CHANTHRABOUTH-LIEBBE Simon
DEL CAMPO David
BENABDELLAH Younes
ING2 GIA 2

Traitement Avancée des Données

Rapport de projet

Optimisation de la base GLPI

Github : https://github.com/churre-cytech/TAD_project

Professeur : Redouane Bouhamoum
Année universitaire : 2024/2025

3 Avril 2025

Introduction

Contexte

Ce projet s'inscrit dans le cadre du module "Traitement Avancée des Données" pour nous, étudiant en ING2 à CY Tech. L'objectif principal de ce projet est de retravailler, repenser et d'optimiser une partie de la base de données de l'outil open-source GLPI (Gestionnaire Libre de Parc Informatique) pour répondre aux besoins spécifiques de CY Tech dans la gestion de son parc informatique sur les deux sites : Cergy et Pau. Le projet vise donc à améliorer les performances, la scalabilité et la gestion des données par rapport à une implémentation "brute" de GLPI, en appliquant les notions et concepts étudiés dans le module.

Cadrage du projet

Le périmètre du projet, qui nous a été précisé dans l'énoncé, se concentre sur plusieurs aspects essentiels d'un parc informatique. Il inclut la gestion des matériels informatiques, avec l'inventaire des équipements, le suivi de leur état (actif, en maintenance ou décommissionné), leur affectation aux utilisateurs, ainsi que les dates d'achat. La gestion des utilisateurs est également un aspect important, avec les comptes, les informations personnelles, les rôles et les permissions associés à chaque profil.

Le projet couvre également la gestion de la structure réseau, notamment les définitions des réseaux propres à chaque site, la gestion des adresses IP et leur affectation aux matériels. Il y a également un aspect qui concerne la gestion des tickets de support, permettant la création, le suivi, la priorisation et la résolution des incidents ou des demandes qui sont liés au parc informatique. Enfin, il y a également la gestion multi-sites qui est prise en compte avec la répartition des ressources entre les sites de Cergy et de Pau.

Méthodologie

La méthodologie suivie repose sur deux choses. La première est le reverse engineering. C'est-à-dire analyser la structure de la base GLPI existante via sa documentation et son code source (disponible sur Github) L'objectif est de comprendre son fonctionnement, identifier ses forces et ses faiblesses potentielles dans le contexte de CY Tech. Cette étape permet de définir les exigences pour la conception de la nouvelle base de données.

Et elle repose également sur une conception qui intègre les concepts avancées de Oracle Database, à savoir : une gestion fine des utilisateurs et rôles, utilisation de tablespaces dédiés, implémentation de clusters et d'index pertinents, création de vues (logiques et matérialisées) pour simplifier l'accès aux données, développement en PL/SQL (avec des procédures, fonctions, triggers), mise en place d'une architecture de Bases De Données Réparties (BDDR) pour la gestion multi-sites entre Cergy et Pau, ainsi qu'une analyse des plans d'exécution pour l'optimisation des requêtes.

Reverse engineering

Étapes réalisées

Notre analyse initiale s'est basée sur la documentation officielle de GLPI, avec l'analyse de la structure de la base de données réalisée via les fichiers SQL disponible dans le dépôt Github GLPI. L'objectif de l'analyse de ces fichiers était d'identifier les tables clés correspondant au périmètre défini. Les tables retenues sont les suivantes : Tables liées aux ordinateurs (**glpi_computers**), périphériques (**glpi_peripherals**), logiciel (**glpi_softwares**). Ainsi que les tables liées aux utilisateurs (**glpi_users**), groupes (**glpi_groups**), profils (**glpi_profiles**). Il y a également les tables réseaux (**glpi_networks**, **glpi_networkports**), les tickets (**glpi_tickets**) et les tables de localisations/entités (**glpi_locations**, **glpi_entities**).

Cette première analyse nous a permis de comprendre les relations entre les entités et la manière dont GLPI gère l'inventaire et le support.

Problèmes et limitations

Bien que GLPI soit un outil bien pensé, constamment amélioré par la communauté grâce au fait qu'il soit open-source, son modèle de données très générique peut présenter des limitations dans notre contexte.

Pour des parcs informatiques de grande taille et avec un historique important, certaines requêtes complexes (ex: un rapport qui croise les informations sur les utilisateurs, matériels et localisations) peuvent très vite devenir lente et lourde pour le système sans optimisation.

La gestion des sites présente dans GLPI, bien que fonctionnelle, peut être simplifiée dans notre cas avec l'usage de Base De Données Réparties (BDDR). Cette approche peut offrir une meilleure séparation logique et physique des données par site. Cela peut améliorer les performances locales et globales du système.

GLPI est compatible avec différents Système de Gestion de Base de Données (SGBD), tels que MySQL, MariaDB, PostgreSQL etc. Bien que cela soit un avantage pour un projet open-source, GLPI n'exploite pas nativement les fonctionnalités spécifiques à Oracle Database (telles que l'usage de PL/SQL, tablespaces spécifiques, clusters etc.) qui pourraient améliorer les performances et la gestion de la base de données.

Justifications des choix de conception

Face à ces constats, nos choix de conception pour la nouvelle base de données ont été guidés par plusieurs objectifs.

Tout d'abord, réduire le nombre de tables en se concentrant sur le périmètre essentiel défini. Par exemple, regrouper les différents types de matériels sous une table **ASSET** avec **ASSET_TYPE**, plutôt qu'avoir une table séparée pour chaque type différent.

Utiliser les fonctionnalités qui nous sont mises à disposition dans Oracle Database (index, clusters, vues matérialisées, tablespaces) pour améliorer les performances des requêtes identifiées comme potentiellement lourdes ou lentes.

Implémenter une solution de Base De Données Réparties (BDDR) pour gérer nativement les sites de Cergy et Pau. Cela permet d'apporter une meilleure isolation et de meilleures performances locales.

Et enfin, intégrer la logique métier (validations et automatisations) directement dans la base de données via des triggers et procédures/fonctions PL/SQL afin de garantir l'intégrité et la cohérence des données.

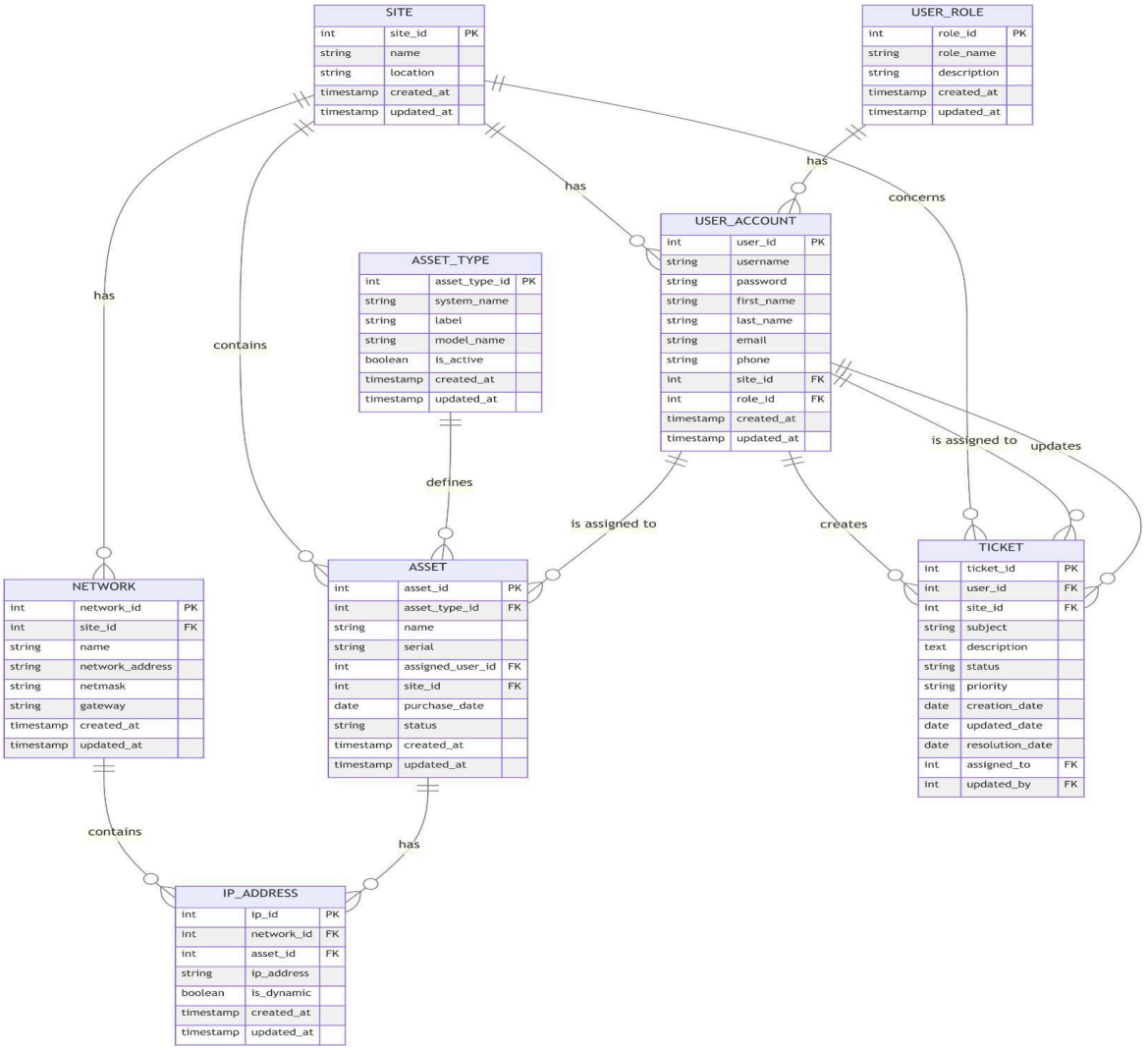
Modélisation et conception de la nouvelle base de données

Structure générale

La nouvelle structure de la base de données a été conçue pour être plus concise et optimisée pour les besoins identifiés. Les tables qui la composent sont les suivantes :

- **SITE** : Gère les informations sur les sites (Cergy et Pau).
- **USER_ROLE** : Définit les différents rôles possibles.
- **USER_ACCOUNT** : Stocke les informations sur les utilisateurs, liées à un site et un rôle.
- **ASSET_TYPE** : Catégorise les types de matériel (PC, souris, claviers, tablettes…)
- **ASSET** : Représente un matériel informatique spécifique, lié à un type, un site et potentiellement un utilisateur.
- **NETWORK** : Définit les réseaux IP pour chaque site.
- **IP_ADDRESS** : Gère les adresses IP, leur appartenance à un réseau et leur éventuelle assignation à un ASSET.
- **TICKET** : Gère les demandes de support ou incidents.
-

Voici un MCD (Modèle Conceptuel de Données) afin de mieux visualiser la structure de notre base de données ainsi que les relations entre les différentes tables :



Génération des données

Afin de pouvoir remplir notre base de données (et donc plus tard, tester la structure et évaluer les performances de manière réaliste), il était essentiel de remplir notre base avec un volume important de données. C'est à cela que sert le script '**seed_data.sql**', il commence par insérer les données statiques dans les tables comme 'SITE' (avec Cergy, Pau) et 'USER_ROLE' (les différents rôles définis).

Ensuite, il utilise des blocs PL/SQL ainsi que le package **'DBMS_RANDOM'**, cela nous permet de générer dynamiquement des données de test (en utilisant des valeurs aléatoires) pour les tables. Par exemple, on a une boucle FOR qui insère 4500 utilisateurs dans 'USER_ACCOUNT' avec des noms, prénoms, emails et numéros de téléphone générés aléatoirement, en leur assignant un rôle et un site (avec une répartition équilibrée entre Cergy et Pau). De même, d'autres boucles génèrent un grand nombre d'actifs 'ASSET' de différents types 'ASSET_TYPE', en leur attribuant chacun un numéro de série, une date d'achat aléatoire, un statut, un site et en les assignant parfois à des utilisateurs existants. La génération concerne également les tables 'NETWORK' et 'IP_ADDRESS' avec environ 25 réseaux par site et des milliers d'adresses IP. Il y a également la table 'TICKET' où l'on y génère 5000 tickets avec des sujets, descriptions, priorités, statuts, liés à des utilisateurs et potentiellement assignés à des techniciens du même site). Tout cela a été permis par l'usage de 'DBMS.RANDOM' qui nous a permis d'obtenir des données variées et plausibles pour simuler un environnement proche d'un cas réel.

Explication des concepts avancés utilisés

Utilisateurs et rôles

Nous avons mis en place une gestion fine des accès via les tables USER_ACCOUNT et USER_ROLE. Des scripts SQL distincts (users_roles-CERGY.sql, users_roles-PAU.sql) créent les utilisateurs et rôles spécifiques à chaque site. Cela donne : ROLE_ADMIN_CERGY ROLE_IT_TECH_CERGY, ROLE_NETWORK_TECH_CERGY, ROLE_ACADEMIC_ADMIN_CERGY, ROLE_STUDENT_TEACHER_CERGY (de même avec Pau).

Les privilèges SELECT, INSERT, UPDATE, DELETE sont accordés aux rôles sur les tables pertientnes (par exemple, le rôle ROLE_IT_TECH peut gérer ASSET et TICKET, le rôle ROLE_NETWORK_TECH gère NETWORK et IP_ADDRESS).

On a également des utilisateurs spécifiques par site, par exemple ADMIN_CERGY et IT_TECH_PAU, avec les rôles correspondants. Les préfixes C## ne sont plus utiles ici car les bases de Cergy et Pau sont deux bases distinctes dans Oracle Database.

Cette approche nous permet de garantir une séparation claire des responsabilités en fonction du rôle et du site de l'utilisateur. Elle facilite la maintenance et les évolutions des politiques d'accès en centralisant les droits au niveau des rôles plutôt que des comptes individuels, ce qui renforce la cohérence et réduit les risques d'erreurs dans notre base.

Organisation des Tablespaces

Pour une meilleure gestion du stockage et des performances entrées/sorties, nous avons créé des tablespaces dédiés, notamment pour séparer les données des index et des vues matérialisées.

On a la tablespace USER par défaut, qui contient les données principales. Ainsi que les tablespaces cergy_indexes et pau_indexes qui sont des tablespaces dédiés au stockage des index pour chaque site (voir index_tables-CERGY.sql et index_tables-PAU.sql). Les index créés manuellement, ainsi que ceux générés par les contraintes (PK, UK) sont déplacés dans ce tablespace via la commande ALTER INDEX … REBUILD TABLESPACE… .

Et enfin nous avons les tablespaces ts_matviews_cergy et ts_matviews_pau. Ces tablespaces sont dédiées au stockage des vues matérialisées pour chaque site (voir view-CERGY.sql et view-PAU.sql).

Cette répartition en trois tablespaces permet une maintenance plus simple et améliore les performances en répartissant les entrées/sorties.

Cluster et Index

L'optimisation des requêtes est un objectif clé dans ce projet, et elle est atteinte par la création stratégique d'index et de clusters. Commençons par présenter les index utilisés dans notre projet.

On a tout d'abord les index qui sont créés automatiquement par Oracle lors de la définition des contraintes de clé primaire (PK) et d'unicité (UQ) sur nos tables. Ces index garantissent l'unicité et accélèrent les recherches basées sur ces clés.

Ensuite, nous avons défini manuellement des index sur certaines clés étrangères (FK) et sur des colonnes fréquemment utilisées dans les clauses "WHERE" ou "JOIN" de nos requêtes. Par exemple, des index comme 'idx_asset_purchase_date_cergy' sur la date d'achat des matériels 'ASSET.purchase_date', 'udx_asset_assigned_user_id_cergy' sur l'utilisateur asigné 'ASSET.assigned_user_id', 'idx_ipasset_asset_cergy' sur l'ID du matériel dans la table 'IP_ADDRESS', ou encore 'idx_ticket_user_cergy' et 'idx_ticket_assigned_to_status_cergy' sur la table 'TICKET' ont été créés pour accélérer les recherches et jointures courantes liées à la gestion du matériel et du support. Afin d'optimiser la gestion du stockage et potentiellement les performances; tous ces index ont été déplacés vers des tablespaces dédiés ('cergy_indexes' et 'pau_indexes') à l'aide de la commande 'ALTER INDEX … REBUILD TABLESPACE …'. Cette séparation physique des données et des index facilite également la maintenance.

En complément des index, nous avons également utilisé des clusters de tables pour améliorer les performances des jointures fréquentes entre certaines tables. Un cluster permet de stocker physiquement les lignes de plusieurs tables qui partagent une même valeur de colonne de cluster dans les mêmes blocs de données. Nous avons implémenté un cluster 'network_cluster' basé sur la colonne 'network_id'. Ce cluster regroupe les tables 'clustered_network' et 'clustered_ip_address', ce qui optimise les requêtes qui soignent un réseau à ses adresses IP associées, car les données liées sont stockées "côte à côte", ce qui réduit les entrées/sorties. Un index 'idx_cluster_network_id' a également été créé sur la clé du cluster pour un accès rapide. De même avec 'asset_type_cluster' sur 'asset_type_id', elle regroupe les tables 'clustered_asset_type' et 'clustered_asset'. Ces clusters sont

également stockés dans un tablespace dédiée ‘cergy_clusters’ (et ‘pau_clusters’ pour le site de Pau).

Vues logiques et matérialisées

Pour simplifier l’accès aux données et optimiser les requêtes récurrentes, nous avons implémenté plusieurs vues logiques et matérialisées en fonction des besoins métiers des différents rôles utilisateurs. Par exemple, nous avons créé la vue logique **view_state_asset_user** qui affiche l’état du matériel affecté à chaque utilisateur. Cette vue est accessible uniquement au rôle **ROLE_IT_TECH_CERGY**, qui regroupe les utilisateurs **IT_TECH_CERGY**. Elle permet aux techniciens d’identifier rapidement le matériel en panne et d'intervenir de manière plus efficace. En complément, d'autres vues logiques ont été créées pour simplifier les tâches spécifiques de chaque profil accédant à la base de données, en masquant la complexité des jointures et en offrant un accès structuré aux données pertinentes.

Concernant les vues matérialisées, nous avons mis en place la vue **vm_unresolved_tickets_details**, qui permet d’afficher tous les tickets non résolus avec les informations associées à l’utilisateur et au site. Cette vue est également réservée au rôle **ROLE_IT_TECH_CERGY**. Elle est rafraîchie manuellement à la demande, chaque fois qu’un technicien souhaite consulter l’état des tickets pour prioriser les réparations à effectuer. Nous avons également conçu la vue matérialisée **materialized_view_students_info**, utile pour le rôle **ROLE_ACADEMIC_ADMIN_CERGY**. Elle donne accès à la liste complète des étudiants de CY Tech, avec leur ID, prénom, nom, pseudo et email. Elle est automatiquement mise à jour après chaque commit sur la table **USER_ACCOUNT**, garantissant ainsi une information toujours à jour. Enfin, la vue **materialized_view_asset_to_replace** a été développée pour les techniciens, afin d’identifier le matériel trop ancien (plus de 4 ans) ou à remplacer. Elle se met à jour **une fois par jour**, ce qui permet aux équipes techniques d’anticiper les renouvellements de matériel de manière proactive.

Script en PL/SQL

L’utilisation de PL/SQL a été très importante pour intégrer la logique métier directement dans la base de données. Son utilisation nous permet de garantir la cohérence et l’intégrité des données indépendamment de l’application qui pourrait y accéder. Nous avons développé plusieurs triggers sur les tables critiques. Par exemple, sur la table ‘TICKET’, le trigger ‘trg_ticket_status_update’ permet de mettre automatiquement à jour la date de modification ‘update_date’ et renseigne la date de résolution dans ‘resolution_date’ lorsqu’un ticket passe au statut ‘closed’. Le trigger ‘trg_ticket_high_priority_assigned’ s’assure qu’un technicien est obligatoirement assigné à un ticket si sa priorité est ‘high’. On a également un autre trigger, ‘trg_ticket_site_consistency’, qui vérifie que le créateur du ticket et le technicien assigné appartiennent bien au même site que celui indiqué dans le ticket. Enfin, ‘trg_ticket_audit’ enregistre les changements importants (modification de ‘status’) dans une table d’audit dédiée ‘TICKET_AUDIT’.

Dans la même idée, des triggers similaires ont été implémentées pour la table ‘ASSET’ via des triggers comme ‘trg_asset_status_check’ qui met à jour ‘updated_at’ et vérifie qu’un matériel décommissionné n’est pas assigné, ‘trg_asset_site_consistency’ qui

vérifie la cohérence du site entre l'asset et l'utilisateur assigné. Mais aussi 'trg_asset_purchase_date_check' qui empêche d'enregistrer une date d'achat future, 'trg_asset_protect_serial' qui interdit la modification du numéro de série une fois enregistré et 'trg_asset_audit' pour l'audit. D'autres triggers plus simples existent sur les tables 'USER_ACCOUNT', 'SITE', 'NETWORK', etc. principalement pour mettre à jour les colonnes concernant les dates de modifications.

En complément des triggers, nous avons également fait l'usage de procédures stockées et des curseurs PL/SQL pour générer des rapports ou effectuer des vérifications complexes. Le fichier 'cursor_ASSET_table.sql' contient par exemple une procédure 'report_maintenance_assets' qui utulise un curseur pour lister tous les matériels en maintenance depuis une longue période.

Gestion multi-sites via Bases De Données Réparties

Tout d'abord, il faut savoir que nous sommes partis sur un système de **base de données centralisée**, c'est-à-dire que tous les élèves, qu'ils soient de **Pau** ou de **Cergy**, sont enregistrés dans une seule base de données principale : **CY_TECH_CERGY**. Cependant, pour répondre à l'exigence de gestion multi-sites tout en gardant une vision globale, nous avons fait évoluer cette base vers une architecture de **Bases de Données Réparties (BDDR)**. Plutôt que de tout gérer dans une seule base via une colonne **site_id** (même si cette colonne existe pour la logique locale), nous avons mis en place deux bases Oracle physiquement distinctes, **CY_TECH_CERGY** et **CY_TECH_PAU**, chacune avec **son propre schéma**, ses **utilisateurs**, ses **rôles**, ses **tablespaces**, etc. Cette séparation physique permet une **meilleure isolation des données**, une **autonomie de gestion pour chaque site**, et potentiellement de **meilleures performances locales**.

Pour permettre la communication entre les deux bases, nous avons utilisé des **Database Links**. Un lien **pau_link** a été créé dans la base de Cergy pour accéder aux objets distants de Pau, et inversement, **cergy_link** a été créé dans la base de Pau. Ces liens contiennent toutes les informations nécessaires à la connexion (utilisateur, mot de passe, TNS). Une fois les liens créés, il a fallu recréer les rôles dans la base de Pau, comme cela avait été fait côté Cergy.

Nous avons ensuite géré la répartition des données entre les deux sites. Sur les **8 tables principales**, nous avons choisi d'en **répliquer 3** (**ASSET_TYPE**, **USER_ROLE**, **SITE**), car elles ne contiennent pas un gros volume de données et sont utiles pour des opérations fréquentes, comme identifier le type d'un appareil. Les **5 autres tables** ont été **fragmentées horizontalement** selon le **site_id** (par exemple, 1 = Cergy, 2 = Pau) : **USER_ACCOUNT**, **ASSET**, **TICKET** et **NETWORK**. Pour **IP_ADDRESS**, la fragmentation se fait selon **network_id**. Ce découpage permet de stocker localement les données spécifiques à chaque site.

Concernant l'insertion des données, **deux méthodes sont possibles** : soit on insère depuis Cergy dans la base de Pau via le database link, soit on demande depuis Pau les données à Cergy, ce qui est **plus coûteux** car cela implique une attente liée à la requête distante. Nous avons donc opté pour la **première méthode**, plus logique et plus pratique. Une fois que les données ont été insérées dans la base de Pau, il est essentiel de les

supprimer de la base de Cergy pour éviter la redondance. Cette suppression peut se faire depuis l'un ou l'autre des deux sites.

Enfin, il a fallu accorder les privilèges nécessaires aux nouveaux utilisateurs de la base de Pau, et recréer les index sur les tables locales pour assurer de bonnes performances.

Pour offrir une **vue consolidée des données** des deux sites à certains utilisateurs (comme les administrateurs globaux ou la direction), nous avons mis en place des **vues globales**, préfixées par **GLOBAL_VIEW_**, telles que **GLOBAL_VIEW_STATE_ASSET_USER**, **GLOBAL_VIEW_NETWORK_USAGE** ou encore **GLOBAL_VIEW_TICKET_STATISTICS**. Ces vues sont définies dans la base de Cergy (ou potentiellement dans une base d'administration dédiée) et utilisent les **database links** pour interroger les données des deux sites via des opérations **UNION ALL**. Par exemple, **GLOBAL_VIEW_STATE_ASSET_USER** combine les résultats de **users_materials_cergy** et **users_materials_pau@pau_link**. Ces vues globales sont stockées dans un tablespace dédié, **ts_global_views**, pour une meilleure organisation.

Performance mesurées et améliorations :

Choix de conception

L'architecture et les choix de conception pour notre base de données ont été pensés pour qu'elle soit la plus performante possible. Plusieurs concepts ont été appliqués pour atteindre notre objectif.

Premièrement, l'organisation du stockage via des tablespaces dédiés ('cergy_indexes', 'pau_indexes', 'ts_matviews_cergy', 'ts_matviews_pau', 'cergy_clusters', 'pau_clusters', 'ts_global_views', 'CY_TECH_CERGY_DATA', 'CY_TECH_PAU_DATA') permet de séparer les différents types d'objets (tables, index, vues matérialisées, clusters) et de répartir les opérations sur différents fichiers de données, ce qui réduit la pression sur le stockage. Deuxièmement, l'utilisation d'index (automatiques sur PK et UQ, manuels sur les FK et nos colonnes de filtre et jointure comme précisé plus tôt) accélère significativement la recherche et la récupération des données en évitant de parcourir complètement les tables. Troisièmement, l'implémentation de clusters de tables ('network_cluster', 'asset_type_cluster') optimise les jointures entre les tables fréquemment accédées ensemble en localisant physiquement leurs données côte à côte sur le disque, ce qui minimise les entrées/sorties pour récupérer les informations. Ensuite, les vues matérialisées offrent un gain de performance notable pour les requêtes complexes ou les agrégations fréquentes en stockant les résultats, ce qui transforme des opérations à l'origine coûteuses en simples lectures de tables. Enfin, l'architecture de base de données répartie, où nous dédions une base par site, améliore les performances locales pour les utilisateurs de chaque site, car la majorité de leurs requêtes concernent des données physiquement proches.

Tests de performance

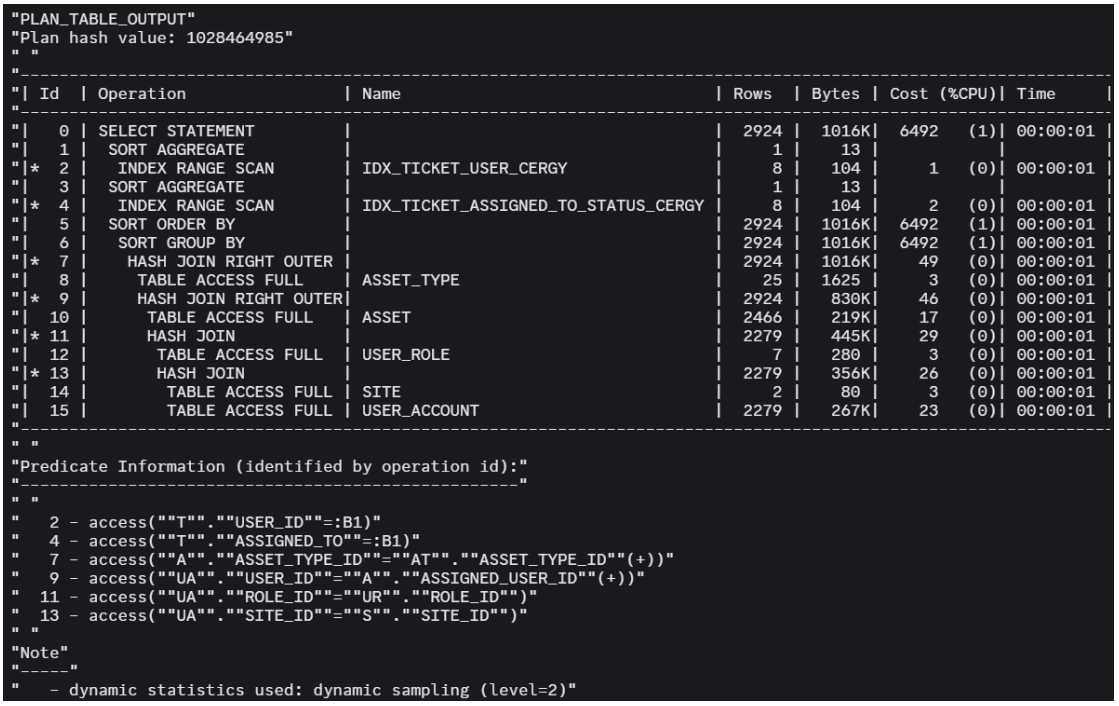
L'évaluation de l'efficacité des optimisations que nous avons mises en place, notamment grâce à l'utilisation d'**index** et de **clusters**, a été réalisée à l'aide de l'outil **EXPLAIN PLAN** d'Oracle. Cet outil permet d'analyser une requête SQL en décrivant le plan **d'exécution** choisi par l'optimiseur Oracle, tout en détaillant les opérations effectuées et en estimant le **coût de chaque étape** ainsi que celui de la requête dans sa globalité.

```
################################################
## SANS INDEX
################################################
"PLAN_TABLE_OUTPUT"
"Plan hash value: 423357554"
" "
"-------------------------------------------------------------------------------"
"| Id  | Operation          | Name  | Rows  | Bytes | Cost (%CPU)| Time     |"
"-------------------------------------------------------------------------------"
"|   0 | SELECT STATEMENT   |       |     4 |   812 |   102   (0)| 00:00:01 |"
"|*  1 |  TABLE ACCESS FULL| ASSET |     4 |   812 |   102   (0)| 00:00:01 |"
"-------------------------------------------------------------------------------"
" "
"Predicate Information (identified by operation id):"
"---------------------------------------------------"
" "
"   1 - filter(""ASSIGNED_USER_ID""=14765)"
" "
"Note"
"-----"
"   - dynamic statistics used: dynamic sampling (level=2)"
```

```
############################################
## AVEC INDEX
############################################
"PLAN_TABLE_OUTPUT"
"Plan hash value: 749125677"
" "
"-----------------------------------------------------------------------------------------------------------"
"| Id  | Operation                           | Name                    | Rows  | Bytes | Cost (%CPU)| Time     |"
"-----------------------------------------------------------------------------------------------------------"
"|   0 | SELECT STATEMENT                    |                         |     2 |   406 |     4   (0)| 00:00:01 |"
"|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| ASSET                   |     2 |   406 |     4   (0)| 00:00:01 |"
"|*  2 |   INDEX RANGE SCAN                  | IDX_ASSET_ASSIGNED_USER_ID |     2 |       |     1   (0)| 00:00:01 |"
"-----------------------------------------------------------------------------------------------------------"
" "
"Predicate Information (identified by operation id):"
"---------------------------------------------------"
" "
"   2 - access(""ASSIGNED_USER_ID""=14765)"
" "
"Note"
"-----"
"   - dynamic statistics used: dynamic sampling (level=2)"
```

Ce test concerne l'ajout d'un index sur la colonne **NAME** de la table **ASSET**. Sans index, on observe l'utilisation d'un **TABLE ACCESS FULL**, c'est-à-dire un scan complet de la table. Le coût d'exécution est relativement élevé, avec une valeur de **102** dans ce cas précis, car toutes les lignes de la table doivent être parcourues pour trouver celles où NAME = '**Asset_DKSZF**'.

Avec l'ajout de l'index **idx_asset_assigned_user_id**, la requête utilise un **INDEX RANGE SCAN**, suivi d'un **TABLE ACCESS BY INDEX ROWID**. Cela permet de **réduire le coût d'exécution d'environ 97 %**, passant de **102 à seulement 3**. Le coût de l'utilisation de l'index seul est de **1**, ce qui permet un **accès direct et ciblé aux lignes pertinentes** sans devoir scanner toute la table. Dans le cas où cette requête serait exécutée sur un volume de données encore plus important (comme après plusieurs années d'utilisation de la base), **les gains de performance seraient d'autant plus significatifs et non négligeables**.



```
"PLAN_TABLE_OUTPUT"
"Plan hash value: 1028464985"
" "
"-----------------------------------------------------------------------------------------------------------------------"
"| Id  | Operation                | Name                                 | Rows  | Bytes | Cost (%CPU)| Time     |
"-----------------------------------------------------------------------------------------------------------------------"
"|   0 | SELECT STATEMENT         |                                      |  2924 |  1016K|  6492   (1)| 00:00:01 |
"|   1 |  SORT AGGREGATE          |                                      |     1 |    13 |            |          |
"|*  2 |   INDEX RANGE SCAN       | IDX_TICKET_USER_CERGY                |     8 |   104 |     1   (0)| 00:00:01 |
"|   3 |  SORT AGGREGATE          |                                      |     1 |    13 |            |          |
"|*  4 |   INDEX RANGE SCAN       | IDX_TICKET_ASSIGNED_TO_STATUS_CERGY  |     8 |   104 |     2   (0)| 00:00:01 |
"|   5 |  SORT ORDER BY           |                                      |  2924 |  1016K|  6492   (1)| 00:00:01 |
"|   6 |   SORT GROUP BY          |                                      |  2924 |  1016K|  6492   (1)| 00:00:01 |
"|*  7 |    HASH JOIN RIGHT OUTER |                                      |  2924 |  1016K|    49   (0)| 00:00:01 |
"|   8 |     TABLE ACCESS FULL    | ASSET_TYPE                           |    25 |  1625 |     3   (0)| 00:00:01 |
"|*  9 |     HASH JOIN RIGHT OUTER|                                      |  2924 |   830K|    46   (0)| 00:00:01 |
"|  10 |      TABLE ACCESS FULL   | ASSET                                |  2466 |   219K|    17   (0)| 00:00:01 |
"|* 11 |      HASH JOIN           |                                      |  2279 |   445K|    29   (0)| 00:00:01 |
"|  12 |       TABLE ACCESS FULL  | USER_ROLE                            |     7 |   280 |     3   (0)| 00:00:01 |
"|* 13 |       HASH JOIN          |                                      |  2279 |   356K|    26   (0)| 00:00:01 |
"|  14 |        TABLE ACCESS FULL | SITE                                 |     2 |    80 |     3   (0)| 00:00:01 |
"|  15 |        TABLE ACCESS FULL | USER_ACCOUNT                         |  2279 |   267K|    23   (0)| 00:00:01 |
"-----------------------------------------------------------------------------------------------------------------------"
" "
"Predicate Information (identified by operation id):"
"---------------------------------------------------"
" "
"   2 - access(""T"".""USER_ID""=:B1)"
"   4 - access(""T"".""ASSIGNED_TO""=:B1)"
"   7 - access(""A"".""ASSET_TYPE_ID""=""AT"".""ASSET_TYPE_ID""(+))"
"   9 - access(""UA"".""USER_ID""=""A"".""ASSIGNED_USER_ID""(+))"
"  11 - access(""UA"".""ROLE_ID""=""UR"".""ROLE_ID"")"
"  13 - access(""UA"".""SITE_ID""=""S"".""SITE_ID"")"
" "
"Note"
"-----"
"   - dynamic statistics used: dynamic sampling (level=2)"
```

Sur la capture d'écran suivante, on peut voir le rapport généré par **EXPLAIN PLAN** pour une requête complexe comportant plusieurs jointures et agrégations. Deux index sont utilisés : **idx_ticket_user_cergy** et **idx_ticket_assigned_to_status**, ce qui montre que notre table **TICKET** bénéficie d'une indexation optimisée.

Conclusion

Ce projet nous a permis de mettre en pratique de manière concrète l'ensemble des notions avancées abordées en cours, en particulier autour de la modélisation de bases de données, des performances, et de la gestion multi-sites. À travers la refonte de l'architecture de la base GLPI pour CY Tech, nous avons fait des choix techniques orientés vers l'efficacité, la sécurité et la scalabilité. La mise en place d'un système de Bases de Données Réparties, combinée à l'utilisation intelligente de vues logiques et matérialisées, a permis de structurer les données de manière claire et optimisée selon les besoins de chaque site (Cergy et Pau), tout en gardant une vision centralisée là où c'était nécessaire.

L'usage des rôles et privilèges, des triggers, procédures PL/SQL, des tests de performance, ou encore des plans de requêtes nous a également permis de garantir une base à la fois robuste, performante et bien organisée. Le projet nous a aussi appris à faire des compromis techniques entre centralisation et distribution, tout en respectant les contraintes métier spécifiques.

En résumé, ce mini-projet a été une vraie opportunité de consolider nos compétences en base de données avancée, tout en travaillant sur un cas concret avec des problématiques proches du monde professionnel. C'est également une excellente préparation pour le partiel à venir, car il nous a permis de revoir et d'appliquer en profondeur toutes les notions clés du module.