
**CS2102 Project Report**

**Group No. 69**

Amas Lua (A0199368E)

Avril Lim (A0204887U)

Clifton Teo (A0199887U)

Kim Subin (A0205676Y)

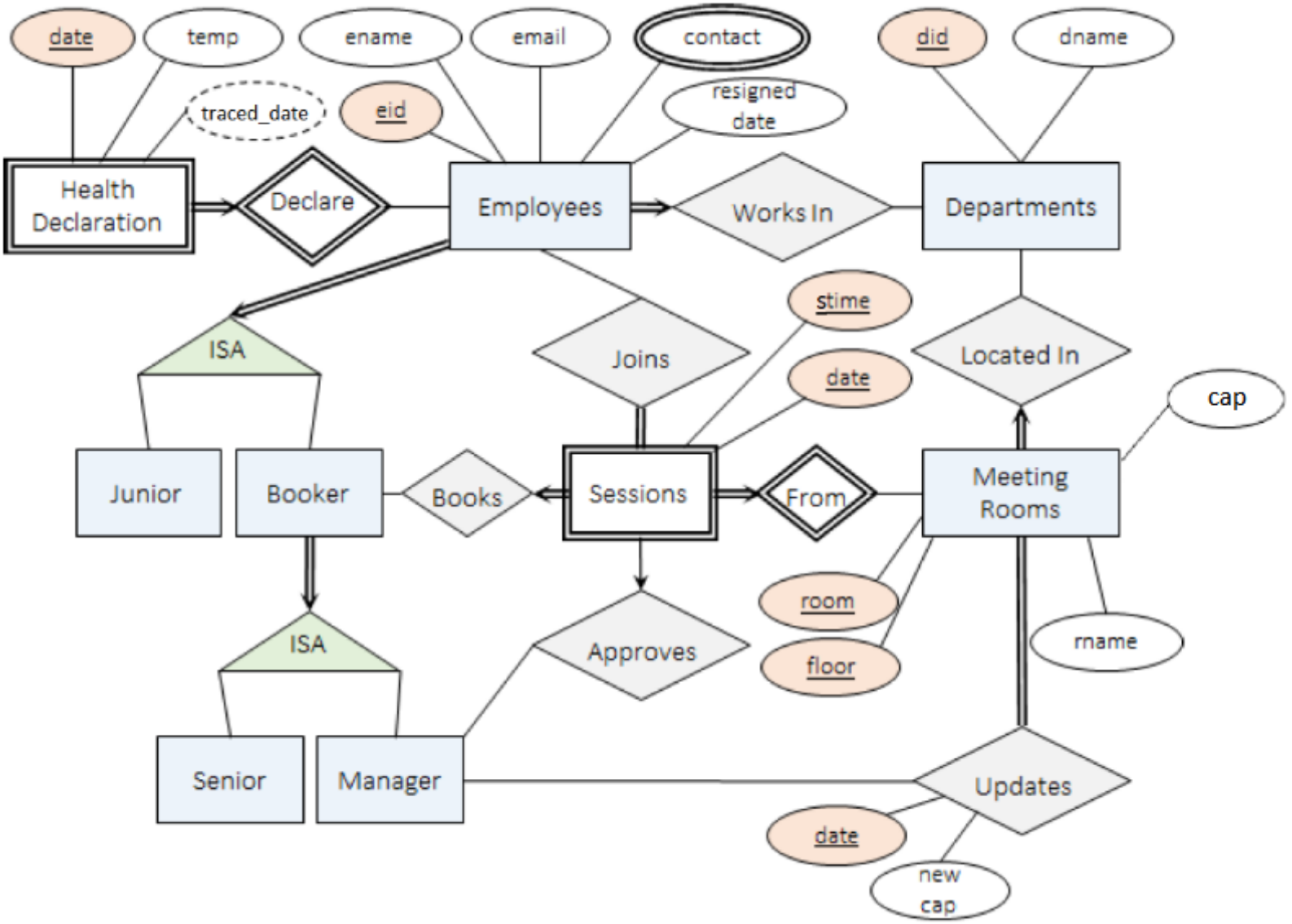
## Project Responsibilities

| | Schema design | Scripting | Testing/ Debugging | NF analysis | Report writing |
|---|---|---|---|---|---|
| Amas Lua | ✔ | ✔ | ✔ | ✔ | ✔ |
| Avril Lim | ✔ | ✔ | ✔ | ✔ | ✔ |
| Clifton Teo | ✔ | ✔ | ✔ | ✔ | ✔ |
| Kim Subin | ✔ | ✔ | ✔ | ✔ | ✔ |

## Additional Notes

The data.sql file uses procedures defined in proc.sql. Hence, the three files should be run in the order schema.sql > proc.sql > data.sql. Additionally, to simplify usage of the functions, all input for date fields are standardized as ‘YYYY/MM/DD’, and all input for start and end time is simply the integer of the start or end hour in 24-hour format (e.g. 17 for 5pm) across all the functions defined in proc.sql. The procedures defined also assume that the local system and the database server operate on the same time zone (GMT +8, due to use of *current_time* and *current_date*). The code for the three interesting triggers discussed herein are attached and labelled in the Appendix.

**ER Diagram**


date
temp
ename
email
contact
did
dname
traced_date
eid
resigned date
Health Declaration
Declare
Employees
Works In
Departments
stime
ISA
Joins
Located In
date
cap
Junior
Booker
Books
Sessions
From
Meeting Rooms
room
rname
ISA
Approves
floor
Senior
Manager
Updates
date
new cap

Application constraints not captured by the ER Diagram:

1. *Resigned* individuals:
    (a) If they were a booker of meetings, all future meetings booked by them will be deleted from **Sessions**.
    (b) If they were a junior, they will be removed from all future meetings that they were going to attend from **Joins**.

2. Updating of room capacity
    (a) When the room capacity is changed, a trigger function *update_updates()* will be triggered. If the room capacity has been updated on the same day, the value of *cap* will be updated in **Updates**. Otherwise, a new row will be inserted into **Updates**.

3. Each employee is assigned a unique eid and unique email address automatically.

4. The employee who booked the meeting will be automatically added to **Joins** via *add_booker()* function.

5. If an employee has a fever, they cannot join any meetings regardless of approval status.

# Relational Database Schema

Justification for Non-trivial Design Decisions:

1. **Employees**
    a. *mobile_contact* is UNIQUE NOT NULL because employee IDs and emails are automatically generated. If *mobile_contact* is not UNIQUE, the same person can be added into our database multiple times as the employee ID is the primary key. By setting *mobile_contact* to be UNIQUE NOT NULL, we can prevent this from happening.
    b. *office_contact* is UNIQUE NOT NULL so that every employee in the company is contactable.
    c. *traced_date* is added as an attribute for easy reference of the last time the employee was flagged due to a fever or because they were a close contact. This allows checks for the duration that has passed since they had a fever/were a close contact to be easily performed for other constraints to be enforced (e.g. close contacts cannot book/join meetings for the next 7 days), rather than a simple boolean to check if the employee was traced.
2. **Meeting_Rooms**
    a. *did* is a foreign key which references **Departments** as each department has their own meeting rooms. When a department is removed, there is no need for the existence of the department's meeting rooms, so we set ON DELETE CASCADE.
3. **Sessions**
    a. *eid* is a foreign key which references **Booker** as a person who is not a booker should not be allowed to book sessions.
    *b.* (*Room, floor*) is a foreign key which references **Meeting_Rooms** to enforce that it is a weak entity set of **Meeting_Rooms**.
4. **Senior**
    a. *eid* is a foreign key which references **Booker** as a senior or manager able to book meeting rooms. When the employee resigns, they will be removed from **Booker**, and consequently **Senior** tables via ON DELETE CASCADE.
5. **Manager**
    a. *eid* is a foreign key which references **Booker** as a manager is able to book meeting rooms. When the employee resigns, they will be removed from **Booker**, and consequently **Manager** tables via ON DELETE CASCADE.
6. **Updates**
    a. (*Date, room, floor*) is the primary key of **Updates** as this allows us to keep records of the update done to the meeting rooms (identified by *room*, *floor*) on each date. If the

room capacity has been updated on the same day, the value of *cap* will be updated in **Updates**. Otherwise, a new row will be inserted into **Updates**.

b. *mid* is a foreign key which references **Manager** as only managers are allowed to update capacity.

7. **Health_declaration**
    a. Check on *temp* attribute is carried out to ensure that the temperature inputted by the employee is within the acceptable range of 34 to 43.
    b. *eid* attribute is part of the primary key as **Health_Declaration** is a weak entity of **Employees**.
8. **Approves**
    a. (*date, stime, room, floor*) is a foreign key which references **Sessions** ON DELETE CASCADE for a situation when:
        i. If the booker of the approved meeting was traced, then a deletion from **Sessions** will occur via *contact_tracing* which will DELETE CASCADE and remove from the **Approves** table.
    b. *eid* is a foreign key which references **Manager**
        i. This is to enforce the constraint that only managers can approve meetings.
        ii. ON DELETE SET NULL is used so that when an employee resigns and is deleted from the **Managers** table, the *eid* in **Approves** will simply be set to NULL to allow the delete to happen. For past records, the approving manager's ID is not needed for contact tracing and future approved sessions will not be affected either. Although it would be ideal to preserve all records, this was a trade-off made to ensure other constraints such as ISA hierarchies could continue to be preserved.
        iii. *eid* is not part of the primary key to allow for ON DELETE SET NULL. Instead, it is enforced by a trigger (*approval_check)* which prevents insertions into **Approves** without a manager's *eid*.
9. **Joins**
    a. (*date, start time, room, floor*) is a foreign key which references **Sessions** to ensure that users can only join meetings that have been booked, while the ON DELETE CASCADE ensures that when sessions are removed, rooms are unbooked, or a booker resigns, sessions related to that booking are removed.

Application constraints not captured by Relational Database Schema:

1. The employees need to declare their temperature daily.

2. The employees will not be able to join meetings
    a. if they have a fever or
    b. if they were in close contact with someone who had a fever in the past 3 days. They cannot join meetings for the next 7 days

3. A manager can only approve a meeting that is booked in his/her department (i.e., the manager and the meeting room is in the same department).

4. Past records (from **Joins**, **Approves** and **Sessions**) cannot be deleted.

5. Resigned individuals are not able to join future meetings.

## Interesting triggers implemented for the application

Trigger #1: leave_check

The leave_check trigger is used to check if a certain employee is allowed to leave a meeting before deleting on **Joins**. There are several conditions that they need to meet in order to ensure that they are allowed to leave a meeting without any issues:

1. If the employee has resigned, they can be removed from the meeting
2. If the booker of the meeting has resigned:
    a. Any future meetings booked by them will be deleted, both for them and for participants (non-bookers) regardless of approval status.
        i. Interestingly, even though this is enforced by foreign key constraints to **Sessions**, the sequence of trigger events requires this trigger to be present for the leave to work.
3. If the meeting has been approved:
    a. Employees are not allowed to leave the meeting at all, except for special cases:
        i. If an employee had a fever or was assigned close contact status in the past 3 days, they will be removed from all future meetings for the next 7 days, regardless of whether it has been approved or not
    b. The booker is allowed to unbook future approved meetings, in which case all non-bookers will automatically have that future meeting removed for them as well (i.e. 'leave' the meeting by deleting entry on **Joins**)

Design of leave_check:
Timing of trigger is before delete on **Joins** using `*leave_meeting*`. It is triggered anytime there is a delete on **Sessions** as well, because any deletion on **Sessions** will activate the trigger `*can_we_unbook*` which executes a delete on Joins as long as the session is not in the past (defined by current date and time). This is logical because any deletion on sessions should also execute a check to delete all related entries on Joins for non-bookers also in that meeting.

1. For the first case of a straightforward delete on **Joins** using `*leave_meeting*`, the trigger checks the conditions in the following order:
    a. If the meeting is a future meeting
    b. If the employee leaving the meeting has resigned
    c. If the booker of the meeting itself in **Sessions** has resigned
    d. If the booker of the meeting has a fever or has been assigned close contact status in the past 7 days

e. If the booker has unbooked the meeting (i.e. the session has been deleted from **Sessions**)
f. If the employee leaving has been assigned close contact status or has a fever
g. If the meeting has not been approved

If any of the above conditions are met, the employee will be deleted from joins.

2. For the second case of deleting from **Joins** as a result of `*unbook_room*` and the resulting trigger `*can_we_unbook*`, once the room is allowed to be unbooked, we want to allow all employees that joined the meeting to leave by checking:
    a. If the session has been deleted from sessions

Trigger #2: booking_check

This trigger is used to check whether the booker meets all of these criteria before being able to book meeting rooms:

1. The booking must not be earlier than the present date and time (defined by *current_date* & *current_time*).
2. The booker has not been assigned close contact status or had a fever in the past 7 days.
3. The booker must not have resigned.
4. The meeting room that they want to book must be available for the duration of their intended meeting.
5. The booker must have declared their temperature on the day they make the booking

Design of booking_check:

The timing of the trigger is before insert on **Sessions** so that if any of the conditions specified by the trigger function are not met, the booking does not take place. Notices are raised for the conditions causing the update on **Sessions** to fail, so that employees know why they are unable to make the booking. Bookers are required to declare their temperature before making the booking to ensure that all employees abide by the rule of daily health declarations for a safe work environment. This is enforced by *booker_join* which is executed when a booking is made and triggers an automatic addition to **Joins**, which in turn triggers *join_check* which will raise an exception and disallow the booking if the booker has not declared their temperature that day.

Trigger #3: join_check

This trigger is used to check whether an employee is allowed to join a meeting (using function `*join_meeting*`) by checking the following conditions:

1. The meeting has not been approved
2. The meeting has been booked
3. The meeting is not in the past
4. The employee was not assigned close contact status in the past 7 days.
5. The employee declared their temperature on the day they use `*join_meeting*`.
    a. This will encourage employees to do their daily declaration of temperature as the exception 'Employees must declare temperature to book meeting' will be raised when employees who have not declared their temperature try to join a meeting
6. The meeting must not be at full capacity.
7. The employee joining the meeting has not resigned.

Design of join_check:
Timing of the trigger is before insert on Joins so that if any of the conditions specified by the trigger function are not met, the employee cannot join the meeting.

Insertion onto joins occurs immediately for the booker who booked the room, and when `*join_meeting*` is used. Queries are written first to assign values into the declared variables for use in the 'IF' statements .Exceptions or notices are raised to the users when conditions to join the meeting are not met, so that employees know why they are unable to join the meeting.

# 3NF Analysis

Employees

| eid | A |
|---|---|
| did | B |
| ename | C |
| email | D |
| home_contact | E |
| mobile_contact | F |
| office_contact | G |
| resigned_date | H |
| traced_date | I |

Employees FDs:

{A} → {A, B, C, D, E, F, G, H, I}

{D} → {A, B, C, D, E, F, G, H, I}

{F} → {A, B, C, D, E, F, G, H, I}

{G} → {A, B, C, D, E, F, G, H, I}

{A, D, F, G} → {A, B, C, D, E, F, G, H, I}

There are no non-trivial and decomposed FDs that violate 3NF conditions.

Meeting_Rooms

| rname | J |
|---|---|
| room | K |
| floor | L |
| did | B |
| cap | M |

{K,L} → {J,K,L,B,M}

There are no non-trivial and decomposed FDs that violate 3NF conditions.

Departments

| did | N |
|---|---|
| dname | O |

{N} → {O}

This is a trivial case as there are only 2 attributes.

Updates

| date | P |
|---|---|
| room_cap | M |
| room | K |
| floor | L |
| mid | A |

{K, L, P} → {A, K, L, M, P}

There are no non-trivial and decomposed FDs that violate 3NF conditions.

Sessions

| date | S |
|---|---|
| stime | R |
| room | K |
| floor | L |
| eid | A |

{K,L,S,R} → {A,K,L,S,R}

There are no non-trivial and decomposed FDs that violate 3NF conditions.

Health_declaration

| | |
|---|---|
| date | Q |
| eid | A |
| temp | T |

{A,Q} → {A,Q,T}

There are no non-trivial and decomposed FDs that violate 3NF conditions.

Joins

| | |
|---|---|
| date | S |
| stime | R |
| eid | A |
| room | K |
| floor | L |

{A,K,L,R,S} → {A,K,L,R,S}

There are no non-trivial and decomposed FDs that violate 3NF conditions.

Approves

| | |
|---|---|
| date | S |
| stime | R |
| room | K |
| floor | L |
| eid | A |

{K,L,R,S} → {A,K,L,R,S}

There are no non-trivial and decomposed FDs that violate 3NF conditions.

## Reflection

1. Implementing the ER Diagram initially took us quite a long time, as we were new to SQL as a language and didn't know what could or could not be implemented. As such, we overthought and tried to fit too much into our ER Diagram which would create problems downstream.
2. SQL:
    a. For certain functions, which will trigger multiple triggers, it was quite difficult to interpret what was causing issues even when error messages were shown. For example, when we wanted to remove from the **Joins** table, '*check_leave'* did not allow the removal even though we thought the conditions were met, which we determined to be caused by the sequence of the functions and triggers being called.
    b. SQL is a new language to all the members of our team, therefore for simple things like substring, converting date, converting time, it took some trial and error and googling to figure out how it works.
3. Before enrolling in this module, SQL seemed like a very basic language as we were only exposed to queries. However, through this project, we realised that database design is actually difficult, and database engineers deserve much more credit than they are given.
4. The sequential design process of a database application
    a. We learnt to first try to capture as many constraints as possible within the ER diagram, followed by the relational database schema. Triggers are a last resort for constraints that cannot be captured by the ER diagram or relational schema
5. Communication and teamwork is essential
    a. With this project, there were many corner cases that we might not have thought of when writing the code for each function. By helping to test each other's functions, we managed to find many bugs in the code or conditions that were missing from our functions which we did not consider before.
    b. Communication was key throughout the project. Although it was a group project, because many functions and triggers were related or linked to each other we had to communicate constantly to ensure that everyone was on the same page. In a project like this, we cannot afford to simply split up the work and put it together like in some other modules.
6. Raising exceptions are helpful for debugging
    a. Raising exceptions or notices allowed us to see when a delete or update statement was being rejected by a trigger, and at which line it was being rejected- which allowed for much easier debugging as we were able to tell where the problematic portion of the code was.

## Appendix

### Interesting Trigger #1: leave_check

```
CREATE OR REPLACE FUNCTION check_leave()
    RETURNS TRIGGER AS
$$
DECLARE
    a_count     INTEGER;
    s_count     INTEGER;
    trace_date  DATE;
    resignation DATE;

BEGIN

    SELECT COUNT(*)
    INTO a_count
    FROM Approves
    WHERE OLD.date = date
      AND OLD.stime = stime
      AND OLD.room = room
      AND OLD.floor = floor;

    SELECT COUNT(*)
    INTO s_count
    FROM Sessions
    WHERE OLD.date = date
      AND OLD.stime = stime
      AND OLD.room = room
      AND OLD.floor = floor;

    SELECT traced_date
    INTO trace_date
    FROM Employees
    WHERE OLD.eid = eid;

    SELECT resigned_date
    INTO resignation
    FROM Employees
    WHERE OLD.eid = eid;

    IF (old.date < current_date) THEN
        RAISE NOTICE 'You cannot leave a past meeting';
        RETURN NULL;
    ELSEIF (old.date = current_date AND SUBSTRING(CAST(old.stime AS TEXT), 1, 2) <=
                                        SUBSTRING(CAST(current_timestamp AS TEXT), 12,
2)) THEN
        RAISE NOTICE 'You cannot leave a past meeting';
        RETURN NULL;
    ELSEIF (resignation IS NOT NULL) THEN
        RAISE NOTICE 'Employee has resigned';
        RETURN OLD;
    ELSEIF (SELECT e.resigned_date
            FROM (Joins j JOIN Sessions s USING (date, stime, room, floor))
                     JOIN Employees e ON s.eid = e.eid
            WHERE j.eid = OLD.eid
            LIMIT 1) IS NOT NULL THEN
```

```
        RAISE NOTICE 'Booker of this meeting has resigned';
        RETURN OLD;
    ELSEIF (current_date - (SELECT e.traced_date
                            FROM (Joins j JOIN Sessions s USING (date, stime, room,
floor))
                                     JOIN Employees e ON s.eid = e.eid
                            WHERE j.eid = OLD.eid
                              AND date = OLD.date
                              AND stime = OLD.stime
                              AND room = OLD.room
                              AND floor = OLD.floor
                            LIMIT 1) <= 7) THEN
        RAISE NOTICE 'Booker of this meeting has a fever or is a close contact';
        RETURN OLD;
    ELSEIF (s_count = 0) THEN
        RAISE NOTICE 'Booker has unbooked meeting';
        RETURN OLD;
    ELSEIF (trace_date IS NOT NULL AND (current_date - trace_date <= 7)) THEN
        RAISE NOTICE 'Employee leaves approved meeting due to fever';
        RETURN OLD;
    ELSEIF (a_count > 0) THEN
        RAISE NOTICE 'You cannot leave an approved meeting';
        RETURN NULL;
    ELSE
        RETURN OLD;
    END IF;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER leave_check
    BEFORE DELETE
    ON Joins
    FOR EACH ROW
EXECUTE FUNCTION check_leave();
```

**Interesting Trigger #2: booking_check**

```
CREATE OR REPLACE FUNCTION check_booking()
    RETURNS TRIGGER AS
$$
DECLARE
    trace_date DATE;
    resigned   DATE;
    count      INTEGER;
    starthour  TIME := NEW.stime;
    bdate      DATE := NEW.date;
BEGIN
    SELECT traced_date
    INTO trace_date
    FROM Employees
    WHERE eid = NEW.eid;

    SELECT resigned_date
    INTO resigned
    FROM Employees
    WHERE eid = NEW.eid;

    SELECT COUNT(*)
    INTO count
    FROM Approves
    WHERE NEW.stime = stime
      AND NEW.date = date
      AND NEW.room = room
      AND NEW.floor = floor;

    IF (bdate < current_date) THEN
        RAISE NOTICE 'You cannot book a date that is earlier than the present.';
        RETURN NULL;
    ELSEIF (bdate = current_date AND
            starthour <= current_time) THEN
        RAISE NOTICE 'You cannot book a time earlier than the present';
        RETURN NULL;
    ELSEIF (current_date - trace_date <= 7) THEN
        RAISE NOTICE 'Close contacts or employees with fevers cannot book rooms';
        RETURN NULL;
    ELSEIF resigned IS NOT NULL THEN
        RAISE NOTICE 'Resigned employees cannot book rooms';
        RETURN NULL;
    ELSEIF count > 0 THEN --already booked by someone else
        RAISE NOTICE 'Room has already been booked at this time';
        RETURN NULL;
    ELSE
        RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER booking_check
    BEFORE INSERT OR UPDATE
    ON Sessions
    FOR EACH ROW
EXECUTE FUNCTION check_booking();
```

**Interesting Trigger #3: join_check**

```
CREATE OR REPLACE FUNCTION check_join()
    RETURNS TRIGGER AS
$$
DECLARE
    a_count          INTEGER;
    s_count          INTEGER;
    trace_date       DATE;
    rm_capacity      INT;
    num_attendees    INT;
    resignation_date DATE;
BEGIN
    SELECT COUNT(*)
    INTO a_count
    FROM Approves
    WHERE NEW.date = date
      AND NEW.stime = stime
      AND NEW.room = room
      AND NEW.floor = floor;

    SELECT COUNT(*)
    INTO S_count
    FROM Sessions
    WHERE NEW.date = date
      AND NEW.stime = stime
      AND NEW.room = room
      AND NEW.floor = floor;

    SELECT traced_date
    INTO trace_date
    FROM Employees
    WHERE NEW.eid = eid;

    SELECT cap
    INTO rm_capacity
    FROM Meeting_Rooms
    WHERE NEW.room = room
      AND NEW.floor = floor;

    SELECT COUNT(*)
    INTO num_attendees
    FROM Joins
    WHERE NEW.stime = stime
      AND NEW.date = date
      AND NEW.room = room
      AND NEW.floor = floor;

    SELECT resigned_date
    INTO resignation_date
    FROM Employees
    WHERE NEW.eid = eid;

    IF a_count > 0 THEN
        RAISE NOTICE 'Approved meetings cannot be joined';
        RETURN NULL;
    ELSEIF s_count = 0 THEN
```

```
        RAISE NOTICE 'This slot has not been booked yet';
        RETURN NULL;
    ELSEIF NEW.date < current_date THEN
        RAISE NOTICE 'You cannot join a meeting in the past';
        RETURN NULL;
    ELSEIF (NEW.date = current_date AND SUBSTRING(CAST(NEW.stime AS TEXT), 1, 2) <=
                                        SUBSTRING(CAST(current_timestamp AS TEXT), 12,
2)) THEN
        RAISE NOTICE 'You cannot join a meeting in the past';
        RETURN NULL;
    ELSEIF ((NEW.date - trace_date) < 7 AND trace_date IS NOT NULL) THEN
        RAISE EXCEPTION 'Close contacts cannot join meetings for 7 days';
    ELSEIF (NOT EXISTS(SELECT *
                       FROM Health_Declaration
                       WHERE eid = NEW.eid
                         AND date = current_date)) THEN
        RAISE EXCEPTION 'Employees must declare temperature to join meetings';
    ELSEIF num_attendees >= rm_capacity THEN
        RAISE EXCEPTION 'This meeting is at full capacity';
    ELSEIF resignation_date <= NEW.date THEN
        RAISE EXCEPTION 'Employee has resigned and thus cannot join the meeting.';
    ELSE
        RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER join_check
    BEFORE INSERT
    ON Joins
    FOR EACH ROW
EXECUTE FUNCTION check_join();
```