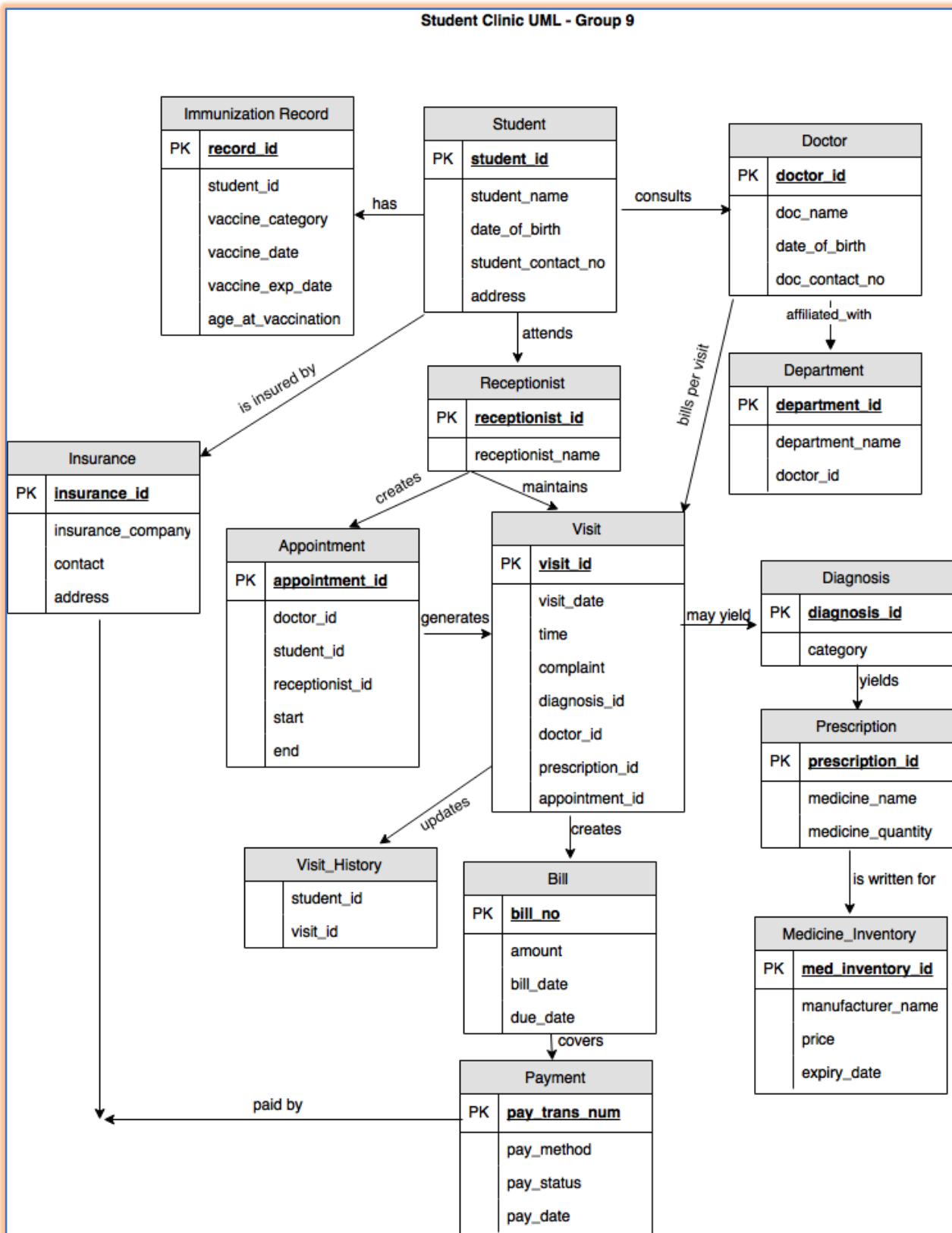
This diagnosis may further yield to treatment and doctor will then prescribe medicines which contains the medicine name, duration and quantity. Once the student gets the treatment, the doctor will send the report to the clinical staff.

The staff will view the report and complete the student record. After that, the staff will prepare the bills for the student. Student will also provide with an option for making payments. They can make their payments either by self or by medical insurance. Student will also have their immunization record maintained by clinical staff. It is easy for the management to maintain record about the student, the time for retrieving the information needed will be less compare to the manual. Then the staff will update the visit information and the student record will be kept in database.

Features

- We have implemented stored procedures in the form of API that allows an EMR system developer to work with the database without having to write SQL statements.
- We have also provided security feature wherein only the authorized users will be able to access the database.
- In addition to that, we have also implemented privacy features in the form of role based access control to limit the functions the user can perform.
- To achieve the regulatory requirements, we have added audit trail feature for the transactional tables wherein the logs for every change to the database made are captured.
- To optimize the performance further, we have implemented the usage of views indexes and views.

UML Data Model



The diagram illustrates a complex database schema for a medical system. It features numerous tables, each with its own set of attributes and constraints. Key tables include:

- privilege**: Attributes include privilege_id, role, and various privilege types (Select_priv, Insert_priv, Update_priv, Delete_priv).
- visit_history**: Attributes include visit_id and student_id.
- prescription_medicine**: Attributes include prescription_id and med_inventory_id.
- medicine**: Attributes include med_inventory_id, manufacturer_name, price, quantity, and expiry_date.
- insurance**: Attributes include insurance_id, insurance_company, contact, and address.
- user_privilege**: Attributes include user, host, role, table_name, privilege_id, and flag.
- immunization_record**: Attributes include record_id, student_id, vaccine_category, vaccine_date, vaccine_exp_date, and age_at_vaccination.
- bill**: Attributes include bill_no, amount, amount_paid, bill_date, and due_date.
- student_immunization_record**: Attributes include record_id and student_id.
- student**: Attributes include student_id, student_name, date_of_birth, student_contact_no, and address.
- appointment**: Attributes include appointment_id, doctor_id, student_id, receptionist_id, start, and end.
- doctor**: Attributes include doctor_id, doc_name, doc_contact_no, and dpt_id.
- receptionist**: Attributes include receptionist_id and receptionist_name.
- student_insurance**: Attributes include appointment_id, student_id, and insurance_id.
- audit_trial_payment**: Attributes include action, revision, pay_datetime, username, pay_trans_num, pay_method, pay_status, and pay_date.
- student_bill**: Attributes include bill_no, student_id, diagnosis_id, and prescription_id.
- user_role**: Attributes include user, password, role, and host.
- diagnosis**: Attributes include diagnosis_id and category.
- visit**: Attributes include visit_id, visit_date, time, complaint, diagnosis_id, doctor_id, prescription_id, and appointment_id.
- prescription**: Attributes include prescription_id, medicine_name, medicine_quantity, and appointment_id.
- payment**: Attributes include pay_trans_num, pay_method, pay_status, and pay_date.
- affiliated_with**: Attributes include dpt_id, doctor_id, and dpt_name.
- audit_trial_visit**: Attributes include action, revision, dt_datetime, user_name, visit_id, visit_date, time, complaint, diagnosis_id, doctor_id, and prescription_id.
- bill_pending_view**: A view representing pending bills.
- medicine_for_student**: A view representing medicines for students.
- bill_paid_view**: A view representing paid bills.
- dept_with_max_visit**: A view representing departments with the maximum number of visits.

The diagram uses standard ER notation to show relationships between tables, including primary keys, foreign keys, and many-to-many relationships. Views are represented by yellow boxes with a magnifying glass icon.

Procedures for database insertion and modification

Insert Procedure

To insert new doctor data

```
DELIMITER $$
CREATE PROCEDURE `NewDoctor_insert`
(IN DOCTOR_ID varchar(12),
IN DOCTOR_NAME varchar(45),
IN DOCTOR_CONTACT varchar(30))
BEGIN
INSERT INTO doctor(doctor_id,doc_name,doc_contact_no)
values(DOCTOR_ID,DOCTOR_NAME,DOCTOR_CONTACT);
END$$
DELIMITER ;

CALL `student_clinic_db_final`.`NewDoctor_insert`('108', 'Dr.Jake', '9803655355');
```

To insert new student data

```
DELIMITER $$
CREATE PROCEDURE `NewStudent_INSERT`(IN Student_ID varchar(12),
IN Student_Name varchar(90),
IN Student_DOB date,
IN Student_Contact varchar(30),
IN Student_Address varchar(90)
)
BEGIN
INSERT INTO student
values(Student_ID,Student_Name,Student_DOB,Student_Contact,Student_Address);
END$$
DELIMITER ;

SET @Student_ID='55';
SET @Student_Name='Harika';
SET @Student_DOB='1993-01-02';
SET @Student_Contact='9803650355';
SET @Student_Address='UTD,NC';

CALL
`student_clinic_db_final`.`NewStudent_INSERT`(@Student_ID,@Student_Name,@Student_DO
B,@Student_Contact,@Student_Address);
```

To insert new insurance company

```
DELIMITER $$
CREATE PROCEDURE `InsuranceCompany_insert`(IN COMPANY_ID varchar(12),
IN COMPANY_NAME varchar(45),
IN COMPANY_CONTACT varchar(30),
IN COMPANY_ADDRESS varchar(90)
)
BEGIN
INSERT INTO insurance(insurance_id,insurance_company,contact,address)
values(COMPANY_ID,COMPANY_NAME,COMPANY_CONTACT,COMPANY_ADDRESS);
END$$
DELIMITER ;

CALL `student_clinic_db_final`.`InsuranceCompany_insert`('50','ISO','7645423457','ABCD');
```

To insert the visit information

```
DELIMITER $$
CREATE PROCEDURE `VISIT_insert`
(IN VISIT_ID varchar(11),
IN VISIT_DATE date,
IN VISIT_TIME time,
IN VISIT_COMPLAINT varchar(90),
IN DIAGNOSIS_ID varchar(11),
IN DOC_ID varchar(12),
IN PRESCR_ID varchar(11)
)
BEGIN
INSERT INTO visit(visit_id,visit_date,time,complaint,diagnosis_id,doctor_id,prescription_id)
values(VISIT_ID,VISIT_DATE,VISIT_TIME,VISIT_COMPLAINT,DIAGNOSIS_ID,DOC_ID,PRESKR_ID);
END$$
DELIMITER ;

CALL `student_clinic_db_final`.`VISIT_insert`('6','2017-05-03','01:00:00','Throat
infection','14','101','12');
```

To enter new diagnosis record

```
DELIMITER $$
CREATE PROCEDURE `New_diagnosis`
```

```
(IN DIAGNOSIS_ID varchar(12),IN DIAGNOSIS_CATEGORY varchar(45))
BEGIN
INSERT INTO diagnosis(diagnosis_id,category) values(DIAGNOSIS_ID,DIAGNOSIS_CATEGORY);
END$$
DELIMITER ;

CALL `student_clinic_db_final`.`New_diagnosis`('16','Throatinfection');
```

To create new bills:

```
DELIMITER $$
CREATE PROCEDURE `Newbill_insert`(IN BILL_ID varchar(11),
IN BILL_AMOUNT decimal(9,6),
IN CREATE_DATE date,
IN BILL_DUE date
)
BEGIN
INSERT INTO bill(bill_no,amount,bill_date,due_date)
values(BILL_ID,BILL_AMOUNT,CREATE_DATE,BILL_DUE);
END$$
DELIMITER ;

CALL `student_clinic_db_final`.`Newbill_insert`('1',' 555','2017-04-03','2017-05-03');
CALL `student_clinic_db_final`.`Newbill_insert`('2',' 550','2017-04-03','2017-05-04');
CALL `student_clinic_db_final`.`Newbill_insert`('3',' 111','2017-03-03','2017-04-05');
CALL `student_clinic_db_final`.`Newbill_insert`('4',' 220','2017-05-03','2017-06-03');
```

To create prescription:

```
DELIMITER $$
CREATE PROCEDURE `Prescription_insert`(IN PRESCRIP_ID varchar(11),
IN MED_QTY int(11)
)
BEGIN
INSERT INTO prescription(prescription_id,medicine_quantity) values(PRESCRIP_ID,MED_QTY);
END$$
DELIMITER ;

CALL `student_clinic_db_final`.`Prescription_insert`('14','50');
CALL `student_clinic_db_final`.`Prescription_insert`('15','75');
CALL `student_clinic_db_final`.`Prescription_insert`('16','95');
```

Insert into prescription medicine:

```
DELIMITER $$
CREATE PROCEDURE `Prescription_med_insert`(IN PRESCRIP_ID varchar(11),
IN MED_INVTRY_ID varchar(11))
BEGIN
INSERT INTO prescription_medicine(prescription_id,med_inventory_id)
values(PRESCRIP_ID,MED_INVTRY_ID);
END$$
DELIMITER ;
```

```
CALL `student_clinic_db_final`.`Prescription_med_insert`('45','897');
CALL `student_clinic_db_final`.`Prescription_med_insert`('55','900');
CALL `student_clinic_db_final`.`Prescription_med_insert`('34','789');
```

To insert into payment table:

```
DELIMITER $$
CREATE PROCEDURE `Payment_insert`(IN TRANSACTION_NUM varchar(11),
IN METHOD varchar(90),
IN PAYSTATUS varchar(45),
IN PAYDATE date
)
BEGIN
INSERT INTO Payment(pay_trans_num,pay_method,pay_status,pay_date)
values(TRANSACTION_NUM,METHOD,PAYSTATUS,PAYDATE);
END$$
DELIMITER ;

CALL `student_clinic_db_final`.`Payment_insert`('107','SELF','PAID','2017-11-09');
CALL `student_clinic_db_final`.`Payment_insert`('108','INSURANCE','PAID','2017-01-12');
CALL `student_clinic_db_final`.`Payment_insert`('109','SELF','PENDING','2017-05-17');
CALL `student_clinic_db_final`.`Payment_insert`('110','INSURANCE','PENDING','2017-08-01');
```

Student insurance insert:

```
DELIMITER $$
CREATE PROCEDURE `Studentinsurance_insert`(IN STUD_ID varchar(7), IN INSURNC_ID
varchar(90))
BEGIN
INSERT INTO student_insurance(student_id,insurance_id)
values(STUD_ID,INSURNC_ID);
END$$
DELIMITER ;

CALL `student_clinic_db_final`.`Studentinsurance_insert`('10','2');
```



```
CALL `student_clinic_db_final`.`Studentinsurance_insert`('55','2');
```

To insert billpayment:

```
DELIMITER $$
CREATE PROCEDURE `BillPayment_insert`(IN BILL_ID varchar(11),
IN TRANS_NUM varchar(11))
BEGIN
INSERT INTO bill_payment(bill_no,pay_trans_num)
values(BILL_ID,TRANS_NUM);
END$$
DELIMITER ;
```

```
CALL `student_clinic_db_final`.`BillPayment_insert`('1','106');
CALL `student_clinic_db_final`.`BillPayment_insert`('2','107');
CALL `student_clinic_db_final`.`BillPayment_insert`('3','108');
CALL `student_clinic_db_final`.`BillPayment_insert`('4','109');
```

Update Procedures:

Given student id and bill amount to be updated:

```
DELIMITER $$
CREATE PROCEDURE `Update_BillAmount`(IN STUD_ID varchar(12),
IN AMOUNT_UPDATE decimal(9,6)
)
BEGIN
UPDATE bill set amount=AMOUNT_UPDATE where bill_no
in(SELECT bill_no from student_bill where student_id=STUD_ID);
END$$
DELIMITER ;
```

```
CALL `student_clinic_db_final`.`Update_BillAmount`('1','555');
```

Given student id and update the payment status:

```
DELIMITER $$
CREATE PROCEDURE `Update_PaymentStatus`(IN STUD_ID varchar(12),
IN STATUS_UPDATE varchar(45)
)
BEGIN
UPDATE payment set pay_status=STATUS_UPDATE where pay_trans_num
in(SELECT pay_trans_num from bill_payment where bill_no
```

```
in(SELECT bill_no from student_bill where student_id=STUD_ID));  
END$$  
DELIMITER ;
```

```
CALL `student_clinic_db_final`.`Update_PaymentStatus`('1','Paid');
```

Updating insurance company id for a given student id:

```
DELIMITER $$  
CREATE PROCEDURE `Update_InsuranceCompany`(IN STUD_ID varchar(12),  
IN Company_UPDATE varchar(7))  
BEGIN  
UPDATE student_insurance set insurance_id=Company_UPDATE  
where student_id=STUD_ID;  
END$$  
DELIMITER ;
```

```
CALL `student_clinic_db_final`.`Update_InsuranceCompany`('1','50');
```

Delete Procedure

To delete student record

```
DELIMITER $$  
CREATE PROCEDURE `Del_Student`  
(IN DEL_STUDENT_ID varchar(12))  
BEGIN  
delete from student where student_id= DEL_STUDENT_ID;  
END$$  
DELIMITER ;
```

```
CALL `student_clinic_db_final`.`Del_Student`('3');
```

To delete diagnosis record

```
DELIMITER $$  
CREATE PROCEDURE `DEL_diagnosis_data`  
(IN Del_DIAGNOSIS_ID varchar(12))  
BEGIN  
DELETE FROM diagnosis WHERE diagnosis_id = Del_DIAGNOSIS_ID;  
END$$  
DELIMITER ;
```

```
CALL `student_clinic_db_final`.`DEL_diagnosis_data`('903');
```

To delete doctor data

```
SELECT * FROM student_clinic_db_final.doctor;
DELIMITER $$
CREATE PROCEDURE `Delete_Doctor`
(IN Del_Doc varchar(12))
BEGIN
DELETE FROM doctor WHERE doctor_id = Del_Doc;
END$$
DELIMITER ;

CALL `student_clinic_db_final`.`Delete_Doctor`('202');
```

To delete insurance company data

```
DELIMITER $$
CREATE PROCEDURE `Del_InsuranceCompany`
(IN DEL_COMPANY_ID varchar(12))
BEGIN
DELETE FROM insurance where insurance_id = DEL_COMPANY_ID;
END$$
DELIMITER ;

CALL `student_clinic_db_final`.`Del_InsuranceCompany`('5');
```

To delete medicine record

```
DELIMITER $$
CREATE PROCEDURE `Delete_medicine`(IN Del_Med varchar(12))
BEGIN
DELETE FROM medicine WHERE med_inventory_id = Del_Med;
END$$
DELIMITER ;

CALL `student_clinic_db_final`.`Delete_medicine`('321');
```

APPLICATION PROGRAMMING INTERFACE – STORED PROCEDURES

Create an application programming interface (API) using stored procedures that allows an EMR system developer to work with the database without having to write SQL statements

Ans: To demonstrate the API we have created stored procedures for following scenario's.

1. Select all the students having insurance

DELIMITER \$\$

```
CREATE PROCEDURE `student_having_insurance`()
```

```
BEGIN
```

```
select s.student_name, i.insurance_company
```

```
from insurance i
```

```
join student_insurance using(insurance_id)
```

```
join student s using (student_id);
```

```
END$$
```

```
DELIMITER;
```

```
CALL `student_clinic_db_final`.`student_having_insurance`();
```

	student_name	insurance_company
►	Suyash Nande	General Insurance
	Rahul	AIG
	Neha	Aurora Boralis
	John	Life Insure
	Abhishek	ABC

2. Get all the visit granted by Dr. Rahul

DELIMITER \$\$

```
CREATE PROCEDURE `ALL_VISIT_FOR_DR_RAHUL`(IN DOC_NAME_INPUT  
varchar(255))
```

```
BEGIN
```

```
select visit.visit_id, visit.visit_date, doctor.doctor_id
```

```
from visit
```

```
join doctor using(doctor_id)
```

```
join diagnosis using(diagnosis_id)
```

```
join prescription using(prescription_id)
```

```
where doctor.doc_name=DOC_NAME_INPUT;
```

END\$\$

DELIMITER ;

set @DOC_NAME_INPUT='Dr. Rahul';

CALL `student_clinic_db_final`.`ALL_VISIT_FOR_DR_RAHUL`(@DOC_NAME_INPUT);

	visit_id	visit_date	doctor_id
▶	1	2017-04-10	101
	2	2017-04-11	101

3. Get the bill details and payment number of Suyash Nande

DELIMITER \$\$

CREATE PROCEDURE `BILL_DETAILS_SUYASH`(IN STU_NAME varchar(255))

BEGIN

select student.student_name, student.student_id, bill.bill_no, bill.bill_date,
payment.pay_trans_num, payment.pay_date
from bill

join bill_payment using (bill_no)

join payment using (pay_trans_num)

join student_bill using (bill_no)

join student using (student_id)

where student.student_name=STU_NAME;

END\$\$

DELIMITER ;

set @STU_NAME='Suyash Nande';

CALL `student_clinic_db_final`.`BILL_DETAILS_SUYASH`(@STU_NAME);

	student_name	student_id	bill_no	bill_date	pay_trans_num	pay_date
▶	Suyash Nande	1	5	2016-10-12	101	2017-02-19

4. Get the names and contact of all doctors who have visited at least 1 student

DELIMITER \$\$

CREATE PROCEDURE `DOC_VISIT_HISTORY`(IN NO_OF_VISITS int)

BEGIN

select doctor.doc_name, doctor.doc_contact_no

from doctor

where doctor_id in (select doctor_id

```

from visit
join visit_history using (visit_id)
group by visit.doctor_id
having max(visit_history.student_id)>=NO_OF_VISITS);
END$$
DELIMITER ;

set @NO_OF_VISITS=2;
CALL `student_clinic_db_final`.`DOC_VISIT_HISTORY`(@NO_OF_VISITS);

```

	doc_name	doc_contact_no
►	Dr. Bomma	+19234768753
	Dr. Singh	+18603768753
	Dr. Vikas	+19803234753

5. Find the names of the students and their address who have visited to more than one doctor.

```

DELIMITER $$
CREATE PROCEDURE `STU_VISIT_HISTORY`
(IN NO_OF_VISITS int)
BEGIN
select s.student_name, s.address, count(v.doctor_id)
from student s, visit v, visit_history vh, doctor d
where v.visit_id = vh.visit_id
and vh.student_id = s.student_id
and v.doctor_id = d.doctor_id
group by s.student_name,s.address
having count(v.doctor_id) > NO_OF_VISITS;
END$$
DELIMITER ;

```

```

set @NO_OF_VISITS = 1;
CALL `student_clinic_db_final`.`STU_VISIT_HISTORY`(@NO_OF_VISITS);

```

	student_name	address	count(v.doctor_id)
►	Suyash Nande	UT Drive, Charlotte, NC	2

6. Find the name of the doctors with their contact numbers whom has been visited by only

student.

```
DELIMITER $$
CREATE PROCEDURE `ONE_STUDENT_DOC`(IN NO_OF_VISIT int)
BEGIN
select d.doc_name, count(v.doctor_id)
from doctor d, visit v
where d.doctor_id = v.doctor_id
group by doc_name,doc_contact_no
having count(v.doctor_id)=NO_OF_VISIT;
END$$
DELIMITER ;

SET @NO_OF_VISIT=1;
CALL `student_clinic_db_final`.`ONE_STUDENT_DOC`(@NO_OF_VISIT);
```

	doc_name	count(v.doctor_id)
►	Dr. Bomma	1
	Dr. Singh	1
	Dr. Vikas	1

7. Find names, id's, record id and vaccine category for students who have been given vaccination of FLU and order by student name

```
DELIMITER $$
CREATE PROCEDURE `STU_WITH_VACC`(IN VACC_NAME varchar(255))
BEGIN
select student.student_name, student.student_id, i.record_id ,i.vaccine_category
from student
join immunization_record i using(student_id)
where i.vaccine_category = VACC_NAME
order by student_name;
END$$
DELIMITER ;

SET @VACC_NAME='FLU';
CALL `student_clinic_db_final`.`STU_WITH_VACC`(@VACC_NAME);
```

	student_name	student_id	record_id	vaccine_category
▶	Abhishek	10	103	Flu
	Suyash Nande	1	101	Flu

8. Find number of visits for any student

```

DELIMITER $$
CREATE PROCEDURE `VISITS_OF_ANY_STUDENT`(IN STU_NAME
varchar(255))
BEGIN
select student.student_name, COUNT(visit_history.visit_id) AS NumofVisits
from student
join visit_history using(student_id)
where student_name = STU_NAME;
END$$
DELIMITER ;

```

```

SET @STU_NAME='Suyash Nande';
CALL `student_clinic_db_final`.`VISITS_OF_ANY_STUDENT`(@STU_NAME);

```

	student_name	NumofVisits
▶	Suyash Nande	2

9. Find the number of Visits on a particular day

```

DELIMITER $$
CREATE PROCEDURE `No_Of_Visit_On_Any_Date`(IN VISIT_DATE date, out
NUM_OF_VISITS int)
BEGIN
select COUNT(visit_id) AS NumofVisits into NUM_OF_VISITS
from visit where visit_date = VISIT_DATE;
END$$
DELIMITER ;

```

```

SET @VISIT_DATE = '2017-03-15';
CALL `student_clinic_db_final`.`No_Of_Visit_On_Any_Date`(@VISIT_DATE,
@NUM_OF_VISITS);
SELECT @NUM_OF_VISITS;

```


@NUM_OF_VISITS
▶ 5

10. Find the student name and payment status whose payment status is pending. Order by name.

```
DELIMITER $$
CREATE PROCEDURE `Payment_Status_Of_Student`(IN STATUS varchar(255))
BEGIN
select student.student_name , student.student_id, payment.pay_status
from student
join student_bill using (student_id)
join bill_payment using (bill_no)
join payment using ( pay_trans_num)
where pay_status = STATUS
order by student.student_name;
END$$
DELIMITER ;
```

```
SET @STATUS='Pending';
CALL `student_clinic_db_final`.`Payment_Status_Of_Student`(@STATUS);
```

	student_name	student_id	pay_status
▶	John	5	Pending
	Neha	4	Pending
	Rahul	3	Pending

11. Find out the department name from any complaint from the student

```
DROP PROCEDURE IF EXISTS dptname_forcomplaint;
DELIMITER $$
CREATE PROCEDURE `dptname_forcomplaint`(IN complaint varchar(55))
begin
select d.doc_name,d.doctor_id, a.dpt_name,a.dpt_id
from doctor d, affiliated_with a,visit v
where d.doctor_id = a.doctor_id
and v.doctor_id = a.doctor_id
and v.complaint= complaint;
END$$
```

DELIMITER ;

SET @Complaint='Hypertension';

CALL `student_clinic_db_final_final`.`dptname_forcomplaint`(@Complaint);

	doc_name	doctor_id	dpt_name	dpt_id
►	Dr. Bomma	103	Physiotherapy	9003

12. Find out details of the doctor for any department

Drop procedure if exists `docname_fordept`;

Delimiter \$\$

CREATE procedure `docname_fordept` (IN department_name varchar(55))

begin

select d.doc_name, d.doctor_id, a.dpt_name

from doctor d, affiliated_with a

where d.doctor_id = a.doctor_id

and a.dpt_name like department_name;

END; \$\$

DELIMITER ;

set @department_name = 'General Medicine';

CALL `student_clinic_db_final_final`.`docname_fordept`(@department_name);

	doc_name	doctor_id	dpt_name
►	Dr. Ryan	101	General Medicine
	Dr. Khan	106	General Medicine
	Dr. Preeti	107	General Medicine

Add tables and other features to enable audit trail so that every query or change of every record in the database is monitored and the entire history of the data in the database is captured. Basically, every time a record is accessed (queried, inserted, or changed), the user and time of access is recorded. Every time any field of a record is updated or deleted, the previous value of the record is saved.

Ans: We have created audit trial for following tables:

- visit table - Update, Delete and Insert operation.
- payment table - Update, Delete and Insert operation.

Audit Trial for Visit table

▪ **Trigger for before update operation**

```
DROP TRIGGER IF EXISTS student_clinic_db_final.audit_visit_bu;
DELIMITER $$
create trigger audit_visit_bu BEFORE UPDATE on Visit
for each row
begin
insert into audit_trial_visit SELECT 'BEFORE UPDATE', NULL, NOW(),
current_user(), V.* from visit as V
where V.visit_id = old.visit_id;
end;$$
DELIMITER ;
```

▪ **Trigger for after update operation**

```
DROP TRIGGER IF EXISTS student_clinic_db_final.audit_visit_au;
DELIMITER $$
create trigger audit_visit_au AFTER UPDATE on Visit
for each row
begin
insert into audit_trial_visit SELECT 'AFTER UPDATE', NULL, NOW(), current_user(),
V.* from visit as V
where V.visit_id = new.visit_id;
end;$$
DELIMITER ;
```

▪ **Trigger for insert operation**

```
DROP TRIGGER IF EXISTS student_clinic_db_final.visit_after_insert
DELIMITER $$
create trigger visit_after_insert AFTER INSERT on Visit
for each row
Begin
insert into audit_trial_visit SELECT 'INSERT', NULL, NOW(), current_user(), V.* from
visit as V
where V.visit_id = new.visit_id;
End; $$
DELIMITER ;
```

▪ Trigger for delete operation

```

DROP TRIGGER IF EXISTS student_clinic_db_final.visit_before_delete
DELIMITER $$
create trigger visit_before_delete BEFORE DELETE on Visit
for each row
Begin
insert into audit_trial_visit SELECT 'DELETE', NULL, NOW(), current_user(), V.* from
visit as V
where V.visit_id = old.visit_id;
End; $$
DELIMITER ;

```

The screenshot shows a database management tool interface. The top pane displays several SQL queries: an update statement for 'Chest Pain New', an insert statement for a new visit, a delete statement, and a select statement from the audit_trial_visit table. The bottom pane shows a 'Result Grid' with 11 columns: action, revision, dt_datetime, user_name, visit_id, visit_date, time, complaint, diagnosis_id, doctor_id, and prescription_... The grid contains 10 rows of data representing audit trail entries for various database actions.

action	revision	dt_datetime	user_name	visit_id	visit_date	time	complaint	diagnosis_id	doctor_id	prescription_...
BEFORE UPDATE	5	2017-04-15 18:50:30	root@localhost	1	2017-03-10	09:10:00	ABC	11	101	12
AFTER UPDATE	6	2017-04-15 18:50:30	root@localhost	1	2017-03-10	09:10:00	ABC1	11	101	12
BEFORE UPDATE	7	2017-04-15 18:52:34	root@localhost	2	2017-03-11	10:00:00	Tharki	12	102	23
AFTER UPDATE	8	2017-04-15 18:52:34	root@localhost	2	2017-03-11	10:00:00	ABC1	12	102	23
INSERT	9	2017-04-15 19:58:54	root@localhost	6	2017-04-11	14:10:00	test	14	104	55
DELETE	10	2017-04-15 19:59:34	root@localhost	6	2017-04-11	14:10:00	test	14	104	55

Audit Trial for Payment table

▪ Trigger for before update operation

```

DROP TRIGGER IF EXISTS student_clinic_db_final.audit_payment_bu;
DELIMITER $$
create trigger audit_payment_bu BEFORE UPDATE on payment
for each row
begin
insert into audit_trial_payment SELECT 'BEFORE UPDATE', NULL, NOW(), current_user(),
P.* from payment as P
where P.pay_trans_num = old.pay_trans_num;
end;$$

```

```
DELIMITER ;
```

- **Trigger for after update operation**

```
DROP TRIGGER IF EXISTS student_clinic_db_final.audit_payment_au;
DELIMITER $$
create trigger audit_payment_au AFTER UPDATE on payment
for each row
begin
insert into audit_trial_payment SELECT 'AFTER UPDATE', NULL, NOW(), current_user(),
P.* from payment as P
where P.pay_trans_num = new.pay_trans_num;
end;$$
DELIMITER ;
```

- **Trigger for insert operation**

```
DROP TRIGGER IF EXISTS student_clinic_db_final.audit_payment_after_insert;
DELIMITER $$
create trigger audit_payment_after_insert AFTER INSERT on payment
for each row
Begin
insert into audit_trial_payment SELECT 'INSERT', NULL, NOW(), current_user(), P.* from
payment as P
where P.pay_trans_num = new.pay_trans_num;
End; $$
DELIMITER ;
```

- **Trigger for delete operation**

```
DROP TRIGGER IF EXISTS student_clinic_db_final.audit_payment_after_delete;
DELIMITER $$
create trigger audit_payment_after_delete BEFORE DELETE on payment
for each row
Begin
insert into audit_trial_payment SELECT 'DELETE', NULL, NOW(), current_user(), P.* from
payment as P
```

```

where P.pay_trans_num = old.pay_trans_num;
End; $$
DELIMITER ;

```

action	revision	pay_datetime	username	pay_trans_num	pay_method	pay_status	pay_date
BEFORE UPDATE	15	2017-05-07 17:58:58	root@localhost	103	insurance	Paid	2017-01-01
AFTER UPDATE	16	2017-05-07 17:58:58	root@localhost	103	insurance	Pending	2017-01-01
INSERT	17	2017-05-07 18:09:51	root@localhost	106	Self	Pending	2017-03-17
DELETE	19	2017-05-07 18:12:24	root@localhost	103	insurance	Pending	2017-01-01

Add user authentication so that only authorized users can access the database.

Ans: The primary aim of the access control mechanism in MySQL is to authenticate a user based on his username, host and password and to associate the user with privileges on different databases and tables such as Select, Update, Insert, etc.

Following are the salient features:

- Users are assigned to certain roles which are based on their job assignment and access privileges.
- These relationships are many-to-many i.e. the user may have several roles and each role can be assigned to several users.
- Similarly, an action may be assigned to several roles while a role may have several actions.
- It provides administrator with the capability to place constraints on role authorization, role activation and operation execution.

user_role table

```

CREATE TABLE `user_role` (
  `user` varchar(45) NOT NULL,
  `password` varchar(45) NOT NULL,
  `role` varchar(45) NOT NULL,
  `host` varchar(45) NOT NULL,
  PRIMARY KEY (`user`,`role`,`host`)
);

```

Procedure for User Authentication

```

DROP PROCEDURE IF EXISTS user_auth;
DELIMITER |
CREATE PROCEDURE user_auth
(IN name CHAR(32), IN pass CHAR(64), OUT role CHAR(36))
BEGIN
SELECT user_role.user, user_role.password, user_role.role INTO @name, @pass, @role
FROM user_role WHERE user_role.user = name;
IF (SELECT COUNT(user_role.user) FROM user_role WHERE user_role.user = name AND
user_role.password = pass)!=1 THEN
SET @message_text = CONCAT ('Login Incorrect for user \'',@name, '\');
SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = @message_text;
ELSE
SIGNAL SQLSTATE '01000' SET MESSAGE_TEXT = 'successfully authenticated';
SELECT @role as role INTO role;
END IF;
END;|
DELIMITER ;

SET @name='suyash';
SET @pass='suyash';
CALL `student_clinic_db`.`user_auth`(@name, @pass, @role);
SELECT @role;

```

```

3 DROP PROCEDURE IF EXISTS user_auth;
4 DELIMITER |
5 CREATE PROCEDURE user_auth(IN name CHAR(32), IN pass CHAR(64), OUT role CHAR(36))
6 BEGIN
7     SELECT user_role.user, user_role.password, user_role.role INTO @name, @pass, @role FROM user_role WHERE user_role.user = name;
8     IF (SELECT COUNT(user_role.user) FROM user_role WHERE user_role.user = name AND user_role.password = pass)!=1 THEN
9         SET @message_text = CONCAT ('Login Incorrect for user \'',@name, '\');
10        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = @message_text;
11    ELSE
12        SIGNAL SQLSTATE '01000' SET MESSAGE_TEXT = 'successfully authenticated';
13        SELECT @role as role INTO role;
14    END IF;
15 END;|
16 DELIMITER ;
17
18
19 SET @name='suyash';
20 SET @pass='suyash';
21 CALL `student_clinic_db`.`user_auth`(@name, @pass, @role);
22 SELECT @role;
23

```

100% 14:22

Result Grid

@role
admin

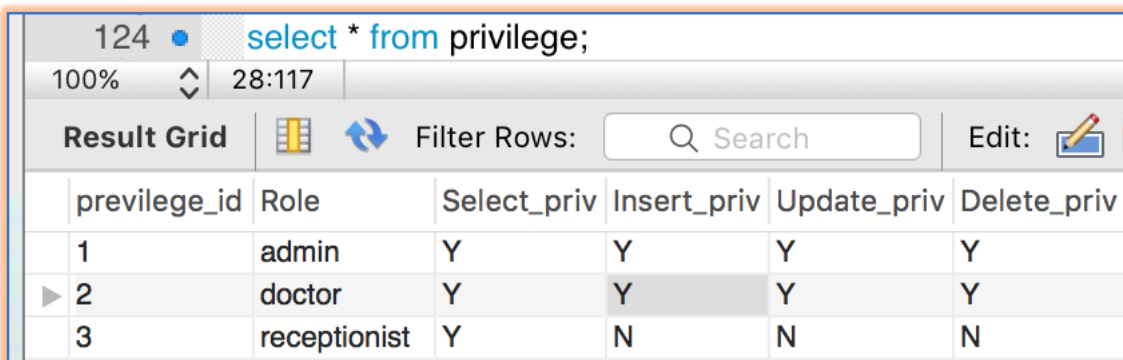
Filter Rows: Search Export

User Authentication Implementation

For every procedure call we ensured only authorized user can execute the procedure. Henceforth, we have created role based privilege table.

privilege table – all the privileges are stored with privilege_id and role as as Primary Key

```
CREATE TABLE `privilege` (
  `privilege_id` int(11) NOT NULL,
  `Role` varchar(45) NOT NULL,
  `Select_priv` enum('N','Y') NOT NULL,
  `Insert_priv` enum('N','Y') NOT NULL,
  `Update_priv` enum('N','Y') NOT NULL,
  `Delete_priv` enum('N','Y') NOT NULL,
  PRIMARY KEY (`privilege_id`,`Role`)
);
```



privilege_id	Role	Select_priv	Insert_priv	Update_priv	Delete_priv
1	admin	Y	Y	Y	Y
2	doctor	Y	Y	Y	Y
3	receptionist	Y	N	N	N

Role based Scenario

When a student consults doctor, then doctor can create, delete and update his prescription. Here only doctor should have privilege to create, delete and update the prescription of the student. Other role such as receptionist can only view the prescription.

Prescription created by the doctor

```
DROP PROCEDURE IF EXISTS create_prescription;
DELIMITER $$
CREATE PROCEDURE `create_prescription`
(IN username varchar(255), IN pass varchar(255), IN presc_id int, IN med_name varchar(255),
IN med_quantity int)
```



```

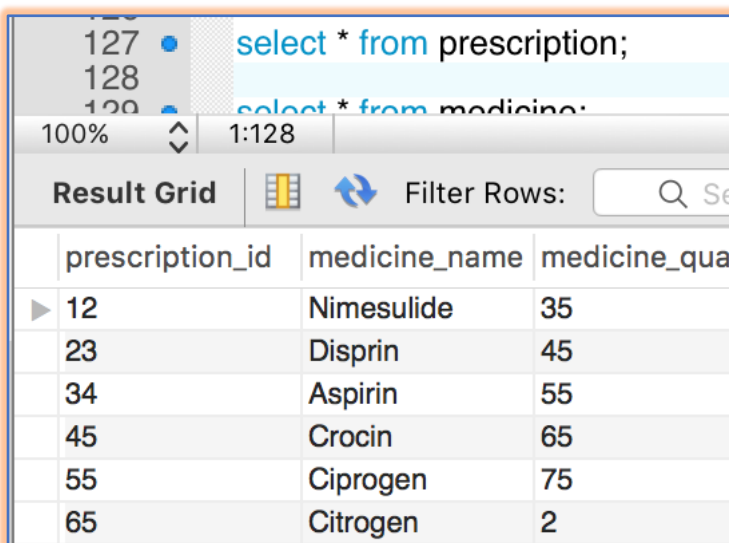
BEGIN
CALL user_auth(username, pass, @role);
IF(@role like 'doctor')
THEN
IF ((select Insert_priv from privilege where Role like 'doctor') like 'Y')
THEN
insert into prescription values(presc_id, med_name, med_quantity);
ELSE
SET @message_text = CONCAT(username, ' donot have insert privilege');
SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = @message_text;
END IF;
ELSE
SET @message_text = CONCAT(username, ' with role ', @role, ' is not authorized to create the
prescription');
SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = @message_text;
END IF;
END$$
DELIMITER ;

```

```

SET @username = 'abhishek';
SET @pass = 'abhishek';
SET @presc_id = 65;
SET @med_name = 'Citrogen';
SET @med_quantity = 2;
CALL `student_clinic_db_final`.`create_prescription`(@username, @pass, @presc_id,
@med_name, @med_quantity);

```



	prescription_id	medicine_name	medicine_qua
▶	12	Nimesulide	35
	23	Disprin	45
	34	Aspirin	55
	45	Crocin	65
	55	Ciprogen	75
	65	Citrogen	2

Now receptionist cannot create a prescription

87

88

89

90

91

92

93

SET @username = 'harika';

SET @pass = 'harika';

SET @presc_id = 66;

SET @med_name = 'Citrogen';

SET @med_quantity = 2;

CALL `student_clinic_db`.`create_prescription` (@username, @pass, @presc_id, @med_name, @med_quantity);

100%

26:87

Action Output

	Time	Action	Response
✓ 1	19:03:00	SET @username = 'harika'	0 row(s) affected
✓ 2	19:03:00	SET @pass = 'harika'	0 row(s) affected
✓ 3	19:03:00	SET @presc_id = 66	0 row(s) affected
✓ 4	19:03:00	SET @med_name = 'Citrogen'	0 row(s) affected
✓ 5	19:03:00	SET @med_quantity = 2	0 row(s) affected
✗ 6	19:03:00	CALL `student_clinic_db`.`create_prescription` (@user...	Error Code: 1644. harika with role receptionist is not authorized to create the prescription

Prescription deleted by the doctor

```

DROP PROCEDURE IF EXISTS delete_prescription;
DELIMITER $$
CREATE PROCEDURE `delete_prescription`
(IN username varchar(255), IN pass varchar(255), IN presc_id int)
BEGIN
CALL user_auth(username, pass, @role);
IF(@role like 'doctor')
THEN
IF ((select Delete_priv from privilege where Role like 'doctor') like 'Y')
THEN
delete from prescription where prescription_id=presc_id;
ELSE
SET @message_text = CONCAT(username, ' donot have delete privilege');
SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = @message_text;
END IF;
ELSE
SET @message_text = CONCAT(username, ' with role ', @role, ' is not authorized to delete the
prescription');
SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = @message_text;
END IF;
END$$
DELIMITER ;

```

117

•

SET @username = 'abhishek';

118

•

SET @pass = 'abhishek';

119

•

SET @presc_id = 65;

120

•

CALL `student_clinic_db`.`delete_prescription`(@username, @pass, @presc_id);

121

122

100%

⬆

8:112

Action Output

⬆

		Time	Action	Response
✓	1	19:07:37	SET @username = 'abhishek'	0 row(s) affected
✓	2	19:07:37	SET @pass = 'abhishek'	0 row(s) affected
✓	3	19:07:37	SET @presc_id = 65	0 row(s) affected
✓	4	19:07:37	CALL `student_clinic_db`.`delete_prescription`(@username, @pass, @...	0 row(s) affected

Now receptionist cannot delete a prescription

117

•

SET @username = 'harika';

118

•

SET @pass = 'harika';

119

•

SET @presc_id = 65;

120

•

CALL `student_clinic_db`.`delete_prescription`(@username, @pass, @presc_id);

121

•

100%

↕

20:119

Action Output

↕

	Time	Action	Response
✓ 1	19:08:33	SET @username = 'harika'	0 row(s) affected
✓ 2	19:08:33	SET @pass = 'harika'	0 row(s) affected
✓ 3	19:08:33	SET @presc_id = 65	0 row(s) affected
✗ 4	19:08:33	CALL `student_clinic_db`.`delete_prescription`(@user...	Error Code: 1644, harika with role receptionist is not authorized to delete the prescription

- Add indexes and views so that frequently used queries and changes to the database are most efficient.

Ans: **Views** are stored queries that when invoked produce a result set. A **view** acts as a virtual table.

We have created views for following scenarios.

Scenario I

When we need to find out the students with pending bills.

Solution: To find out the students who didn't paid the full amount i.e. whose bills are pending; we need to join **five tables** Student, Bill, Payment, bill_payment and student_bill. Henceforth we created a view wherein we joined all these five tables.

create or replace view bill_pending_view as
 select student.student_id, student.student_name, student_bill.bill_no, bill.amount,
 bill.amount_paid, payment.pay_trans_num, bill.due_date

```

from bill
join bill_payment using (bill_no)
join payment using (pay_trans_num)
join student_bill using (bill_no)
join student using (student_id)
where (bill.amount_paid < bill.amount);

```

The screenshot shows a database query editor with a SQL query in the top pane and a 'Result Grid' in the bottom pane. The query is: `select * from bill_pending_view;`. The result grid displays two rows of data with columns: student_id, student_name, bill_no, amount, amount_paid, pay_trans_num, and due_date.

student_id	student_name	bill_no	amount	amount_paid	pay_trans_num	due_date
3	Rahul	6	413.400000	213.400000	102	2016-12-12
4	Neha	10	123.100000	100.100000	106	2017-02-04

Now we'll use the above created view to calculate the remaining amount to be paid by the student with his due date.

```

select bv.student_id, bv.student_name, bv.bill_no, (bv.amount-bv.amount_paid) as
amount_remaining, bv.due_date
from bill_pending_view bv
join payment p using(pay_trans_num)
where bv.student_name = 'Rahul';

```

The screenshot shows a database query editor with a SQL query in the top pane and a 'Result Grid' in the bottom pane. The query is: `select bv.student_id, bv.student_name, bv.bill_no, (bv.amount-bv.amount_paid) as amount_remaining, bv.due_date from bill_pending_view bv join payment p using(pay_trans_num) where bv.student_name = 'Rahul';`. The result grid displays one row of data with columns: student_id, student_name, bill_no, amount_remaining, and due_date.

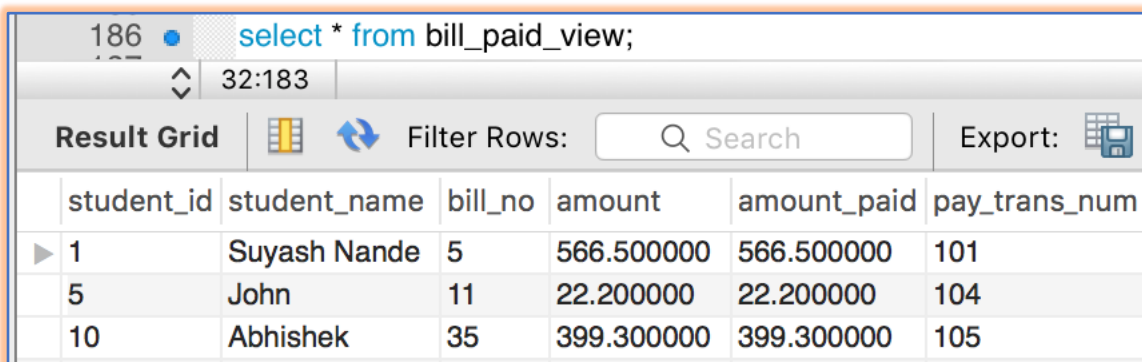
student_id	student_name	bill_no	amount_remaining	due_date
3	Rahul	6	200.000000	2016-12-12

Scenario II

When we need to find out the students with paid bills.

Solution: To find out the students who already paid the full amount i.e. whose bills are paid; we need to join **five tables** Student, Bill, Payment, bill_payment and student_bill. Henceforth we created a view wherein we joined all these five tables.

```
create or replace view bill_paid_view as
select student.student_id, student.student_name, student_bill.bill_no, bill.amount,
bill.amount_paid, payment.pay_trans_num
from bill
join bill_payment using (bill_no)
join payment using (pay_trans_num)
join student_bill using (bill_no)
join student using (student_id)
where (bill.amount_paid = bill.amount);
```



student_id	student_name	bill_no	amount	amount_paid	pay_trans_num
1	Suyash Nande	5	566.500000	566.500000	101
5	John	11	22.200000	22.200000	104
10	Abhishek	35	399.300000	399.300000	105

Scenario III

When we need to find out total number of medicines prescribed to a student 'Suyash Nande'

Solution: To find total number of medicines prescribed to a student we need to join **five tables** Student, Visit, visit_history, Doctor and Prescription. Henceforth we created a view which consists of results by joining all these tables for any student.

```
create or replace view Medicine_For_Student as
select s.student_id,s.student_name,v.visit_id,d.doc_name, p.medicine_name AS
Prescribed_Medicine
from student s
join visit_history using(student_id)
```



```

join visit v using (visit_id)
join doctor d using(doctor_id)
join prescription p using(prescription_id)
where s.student_name = 'Suyash Nande';

```

200 ● select * from Medicine_For_Student;
201




36:200

Result Grid   Filter Rows: Expo

	student_id	student_name	visit_id	doc_name	Prescribed_Medici...
▶	1	Suyash Nande	1	Dr. Rahul	Nimesulide
	1	Suyash Nande	3	Dr. Rahul	Aspirin

Now we'll use above created view to calculate the total number of medicines prescribed for that student.

202 ● select student_id, student_name, count(*) as Total_Num_Medicines from Medicine_For_Student;
1:199

Result Grid   Filter Rows: Export: 

	student_id	student_name	Total_Num_Medicines
▶	1	Suyash Nande	2

Scenario IV

When we need to find out maximum visit for any department.

Solution: To find maximum number of visit for any department we need to join two tables affiliated_with and visit and the result set should contain information about all the department with their visit count.



```

create or replace view dept_with_max_visit as
select a.dpt_id, a.dpt_name, count(a.dpt_name) as max_visit
from affiliated_with a
join visit v using(doctor_id)
group by a.dpt_id, a.dpt_name
HAVING max(v.doctor_id);

```

217 • `select * from dept_with_max_visit;`

1:208

Result Grid   Filter Rows:




dpt_id	dpt_name	max_visit
9001	General Medicine	2
9002	Psychiatry	1
9004	Orthopedics	1
9005	Cardiology	1

Now among these we need to select the department with the maximum visit count.

```
select dpt_id, dpt_name, max(max_visit) from dept_with_max_visit;
```

219 • `select dpt_id, dpt_name, max(max_visit) from dept_with_max_visit;`

1:218

Result Grid   Filter Rows: Export: 

dpt_id	dpt_name	max(max_visit)
9001	General Medicine	2

Indexes

Index is a data structure that improves the speed of search operation in a table. It can be created using one or more columns. Indexes are also type of tables, which keep primary key or index field and a pointer to each record into the actual table.

Users cannot see the indexes, they are just to speed up queries and will be used by database search engine to locate records very fast.

We have created indexes for every tables based on primary key and foreign key.

Scenario I: Index for visit table

225 • show index from visit;

100% 23:225

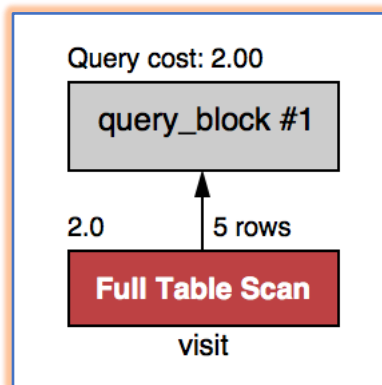
Result Grid Filter Rows: Search Export:

	Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
▶	visit	0	PRIMARY	1	visit_id	A	5	NULL	NULL		BTREE
	visit	1	diagnosis_id	1	diagnosis_id	A	5	NULL	NULL	YES	BTREE
	visit	1	doctor_id	1	doctor_id	A	4	NULL	NULL	YES	BTREE
	visit	1	prescription_id	1	prescription_id	A	5	NULL	NULL	YES	BTREE
	visit	1	visit_ibfk_4_idx	1	appointment_id	A	1	NULL	NULL	YES	BTREE

In the above visit table, **visit_id** is the **primary key** and **diagnosis_id**, **doctor_id**, **prescription_id** and **appointment_id** are the **foreign keys**. Hence we have created indexes for all the keys.

Find the visit details of a student based on non-index field visit_date.

Ans: select * from visit where visit_date='2017-03-16';

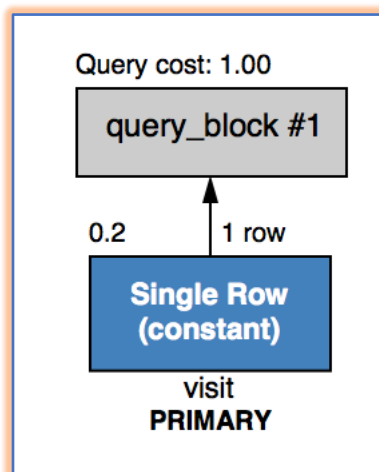


<pre>234 • select * from visit where visit_date='2017-03-16';</pre>	
100% 1:233	
Query Statistics	
Timing (as measured at client side): Execution time: 0:00:0.00040317	Joins per Type: Full table scans (Select_scan): 1 Joins using table scans (Select_full_join): 0 Joins using range search (Select_full_range_join): 0 Joins with range checks (Select_range_check): 0 Joins using range (Select_range): 0
Timing (as measured by the server): Execution time: 0:00:0.00024748 Table lock wait time: 0:00:0.00011400	Sorting: Sorted rows (Sort_rows): 0 Sort merge passes (Sort_merge_passes): 0 Sorts with ranges (Sort_range): 0 Sorts with table scans (Sort_scan): 0
Errors: Had Errors: NO Warnings: 0	Index Usage: No Index used
Rows Processed: Rows affected: 0 Rows sent to client: 1 Rows examined: 5	

In the above query, full table is scanned as no index were used. Therefore, the **query cost is 2.0**. And hence execution time is **0.00040317 seconds**.

Now on using the index field *visit_id* we will perform the same operation.

Ans: select * from visit where visit_id=5;



228 • select * from visit where visit_id=5;	
100% 38:228	
Query Statistics	
Timing (as measured at client side): Execution time: 0:00:0.00031805	Joins per Type: Full table scans (Select_scan): 0 Joins using table scans (Select_full_join): 0 Joins using range search (Select_full_range_join): 0 Joins with range checks (Select_range_check): 0 Joins using range (Select_range): 0
Timing (as measured by the server): Execution time: 0:00:0.00019451 Table lock wait time: 0:00:0.00009200	Sorting: Sorted rows (Sort_rows): 0 Sort merge passes (Sort_merge_passes): 0 Sorts with ranges (Sort_range): 0 Sorts with table scans (Sort_scan): 0
Errors: Had Errors: NO Warnings: 0	Index Usage: At least one Index was used
Rows Processed: Rows affected: 0 Rows sent to client: 1 Rows examined: 1	

In the above query index was used, **query cost is 1.0** and time taken to execute the query was **0.00031805 seconds**.

Hence, by using the index the same result set takes **0.00008512 seconds** less time to execute so it is faster to search via an index field.

Experience

- Creating audit trial was not an easy task, as it involves triggers that run before and after each query execution for that table, so we need to carefully examine the tuple data before inserting and updating it.
- Implementing user authentication and role based access was a challenging task. We have implemented the same into the stored procedures so that every time we call these procedures it will first check the authorization of the user and if the user is authorized then it will again check for that user privilege to execute the SQL statements written in the stored procedures.
- Merging the relations was made easy since foreign keys existed that referenced the various tables that were merged.
- The SQL statement became complex and more error-prone as the number of relations to be merged increased.