National University of Singapore

CS2102 Project Report Team 22

Ong Wei Sheng A0206042X

Damien Lim Yu Hao A0223892Y

Hiong Kai Han A0199814M

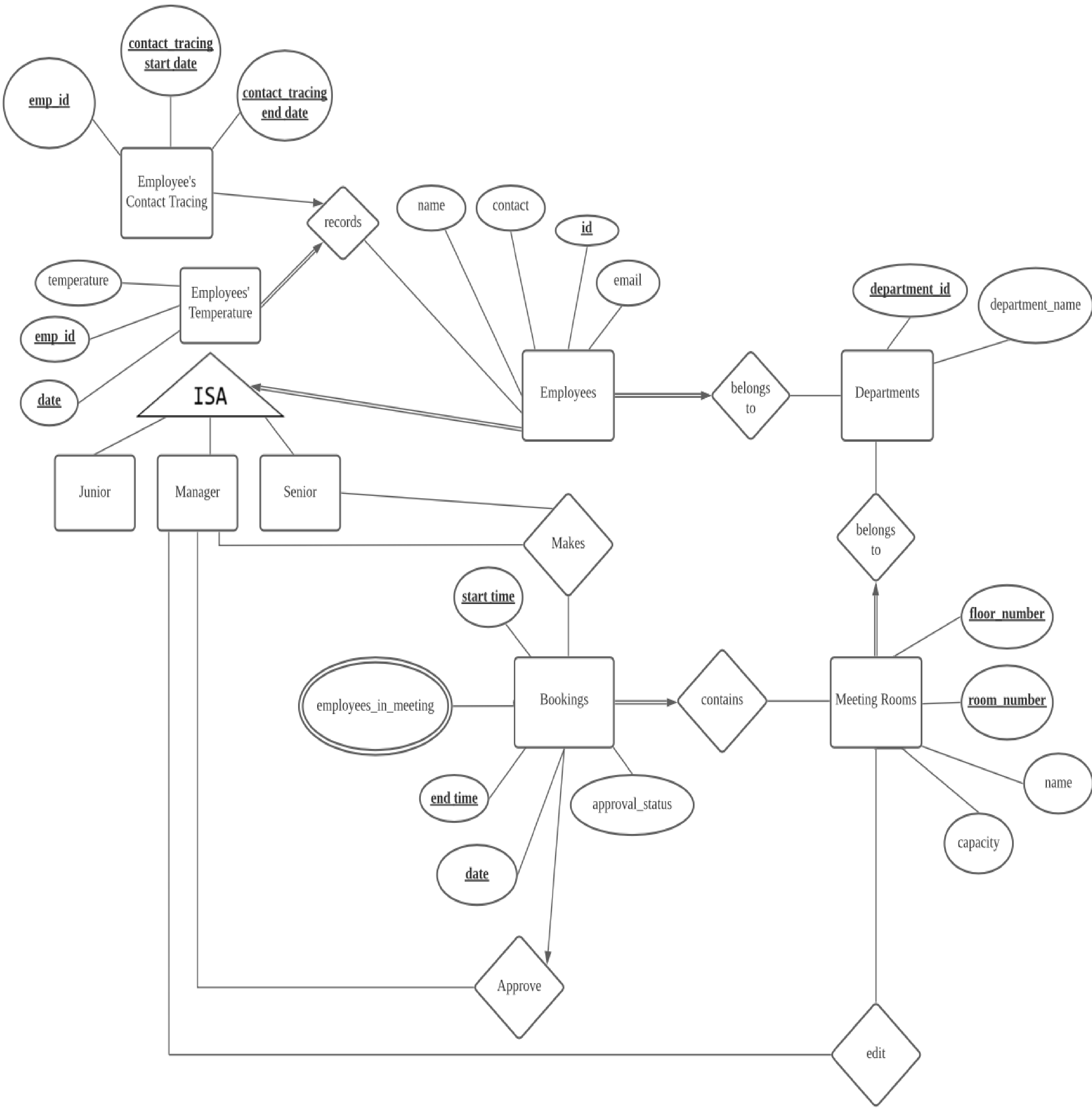
Tan Le Jun A0199755E

1.0. Project Responsibilities

Member	Responsibilities
Ong Wei Sheng	Functionalities - *remove_department, search_room*, *unbook_room*, *join_meeting*, *leave_meeting*, *view_future_meeting*, *view_manager_report* Schema - *Sessions* Projet Report - Section 2, 3, 4.2, 6
Damien Lim Yu Hao	Functionalities - *add_employee*, *remove_employee*, *book_room*, *approve_meeting*, *view_booking_report* ER Diagram (edited and adopted from recommended one) Schema - *healthDeclaration*, *Declare*, *Employees* (*Junior*/*Booker*/*Senior*/*Manager*), *Departments* Project Report - Section 4.1, 5, 6
Hiong Kai Han	Functionalities - *add_room*, *add_department*, *change_capacity*, *declare_health*, *contact_tracing* and *non_compliance* Schema - *From*, *Books*, *Joins*, *Approves* Project Report - Section 3.3, 4.3, 5, 6
Tan Le Jun	Schema - *locatedIn*, *meetingRooms*, *Updates*

2.0. ER Data Model

2.1. Initial ER data model



2.1.1 Justification for any non-trivial design decisions made for the initial ER data model

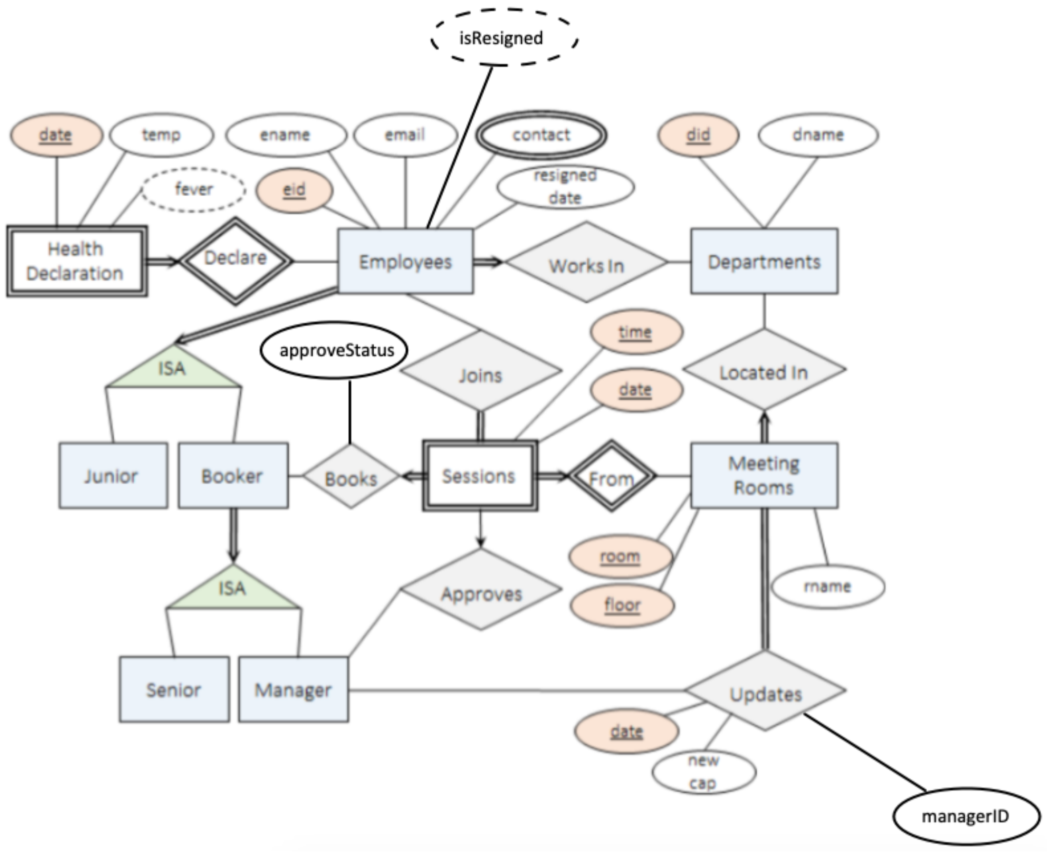
1. *Employees Contact Tracing* must participate **at most** once in *Records* (contact trace of one emp_id cannot be mapped to two different employees), but it's not mandatory to participate also (if no one is in close contact between *Employees*, there is no need to record in *Employees Contact Tracing*)
2. *Employee's Temperature* participates **exactly once** in *Records.* Although an employee can choose not to record their temperature, *Employee's Temperature* must still participate and record this infraction down (as a NULL value under temperature attribute).
3. *Bookings* must participate **at most once** with an Approve relationship. This is because it is not mandatory for a Booking to be approved; a Manager can choose not to approve/disapprove, causing the Booking to be pending. It also cannot be approved/disapproved by two different Managers.
4. *Employee's Temperature* and *Employee's Contact Tracing* are separate but closely related entities. *Employee's Temperature* simply keeps a list of each employee's daily temperature taking (and who failed to do so). *Employee's Contact Tracing* is only used to keep track of who was in contact with a feverish employee. This is also why they form a ternary relationship with *Employees*.
   We have chosen to seperate them for two reasons.
   Firstly, we felt that it would be easier to establish whether a Booking was to be allowed (check if the employee doing the booking has no fever under *Employee's Temperature*, then check that none of the employees are under close contact under *Employee's Contact Tracing*).
   Secondly, if *Employee Contact Tracing* was included in *Employee's Temperature*, the contact tracing status of the employee has to be computed everyday, which is inefficient.
5. Every *Meeting Room* belongs to **exactly one** *Department*. This follows from a Manager only being able to approve a *Meeting Room* if they are both from the same *Departmen*t; which suggests the relationship between *Meeting Room* and *Department.*
6. Employees have a **covering, non-overlap ISA Hierarchy** with Junior, Senior, Manager. We chose this to best reflect the condition that an Employee can only be one of the three positions.

2.1.2 Application’s constraints that are not captured by the initial ER data model

1. If an employee has a fever, they cannot book a meeting room and all meetings that he/she books are cancelled. If an employee with a fever or is in close contact with an employee with a fever (in the same approved meeting in the past 3 days with the employee with a fever), they will be removed and can’t be added to future meetings in the next 7 days.
2. An approved booking cannot add more participants into the booking.
3. An employee that resigned cannot book and approve meetings.
4. Managers not from the same department as the meeting room cannot approve the booking and change the capacity of the meeting room.
5. Only the employee that made the booking can add and remove participants or unbook the meeting room.

2.2. Actual ER data model used

After receiving feedback about our initial ER data model, there were flaws and limitations identified that may not be easily fixed. Therefore, we decided to adapt the proposed ER Data model from the module instead.



2.2.1 Justification for any non-trivial design decisions made in ER data model

The following design decisions are referenced from the provided document for the proposed ER data model.

1. For employee contact numbers, the multi-valued attributes for contact only allows up to three contact numbers which are home phone number, mobile phone number & office phone number.
2. Health declaration is made into a weak entity set dependent on employees because each health declaration must be made by an employee. This allows the date to be a partial key. A more fine-grained partial key of date and time is unnecessary as we can simply replace the old value. Additionally, fever is a derived attribute because we can set it automatically from temperature recorded.
3. The meeting room has no attribute (derived or not) for capacity. An update provides the date as a partial key which can be used to distinguish the actual capacity at any given time. To ensure that a meeting room has a capacity, total participation constraints between meeting rooms and updates is added. However, this design complicates implementation due to a search required (via SQL queries) for the current applicable capacity.
4. A session is a weak entity set dependent on the meeting room to ensure that each meeting room can only have exactly one session with a session having a partial key of date and time. Here we have the assumption that time is the start time with each session lasting only one hour. To ensure that a session must be created at booking time, a key and total participation constraints to books is added. Furthermore, to ensure that only a single approval is made, a key participation constraint to approve is added. Lastly, to ensure that an employee booking the room immediately joins the booked meeting, a total participation constraints between employees and joins is added.
5. The two-level ISA hierarchy is used to first separate an employee that is allowed to book (i.e., a booker) and an employee that is not allowed to book (i.e., a junior). Secondly, we then separate a manager from a senior since only a manager can approve a meeting. Additionally, the chosen design allows for a possible distinct responsibilities between senior employees and managers in the future
6. Updates include a *managerID* field now. When first creating a room, a Manager has to be the one that creates the room, and sets the initial capacity. Hence, this managerID and initial capacity will be captured in Updates.
7. Books has an additional *approveStatus* field for Bookings. This is an integer representation of three-valued logic, 0 = notApproved, 1 = disApproved, 2 = Approved. However since the project does not involve the Manager explicitly disapproving any Bookings, 1 is unused.
8. Employees has an additional *isResigned* field for their employment status. This is a derived attribute that can be inferred from the resignation date. Any resignation date that is ‘1001-01-01’ will mean that *isResigned* is false; any date other than ‘1001-01-01’ will mean *isResigned* is true.

2.2.2 Constraints not captured by the ER data model

1. If an employee is having a fever, they cannot join a booked meeting.
2. Once a meeting is approved, there should be no more changes in the participants and the participants will definitely come to the meeting on the stipulated day.
3. When an employee resigns, they are no longer allowed to book or approve any meetings.
4. Employee recorded to have fever at D day removed from all future meeting room booking regardless.
5. All employees in the same meeting room from D-3 to D as the employee with fever will be removed from future meetings in the next 7 days (D to D7).

3.0. Relational database schema

```sql
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

CREATE TABLE Employees (
   eid BIGSERIAL,
   ename TEXT NOT NULL,
   mobilePhoneContact NUMERIC(8) NOT NULL UNIQUE,
   homePhoneContact NUMERIC(8),
   officePhoneContact NUMERIC(8),
   email TEXT NOT NULL UNIQUE,
   resignedDate DATE DEFAULT '1001-01-01',
   isResigned BOOLEAN DEFAULT FALSE,
   PRIMARY KEY (eid)
);

CREATE TABLE Junior (
   juniorID INTEGER,
   PRIMARY KEY (juniorID),
   FOREIGN KEY (juniorID) REFERENCES Employees (eid) ON DELETE CASCADE
);

CREATE TABLE Booker (
   bookerID INTEGER,
   PRIMARY KEY (bookerID),
   FOREIGN KEY (bookerID) REFERENCES Employees (eid) ON DELETE CASCADE
);
```

```
CREATE TABLE Senior (
   seniorID INTEGER,
   PRIMARY KEY (seniorID),
   FOREIGN KEY (seniorID) REFERENCES Booker (bookerID) ON DELETE CASCADE
);

CREATE TABLE Manager (
   managerID INTEGER,
   PRIMARY KEY (managerID),
   FOREIGN KEY (managerID) REFERENCES Booker (bookerID) ON DELETE CASCADE
);

CREATE TABLE healthDeclaration (
   date DATE NOT NULL,
   temp NUMERIC(3,1) NOT NULL CHECK (34.0 <= temp AND temp <= 43.0),
   fever BOOLEAN,
   eid INTEGER,
   PRIMARY KEY (date, eid),
   FOREIGN KEY (eid) REFERENCES Employees (eid) ON DELETE CASCADE
);

CREATE TABLE Departments (
   did INTEGER,
   dname TEXT NOT NULL,
   PRIMARY KEY (did)
);

CREATE TABLE worksIn (
   eid INTEGER,
   did INTEGER,
   PRIMARY KEY (eid),
   FOREIGN KEY (eid) REFERENCES Employees (eid),
   FOREIGN KEY (did) REFERENCES Departments (did)
);

CREATE TABLE meetingRooms (
   room INTEGER,
   floor INTEGER,
   rname TEXT NOT NULL,
   PRIMARY KEY (room, floor)
);
```

```sql
CREATE TABLE locatedIn (
   room INTEGER,
   floor INTEGER,
   did INTEGER NOT NULL,
   PRIMARY KEY (room, floor),
   FOREIGN KEY (room, floor) REFERENCES meetingRooms (room, floor) ON DELETE CASCADE,
   FOREIGN KEY (did) REFERENCES Departments (did)
);

CREATE TABLE Updates (
   managerID INTEGER,
   date DATE DEFAULT '1001-01-01',
   newCap INTEGER NOT NULL CHECK (newCap >= 0),
   room INTEGER,
   floor INTEGER,
   PRIMARY KEY (date, room, floor),
   FOREIGN KEY (room, floor) REFERENCES meetingRooms (room, floor) ON DELETE CASCADE,
   FOREIGN KEY (managerID) REFERENCES Manager (managerID)
);

CREATE TABLE Sessions (
   room INTEGER,
   floor INTEGER,
   date DATE,
   time INTEGER CHECK (time>=0 AND time<24),
   PRIMARY KEY (room, floor, date, time),
   FOREIGN KEY (room, floor) REFERENCES meetingRooms (room,floor) ON DELETE CASCADE
);

CREATE TABLE Approves (
   managerID INTEGER,
   room INTEGER,
   floor INTEGER,
   date DATE,
   time INTEGER CHECK (time>=0 AND time<24),
   PRIMARY KEY (room, floor, date, time),
   FOREIGN KEY (managerID) REFERENCES Manager (managerID),
   FOREIGN KEY (room, floor, date, time) REFERENCES Sessions (room, floor, date, time)
ON DELETE CASCADE
);

CREATE TABLE Books (
   bookerID INTEGER NOT NULL,
   room INTEGER,
   floor INTEGER,
   date DATE,
   time INTEGER CHECK (time>=0 AND time<24),
   approveStatus INTEGER DEFAULT 0 CHECK (approveStatus >= 0 AND approveStatus <= 2),
   /* 0 -> pending approval, 1 -> disapproved, 2 -> approved */
   PRIMARY KEY (room, floor, date, time),
   FOREIGN KEY (bookerID) REFERENCES Booker (bookerID) ON DELETE CASCADE,
   FOREIGN KEY (room, floor, date, time) REFERENCES Sessions (room, floor, date, time)
ON DELETE CASCADE
);

CREATE TABLE Joins (
   eid INTEGER NOT NULL,
   room INTEGER,
   floor INTEGER,
   date DATE,
   time INTEGER CHECK (time>=0 AND time<24),
   PRIMARY KEY (eid, room, floor, date, time),
   FOREIGN KEY (eid) REFERENCES Employees (eid) ON DELETE CASCADE,
   FOREIGN KEY (room, floor, date, time) REFERENCES Sessions (room, floor, date, time)
ON DELETE CASCADE
);
```

3.1. Justification of non-trivial design in Schema

1. Employees can have mobilePhone, homePhone, and officePhone contact numbers; but only mobilePhone number is compulsory. We assumed that employees could choose not to include their homePhone numbers for privacy concerns, and not every employee would have an officePhone number.
2. **BIGSERIAL** data type is used for Employee's eid, as it allows the system to auto generate the eid.
3. Employees' contact information (*mobilePhoneContact*, *homePhoneContact*, *officePhoneContact*) must be 8 digits exactly.This is because Singaporean mobile numbers all have exactly 8 digits.
4. In the *Employees* and *Updates* table, the *isResigned* and *date* fields have a **DEFAULT** value of '1001-01-01' to represent something that 'has not happened yet'. For example, an Employee with a resignation date of '1001-01-01' means that he has not resigned yet. We could have chosen to use **NULL** values to represent this instead; however handling **NULL** values is often tricky and easy to make mistakes on.

5. A booked room can have multiple states of Approval. The booking could not be processed yet, explicitly disapproved, or explicitly approved. Hence, there is an additional *approveStatus* field within the *Books* table to capture this. Upon insertion into the Books table, the approveStatus will be set to a default of 0 until the meeting is approved where it will be updated to 2.
6. *Approves*, *Books*, *Joins* all **REFERENCES** *Sessions*, with an **ON DELETE CASCADE**. This is because they cannot exist without an accompanying entry in *Sessions*. For example, you cannot have an approved session that does not exist in *Sessions* in the first place.
7. All tables that have time attributes are of type integer to allow for simpler classification. The allowed values for time are also from 0 to 23 to simulate valid timings in a day.
8. In the *Updates* table, the *newCap* attribute which represents the capacity of the meeting room has to be > 0.
9. In the *healthDeclaration* table, the *temperature* attribute has to be 34.0 and above and 43.0 and below.
10. Deletion from the employee field will cascade down to either *Junior*, *Senior*, *Booker*, *Manager* to ensure that the employee will also be deleted in those tables.
11. Deletion from *locatedIn* and *Updates* table will be cascaded even though there is no requirement to do so because it will allow inserting data through data.sql easier as we would only need to delete from the meetingRooms table only.

3.2. Application constraints not captured by Schema

1. If an employee is having a fever, they cannot book a room.
2. The employee booking the room immediately joins the booked meeting.
3. If an employee is having a fever, they cannot join a booked meeting.
4. A manager can only approve a booked meeting in the same department as the manager.
5. Once approved, no more changes to participants and all participants will attend the meeting.

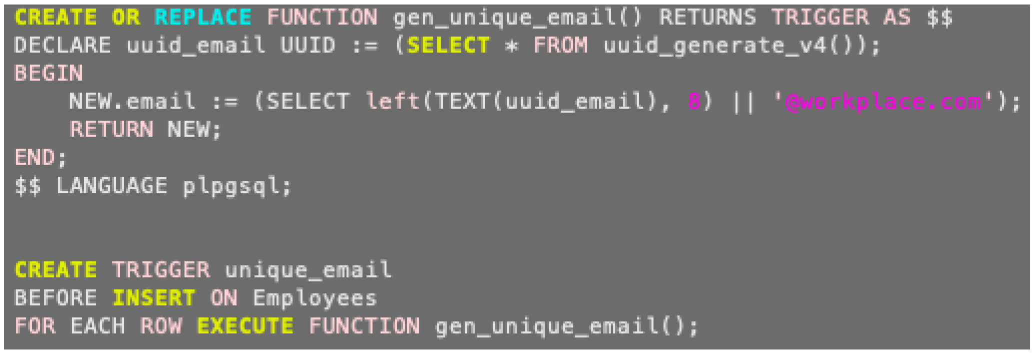
3.3. Assumptions made

1. Employees who are removed from Bookings due to Close Contact, will not join any Bookings during their 7 day ban.
2. When checking if an Employee has a fever, we will take reference from the latest health declaration made.
3. An employee that resigns will be removed from an Approved Meeting.
4. For each day, a Manager can only change the capacity for a room once.
5. Only a Manager from the same department can add a room, approve a booking.
6. When adding a room, a Manager has to be the one that creates the room. The Manager will also be the one that sets the initial capacity of the room.

7. When adding a room, the department that the room is located in is assumed to be the same department as the manager adding the room. The capacity of the room will be added in the Updates table using CURRENT_DATE as the date.
8. There is nothing stopping an employee removed from meetings due to close contact to be added back into meetings.
9. Every employee must do a daily health declaration.
10. Bookings can only be made in 1hr blocks.
11. Only managers from the same department as the meeting room can approve a meeting in that meeting room.
12. Any booking made is assumed to be in the future.
13. Any meeting that employee is trying to join is assumed to be in the future.
14. Any manager approving a booking, the booking is assumed to be in the future already.

4.0. Description of interesting Triggers

4.1. Trigger 1 (*unique_email*)



```
CREATE OR REPLACE FUNCTION gen_unique_email() RETURNS TRIGGER AS $$
DECLARE uuid_email UUID := (SELECT * FROM uuid_generate_v4());
BEGIN
    NEW.email := (SELECT left(TEXT(uuid_email), 8) || '@workplace.com');
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;


CREATE TRIGGER unique_email
BEFORE INSERT ON Employees
FOR EACH ROW EXECUTE FUNCTION gen_unique_email();
```

Usage of Trigger

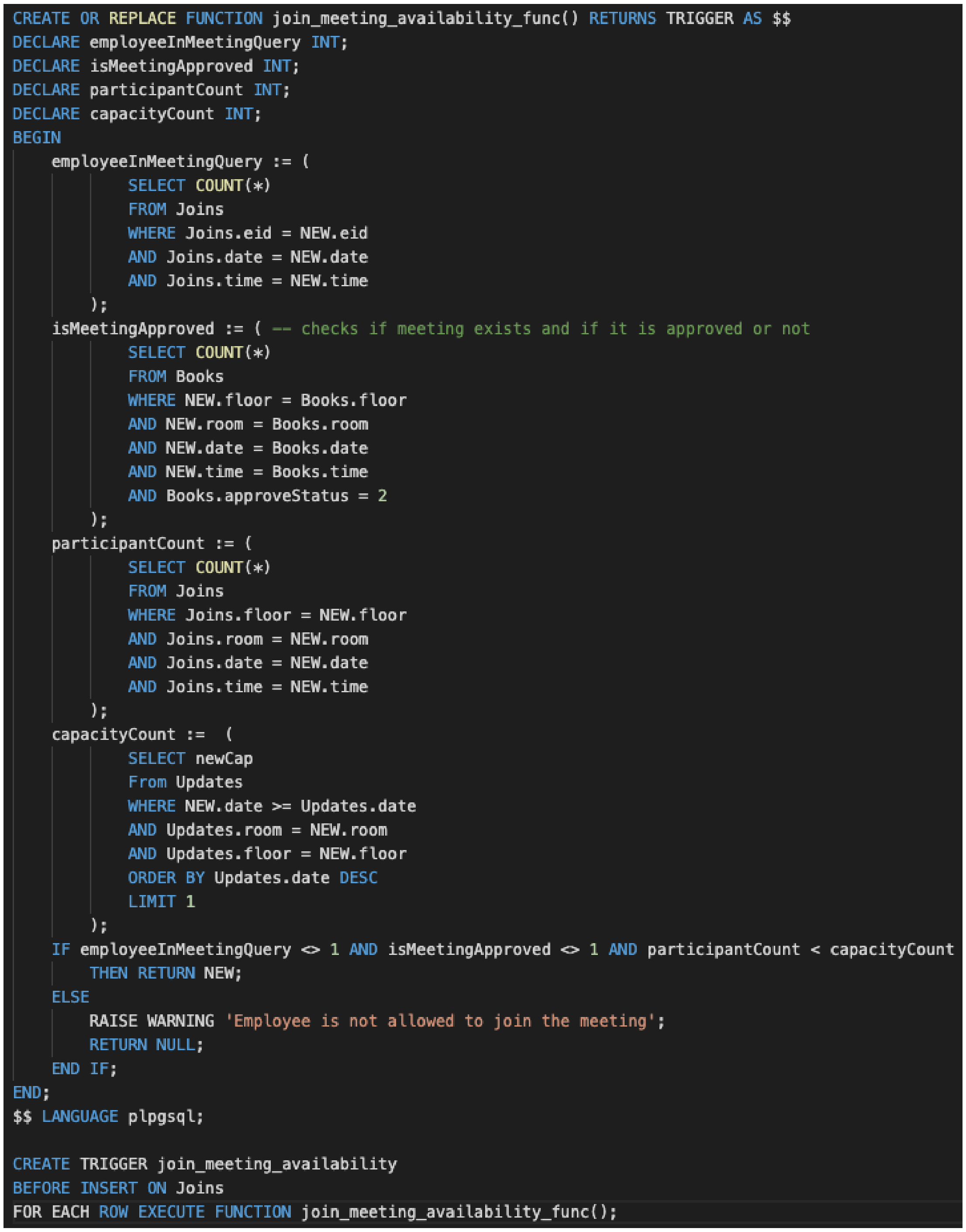
This trigger is used to automatically generate a unique email address for each Employee. It fires before insertion into *Employees*, generating the email and then inserting it into the *Employees* table, all 'within' a single INSERT INTO call.

Justification of Design

The crux of this trigger lies in *uuid_generate_v4()* function. This function generates a **UUID** value (sequence of 32 hexadecimal digits) solely based on random numbers. For all practical purposes, this **UUID** is guaranteed to be unique.

This **UUID** is used to create a unique employee email. We take the leftmost 8 digits from the **UUID**, and cast it to **TEXT** data format. This allows us to concatenate it with a constant '*@workplace.com*' string, which provides us with a unique email address. Although shortening it from 32 to 8 digits does reduce its 'uniqueness' factor somewhat; for all practical purposes it is still unique. We then assign the result above to NEW.email and RETURN NEW, which allows the INSERTION to complete successfully.

4.2. Trigger 2 (join_meeting_availability)



```
CREATE OR REPLACE FUNCTION join_meeting_availability_func() RETURNS TRIGGER AS $$
DECLARE employeeInMeetingQuery INT;
DECLARE isMeetingApproved INT;
DECLARE participantCount INT;
DECLARE capacityCount INT;
BEGIN
    employeeInMeetingQuery := (
            SELECT COUNT(*)
            FROM Joins
            WHERE Joins.eid = NEW.eid
            AND Joins.date = NEW.date
            AND Joins.time = NEW.time
        );
    isMeetingApproved := ( -- checks if meeting exists and if it is approved or not
            SELECT COUNT(*)
            FROM Books
            WHERE NEW.floor = Books.floor
            AND NEW.room = Books.room
            AND NEW.date = Books.date
            AND NEW.time = Books.time
            AND Books.approveStatus = 2
        );
    participantCount := (
            SELECT COUNT(*)
            FROM Joins
            WHERE Joins.floor = NEW.floor
            AND Joins.room = NEW.room
            AND Joins.date = NEW.date
            AND Joins.time = NEW.time
        );
    capacityCount :=  (
            SELECT newCap
            From Updates
            WHERE NEW.date >= Updates.date
            AND Updates.room = NEW.room
            AND Updates.floor = NEW.floor
            ORDER BY Updates.date DESC
            LIMIT 1
        );
    IF employeeInMeetingQuery <> 1 AND isMeetingApproved <> 1 AND participantCount < capacityCount
        THEN RETURN NEW;
    ELSE
        RAISE WARNING 'Employee is not allowed to join the meeting';
        RETURN NULL;
    END IF;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER join_meeting_availability
BEFORE INSERT ON Joins
FOR EACH ROW EXECUTE FUNCTION join_meeting_availability_func();
```

Usage of Trigger

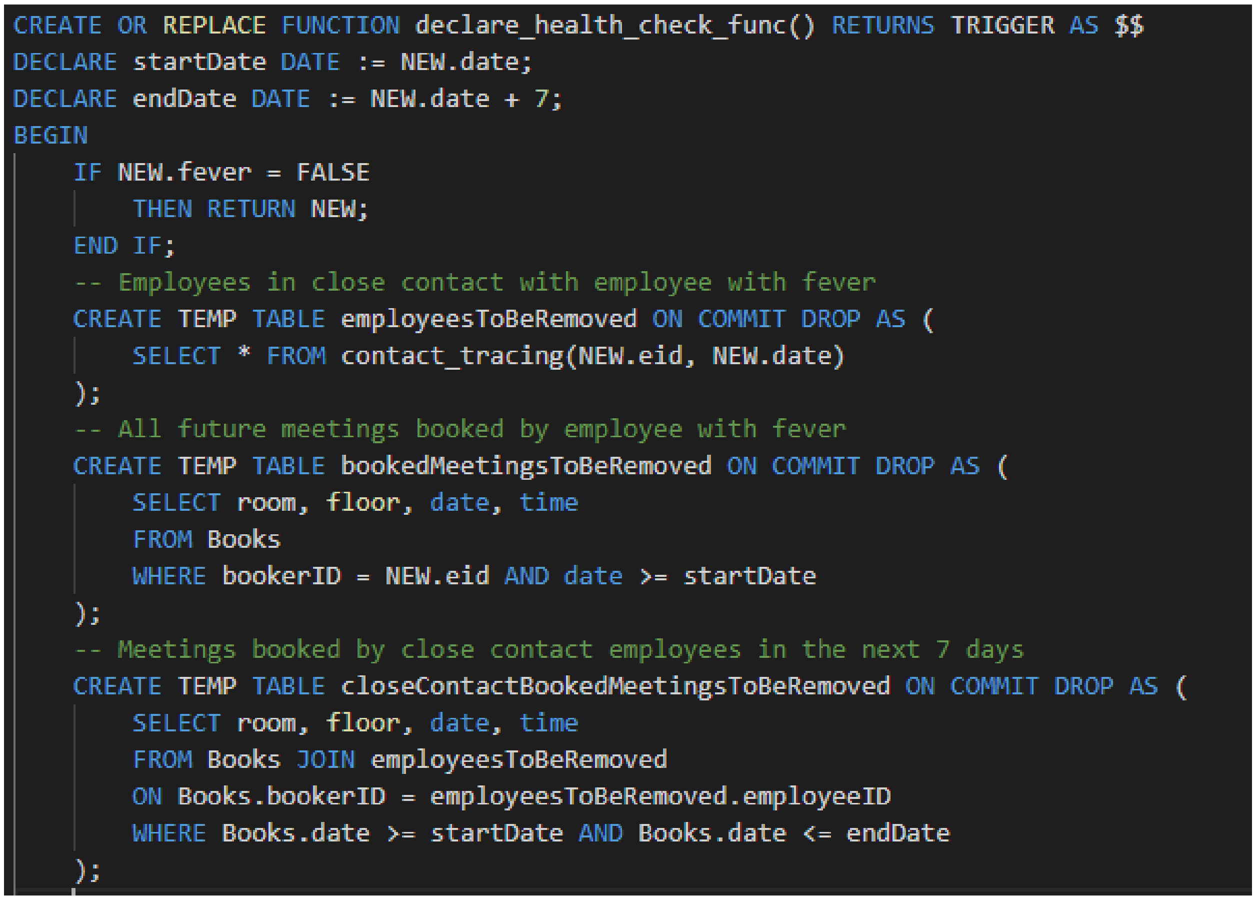
This trigger is used to ensure that when executing the join_meeting function, the employee attempting to join the room is able to join that particular meeting.

Justification of design

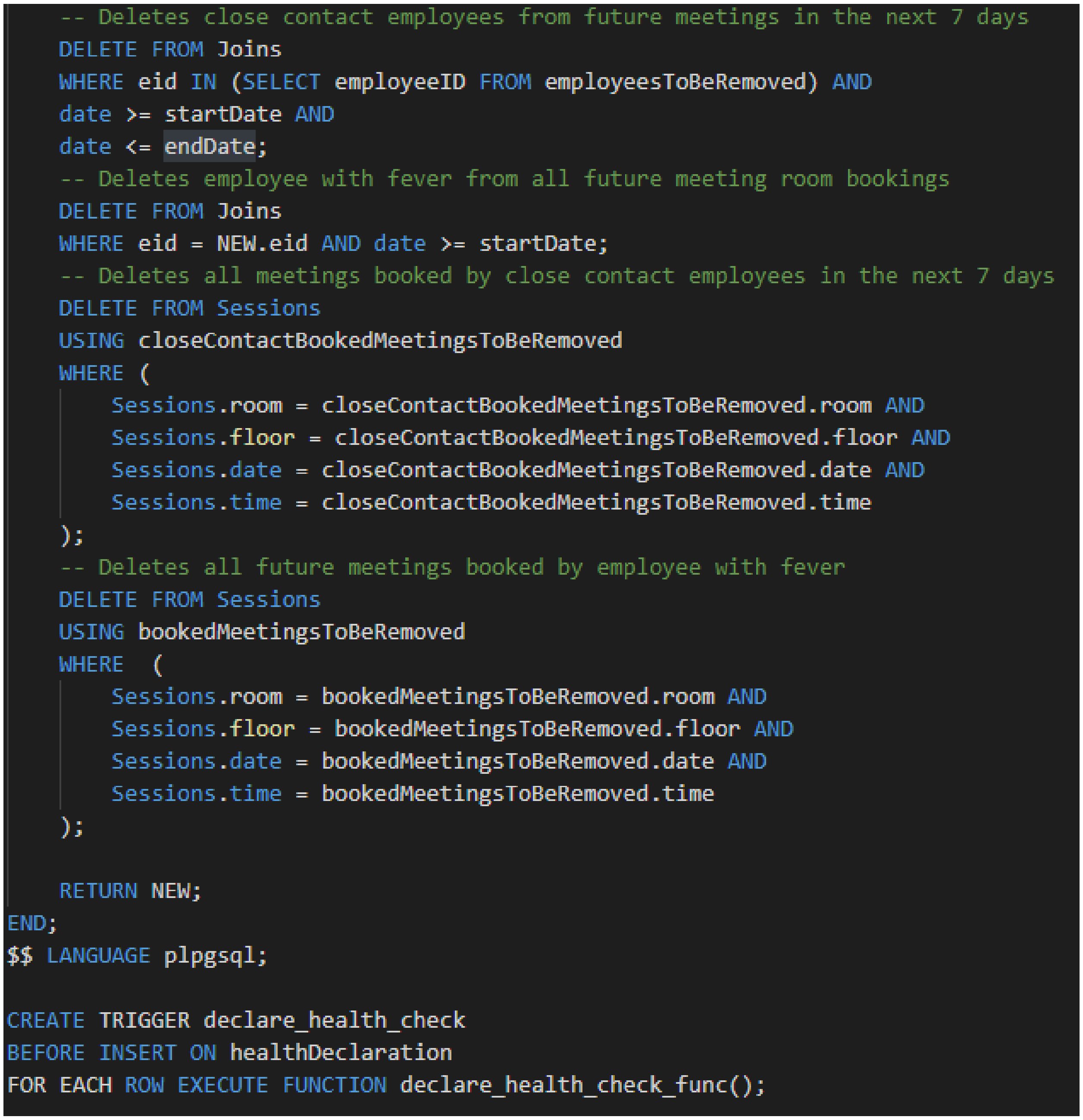
There are many constraints when it comes to whether an employee is able to join a particular meeting. Some of the constraints are addressed in the join_meeting function itself such as checking whether the employee has fever or whether the employee has resigned. However, some of the constraints when it comes to joining a meeting has to be checked using triggers to guard against manual insertion into our Joins table. Therefore, in this trigger we checked for the following constraints.

- Whether the employee is already in an upcoming meeting at the same date and time.
- Whether the meeting that the employee is trying to join exists.
- Whether the meeting that the employee is attempting to join is approved.
- Whether the meeting that the employee is attempting to join still has capacity to join.

4.3. Trigger 3 (declare_health_check)



```
CREATE OR REPLACE FUNCTION declare_health_check_func() RETURNS TRIGGER AS $$
DECLARE startDate DATE := NEW.date;
DECLARE endDate DATE := NEW.date + 7;
BEGIN
    IF NEW.fever = FALSE
        THEN RETURN NEW;
    END IF;
    -- Employees in close contact with employee with fever
    CREATE TEMP TABLE employeesToBeRemoved ON COMMIT DROP AS (
        SELECT * FROM contact_tracing(NEW.eid, NEW.date)
    );
    -- All future meetings booked by employee with fever
    CREATE TEMP TABLE bookedMeetingsToBeRemoved ON COMMIT DROP AS (
        SELECT room, floor, date, time
        FROM Books
        WHERE bookerID = NEW.eid AND date >= startDate
    );
    -- Meetings booked by close contact employees in the next 7 days
    CREATE TEMP TABLE closeContactBookedMeetingsToBeRemoved ON COMMIT DROP AS (
        SELECT room, floor, date, time
        FROM Books JOIN employeesToBeRemoved
        ON Books.bookerID = employeesToBeRemoved.employeeID
        WHERE Books.date >= startDate AND Books.date <= endDate
    );
```



```
    -- Deletes close contact employees from future meetings in the next 7 days
    DELETE FROM Joins
    WHERE eid IN (SELECT employeeID FROM employeesToBeRemoved) AND
    date >= startDate AND
    date <= endDate;
    -- Deletes employee with fever from all future meeting room bookings
    DELETE FROM Joins
    WHERE eid = NEW.eid AND date >= startDate;
    -- Deletes all meetings booked by close contact employees in the next 7 days
    DELETE FROM Sessions
    USING closeContactBookedMeetingsToBeRemoved
    WHERE (
        Sessions.room = closeContactBookedMeetingsToBeRemoved.room AND
        Sessions.floor = closeContactBookedMeetingsToBeRemoved.floor AND
        Sessions.date = closeContactBookedMeetingsToBeRemoved.date AND
        Sessions.time = closeContactBookedMeetingsToBeRemoved.time
    );
    -- Deletes all future meetings booked by employee with fever
    DELETE FROM Sessions
    USING bookedMeetingsToBeRemoved
    WHERE  (
        Sessions.room = bookedMeetingsToBeRemoved.room AND
        Sessions.floor = bookedMeetingsToBeRemoved.floor AND
        Sessions.date = bookedMeetingsToBeRemoved.date AND
        Sessions.time = bookedMeetingsToBeRemoved.time
    );

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER declare_health_check
BEFORE INSERT ON healthDeclaration
FOR EACH ROW EXECUTE FUNCTION declare_health_check_func();
```

Usage of Trigger

The role of the trigger is to conduct contact tracing procedures when an employee declares fever. If an employee declares fever, the trigger will proceed to remove the employee and close contacts from future meetings.

Justification of design

The trigger checks if the employee doing their daily health declaration has a fever. If the employee is recorded to have a fever, the trigger will start by querying for employees who were in close contact with the employee for the past 3 days. Using this query, the trigger will proceed to remove the close contact employees from any future meetings in the next 7 days, while the employee with fever will be removed from all future meetings. They will be removed regardless of whether the bookings were approved or not. Additionally, if any employees being removed are the bookers of the room, the entire booking will be cancelled.

5.0. Normal Form Analysis of Schema

- Employees
    - *Employees* is not in BCNF initially.
    - After doing BCNF decomposition, the tables will be as such
        - R1(resignedDate, isResigned)
        - R2(resignedDate, eid, ename, mobilePhoneContact, homePhoneContact, officePhoneContact, email)
    - Note that attributes *eid*, *mobilePhoneContact* and *email* are functionally equivalent.
- Junior
    - *Junior* is in BCNF already.
- Booker
    - *Booker* is in BCNF already.
- Senior
    - *Senior* is in BCNF already.
- Manager
    - *Manager* is in BCNF already.
- healthDeclaration
    - *healthDeclaration* is not in BCNF initially.
    - After doing BCNF decomposition, the tables will be as such
        - R1(temp,fever)
        - R2(date,temp,eid)
- Departments
    - *Departments* is in BCNF already.
- worksIn
    - *worksIn* is in BCNF already.
- meetingRooms
    - *meetingRooms* is in BCNF already.
- locatedIn
    - *locatedIn* is in BCNF already.
- Updates
    - *Updates* is in BCNF already.
- Sessions
    - *Sessions* is in BCNF already.
- Approves
    - *Approves* is in BCNF already.

- Books
    - *Books* is in BCNF already.
- Joins
    - *Joins* is in BCNF already.

6.0. Reflection

We learned to appreciate the power of PL/pgSQL. The availability of variables and control structures in PL/pgSQL affords us greater power in querying the database, and the procedural aspect is very comfortable for us to use. However, the documentation for it is difficult to digest at times and syntax errors can be tough to rectify.

We should have better allocated the deadlines of functionalities to do as some of us had to wait for a certain function to be done in order to do testing on a dependent function e.g. book_room function has to be done first before join_meeting function can be tested. Therefore, the core functions should have been done earlier. This would have allowed bugs to have been spotted earlier and save a lot of time instead of coming back again and waiting for a required function to be done before checking a dependent function.

Additionally, there were cases where certain triggers enforcing constraints were affecting the usage of other triggers and functions, emphasizing that meticulous testing needs to be done to ensure that the functionalities and triggers cover all test cases and work as they should. Looking back, we should have done data population for the database earlier so that we can test and check the functions implemented as we progressed.