

A6: Indices, triggers, user functions and population

SegFault is a collaborative platform for programmers to learn, discuss different approaches, present ideas and share knowledge in a Q&A style.

To this end, the following sections provide detailed insight into the inner workings of the project's database. The first section depicts the expected workload on the system, the second section specifies and explains the proposed indices to the database, and the third section comprises the database's triggers.

1. Database Workload

A study of the predicted system load (database load), organized in subsections.

1.1. Tuple Estimation

Estimate of tuples at each relation.

Relation reference	Relation Name	Order of magnitude	Estimated growth
R01	Category	units	dozens
R02	QuestionCategory	units	dozens
R03	Question	units	dozens
R04	Answer	units	dozens
R05	Commentable	units	dozens
R06	Comment	units	dozens
R07	Message	units	dozens
R08	MessageVersion	units	dozens
R09	Vote	units	dozens
R10	User	units	dozens
R11	Moderator	units	dozens
R12	Notification	units	dozens
R13	CommentableNotification	units	dozens
R14	BadgeNotification	units	dozens
R15	BadgeAttainment	units	dozens
R16	Badge	units	dozens
R17	ModeratorBadge	units	dozens
R18	TrustedBadge	units	dozens

1.2. Frequent Queries

SELECT01

Query reference	SELECT01
Query	Select all comments of a Message, order by their
description	descending score
Query frequency	magnitude per time

```
SELECT *
FROM (
  SELECT DISTINCT ON (comment.id) comment.id, commentable.id, score, is_banned, author, content
  FROM commentable, comment, message, message_version
  WHERE
    commentable.id = $messageId AND
    commentable.id = comment.commentable_id AND
    comment.id = message.id AND
    message.id = message_version.message_id
  ORDER BY
    comment.id,
    creation_time DESC
) updated_comments
ORDER BY
  updated_comments.score DESC;
```

SELECT02

Query reference	SELECT02
Query	Select the first 25 questions, ordered by descending
description	date of the last edition
Query frequency	magnitude per time

```
SELECT *
FROM (
  SELECT DISTINCT ON (question.id) question.id, title, correct_answer, score, is_banned, author
  FROM question, commentable, message, message_version
  WHERE
    question.id = commentable.id AND
    commentable.id = message.id AND
    message.id = message_version.message_id
  ORDER BY
    question.id,
    message_version.creation_time DESC
LIMIT 25
```

```

    ) unordered_questions
ORDER BY
    unordered_questions.creation_time DESC

```

SELECT03

Query reference	SELECT03
Query description	Select the 25 questions with most answers (the most discussed questions)
Query frequency	magnitude per time

```

SELECT question.id, COUNT(answer.question_id) AS num_answers
FROM question, answer, message
WHERE
    question.id = message.id AND
    answer.question_id = question.id
GROUP BY
    question.id
ORDER BY
    num_answers DESC
LIMIT 25;

```

SELECT04

Query reference	SELECT04
Query description	Select the contents of the 25 most answered questions
Query frequency	magnitude per time

```

SELECT * FROM (
    SELECT question.id, COUNT(answer.question_id) AS num_answers
    FROM question, answer, message
    WHERE
        question.id = message.id AND
        answer.question_id = question.id
    GROUP BY
        question.id
    ORDER BY
        num_answers DESC
    LIMIT 25
) most_answered
JOIN (
    SELECT DISTINCT ON (question.id) question.id, title, correct_answer, score, is_banned, aut
    FROM question, commentable, message, message_version

```

```

WHERE
    question.id = commentable.id AND
    commentable.id = message.id AND
    message.id = message_version.message_id
) info
ON
    most_answered.id = info.id
ORDER BY
    most_answered.num_answers DESC;

```

SELECT05

Query reference	SELECT05
Query description	Select the categories ordered by number of posts/questions in each category
Query frequency	magnitude per time

```

SELECT name, num_posts
FROM category
ORDER BY
    num_posts DESC;

```

SELECT06

Query reference	SELECT06
Query description	For a given category, select the 25 most recent questions and their contents (and select only those that aren't banned)
Query frequency	magnitude per time

```

SELECT *
FROM (
    SELECT DISTINCT ON (question.id) category.id, question_id, title, content, correct_answer,
    FROM category, question, question_category, message, message_version
    WHERE
        category.id = $categoryId AND
        question_category.question_id = question.id AND
        question_category.category_id = category.id AND
        question.id = message.id AND
        message.id = message_version.message_id
    GROUP BY question.id, category.id, question_category.question_id, title, content, correct_answer
    HAVING
        is_banned = FALSE

```

```

ORDER BY
    question.id,
    creation_time DESC
LIMIT 25
) category_questions
ORDER BY
    category_questions.creation_time DESC

```

SELECT07

Query reference	SELECT07
Query description	Select all the answers of a given question, from newest to oldest
Query frequency	magnitude per time

```

SELECT *
FROM (
    SELECT DISTINCT ON (answer.id) answer.id, content, creation_time, is_banned, author
    FROM question, answer, message, message_version
    WHERE
        question.id = $questionId AND
        question.id = answer.question_id AND
        answer.id = message.id AND
        message.id = message_version.message_id
    GROUP BY
        answer.id, content, creation_time, is_banned, author
    ORDER BY
        answer.id,
        creation_time DESC
) question_answers
ORDER BY
    question_answers.creation_time;

```

SELECT08

Query reference	SELECT08
Query description	Select all of a User's questions
Query frequency	magnitude per time

```

SELECT *
FROM (
    SELECT DISTINCT ON (question.id) question.id, title, content, score, creation_time, is_banned
    FROM "user" u, message, message_version, question

```

```

WHERE
    u.id = $user.Id AND
    u.id = message.author AND
    message.id = question.id AND
    message.id = message_version.message_id
GROUP BY
    question.id, title, content, score, creation_time, is_banned
ORDER BY
    question.id,
    creation_time DESC
) updated_questions
ORDER BY
    updated_questions.creation_time DESC;

```

SELECT09

Query reference	SELECT09
Query description	Select all of a User's answers
Query frequency	magnitude per time

```

SELECT *
FROM (
    SELECT DISTINCT ON (answer.id) answer.id, content, score, creation_time, is_banned
    FROM "user" u, message, message_version, answer
    WHERE
        u.id = $user.Id AND
        u.id = message.author AND
        message.id = answer.id AND
        message.id = message_version.message_id
    GROUP BY
        answer.id, content, score, creation_time, is_banned
    ORDER BY
        answer.id,
        creation_time DESC
) updated_answers
ORDER BY
    updated_answers.creation_time DESC;

```

SELECT10

Query reference	SELECT10
Query description	Select all of a User's comments
Query frequency	magnitude per time

```

SELECT *
FROM (
    SELECT DISTINCT ON (comment.id) comment.id, content, score, creation_time, is_banned
    FROM "user" u, message, message_version, comment
    WHERE
        u.id = $usedId AND
        u.id = message.author AND
        message.id = comment.id AND
        message.id = message_version.message_id
    GROUP BY
        comment.id, content, score, creation_time, is_banned
    ORDER BY
        comment.id,
        creation_time DESC
) updated_comments
ORDER BY
    updated_comments.creation_time DESC;

```

SELECT11

Query reference	SELECT11
Query description	Select all of a User's correct answers
Query frequency	magnitude per time

```

SELECT answer.id, score, is_banned
FROM answer, question, message, "user" u
WHERE
    u.id = $ usedId AND
    u.id = message.author AND
    message.id = answer.id AND
    answer.id = question.correct_answer;

```

SELECT12

Query reference	SELECT12
Query description	Select all of a User's unread notifications
Query frequency	magnitude per time

```

SELECT notification.id, notification.date
FROM "user" u, notification
WHERE
    u.id = $userId AND
    u.id = notification.user_id

```

```

GROUP BY
    u.id, notification.id
HAVING
    notification.read = FALSE;

```

SELECT13

Query reference	SELECT13
Query description	Select all of a User's badges
Query frequency	magnitude per time

```

SELECT badge.id, description, attainment_date
FROM "user" u, badge_attainment b_a, badge
WHERE
    u.id = b_a.user_id AND
    b_a.badge_id = badge.id;

```

SELECT14

Query reference	SELECT14
Query description	Select a User's profile information
Query frequency	magnitude per time

```

SELECT username, email, biography, reputation
FROM "user" u
WHERE
    u.id = $userId;

```

SELECT15

Query reference	SELECT15
Query description	Select a User's total number of questions
Query frequency	magnitude per time

```

SELECT u.id, COUNT(*)
FROM "user" u, message, question
WHERE
    u.id = $userId AND
    u.id = message.author AND
    message.id = question.id
GROUP BY
    u.id;

```


SELECT16

Query reference	SELECT16
Query description	Select a User's total number of answers
Query frequency	magnitude per time

```
SELECT u.id, COUNT(*)
FROM "user" u, message, answer
WHERE
  u.id = $userId AND
  u.id = message.author AND
  message.id = answer.id
GROUP BY
  u.id;
```

SELECT17

Query reference	SELECT17
Query description	Select a User's total number of comments
Query frequency	magnitude per time

```
SELECT u.id, COUNT(*)
FROM "user" u, message, comment
WHERE
  u.id = $userId AND
  u.id = message.author AND
  message.id = comment.id
GROUP BY
  u.id;
```

SELECT18

Query reference	SELECT18
Query description	Select all tags that partially match a given string
Query frequency	magnitude per time

```
SELECT *
FROM category
WHERE
  name LIKE "%$search%";
```

SELECT19

Query reference	SELECT19
Query description	Select all questions whose title partially matches a given string
Query frequency	magnitude per time

```
SELECT *
FROM question
WHERE
    title LIKE "%$search%";
```

1.3. Frequent Updates

Most important updates (INSERT, UPDATE, DELETE) and their frequency.

Query reference	INSERT01
Query description	Adding a new message version, either meaning the message was edited or is being added
Query frequency	magnitude per time

```
INSERT INTO message_version (id, content, message_id, creation_time, moderator_id)
VALUES (nextval('message_version_id_seq'::regclass), $content, $message_id, now(), $moderat
```

Query reference	INSERT02
Query description	Insert a new Question
Query frequency	magnitude per time

```
INSERT INTO question (id, title, correct_answer)
VALUES ($id, $title, $correct_answer);
```

Query reference	INSERT03
Query description	Create a new Answer
Query frequency	magnitude per time

```
INSERT INTO answer(id, question_id)
VALUES ($id, $question_id);
```

Query reference	INSERT04
Query description	Create new Comment
Query frequency	magnitude per time

```
INSERT INTO comment(id, commentable_id)
```

```
VALUES ($id, $commentable_id);
```

Query reference	INSERT05
Query description	new user registered
Query frequency	magnitude per time

```
INSERT INTO "user"(id, username, email, password_hash, biography, reputation)
VALUES (nextval('user_id_seq'::regclass), $username, $email, $password_hash, $biography, 0);
```

Query reference	INSERT06
Query description	Vote in a Message
Query frequency	magnitude per time

```
INSERT INTO vote(message_id, user_id, positive)
VALUES ($message_id, $user_id, $positive);
```

Query reference	UPDATE01
Query description	Update User Info
Query frequency	magnitude per time

```
INSERT INTO message_version (id, content, message_id, creation_time, moderator_id)
VALUES (nextval('message_version_id_seq'::regclass), $content, $message_id, now(), $moderator_id);
```

2. Proposed Indices

This section presents the proposed indices on the database. It is important to note that many indices, mainly on high cardinality, would theoretically be better off being implemented as hash indices. We purposefully did not choose these, because the PostgreSQL documentation actively discourages the usage of hash indices, as seen on the warning below.

Caution

Hash index operations are not presently WAL-logged, so hash indexes might need to be rebuilt with `REINDEX` after a database crash if there were unwritten changes. Also, changes to hash indexes are not replicated over streaming or file-based replication after the initial base backup, so they give wrong answers to queries that subsequently use them. For these reasons, hash index use is presently discouraged.

Figure 1: Hash Indices - Caution

2.1. Performance Indices

IDX01

Index reference	IDX01
Related queries	SELECT01
Index relation	comment
Index attribute	commentable_id
Index type	B-tree
Cardinality	medium
Clustering	yes
Justification	The Table is very large, and query SELECT01 must run efficiently as it is executed several times. It doesn't need range query support, and is a good candidate for clustering as its cardinality is medium.

```
CREATE INDEX comment_commentable ON comment USING btree(commentable_id);
```

(This index could be implemented as a hash index, but, as explained in this section's introduction, this is actively discouraged.)

IDX02

Index reference	IDX02
Related queries	SELECT01, SELECT02, SELECT04, SELECT06, SELECT07, SELECT08, SELECT09, SELECT10
Index relation	message_version
Index attribute	message_id
Index type	B-tree
Cardinality	medium
Clustering	yes
Justification	The Table is very large, and the corresponding queries are abundant and recurrent, thus must run efficiently. It doesn't need range query support, and is a good candidate for clustering as its cardinality is medium.

```
CREATE INDEX message_version_message ON message_version USING btree(message_id);
```

IDX03

Index reference	IDX03
Related queries	SELECT10
Index relation	message
Index attribute	author

Index type	B-tree
Cardinality	medium
Clustering	yes
Justification	The Table is very large, and the corresponding queries are abundant and recurrent, thus must run efficiently. It doesn't need range query support, and is a good candidate for clustering as its cardinality is medium.

```
CREATE INDEX message_author ON message USING btree(author);
```

IDX04

Index reference	IDX04
Related queries	SELECT12
Index relation	notification
Index attribute	user_id
Index type	B-tree
Cardinality	medium
Clustering	yes
Justification	The Table is very large, and the corresponding queries are abundant and recurrent, thus must run efficiently. It doesn't need range query support, and is a good candidate for clustering as its cardinality is medium.

```
CREATE INDEX notification_user ON notification USING btree(user_id);
```

2.2. Full-text Search Indices

IDX05

Index reference	IDX05
Related queries	SELECT18
Index relation	tag
Index attribute	name
Index type	GIN
Clustering	no
Justification	To improve the performance of full text searches on the tag's name. GIN because the table is infrequently updated, and this type of indices takes longer to create/update but lead to faster lookups. No ts_vector encoding is used because the tag's name is just one word long, and it's lemma is rarely identifiable.

```
CREATE INDEX tag_name ON tag USING gin(name);
```

IDX06

Index reference	IDX06
Related queries	SELECT19
Index relation	question
Index attribute	title
Index type	GiST
Clustering	no
Justification	To improve the performance of full text searches on the question's title. GiST because it's better for dynamic data.

```
CREATE INDEX question_title ON question USING gist(to_tsvector('english', title));
```

2.3. Constraint-enforcing Indices

The following indices are used to enforce special unique constraints, such as guaranteeing uniqueness of case insensitive usernames and emails.

```
CREATE INDEX unique_lowercase_username ON "user" (lower(username));
CREATE INDEX unique_lowercase_email ON "user" (lower(email));
```

3. Triggers

User-defined functions and trigger procedures that add control structures to the SQL language or perform complex computations, are identified and described to be trusted by the database server. Every kind of function (SQL functions, Stored procedures, Trigger procedures) can take base types, composite types, or combinations of these as arguments (parameters). In addition, every kind of function can return a base type or a composite type. Functions can also be defined to return sets of base or composite values.

Trigger reference	TRIGGER01	Trigger description	A message is banned if the amount of reports exceeds the limit define in BR08	
-------------------	-----------	---------------------	---	--

```
CREATE FUNCTION ban_message() RETURNS TRIGGER AS $$
BEGIN
    UPDATE message
    SET is_banned = TRUE
    WHERE NEW.id = message.id;
```

```

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

```

CREATE TRIGGER ban_message
AFTER UPDATE OF num_reports ON message
FOR EACH ROW
    WHEN ( NEW.num_reports >= 5 + NEW.score^(1/3) )
        EXECUTE PROCEDURE ban_message();

```

Trigger reference	TRIGGER02	Trigger description	An answer can only be marked as correct if it's an answer of that question

```

CREATE FUNCTION check_correct() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.correct_answer IS NOT NULL AND
        NOT EXISTS (SELECT * FROM answer WHERE NEW.correct_answer = id AND NEW.id = question_id)
        RAISE EXCEPTION 'An answer can only be marked as correct if it is an answer of the question';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

```

CREATE TRIGGER check_correct
BEFORE UPDATE OF correct_answer ON question
FOR EACH ROW EXECUTE PROCEDURE check_correct();

```

Trigger reference	TRIGGER03	Trigger description	A question must have between 1 and 5 categories

```

CREATE FUNCTION check_categories() RETURNS TRIGGER AS $$
DECLARE num_categories SMALLINT;
DECLARE current RECORD;
BEGIN
    IF TG_OP = 'INSERT' THEN
        current = NEW;
    ELSE
        current = OLD;
    END IF;
    SELECT INTO num_categories count(*)
    FROM question_category
    WHERE current.question_id = question_category.question_id;
    IF num_categories > 5 THEN
        RAISE EXCEPTION 'A question can only have a maximum of 5 categories';
    ELSIF num_categories < 1 THEN
        RAISE EXCEPTION 'A question must have at least 1 category';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

```

        END IF;
        RETURN NEW;
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_categories
    AFTER INSERT OR DELETE ON question_category
    FOR EACH ROW EXECUTE PROCEDURE check_categories();

```

Trigger reference	TRIGGER04	Trigger description	Update the number of posts a category is tagged in when another one is inserted
-------------------	-----------	---------------------	---

```

CREATE FUNCTION insert_category() RETURNS TRIGGER AS $$
BEGIN
    UPDATE category
        SET num_posts = num_posts + 1
        WHERE NEW.category_id = category.id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER insert_category
    AFTER INSERT ON question_category
    FOR EACH ROW EXECUTE PROCEDURE insert_category();

```

Trigger reference	TRIGGER05	Trigger description	Update the message's score once a vote is modified
-------------------	-----------	---------------------	--

```

CREATE FUNCTION update_score_vote() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.positive AND NOT OLD.positive THEN
        UPDATE message
            SET score = score + 2
            WHERE NEW.message_id = message.id;
    ELSIF NOT NEW.positive AND OLD.positive THEN
        UPDATE message
            SET score = score - 2
            WHERE NEW.message_id = message.id;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_score_vote
    BEFORE UPDATE ON Vote
    FOR EACH ROW EXECUTE PROCEDURE update_score_vote();

```


Trigger reference	TRIGGER06	Trigger description	Update the message's score once a vote is inserted
-------------------	-----------	---------------------	--

```

CREATE FUNCTION insert_score_vote() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.positive THEN
        UPDATE message
        SET score = score + 1
        WHERE NEW.message_id = message.id;
    ELSIF NOT NEW.positive THEN
        UPDATE message
        SET score = score - 1
        WHERE NEW.message_id = message.id;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

```

CREATE TRIGGER insert_score_vote
BEFORE INSERT ON Vote
FOR EACH ROW EXECUTE PROCEDURE insert_score_vote();

```

Trigger reference	TRIGGER07	Trigger description	Update the message's score once a vote is deleted
-------------------	-----------	---------------------	---

```

CREATE FUNCTION delete_score_vote() RETURNS TRIGGER AS $$
BEGIN
    IF OLD.positive THEN
        UPDATE message
        SET score = score - 1
        WHERE OLD.message_id = message.id;
    ELSIF NOT OLD.positive THEN
        UPDATE message
        SET score = score + 1
        WHERE OLD.message_id = message.id;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

```

CREATE TRIGGER delete_score_vote
BEFORE DELETE ON Vote
FOR EACH ROW EXECUTE PROCEDURE delete_score_vote();

```

Trigger reference	TRIGGER08	Trigger description	Update a user's reputation when one of its messages is reported as defined in BR03
-------------------	-----------	---------------------	--

```

CREATE FUNCTION update_reputation_reports() RETURNS TRIGGER AS $$
BEGIN
    UPDATE "user"
        SET reputation = reputation - (NEW.num_reports - OLD.num_reports)*10
        WHERE NEW.author = "user".id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

```

CREATE TRIGGER update_reputation_reports
    BEFORE UPDATE OF num_reports ON message
    FOR EACH ROW EXECUTE PROCEDURE update_reputation_reports();

```

Trigger reference	TRIGGER09	Trigger description	Update a user's reputation when one of its messages is voted by another user as defined in BR03
-------------------	-----------	---------------------	---

```

CREATE FUNCTION update_reputation_scores() RETURNS TRIGGER AS $$
BEGIN
    IF EXISTS (SELECT * FROM commentable WHERE NEW.id = commentable.id) THEN
        UPDATE "user"
            SET reputation = reputation + (NEW.score - OLD.score)
            WHERE NEW.author = "user".id;
    ELSIF EXISTS (SELECT * FROM comment WHERE NEW.id = comment.id) THEN
        UPDATE "user"
            SET reputation = reputation + (NEW.score - OLD.score)/2.0
            WHERE NEW.author = "user".id;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

```

CREATE TRIGGER update_reputation_scores
    BEFORE UPDATE OF score ON message
    FOR EACH ROW EXECUTE PROCEDURE update_reputation_scores();

```

Trigger reference	TRIGGER10	Trigger description	A user is awarded a "trusted" badge when they've correctly answered at least 50 questions
-------------------	-----------	---------------------	---

```

CREATE FUNCTION award_trusted() RETURNS TRIGGER AS $$
DECLARE answer_author BIGINT;
DECLARE trusted_id SMALLINT;
DECLARE num_correct_answers INTEGER;
BEGIN
    SELECT INTO answer_author author

```

```

        FROM message
        WHERE message.id = NEW.correct_answer;
SELECT INTO trusted_id id FROM trusted_badge;
IF NOT EXISTS
    (SELECT *
      FROM badge_attainment
      WHERE answer_author = badge_attainment.user_id AND trusted_id = badge_attainment.badge_id)
THEN
    SELECT INTO num_correct_answers count(*)
      FROM message, question
      WHERE message.id = question.correct_answer AND message.author = answer_author;
    IF num_correct_answers >= 50 THEN
        INSERT INTO badge_attainment (user_id, badge_id) VALUES (answer_author, trusted_id);
    END IF;
END IF;
RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

```

CREATE TRIGGER award_trusted
AFTER UPDATE OF correct_answer ON question
FOR EACH ROW EXECUTE PROCEDURE award_trusted();

```

Trigger reference	Trigger description
TRIGGER11	A user is awarded a “moderator” badge when they’ve been awarded the “trusted” badge and then achieved at least 500 reputation points

```

CREATE FUNCTION award_moderator_reputation() RETURNS TRIGGER AS $$
DECLARE moderator_id SMALLINT;
DECLARE trusted_id SMALLINT;
BEGIN
    SELECT INTO moderator_id id FROM moderator_badge;
    SELECT INTO trusted_id id FROM trusted_badge;
    IF NOT EXISTS
        (SELECT *
          FROM badge_attainment
          WHERE NEW.id = badge_attainment.user_id AND moderator_id = badge_attainment.badge_id)
    AND EXISTS
        (SELECT *
          FROM badge_attainment
          WHERE NEW.id = badge_attainment.user_id AND trusted_id = badge_attainment.badge_id)
    AND NEW.reputation >= 500 THEN
        INSERT INTO badge_attainment (user_id, badge_id) VALUES (NEW.id, moderator_id);
        INSERT INTO moderator (id) VALUES (NEW.id);
    END IF;
RETURN NEW;
END;

```

```

END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER award_moderator_reputation
AFTER UPDATE OF reputation ON "user"
FOR EACH ROW EXECUTE PROCEDURE award_moderator_reputation();
| Trigger reference | TRIGGER12 | | Trigger description | A user is awarded a
"moderator" badge when they've achieved at least 500 reputation points and
then were awarded the "trusted" badge | | ----- | -----
| ----- |

CREATE FUNCTION award_moderator_trusted() RETURNS TRIGGER AS $$
DECLARE moderator_id SMALLINT;
DECLARE trusted_id SMALLINT;
DECLARE rep REAL;
BEGIN
SELECT INTO moderator_id id FROM moderator_badge;
SELECT INTO trusted_id id FROM trusted_badge;
SELECT INTO rep reputation FROM "user" WHERE "user".id = NEW.user_id;
IF NEW.badge_id = trusted_id
AND NOT EXISTS
(SELECT *
FROM badge_attainment
WHERE NEW.user_id = badge_attainment.user_id AND moderator_id = badge_attainment.moderator_id
AND rep >= 500 THEN
INSERT INTO badge_attainment (user_id, badge_id) VALUES (NEW.user_id, moderator_id);
INSERT INTO moderator (id) VALUES (NEW.user_id);
END IF;
RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER award_moderator_trusted
AFTER INSERT ON badge_attainment
FOR EACH ROW EXECUTE PROCEDURE award_moderator_trusted();
| Trigger reference | TRIGGER13 | | Trigger description | A user can't vote their
own messages as stated in BR02 | | ----- | -----
| ----- |

CREATE FUNCTION check_own_vote() RETURNS TRIGGER AS $$
DECLARE message_author BIGINT;
BEGIN
SELECT INTO message_author author
FROM message
WHERE message.id = NEW.message_id;
IF message_author = NEW.user_id THEN

```

```

        RAISE EXCEPTION 'A user is not allowed to vote their own messages';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_own_vote
BEFORE INSERT ON Vote
FOR EACH ROW EXECUTE PROCEDURE check_own_vote();

```

Trigger reference	Trigger description
TRIGGER14	Update the number of reports in a message when one is made to it

```

CREATE FUNCTION insert_report() RETURNS TRIGGER AS $$
BEGIN
    UPDATE message
    SET num_reports = num_reports + 1
    WHERE NEW.message_id = message.id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER insert_report
BEFORE INSERT ON report
FOR EACH ROW EXECUTE PROCEDURE insert_report();

```

Trigger reference	Trigger description
TRIGGER15	Update the number of reports in a message when one is removed

```

CREATE FUNCTION delete_report() RETURNS TRIGGER AS $$
BEGIN
    UPDATE message
    SET num_reports = num_reports - 1
    WHERE NEW.message_id = message.id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER delete_report
BEFORE DELETE ON report
FOR EACH ROW EXECUTE PROCEDURE delete_report();

```

Trigger reference	Trigger description
TRIGGER16	A comment made to commentable item generates a notification towards the author of said commentable item

```

CREATE FUNCTION gen_comment_notification() RETURNS TRIGGER AS $$

```

```

DECLARE current_id BIGINT;
DECLARE notified_user BIGINT;
BEGIN
    SELECT INTO current_id nextval(pg_get_serial_sequence('notification', 'id'));
    SELECT INTO notified_user author FROM message WHERE message.id = NEW.commentable_id;
    INSERT INTO notification (id, user_id) VALUES (current_id, notified_user);
    INSERT INTO commentable_notification (id, notified_msg, trigger_msg) VALUES (current_id, NEW.id, NEW.id);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

```

CREATE TRIGGER gen_comment_notification
AFTER INSERT ON comment
FOR EACH ROW EXECUTE PROCEDURE gen_comment_notification();

```

Trigger reference	TRIGGER17	Trigger description
An answer to a question generates a notification towards the author of the question		

```

CREATE FUNCTION gen_answer_notification() RETURNS TRIGGER AS $$
DECLARE current_id BIGINT;
DECLARE notified_user BIGINT;
BEGIN
    SELECT INTO current_id nextval(pg_get_serial_sequence('notification', 'id'));
    SELECT INTO notified_user author FROM message WHERE message.id = NEW.question_id;
    INSERT INTO notification (id, user_id) VALUES (current_id, notified_user);
    INSERT INTO commentable_notification (id, notified_msg, trigger_msg) VALUES (current_id, NEW.id, NEW.id);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

```

CREATE TRIGGER gen_answer_notification
AFTER INSERT ON answer
FOR EACH ROW EXECUTE PROCEDURE gen_answer_notification();

```

Trigger reference	TRIGGER18	Trigger description
When a badge is awarded to a user a notification to that user is generated		

```

CREATE FUNCTION gen_badge_notification() RETURNS TRIGGER AS $$
DECLARE current_id BIGINT;
BEGIN
    SELECT INTO current_id nextval(pg_get_serial_sequence('notification', 'id'));
    INSERT INTO notification (id, user_id) VALUES (current_id, NEW.user_id);
    INSERT INTO badge_notification (id, badge_id) VALUES (current_id, NEW.badge_id);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

```
CREATE TRIGGER gen_badge_notification
AFTER INSERT ON badge_attainment
FOR EACH ROW EXECUTE PROCEDURE gen_badge_notification();
```

4. Complete SQL Code

SQL Code available in files `create.sql` and `populate.sql`

Revision history

Changes made to the first submission: 1. Item 1 1. Item 2

GROUP1763, 03/04/2018

André Cruz, up201503776@fe.up.pt
Daniel Marques, up201503822@fe.up.pt
Edgar Carneiro, up201503784@fe.up.pt
João Carvalho, up201504875@fe.up.pt