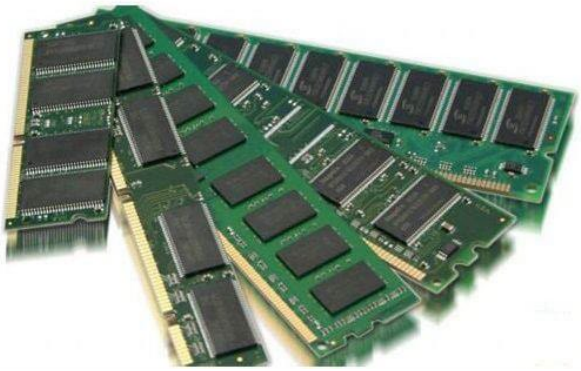
Filez DB



Tables**:**

1. **Files(**id, type, size_needed**)**
    - holds all attributes in File struct, id is primary key
    - checks type is not null, size needed not null >= 0, id >0
2. **Disks(**id, company, speed, free_space, cost**)**
    - holds all attributes in Disk struct, id is primary key
    - checks company is not null, speed not null > 0, id >0, free_space not null >= 0, cost not null > 0
3. **RAMs(**id, company, size,**)**
    - holds all attributes in RAM struct, id is primary key
    - checks company is not null, size not null > 0, id >0

4. **FilesOFDisk(**file_id, disk_id**)**
    - Connection between files and the disks they are on
    - file_id, disk_id are foreign keys, file_id, disk_id is the primary key

5. **RAMsOFDisk(**RAM_id, disk_id**)**
    - Connection between RAMS and the disks they are on
    - RAM_id, disk_id are foreign keys, RAM_id, disk_id is the primary key

6. **DisksCheck(**disk_id**)**
    - Holds all disks id is db as a foreign and primary key
    - Used for validation

Views**:**

1. **RamSizeOfDisk(**disk_id, totalRamSize**)**
    - Holds all disks id with Rams and the accumulated size of their RAMs

2. **PotentialFilesForDisk(**disk_id, File_id**)**
    - Connetction between disks and files that can fit in their free_space

3. **FilesWithCommonDisks(**file_id, shared_file_id, disk_id**)**
    - Connection between 2 files through same disk

4. **CommonDiskCount(**file_id, shared_file_id, sharedDiskCount**)**
    - Connection between 2 files that have common disks and the count of disks they share

5. **CommonVsTotalDisks(**file_id, shared_file_id, sharedDiskCount, total disk**)**
    - Connection between 2 files that have common disks, the count of disks they share and the total number of disk for file_id.
    - The reason we need both this table and CommonDiskCount is that we need to fix the counting for shared Disk before crossing with FilesOfDisk for counting totalDisks.

6. **isCloseFiles(**file_id, shared_file_id, isClose**)**
    - Connection between 2 files and if they are close (by definition, file_id shares more than 50% of its disks with shared_file_id.)
    - Note that this relation is not transitive therefor file_id and shared_file_id are not interchangeable.

API:

- def addFile(file: File) -> Status:

```
query = sql.SQL(""" INSERT INTO Files(id, type, size_needed)
                    VALUES({id}, {type} ,{size});
                    """).format(id=sql.Literal(file.getFileID()), type=sql.Literal(file.getType()),
                                size=sql.Literal(file.getSize()))
```

Simple insertion of attributes to table with the appropriate exception check.

- def getFileByID(fileID: int) -> File:

```
query = sql.SQL("""SELECT *
                   FROM Files
                   WHERE id = {id};
                   """).format(id=sql.Literal(fileID))
```

Simple selection of attributes from table with the appropriate exception check.

- def deleteFile(file: File) -> Status:

```
query = sql.SQL("""BEGIN;
                   UPDATE Disks
                   set free_space = Disks.free_space +
                   COALESCE ((SELECT Files.size_needed
                              FROM FilesOfDisk, Files
                              WHERE Disks.id = FilesOfDisk.Disk_id
                                    AND FilesOfDisk.File_id = Files.id
                                    AND Files.id = {fileID}), 0);
                   DELETE FROM Files WHERE id={fileID};

                   """).format(fileID=sql.Literal(file.getFileID()))
```

First, we update the relevant disks free space and then delete file, if one operation fails the hole query will not be committed. We use coalesce to protect from NULL incase the file is not on the disk).

- def addDisk(disk: Disk) -> Status:

```
query = sql.SQL(""" INSERT INTO Disks(id, company, speed, free_space, cost)
                    VALUES ({id}, {company}, {speed}, {free_space}, {cost});
                    """).format(id=sql.Literal(disk.getDiskID()), company=sql.Literal(disk.getCompany()),
                                speed=sql.Literal(disk.getSpeed()), free_space=sql.Literal(disk.getFreeSpace()),
                                cost=sql.Literal(disk.getCost()))
```

Simple insertion of attributes to table with the appropriate exception check.

- def getDiskByID(diskID: int) -> Disk:

```
query = sql.SQL("""SELECT *
                   FROM Disks
                   WHERE id = {id};
                   """).format(id=sql.Literal(diskID))
```

Simple selection of attributes from table with the appropriate exception check.

- def deleteDisk(diskID: int) -> Status:

```
query = sql.SQL("""DELETE FROM Disks
                   WHERE Disks.id={id};
                   """).format(id=sql.Literal(diskID))
```

Simple deletion of row from table with the appropriate exception check.

- def addRAM(ram: RAM) -> Status:

```
query = sql.SQL("""BEGIN;
                   INSERT INTO RAMS(id, size, company)
                   VALUES({id}, {RAMSize}, {company});
                   """).format(id=sql.Literal(ram.getRamID()), RAMSize=sql.Literal(ram.getSize()),
                               company=sql.Literal(ram.getCompany()))
```

Simple insertion of attributes to table with the appropriate exception check.

- def getRAMByID(ramID: int) -> RAM:

```
query = sql.SQL("""SELECT *
                   FROM RAMs
                   WHERE id = {id}
                   """).format(id=sql.Literal(ramID))
```

Simple selection of attributes from table with the appropriate exception check.

- def deleteRAM(ramID: int) -> Status:

```
query = sql.SQL("""DELETE FROM RAMs
                    WHERE id={id};
                    """).format(id=sql.Literal(ramID))
```

Simple deletion of row from table with the appropriate exception check.

- def addDiskAndFile(disk: Disk, file: File) -> Status:

```
query = sql.SQL("""BEGIN;
                    INSERT INTO Files(id, type, size_needed)
                    VALUES({fileID}, {fileType} ,{fileSize});
                    INSERT INTO Disks(id, company, speed, free_space,cost)
                    VALUES({diskID}, {diskCompany}, {diskSpeed}, {diskFreeSpace}, {diskCost});
                    """).format(fileID=sql.Literal(file.getFileID()),
                                fileType=sql.Literal(file.getType()),
                                fileSize=sql.Literal(file.getSize()),
                                diskID=sql.Literal(disk.getDiskID()),
                                diskCompany=sql.Literal(disk.getCompany()),
                                diskSpeed=sql.Literal(disk.getSpeed()),
                                diskFreeSpace=sql.Literal(disk.getFreeSpace()),
                                diskCost=sql.Literal(disk.getCost()))
```

Simple transaction of two insertions of attributes to table with the appropriate exception check.

- def addFileToDisk(file: File, diskID: int) -> Status:

```
query = sql.SQL("""BEGIN;
                   UPDATE Disks
                   set free_space = Disks.free_space -
                       COALESCE((SELECT Files.size_needed
                       FROM Files
                       WHERE Files.id = {file_ID}), 0)
                   WHERE Disks.id = {disk_ID};

                   INSERT INTO FilesOfDisk(File_id, Disk_id)
                   VALUES ({file_ID}, {disk_ID});
                   """).format(disk_ID=sql.Literal(diskID), file_ID=sql.Literal(file.getFileID()))
```

First, we update the relevant disks free space and then we add both file and disk to the filesOFDisk relation, if one operation fails the hole query will not be committed. We use coalesce to protect from NULL incase the file is not on the disk).

- def removeFileFromDisk(file: File, diskID: int) -> Status:

```
query = sql.SQL("""BEGIN;
                    UPDATE Disks
                    set free_space = Disks.free_space +
                    COALESCE ((SELECT Files.size_needed
                               FROM FilesOfDisk, Files
                               WHERE {disk_id} = FilesOfDisk.Disk_id
                                   AND FilesOfDisk.File_id = Files.id
                                   AND Files.id = {file_id}), 0)
                    WHERE Disks.id = {disk_id};
                    DELETE
                    FROM FilesOfDisk
                    WHERE FilesOfDisk.File_id={file_id};
                    """).format(disk_id=sql.Literal(diskID), file_id=sql.Literal(file.getFileID()))
```

First, we update the relevant disks free space and then we remove both file and disk to the filesOFDisk relation, if one operation fails the hole query will not be committed. We use coalesce to protect from NULL incase the file is not on the disk).

- def addRAMToDisk(ramID: int, diskID: int) -> Status:

```
query = sql.SQL("""BEGIN;
                    INSERT INTO RAMsOfDisk(RAM_id, Disk_id)
                    VALUES ({ram_id}, {disk_id});
                    """).format(ram_id=sql.Literal(ramID), disk_id=sql.Literal(diskID))
```

Simple insertions of attributes to table with the appropriate exception check.
The RamSizeOfDisk view takes care of accumulating the ram sizes of each disk.

- def removeRAMFromDisk(ramID: int, diskID: int) -> Status:

```
query = sql.SQL("""BEGIN;
                    DELETE FROM RAMsOfDisk
                    WHERE RAM_id={ram_id} and Disk_id={disk_id};
                    """).format(ram_id=sql.Literal(ramID), disk_id=sql.Literal(diskID))
```

Simple deletion of attributes to table with the appropriate exception check.
The RamSizeOfDisk view takes care of reaccumulating the ram sizes of each disk.

- def averageFileSizeOnDisk(diskID: int) -> float:

```
query = sql.SQL("""
                    BEGIN;
                    SELECT AVG(Files.size_needed)
                    FROM Files, FilesOfDisk
                    WHERE Files.id = FilesOfDisk.File_id
                        AND FilesOfDisk.Disk_id = {disk_id};

                    """).format(disk_id=sql.Literal(diskID))
```

We use avg on Files.size_needed when crossing Files and FilesOFDisk and grabbing only relevant lines to the file requested.

- def diskTotalRAM(diskID: int) -> int:

```
query = sql.SQL("""
                BEGIN;
                SELECT totalRAMSize
                FROM RAMSizeOFDisk
                WHERE RAMSizeOFDisk.Disk_id = {disk_id};
                """).format(disk_id=sql.Literal(diskID))
```

Simple selection of attributes from view with the appropriate exception check.

- def getCostForType(type: str) -> int:

```
query = sql.SQL("""
                BEGIN;
                SELECT SUM(Disks.cost * Files.size_needed)
                FROM Disks, Files, FilesOfDisk
                WHERE Disks.id = FilesOfDisk.Disk_id
                    AND FilesOfDisk.File_id = Files.id
                    AND Files.type = {type};
                """).format(type=sql.Literal(type))
```

We use sum to accumulate the selected column (Disks.cost*Files.size_needed) when crossing Disks, Files and FilesOFDisk and grabbing only relevant lines to the file type requested.

- def getFilesCanBeAddedToDisk(diskID: int) -> List[int]:

```
query = sql.SQL("""
                    BEGIN;
                    SELECT DISTINCT potentialFilesForDisk.file_id AS id
                    FROM potentialFilesForDisk
                    WHERE (potentialFilesForDisk.disk_id = {disk_id})
                    ORDER BY id DESC
                    LIMIT 5;
                    """).format(disk_id=sql.Literal(diskID))
```

Simple selection of attributes from view with the appropriate exception check.

- def getFilesCanBeAddedToDiskAndRAM(diskID: int) -> List[int]:

```
query = sql.SQL("""BEGIN;
                    SELECT DISTINCT Files.id AS id
                    FROM Disks, Files, RAMSizeOFDisk
                    WHERE (Files.size_needed <= Disks.free_space
                        AND Disks.id = {disk_id})
                       AND (Files.size_needed <= RAMSizeOFDisk.totalRAMSize
                        AND RAMSizeOFDisk.Disk_id = {disk_id})
                    ORDER BY id ASC
                    LIMIT 5;
                    """).format(disk_id=sql.Literal(diskID))
```

We select the files.id from the cross between files, disk and RamSizeOfDisk.
This way we can use both conditions.
We chose to not use the former view because the crossing with RamSizeOfDisk was not ideal.

- def isCompanyExclusive(diskID: int) -> bool:

```
query = sql.SQL("""
                BEGIN;
                INSERT INTO DisksCheck(id)
                VALUES({disk_id});
                DELETE FROM DisksCheck WHERE id={disk_id};
                SELECT DISTINCT RAMs.company
                FROM Disks, RAMSOFDisk, RAMs
                WHERE (Disks.id = RAMSOFDisk.Disk_id
                       AND Disks.id = {disk_id}
                       AND RAMSOFDisk.RAM_id = RAMs.id
                       AND RAMs.company != Disks.company);
```

First, we use DisksCheck to check if the id exists, if not we get a foreign key exception and can return false.
Then, we can select all distinctively different companies of the Disks RAMs that are in turn different then the Disks Company. If we get an empty table, we know that the disk is company exclusive.

- def getConflictingDisks() -> List[int]:

```
query = sql.SQL("""BEGIN;
                SELECT DISTINCT FOD1.disk_id AS id
                FROM FilesOFDisk AS FOD1, FilesOFDisk AS FOD2
                WHERE (FOD1.disk_id != FOD2.disk_id
                       AND FOD1.file_id = FOD2.file_id)
                ORDER BY id ASC;
                """)
```

When we cross FilesOfDisk with itself we can get a relation between all the files and all the files.
Then we can filter out all crossings of files with themselves.
Lastly, we filter out all crossings of files with different disks and we remain only with conflicting Disks.

- def mostAvailableDisks() -> List[int]:

```
query = sql.SQL("""BEGIN;
                SELECT potentialFilesForDisk.disk_id AS disk_id, COUNT(potentialFilesForDisk.file_id) as filesCount, Disks.speed
                FROM potentialFilesForDisk, Disks
                WHERE (potentialFilesForDisk.disk_id = Disks.id)
                GROUP BY potentialFilesForDisk.disk_id, Disks.speed
                ORDER BY filesCount DESC, speed DESC, disk_id ASC
                LIMIT 5;
                """)
```

We count for each disk the number of files that can fit in its free_space (noted in the potentialFilesForDisk view).

- def getCloseFiles(fileID: int) -> List[int]:

```
query = sql.SQL(""" BEGIN;
                SELECT shared_file_id
                FROM isclosefiles
                WHERE isClose = true
                       AND file_id = {file_id}
                ORDER BY shared_file_id ASC
                LIMIT 10;
                """).format(file_id=sql.Literal(fileID))
```

From the isCloseFiles view we can filter in only the close files to the requested file.
The views did all the job for us.