Data Base Project

Mathieu Lemain, Egor Semenov

Avril 2025

Info4B - Projet Sujet: Base de données orientée objets

Étudiant 1: Egor Semenov-Tyan-Shanskiy, egor_semenov-tyan-shanskiy@etu.u-bourgogne.fr
Étudiant 2: Mathieu Lemain, mathieu_lemain@etu.u-bourgogne.fr

GitHub: https://github.com/HobbitTheCat/dataBase.git

Analyse du problème

Dans le cadre de ce projet, nous avons choisi le sujet sur le développement d'une base de données. Cette tâche représente non seulement un défi technique, mais aussi une occasion d'étudier en profondeur les principes de la programmation système, du multithreading et de la gestion des structures de fichiers. Le projet vise à développer un système de gestion de base de données orientée objet, permettant de sauvegarder des objets dans des fichiers et de les rechercher selon divers critères. Cela impose des restrictions sur le choix du format de stockage des données : il doit permettre une recherche rapide sans nécessiter la désérialisation complète de tous les objets. De plus, le système suppose une architecture client-serveur, incluant un serveur, un client textuel et une bibliothèque cliente. Cela nécessite la création d'un mécanisme de communication entre les clients et le serveur, ainsi que la mise en œuvre de l'authentification des utilisateurs et de la gestion des droits d'accès. Au cours du développement, il sera nécessaire de prendre en compte la manière dont ces exigences influencent le choix des structures de données, des formats de sérialisation et de l'organisation de l'interaction réseau.

Chapitre 1 dans lequel nous choisissons une organisation du stockage des objets

Les premières approches.

> L'objectif de ce projet est de fournir un service système de gestion de base de données qui, au travers du mécanisme de sérialisation, permet de sauvegarder des collections d'objets dans des fichiers, mais aussi de fournir au programmeur des fonctions pour rechercher des objets selon des critères numériques ou textuels.

Cela implique que le système permette de rechercher efficacement des objets en fonction de leurs attributs, ce qui influence directement le choix de la structure de stockage. Examinons les options possibles.

Option 1 : Un objet — un fichier.

Dans cette approche, chaque objet est stocké dans un fichier séparé, et les classes sont représentées par des répertoires.

Avantages :

- Implémentation simple : chaque objet est un fichier séparé, et chaque classe est un dossier.
- La suppression des objets est simple : on supprime le fichier — on supprime l'objet.

Inconvénients :

- Complexité de la recherche : pour effectuer une recherche par attribut, il faudra ouvrir et désérialiser tous les fichiers.
- Impossible de créer des relations entre les objets : un objet — un fichier, ce qui empêche d'établir des liens entre les tables.
- Structure de fichiers encombrante : avec un grand nombre d'objets, une énorme quantité de fichiers et de dossiers sera créée.

Conclusion : Cette option est trop inefficace en raison des problèmes de recherche et de la structure peu pratique.

Option 2 : Une classe, un fichier.

Dans ce cas, tous les objets d'une même classe sont stockés dans un seul fichier.

Avantages :

- Meilleure organisation de la structure des fichiers.
- Implémentation simple grâce à la sérialisation standard.

Inconvénients :

- La recherche par attribut reste complexe : il faut charger tout le fichier en mémoire.
- Impossible de créer des relations entre les objets.
- Un nouveau problème apparaît : comment séparer les objets à l'intérieur du fichier et comment les supprimer sans fragmentation ?

Conclusion : Cette option résout le problème du nombre de fichiers, mais ne supprime pas le principal inconvénient, à savoir la complexité de la recherche.

Option 3 : Base de données - un fichier

Dans cette option, tous les objets de toutes les classes sont stockés dans un seul fichier, en utilisant des mécanismes de sérialisation standard.

Avantages :

- Meilleure organisation de la structure des fichiers.
- Gestion simplifiée des fichiers (pas de nombreux petits fichiers).

Inconvénients :

- La recherche reste inefficace : il est toujours nécessaire de charger entièrement le fichier pour extraire un objet.
- Le problème de la fragmentation est amplifié : les objets supprimés laissent des espaces vides dans le fichier, difficiles à réutiliser efficacement.

Conclusion : Cette option offre une meilleure organisation des données, mais ne résout pas le problème de la recherche et complique la gestion de l'espace libre.

Conclusion

Les options étudiées ont mis en évidence plusieurs problèmes clés :

- La recherche des données par attribut reste complexe en raison de la désérialisation complète.
- La fragmentation des données rend la gestion de la mémoire inefficace.
- Il est impossible de créer des relations entre les objets sans charger complètement les données.

La plupart de ces problèmes sont causés par l'utilisation du mécanisme de sérialisation standard. Le chargement complet du fichier en mémoire devient impossible à mesure que la quantité de données augmente.

Architecture appliquée

Pour résoudre les problèmes identifiés, nous avons abandonné la sérialisation standard et opté pour une organisation des données par pages. Dans cette approche, le fichier de la base de données est divisé en pages de taille fixe, chargées en mémoire au besoin. Cela permet de :

- Travailler avec de grandes quantités de données en chargeant uniquement les pages nécessaires.
- Améliorer la recherche grâce à un accès efficace aux attributs et à l'absence de besoin de désérialiser les attributs.
- Minimiser le problème de la fragmentation grâce à une gestion réfléchie des pages.

Choix de la taille de la page

Pour organiser les données dans le fichier, un mécanisme de stockage par pages a été choisi. La taille de la page a été fixée à 4 Ko, ce qui est motivé par les facteurs suivants :

- 4 Ko est la taille standard de la page mémoire dans la plupart des systèmes d'exploitation, ce qui permet de travailler plus efficacement avec le cache.
- De nombreux systèmes de fichiers sont optimisés pour travailler avec des blocs de 4 Ko, ce qui peut améliorer les performances lors de l'accès au fichier.
- Cette taille représente un compromis entre le nombre de pages et les coûts de leur traitement : des pages plus petites augmentent les frais généraux, tandis que des pages plus grandes peuvent ralentir le traitement des petits objets.

Création des types de pages

Considérons un objet simple qui doit être stocké dans la base de données :

Ville :
- Nom
- Population

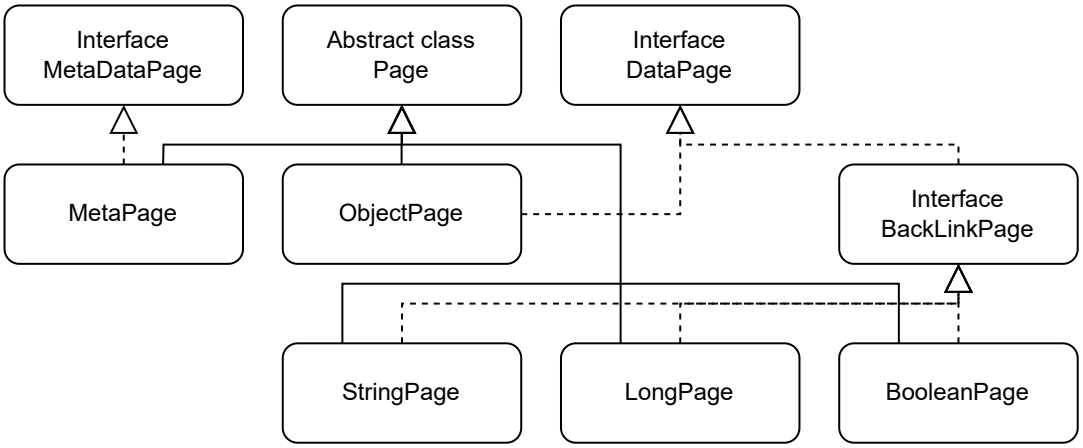

Figure 1: Page class diagram

Comment pouvons-nous stocker cet objet dans un fichier de manière à faciliter la recherche par attributs et permettre des références à d'autres objets ?

Première option : pages d'objets La première idée était d'allouer une page pour décrire les classes et des pages séparées pour stocker les objets. Cependant, cette méthode a conduit à plusieurs problèmes. Tout d'abord, il était impossible de stocker des références à d'autres objets, car tous les attributs se trouvaient sur une seule page. Deuxièmement, si un objet prenait plus de place que ce qu'une page pouvait contenir, il fallait le diviser, ce qui compliquait considérablement l'implémentation.

Approche choisie : organisation des données par pages Nous avons proposé une autre approche. Comme dans le premier cas, les informations sur les classes sont placées sur une page séparée, mais les objets ne stockent désormais pas directement les données. Au lieu de cela, ils contiennent des références vers des pages spécialisées destinées à stocker différents types de données : chaînes de caractères, nombres et valeurs booléennes.

Cette solution a permis de résoudre plusieurs problèmes. Tout d'abord, elle a éliminé la nécessité de diviser les objets entre les pages : même si un objet contient des centaines d'attributs, sa structure reste intacte. Deuxièmement, la recherche par attributs est devenue plus facile, car il est possible de travailler uniquement avec les pages dont nous avons besoin. Le seul inconvénient de cette approche est la complexification du mécanisme de distribution des données et de leur reconstitution. De plus, si la base de données contient de nombreuses tables avec un nombre limité d'enregistrements, une partie de l'espace pourrait être gaspillée de manière inefficace.

Définition de la structure des pages

Méta-informations de la page

Chaque page commence par un en-tête contenant des informations clés. En premier lieu, il s'agit de son type (short, 2 octets), ainsi qu'un lien vers la page suivante (si elle existe, sinon la valeur -1 est enregistrée) et la taille des données sur la page (si elle est dynamique, la valeur est -1).

Adressage des données

Pour organiser les relations entre les pages, un mécanisme de liens a été introduit. Chaque valeur dans le fichier peut être clairement définie par une paire de nombres : l'index de la page (int, 4 octets) et le décalage à l'intérieur de celle-ci (short, 2 octets). Initialement, nous avions envisagé l'utilisation d'un nombre de 4 octets, mais cela limitait le nombre de pages. En conséquence, une représentation combinée de 6 octets a été choisie.

Gestion de l'espace libre

L'une des tâches clés consiste à organiser la suppression des données. Il est nécessaire de suivre quelles zones de mémoire sont libres afin de pouvoir les réutiliser sans redimensionner toute la page.

Taille fixe des éléments

Si les objets sur la page ont une taille uniforme, l'espace libre peut être suivi à l'aide d'une carte de bits. Cependant, pour de grandes pages, cette approche devient inefficace : par exemple, si la page contient des milliers de valeurs booléennes, la carte occupera trop de place.

Une approche plus pratique consiste en une liste chaînée des cellules libres. Dans ce cas, chaque espace vide contient un lien vers la cellule libre suivante ainsi que sa taille. Si aucun espace libre n'est disponible, le lien est défini sur -1.

Taille dynamique des éléments

Si les objets occupent une quantité de mémoire différente (par exemple, les métadonnées sur les classes), la carte de bits n'est pas adaptée. Au lieu de cela, la page entière est considérée comme libre par défaut, et lors de l'ajout de données, elle est divisée en fragments. Au début de la page, il y a toujours un pointeur vers le premier espace libre.

Types de pages de la base de données

Il a été décidé d'introduire sept types de pages dans le projet :

1. `MetaPage` — page pour la description des classes.
2. `ObjectPage` — page pour la description des objets.
3. `StringPage` — page pour le stockage des chaînes de caractères.
4. `LongPage` — page pour le stockage des nombres.
5. `BooleanPage` — page pour le stockage des valeurs booléennes.
6. `HeaderPage` — page d'en-tête de la base de données.
7. `FreePage` — page vide.

Toutes les pages héritent de la classe abstraite `Page`. Examinons-la plus en détail.

Classe abstraite `Page`

La base de toutes les pages est une instance de `ByteBuffer`, qui stocke toutes les informations relatives à la page. Lors de l'écriture sur disque, seul ce tampon est écrit.

La classe `Page` fournit une interface pour travailler avec les attributs de la page, stockés dans le `ByteBuffer` (les 10 premiers octets). De plus, la classe propose une abstraction pour représenter les positions dans le tampon pour les pages filles, ce qui permet d'éviter de prendre en compte la méta-information de 10 octets.

La classe `Page` possède les méthodes suivantes :

- `reformatPage(short type, short dataSize)` — méthode pour formater la page ;
- `getNextFreeOffset(int size)` — méthode pour allouer de l'espace libre sur la page ;
- `releaseOffset()` — méthode pour libérer l'espace précédemment alloué.

`getNextOffset` fonctionne comme suit :

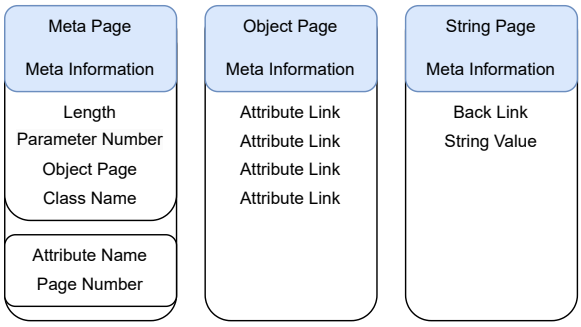

Figure 2: Page Type

- Lit et vérifie le lien vers le premier espace libre ;
- Tant qu'il n'atteint pas le lien -1 ;
- Place le curseur sur l'espace libre ;
- Lit le prochain espace libre et la taille de l'espace actuel ;
- Si la taille correspond, alors
- Déplace le lien vers le premier espace libre vers le lien lu pointant vers le prochain espace libre.

Classe `MetaPage`

Ce type de page est destiné au stockage des informations sur les classes. Cette page a une taille de données dynamique, car les classes possèdent un nombre variable d'attributs. Pour chaque objet, les informations suivantes sont enregistrées :

- La longueur sur la page actuelle (2 octets)
- Le nombre d'attributs, ce qui permet de connaître la longueur de l'objet sur une `ObjectPage` (2 octets)
- Un pointeur vers la `ObjectPage` où sont stockés les objets de cette classe (4 octets)
- Le nom de la classe (64 octets)
- Ainsi qu'une liste de noms d'attributs (64 octets chacun) et des pointeurs vers les premières pages où ces attributs sont stockés (4 octets chacun)

Cette classe implémente l'interface `MetaDataPage` et définit les méthodes suivantes :

- `add(TableDescription dataClass)` — méthode permettant d'ajouter une classe à la page
- `getClassByName(String name)` — retourne un `TableDescription` correspondant au nom du fichier, ou `null` s'il n'existe pas
- `deleteClassByName(String name)` — supprime la description d'une classe de la page, si elle est présente

Fonction `getClassByName(String name)`

Objectif de la fonction : Trouver la description d'une table (classe) par son nom dans une page de métadonnées et retourner un objet `TableDescription`.

Algorithme de fonctionnement :

- La fonction appelle `searchTableByName(className)`, qui effectue les étapes suivantes :
  1. Convertit le nom de la table en un tableau d'octets pour la comparaison.

2. Récupère la carte des zones libres sur la page (`pageMap`) à l'aide de `getPageMap()`.
3. La carte `pageMap` contient des paires `(beginFreeSpace, endFreeSpace)`.
4. Utilise un `cursor` pour parcourir la page :
   - Si la position actuelle du curseur ne fait pas partie d'un bloc libre (c'est-à-dire absente de `pageMap`), cela signifie qu'un objet est présent à cet emplacement.
   - Lit la longueur de la description de la table, puis compare son nom avec le nom recherché.
   - Si le nom correspond, retourne l'`offset` de la table.
   - Sinon, déplace le curseur de la longueur de l'objet actuel et continue la recherche.
5. Si aucun nom ne correspond, retourne `-1`.

- Si un décalage est trouvé `(offset != -1)`, la fonction `getClassByOffset(offset)` est appelée. Elle effectue les opérations suivantes :

noitemsep Lit les données de la table à l'emplacement donné :

- Le nombre de paramètres : `paramNumber`
- Le numéro de la page : `page`
- Le nom de la table : `className`
- Les tableaux de noms d'attributs : `attributeNames` et les numéros de pages correspondants : `attributesPages`

noiitemsep Crée et retourne un objet `TableDescription` avec les données obtenues.

Particularités :

- `Taille dynamique des éléments` : Étant donné que la taille des éléments sur la page n'est pas fixe, une carte des blocs libres (`pageMap`) est utilisée pour déterminer les zones occupées.
- `Recherche efficace` : L'algorithme ignore les blocs libres et ne vérifie que les zones occupées, ce qui accélère considérablement la recherche.

Classe `ObjectPage`

Cette page stocke les adresses des valeurs des objets. Chaque adresse est composée d'un numéro de page (`int`) et d'un décalage (`short`). La classe implémente l'interface `DataPage`, utilisée pour uniformiser les interactions avec les pages au niveau du gestionnaire de tables (`TableManager`).

Elle définit les méthodes suivantes :

- `allocate()` — méthode permettant d'allouer un nouvel emplacement libre
- `insertToIndex(Address[] objectLinks, int index)` — méthode permettant d'ajouter des liens (adresses) à une position donnée
- `delete(short index)` — supprime la liste de liens à l'index spécifié

Le placement d'un objet sur la page se fait en deux étapes : D'abord l'allocation de l'espace, puis l'insertion du tableau d'adresses.

Classes `StringPage`, `LongPage`, `BooleanPage`

Ces pages sont destinées au stockage direct des données des objets. Chaque élément est stocké avec un lien inverse (backlink) vers l'objet correspondant sur la `ObjectPage`, ce qui permet, lors d'une recherche, d'accéder directement à l'objet et de le reconstruire.

Toutes ces classes implémentent l'interface `BackLinkPage`, qui permet d'unifier toutes les interactions avec ces pages dans le gestionnaire de tables (`TableManager`).

- `add(Type value, Address address)` — méthode pour insérer de nouvelles données sur la page

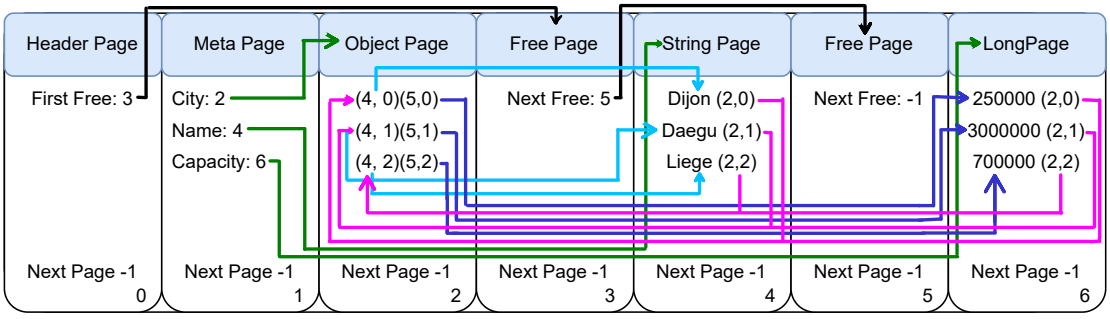

Figure 3: Organisation générale du fichier

- `replace(short index, Type newValue, Address newAddress)` — méthode pour modifier les données existantes
- `search(Condition condition)` — méthode de recherche basée sur une condition donnée

Les pages destinées au stockage des types sont massivement scalables : pour ajouter un nouveau type, il suffit de déclarer une page, de s'assurer qu'elle implémente l'interface `BackLinkPage<Type>`, puis de définir la taille des métadonnées et des objets eux-mêmes. Ensuite, il faut adapter les méthodes de l'interface au nouveau type de données.

Organisation générale du fichier

Dans le fichier, tout comme sur les pages, des espaces vides peuvent se former en cas de suppression de classes ou d'attributs. Il est nécessaire de prendre cela en compte pour allouer efficacement de nouvelles pages. Mais comment savoir quelles pages sont vides ? Pour se faire, nous avons décidé d'introduire une `FreePage`, chaque page vide stockant un lien vers la page suivante ou `-1` si aucune page vide n'est disponible.

Mais où stocker le lien vers la première page vide ? Pour cela, nous avons créé une `HeaderPage` — une page d'en-tête de la base de données, qui contient la version de la base et une chaîne magique pour vérifier l'authenticité de la base de données.

Pour organiser les pages, nous avons appliqué le patron de la conception.

Chapitre 2 dans lequel nous écrivons le gestionnaire de pages

Après le chapitre précédent, dans lequel la structure du fichier a été définie, l'étape suivante a été de comprendre comment implémenter le stockage et la récupération des pages depuis le disque. Pour commencer, examinons l'architecture en couches des composants du programme. Au niveau le plus bas se trouvent les fonctions responsables du travail direct avec le disque. Cependant, le chargement constant des pages depuis le disque et leur réécriture est une opération coûteuse en temps et en ressources. Il est donc logique d'introduire un cache dans lequel seront stockées les pages récemment utilisées. Le cache est implémenté sous la forme d'un 'HashMap' qui associe les numéros de pages (décalages depuis le début du document) aux objets des pages. Cependant, le cache a une taille limitée, c'est pourquoi une stratégie d'éviction a été mise en place. La stratégie choisie est **LFU (Least Frequently Used)** — la moins fréquemment utilisée, car sa mise en œuvre s'est avérée la plus simple dans le contexte de notre tâche. Pour suivre les pages modifiées, un mécanisme de "pages sales" a été introduit. À cet effet, un 'HashMap' supplémentaire est utilisé, qui associe le numéro de la page à un indicateur booléen indiquant si la page a été modifiée et nécessite une écriture sur le disque. De plus, la tâche de suppression des pages vides et de création de nouvelles pages à la fin du document a été résolue.

Histoire du développement

Première approche

La première implémentation consistait à charger une page depuis le disque à chaque demande et à l'écrire de nouveau après traitement. Cette méthode s'est avérée extrêmement inefficace et servait plutôt de prototype. (voir 'Pager.java' dans le dossier 'PreviousIdea').

Deuxième approche

L'étape suivante a été la création de 'VirtualMemoryManager'.

Tout d'abord, l'accès direct aux fichiers a été abstrait via les interfaces 'PageLoader' et 'PageSaver', ce qui a permis d'implémenter, par exemple, le chargement des pages via le réseau.

Ensuite, un cache a été ajouté, ce qui a considérablement amélioré les performances. Une stratégie LFU a également été mise en place, ainsi qu'un mécanisme de vieillissement, empêchant la conservation dans le cache de pages anciennes mais encore populaires, mais désormais inutiles.

À ce stade, la classe 'PageManager' a commencé à fonctionner directement avec 'ByteBuffer', ce qui a permis de maintenir l'abstraction.

Troisième approche

Dans la troisième version, 'VirtualMemoryManager' a commencé à résoudre le problème d'accès concurrent. Étant donné que des blocages peuvent survenir en raison de ressources multiples, nous avons implémenté un mécanisme de versioning des pages.

Chaque page a reçu une version, qui augmente à chaque modification. Le thread travaillant avec une page la demande, effectue des modifications, puis la renvoie. Si la version de la page a changé pendant le traitement, les modifications sont annulées et la tentative est répétée. Si la version correspond, les modifications sont enregistrées et la version est incrémentée.

Cette approche a permis d'éviter les blocages, mais a conduit à la nécessité de réappliquer les modifications en cas de conflit. Un autre problème a été la création d'une classe redondante 'DataWrapper', qui stocke à la fois le 'ByteBuffer' et la 'PageVersion', ce qui duplique en fait la classe 'Page'. Finalement, cette approche a été jugée inefficace.

Quatrième approche

Cette version est devenue la base de la solution finale. Le problème de la concurrence a été déplacé dans un module séparé. La classe 'Resource' a été introduite, contenant le 'ByteBuffer', des informations sur le thread propriétaire de la ressource et un verrou assurant la capture de la page lors de la synchronisation.

Cependant, 'Resource' répétait la fonctionnalité de 'Page', et ne permettait pas d'obtenir une page vide en milieu de document.

Approche finale

L'approche finale est la suivante : la page elle-même est une ressource. Chaque page a un propriétaire. Les pages sont chargées depuis le disque via 'PageLoader' et créées à l'aide de la fabrique 'PageFactory' au niveau de 'MemoryManager'.

Le cache contient des objets 'Page' complets, ce qui élimine la nécessité de les recréer. Toutes les questions de synchronisation et de concurrence sont résolues au niveau de 'PageManager', y compris la prévention des blocages.

Une possibilité de suivre le nombre de pages dans la base a été ajoutée, ainsi que celle de réserver des pages libres depuis le milieu du fichier — à l'aide de 'FreePage' et 'HeaderPage'.

Algorithme de demande de page

L'algorithme de récupération des pages ressemble beaucoup au processus de pagination (SWAP) :

`public List<Page> indexesToPages(List<Integer> requestedPageIndexes)`

1. Si une page existante est demandée :
   (a) Capturez le verrou sur headerPage, vérifiez si l'indice dépasse les limites de la base, puis relâchez le verrou.
   (b) Vérifiez le cache. Si la page est trouvée, retournez-la.
   (c) Si elle n'est pas trouvée :
      i. Capturez le verrou sur pageCache.
      ii. Vérifiez à nouveau la présence de la page dans le cache (Double Check pattern).
      iii. Si elle est déjà ajoutée, retournez-la.
      iv. Sinon, chargez la page depuis le disque via PageLoader.
      v. Si la page est vide, lancez une exception.
      vi. Ajoutez la page dans le cache.
      vii. Retournez la page.
2. Si une nouvelle page doit être créée :
   (a) Capturez les verrous sur headerPage et pageCache.
   (b) Cherchez la première page vide via headerPage.
   (c) Si trouvée :
      i. Chargez la page depuis le disque.
      ii. Mettez à jour l'indice de la prochaine page vide, marquez headerPage comme sale.
      iii. Ajoutez la nouvelle page dans le cache.
      iv. Retournez-la.
   (d) Si non trouvée :
      i. Obtenez le nombre total de pages dans la base.
      ii. Mettez à jour cette valeur.
      iii. Créez une nouvelle page.
      iv. Ajoutez-la dans le cache.
      v. Si nécessaire, agrandissez le fichier sur le disque.
      vi. Retournez la nouvelle page.
3. Relâchez les verrous cacheLock et headerLock.

Mécanismes de synchronisation au niveau de MemoryManager

Bien que la majeure partie de la synchronisation soit réalisée dans `PageManager`, il existe toujours une possibilité d'appel simultané de plusieurs méthodes de `MemoryManager` par différents threads. C'est pourquoi des mécanismes de synchronisation supplémentaires ont été introduits.

Définition des ressources

Dans la classe `MemoryManager`, les structures clés suivantes ont été définies :

- `pageCache` — structure de mise en cache des pages, implémentée sous forme de `ConcurrentHashMap<Integer, Page>`. Il s'agit d'une ressource partagée, l'accès à laquelle doit être synchronisé.

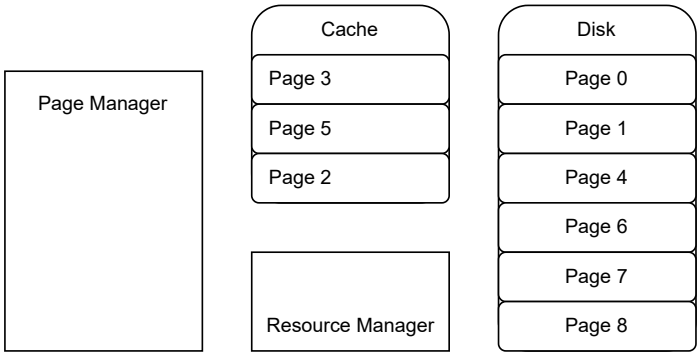

Figure 4: State of data structures during program operation

- `dirtyPages` — ensemble des pages modifiées, implémenté sous forme de `ConcurrentSkipListSet<Integer>`. Cela est également considéré comme une ressource, car il contient des données critiques devant être écrites sur disque.
- `accessCounts` — structure pour stocker les compteurs d'accès aux pages, `ConcurrentHashMap<Integer, Integer>`. Ce n'est pas considéré comme une ressource, car il n'influence pas l'intégrité des données et est utilisé uniquement pour implémenter la politique d'éviction.

De plus, `headerPage` est une ressource distincte, car elle contient des métadonnées sur la structure de la base de données. Pour garantir la cohérence des données même en cas de panne, toutes les modifications dans `headerPage` doivent être correctement synchronisées et écrites sur disque en temps utile.

Solution au problème de la concurrence

Pour synchroniser l'accès à `pageCache` et `dirtyPages`, un verrou commun `Lock` a été introduit pour empêcher les écritures parallèles.

L'accès à `headerPage` est réalisé via un `ReadWriteLock`, ce qui permet une lecture simultanée par plusieurs threads, avec un verrou exclusif pour l'écriture.

Il convient de noter que plusieurs ressources sont définies dans le système, ce qui pourrait potentiellement entraîner des blocages (deadlock). Cependant :

- la plupart des fonctions n'interagissent qu'avec une seule ressource ;
- dans les autres cas, l'ordre de prise des verrous est strictement régulé.

Ainsi, théoriquement, la possibilité de deadlocks est exclue.

Conclusion

La classe `MemoryManager` encapsule le mécanisme de chargement des données depuis le disque et le cache, ainsi que la gestion du cache et des requêtes de pages. Elle permet de remplacer les pages, de travailler avec des zones de mémoire vides, de mettre en œuvre l'éviction des pages du cache et de synchroniser les modifications avec le disque. Grâce aux mécanismes intégrés de protection contre la concurrence, `MemoryManager` assure un fonctionnement correct en mode multithread avec les données, même dans des conditions de forte charge.

Chapitre 3, dans lequel arrive le multithreading.

Il est temps d'examiner la gestion de la concurrence entre les threads pour l'accès aux ressources. Dans le contexte de ce système, les ressources sont définies comme les pages, représentées par des objets de la classe `Page`, ou des tampons au format `ByteBuffer`.

Il peut y avoir des situations où deux ou plusieurs threads ont besoin des mêmes pages, ce qui, compte tenu de l'architecture des pages choisie, n'est pas rare. Cela entraîne la nécessité de mettre en œuvre des mécanismes pour éviter les blocages mutuels (deadlocks).

Examinons les principales approches pour résoudre ce problème :

- La première — la méthode de versioning des pages déjà abordée dans le deuxième chapitre, avec résolution des conflits par l'annulation des modifications.
- La deuxième — l'ordre de capture des ressources : puisque chaque page a un numéro unique, les threads capturent les pages dans un ordre strictement croissant. Cette approche nécessite de connaître à l'avance toutes les pages nécessaires.
- La troisième — l'utilisation de minuteries : un thread, n'ayant pas obtenu toutes les ressources nécessaires, libère celles déjà capturées après un certain délai d'attente et tente à nouveau.
- La quatrième — la construction d'un graphe de dépendances entre les threads et l'analyse des cycles d'attente.

Approche appliquée

Dans notre implémentation, toutes les méthodes décrites ci-dessus ont été combinées. Les threads capturent les pages dans l'ordre croissant de leurs indices. Si cela échoue, un minuteur est lancé et le thread tente à nouveau. Pour garantir l'absence de blocages, un graphe de dépendances est construit pour suivre les cycles. De plus, une limite sur le nombre de tentatives est imposée, après quoi le thread cesse de tenter de capturer la ressource.

La capture de la ressource se fait en deux étapes :

1. Capture du verrou (lock).
2. Obtention de la ressource en possession.

Analyse des algorithmes d'acquisition des ressources

L'idée principale est de construire un graphe dans lequel sont enregistrées les dépendances des threads vis-à-vis des ressources détenues par d'autres threads.

Les structures suivantes sont utilisées :

- `threadToResourcesHeld` — une carte associant chaque thread aux ressources qu'il détient.
- `threadToResourcesWaiting` — une carte associant chaque thread aux ressources qu'il attend.

Pour la synchronisation, les éléments suivants sont utilisés :

- `graphLock` — un verrou pour protéger la structure du graphe de dépendances.
- `resourceReleased` — une condition (condition variable) qui avertit les threads lorsque des ressources sont libérées.

	Held	Waiting
Thread 1	A,C	B
Thread 2	B	D,E
Thread 3	E	A

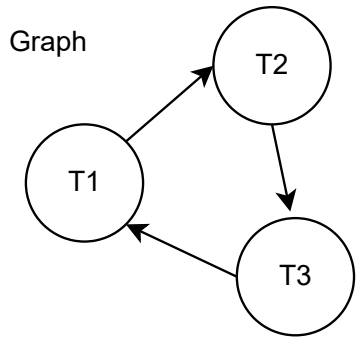

Figure 5: Deadlock example

Algorithme de prévention des deadlocks

1. **Construction du graphe d'attente (Wait-for Graph)**
   La méthode `buildWaitForGraph()` crée un graphe orienté, où :
   - Les sommets sont des threads.
   - Une arête $A \rightarrow B$ est ajoutée si le thread A attend une ressource détenue par le thread B.

   Le graphe est représenté sous la forme `Map<Thread, Set<Thread»`.
2. **Détection des cycles**
   La méthode `hasCycle()` utilise une recherche en profondeur (DFS) pour détecter les cycles dans le graphe. La présence d'un cycle indique un deadlock potentiel.
3. **Vérification préventive**
   La méthode `wouldFormCycle()` permet de déterminer si l'attribution d'une ressource entraînera la formation d'un cycle. Si c'est le cas, la demande est rejetée.

Méthodes supplémentaires de prévention des deadlocks

Ordonnancement des ressources

```
requestedResources.sort(Comparator.comparingInt(Page::getPageNumber));
```

Les ressources demandées sont triées par ordre croissant de numéro de page, ce qui empêche l'attente cyclique et améliore la prévisibilité du comportement du système.

Mécanisme de timeout et de tentatives répétées

```
this.resourceReleased.await(retryDelayMs, TimeUnit.MILLISECONDS);
```

Le thread n'est pas bloqué indéfiniment — il attend pendant un certain temps, après quoi il tente à nouveau. Cela permet au système de résoudre les conflits temporaires sans blocage.

Concept de zone d'influence

La zone d'influence est l'ensemble des ressources détenues par un thread. L'extension de la zone est réalisée par la méthode suivante :

```
public List<Page> expandResourceZone(List<Integer> additionalPagesIndexes,
                                     int maxRetries,
                                     long retryDelayMs)
```

Cette méthode permet à un thread, déjà propriétaire de certaines ressources, d'en acquérir d'autres tout en conservant celles qu'il possède déjà. Cela est crucial, car de nouvelles pages peuvent être allouées dynamiquement pendant l'exécution.

Avantages de l'approche :

- Maintien de l'atomicité des opérations sur plusieurs pages.
- Extension flexible de la zone de possession.
- Utilisation de tous les mécanismes de prévention des deadlocks lors de l'ajout de nouvelles ressources.

Mécanisme de libération des ressources

```
public void releasePages(List<Integer> resourcesToReleaseIndexes)
```

Lors de la libération des ressources :

1. Le thread perd la possession des pages correspondantes.
2. Les structures auxiliaires sont mises à jour.
3. Tous les threads en attente sont notifiés (`resourceReleased.signalAll()`).
4. Si la page a été modifiée, elle est marquée comme "dirty".

Opérations supplémentaires sur les ressources

Suppression de pages

```
public void deletePages(List<Integer> pageToDeleteIndexes)
```

Seules les pages possédées par le thread actuel sont supprimées.

Échange de pages

```
public void exchangePage(Page oldPage, Page newPage)
```

Permet de remplacer une page par une autre dans la zone d'influence du thread. Cela est nécessaire lorsqu'un thread reçoit une `FreePage` lors de l'allocation d'une nouvelle page depuis la base de données, et cette `FreePage` doit être remplacée par un type de page opérationnelle, tant dans `MemoryManager` que dans `PageManager`.

Chapitre 4, dans lequel nous essayons de faire fonctionner tout cela, mais il manque quelque chose.

Dans les trois chapitres précédents, nous avons étudié l'implémentation bas niveau de la base de données. Maintenant, notre objectif est de relier ce qui a été écrit à une bibliothèque utilisateur de haut niveau. Pour ce faire, nous devons définir quelles actions nous voulons exécuter sur la base de données et comment les accomplir en utilisant les mécanismes que nous avons créés au niveau des pages.

La base de données doit être capable d'exécuter quatre actions principales (CRUD) :

- **CREATE** — création d'une nouvelle classe et ajout de nouveaux objets à celle-ci.
- **READ** — recherche d'objets par un ou plusieurs attributs.
- **UPDATE** — modification de l'attribut d'un objet spécifique ou ajout d'un nouvel attribut à une classe.

- **DELETE** — suppression d'un objet ou d'une classe entière.

Ainsi, pour effectuer ces actions, nous devons définir quelles données sont nécessaires, d'où elles proviennent (de l'utilisateur), et quelles sont les restrictions imposées par l'architecture du système.

La structure `TableDescription` est utilisée pour décrire les classes et refléter leur état actuel dans la base de données. Elle contient le nom de la classe, les noms des attributs, les pages où les données relatives aux attributs commencent, ainsi que `ObjectPage` pour la reconstruction des objets. Les objets `TableDescription` sont sauvegardés dans `MetaPage` et extraits de là lors de l'accès à une classe par son nom.

CREATE

Création de classe

```
public void createTable(TableDescription newTable)
```

Pour créer une classe, il est nécessaire de transmettre son nom, une liste d'attributs et leurs types. Le nom est utilisé pour rechercher dans `MetaPage`, les attributs sont utilisés pour effectuer des opérations, et les types sont importants pour l'allocation et la mise en forme des pages.

Le processus de création commence par l'obtention de `MetaPage`. D'abord, une page est allouée pour `ObjectPage`, puis des pages sont allouées pour chaque attribut. Elles sont formatées en fonction des types de données. Cette approche en deux étapes (allocation, puis formatage) permet d'annuler les modifications en cas d'erreur. L'objet `TableDescription` est mis à jour au fur et à mesure de l'exécution et est finalement sauvegardé dans `MetaPage`.

Une fonction auxiliaire est également implémentée :

```
public void createTableIfNotExist(TableDescription newTable)
```

Elle permet d'éviter l'exception si la table existe déjà.

Création d'un objet de classe

```
public void addObject(TableDescription newTable, Map<String, Object> attributeValues)
```

Cette fonction ajoute un nouvel objet à la classe. Elle prend en entrée la description de la classe (`newTable`), contenant les noms et types des attributs. Cela est nécessaire, car les types ne sont pas stockés directement dans la base de données — ils sont extraits uniquement de la structure `TableDescription`.

Important : l'objet `TableDescription` transmis ici est différent de celui extrait de `MetaPage`. Le premier contient uniquement les noms et types des attributs, tandis que le second contient également les pages de stockage.

Algorithme : 1. Extraire la description complète de la classe depuis `MetaPage`. 2. Vérifier la correspondance des noms des attributs. 3. Obtenir les pages pour l'insertion. 4. Vérifier la longueur des pages et la possibilité d'insertion. 5. Effectuer l'insertion. (Le rollback en cas d'erreur n'est pas encore implémenté.)

Recherche et insertion : détails sur les mécanismes

Chaque page insère des données uniquement dans les limites de son propre `ByteBuffer`. Si l'espace est épuisé, elle retourne `-1`. La page vérifie la présence de la page "voisine" et, si elle n'existe pas, demande une nouvelle page à la base. La nouvelle page est liée à la page actuelle, et l'insertion continue.

Le processus de recherche est similaire, mais sans ajout de pages.

Cette architecture rend chaque page autonome et isolée.

READ

```
public ArrayList<Map<String, Object>> searchObject(TableDescription searchVictim,
ArrayList<Condition> conditions)
```

Le mécanisme de recherche est l'une des raisons pour lesquelles cette architecture a été choisie.

1. Extraire la description de la classe avec les pages associées.
2. Filtrer les conditions applicables.
3. S'il n'y a pas de conditions — restaurer tous les objets.
4. S'il y a des conditions — obtenir la page de l'attribut nécessaire depuis `MetaPage` et filtrer les objets.
5. Appliquer les conditions à chaque objet. Ceux qui ne passent pas le filtre sont supprimés des résultats.

Classe Condition

`Condition`

Contient : le nom de l'attribut, l'opérateur, la valeur. La vérification de la validité des conditions est effectuée du côté client.

Pour les chaînes, l'opérateur "contains" est disponible, et dans le futur, il est prévu d'ajouter les opérateurs "startsWith", "endsWith" et un opérateur de "proximité" (fuzzy match). Le système d'opérateurs est écrit de manière extensible, tout comme la typisation des pages.

Restauration des objets

```
private Map<String, Object> restoreObject(TableDescription classThatWeTryToRestore,
Address objectPageAddress)
```

À partir de l'attribut, l'adresse inverse de la `ObjectPage` est extraite, puis les adresses de tous les attributs. Les pages correspondantes sont demandées, les valeurs sont associées aux noms des attributs et retournées sous forme de `Map<String, Object>`, prête à être utilisée pour restaurer l'objet côté client.

DELETE

Suppression d'un objet

```
public void deleteObject(TableDescription deleteVictim, ArrayList<Condition> conditions)
```

La suppression suit le même mécanisme que la recherche. La différence est que lors de la restauration des objets, leurs adresses sur `ObjectPage` sont également sauvegardées.

Ensuite, la suppression est effectuée : pour chaque adresse d'attribut, la page correspondante est demandée et la valeur est supprimée. L'adresse est remplacée par `new Address(-1, -1)`, puis l'enregistrement de `ObjectPage` est également supprimé.

Suppression d'une classe

```
public void deleteTable(TableDescription deleteVictim)
```

La fonction effectue la suppression d'une classe en trois étapes :

1. Récupération de toutes les pages (y compris les pages voisines) à partir des données de `MetaPage`.
2. Suppression des pages.
3. Suppression de l'enregistrement dans `MetaPage`.

UPDATE

La fonction de mise à jour n'est pas encore implémentée. Cependant, sa mise en œuvre est simple : en réalité, c'est une combinaison de `READ` et `CREATE`.

Les étapes nécessaires sont les suivantes :

1. Trouver l'objet et enregistrer les adresses des attributs (comme dans `DELETE`).
2. Implémenter la méthode de mise à jour des attributs au niveau de la page.

Organisation de la classe TableManager

Gestion des ressources

Le niveau inférieur gère les ressources. Lorsqu'une ressource est demandée à nouveau par le même thread, elle est fournie sans condition. Pour optimiser cela, un cache est introduit dans `TableManager`, qui stocke les ressources déjà allouées.

Les méthodes `releasePage(Page page)` et `releaseAllPages()` nettoient le cache pour éviter les conflits.

Structure générale

Cette section n'est pas encore implémentée. À un moment donné, le `TableManager` est devenu trop complexe et une décomposition était nécessaire. Deux approches ont été envisagées :

- Une structure de classes à plusieurs niveaux.
- L'utilisation du patron de conception "Facade" pour masquer la complexité interne.

La première approche a été choisie. La structure des niveaux est la suivante :

1. **ResourceManager** — gestion du cache et allocation des ressources.
2. **PageManager** — gestion des pages et organisation des données.
3. **ObjectManager** — gestion des objets logiques.
4. **TableManager** — opérations de haut niveau : création, recherche, suppression.

Malheureusement, la mise en œuvre complète n'a pas été achevée.

Problème de manque de données

Le projet a évolué de bas en haut : d'abord les pages, puis leur interaction avec le fichier, la synchronisation des threads, et enfin les fonctions de haut niveau.

Définir l'interface d'interaction avec l'utilisateur s'est avéré une tâche complexe : il fallait la rendre à la fois intuitive et suffisamment expressive.

La solution a été la création d'une classe "adaptateur" `QueryToAction`. Elle sera détaillée dans le chapitre suivant.

Chapitre 5, dans lequel on pense a l'interaction utilisateur.

Approche naive

L'objectif est de permettre à l'utilisateur de pouvoir interagir avec une base de données contenant des objets. Pour ce faire on a implémenté un mini SGBD permettant l'ensemble des opérations CRUD (Create Read Update Delete), c'est-à-dire que l'utilisateur doit pouvoir créer des classes et des objets, lire des objets dans la base et mettre à jour ou supprimer des objets ou des classes dans la base de données. Ce SGBD doit également implémenter des transactions répondant aux propriétés ACID (Atomicity Consistency Isolation Durability).

Cette approche n'a pas été développée par la suite mais elle fait partie du travail de réflexion qui a mené à l'état actuel du projet. Le code de l'interpréteur, ainsi que du client et du serveur ont été écrit et sont disponibles en annexe 1.

> **Problem 1: L'entrée utilisateur**
>
> Pour permettre à l'utilisateur de communiquer avec la base de données, nous avons créé un langage.
> La syntaxe du langage est très simple. Les séparateurs pour les mots-clé et les valeurs sont simplement des espaces blancs. Un script peut décrire plusieurs transactions. Chaque transaction est constituées d'un ensemble de requêtes à exécuter de manière séquentiel et en obéissant aux propriétés ACID. Les mots clé décrivant respectivement le début et la fin d'une transaction sont "start" et "end". Une requête est décrite de la manière suivante:
>
> ```
> [action] from [nom de la classe] where [nom d'attribut 1]
>     [opérateur de comparaison 1] [valeur d'attribut 1]
>     [opérateur logique 1] [nom d'attribut 2]
>     [opérateur de comparaison 2] [valeur d'attribut 2]...
> ```
>
> Les opérateurs de comparaison sont les mêmes qu'en Java. Les opérateurs logiques sont "and" pour le ET logique et "or" pour le OU logique.

Voici un exemple de script contenant deux transactions:

```
start
create from Classe1 where Attribut1 == val1 and Attribut2 != val2
delete from Classe2 where Attribut2 > val3 or Attribut3 < val4
end
start
update from Classe3 where Attribut4 >= val5 or Attribut5 <= val6
read from Classe4 where Attribut6 == val7
end
```

Une nouvelle tentative dans cette direction a ete faite par la suite mais elle n'a pas abouti, faute de temps. L;'automate est disponible en annexe.

> **Problem 2: Représenter les transactions et les requêtes**
>
> Toutes les requêtes sont stockées dans un tableau dynamique (buffer) de requêtes qui contient aussi des requêtes spéciales marquant les débuts et fin des transactions. Chaque requête (objets de la classe Query.Query) est constituée d'une action (create, read, update ou delete), du nom de la classe sur laquelle porte la requête, et d'un tableau dynamique de conditions (objets de la classe Query.Condition) qui représente l'ensemble des conditions indiquées après le mot-clé "where" à la fin de la description d'une requête. Un interpréteur (classe Interpreter) est alors utilisé par le serveur pour transformer le langage écrit par l'utilisateur en un tableau de requêtes qu'il ajoute à un tableau existant en utilisant le

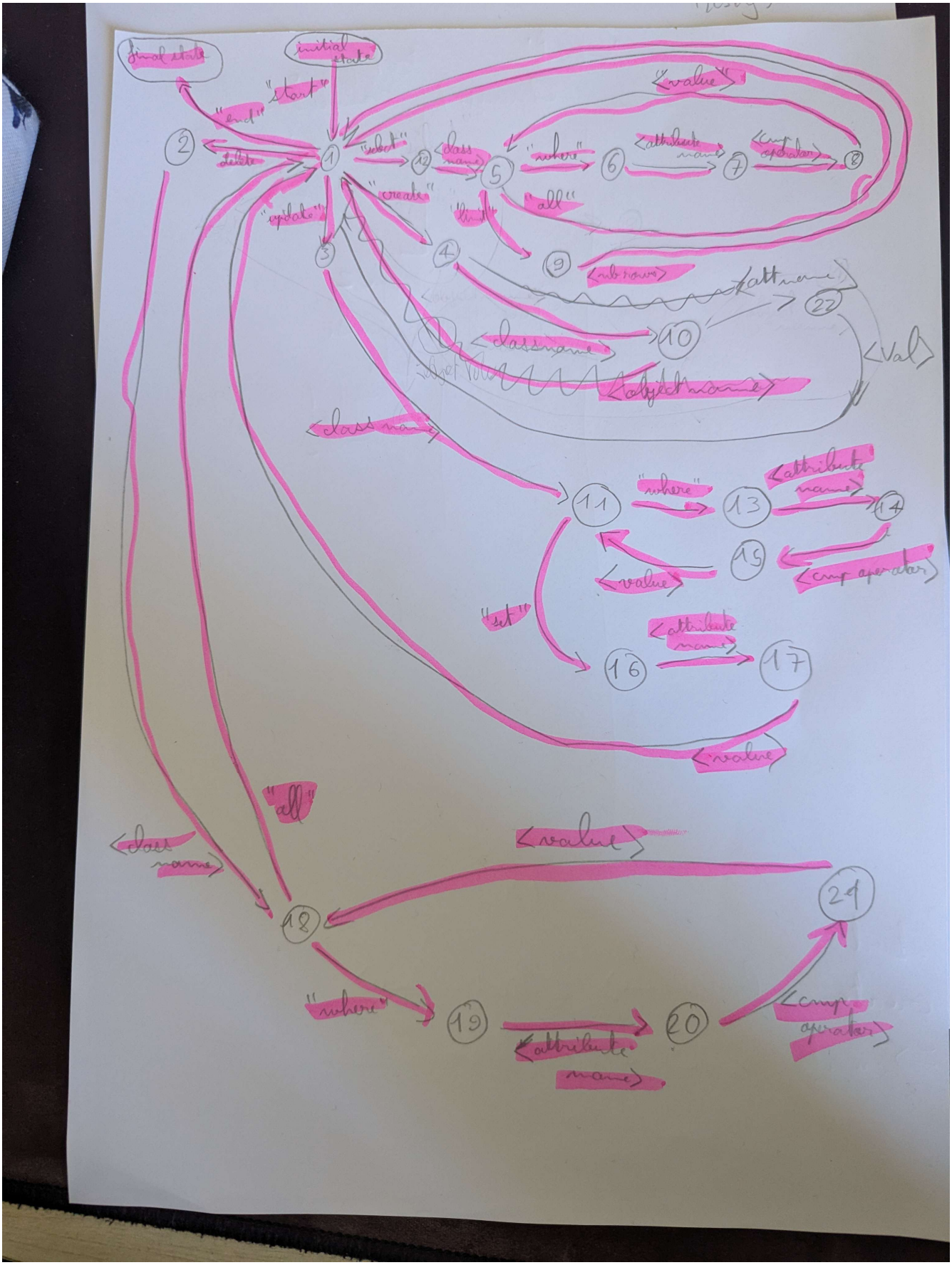

Figure 6: Nouvel automate

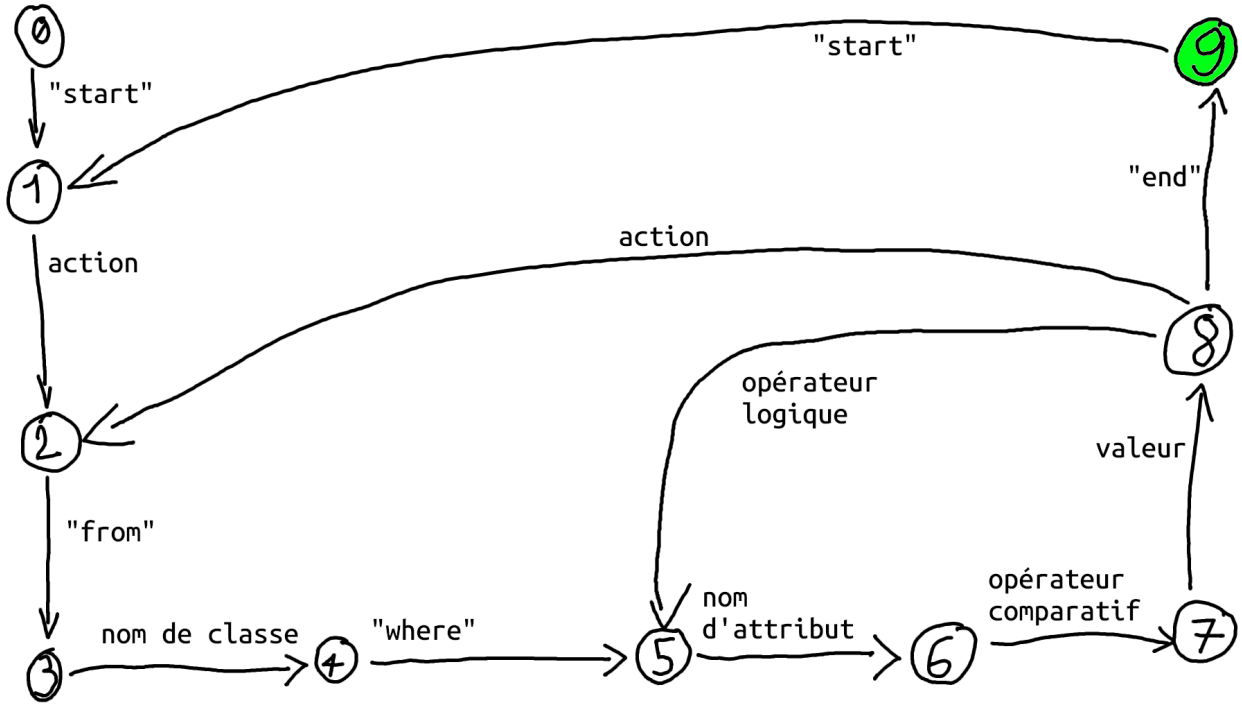

Figure 7: Machine à état de l'interpréteur

> modèle producteur-consommateur. L'interpréteur est basé sur une machine à état décrite par le schéma ci-dessus. Remarquez que l'état 9 est représenté en vert, c'est pour signifier que cet état accepte la fin du fichier.

Nouvelle approche avec un nouveau langage

Lors de l'interaction avec l'utilisateur, nous avons décidé de créer une syntaxe aussi simple que possible. Lors de sa création, nous nous sommes inspirés de la bibliothèque SQLModel pour Python https://sqlmodel.tiangolo.com/ et de la syntaxe SQL. L'étape la plus difficile a été la création de la structure des classes en Java qui permettait de réaliser la syntaxe que nous avons inventée.

Création de la syntaxe

Commençons par l'essentiel — la recherche. Pour trouver un objet dans une table, il faut d'abord déterminer ce que nous cherchons. La première étape sera d'indiquer la classe recherchée. Ensuite, il est nécessaire de définir la condition de recherche. Pour cela, nous utilisons la classe `Condition`, mentionnée précédemment.

Il est logique et intuitif d'exprimer la condition de recherche sous une forme analogue aux opérateurs conditionnels. Par exemple, si nous voulons trouver une ville nommée `Dijon`, l'enregistrement sera le suivant :

```
where("name", "==", "Dijon")
```

La requête elle-même s'appelle `Query`. Nous voulons créer une nouvelle requête, et la première étape consiste à indiquer le type d'opération : `READ`, `CREATE`, `UPDATE` ou `DELETE`. Pour la lecture (la recherche), il est logique d'utiliser `select()`, puis de passer la classe avec laquelle nous travaillons, par exemple :

```
Query.select(City.Class)
```

Ensuite, nous pouvons ajouter une condition de recherche à l'aide de `where()`, par exemple :

```
Query.select(City.Class).where("name", "==", "Dijon")
```

Maintenant, considérons la création (`CREATE`). Que voulons-nous créer ?

- Si nous créons **une nouvelle classe** dans la base de données, il suffit d'indiquer les champs et leurs types. Dans ce cas, la construction sera simplement :

  ```
  Query.create(City.Class)
  ```

- Si nous créons **un nouvel objet**, nous devons passer les valeurs des attributs. Intuitivement, la méthode `add()` convient pour cela. Nous pouvons passer des paires "attribut — valeur" :

  ```
  Query.create(City.Class).add("name", "Dijon")
  ```

  Ou simplifier la syntaxe et passer l'objet prêt à l'emploi à la méthode `add()`, ce qui permettra d'obtenir automatiquement les valeurs de ses attributs via la réflexion.

Passons à la suppression (`DELETE`). Pour supprimer, nous devons spécifier les objets que nous voulons supprimer — de manière analogue à la recherche :

```
Query.delete(City.Class).where("name", "==", "Paris")
```

La mise à jour (`UPDATE`) est un peu plus complexe. Nous devons :

- indiquer la condition de recherche via `where()`
- et spécifier les nouvelles valeurs via `add()`

Par exemple :

```
Query.update(City.Class).where("name", "==", "Paris").add("population", 3000000)
```

Ajoutons maintenant le support des transactions. Nous considérerons que toute requête à la base de données fait partie d'une transaction. Pour cela, nous introduisons la classe `Transaction`, qui contient un identifiant unique `transactionId` et une liste de requêtes `Query` faisant partie de la transaction.

Intuitivement, nous voulons ajouter des requêtes à la transaction de la manière suivante :

```
Transaction.add(Query.select(City.Class).where("name", "==", "Dijon"))
```

Chaque transaction doit être envoyée au serveur. Pour ce faire, nous définissons un client qui établit la connexion avec le serveur — la classe `Session`. Comme chaque transaction doit avoir un ID unique, elle est créée via une session :

```
Session.createNewTransaction()
```

Cette méthode retourne une nouvelle transaction.

Pour exécuter la transaction, nous utilisons :

```
Session.execute(transaction)
```

Nous pourrions exécuter la dernière transaction ajoutée sans spécification explicite, mais une telle syntaxe serait moins déclarative.

Conclusion

Une telle syntaxe est progressive, claire et intuitive, se rapprochant de la logique naturelle de travail avec une base de données. La syntaxe finale d'interaction avec la base de données est la suivante :

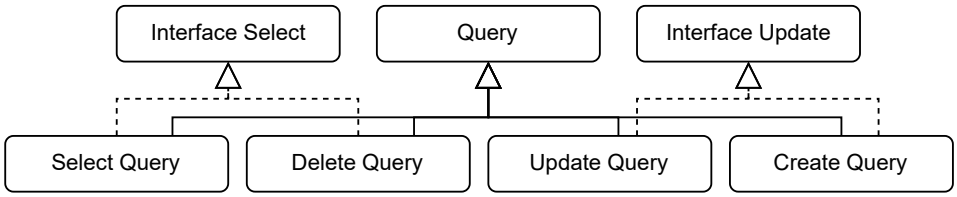

Figure 8: Query class diagram

```java
try(Session session = new Session(host, port)) {
            Transaction transaction = session.createNewTransaction();
            transaction.add(Query.create(Hero.class));
            transaction.add(Query.select(Hero.class).where("name", "==", "Tommy
                Sharp").where("age", ">", 35).all());
            transaction.add(Query.create(Hero.class).object(hero));
            transaction.add(Query.update(Hero.class).where("name", "contains",
                "Pedro").set("name", "Pedro Ivanov"));
            transaction.add(Query.delete(Hero.class).where("name", "=", "tommy sharp"));
            System.out.println(transaction);
            System.out.println(transaction.getQueries()[0].getAttributeValues().isEmpty());
            List<Result> results = session.execute(transaction);
            System.out.println(results);
        } catch (Exception e){
            e.printStackTrace();
        }
```

Listing 1: Transaction execution

Implémentation de la syntaxe en Java

Étant donné que chaque requête commence par le mot-clé `Query`, la première étape consiste à créer la classe correspondante.

Considérons la syntaxe suivante :

`Query.select(Object.Class)`

Ici, nous appelons la méthode statique `select()` de la classe `Query`, et nous devons :

- conserver le type d'opération — dans ce cas, `SELECT` ;
- enregistrer la classe à laquelle l'opération est appliquée ;
- garantir la possibilité de continuer la chaîne avec la méthode `where()`.

Cependant, il est nécessaire de différencier les méthodes applicables aux opérations `SELECT`/`DELETE` de celles applicables aux opérations `CREATE`/`UPDATE`. Pour cela, nous faisons en sorte que la méthode `select()` ne retourne pas le `Query` lui-même, mais une instance de la classe `SelectQuery`, qui définit uniquement les méthodes pertinentes (par exemple, `where()`).

La méthode `select()` sera `static`, ce qui signifie qu'elle ne garde pas d'état en elle-même. Par conséquent, à l'intérieur de celle-ci, nous appelons le constructeur de `SelectQuery` et lui passons :

- une référence à la classe avec laquelle l'opération sera effectuée ;
- le type de la requête (`SELECT`) ;
- des métadonnées sur les champs et leurs types.

Le constructeur de la classe `SelectQuery`, à son tour, délègue l'initialisation à la superclasse `Query`, en lui transmettant toutes les données nécessaires — garantissant ainsi un point unique de stockage des informations communes à la requête.

Il est important que, lors de l'ajout de paramètres à la requête (par exemple, via `where()`), nous devions effectuer :

- une vérification des types — la valeur passée doit correspondre au type de l'attribut obtenu à partir des métadonnées de la classe ;
- une vérification de la validité de l'opération — par exemple, tenter d'appliquer l'opérateur `"contains"` à un champ numérique doit entraîner une erreur.

Étant donné que les opérations `SELECT` et `DELETE` utilisent une logique similaire (travaillant avec des filtres), tandis que `CREATE` et `UPDATE` en utilisent une différente (travaillant avec l'ajout de valeurs), nous avons introduit des interfaces pour séparer ces logiques :

- `Select` — pour les opérations qui supportent la filtration (`where()`);
- `Update` — pour les opérations qui supportent l'ajout ou la modification des données (`add()`).

Cette approche a permis :

- d'uniformiser la logique de travail avec les requêtes au niveau de la classe de base `Query`;
- de garantir une structure de code sûre et logique;
- de réduire la probabilité d'erreurs lors de l'utilisation de la syntaxe;
- d'améliorer la lisibilité et la scalabilité du code.

Ainsi, l'architecture des classes est la suivante :

- **Query** — classe de base contenant les informations générales sur la requête;
- **SelectQuery, DeleteQuery** — héritiers de `Query`, implémentant l'interface `Select`;
- **CreateQuery, UpdateQuery** — héritiers de `Query`, implémentant l'interface `Update`;
- **Select**, **Update** — interfaces définissant les actions autorisées sur les requêtes.

La décision prise est intuitive et découle des exigences d'un DSL (domain-specific language) pour travailler avec une base de données.

Retour du résultat

Étant donné que nous avons initialement conçu l'architecture en tenant compte de l'interaction entre le client et le serveur, il est apparu nécessaire d'avoir une classe représentant le résultat de l'exécution de la transaction.

Tout d'abord, le résultat doit contenir le statut de l'exécution. Deux états sont possibles :

- `"OK"` — l'opération a été exécutée avec succès ;
- `"ERROR"` — une erreur est survenue pendant l'exécution.

En cas d'erreur, il est également nécessaire de renvoyer un message expliquant la cause de l'erreur. Cela permettra à l'utilisateur ou au développeur de comprendre rapidement ce qui a mal tourné.

Si le statut est `"OK"`, il est nécessaire de renvoyer le résultat de l'exécution. Celui-ci peut être varié :

- Dans le cas de la création ou de la suppression d'une table, généralement, il n'y a pas de résultat ;

- Dans le cas des opérations de lecture, le résultat sera une liste d'objets, représentée sous la forme :

  `ArrayList<Map<String, Object»`

  Chaque élément de la liste correspond à un objet, où la clé est le nom de l'attribut et la valeur correspond à la valeur associée.

Ainsi, nous obtenons un mécanisme universel de retour de résultat, couvrant à la fois les scénarios réussis et erronés.

Connexion `Query` et `TableManager`

Maintenant que le format d'interaction entre le client et le serveur est défini, nous pouvons passer à l'exécution réelle de la requête. Pour ce faire, nous avons mis en place une liaison entre `Query` et `TableManager`.

Nous avons introduit une classe auxiliaire `QueryToAction`, responsable de l'interprétation de la requête et de l'appel des méthodes correspondantes dans `TableManager`. Cette classe ne possède pas de constructeur et fonctionne via une méthode statique, prenant en entrée un objet de requête.

La principale tâche de `QueryToAction` est de déterminer quelle opération l'utilisateur souhaite effectuer et comment la traduire correctement en actions sur les tables.

Par exemple :

- Si une requête de type `Query.create(City.Class)` est reçue sans spécification d'attributs — cela signifie qu'une nouvelle table doit être créée.
- Si des valeurs d'attributs sont transmises avec la requête via `add()`, cela est interprété comme la création d'un nouvel objet du type correspondant.

L'approche choisie assure une séparation claire entre la logique de création de la requête (côté client) et son exécution (côté serveur), tout en rendant le système facilement extensible pour ajouter de nouvelles opérations.

Chapitre, 6 dans lequel on ecrit le serveur et tout se met en place.

Approche naive

> **Problem 3: Connexion entre le client et le serveur**
>
> Le client prend le script contenant les instructions sous forme de fichier texte et le transforme en flux binaire qu'il envoie au serveur à travers le réseau. Le serveur stocke ensuite le flux binaire reçu dans une chaine de caractères et donne la chaîne à l'interpréteur.

Nouvelle Approche

Description générale du système

Le serveur de transactions est implémenté sur la base d'une architecture multithreadée, garantissant un traitement efficace de plusieurs connexions clients simultanément. Le système repose sur plusieurs composants clés : un pool de threads pour gérer les connexions des clients, des threads séparés pour l'exécution des transactions, un buffer pour l'échange des tâches entre eux, ainsi qu'un mécanisme basé sur `CompletableFuture` assurant la livraison asynchrone des résultats.

Composants principaux

TransactionServer

La classe `TransactionServer` joue le rôle de point d'entrée principal du système. Lors du démarrage, elle initialise le socket serveur, lance un pool fixe de threads pour gérer les connexions des clients, ainsi qu'un nombre configurable

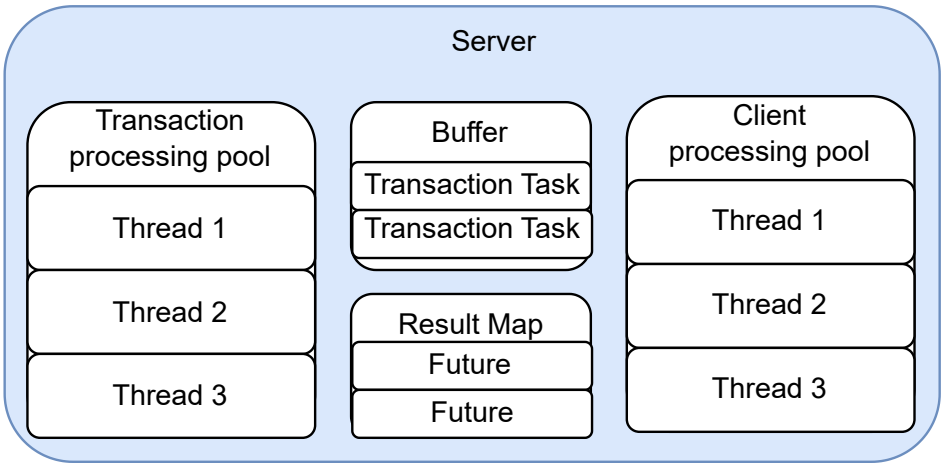

Figure 9: Server scheme

de threads pour l'exécution des transactions. Toutes les connexions entrantes sont traitées et distribuées entre les gestionnaires, et la fermeture du serveur se fait en terminant correctement tous les composants impliqués. La configuration du serveur comprend le port, la taille du pool de gestionnaires, le nombre de threads pour l'exécution des transactions et l'interface d'interaction avec la base de données.

ClientHandler

`ClientHandler` implémente l'interface `Runnable` et est responsable de la lecture des transactions à partir des connexions entrantes, de la création des tâches et de leur transfert dans le buffer. Ensuite, il attend le résultat de l'exécution en utilisant `CompletableFuture` et le renvoie au client. Une fois le travail terminé, la connexion est fermée.

TransactionExecutor

Ce composant, qui implémente également `Runnable`, interroge en continu le buffer des tâches. Lorsqu'une nouvelle tâche est reçue, il exécute toutes les requêtes à l'intérieur de la transaction et termine le `CompletableFuture` associé, en renvoyant les résultats obtenus.

TransactionBuffer

Le buffer des transactions sert de lien entre les gestionnaires et les exécutants. Il implémente une file d'attente de tâches thread-safe, génère des identifiants uniques pour les transactions et les associe à des objets `CompletableFuture`, permettant la livraison asynchrone des résultats.

TransactionTask

La tâche de transaction (`TransactionTask`) encapsule l'objet transaction et la variable future associée aux résultats. Elle est utilisée comme unité de travail dans le processus de traitement.

Flux de traitement des transactions

Le processus de traitement des transactions commence par la connexion du client au serveur. Une fois la connexion acceptée, le serveur la transmet à l'un des gestionnaires. Celui-ci lit l'objet de transaction, forme une tâche (`TransactionTask`) et la place dans le buffer. L'exécutant, ayant reçu la tâche du buffer, exécute les requêtes

et termine le `CompletableFuture`. Le gestionnaire, ayant attendu le résultat, le reçoit et l'envoie au client, puis termine la connexion. Toute la chaîne de travail est construite sur le modèle producteur-consommateur, avec une coordination asynchrone.

Synchronisation des threads

Le système utilise plusieurs structures de données thread-safe pour garantir une synchronisation fiable entre les threads. La file d'attente des tâches est implémentée via `ConcurrentLinkedQueue`, les associations entre les identifiants des transactions et les résultats sont gérées par `ConcurrentHashMap`, et la génération d'identifiants uniques se fait grâce à `AtomicInteger`. La coordination de l'exécution est réalisée via `CompletableFuture`.

Détails de l'implémentation du buffer

Le buffer des transactions implémente le modèle classique producteur-consommateur. Les threads gestionnaires (producers) ajoutent des tâches dans la file d'attente, et si la transaction n'a pas encore d'identifiant, celui-ci est généré automatiquement. Les résultats de l'exécution de la transaction sont envoyés dans le `CompletableFuture` correspondant à son identifiant. Un exemple d'implémentation est donné ci-dessous :

```
public class TransactionBuffer {
    private final ConcurrentLinkedQueue<TransactionTask> buffer;
    private final ConcurrentHashMap<Integer, CompletableFuture<List<Result>>> resultMap;
    private final AtomicInteger transactionIdGenerator;

    // The method is called by ClientHandler threads (producers)
    public void add(TransactionTask task) {
        Transaction transaction = task.getTransaction();
        if(transaction.getTransactionId() == -1)
            transaction.changeTransactionId(generateTransactionId());

        buffer.add(task);
        resultMap.put(transaction.getTransactionId(), task.getResultFuture());
    }

    // The method is called by TransactionExecutor threads (consumers)
    public TransactionTask poll() {
        return buffer.poll();
    }

    // The method is called by transaction executors to deliver results
    public void completeTransaction(int transactionId, List<Result> results) {
        CompletableFuture<List<Result>> future = resultMap.remove(transactionId);
        if (future != null) {
            future.complete(results);
        }
    }
}
```

Performance et tolérance aux pannes

L'architecture du système est conçue en mettant l'accent sur la performance : séparation de la logique client et transactionnelle, utilisation de pools de threads configurables, travail non-bloquant via `CompletableFuture`, ainsi que minimisation des coûts d'overhead grâce aux collections thread-safe.

En ce qui concerne la tolérance aux pannes, le système gère les exceptions du côté des connexions clients, maintient une terminaison correcte des composants et utilise des constructions `try-with-resources` pour la gestion des ressources. Les transactions sont traitées de manière isolée, ce qui améliore la résistance aux erreurs à chaque étape.

Chapitre 7, dans lequel nous résumons et explorons des alternatives.

Résultats obtenus

Dans le cadre du projet, une base de données orientée objet basée sur une organisation par pages a été développée avec succès, offrant les principales fonctionnalités suivantes :

- Création et suppression de classes et d'objets.
- Recherche efficace d'objets par attributs avec prise en charge des conditions.
- Traitement multithread des transactions avec mécanismes de prévention des conflits et des blocages.
- Architecture client-serveur comprenant une bibliothèque utilisateur et un serveur pour l'interaction distante.
- Syntaxe intuitive des requêtes, inspirée de SQL et des ORM modernes.

Les principales solutions techniques reposent sur une organisation des données par pages (4 Ko) pour optimiser la performance, la séparation des types de pages, la mise en cache avec stratégie d'éviction. Le traitement asynchrone des transactions sur le serveur utilise `CompletableFuture`.

Limitations et fonctionnalités non implémentées

Malgré les résultats obtenus, certains aspects restent non réalisés :

1. `Mise à jour des attributs des objets` - cette fonctionnalité était planifiée, mais non terminée. La mise en œuvre nécessite quelques ajustements mineurs dans `TableManager`.
2. `Interface CLI` - les premières ébauches sont disponibles dans le dossier `PreviousIdea`.
3. `Authentification des utilisateurs` - ce mécanisme n'a pas été intégré, bien que sa structure de base soit déjà prévue dans l'architecture des classes.

Ces tâches pourront être développées dans de futures versions.

Améliorations possibles

Le projet possède un potentiel considérable pour des extensions futures :

1. Extension des types de données :
    - Prise en charge des chaînes de longueur variable dans `StringPage`.
    - Ajout de nouveaux types de pages (par exemple, `TimePage` pour la gestion des horodatages).
2. Amélioration de la recherche : Nouveaux opérateurs (`startsWith`, `endsWith`, `fuzzy match` pour les chaînes).
3. Extension du modèle objet : Imbrication des objets (attributs-objets d'autres classes).
4. Optimisations :
    - Gestion mémoire plus flexible pour les pages à taille dynamique.
    - Amélioration du mécanisme de transaction pour supporter les propriétés ACID.

Approches alternatives

Durant le développement, plusieurs solutions alternatives ont été envisagées :

- `Sérialisation standard` — rejetée en raison de problèmes de performance et de fragmentation.
- `Versionnage des pages` — remplacé par un mécanisme de verrouillage ordonné pour éviter les blocages.
- `Un fichier unique pour tous les objets` — moins flexible par rapport à l'organisation par pages.

Conclusion

Le projet a démontré la viabilité de l'architecture choisie et a atteint les objectifs fixés. Bien que certains éléments restent à compléter, la base de données présente une grande extensibilité et peut servir de fondation pour de futurs développements. Les principaux avantages sont les suivants :

- Performance grâce au stockage par pages.
- Flexibilité grâce à une structure modulaire.
- Facilité d'utilisation via une syntaxe de requêtes bien pensée.

Repartition des taches

- Automates et interpreteurs: Mathieu Lemain
- Back-end: Egor Semenov
- Front-end: Egor Semenov et Mathieu Lemain
- Rapport: Egor Semenov et Mathieu Lemain
- Assemblage final: Egor Semenov et Mathieu Lemain

Liste de références

References

[1] T. Connolly, C. Begg, *Database Systems: A Practical Approach to Design, Implementation, and Management.*

[2] Jean-Luc Hainaut, *Bases de Données: Concepts, Utilisation et Développement.*

[3] Sebastián Ramírez, *SQLModel.* [Ressource électronique]. — Mode d'accès: https://sqlmodel.tiangolo.com/

[4] Robert Nystrom, *Crafting Interpreters.* [Ressource électronique]. — Mode d'accès: https://craftinginterpreters.com/

[5] Vladimir Scherbakov, *Comment créer son propre SGBD à partir de zéro sans devenir fou. Guide pratique du nécromancien débutant*, Habr, 2022. [Ressource électronique]. — Mode d'accès: https://habr.com/ru/articles/709234/