Information Management II – SQL Project

Anton Yamkovoy – 17331565
29/11/2019

Version Control
A version control system should provide the tools for a user to keep track of progress and changes or deletions on a certain repository, all data should be held on to so that in the future a rollback of commits is possible and users have records of when they have commited to a certain repository, or statistics about who is commiting the most or least on a repo, I also thought that it would be useful to represent users similarly to a site like Github or where users may be repository contributors but may also have social connections such as followers and other users that they follow.
I created a database that models a version control system, for managing a software project. It has the functionality, to create a user, who may create pull requests to a repository, and create issues in a repository.
The user may join a repository as a contributor, and commit changes to the commit tree, comments are available to be posted on individual commits, and a user has a followers and following list views, which can be used to show their connections with other users
A user may be part of an organisation, which also contains other users, the repository system of collecting commits into a commit tree data structure, works based on state changes and file change objects, the commit contains a structure that allows for paths (branches) to be tracked through the tree, therefore implementing a branching system into the repository, more about the technicalities of the commit tree will be explained in a following section

Entity Diagram:
I started by modelling a entity diagram containing the initial entities that I thought a revision control system should have, page 2

Entity Relational Modelling:
After completing the model of the entities, I added relationships between them, containing the semantic meanings of the relations, the cardinalities that they had, and this would in the next step show the foreign and primary key constraints when modelling a relational schema : page 3

Relational schema:
From a relational schema a set of sql tables can be made, it is the natural next step of progression from the entity relational diagram : page 4

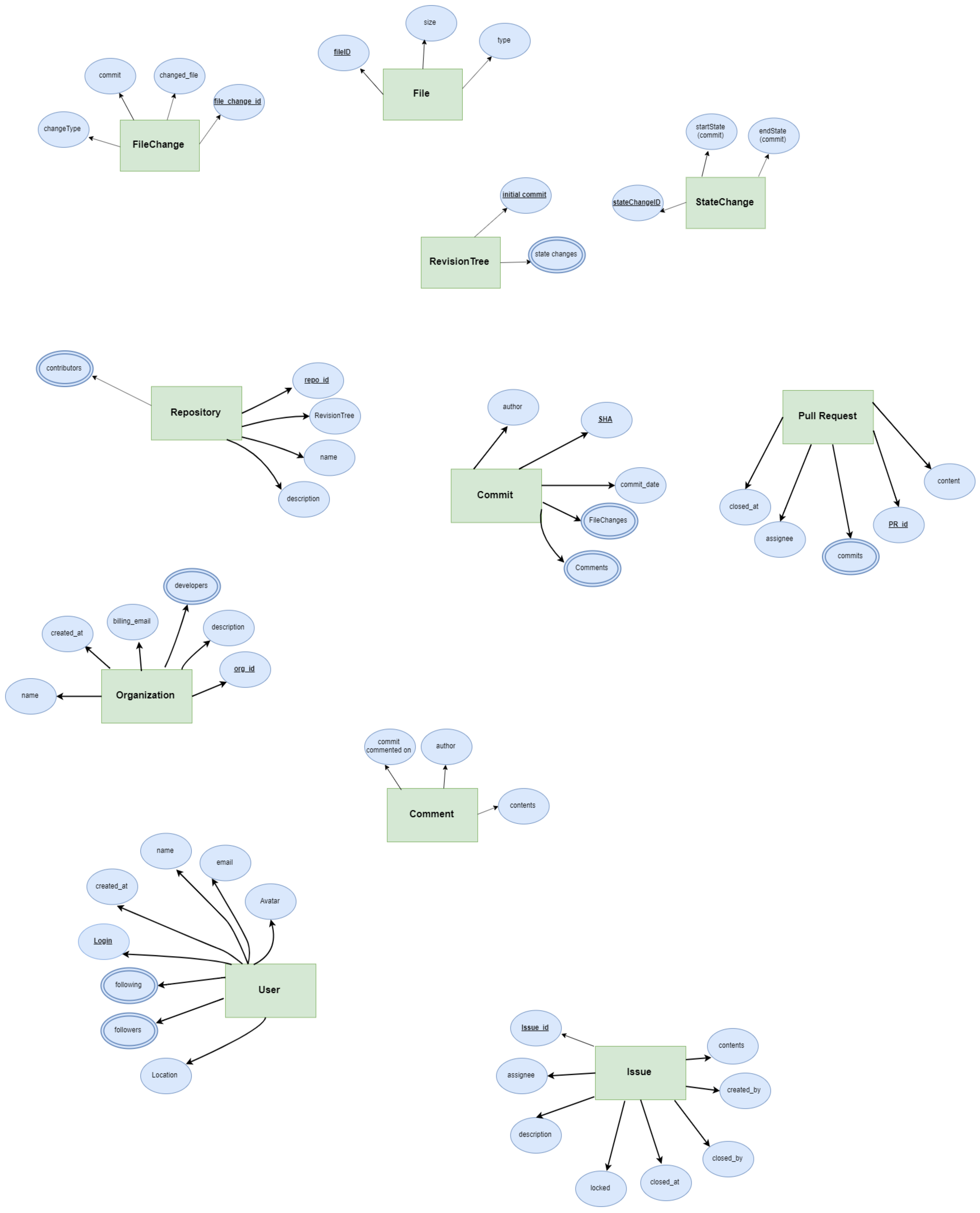
Functional Dependency Diagram:

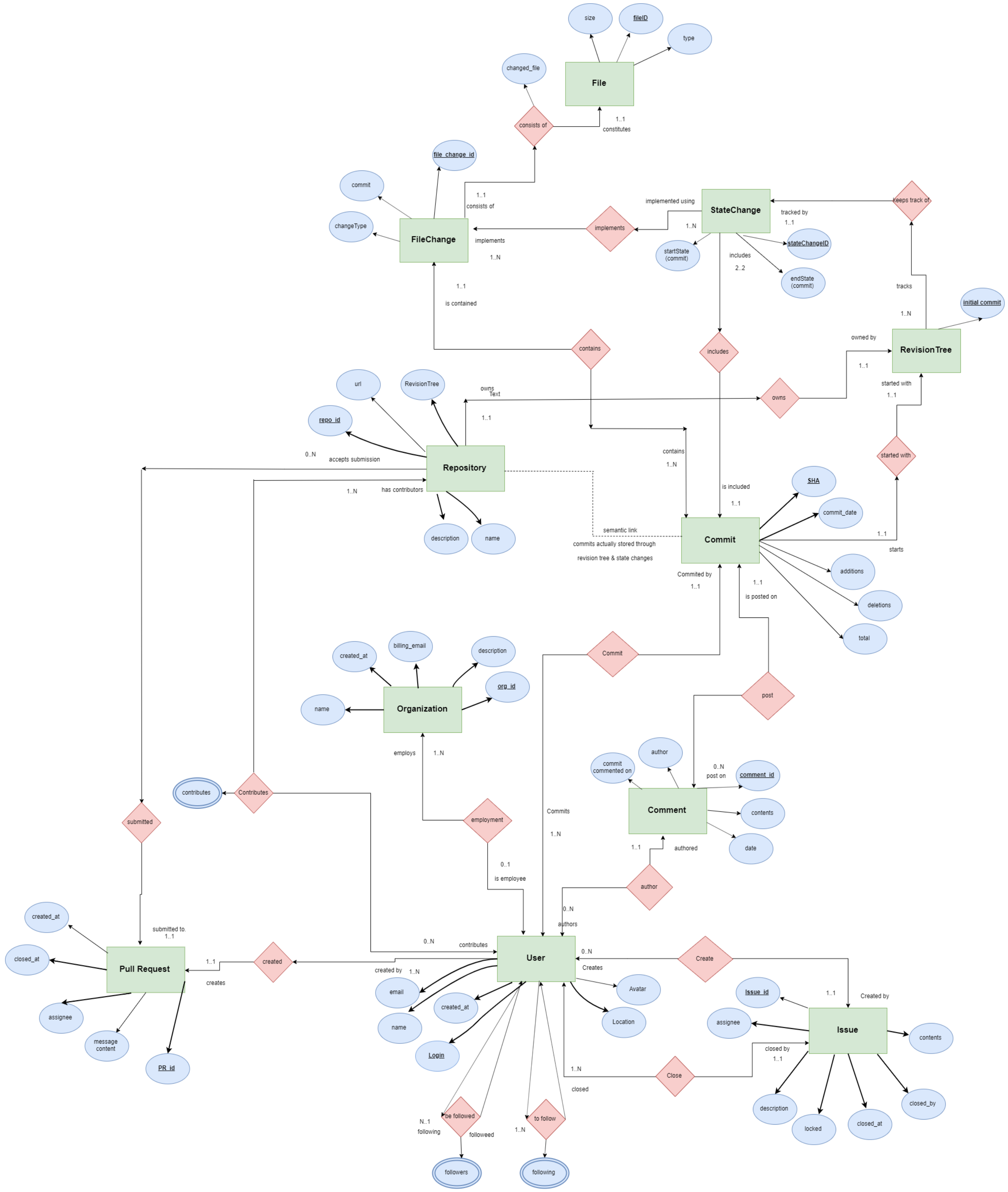
This confirms that the tables are normalised in Boyce-Codd normal form, where

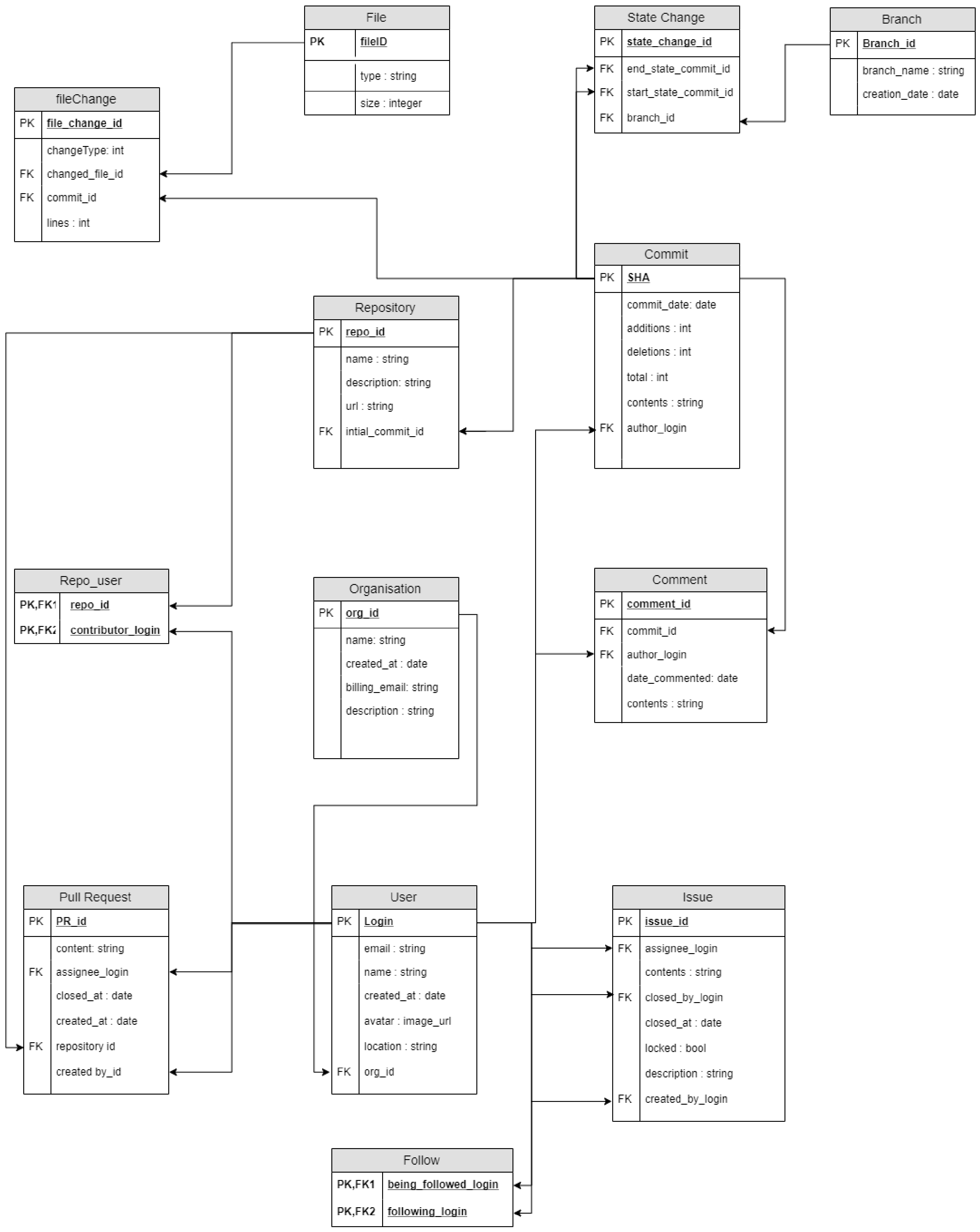
If a relational schema is in BCNF then all redundancy based on functional dependency has been removed, although other types of redundancy may still exist. A relational schema *R* is in Boyce–Codd normal form if and only if for every one of its dependencies $X \rightarrow Y$, at least one of the following conditions hold:

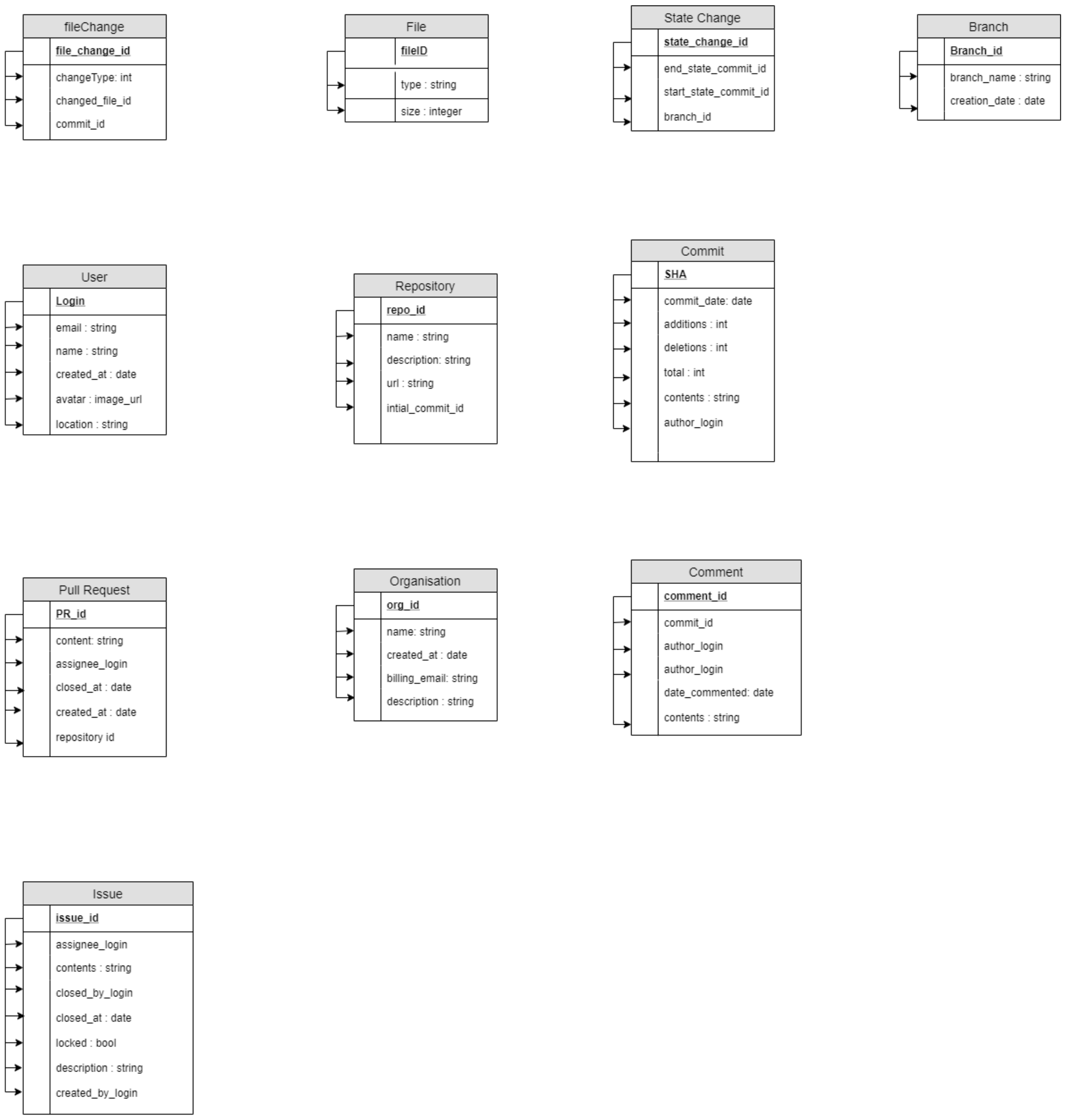
- $X \rightarrow Y$ is a trivial functional dependency ($Y \subseteq X$),
- *X* is a superkey for schema *R*.

Page 5


File
fileID
size
type
FileChange
commit
changed_file
file_change_id
changeType
StateChange
startState (commit)
endState (commit)
stateChangeID
RevisionTree
initial commit
state changes
Repository
contributors
repo_id
RevisionTree
name
description
Commit
author
SHA
commit_date
FileChanges
Comments
Pull Request
closed_at
assignee
commits
PR_id
content
Organization
developers
billing_email
created_at
description
org_id
name
Comment
commit commented on
author
contents
User
name
email
created_at
Avatar
Login
following
followers
Location
Issue
Issue_id
contents
assignee
created_by
description
closed_by
locked
closed_at


size
fileID
type
changed_file
File
consists of
1..1
constitutes
file_change_id
commit
changeType
FileChange
1..1
consists of
implements
1..N
implements
implemented using
StateChange
tracked by
1..1
keeps track of
1..N
stateChangeID
startState (commit)
endState (commit)
includes
2..2
tracks
1..N
initial commit
1..1
is contained
contains
includes
owned by
RevisionTree
1..1
started with
1..1
url
RevisionTree
repo_id
owns
Text
owns
1..1
Repository
0..N
accepts submission
1..N
has contributors
description
name
contains
1..N
is included
1..1
started with
SHA
commit_date
Commit
1..1
starts
semantic link
commits actually stored through
revision tree & state changes
additions
deletions
total
Commited by
1..1
1..1
is posted on
Commit
post
created_at
billing_email
description
org_id
name
Organization
employs
1..N
author
commit commented on
0..N
post on
comment_id
Comment
contents
date
1..1
authored
contributes
Contributes
submitted
employment
0..1
is employee
Commits
1..N
author
0..N
authors
created_at
closed_at
submitted to.
1..1
Pull Request
1..1
creates
created
0..N
contributes
User
0..N
Create
Creates
created by
1..N
email
created_at
Avatar
name
Location
Login
assignee
message content
PR_id
Issue_id
1..1
Created by
assignee
Issue
contents
closed by
1..1
Close
1..N
closed
description
locked
closed_at
closed_by
be followed
N..1
following
followeed
to follow
1..N
followers
following


File
PK fileID
type : string
size : integer
State Change
PK state_change_id
FK end_state_commit_id
FK start_state_commit_id
FK branch_id
Branch
PK Branch_id
branch_name : string
creation_date : date
fileChange
PK file_change_id
changeType: int
FK changed_file_id
FK commit_id
lines : int
Commit
PK SHA
commit_date: date
additions : int
deletions : int
total : int
contents : string
FK author_login
Repository
PK repo_id
name : string
description: string
url : string
FK intial_commit_id
Repo_user
PK,FK1 repo_id
PK,FK2 contributor_login
Organisation
PK org_id
name: string
created_at : date
billing_email: string
description : string
Comment
PK comment_id
FK commit_id
FK author_login
date_commented: date
contents : string
Pull Request
PK PR_id
content: string
FK assignee_login
closed_at : date
created_at : date
FK repository id
created by_id
User
PK Login
email : string
name : string
created_at : date
avatar : image_url
location : string
FK org_id
Issue
PK issue_id
FK assignee_login
contents : string
FK closed_by_login
closed_at : date
locked : bool
description : string
FK created_by_login
Follow
PK,FK1 being_followed_login
PK,FK2 following_login


fileChange
file_change_id
changeType: int
changed_file_id
commit_id
File
fileID
type : string
size : integer
State Change
state_change_id
end_state_commit_id
start_state_commit_id
branch_id
Branch
Branch_id
branch_name : string
creation_date : date
User
Login
email : string
name : string
created_at : date
avatar : image_url
location : string
Repository
repo_id
name : string
description: string
url : string
intial_commit_id
Commit
SHA
commit_date: date
additions : int
deletions : int
total : int
contents : string
author_login
Pull Request
PR_id
content: string
assignee_login
closed_at : date
created_at : date
repository id
Organisation
org_id
name: string
created_at : date
billing_email: string
description : string
Comment
comment_id
commit_id
author_login
author_login
date_commented: date
contents : string
Issue
issue_id
assignee_login
contents : string
closed_by_login
closed_at : date
locked : bool
description : string
created_by_login

Semantic Constraints:

Throughout the different tables there exist, not null and null constraints, since it is not trivial in many cases, you cannot in this system set all foreign keys not to be null, since in some cases this breaks the semantic meanings:

Examples of correct not null constraints:

> All of the primary keys must be not null, since they are used as the primary reference mechanism for all Links between the table objects.
>
> When creating a commit tuple, you need to specify the commit_date, hash, additions, deletions and total, And the author, since it doesn't make sense to add these fields later through an update for the meaning of Commit.

Examples of correct null constraints:

> Alternatively, in some other examples primarily the 'issue table', for the foreign keys, assignee_id and closed_by_id, It doesn't make sense to initialise these in the insert statement, since when an issue is opened, another user will eventually be assigned to it, or not at all, and the issue may not be closed over it's lifetime, so these fields will definitely need to be added in the future.
>
> Another example is the pull request table, It also contains an assignee_id foreign key, which needs to be updated at a later date once the PR is resolved.

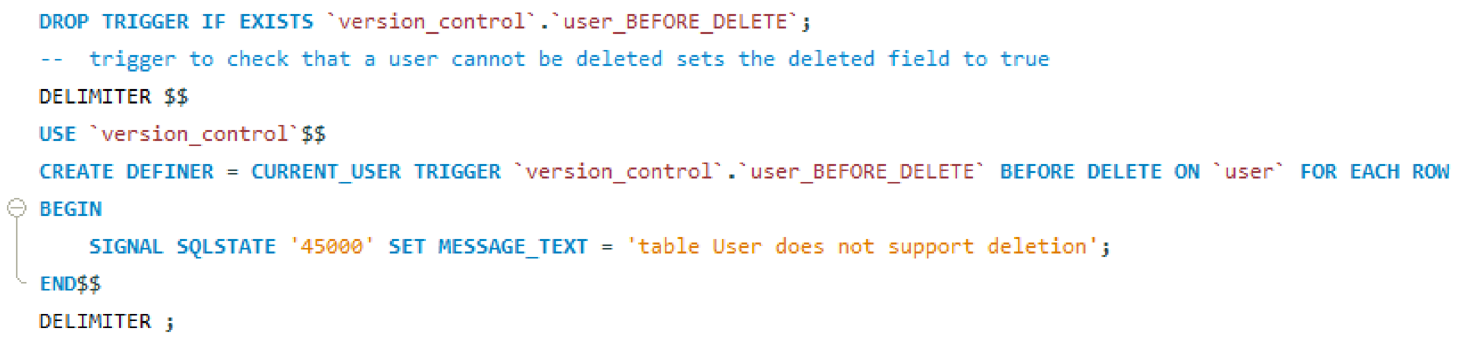
For other tables including the ones mentioned, I have some constraints on the foreign keys, to do with deletions and updates, for some they need to be restricted, as it doesn't semantically make sense to delete commits in general, since if a user is deleted, their commits still must stay in the repository, so that the commit tree remains stable, this issue is solved using a "deleted" field in the user, which is changed by a trigger, if someone tries to delete a user.

In other cases, it makes sense to cascade the changes made to a foreign user key on a update, or for example in the fileChange table, it makes sense to cascade the changes of files when inside a fileChange object, the commit_id foreign key is updated.

There are also constraints on some specific variables, for example a check that the change type remains only a integer between -1 and 1, which provides it the capabilities to act like a "three state variable".

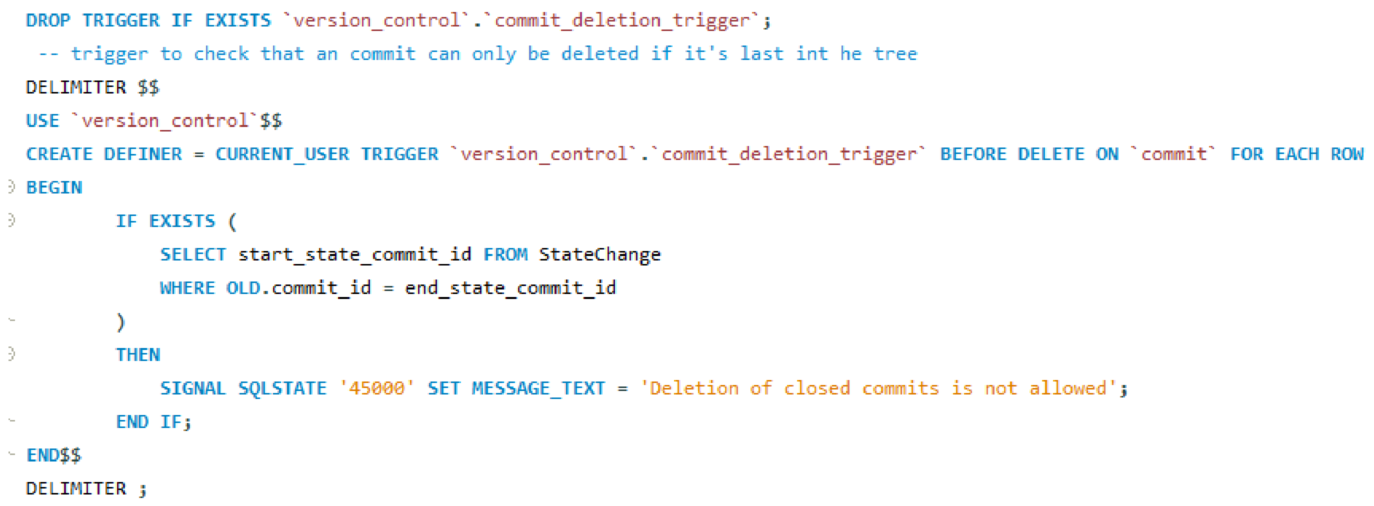
Triggers Implemented:

The first trigger defined, is used to check that a user cannot be deleted fully, instead the 'deleted field inside the user tuple is changed to signify this change.



```
DROP TRIGGER IF EXISTS `version_control`.`user_BEFORE_DELETE`;
--  trigger to check that a user cannot be deleted sets the deleted field to true
DELIMITER $$
USE `version_control`$$
CREATE DEFINER = CURRENT_USER TRIGGER `version_control`.`user_BEFORE_DELETE` BEFORE DELETE ON `user` FOR EACH ROW
BEGIN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'table User does not support deletion';
END$$
DELIMITER ;
```

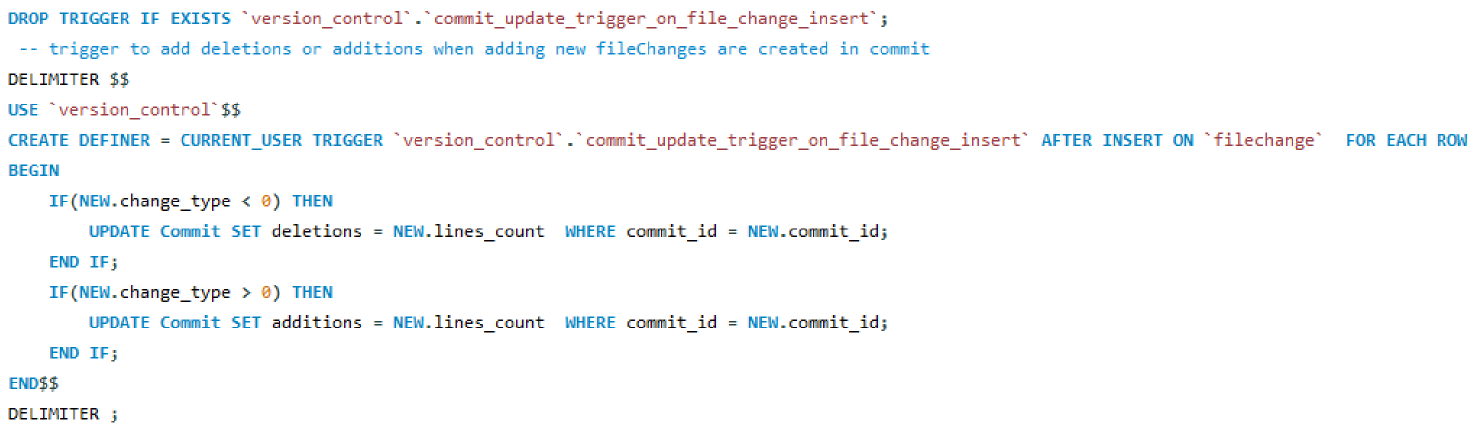
The second trigger defined, is to do with allowing the deletion or rollback of a commit only if it is a leaf node on the commit tree, if there are no parent links in this commit, meaning that for it's start state commit, there are no end state commits with the same id, it is allowed to be deleted.



```
DROP TRIGGER IF EXISTS `version_control`.`commit_deletion_trigger`;
 -- trigger to check that an commit can only be deleted if it's last int he tree
DELIMITER $$
USE `version_control`$$
CREATE DEFINER = CURRENT_USER TRIGGER `version_control`.`commit_deletion_trigger` BEFORE DELETE ON `commit` FOR EACH ROW
BEGIN
        IF EXISTS (
            SELECT start_state_commit_id FROM StateChange
            WHERE OLD.commit_id = end_state_commit_id
        )
        THEN
            SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Deletion of closed commits is not allowed';
        END IF;
END$$
DELIMITER ;
```

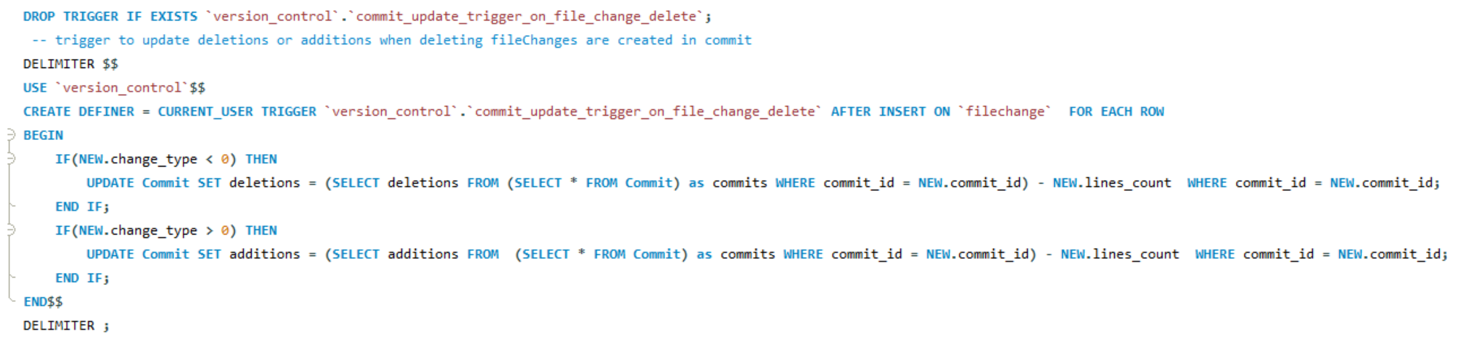
The next three triggers are variations of each other for insertion, deletion and update actions for updating the deletions additions, fields in the commit variable, when a new fileChange is created in a commit.

Update deletions and additions when adding new fileChange in commit:



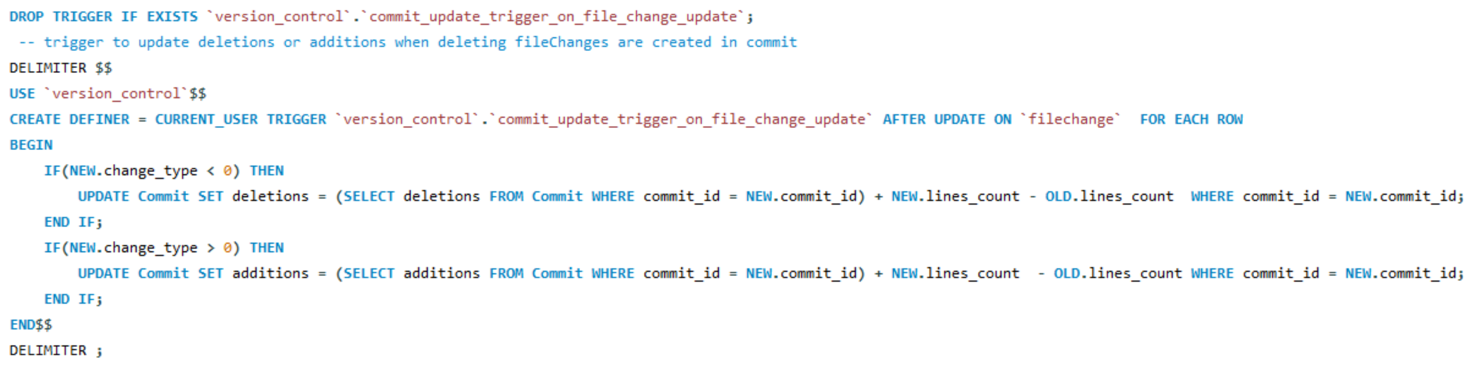
```
DROP TRIGGER IF EXISTS `version_control`.`commit_update_trigger_on_file_change_insert`;
 -- trigger to add deletions or additions when adding new fileChanges are created in commit
DELIMITER $$
USE `version_control`$$
CREATE DEFINER = CURRENT_USER TRIGGER `version_control`.`commit_update_trigger_on_file_change_insert` AFTER INSERT ON `filechange`  FOR EACH ROW
BEGIN
    IF(NEW.change_type < 0) THEN
        UPDATE Commit SET deletions = NEW.lines_count  WHERE commit_id = NEW.commit_id;
    END IF;
    IF(NEW.change_type > 0) THEN
        UPDATE Commit SET additions = NEW.lines_count  WHERE commit_id = NEW.commit_id;
    END IF;
END$$
DELIMITER ;
```

Update deletions and additions when deleting a fileChange in commit:



```
DROP TRIGGER IF EXISTS `version_control`.`commit_update_trigger_on_file_change_delete`;
 -- trigger to update deletions or additions when deleting fileChanges are created in commit
DELIMITER $$
USE `version_control`$$
CREATE DEFINER = CURRENT_USER TRIGGER `version_control`.`commit_update_trigger_on_file_change_delete` AFTER INSERT ON `filechange`  FOR EACH ROW
BEGIN
    IF(NEW.change_type < 0) THEN
        UPDATE Commit SET deletions = (SELECT deletions FROM (SELECT * FROM Commit) as commits WHERE commit_id = NEW.commit_id) - NEW.lines_count  WHERE commit_id = NEW.commit_id;
    END IF;
    IF(NEW.change_type > 0) THEN
        UPDATE Commit SET additions = (SELECT additions FROM  (SELECT * FROM Commit) as commits WHERE commit_id = NEW.commit_id) - NEW.lines_count  WHERE commit_id = NEW.commit_id;
    END IF;
END$$
DELIMITER ;
```

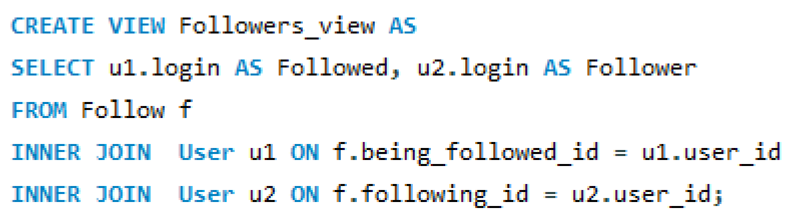
Update deletions and additions when updating a fileChange in commit:



```
DROP TRIGGER IF EXISTS `version_control`.`commit_update_trigger_on_file_change_update`;
 -- trigger to update deletions or additions when deleting fileChanges are created in commit
DELIMITER $$
USE `version_control`$$
CREATE DEFINER = CURRENT_USER TRIGGER `version_control`.`commit_update_trigger_on_file_change_update` AFTER UPDATE ON `filechange`  FOR EACH ROW
BEGIN
    IF(NEW.change_type < 0) THEN
        UPDATE Commit SET deletions = (SELECT deletions FROM Commit WHERE commit_id = NEW.commit_id) + NEW.lines_count - OLD.lines_count  WHERE commit_id = NEW.commit_id;
    END IF;
    IF(NEW.change_type > 0) THEN
        UPDATE Commit SET additions = (SELECT additions FROM Commit WHERE commit_id = NEW.commit_id) + NEW.lines_count  - OLD.lines_count WHERE commit_id = NEW.commit_id;
    END IF;
END$$
DELIMITER ;
```

Views Implemented:

There are two views implemented in this version, one is a view relating to finding all of the followers of a given user in the Followers table for the given user, this allows easily to do queries on the follower set of a user, and to check who is following a certain user.



```
CREATE VIEW Followers_view AS
SELECT u1.login AS Followed, u2.login AS Follower
FROM Follow f
INNER JOIN  User u1 ON f.being_followed_id = u1.user_id
INNER JOIN  User u2 ON f.following_id = u2.user_id;
```

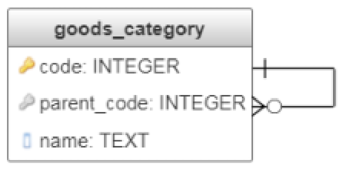
The second view is related to viewing all the child nodes of a parent node in the commit tree, I will explain about this operation in detail in the next section.

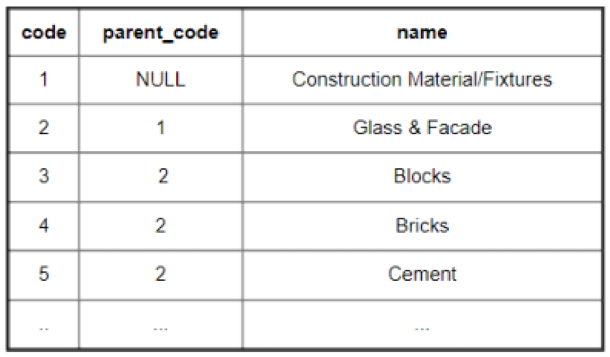
Commit Tree Structure Explanation & Procedures:

There are quite a few ways of implementing tree / hierarchical data structures and storing them efficiently for different operations in the database.

Some common implementations include:

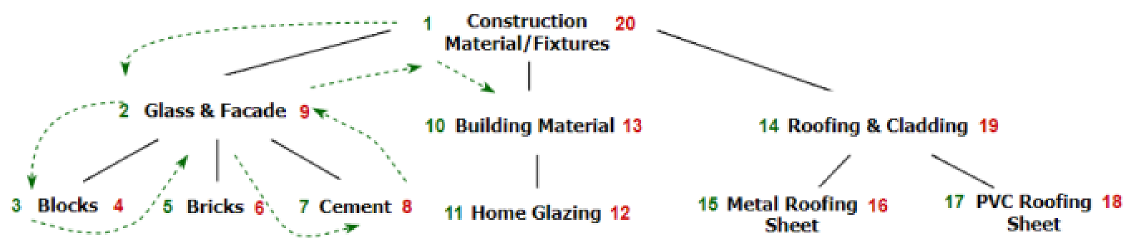
Adjacency lists in graph theory is a way to represent a graph by storing a list of neighbors (that is, adjacent vertices) for each vertex. For trees it is possible to store only the parent node, and then each of these lists contains a single value that can be stored in a database along with the vertex. This is one of the most popular representations, and also the most intuitive one: the table only has references to itself (fig. 2). Root node then contains an empty value (NULL) for their parent.

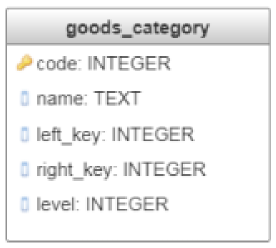




code	parent_code	name
1	NULL	Construction Material/Fixtures
2	1	Glass & Facade
3	2	Blocks
4	2	Bricks
5	2	Cement
..	...	...

Nested Sets the idea of this method is to store the prefix traversal of the tree. Here the tree root is visited first, then all the nodes of the left subtree are visited in the prefix order, then the nodes of the right subtree are visited once again in the prefix order. The order of traversal is stored in two additional fields: `left_key` and `right_key`. The `left_key` field contains the number of traversal steps before entering the subtree, while the `right_key` contains the number of steps before leaving the subtree. As a result, for each node its descendants have their numbers between the node’s numbers, independently of their depth level. This property allows writing queries without employing recursion.





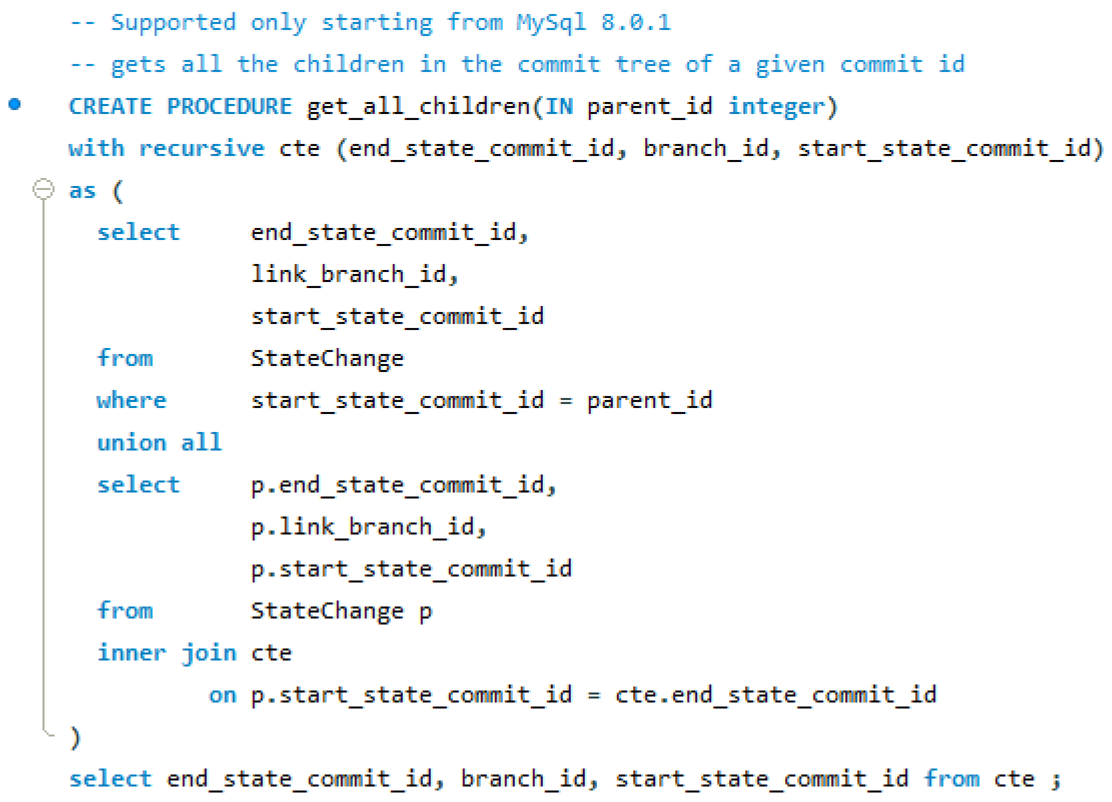
Lineage Columns
The idea of this method is to explicitly store the whole path from the root as a primary key for the node (fig. 6). Materialized paths is an elegant way of representing trees: every node has an intuitive identifier, individual parts of which have well-defined semantics. This property is very important for general-use classifications, including International Classification of Diseases (ICD), Universal Decimal Classification (UDC), used in scientific articles, PACS (Physics and Astronomy Classification Scheme). Queries for this method are concise, but not always efficient, since they involve substring matching.

These explanations are from https://bitworks.software/en/2017-10-20-storing-trees-in-rdbms.html

However in our case we decided to use, CTE recursion to implement the hierarchical commit datastructure, we Base the tree on links based on the end_state_commit_id and start_state_commit_id, and a branch_id, this makes the commits act like links, and the stateChange between them contains a list of changedFiles, in the structure, representing semantically, a change between two commit states in a branch.

A CTE is a common table expression, these features were added in mySQL 8.0, so it is important to keep this in mind if trying to implement a feature using them:

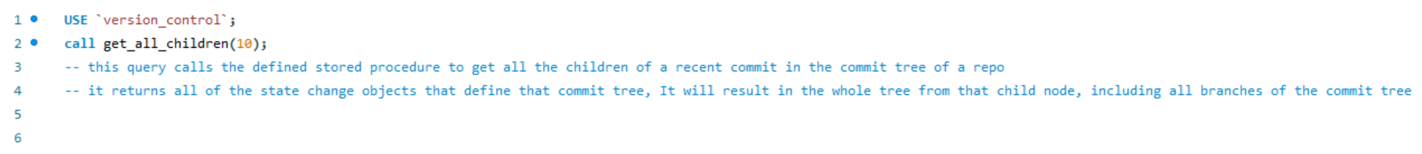
A CTE is defined using WITH clause.
Using WITH clause we can define more than one CTEs in a single statement.
A CTE can be referenced in the other CTEs that are part of same WITH clause but those CTEs should be defined earlier.
The scope of every CTE exist within the statement in which it is defined.
A **recursive CTE** is a subquery which refer to itself using its own name.



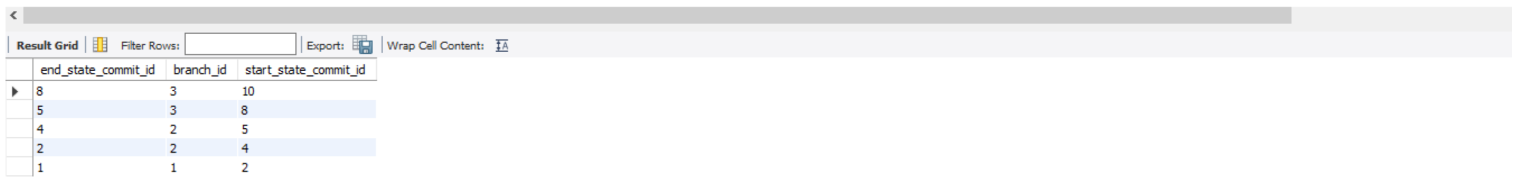
```
-- Supported only starting from MySql 8.0.1
-- gets all the children in the commit tree of a given commit id
CREATE PROCEDURE get_all_children(IN parent_id integer)
with recursive cte (end_state_commit_id, branch_id, start_state_commit_id)
as (
  select     end_state_commit_id,
             link_branch_id,
             start_state_commit_id
  from       StateChange
  where      start_state_commit_id = parent_id
  union all
  select     p.end_state_commit_id,
             p.link_branch_id,
             p.start_state_commit_id
  from       StateChange p
  inner join cte
          on p.start_state_commit_id = cte.end_state_commit_id
)
select end_state_commit_id, branch_id, start_state_commit_id from cte ;
```

This is the stored procedure that I wrote to walk the tree from a source node, it creates a view that shows the walked nodes from the source, for example you could input a commit_id and the procedure would return all the commits that were before this given commit, in the commit tree across any branches created before this parent

Example:



```
USE `version_control`;
call get_all_children(10);
-- this query calls the defined stored procedure to get all the children of a recent commit in the commit tree of a repo
-- it returns all of the state change objects that define that commit tree, It will result in the whole tree from that child node, including all branches of the commit tree
```



end_state_commit_id	branch_id	start_state_commit_id
8	3	10
5	3	8
4	2	5
2	2	4
1	1	2

I call the procedure on the 10th commit added in my small example, it returns all the state change rows, that represent the links between the commits in the tree.

Appendix:

SQL project file:

```
-- If creating this project in mySQL workbench
-- version must be over mysql 8.0 to use tree functionality

-- database schema must be called 'version_control' for triggers to work and be added


CREATE TABLE Organisation(
org_id Integer not null,
org_name varchar(30) not null,
created_at datetime not null DEFAULT NOW(),
billing_email varchar(40) not null,
descript varchar(140) not null,
PRIMARY KEY (org_id)
);




CREATE TABLE User(
user_id Integer not null,
login varchar(20) not null,
email VARCHAR(40) not null,
nickname VARCHAR(20) null,
avatar_url varchar(100),
location VARCHAR(30),
org_id Integer,
deleted boolean not null DEFAULT false,
PRIMARY KEY (user_id), -- add unique constraint for login
FOREIGN KEY (org_id) REFERENCES Organisation(org_id)
ON DELETE RESTRICT
ON UPDATE CASCADE,
CONSTRAINT UC_Login UNIQUE (login)
);



CREATE TABLE Commit (
commit_id Integer not null,
sha varchar(50) not null,
commit_date datetime not null DEFAULT NOW(),
additions int not null DEFAULT 0,
deletions int not null DEFAULT 0,
total int not null DEFAULT 0,
contents varchar(140) not null,
author_id Integer not null,
PRIMARY KEY (commit_id), -- add unique constraint for "sha"
FOREIGN KEY (author_id) REFERENCES User(user_id)
ON DELETE RESTRICT
 ON UPDATE CASCADE,
CONSTRAINT UC_commit_sha UNIQUE (sha)

);

CREATE TABLE Repository(
repo_id Integer not null,
repo_name varchar(20) not null,
descript varchar(140) not null,
url varchar(50) not null,
initial_commit_id Integer,
PRIMARY KEY (repo_id),
FOREIGN KEY (initial_commit_id) REFERENCES Commit(commit_id)
ON DELETE SET NULL
ON UPDATE CASCADE
);



CREATE TABLE PullRequest(
pr_id Integer not null,
created_by_id Integer not null,
content VARCHAR(140) not null,
assignee_id Integer,
closed_at datetime null,
created_at datetime not null DEFAULT NOW(),
repository_id Integer not null,
PRIMARY KEY (pr_id),
FOREIGN KEY (assignee_id) REFERENCES User(user_id)
ON DELETE RESTRICT
ON UPDATE CASCADE,
FOREIGN KEY (repository_id) REFERENCES Repository(repo_id)
ON DELETE RESTRICT
ON UPDATE CASCADE,
FOREIGN KEY (created_by_id) REFERENCES User(user_id)
ON DELETE RESTRICT
ON UPDATE CASCADE

);


CREATE TABLE Follow(
being_followed_id Integer not null,
following_id Integer not null,
PRIMARY KEY (being_followed_id,following_id),
FOREIGN KEY (being_followed_id) REFERENCES User(user_id)
ON DELETE RESTRICT
ON UPDATE CASCADE,
FOREIGN KEY (following_id) REFERENCES User(user_id)
ON DELETE RESTRICT
ON UPDATE CASCADE
);

CREATE TABLE Issue(
issue_id Integer not null,
descript VARCHAR(140) not null,
assignee_id Integer,
closed_at datetime,
created_by_id Integer,
created_at datetime DEFAULT NOW(),
closed_by_id Integer,
repository_id Integer not null,
PRIMARY KEY (issue_id),
FOREIGN KEY (assignee_id) REFERENCES User(user_id)
ON DELETE RESTRICT
ON UPDATE CASCADE,
FOREIGN KEY (closed_by_id) REFERENCES User(user_id)
ON DELETE RESTRICT
ON UPDATE CASCADE,
FOREIGN KEY (created_by_id) REFERENCES User(user_id)
ON DELETE RESTRICT
ON UPDATE CASCADE
);


CREATE TABLE Comment(
comment_id Integer not null,
commit_id Integer not null,
author_id Integer not null,
date_posted datetime not null DEFAULT NOW(),
contents varchar(140) not null,
PRIMARY KEY (comment_id),
FOREIGN KEY (author_id) REFERENCES User(user_id)
ON DELETE RESTRICT
ON UPDATE CASCADE,
FOREIGN KEY (commit_id) REFERENCES Commit(commit_id)
ON DELETE CASCADE
ON UPDATE CASCADE
);


CREATE TABLE Repo_User(
repo_id Integer not null,
contributor_id Integer not null,
PRIMARY KEY (repo_id,contributor_id),
FOREIGN KEY (repo_id) REFERENCES Repository(repo_id)
ON DELETE RESTRICT
ON UPDATE CASCADE,
FOREIGN KEY (contributor_id) REFERENCES User(user_id)
ON DELETE RESTRICT
ON UPDATE CASCADE

);


CREATE TABLE Branch (
branch_id Integer not null,
branch_name varchar(20) not null,
creation_date datetime not null DEFAULT NOW(),
PRIMARY KEY (branch_id)
);

CREATE TABLE StateChange (
state_change_id Integer not null,
end_state_commit_id Integer not null,
start_state_commit_id Integer not null,
link_branch_id Integer not null,
PRIMARY KEY (state_change_id),
FOREIGN KEY (end_state_commit_id) REFERENCES Commit(commit_id)
ON DELETE CASCADE
ON UPDATE CASCADE,
FOREIGN KEY (start_state_commit_id) REFERENCES Commit(commit_id)
ON DELETE RESTRICT
ON UPDATE CASCADE,
FOREIGN KEY (link_branch_id) REFERENCES Branch(branch_id)
ON DELETE RESTRICT
ON UPDATE CASCADE

);


CREATE TABLE File(
file_id Integer not null,
size Integer not null,
file_path varchar(50) not null,
PRIMARY KEY (file_id)
);
```

```
CREATE TABLE FileChange (
file_change_id Integer not null,
change_type Integer not null,
changed_file_id Integer not null,
commit_id Integer not null,
lines_count Integer not null,
PRIMARY KEY(file_change_id),
FOREIGN KEY (changed_file_id) REFERENCES file(file_id)
 ON DELETE CASCADE
 ON UPDATE CASCADE,
FOREIGN KEY (commit_id) REFERENCES Commit(commit_id)
 ON DELETE CASCADE
 ON UPDATE CASCADE,
CONSTRAINT CHK_change_type CHECK (change_type >= -1 AND change_type <= 1)
);


DROP TRIGGER IF EXISTS `version_control`.`user_BEFORE_DELETE`;
--  trigger to check that a user cannot be deleted sets the deleted field to true
DELIMITER $$
USE `version_control`$$
CREATE DEFINER = CURRENT_USER TRIGGER `version_control`.`user_BEFORE_DELETE` BEFORE DELETE
ON `user` FOR EACH ROW
BEGIN
	SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'table User does not support deletion';
END$$
DELIMITER ;


DROP TRIGGER IF EXISTS `version_control`.`on_closed_issue_delete`;
 -- trigger to check that an issue can only be deleted if it's open
DELIMITER $$
USE `version_control`$$
CREATE DEFINER = CURRENT_USER TRIGGER `version_control`.`on_closed_issue_delete` BEFORE DELETE
ON `issue` FOR EACH ROW
BEGIN
	IF (OLD.closed_by_id IS NOT NULL) THEN
		SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Deletion of closed Issues is not allowed';
	END IF;
END$$
DELIMITER ;
```

```
DROP TRIGGER IF EXISTS `version_control`.`commit_deletion_trigger`;
 -- trigger to check that an commit can only be deleted if it's last int he tree
DELIMITER $$
USE `version_control`$$
CREATE DEFINER = CURRENT_USER TRIGGER `version_control`.`commit_deletion_trigger` BEFORE DELETE
ON `commit` FOR EACH ROW
BEGIN
            IF EXISTS (
                  SELECT start_state_commit_id FROM StateChange
      WHERE OLD.commit_id = end_state_commit_id
    )
    THEN
      SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Deletion of closed commits is not allowed';
            END IF;
END$$
DELIMITER ;



DROP TRIGGER IF EXISTS `version_control`.`commit_update_trigger_on_file_change_insert`;
 -- trigger to add deletions or additions when adding new fileChanges are created in commit
DELIMITER $$
USE `version_control`$$
CREATE DEFINER = CURRENT_USER TRIGGER
`version_control`.`commit_update_trigger_on_file_change_insert` AFTER INSERT ON `filechange`  FOR EACH
ROW
BEGIN
  IF(NEW.change_type < 0) THEN
            UPDATE Commit SET deletions = NEW.lines_count  WHERE commit_id = NEW.commit_id;
      END IF;
      IF(NEW.change_type > 0) THEN
            UPDATE Commit SET additions = NEW.lines_count  WHERE commit_id = NEW.commit_id;
      END IF;
END$$
DELIMITER ;
```

```
DROP TRIGGER IF EXISTS `version_control`.`commit_update_trigger_on_file_change_delete`;
 -- trigger to update deletions or additions when deleting fileChanges are created in commit
DELIMITER $$
USE `version_control`$$
CREATE DEFINER = CURRENT_USER TRIGGER
`version_control`.`commit_update_trigger_on_file_change_delete` AFTER INSERT ON `filechange`  FOR EACH
ROW
BEGIN
   IF(NEW.change_type < 0) THEN
                    UPDATE Commit SET deletions = (SELECT deletions FROM (SELECT * FROM Commit) as
commits WHERE commit_id = NEW.commit_id) - NEW.lines_count  WHERE commit_id = NEW.commit_id;
          END IF;
          IF(NEW.change_type > 0) THEN
                    UPDATE Commit SET additions = (SELECT additions FROM  (SELECT * FROM Commit) as
commits WHERE commit_id = NEW.commit_id) - NEW.lines_count  WHERE commit_id = NEW.commit_id;
          END IF;
END$$
DELIMITER ;
```

```
DROP TRIGGER IF EXISTS `version_control`.`commit_update_trigger_on_file_change_update`;
 -- trigger to update deletions or additions when deleting fileChanges are created in commit
DELIMITER $$
USE `version_control`$$
CREATE DEFINER = CURRENT_USER TRIGGER
`version_control`.`commit_update_trigger_on_file_change_update` AFTER UPDATE ON `filechange`  FOR EACH
ROW
BEGIN
   IF(NEW.change_type < 0) THEN
                    UPDATE Commit SET deletions = (SELECT deletions FROM Commit WHERE commit_id =
NEW.commit_id) + NEW.lines_count - OLD.lines_count  WHERE commit_id = NEW.commit_id;
          END IF;
          IF(NEW.change_type > 0) THEN
                    UPDATE Commit SET additions = (SELECT additions FROM Commit WHERE commit_id =
NEW.commit_id) + NEW.lines_count  - OLD.lines_count WHERE commit_id = NEW.commit_id;
          END IF;
END$$
DELIMITER ;
```

```
CREATE VIEW Followers_view AS
SELECT u1.login AS Followed, u2.login AS Follower
FROM Follow f
INNER JOIN  User u1 ON f.being_followed_id = u1.user_id
INNER JOIN  User u2 ON f.following_id = u2.user_id;

-- Supported only starting from MySql 8.0.1
-- gets all the children in the commit tree of a given commit id
CREATE PROCEDURE get_all_children(IN parent_id integer)
with recursive cte (end_state_commit_id, branch_id, start_state_commit_id)
as (
  select     end_state_commit_id,
                                 link_branch_id,
             start_state_commit_id
  from       StateChange
  where      start_state_commit_id = parent_id
  union all
  select     p.end_state_commit_id,
             p.link_branch_id,
             p.start_state_commit_id
  from       StateChange p
  inner join cte
          on p.start_state_commit_id = cte.end_state_commit_id
)
select end_state_commit_id, branch_id, start_state_commit_id from cte ;
```

```
INSERT INTO Organisation (org_id,org_name,created_at,billing_email,descript)
VALUES (1,'anton_org',NOW(),'yamkovoa@tcd.ie','anton organisation');

INSERT INTO Organisation (org_id,org_name,created_at,billing_email,descript)
VALUES (2,'kamil_org',NOW(),'kamilprz@tcd.ie','kamil organisation');


INSERT INTO User(user_id,login,email,nickname,avatar_url,location,org_id)
VALUES (1,'AntonYamkovoy','yamkovoa@tcd.ie','Anton','image_anton','Dublin',1);

INSERT INTO User(user_id,login,email,nickname,avatar_url,location,org_id)
VALUES (2,'KamilPrz','kamilprz@tcd.ie','Kamil','image_kamil','Dublin',2);

INSERT INTO User(user_id,login,email,nickname,avatar_url,location,org_id)
VALUES (3,'AndreyYamkovoy','andrey@tcd.ie','Andrey','image_andrey','Dublin',1);

INSERT INTO User(user_id,login,email,nickname,avatar_url,location)
VALUES (4,'yungene','eugene@tcd.ie','Eugene','image_eugene','Dublin');

INSERT INTO COMMIT(commit_id,sha,additions,deletions,total,contents,author_id)
VALUES (1,'0001A',5,3,2,'first commit anton',1);

INSERT INTO COMMIT(commit_id,sha,additions,deletions,total,contents,author_id)
VALUES (2,'0001B',10,3,7,'first commit kamil',2);

INSERT INTO COMMIT(commit_id,sha,additions,deletions,total,contents,author_id)
VALUES (3,'0001C',50,50,0,'first commit andrey',3);

INSERT INTO COMMIT(commit_id,sha,additions,deletions,total,contents,author_id)
VALUES (4,'0001D',90,40,50,'second commit kamil',2);

INSERT INTO COMMIT(commit_id,sha,additions,deletions,total,contents,author_id)
VALUES (5,'0001E',5,10,-5,'third commit kamil',2);

INSERT INTO COMMIT(commit_id,sha,additions,deletions,total,contents,author_id)
VALUES (6,'0001F',10,12,-2,'second commit anton',1);

INSERT INTO COMMIT(commit_id,sha,additions,deletions,total,contents,author_id)
VALUES (7,'0001G',5,3,2,'second commit andrey',3);

INSERT INTO COMMIT(commit_id,sha,additions,deletions,total,contents,author_id)
VALUES (8,'0001H',1,1,0,'third commit anton',1);

INSERT INTO COMMIT(commit_id,sha,additions,deletions,total,contents,author_id)
VALUES (9,'0001I',52,3,49,'third commit andrey',3);

INSERT INTO COMMIT(commit_id,sha,additions,deletions,total,contents,author_id)
VALUES (10,'0001J',1,3,-2,'fourth commit anton',1);

INSERT INTO COMMIT(commit_id,sha,additions,deletions,total,contents,author_id)
```

```
VALUES (11,'0001K',10,5,5,'first commit eugene',4);



INSERT INTO COMMIT(commit_id,sha,additions,deletions,total,contents,author_id)
VALUES (12,'0001L',20,1,19,'second commit eugene',4);

INSERT INTO Repository(repo_id,repo_name,descript,url,initial_commit_id)
VALUES (1,'repo1','Anton Kamil Andrey repo','github/repo1',1);

INSERT INTO Repository(repo_id,repo_name,descript,url,initial_commit_id)
VALUES (2,'repo2','Eugene repo','github/repo2',11);

INSERT INTO PullRequest(pr_id,created_by_id,content,repository_id)
VALUES (1,3,'andreys pull request 1',1);

INSERT INTO PullRequest(pr_id,created_by_id,content,repository_id)
VALUES (2,4,'eugenes pull request 1',2);

INSERT INTO Follow(being_followed_id,following_id)
VALUES (1,2); -- kamil following anton

INSERT INTO Follow(being_followed_id,following_id)
VALUES (1,3); -- andrey following anton

INSERT INTO Follow(being_followed_id,following_id)
VALUES (1,4); -- yungene following anton

INSERT INTO Follow(being_followed_id,following_id)
VALUES (2,4); -- yungene following kamil

INSERT INTO Follow(being_followed_id,following_id)
VALUES (2,1); -- anton following kamil

INSERT INTO Follow(being_followed_id,following_id)
VALUES (4,1); -- anton following yungene

INSERT INTO Follow(being_followed_id,following_id)
VALUES (4,2); -- kamil following yungene

INSERT INTO Issue(issue_id,descript,created_by_id,repository_id)
VALUES (1,'anton issue in repo1', 1,1);

INSERT INTO Issue(issue_id,descript,created_by_id,repository_id)
VALUES (2,'eugene issue in repo2', 4,2);

INSERT INTO Comment(comment_id,commit_id,author_id,contents)
VALUES (1,10,1,'anton comment on commit 10');

INSERT INTO Comment(comment_id,commit_id,author_id,contents)
VALUES (2,2,2,'kamil comment on commit 2');


INSERT INTO Comment(comment_id,commit_id,author_id,contents)
VALUES (3,11,4,'eugene comment on commit 11');
```

```
INSERT INTO Comment(comment_id,commit_id,author_id,contents)
VALUES (4,12,4,'eugene comment on commit 12');


INSERT INTO Repo_User(repo_id,contributor_id)
VALUES (1,1); -- anton in repo 1

INSERT INTO Repo_User(repo_id,contributor_id)
VALUES (1,2); -- kamil in repo 1

INSERT INTO Repo_User(repo_id,contributor_id)
VALUES (1,3); -- andrey in repo 1

INSERT INTO Repo_User(repo_id,contributor_id)
VALUES (2,4); -- eugene in repo 2



INSERT INTO Branch(branch_id,branch_name)
VALUES (1,'repo1 master');

INSERT INTO Branch(branch_id,branch_name)
VALUES (2,'repo1 branch1');

INSERT INTO Branch(branch_id,branch_name)
VALUES (3,'repo1 branch2');

INSERT INTO Branch(branch_id,branch_name)
VALUES (4,'repo2 master');



-- master branch in repo1
INSERT INTO StateChange(state_change_id,end_state_commit_id,start_state_commit_id,link_branch_id)
VALUES (1,1,2,1);

INSERT INTO StateChange(state_change_id,end_state_commit_id,start_state_commit_id,link_branch_id)
VALUES (2,2,3,1);

INSERT INTO StateChange(state_change_id,end_state_commit_id,start_state_commit_id,link_branch_id)
VALUES (3,3,6,1);

INSERT INTO StateChange(state_change_id,end_state_commit_id,start_state_commit_id,link_branch_id)
VALUES (4,6,9,1);

-- branch1 in repo1

INSERT INTO StateChange(state_change_id,end_state_commit_id,start_state_commit_id,link_branch_id)
VALUES (5,2,4,2);

INSERT INTO StateChange(state_change_id,end_state_commit_id,start_state_commit_id,link_branch_id)
VALUES (6,4,5,2);

INSERT INTO StateChange(state_change_id,end_state_commit_id,start_state_commit_id,link_branch_id)
```

```
VALUES (7,5,7,2);


-- branch2 in repo1

INSERT INTO StateChange(state_change_id,end_state_commit_id,start_state_commit_id,link_branch_id)
VALUES (8,5,8,3);

INSERT INTO StateChange(state_change_id,end_state_commit_id,start_state_commit_id,link_branch_id)
VALUES (9,8,10,3);

-- master in repo2

INSERT INTO StateChange(state_change_id,end_state_commit_id,start_state_commit_id,link_branch_id)
VALUES (10,11,12,4);




INSERT INTO File(file_id,size,file_path)
VALUES (1,10,'file1');

INSERT INTO File(file_id,size,file_path)
VALUES (2,100,'file2');

INSERT INTO File(file_id,size,file_path)
VALUES (3,50,'file3');

INSERT INTO File(file_id,size,file_path)
VALUES (4,12,'file4');

INSERT INTO File(file_id,size,file_path)
VALUES (5,100,'file5');

INSERT INTO File(file_id,size,file_path)
VALUES (6,50,'file6');

INSERT INTO File(file_id,size,file_path)
VALUES (7,12,'file7');

INSERT INTO File(file_id,size,file_path)
VALUES (8,100,'file8');

INSERT INTO File(file_id,size,file_path)
VALUES (9,50,'file9');

INSERT INTO File(file_id,size,file_path)
VALUES (10,4,'file10');

INSERT INTO File(file_id,size,file_path)
VALUES (11,90,'file11');

INSERT INTO File(file_id,size,file_path)
VALUES (12,60,'file12');
```

```
INSERT INTO File(file_id,size,file_path)
VALUES (13,15,'file13');

INSERT INTO File(file_id,size,file_path)
VALUES (14,100,'file14');

INSERT INTO File(file_id,size,file_path)
VALUES (15,50,'file15');

INSERT INTO File(file_id,size,file_path)
VALUES (16,12,'file16');



INSERT INTO FileChange(file_change_id,change_type,changed_file_id,commit_id,lines_count)
VALUES (1,1,1,1,5);

INSERT INTO FileChange(file_change_id,change_type,changed_file_id,commit_id,lines_count)
VALUES (2,-1,2,2,10);

INSERT INTO FileChange(file_change_id,change_type,changed_file_id,commit_id,lines_count)
VALUES (3,1,3,3,50);

INSERT INTO FileChange(file_change_id,change_type,changed_file_id,commit_id,lines_count)
VALUES (4,-1,4,4,12);

INSERT INTO FileChange(file_change_id,change_type,changed_file_id,commit_id,lines_count)
VALUES (5,-1,5,4,10);

INSERT INTO FileChange(file_change_id,change_type,changed_file_id,commit_id,lines_count)
VALUES (6,1,6,5,100);

INSERT INTO FileChange(file_change_id,change_type,changed_file_id,commit_id,lines_count)
VALUES (7,1,7,5,11);

INSERT INTO FileChange(file_change_id,change_type,changed_file_id,commit_id,lines_count)
VALUES (8,-1,8,7,20);

INSERT INTO FileChange(file_change_id,change_type,changed_file_id,commit_id,lines_count)
VALUES (9,1,9,10,30);

INSERT INTO FileChange(file_change_id,change_type,changed_file_id,commit_id,lines_count)
VALUES (10,1,10,6,20);

INSERT INTO FileChange(file_change_id,change_type,changed_file_id,commit_id,lines_count)
VALUES (11,1,11,9,20);

INSERT INTO FileChange(file_change_id,change_type,changed_file_id,commit_id,lines_count)
VALUES (12,1,12,9,50);

INSERT INTO FileChange(file_change_id,change_type,changed_file_id,commit_id,lines_count)
VALUES (13,1,13,11,20);
```

```
INSERT INTO FileChange(file_change_id,change_type,changed_file_id,commit_id,lines_count)
VALUES (14,1,14,11,90);

INSERT INTO FileChange(file_change_id,change_type,changed_file_id,commit_id,lines_count)
VALUES (15,-1,15,11,20);

INSERT INTO FileChange(file_change_id,change_type,changed_file_id,commit_id,lines_count)
VALUES (16,1,16,12,40);
```

Example calling tree:

```
USE `version_control`;
call get_all_children(10);
-- this query calls the defined stored procedure to get all the children of a recent commit in the commit tree of a repo
-- it returns all of the state change objects that define that commit tree, It will result in the whole tree from that child
node, including all branches of the commit tree
```

Example calling sql queries:

```
-- SELECT * FROM Commit WHERE author_id = 2
-- Example query of getting all commits of a given user


-- SELECT * FROM User WHERE user_id  IN (SELECT contributor_id FROM Repo_User WHERE repo_id = 1)
 -- Example query to get all the users in a specific repository

  SELECT * FROM User WHERE user_id  IN (SELECT author_id FROM Comment)
 -- Example query : getting all users who post comments
```

Instructions for running project:

Make sure that the project is running in mySQL 8.0
Make sure that the project schema is called ‘version_control’