1. Проанализировать какие запросы могут выполняться наиболее часто в процессе работы приложения и добавить необходимые индексы.

USE vk;

SHOW TABLES;

SELECT \* FROM users LIMIT 1;

CREATE UNIQUE INDEX users\_email\_uq ON users(email);

CREATE INDEX users\_first\_name\_last\_name\_idx ON users (first\_name, last\_name);

CREATE INDEX profiles\_birthday\_idx ON profiles(birthday);

CREATE INDEX profiles\_hometown\_idx ON profiles(hometown);

CREATE UNIQUE INDEX communities\_name\_idx ON communities(name);

CREATE INDEX friendship\_friend\_id\_idx ON friendship(friend\_id);

CREATE INDEX likes\_target\_id\_idx ON likes(target\_id);

таблица media уже имеет составной KEY media\_type\_id и user\_id

CREATE INDEX media\_filename\_idx ON media(filename);

CREATE UNIQUE INDEX meetings\_name\_idx ON meetings(name);

таблица meetings\_users уже имеет составной первичный ключ meeting\_id и user\_id

таблица posts уже имеет составной KEY media\_id и user\_id

CREATE INDEX posts\_header\_idx ON posts(header);

таблица messages уже имеет составной KEY from\_user\_id и to\_ user\_id

1. Задание на оконные функции

Построить запрос, который будет выводить следующие столбцы:

имя группы

среднее количество пользователей в группах

самый молодой пользователь в группе

самый пожилой пользователь в группе

общее количество пользователей в группе

всего пользователей в системе

отношение в процентах (общее количество пользователей в группе / всего пользователей в системе) \* 100

SELECT DISTINCT communities.name,

COUNT(communities\_users.community\_id) OVER(PARTITION BY communities\_users.community\_id) AS USR\_per\_GRP,

COUNT(communities\_users.user\_id) OVER() / LAST\_VALUE(communities.id) OVER() AS AVG\_USR,

COUNT(profiles.user\_id) OVER() AS TOTAL\_USR,

FIRST\_VALUE(profiles.birthday) OVER (PARTITION BY communities.name ORDER BY profiles.birthday) AS OLDEST,

FIRST\_VALUE(profiles.birthday) OVER (PARTITION BY communities.name ORDER BY profiles.birthday DESC) AS YOUNGEST,

COUNT(communities\_users.community\_id) OVER usrgr / COUNT(profiles.user\_id) OVER() \* 100 AS "%%"

FROM communities

JOIN communities\_users

ON communities.id = communities\_users.community\_id

JOIN profiles

ON communities\_users.user\_id = profiles.user\_id

WINDOW usrgr AS (PARTITION BY communities\_users.community\_id);

\*\*\* правильно

SELECT DISTINCT

communities.name AS group\_name,

COUNT(communities\_users.user\_id) OVER()

/ (SELECT COUNT(\*) FROM communities) AS avg\_users\_in\_groups,

FIRST\_VALUE(users.id)

OVER(PARTITION BY communities.id ORDER BY profiles.birthday DESC) AS youngest,

FIRST\_VALUE(users.id)

OVER(PARTITION BY communities.id ORDER BY profiles.birthday) AS oldest,

COUNT(communities\_users.user\_id)

OVER(PARTITION BY communities.id) AS users\_in\_groups,

COUNT(users.id) OVER() AS users\_total,

COUNT(communities\_users.user\_id) OVER(PARTITION BY communities.id)

/ COUNT(users.id) OVER() \*100 AS '%%'

FROM communities

JOIN communities\_users

ON communities\_users.community\_id = communities.id

JOIN users

ON communities\_users.user\_id = users.id

JOIN profiles

ON profiles.user\_id = users.id;

-- Выводим имена пользователей

SELECT DISTINCT

communities.name AS group\_name,

COUNT(communities\_users.user\_id) OVER()

/ (SELECT COUNT(\*) FROM communities) AS avg\_users\_in\_groups,

FIRST\_VALUE(users.first\_name)

OVER(PARTITION BY communities.id ORDER BY profiles.birthday DESC) AS youngest\_first\_name,

FIRST\_VALUE(users.last\_name)

OVER(PARTITION BY communities.id ORDER BY profiles.birthday DESC) AS youngest\_last\_name,

FIRST\_VALUE(users.first\_name)

OVER(PARTITION BY communities.id ORDER BY profiles.birthday) AS oldest\_first\_name,

FIRST\_VALUE(users.last\_name)

OVER(PARTITION BY communities.id ORDER BY profiles.birthday) AS oldest\_last\_name,

COUNT(communities\_users.user\_id)

OVER(PARTITION BY communities.id) AS users\_in\_groups,

COUNT(users.id) OVER() AS users\_total,

COUNT(communities\_users.user\_id) OVER(PARTITION BY communities.id)

/ COUNT(users.id) OVER() \*100 AS '%%'

FROM communities

JOIN communities\_users

ON communities\_users.community\_id = communities.id

JOIN users

ON communities\_users.user\_id = users.id

JOIN profiles

ON profiles.user\_id = users.id;

-- Выносим определения некоторых окон

SELECT DISTINCT

communities.name AS group\_name,

COUNT(communities\_users.user\_id) OVER()

/ (SELECT COUNT(\*) FROM communities) AS avg\_users\_in\_groups,

FIRST\_VALUE(users.first\_name)

OVER birthday\_desc AS youngest\_first\_name,

FIRST\_VALUE(users.last\_name)

OVER birthday\_desc AS youngest\_last\_name,

FIRST\_VALUE(users.first\_name)

OVER birthday\_asc AS oldest\_first\_name,

FIRST\_VALUE(users.last\_name)

OVER birthday\_asc AS oldest\_last\_name,

COUNT(communities\_users.user\_id)

OVER(PARTITION BY communities.id) AS users\_in\_groups,

COUNT(users.id) OVER() AS users\_total,

COUNT(communities\_users.user\_id) OVER(PARTITION BY communities.id)

/ COUNT(users.id) OVER() \*100 AS '%%'

FROM communities

JOIN communities\_users

ON communities\_users.community\_id = communities.id

JOIN users

ON communities\_users.user\_id = users.id

JOIN profiles

ON profiles.user\_id = users.id

WINDOW birthday\_desc AS (PARTITION BY communities.id ORDER BY profiles.birthday DESC),

birthday\_asc AS (PARTITION BY communities.id ORDER BY profiles.birthday);

-- Доработка расчёта общего количества пользователей в системе

-- Применяем внешнее объединение так как иначе не будут выведены группы, у которых нет членов

SELECT DISTINCT

communities.name AS group\_name,

COUNT(communities\_users.user\_id) OVER()

/ (SELECT COUNT(\*) FROM communities) AS avg\_users\_in\_groups,

FIRST\_VALUE(users.first\_name)

OVER birthday\_desc AS youngest\_first\_name,

FIRST\_VALUE(users.last\_name)

OVER birthday\_desc AS youngest\_last\_name,

FIRST\_VALUE(users.first\_name)

OVER birthday\_asc AS oldest\_first\_name,

FIRST\_VALUE(users.last\_name)

OVER birthday\_asc AS oldest\_last\_name,

COUNT(communities\_users.user\_id)

OVER(PARTITION BY communities.id) AS users\_in\_groups,

(SELECT COUNT(\*) FROM users) AS users\_total,

COUNT(communities\_users.user\_id) OVER(PARTITION BY communities.id)

/ (SELECT COUNT(\*) FROM users) \*100 AS '%%'

FROM communities

LEFT JOIN communities\_users

ON communities\_users.community\_id = communities.id

LEFT JOIN users

ON communities\_users.user\_id = users.id

LEFT JOIN profiles

ON profiles.user\_id = users.id

WINDOW birthday\_desc AS (PARTITION BY communities.id ORDER BY profiles.birthday DESC),

birthday\_asc AS (PARTITION BY communities.id ORDER BY profiles.birthday);

1. (по желанию) Задание на денормализацию

pазобраться как построен и работает следующий запрос:

Найти 10 пользователей, которые проявляют наименьшую активность в использовании социальной сети.

SELECT users.id,

COUNT(DISTINCT messages.id) +

COUNT(DISTINCT likes.id) +

COUNT(DISTINCT media.id) AS activity

FROM users

LEFT JOIN messages

ON users.id = messages.from\_user\_id

LEFT JOIN likes

ON users.id = likes.user\_id

LEFT JOIN media

ON users.id = media.user\_id

GROUP BY users.id

ORDER BY activity media.user\_id

LIMIT 10;

Правильно-ли он построен?

Какие изменения, включая денормализацию, можно внести в структуру БД

чтобы существенно повысить скорость работы этого запроса?

С моей точки зрения, запрос построен правильно.

1. Чтобы увеличить скорость запроса можно создать индексы для messages.from\_user\_id, likes.user\_id, likes.user\_id или даже составные индексы messages.id и messages.from\_user\_id, likes.id и likes.user\_id, media.id и likes.user\_id
2. В таблицу users можно добавить колонку likes, где будет храниться кол-во лайков поставленных данным пользователем. Тогда данную информацию можно будет получать значительно быстрее. А таблицей likes пользоваться лишь для ограничения выставления нескольких лайков одним пользователем и для персонифицированной информации по лайкам.
3. Если данная информация очень важна и часто анализируется, то можно создать отдельную таблицу активности пользователя с подсчетом для каждого пользователя кол-во лайков по сообщениям, медиа файлам, постам и некоторые другие параметры (кол-во организованных пользователем групп, кол-во групп в которых состоит пользователь, кол-во организованных встреч и встреч в которых участвует пользователь, процент заполненности информации о себе, кол-во друзей и т.п. в зависимости от конкретной потребности).

\*\*\*правильно

-- Добавляем столбцы

ALTER TABLE users ADD COLUMN message\_cnt INT NOT NULL DEFAULT 0;

ALTER TABLE users ADD COLUMN like\_cnt INT NOT NULL DEFAULT 0;

ALTER TABLE users ADD COLUMN media\_cnt INT NOT NULL DEFAULT 0;

ALTER TABLE users ADD activity INT AS (message\_cnt + like\_cnt + media\_cnt);

-- заполняем исходя из текущих данных

UPDATE users u

SET u.message\_cnt = (SELECT COUNT(1) FROM messages m WHERE m.from\_user\_id = u.id),

u.like\_cnt = (SELECT COUNT(1) FROM likes l WHERE l.user\_id = u.id),

u.media\_cnt = (SELECT COUNT(1) FROM media m WHERE m.user\_id = u.id);

CREATE INDEX users\_activity\_idx ON users(activity);

DELIMITER //

-- Создаем триггеры, поддерживающие актуальность значений

-- messages

CREATE TRIGGER tai\_messages AFTER INSERT ON messages

FOR EACH ROW BEGIN

UPDATE users u

SET u.message\_cnt = u.message\_cnt + 1

WHERE u.id = NEW.from\_user\_id;

END//

CREATE TRIGGER tau\_messages AFTER UPDATE ON messages

FOR EACH ROW BEGIN

IF (OLD.from\_user\_id <> NEW.from\_user\_id) THEN

UPDATE users u

SET u.message\_cnt = u.message\_cnt + 1

WHERE u.id = NEW.from\_user\_id;

UPDATE users u

SET u.message\_cnt = u.message\_cnt - 1

WHERE u.id = OLD.from\_user\_id;

END IF;

END//

CREATE TRIGGER tad\_messages AFTER DELETE ON messages

FOR EACH ROW BEGIN

UPDATE users u

SET u.message\_cnt = u.message\_cnt - 1

WHERE u.id = OLD.from\_user\_id;

END//

-- likes

CREATE TRIGGER tai\_likes AFTER INSERT ON likes

FOR EACH ROW BEGIN

UPDATE users u

SET u.like\_cnt = u.like\_cnt + 1

WHERE u.id = NEW.user\_id;

END//

CREATE TRIGGER tau\_likes AFTER UPDATE ON likes

FOR EACH ROW BEGIN

IF (OLD.user\_id <> NEW.user\_id) THEN

UPDATE users u

SET u.like\_cnt = u.like\_cnt + 1

WHERE u.id = NEW.user\_id;

UPDATE users u

SET u.like\_cnt = u.like\_cnt - 1

WHERE u.id = OLD.user\_id;

END IF;

END//

CREATE TRIGGER tad\_likes AFTER DELETE ON likes

FOR EACH ROW BEGIN

UPDATE users u

SET u.like\_cnt = u.like\_cnt - 1

WHERE u.id = OLD.user\_id;

END//

-- media

CREATE TRIGGER tai\_media AFTER INSERT ON media

FOR EACH ROW BEGIN

UPDATE users u

SET u.media\_cnt = u.media\_cnt + 1

WHERE u.id = NEW.user\_id;

END//

CREATE TRIGGER tau\_media AFTER UPDATE ON media

FOR EACH ROW BEGIN

IF (OLD.user\_id <> NEW.user\_id) THEN

UPDATE users u

SET u.media\_cnt = u.media\_cnt + 1

WHERE u.id = NEW.user\_id;

UPDATE users u

SET u.media\_cnt = u.media\_cnt - 1

WHERE u.id = OLD.user\_id;

END IF;

END//

CREATE TRIGGER tad\_media AFTER DELETE ON media

FOR EACH ROW BEGIN

UPDATE users u

SET u.media\_cnt = u.media\_cnt - 1

WHERE u.id = OLD.user\_id;

END//

-- проверяем работу триггеров

SELECT \* FROM users;

SELECT \* FROM media;

INSERT INTO messages (from\_user\_id, to\_user\_id, body)

VALUES (1,2,'test');

update messages

SET from\_user\_id = 3

WHERE id = 101;

-- ИТОГОВЫЙ ЗАПРОС (cost в плане исходного запроса был 224,35, в данном запросе стал 10,25).

SELECT u.id, u.first\_name, u.last\_name, u.activity

FROM users u

ORDER BY u.activity

LIMIT 10;