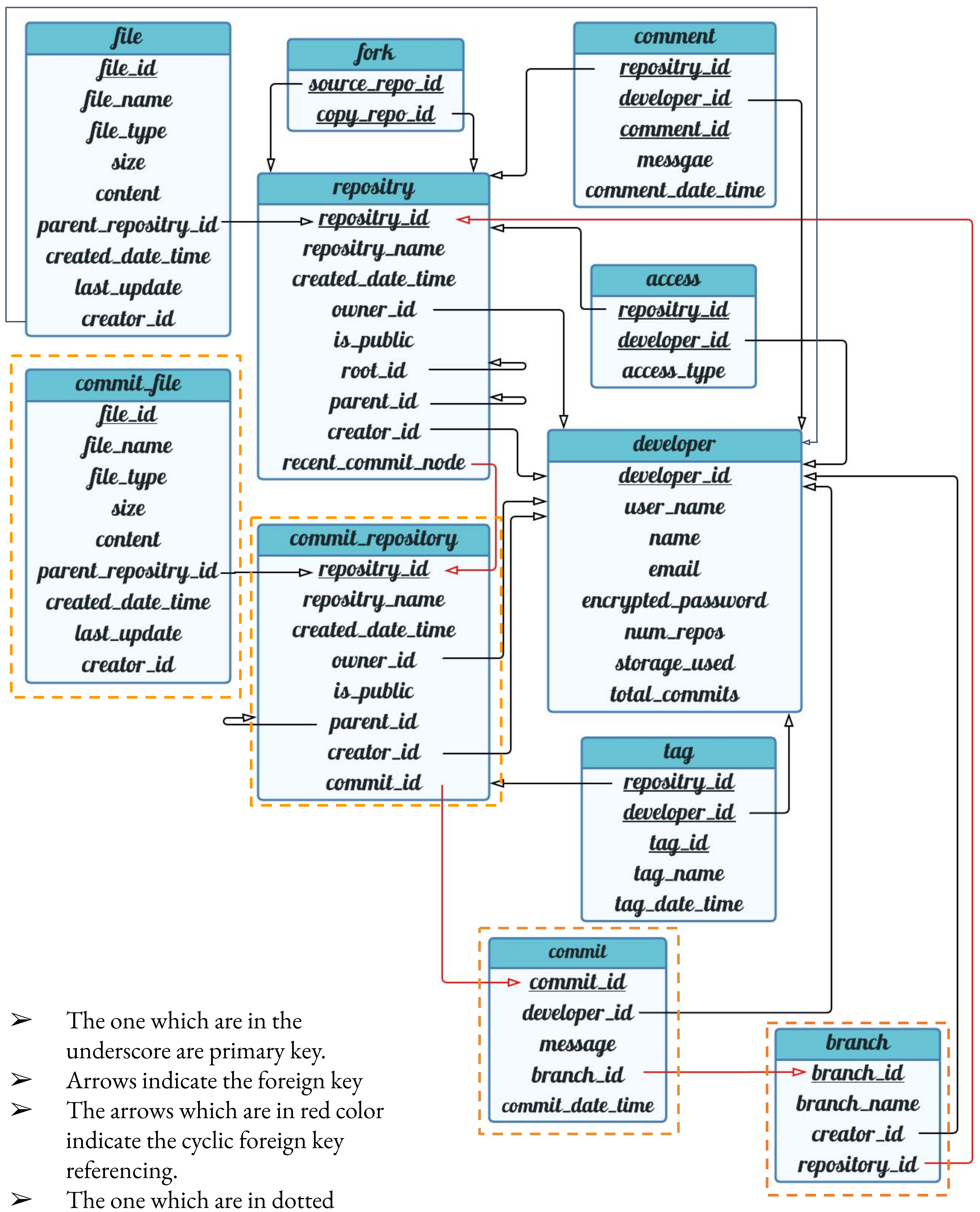
PROJECT REPORT

OCTOPUS

A CLOUD BASED PLATFORM

RELATIONAL DESIGN



- ➢ The one which are in the underscore are primary key.
- ➢ Arrows indicate the foreign key
- ➢ The arrows which are in red color indicate the cyclic foreign key referencing.
- ➢ The one which are in dotted boundaries comes under commit area where the sophisticated commit process uses commit area.

DEFINITIONS

➢ **super user/developer:** A developer who has access to everything in the database is super user. This is created by database administrator. Username of the super user is **_octopus_.** The need of this super user is specified in this page.

➢ **repository tree:**
Repository tree is a graphical representation of all the data (repositories, files) of a developer. (rooted directed tree).
Below are some invariants maintained in the tree.

- All the child repositories of a repository must have different name
- All the child files of a repository must have different (name, file_type).
- NOTE:
    - A repository and a file can be siblings with same name.
    - Two files can be siblings and have same name if they have different file types

---

➢ **universal repository:**
As soon as a developer account is created, **@username** repository is created by the **_octopus_** , which is hereafter called universal repository.
Universal repository contains the description.
@_sandeep_ is automatically created by the platform(i.e, when _sandeep_ user account is created, @_sandeep_ repository is also created) and serves as the repository under which other repositories are organized. It acts as a central hub for _sandeep_ projects and facilitates the management and organization of related repositories and files.

➢ **root:**
Root is the repository whose parent is universal repository.

➢ **parent:**
A repository r is in repository n, then n is the parent of r.

➢ **owner:** The owner of a repository is the individual or entity who originally created the repository or was assigned ownership rights by the creator. As **the** owner, they have full administrative control over the repository, which includes the following abilities

Managing Access Permissions
Setting Repository Settings
Merging Pull Request

➢ **collaborator:**
A collaborator refers to an individual or user account who is granted access to a repository by its owner or administrators. Collaborators typically have specific permissions to contribute to the repository, such as editing files, creating branches, or merging changes. They work alongside the repository owner and other collaborators to contribute to projects, implement changes, and collaborate on code development or other tasks within the repository.

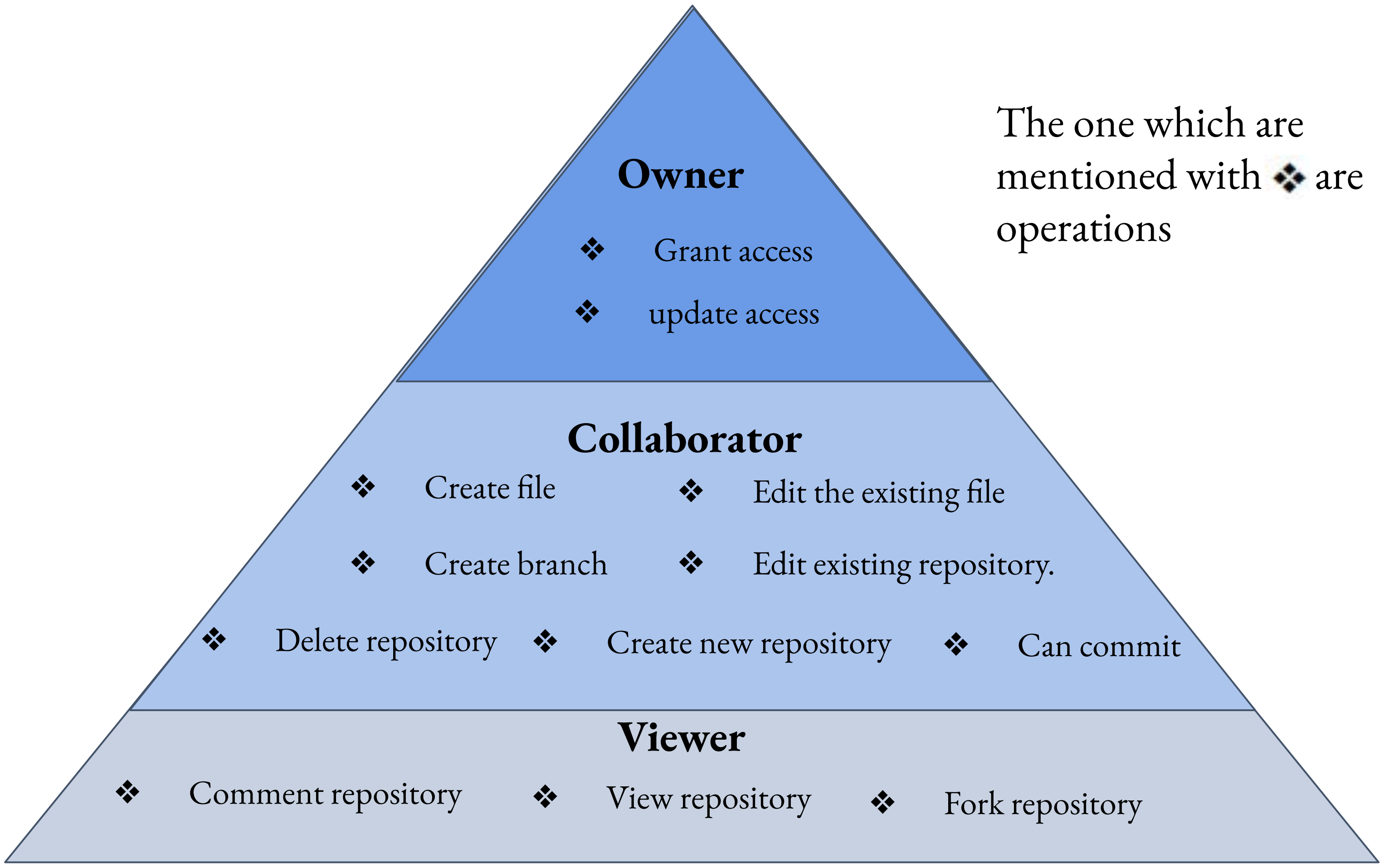
➢ **viewer:**

As a viewer, developer *d* can access the contents of the repository, view files, browse commits, branches, and pull requests.
However, they typically do not have permission to make changes to the repository's content, such as pushing new commits, creating branches, or merging pull requests.
A developer d is viewer of repository r if the tuple (d.developer_id, r.repository_id) is in access table with ‘viewer’ flag.

➢ **worker:**

A worker of a repository r at time t is a developer, who is either owner of the repository r (or) collaborator of the repository r at time t.

These individuals collectively contribute to the repository's development, collaborate on code changes, and work together to achieve project's objectives. Their combined efforts drive the repository's progress and impact within the development community.



The one which are mentioned with ❖ are operations

TABLES AND CONSTRAINTS

```sql
CREATE TABLE developer(
   developer_id SERIAL PRIMARY KEY,
   user_name VARCHAR(100) UNIQUE NOT NULL,
   name VARCHAR(100) NOT NULL,
   email VARCHAR(100) UNIQUE NOT NULL,

   encrypted_password VARCHAR(100) NOT NULL,
   num_repos INT DEFAULT 0 CHECK(num_repos >= 0),
   storage_used INT DEFAULT 0 CHECK(storage_used >= 0),
   total_commits INT DEFAULT 0 CHECK(total_commits >= 0)
);
```

- ➢ We are not allowing to have users with same email (or) same user_name. But we are allowing developers with same name.
- ➢ The password is encrypted and stored in the encrypted_password column
- ➢ Salt used for encryption is '$2024$DBMS$fixedsalt'

```sql
CREATE TABLE repository(
   repository_id INT PRIMARY KEY,
   repository_name VARCHAR(100) NOT NULL,
   created_date_time timestamp,
   owner_id INT NOT NULL,
   is_public BOOLEAN DEFAULT true,
   root_id INT,                        /* root of the tree   */
   parent_id INT,                      /* parent of the repo */
   creator_id INT,                     /* worker who created */
   recent_commit_node INT DEFAULT NULL,

   FOREIGN KEY (owner_id)
   REFERENCES developer(developer_id) ON DELETE CASCADE,
   FOREIGN KEY (creator_id)
   REFERENCES developer(developer_id) ON DELETE SET NULL,
   FOREIGN KEY (root_id)
   REFERENCES repository(repository_id) ON DELETE NO ACTION,
   FOREIGN KEY (parent_id)
   REFERENCES repository(repository_id) ON DELETE CASCADE
);
```

- ➢ repository_id is the primary key to uniquely identify a repository.
- ➢ repository_name is the name of a repository. (The restrictions on having a repository_name for a repository is discussed in describing the function create_repo in next pages)
- ➢ owner_id is a foreign key referencing developer_id of developer table (developer who owns the repository), who is owning the repository.

NOT NULL on owner_id ensures that there exists no repository which does not have an owner. *ON DELETE CASCADE* of foreign key owener_id will force to delete all repositories owned by the developer automatically, when the developer's account is getting deleted in the octopus database.

- ➢ is_public flag indicates whether a repository is public to view (or) not. (make_private() function makes repository private with the help of recursive triggers.)
- ➢ parent_id is a foreign key referencing the repository_id of the parent of a repository. *ON DELETE CASCADE* automatically deletes a repository whenever its parent is deleted.
- ➢ root_id is a foreign key which references to root of the tree in which the repository is part of. This is a design choice!! root_id can always be found by multiple ways. Since root_id is required many times the decision to keep column root_id is taken. This is a good example of **trade off** between space and time. (**Frequently required data is made faster to access**)
  There is no need of any action when root is deleted (because ON DELETE CASCADE of parent_id will recursively delete all the descendants of a root.
- ➢ creator_id references to developer_id of developer table, who created the repository. A developer can create a repository, when the developer is a worker of the parent repository at the instant of creation. (note that the permissions are dynamic).
  creator_id is set to NULL when the creator's account got deleted.

```sql
ALTER TABLE repository
ADD CONSTRAINT fk_constraint
FOREIGN KEY(recent_commit_node)
REFERENCES commit_repository(repository_id)
ON DELETE SET NULL;
```

- *recent_commit_node* is a foreign key points to the latest commit node of the repository. This is useful in maintaining VERSION CONTROL SYSTEM of octopus.

Design challenge: This constraint is added separately after commit_repository table creation because of **cyclic foreign key referencing**.
(repository → commit_repository → commit → branch → repository) design.

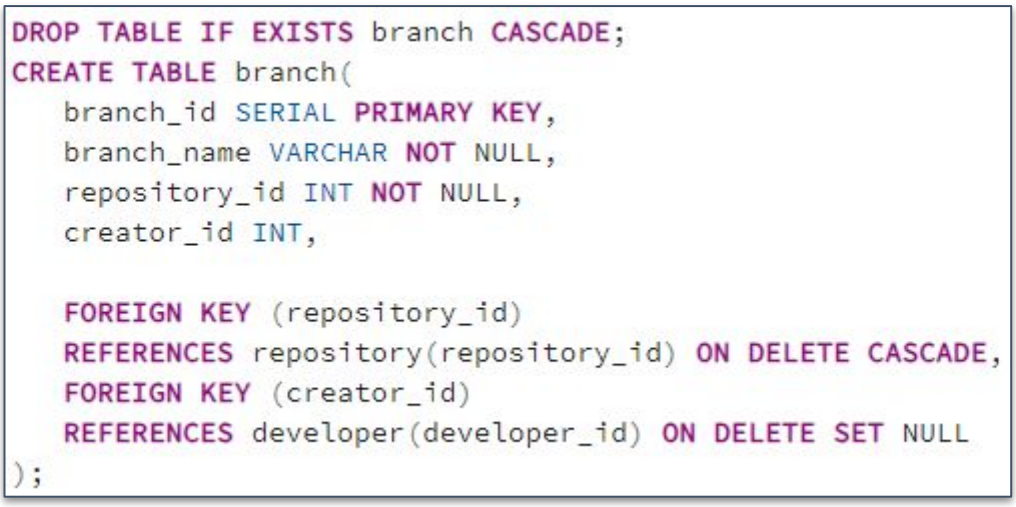
```sql
CREATE TABLE file(
   file_id SERIAL PRIMARY KEY,
   file_name VARCHAR(50) NOT NULL,
   file_type VARCHAR(10),
   size INT DEFAULT 0,
   content text DEFAULT '',
   parent_repository_id INT NOT NULL,
   created_date_time timestamp,
   last_update timestamp,
   creator_id INT,

   FOREIGN KEY(parent_repository_id)
   REFERENCES repository(repository_id) ON DELETE CASCADE,
   FOREIGN KEY(creator_id)
   REFERENCES developer(developer_id) ON DELETE SET NULL
);
```

- parent_repository_id is the id of parent repository.
- A file (uniquely identified by file_id) must be present in exactly one parent repository. (see the *NOT NULL* constraint near parent_repositry_id).
- *ON DELETE CASCADE* of parent_repository_id foreign key will result in deleting a file whenever its parent repository is deleted.
- file_name cannot be *NULL*.

- content is the content of the file in form of text
- creator_id is the foreign key referencing to developer_id in developer table. creator_id is the developer_id of the developer who created the file. A worker who has permission to parent repository can create a file under it.
- **file is a leaf in the repository tree.**

```sql
CREATE TYPE access_flag AS ENUM ('collaborator', 'viewer');

CREATE TABLE access(
   repository_id INT,
   developer_id INT,
   access_type access_flag NOT NULL,

   PRIMARY KEY(repository_id, developer_id),
   FOREIGN KEY (repository_id)
   REFERENCES repository(repository_id) ON DELETE CASCADE,
   FOREIGN KEY (developer_id)
   REFERENCES developer(developer_id) ON DELETE CASCADE
);
```

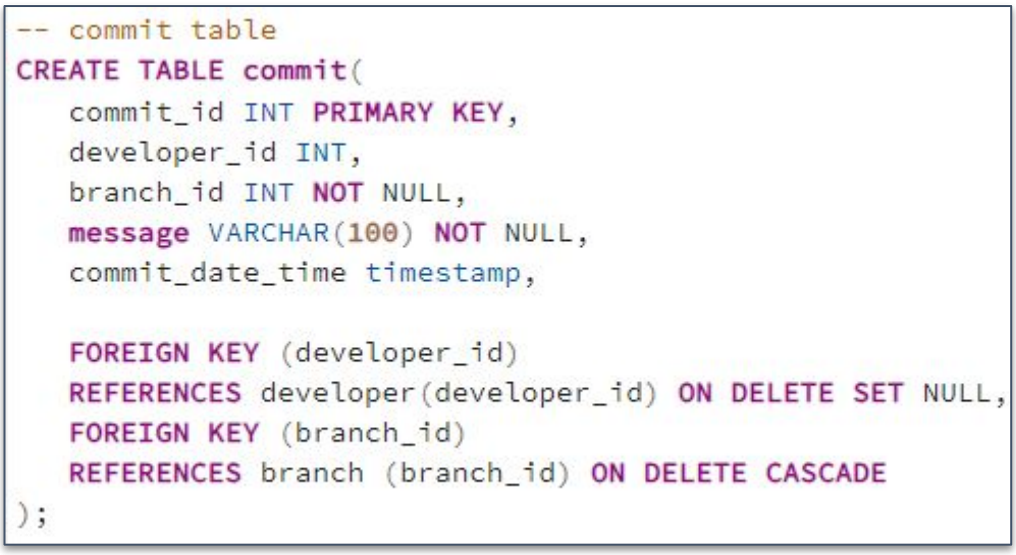
- access_flag type is created whose domain is 'collaborator' (or) 'viewer'
- access_type indicates the type of access between a developer and repository.
- This access table is used to check permissions of a developer whenever developer tries to do some operation on repository. (like view, commit and many more)



```sql
DROP TABLE IF EXISTS branch CASCADE;
CREATE TABLE branch(
   branch_id SERIAL PRIMARY KEY,
   branch_name VARCHAR NOT NULL,
   repository_id INT NOT NULL,
   creator_id INT,

   FOREIGN KEY (repository_id)
   REFERENCES repository(repository_id) ON DELETE CASCADE,
   FOREIGN KEY (creator_id)
   REFERENCES developer(developer_id) ON DELETE SET NULL
);
```

- ➢ Deleting a repository triggers a chain reaction whereby all its associated branches are erased, resulting in complete removal from the **version control system**.

- ➢ When a repository is branched then **branch_id** is created, branch name is given by the user, the repo id of the repo, and the developer id of the developer who branched in **creator_id** is stored in this table.

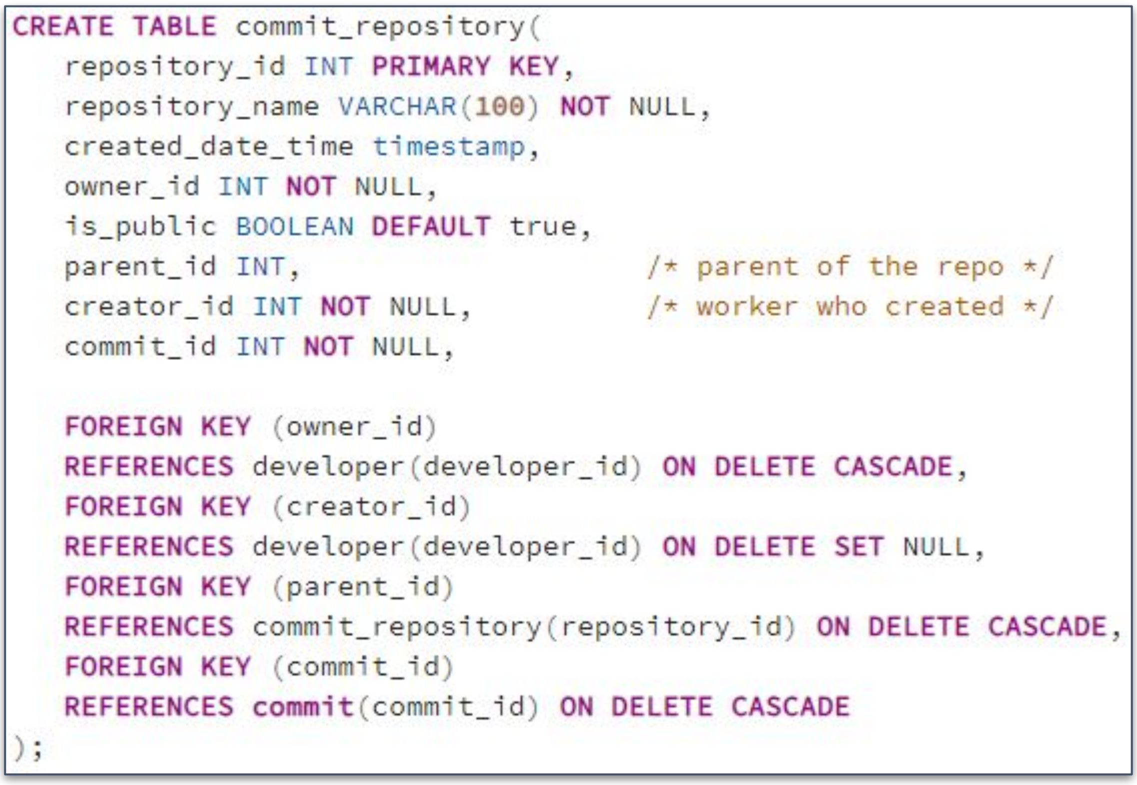


```sql
-- commit table
CREATE TABLE commit(
   commit_id INT PRIMARY KEY,
   developer_id INT,
   branch_id INT NOT NULL,
   message VARCHAR(100) NOT NULL,
   commit_date_time timestamp,

   FOREIGN KEY (developer_id)
   REFERENCES developer(developer_id) ON DELETE SET NULL,
   FOREIGN KEY (branch_id)
   REFERENCES branch (branch_id) ON DELETE CASCADE
);
```

- ➢ A commit represents an action made by precisely one developer on precisely one branch of a repository.
- ➢ The way the branch is moving can be inferred from the commits made to it. This is one of the important concepts in git.

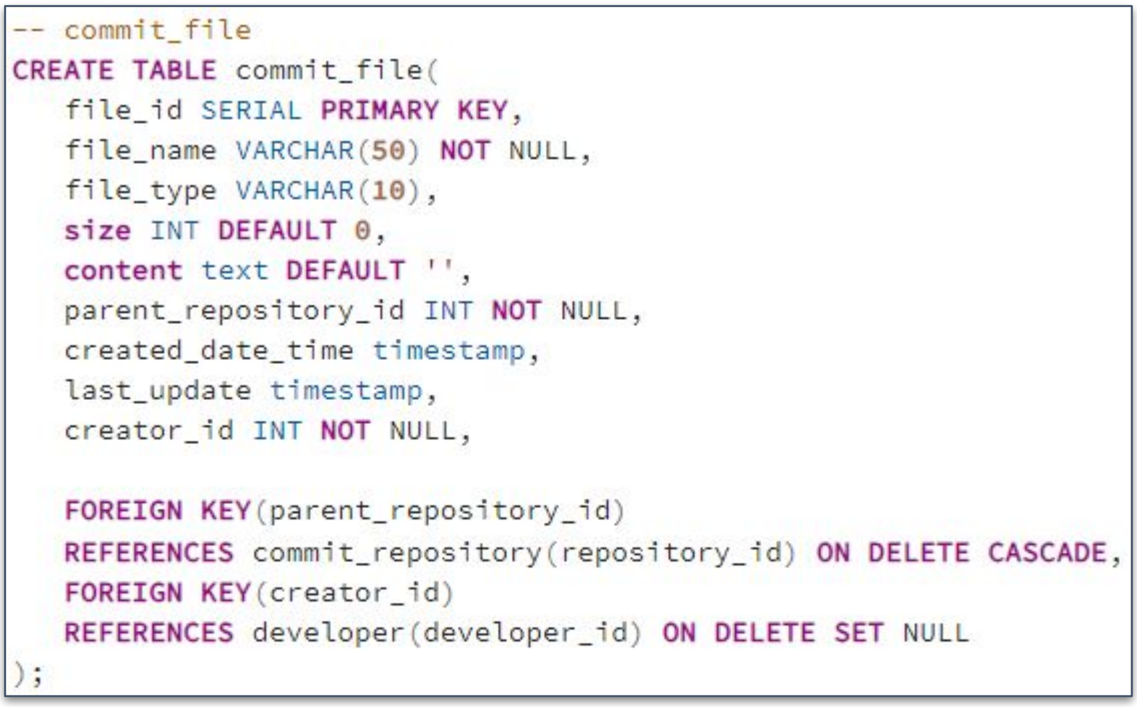
- ➢ The presence of a copy of the repository in the "copy_repository" table ensures that even if the original repository is deleted or modified, a preserved version remains accessible for reference or restoration purposes. This version control mechanism can acts as a safeguard against accidental loss or irreversible changes to the repository's content, providing developers with a safety net to mitigate potential data loss scenarios.

- ➢ Deleting a developer account does not impact the commits associated with that developer within the version control system. Regardless of whether a developer's account is removed, the commits they have made remain intact and attributed to their respective branches or repositories. This decoupling of developer accounts from commits ensures the continuity and integrity of the version history, allowing teams to manage access permissions without compromising the historical record of contributions and changes.



```sql
CREATE TABLE commit_repository(
   repository_id INT PRIMARY KEY,
   repository_name VARCHAR(100) NOT NULL,
   created_date_time timestamp,
   owner_id INT NOT NULL,
   is_public BOOLEAN DEFAULT true,
   parent_id INT,                      /* parent of the repo */
   creator_id INT NOT NULL,            /* worker who created */
   commit_id INT NOT NULL,

   FOREIGN KEY (owner_id)
   REFERENCES developer(developer_id) ON DELETE CASCADE,
   FOREIGN KEY (creator_id)
   REFERENCES developer(developer_id) ON DELETE SET NULL,
   FOREIGN KEY (parent_id)
   REFERENCES commit_repository(repository_id) ON DELETE CASCADE,
   FOREIGN KEY (commit_id)
   REFERENCES commit(commit_id) ON DELETE CASCADE
);
```

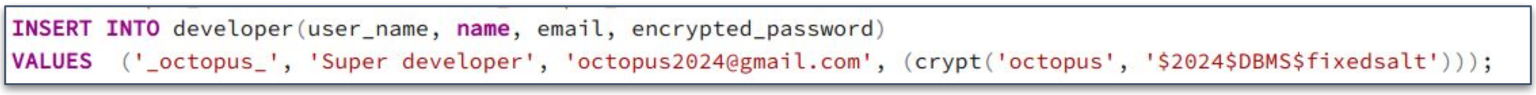
➢ The presence of a "commit_repository" table indicates that there is a mechanism in place to maintain a duplicate or backup copy of the repository's data. This redundancy serves as a safety measure against accidental loss or modification of the repository's content. However, the focus of the current report does not extend to detailing the specific actions related to commits within the repository.



```sql
-- commit_file
CREATE TABLE commit_file(
   file_id SERIAL PRIMARY KEY,
   file_name VARCHAR(50) NOT NULL,
   file_type VARCHAR(10),
   size INT DEFAULT 0,
   content text DEFAULT '',
   parent_repository_id INT NOT NULL,
   created_date_time timestamp,
   last_update timestamp,
   creator_id INT NOT NULL,

   FOREIGN KEY(parent_repository_id)
   REFERENCES commit_repository(repository_id) ON DELETE CASCADE,
   FOREIGN KEY(creator_id)
   REFERENCES developer(developer_id) ON DELETE SET NULL
);
```

➢ file_type limit the file type to certain values (e.g., 'txt', 'pdf', 'jpg')
➢ creator_id ensure that the creator_id exists in the "developer" table.
➢ Timestamps ensure that created_date_time and last_update timestamps are reasonable and consistent.

- In the octopus db there are some situations where few tuples are created automatically created in order to maintain consistency and ease of understanding. In the database there is a super user _octopus_ (the reasons why there is a need of this will be clear by the following examples)
  for example:
  1) Let us say that some developer whose username _swarna_ created a account. Then automatically a universal repository @_swarna_ is created by octopus whose owner is _swarna_.
  2) Branches '_master_' is created for every root by the _octopus_.



```
INSERT INTO developer(user_name, name, email, encrypted_password)
VALUES ('_octopus_', 'Super developer', 'octopus2024@gmail.com', (crypt('octopus', '$2024$DBMS$fixedsalt')));
```

Functions and Triggers

- In this section, functions and triggers are described parallely to make the flow of control more clear.
- Helper functions are functions which are used internally (in some cases many times by many other functions and triggers).
  Example:
  is_worker() function described below is used in all most every function. (because the need to know whether a developer is a worker (or) not is needed for many operations to be successful)
- **API functions are those functions called from back end, so as to make the request flow easily from back end to database and vice versa.**
- Trigger functions are those functions which are specifically created for helping triggers.

FUNCTION	ARGUMENTS	RETURN VALUE
	DESCRIPTION	

- Above picture describes about the pattern followed in describing functions in the next few pages.

HELPER function: is_worker	1. developer_id 2. repository_id	returns BOOLEAN
	returns true if the developer is either ● owner of the repository ● collaborator of the repository else returns false	
HELPER function: can_view	1. user_name 2. repository_id	returns BOOLEAN
	returns true if at least one of the following holds ● repository is public to view ● developer is the owner of the repository ● developer is either viewer (or) collaborator of the repository else returns false	

API functions and corresponding triggers(if any)

Below functions can be viewed as API functions. Because the below functions are majorly called from **node.js.** The software practise of making things modular helps in making easy API calls, and abstracting the complexity whenever possible. Arrows in below table indicate the control flow pattern.

API function: create_file()	1. file_name (name of the new file) 1. file_type (type of the new file) 1. content (content of the file to be saved in the new file) 1. parent_repository_id (repository_id of the parent) 1. developer_user_name (developer who initiated the creation of file)	returns table (created BOOLEAN, msg VARCHAR)
	● Creates a new file, if all the parameters passed validity checks, siblings files should be having different (file_name, file_type) tuple, developer who initiated the creation should be worker of parent.	
Trigger function: function_num_repos **Trigger:** trigger_num_repos **API function:** create_repo()	1. repository_name (name of the new repository) 1. parent_repository_id (repository_id of parent under which new repository is created) 1. user_name (user_name of the developer who initiated the creation)	returns table (is_created BOOLEAN, msg VARCHAR)
	● let d be the developer whose user_name = create_repo.user_name. ● let r be the repository whose repository_id = create_repo.parent_repository_id 1) If d is not a **worker** of r then false and relevant message is returned. **2) Different siblings:** If there exists a repository which is child of r and with same name as create_repo.repository_name then false, relevant message is returned. 1) If the control comes up until here (that is the function did not return above), then ❖ A new repository **n** is created as child to r. ❖ If r is universal_repository then, a branch 'master' is created by super_user 'octopus' for n, and n is public. else n.is_public = r.is_public ❖ Incrementing num_repos field of owner of r. 1) trigger_num_repos is triggered when a tuple is inserted in repository table, which will update the owner profile(num_repos++)	

Trigger function: function_grant_or_update_access ______________________ **Recursive trigger :** trigger_grant_or_update_access ______________________ **API function:** grant_or_update_access()	1. owner_user_name (owner of a repository) 2. repository_id (repository_id of repository for which access is being given (or) updated) 3. to_user_name (developer for access needs to be given) 4. access_type ('collaborator' (or) 'viewer')	returns table (given BOOLEAN, msg VARCHAR)
	1. edge case: parent and child cannot have collaborator access and view access respectively simultaneously. (other way round is fine) 2. If the to_user_name has already some access to the repository then the access is updated. else a tuple is inserted into access table. 3. Whenever updation (or) insertion is initiated in access table trigger **trigger_grant_or_update_access** gets triggered.It calls **function_grant_or_update_access,** which also grants access to descendants of repository. This will again trigger the above trigger. (thanks to recursion!!)	
API function: create_branch()	1. branch_name (new branch name) 2. repository_id (location where developer initiated the branch creation) 3. developer_user_name (developer who initiated the branch)	returns table (created BOOLEAN, msg VARCHAR)
	● let r be the repository whose repository_id = create_branch.repository_id 1. After validity checking of arguments passed, create_branch() will return false, appropriate message if a branch with the same name already exists associated with root of repository r. 2. If r is universal repository (r does not have root), then branch will be associated with r. 3. else: Whenever a branch creation is initiated at some repository r, the new branch (if created) will be associated with the root of the repository. 4. Commits are made to a branch. We say that branch moves head whenever a commit is made to a branch.	

	1. repository_id (repository where commit is made) 1. branch_name (branch specified by the user) 1. user_name (developer who initiated commit) 1. message	returns table (status BOOLEAN, msg VARCHAR)

HELPER function:
add_file

RECURSIVE HELPER procedure:
add_to_commit_area

API function:
add_commit

- Let t be the time when developer initiated commit.
- Let r be the repository whose repository_id = add_commit.repository_id. (if exists)
- Let root_r be the root of repository r. (if r is universal repository then root_r = r)
- Let d be the developer whose user_name = add_commit.user_name (if one exists)
- Let b be the branch whose branch_name = add_commit.branch_name and associated with the repository root_r.

Who can commit?

If developer d is the worker of repository r at time t, then d can commit repository r to branch b.

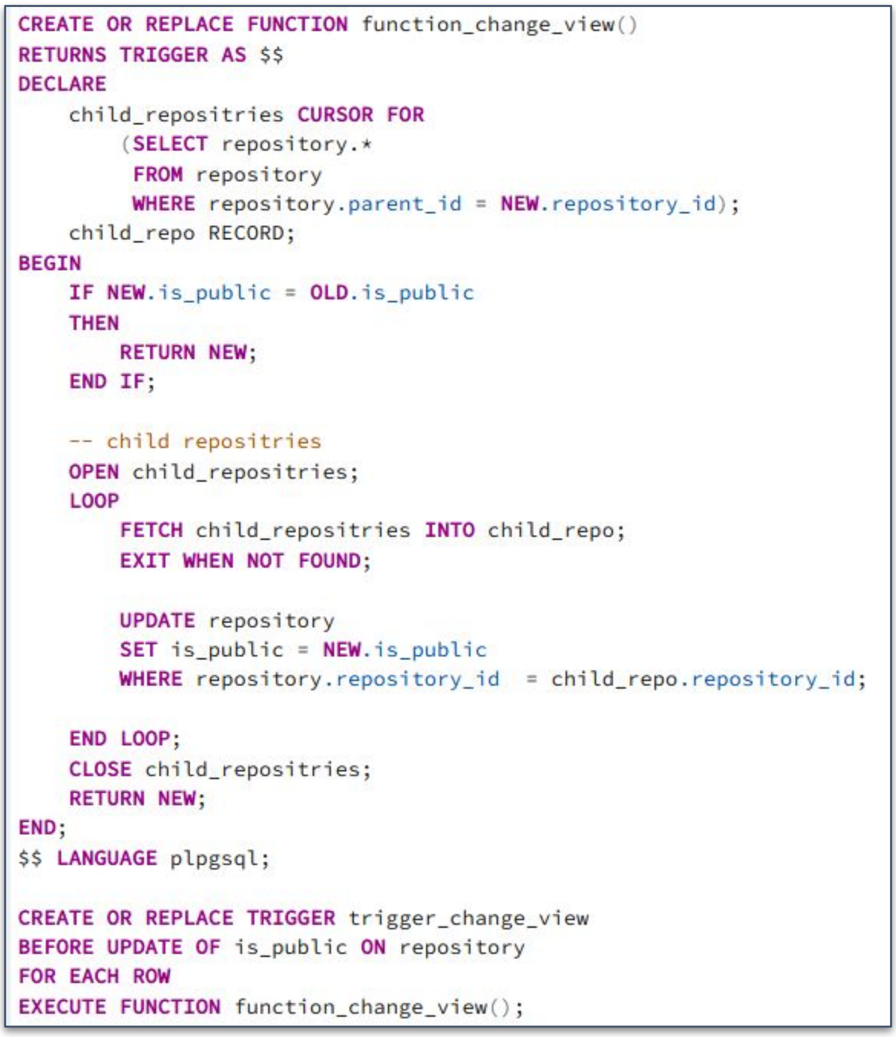
If any of the parameters have invalid value, then (false, relevant msg) is returned.

COMMIT ALGORITHM:

1. Recursively duplicate the contents of r and all its descendants to separate tables (commit_repository, commit_file).
   make parent(duplicate(r)) = NULL.
   This is achieved by calling **add_to_commit_area** procedure which will recursively call itself to add all the children repositories, and call **add_file** to add child files.
1. p = parent(r)

```
While(true){
        if (p is NULL){
                exit;
        }
        if (p.recent_commit_node is NOT NULL){
                // then p is committed before
                parent(duplicate(r)) = p.recent_commit_node;
                exit;
        }
        // p is not committed before
        duplicate p into commit_repository;
        // p's descendants are not added !!
        p.recent_commit_node = duplicate(p);
        r = p;
        p = parent(p);
}
```

Trigger function: function_change_view **RECURSIVE trigger:** trigger_change_view **API function:** change_view()	1. repository_id 2. owner_user_name 3. is_public	returns TABLE(is_changed BOOLEAN, msg VARCHAR)
	1. Only owner can make a repository private (or) public to view. 2. Parent and child cannot be private and public respectively at the same time. (but other way round is fine) 3. After checking above conditions, then the flag of repository is updated. This will trigger **trigger_change_view**, which will execute **function_change_view**, which will update the flag of all child repositories of the repositories. This recursion continues until all descendents are updated. (thanks to recursion!!)	



```
CREATE OR REPLACE FUNCTION function_change_view()
RETURNS TRIGGER AS $$
DECLARE
    child_repositries CURSOR FOR
        (SELECT repository.*
         FROM repository
         WHERE repository.parent_id = NEW.repository_id);
    child_repo RECORD;
BEGIN
    IF NEW.is_public = OLD.is_public
    THEN
        RETURN NEW;
    END IF;

    -- child repositries
    OPEN child_repositries;
    LOOP
        FETCH child_repositries INTO child_repo;
        EXIT WHEN NOT FOUND;

        UPDATE repository
        SET is_public = NEW.is_public
        WHERE repository.repository_id  = child_repo.repository_id;

    END LOOP;
    CLOSE child_repositries;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER trigger_change_view
BEFORE UPDATE OF is_public ON repository
FOR EACH ROW
EXECUTE FUNCTION function_change_view();
```

➔ Since describing all triggers is not possible to explain in this report, we would like to explain the recursive trigger used while changing the view of a repository.
➔ When a repository is made private all its descendants should be made private.
➔ When is_public flag is modified in change_view() function, **trigger_change_view** is triggered.
➔ trigger_change_view calls **function_change_view.**
➔ function_change_view will loop across all children of a repository and modify the flags to the flag of parent.
➔ This modification will again trigger **trigger_change_view** and this continues until all appropriate changes are done in the repository.

INDICES

- **index_user_name** index:
  - creation: index_user_name is a HASH index on user_name of the developer.
  - purpose: The fact that the user_name of all the developers is unique, user_name is used very often to communicate with front end and back end. So a hash index is created for it so that fast access can happen.
  - use case: In functions like create_repo(), create_file(), create_branch() etc.. this index is helpful. Because in all these functions, helper functions and in some triggers user_name is used. Unique constraint is specified in foreign key constraint so no need to specify in index.
- **sort_time** index:
  - creation: sort_time index is created on commit_date_time
  - purpose and use_case: It is a general requirement that any developer want to all the commits made to a branch, to know how the branch grew over time.
- **index_repository_id** index:
  - creation: Index is created on repository_id
  - purpose: Since repository name is not unique, every access of a repository is using repository_id, so to make it faster we created this index.
  - use_case: In all the functions, this index is helpful. (repository_id is used everywhere)

There are not many instances where we can create indices in this database, because of extreme flexibility which the database offers. The order in which repositories is accessed is completely dependent on the underlying repository tree.

VIEWS

- **head_branch** view:
  - creation: head_branch view is created on tables branch, commit table. Where each tuple in virtual table head_branch has tuple (branch_id, commit_id), where commit_id is the latest commit of a branch.
    If there are no commits to a branch, commit_id is null in the virtual table.
  - use_case: Very often a developer (generally) wants to know the head of a branch. The data where the head of branch is currently pointing to is needed many times. So this view is created for this purpose.
- **all_roots** view:
  - creation:
    This view is created on tables repository and developer.
    Each tuple in virtual table **all_roots** has tuples of the form (developer_id, repository_id) where repository_id is the root owned by developer_id developer
  - use_case:
    - Very often a developer wants to know all the repostires developer has which are direct children of the universal repository of the developer.
    - These roots are the entry points of the actual warehouse of the work/info/projects of a developer.
- **commit_hisotry** view:
  - creation:
    This view is created on tables branch, repository, commit_repository
    Each tuple contains the repository snapshot pointing to commit_repository for each branch in keeping latest commit first. (order of commits latest (first) ~~> old(last))
  - use_case:
    commit history is a common feature of version control system.This is one of the very useful features every developer wants to use for his/her repository.

ROLES

Database Administrator (DBA):

The DBA role is responsible for managing the PostgreSQL database, including tasks such as creating and managing databases, configuring security settings, monitoring performance, and troubleshooting issues.

Superuser:

The superuser role has full privileges on the PostgreSQL database, including the ability to perform any operation and access any data. Superusers have unrestricted access to all databases and can perform administrative tasks that regular users cannot.

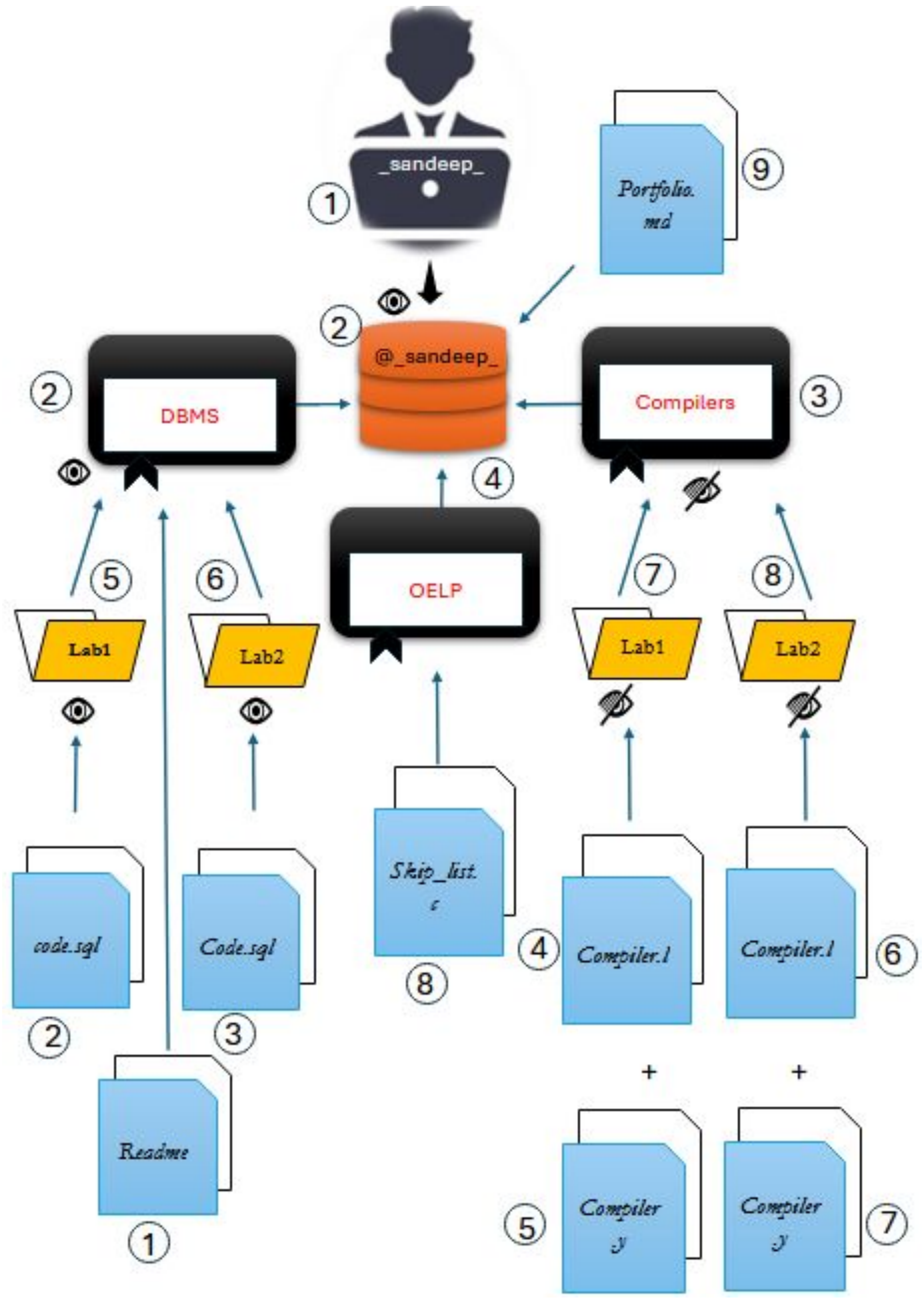
Monitoring User:

This role might be assigned to users or applications that need access to monitoring and performance-related information from the database. Monitoring users typically have privileges to view system catalogs, query execution statistics, and other monitoring-related data.

Below are the icons descriptions which are used in the next page

👤 _sandeep_	→ Developer	📁	→ Repository (other than universal and root repository)
🛢 @_sandeep_	→ Universal repository	📄	→ File
▭	→ Root repository	👁	→ Public to view
		🙈	→ Private to view

- ➢ When a developer is created here **_sandeep_** a repository is automatically created by octopus named @developer here **@_sandeep_** , this is for the feature that is a user can upload a file without creating a repository.
- ➢ Here **Portfolio.md** file is using that feature .Then if user will create a repository it will be a sub repository of @_sandeep_ Like **DBMS**, **Compilers**, **OELP**.
- ➢ Here DBMS repository,OELP repositories are public but compilers repository is private so all sub repositories and files of DBMS and OELP repositories are public and private in compilers repository.
- ➢ The DBMS repository contains a file named **readme**, and two folders named **Lab1** and **Lab** 2.Both folders contain **Code.sql** files.Similarly OELP repository contains **skip_list .c** file.
- ➢ The compilers repository which is private contains two folders namely Lab 1 and Lab 2 and both contain **compiler.l** and **compiler.y** files which are private.

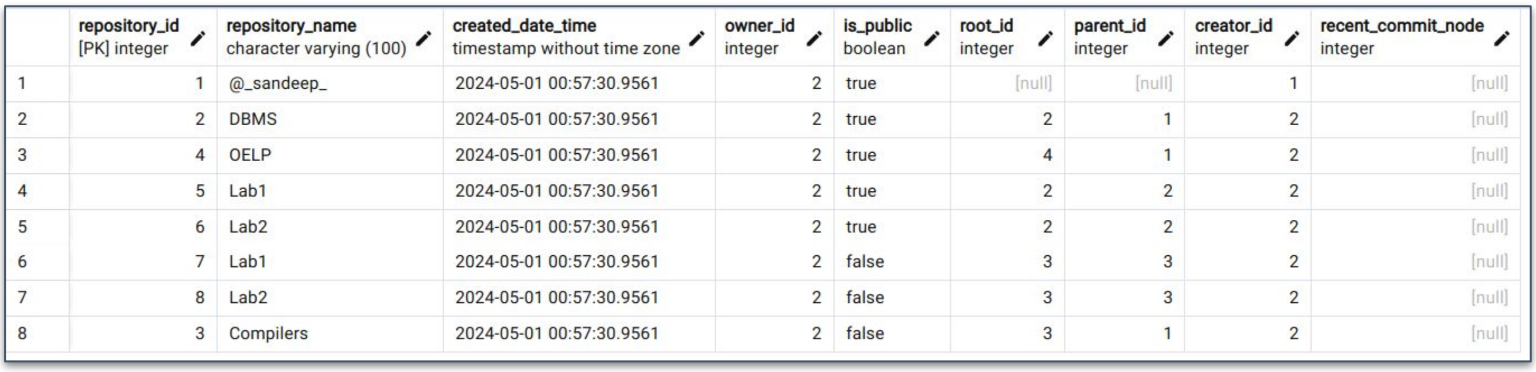


sandeep
1
Portfolio.
md
9
2
@_sandeep_
2
DBMS
Compilers
3
4
OELP
5
6
7
8
Lab1
Lab2
Lab1
Lab2
code.sql
2
Code.sql
3
Readme
1
Skip_list.
c
8
4
Compiler.l
Compiler.l
6
+
+
5
Compiler
.y
Compiler
.y
7

MULTI RELATIONAL QUERIES

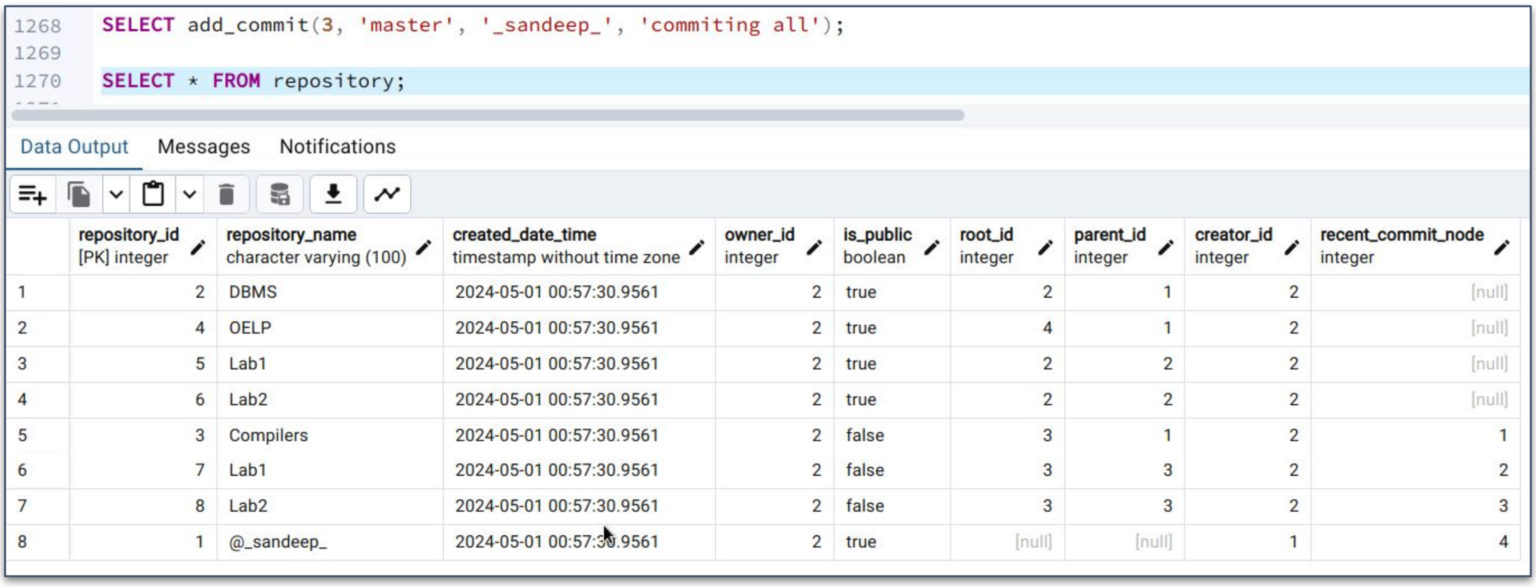
1. Demonstrating how a commit is working. What all repositories are copied? What about the changes outside of repository ?

Before commit:



	repository_id [PK] integer	repository_name character varying (100)	created_date_time timestamp without time zone	owner_id integer	is_public boolean	root_id integer	parent_id integer	creator_id integer	recent_commit_node integer
1	1	@_sandeep_	2024-05-01 00:57:30.9561	2	true	[null]	[null]	1	[null]
2	2	DBMS	2024-05-01 00:57:30.9561	2	true	2	1	2	[null]
3	4	OELP	2024-05-01 00:57:30.9561	2	true	4	1	2	[null]
4	5	Lab1	2024-05-01 00:57:30.9561	2	true	2	2	2	[null]
5	6	Lab2	2024-05-01 00:57:30.9561	2	true	2	2	2	[null]
6	7	Lab1	2024-05-01 00:57:30.9561	2	false	3	3	2	[null]
7	8	Lab2	2024-05-01 00:57:30.9561	2	false	3	3	2	[null]
8	3	Compilers	2024-05-01 00:57:30.9561	2	false	3	1	2	[null]

After committing:



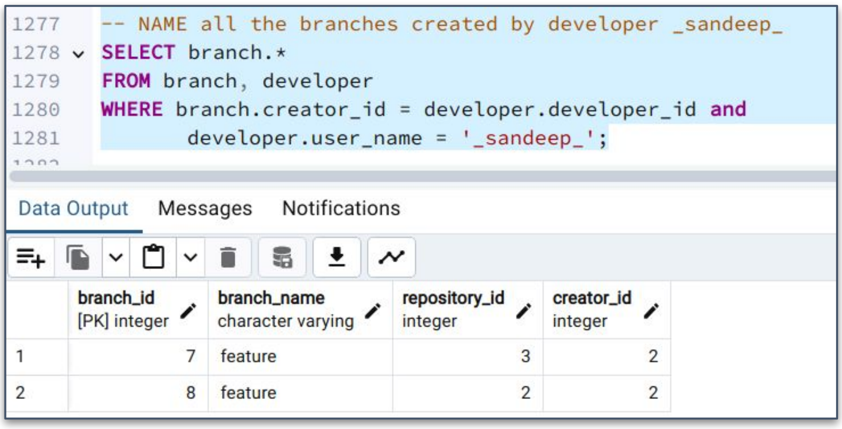
```
SELECT add_commit(3, 'master', '_sandeep_', 'commiting all');

SELECT * FROM repository;
```

	repository_id [PK] integer	repository_name character varying (100)	created_date_time timestamp without time zone	owner_id integer	is_public boolean	root_id integer	parent_id integer	creator_id integer	recent_commit_node integer
1	2	DBMS	2024-05-01 00:57:30.9561	2	true	2	1	2	[null]
2	4	OELP	2024-05-01 00:57:30.9561	2	true	4	1	2	[null]
3	5	Lab1	2024-05-01 00:57:30.9561	2	true	2	2	2	[null]
4	6	Lab2	2024-05-01 00:57:30.9561	2	true	2	2	2	[null]
5	3	Compilers	2024-05-01 00:57:30.9561	2	false	3	1	2	1
6	7	Lab1	2024-05-01 00:57:30.9561	2	false	3	3	2	2
7	8	Lab2	2024-05-01 00:57:30.9561	2	false	3	3	2	3
8	1	@_sandeep_	2024-05-01 00:57:30.9561	2	true	[null]	[null]	1	4

Description:

1. When a commit is made to 'master' branch by developer '_sandeep_' to the repository Compilers all descendants of Compilers are committed. Since the parent of Compilers is not committed earlier, only parent repo is committed (no other descendants are committed, like DBMS is not committed in the above picture, that is the reason recent_commit_node of DBMS is null)
2. The sophisticated commit algorithm is hidden in add_commit() function.
3. This example demonstrated the working of commit algorithm. When we are committing a repository the changes to other data which are not descendants of this repository should be ignored. That is successfully executed in our database.

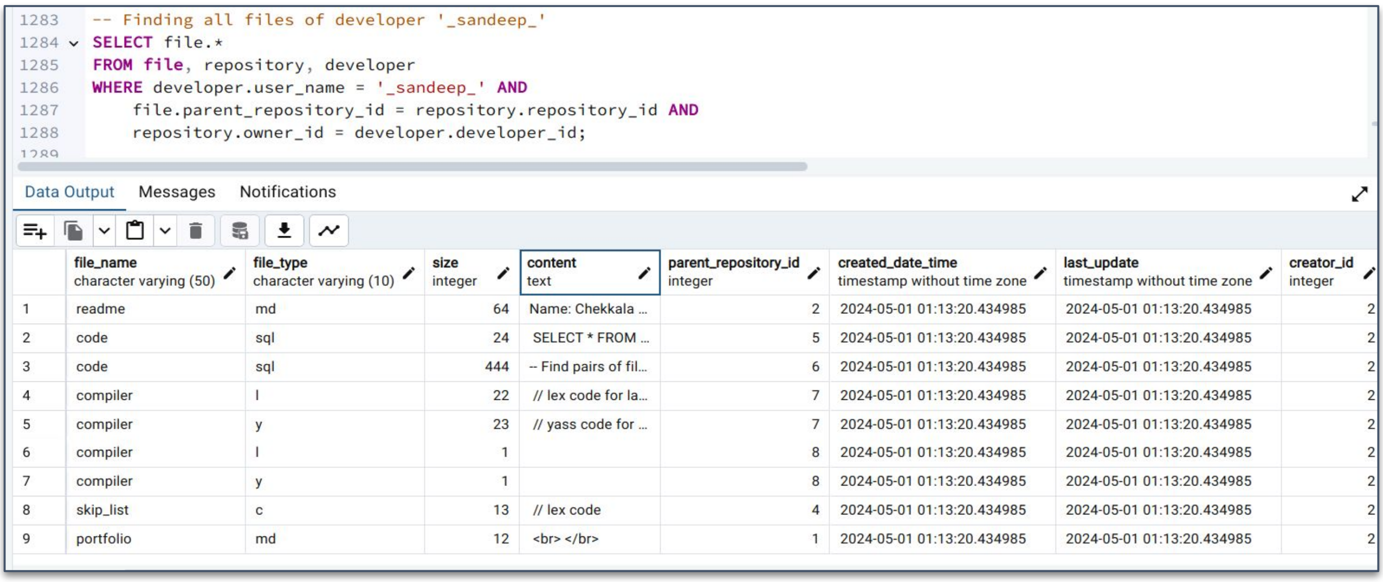
2. We have populated some more data. The query is to find all branches created by a developer.



```
-- NAME all the branches created by developer _sandeep_
SELECT branch.*
FROM branch, developer
WHERE branch.creator_id = developer.developer_id and
      developer.user_name = '_sandeep_';
```

	branch_id [PK] integer	branch_name character varying	repository_id integer	creator_id integer
1	7	feature	3	2
2	8	feature	2	2

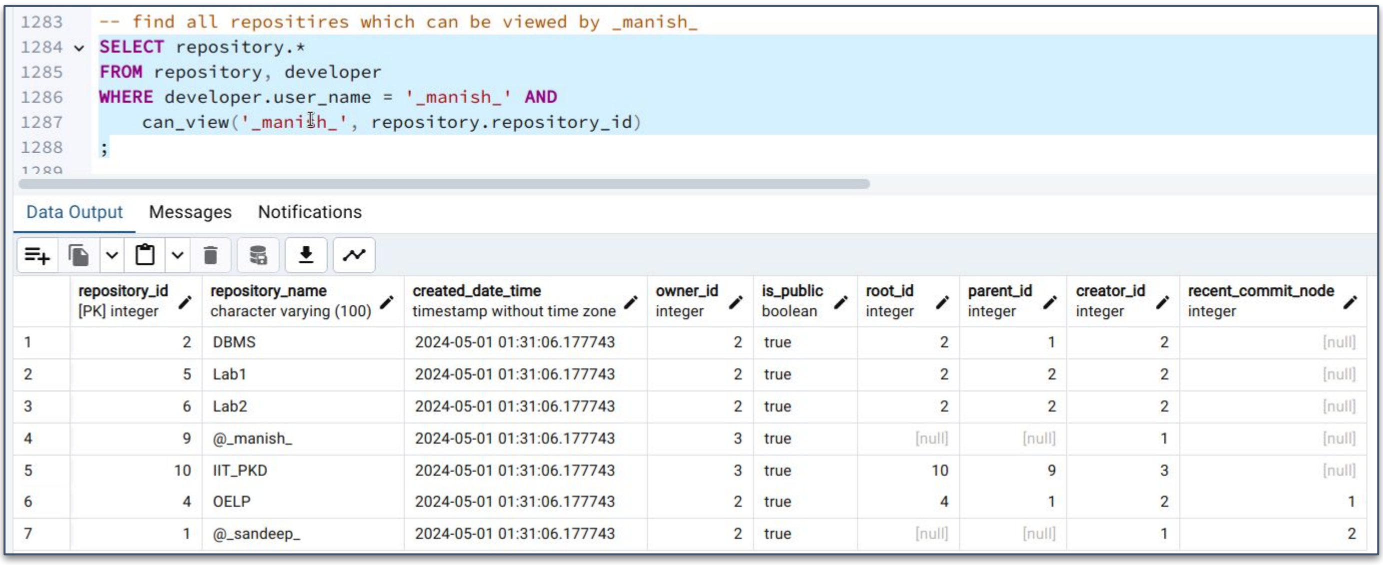
3. Finding all files owned by a developer. (not that there is no owner_id field in file)



```sql
-- Finding all files of developer '_sandeep_'
SELECT file.*
FROM file, repository, developer
WHERE developer.user_name = '_sandeep_' AND
    file.parent_repository_id = repository.repository_id AND
    repository.owner_id = developer.developer_id;
```

	file_name character varying (50)	file_type character varying (10)	size integer	content text	parent_repository_id integer	created_date_time timestamp without time zone	last_update timestamp without time zone	creator_id integer
1	readme	md	64	Name: Chekkala ...	2	2024-05-01 01:13:20.434985	2024-05-01 01:13:20.434985	2
2	code	sql	24	SELECT * FROM ...	5	2024-05-01 01:13:20.434985	2024-05-01 01:13:20.434985	2
3	code	sql	444	-- Find pairs of fil...	6	2024-05-01 01:13:20.434985	2024-05-01 01:13:20.434985	2
4	compiler	l	22	// lex code for la...	7	2024-05-01 01:13:20.434985	2024-05-01 01:13:20.434985	2
5	compiler	y	23	// yass code for ...	7	2024-05-01 01:13:20.434985	2024-05-01 01:13:20.434985	2
6	compiler	l	1		8	2024-05-01 01:13:20.434985	2024-05-01 01:13:20.434985	2
7	compiler	y	1		8	2024-05-01 01:13:20.434985	2024-05-01 01:13:20.434985	2
8	skip_list	c	13	// lex code	4	2024-05-01 01:13:20.434985	2024-05-01 01:13:20.434985	2
9	portfolio	md	12	
 </br>	1	2024-05-01 01:13:20.434985	2024-05-01 01:13:20.434985	2

4. Finding all repositories which can be viewed by a specific developer.

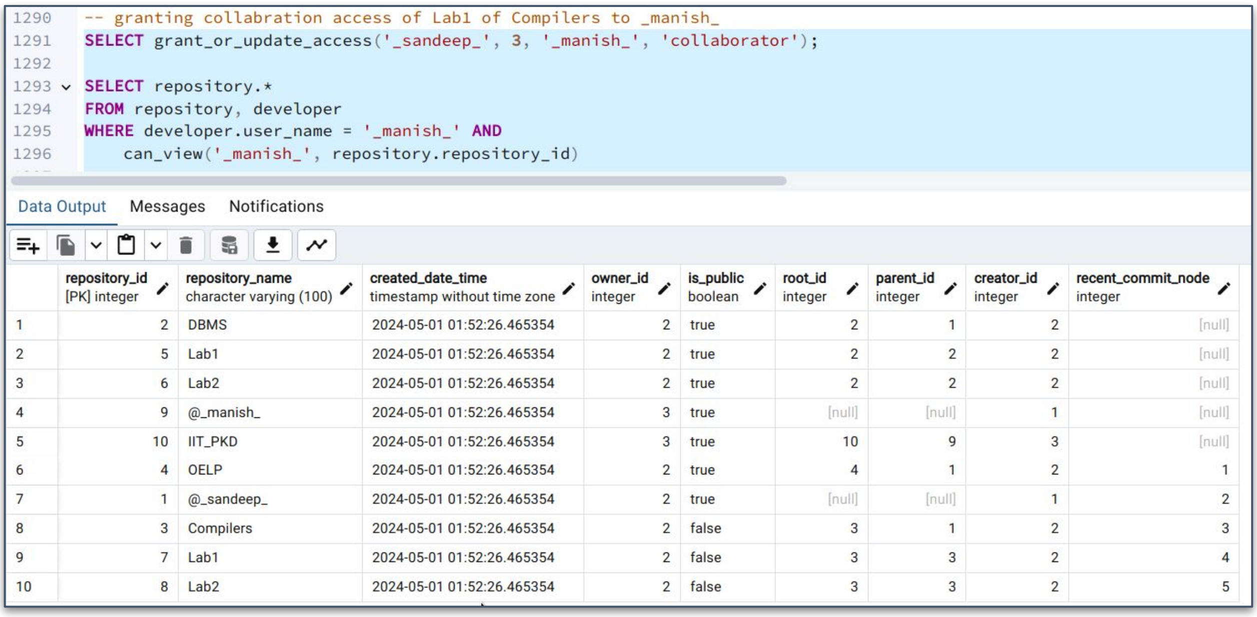


```sql
-- find all repositires which can be viewed by _manish_
SELECT repository.*
FROM repository, developer
WHERE developer.user_name = '_manish_' AND
    can_view('_manish_', repository.repository_id)
;
```

	repository_id [PK] integer	repository_name character varying (100)	created_date_time timestamp without time zone	owner_id integer	is_public boolean	root_id integer	parent_id integer	creator_id integer	recent_commit_node integer
1	2	DBMS	2024-05-01 01:31:06.177743	2	true	2	1	2	[null]
2	5	Lab1	2024-05-01 01:31:06.177743	2	true	2	2	2	[null]
3	6	Lab2	2024-05-01 01:31:06.177743	2	true	2	2	2	[null]
4	9	@_manish_	2024-05-01 01:31:06.177743	3	true	[null]	[null]	1	[null]
5	10	IIT_PKD	2024-05-01 01:31:06.177743	3	true	10	9	3	[null]
6	4	OELP	2024-05-01 01:31:06.177743	2	true	4	1	2	1
7	1	@_sandeep_	2024-05-01 01:31:06.177743	2	true	[null]	[null]	1	2

can_view() is a function which is specified earlier in functions section. Note that Compilers repository is private, so Compilers and all its descendants cannot be viewed by _manish_.
So Compilers and all its descendants are not there in the above output.

5. Granting collaborator access to _manish_ by _sandeep_ of repository Compilers. (see output changes)



```sql
-- granting collabration access of Lab1 of Compilers to _manish_
SELECT grant_or_update_access('_sandeep_', 3, '_manish_', 'collaborator');

SELECT repository.*
FROM repository, developer
WHERE developer.user_name = '_manish_' AND
    can_view('_manish_', repository.repository_id)
```

	repository_id [PK] integer	repository_name character varying (100)	created_date_time timestamp without time zone	owner_id integer	is_public boolean	root_id integer	parent_id integer	creator_id integer	recent_commit_node integer
1	2	DBMS	2024-05-01 01:52:26.465354	2	true	2	1	2	[null]
2	5	Lab1	2024-05-01 01:52:26.465354	2	true	2	2	2	[null]
3	6	Lab2	2024-05-01 01:52:26.465354	2	true	2	2	2	[null]
4	9	@_manish_	2024-05-01 01:52:26.465354	3	true	[null]	[null]	1	[null]
5	10	IIT_PKD	2024-05-01 01:52:26.465354	3	true	10	9	3	[null]
6	4	OELP	2024-05-01 01:52:26.465354	2	true	4	1	2	1
7	1	@_sandeep_	2024-05-01 01:52:26.465354	2	true	[null]	[null]	1	2
8	3	Compilers	2024-05-01 01:52:26.465354	2	false	3	1	2	3
9	7	Lab1	2024-05-01 01:52:26.465354	2	false	3	3	2	4
10	8	Lab2	2024-05-01 01:52:26.465354	2	false	3	3	2	5

Milestones

1. Any repository can be made private not just root repositories.
2. Access of any repository can be controlled by the owner. not just root repositories. A owner can grant collaborator (or) viewer access to any developer of any repository.
3. A developer can have files without having any repositories. (this became possible because of adding universal repository concept in the database)
4. All remaining functionalities are implied.

REFERENCES

1) Most of the concepts used in building octopus is inspired from Database System Concepts 7e.
2) PostgreSQL documentation for implementation in postgreSQL.