Base de données

- 11-06 matin Cours : Bases de Données - Conception, Modélisation et Technologies Modernes
- 11-06 après-midi Formation : Architecture et Administration des Bases de Données - Théorie et Pratique
- 21-05 cours : Chapitre 5 : Optimisation, Maintenance et Architectures Avancées des Bases de Données
- 21-05 Cours: Modélisation NoSQL et Sécurité des Systèmes d’Information — schémas, index, chiffrement, audit, monitoring

11-06 matin Cours : Bases de Données - Conception, Modélisation et Technologies Modernes

Bases de Données : Conception, Modélisation et Technologies Modernes

Introduction : L'Évolution des Systèmes de Gestion de Données

Les bases de données constituent l'épine dorsale des systèmes d'information modernes, servant de fondement à pratiquement toutes les applications informatiques contemporaines. L'évolution technologique récente a considérablement élargi le spectre des solutions disponibles, allant des bases de données relationnelles traditionnelles aux systèmes NoSQL spécialisés, en passant par les architectures distribuées à grande échelle.
Cette diversification répond à des besoins croissants de scalabilité, de flexibilité et de performance dans un contexte où les volumes de données atteignent des proportions sans précédent.
Comprendre ces technologies, leurs forces et leurs limitations devient essentiel pour tout professionnel de l'informatique moderne.

Chapitre 1 : Types de Données et Optimisation

1.1 Fondements de l'Encodage des Données

L'encodage des données représente le processus critique de conversion d'informations compréhensibles par l'humain vers un format manipulable par les systèmes informatiques. Cette transformation doit prendre en compte non seulement la **partie entière**des valeurs numériques,

mais également leur **partie fractionnaire**, particulièrement cruciale dans certains domaines d'application.

Domaines critiques pour la précision :

- **Applications financières**: où chaque décimale peut représenter des montants considérables
- **Calculs scientifiques**: nécessitant une précision maximale pour éviter les erreurs de propagation
- **Applications temps réel**: où la vitesse de traitement est critique mais ne doit pas compromettre l'exactitude

1.2 Stratégies d'Optimisation des Types Numériques

Le dimensionnement approprié des types de données constitue un enjeu majeur d'optimisation. Le principe fondamental consiste à **adapter la taille du type aux besoins réels**de l'application, évitant ainsi le gaspillage de ressources.

Tailles standard et leurs plages :

- **8 bits**: 0 à 255 (unsigned) ou -128 à 127 (signed)
- **16 bits**: 0 à 65 535 (unsigned) ou -32 768 à 32 767 (signed)
- **32 bits**: 0 à 4 294 967 295 (unsigned)
- **64 bits**: plage considérablement élargie pour les très grandes valeurs
  Cas d'application concret : Base Interpol
  Un exemple emblématique illustre ces enjeux : le développement d'une base de données pour Interpol destinée à stocker les passeports volés, plaques d'immatriculation et personnes recherchées. Cette base devait être répliquée sur des postes portables déployés dans les forces de police mondiales, avec des contraintes particulières :
- **Fonctionnement hors ligne**: autonomie complète nécessaire
- **Synchronisation limitée**: mise à jour quotidienne ou hebdomadaire
- **Connexions dégradées**: débit faible, taux de perte de paquets élevé
- **Intégrité critique**: gestion des checksums et retry
  Dans ce contexte, chaque octet économisé réduisait le temps de transfert, améliorait la fiabilité de synchronisation et diminuait les coûts de communication satellitaire.

1.3 Types de Données Textuelles : CHAR versus VARCHAR

La gestion des chaînes de caractères implique un choix stratégique entre deux approches fondamentalement différentes :

Type CHAR (taille fixe) :

- Réserve toujours le même espace disque
- Performance prévisible grâce à l'accès direct par offset
- Risque de gaspillage si la chaîne est plus courte que l'espace alloué
  Type VARCHAR (taille variable) :
- Adapte l'espace à la longueur réelle du contenu
- Optimise l'utilisation de l'espace disque

- Nécessite un overhead de gestion pour stocker la longueur réelle
  Stratégies de choix :
  Utiliser CHAR lorsque :
- Les données ont toujours la même longueur (codes pays, identifiants fixes)
- Les performances d'accès sont prioritaires
- Le volume de données est relativement faible
  Utiliser VARCHAR lorsque :
- Les données ont des longueurs très variables
- L'économie d'espace est prioritaire
- Le volume total est important (contexte Big Data)

Chapitre 2 : Systèmes de Clés et Relations

2.1 Concept Fondamental des Clés

Une **clé** en base de données fonctionne conceptuellement comme une clé physique : elle permet d'**accéder de manière unique** à une ressource spécifique. Dans le contexte des bases de données, elle garantit l'identification univoque d'un enregistrement parmi potentiellement des millions d'autres.

2.2 Clés Primaires : Identification et Évolution

La clé primaire constitue l'**identifiant unique** de chaque enregistrement dans une table, devant respecter deux contraintes fondamentales :

- **Unicité** : aucune duplication possible
- **Non-nullité** : toujours présente et définie
  Types de clés primaires :
  Clés naturelles : basées sur les attributs métier existants (numéro de sécurité sociale, email unique)
  Clés artificielles (surrogate keys) : identifiants techniques générés spécifiquement pour l'identification, offrant l'avantage d'être indépendants des données métier mais impliquant des données supplémentaires sans valeur métier directe.

2.3 Évolution vers les Systèmes Distribués

L'approche traditionnelle d'auto-incrémentation (ID : 1, 2, 3, 4..) présente des limitations majeures dans les systèmes distribués :

- **Problèmes de scalabilité** sur systèmes distribués
- **Conflits de synchronisation** entre serveurs multiples
- **Goulots d'étranglement** dans les architectures parallèles
  Solutions modernes :
- **UUID** (Universally Unique Identifier) : identifiants de 128 bits garantissant l'unicité globale

- **GUID** (Globally Unique Identifier) : basés sur des algorithmes cryptographiques et temporels
- **Stratégies de partitionnement** : nombres pairs/impairs par serveur, préfixes géographiques
  Ces approches éliminent les collisions et permettent une génération locale sur chaque nœud sans synchronisation.

2.4 Clés Étrangères et Intégrité Référentielle

Les **clés étrangères** établissent les liens entre tables, formant l'architecture relationnelle. Contrairement aux clés primaires, elles peuvent apparaître plusieurs fois dans une table et référencent obligatoirement la clé primaire d'une autre table.
Problématique des clés métier :
Les clés métier possèdent souvent une signification fonctionnelle qui peut évoluer. Par exemple, dans un système RH :

- Matricules commençant par 8 : cadres
- Matricules commençant par 2 : ETAM (Employés, Techniciens et Agents de Maîtrise)
- Matricules commençant par 1 : externes
  Le problème majeur réside dans leur **mutabilité** : lorsqu'un employé change de statut, son matricule doit évoluer, nécessitant des modifications complexes dans toutes les tables liées.
  Solution : clés de substitution
  Les **clés techniques** résolvent cette problématique en étant :
- Artificielles sans signification métier
- **Immuables** une fois attribuées
- Compatibles avec l'existence simultanée de clés secondaires uniques pour les besoins métier

2.5 Contraintes d'Intégrité Référentielle

La déclaration explicite des contraintes lors de la création des tables est essentielle. SQL propose plusieurs options via la clause `ON DELETE` :
RESTRICT (comportement par défaut) : empêche la suppression tant qu'une référence existe
SET NULL : remplace automatiquement la référence par NULL lors de la suppression
CASCADE : supprime automatiquement tous les enregistrements référençant l'élément supprimé
Ces contraintes offrent une protection automatique par le SGBD et constituent une documentation vivante des relations entre entités.

Chapitre 3 : Transactions et Propriétés ACID

3.1 Contexte Historique et Enjeux

Les systèmes de gestion de bases de données relationnelles ont émergé dans le secteur bancaire, où la gestion simultanée de milliers de transferts d'argent nécessitait des garanties de cohérence absolues. Cette origine explique l'importance accordée aux propriétés transactionnelles.
Exemple illustratif : transfert bancaire
Un transfert de 1000€ implique :

1. Débiter 1000€ du compte source
2. Créditer 1000€ sur le compte destination
   Les problèmes de concurrence peuvent provoquer :

- **Panne système** : argent disparu ou dupliqué si interruption entre les deux opérations
- **Accès concurrent** : lecture simultanée du même solde par deux transactions, causant des pertes

3.2 Propriétés ACID Détaillées

A - Atomicité

Principe "tout ou rien": une transaction est indivisible. Soit toutes les opérations réussissent, soit aucune n'est appliquée. En cas d'échec partiel, le système effectue un **rollback** automatique.

C - Cohérence

La base doit respecter toutes les règles de cohérence définies avant et après chaque transaction. En environnement distribué, cette propriété devient particulièrement complexe car elle doit considérer l'ensemble du système, non chaque nœud isolément.
Problématique distribuée : Si deux serveurs (Paris et Lyon) tentent simultanément de débiter 10 000€ d'un compte contenant exactement cette somme, chaque serveur peut valider individuellement l'opération, résultant en un solde final de -10 000€.

I - Isolation

Les transactions concurrentes ne doivent pas interférer entre elles. Cette propriété présente un dilemme entre **performance** et **exactitude**, résolu par différents niveaux d'isolation :
Niveaux d'isolation (du plus strict au plus permissif) :

1. **Serializable** : isolation maximale, performance minimale
2. **Repeatable Read** : lectures cohérentes dans une transaction
3. **Read Committed** : lecture des données committées uniquement
4. **Read Uncommitted** : performance maximale, risques de dirty reads

D - Durabilité

Une fois validée (commit), une transaction doit persister même en cas de panne système ultérieure. Cette propriété nécessite une écriture physique sur disque, pas seulement en cache.
Défis techniques : Les multiples couches de cache (OS, contrôleur RAID, virtualisation) peuvent compromettre la durabilité si une coupure survient avant l'écriture physique effective.

3.3 Stratégies de Verrouillage

Pessimistic Locking : verrouillage préventif des ressources, correspondant au niveau Serializable
Optimistic Locking : pari sur l'absence de conflits avec détection a posteriori. Problématique dans les applications à fort trafic où les collisions deviennent fréquentes.

Chapitre 4 : Fonctionnement du Moteur et Optimisation des Requêtes

4.1 Architecture Physique et Logique

Une base de données constitue fondamentalement un **espace de stockage sur disque** contenant des milliards d'octets organisés en bits. Le SGBD agit comme intermédiaire intelligent entre l'utilisateur et ces données brutes, gérant le chargement depuis le stockage permanent vers la mémoire vive selon les besoins.
Cette opération de chargement détermine les performances de toute requête, le moteur utilisant des **pointeurs** pour naviguer dans les structures et ne chargeant initialement que les métadonnées nécessaires.

4.2 Ordre d'Exécution Logique des Requêtes

Contrairement à l'intuition naturelle, l'ordre d'exécution ne suit pas l'ordre syntaxique. La séquence logique est :

1. **FROM** : identification et chargement des tables sources
2. **JOIN** : création de tables temporaires par produit cartésien filtré
3. **WHERE** : filtrage ligne par ligne des données
4. **GROUP BY** : regroupement logique des données
5. **HAVING** : filtrage des groupes après agrégation
6. **SELECT** : sélection et calcul des colonnes finales
7. **UNION** : combinaison de jeux de résultats
8. **ORDER BY** : tri final

4.3 Mécanismes des Jointures

Les jointures créent des **tables temporaires**en mémoire via un produit cartésien conceptuel entre les tables impliquées. Le moteur optimise ce processus en :

- Choisissant intelligemment la table de base
- Appliquant les critères de jointure pour éliminer les combinaisons non pertinentes
- Utilisant les index disponibles pour accélérer les opérations
  La taille de ces tables temporaires impacte directement les performances, particulièrement dans le contexte Big Data où joindre des tables de millions de lignes peut générer des structures de plusieurs gigaoctets.

4.4 Optimisations et Recommandations

Méthodologie de construction : commencer par FROM et identifier toutes les sources de données, puis progresser selon l'ordre logique d'exécution.
Optimisation mémoire : l'ajout de RAM peut diviser les temps d'exécution par 10 en permettant l'utilisation de stratégies de jointure en mémoire plutôt que sur disque.
Standards de portabilité : SQL est normalisé depuis ANSI SQL 92 (1992), couvrant environ 80% des besoins courants. Les extensions spécifiques (fonctions spatiales, analytiques) varient selon les SGBD mais suivent une logique similaire.

Chapitre 5 : SQL Avancé et Agrégations

5.1 La Clause HAVING : Filtrage Post-Agrégation

La clause **HAVING** constitue l'un des mécanismes les plus puissants pour le filtrage de données après agrégation. Sa distinction temporelle avec WHERE est cruciale : **WHERE filtre avant agrégation, HAVING filtre après**.
Exemple pratique d'analyse salariale :

```
SELECT annee_embauche,
MAX(salaire) as salairemax,
   AVG(salaire) as salairemoyen
FROM employes
GROUP BY annee_embauche
HAVING MAX(salaire) > 2 * AVG(salaire);
```

Cette requête identifie les années présentant des disparités salariales importantes, révélant des politiques de rémunération potentiellement déséquilibrées.
Règle fondamentale : HAVING ne peut exister sans GROUP BY, cette interdépendance reflétant la logique de traitement séquentiel des données.

5.2 Ordre Logique et Optimisation Conceptuelle

L'approche séquentielle optimise naturellement les performances :

1. Filtrage des données non pertinentes (WHERE)

2. Regroupement sur un ensemble réduit
3. Calculs uniquement sur les données finales
   Cette méthode évite des calculs coûteux (prix TTC = prix × quantité × TVA) sur des lignes destinées à être supprimées.

5.3 Opérations d'Union et Tri Global

UNION vs UNION ALL :

- **UNION** : élimine automatiquement les doublons
- **UNION ALL** : conserve toutes les lignes, plus performant
  L'ORDER BY s'applique au résultat consolidé après toutes les unions, nécessitant des parenthèses explicites pour trier avant fusion.

Chapitre 6 : Introduction au NoSQL

6.1 Philosophie et Définition

NoSQL signifie "**Not Only SQL**" et non "No SQL", reflétant une approche complémentaire plutôt qu'antagoniste. Cette nuance traduit une philosophie d'ouverture reconnaissant que d'autres approches peuvent être plus adaptées dans certains contextes.

6.2 Motivations et Avantages

Scalabilité horizontale : contrairement à l'approche verticale (ajout de puissance), l'approche horizontale permet d'ajouter de nouveaux nœuds pour absorber la charge croissante, crucial dans le contexte Big Data.
Relaxation des contraintes ACID : échange d'une partie des garanties contre des performances accrues lorsque les exigences fonctionnelles le permettent.
Flexibilité schématique : évolution dynamique des structures, adaptation rapide aux changements de besoins, contrairement à la rigidité relationnelle.

6.3 Modèles de Données et Applications

Document (MongoDB, Elasticsearch) : stockage de documents structurés, optimal pour les catalogues hétérogènes
Clé-valeur (Redis, DynamoDB) : dictionnaire géant optimisé pour la performance, idéal pour les caches et sessions
Colonnes (Cassandra, HBase) : optimisation pour l'analytique avec stockage columnar permettant des compressions efficaces
Graphes (Neo4j, ArangoDB) : gestion des relations complexes, excellent pour les réseaux sociaux et recommandations
Time Series (InfluxDB, Prometheus) : optimisation temporelle avec architecture immutable "Write-Once, Read-Many"

Chapitre 7 : NoSQL Spécialisé et Théorème CAP

7.1 Problématiques des Schémas Variables

Les systèmes NoSQL permettent une **flexibilité schématique** remarquable. Un document JSON peut contenir des attributs absents dans d'autres documents de la même collection, avantage crucial pour les marketplaces hétérogènes où chaque catégorie possède des caractéristiques spécifiques.
Exemple e-commerce : un produit livre aura `{titre, auteur, ISBN, pages}` tandis qu'un vêtement aura `{nom, couleur, taille, matière}`. En relationnel, cela nécessiterait une table avec tous les champs possibles, générant majoritairement des valeurs NULL.

7.2 Architecture des Data Lakes

L'approche Data Lake consiste à :

1. **Enregistrer** chaque événement sous forme brute
2. **Stocker** sans transformation préalable
3. **Traiter** a posteriori selon les besoins analytiques

Cette philosophie s'intègre parfaitement avec les capacités NoSQL de gestion de données semi-structurées.

7.3 Théorème CAP : Compromis Fondamentaux

Le théorème CAP établit qu'un système distribué ne peut garantir simultanément :
C (Consistency) : tous les nœuds voient les mêmes données simultanément
A (Availability) : le système répond toujours aux requêtes
P (Partition Tolerance) : fonctionnement malgré les pannes réseau
Trois combinaisons possibles :
CP : sacrifice de la disponibilité pour maintenir cohérence (bases relationnelles distribuées)
AP : acceptation d'inconsistance temporaire pour rester opérationnel (réseaux sociaux)
CA : fonctionnement parfait sans pannes réseau (bases traditionnelles mono-site)

7.4 Eventual Consistency

L'**eventual consistency** représente le compromis pragmatique des systèmes AP : toutes les écritures seront propagées et la convergence surviendra "éventuellement", sans garantie de délai précis.
Illustration : publication Facebook immédiatement sauvegardée (availability) mais propagation progressive vers les timelines (eventual consistency).

Chapitre 8 : Choix Technologique et Architectures Hybrides

8.1 Critères de Décision

Le choix entre SQL et NoSQL dépend fondamentalement du **pattern d'accès** aux données :
Favoriser SQL :

- Requêtes imprévisibles
- Jointures complexes fréquentes
- Propriétés ACID critiques
- Données fortement structurées
  Favoriser NoSQL :
- Patterns d'accès prévisibles
- Volume nécessitant scalabilité horizontale
- Performance prioritaire sur consistance immédiate
- Données semi-structurées ou variables

8.2 Cas d'Usage Sectoriels

Système de réservation (aviation) : base relationnelle pour propriétés transactionnelles critiques, cohérence forte, relations complexes
Catalogue e-commerce : architecture hybride avec MongoDB pour le catalogue (attributs variables) et PostgreSQL pour les transactions
Application IoT : base Time Series (InfluxDB) pour volume d'écriture élevé, données horodatées, agrégations temporelles

8.3 Architectures Polyglotte

Dans la pratique, les grandes applications utilisent **plusieurs types de bases** simultanément :

- **Base relationnelle** : données de référence, transactions financières
- **Base documentaire** : catalogue produits, contenu utilisateur
- **Cache clé-valeur** : sessions, données temporaires
- **Base time series** : métriques, logs, monitoring
  Cette approche nécessite des **pipelines ETL** pour synchroniser les données entre systèmes, maintenir la cohérence globale, mais ajoute une complexité administrative et opérationnelle considérable.

Chapitre 9 : Modélisation et Conception

9.1 Universalité de la Modélisation

La modélisation constitue une étape fondamentale **indépendamment du type de base** choisi. Même les bases NoSQL nécessitent des décisions cruciales : pour Elasticsearch, définir un champ "titre" implique des choix de tokenisation, indexation, recherche et sémantique impactant directement performances et pertinence.

9.2 Niveaux d'Abstraction

Niveau conceptuel (QUOI) : concepts métiers et relations, indépendance technologique totale, langage accessible aux non-techniques
Niveau logique (COMMENT) : traduction en structures de données, indépendant de l'implémentation spécifique
Niveau physique (AVEC QUOI) : spécifications techniques liées au SGBD choisi

9.3 Transformation et Règles

Règles fondamentales :

- Entité → Table
- Association 1..N → Clé étrangère côté N
- Association N..N → Table de jointure
- Optimisations 1..1 pour performance
  La compréhension des mécanismes de stockage (chargement par blocs de 32-64 Ko) guide les décisions de modélisation, justifiant l'externalisation des données volumineuses.

Chapitre 10 : Normalisation Théorique et Pratique

10.1 Objectifs Fondamentaux

La normalisation vise à :

- **Éliminer la redondance** : éviter la duplication d'informations
- **Prévenir les anomalies** : insertion, suppression, mise à jour
- **Optimiser la structure** : maintenir la cohérence et l'intégrité

10.2 Formes Normales Essentielles

Première Forme Normale (1NF) : atomicité du contenu, exclusion des listes de valeurs dans une colonne
Deuxième Forme Normale (2NF) : dépendance fonctionnelle complète de la clé primaire, particulièrement importante pour les clés composites
Troisième Forme Normale (3NF) : élimination des dépendances transitives, toutes les informations concernent directement l'entité identifiée par la clé primaire

10.3 Formes Normales Avancées (4NF et 5NF)

Quatrième Forme Normale (4NF): S'applique aux dépendances multi-valuées, suggérant de séparer les attributs non-clés qui dépendent de plusieurs autres attributs non-clés (ex: listes de compétences et de langues d'un employé). Son application stricte peut complexifier l'usage par la multiplication des tables et jointures.
Cinquième Forme Normale (5NF): Concerne les dépendances de jointures, visant à optimiser la reconstruction des données via des jointures. Elle est considérée comme très théorique et rarement appliquée en pratique.
Pertinence pratique: Ces formes avancées sont peu utilisées explicitement dans la modélisation quotidienne, les professionnels visant intuitivement la 3NF et n'envisageant la dénormalisation que de manière stratégique.

10.4 Dénormalisation Stratégique

La **dénormalisation**consiste à introduire volontairement de la redondance pour optimiser les performances. Justifiée dans les contextes :

- **Big Data et analytique** : tables de faits pré-jointes
- **Données principalement en lecture** : optimisation des requêtes fréquentes
- **Performance critique** : réduction des jointures nécessaires
  Cette approche nécessite d'accepter consciemment la redondance, complexité de mise à jour et risques d'incohérence en échange des bénéfices escomptés.

10.5 Réalité Professionnelle

Dans la pratique, l'application des formes normales est souvent **intuitive**, guidée par les bonnes pratiques de programmation orientée objet. La référence explicite à la théorie reste rare, l'accent étant mis sur le pragmatisme et les résultats business.

Schéma Récapitulatif

https://archive.qoyri.fr/mindmaps/cours06.11.25.html

Concepts Principaux

Types de Données et Optimisation :

- Dimensionnement selon besoins réels
- CHAR (fixe) vs VARCHAR (variable)
- Impact performance/stockage

Système de Clés :

- Clé primaire : identification unique
- Clés artificielles : indépendance métier
- UUID/GUID : solutions distribuées
- Intégrité référentielle : contraintes ON DELETE

Transactions et ACID :

- Atomicité : tout ou rien
- Cohérence : respect des règles
- Isolation : niveaux et compromis performance
- Durabilité : persistance garantie

SQL et Moteur :

- Ordre logique : FROM → JOIN → WHERE → GROUP BY → HAVING → SELECT → UNION → ORDER BY
- Optimisation : tables temporaires, jointures
- HAVING : filtrage post-agrégation

NoSQL et Alternatives :

- Définition : "Not Only SQL"
- Types : Document, Clé-valeur, Colonnes, Graphes, Time Series
- Théorème CAP : Consistency, Availability, Partition Tolerance
- Eventual consistency vs cohérence forte

Modélisation :

- Niveaux : Conceptuel, Logique, Physique
- Règles de transformation
- Normalisation vs Dénormalisation

Mots-Clés Essentiels

Technique : ACID, UUID, Sharding, Réplication, Index, Jointures, Produit cartésien, Table temporaire, ETL
Conceptuel : Entité, Association, Cardinalité, Clé primaire/étrangère, Atomicité, Cohérence, Isolation, Durabilité
Architectures : Scalabilité horizontale/verticale, Architecture polyglotte, Data Lake, Eventual consistency, CAP, NoSQL, Time Series
Modélisation : Normalisation, Dénormalisation, Forme normale, Redondance, Anomalie, Dépendance fonctionnelle, Clé de substitution

Cette synthèse constitue un fondement solide pour comprendre l'évolution des technologies de bases de données et maîtriser les choix architecturaux adaptés aux contextes applicatifs modernes. La compréhension de ces concepts permet de naviguer efficacement entre solutions

relationnelles traditionnelles et alternatives NoSQL, en fonction des contraintes spécifiques de chaque projet.

11-06 après-midi Formation : Architecture et Administration des Bases de Données - Théorie et Pratique

Architecture et Administration des Bases de Données : Théorie et Pratique

Introduction : L'écosystème des systèmes d'information modernes

Dans le paysage complexe des technologies de l'information contemporaines, la gestion des bases de données constitue un pilier fondamental de toute infrastructure informatique. Contrairement à une vision simpliste qui imaginerait une organisation reposant sur une unique base de données monolithique, la réalité moderne révèle un écosystème sophistiqué et distribué, où chaque

composant répond à des besoins spécifiques et complémentaires. Cette complexité n'est pas accidentelle : elle résulte d'une évolution naturelle vers la spécialisation et l'optimisation. À mesure que les organisations croissent et que leurs besoins se diversifient, l'architecture des données se structure selon une logique de distribution fonctionnelle qui permet d'optimiser les performances, la maintenance et la sécurité.

L'architecture distribuée : une nécessité fonctionnelle

Au cœur de toute organisation moderne, qu'il s'agisse d'une institution financière ou d'une entreprise industrielle, on retrouve des **systèmes centraux** qui constituent l'épine dorsale informationnelle. Dans le secteur bancaire, le *core banking* centralise l'ensemble des opérations fondamentales : gestion des comptes clients, traitement des transactions, administration des produits financiers et respect des contraintes réglementaires. Pour les entreprises classiques, l'**ERP** (Enterprise Resource Planning) - qu'il s'agisse de solutions comme SAP ou de développements propriétaires - intègre comptabilité, gestion des stocks, ressources humaines et l'ensemble des processus métier critiques. Ces systèmes centraux s'appuient généralement sur de volumineuses bases de données relationnelles capables de traiter des téraoctets d'informations avec des exigences de cohérence, de disponibilité et de performance extrêmement élevées. Cependant, ils ne fonctionnent jamais de manière isolée. En parallèle, une multitude de bases de données spécialisées coexistent dans l'écosystème informatique : systèmes de gestion de contenu (CMS) pour les sites web d'entreprise, applications métier spécifiques comme les CRM ou les outils de reporting, bases de données analytiques dédiées au business intelligence, systèmes de cache pour l'optimisation des performances, et bien d'autres encore. Cette architecture distribuée présente des avantages considérables : spécialisation fonctionnelle permettant d'optimiser chaque composant pour son usage spécifique, isolation des risques évitant qu'un problème sur un système affecte l'ensemble, facilité de maintenance et d'évolution, et possibilité de faire appel à des technologies différentes selon les besoins.

Le métier de Database Administrator : évolution et responsabilités

Transformation du rôle du DBA

Le métier de Database Administrator (DBA) a connu une évolution remarquable au cours des dernières décennies. Autrefois poste dédié à temps plein dans les grandes organisations, avec des spécialistes exclusivement focalisés sur la gestion des bases de données, il tend aujourd'hui à devenir une **compétence transversale** que les professionnels IT doivent maîtriser en complément d'autres responsabilités. Cette transformation s'explique par plusieurs facteurs convergents : l'automatisation croissante des tâches d'administration grâce aux outils modernes, la démocratisation des systèmes de gestion de bases de données avec des interfaces plus accessibles, et surtout la nécessité d'une approche DevOps qui intègre développement, administration système et gestion des données dans une vision cohérente. Le DBA moderne doit naviguer entre plusieurs domaines d'expertise : une compréhension fine de l'infrastructure (hardware, réseau, stockage), une collaboration étroite avec les équipes de développement pour l'optimisation applicative, et une connaissance approfondie des enjeux métier pour une modélisation optimale des données.

Responsabilités techniques fondamentales

Installation et configuration : approches multiples

Le DBA maîtrise plusieurs méthodologies d'installation selon les contextes et les contraintes. L'**installation native** suit la documentation officielle du SGBD et offre un contrôle maximal sur la configuration, mais nécessite une expertise technique approfondie. L'**installation par gestionnaires de paquets** (APT, YUM, ou autres) simplifie le processus et assure une meilleure intégration avec l'écosystème système, au prix d'une certaine standardisation. La **containerisation** via Docker permet un déploiement rapide avec des configurations pré-établies et une isolation des environnements. Cependant, il convient de noter un point d'attention critique concernant la containerisation des bases de données. Bien que cette approche facilite grandement le déploiement et le test, elle présente des défis spécifiques dans les environnements de production, notamment dans les plateformes d'orchestration comme Kubernetes ou Docker Swarm. Les problématiques de **stockage persistant** et les questions de **performance** dues aux couches d'abstraction supplémentaires rendent souvent préférable un déploiement sur infrastructure dédiée pour les systèmes critiques.

Sécurité et gestion des accès : responsabilités critiques

Le DBA détient des privilèges administrateur complets sur les systèmes de bases de données, ce qui implique des responsabilités particulièrement lourdes. Il doit mettre en place une **gestion granulaire des droits d'accès** par utilisateur et par objet, assumant une **responsabilité de confidentialité** absolue sur les données sensibles. Dans le contexte réglementaire actuel, notamment avec l'entrée en vigueur du RGPD, le DBA peut endosser le rôle de Data Protection Officer ou du moins jouer un rôle central dans la conformité réglementaire. Il doit implémenter les politiques de sécurité incluant chiffrement, audit et traçabilité, tout en maintenant l'équilibre délicat entre sécurité et performance.

Haute disponibilité : enjeux critiques

La garantie de disponibilité constitue l'un des défis majeurs du métier de DBA, particulièrement dans les environnements industriels critiques. L'exemple souvent cité de l'aciérie illustre parfaitement ces enjeux : lorsqu'une poche de métal en fusion est en cours de traitement, un arrêt de la base de données contrôlant le processus peut avoir des conséquences dramatiques, allant de la perte économique massive aux risques pour la sécurité des personnes. Cette criticité impose la mise en place de systèmes **résilients** avec des mécanismes sophistiqués : sauvegardes à chaud permettant la sauvegarde sans interruption de service, réplication en temps réel pour assurer la redondance, plans de reprise d'activité (PRA) et de continuité d'activité (PCA) testés régulièrement, et monitoring proactif pour la détection précoce des incidents. Dans de tels contextes, les DBA sont souvent soumis à des astreintes 24h/7j.

Optimisation et tuning : l'art de la performance

L'optimisation constitue l'aspect le plus technique et le plus gratifiant du métier de DBA. Elle englobe plusieurs dimensions complémentaires : l'optimisation des requêtes par l'analyse des plans d'exécution et l'indexation intelligente, le tuning infrastructure incluant la configuration des I/O, de la mémoire et des processeurs, la gestion sophistiquée du stockage avec l'optimisation des systèmes RAID et SAN, et la conception d'architectures haute performance utilisant clustering et partitioning. Cette expertise technique place le DBA au cœur de la performance globale du système d'information, avec un impact direct sur l'expérience utilisateur et l'efficacité opérationnelle de l'organisation.

Architecture interne des bases de données : le modèle Oracle

Séparation logique et physique : un principe fondamental

Pour comprendre les mécanismes internes des bases de données modernes, le modèle architectural d'Oracle offre un exemple particulièrement éclairant de la séparation entre abstraction logique et réalité physique. Cette séparation constitue l'un des piliers conceptuels qui permettent aux SGBD d'offrir flexibilité et performance.

Structure logique hiérarchique

La structure logique s'organise selon une hiérarchie claire et cohérente : Au niveau le plus élevé, la **base de données** représente l'entité logique globale, regroupant l'ensemble des informations d'une application ou d'une organisation. Cette base se subdivise en **tablespaces**, espaces de stockage logiques qui regroupent des objets selon leur fonction ou leur criticité. Chaque tablespace

contient des **segments**, représentations logiques d'objets spécifiques comme les tables ou les index. Ces segments sont eux-mêmes composés d'**extents**, ensembles contigus de blocs alloués dynamiquement selon les besoins. Enfin, au niveau le plus fin, les **data blocks** constituent la plus petite unité de stockage logique, généralement de quelques kilo-octets. Cette organisation hiérarchique permet une gestion granulaire et optimisée des ressources, où chaque niveau peut être configuré et optimisé indépendamment selon les besoins spécifiques.

Structure physique correspondante

La structure physique mappe cette organisation logique sur les supports de stockage réels. Les **datafiles** constituent les fichiers physiques stockés sur le système de fichiers ou directement sur les dispositifs de stockage. Le mapping entre tablespaces logiques et datafiles physiques offre une flexibilité considérable pour l'optimisation des performances et la gestion de l'espace. Pour les environnements à très hautes exigences de performance, Oracle propose également l'accès aux **raw devices**, permettant un accès direct aux blocs physiques sans passer par le système de fichiers. Cette approche, bien que complexe à gérer, peut apporter des gains de performance significatifs dans des contextes spécifiques.

Stratégies d'optimisation avancées

Séparation fonctionnelle des données

L'architecture en tablespaces permet une optimisation fine en fonction des caractéristiques d'usage des données. Un **tablespace Index** peut être configuré sur des supports rapides comme les SSD pour optimiser les accès aux index, tandis que le **tablespace Data** contenant les données principales peut résider sur un stockage standard offrant un bon compromis performance/coût. Pour les données historiques peu consultées, un **tablespace Archive** peut utiliser un stockage lent mais très capacitif.

Stratégie Hot/Warm/Cold

Cette approche classifie les données selon leur fréquence d'accès et adapte la stratégie de stockage en conséquence. Les données **Hot**, fréquemment accédées, bénéficient d'un stockage haute performance. Les données **Warm**, occasionnellement utilisées, sont placées sur un stockage standard. Les données **Cold**, rarement consultées mais devant rester accessibles, sont archivées sur un stockage économique haute capacité.

Raw Devices : optimisation extrême

Pour les environnements à exigences de performance exceptionnelles, l'utilisation de raw devices permet d'éliminer les couches d'abstraction du système de fichiers et d'accéder directement aux blocs physiques. Cette approche trouve sa justification dans des contextes très spécifiques : trading haute fréquence où quelques millisecondes déterminent la rentabilité, calcul scientifique

intensif traitant des téraoctets de données (CERN, simulations climatiques), ou ERP critiques gérant des processus industriels continus. Les gains de performance peuvent être spectaculaires, mais la complexité de gestion s'accroît considérablement, nécessitant une expertise très pointue.

Gestion des transactions et garantie de cohérence

Architecture des logs de transaction : universalité du concept

Tous les SGBD modernes implémentent un système de journalisation sophistiqué pour garantir la cohérence des données, bien que les terminologies varient selon les éditeurs. PostgreSQL utilise le **WAL** (Write Ahead Log), MySQL s'appuie sur le **Binary Log** (bin log), Oracle maintient des **Transaction Logs** et **Redo Logs**, tandis que SQL Server gère ses propres **Transaction Logs**.

Mécanisme de fonctionnement

Le principe de fonctionnement repose sur l'**écriture préalable** : toute modification est d'abord écrite dans le journal avant d'être appliquée aux données. L'**écriture différée** permet ensuite d'écrire les données en mémoire puis sur disque de manière asynchrone, optimisant ainsi les performances. En cas d'incident, le processus de **recovery** utilise le journal pour reconstruire un état cohérent. Cette architecture garantit les propriétés **ACID** (Atomicité, Cohérence, Isolation, Durabilité) fondamentales aux bases de données transactionnelles, permettant de maintenir l'intégrité des données même en cas de panne système.

Processus de récupération

En cas de panne système, le processus de recovery suit une séquence précise et rigoureuse. L'**analyse du log** identifie d'abord les transactions incomplètes et détermine l'état de chaque transaction au moment de la panne. La phase de **Redo** rejoue ensuite les transactions validées mais non encore écrites sur disque, garantissant que toutes les modifications confirmées sont bien persistées. La phase d'**Undo** annule les transactions non validées, restaurant la cohérence. Enfin, une **vérification de cohérence** contrôle l'intégrité globale des données.

Applications avancées : réplication et architectures modernes

Le journal de transaction trouve des applications particulièrement élégantes dans la **réplication** entre serveurs. Dans une architecture Master-Slave, le serveur maître n'envoie que son journal de transaction, que le serveur esclave applique à son propre datafile. Cette approche est infiniment plus efficace que la réplication complète des fichiers, car seules les modifications effectives sont transmises, les opérations de lecture et les transactions échouées n'étant pas incluses. Cette séparation entre données et journal ouvre également la voie à des architectures avancées comme **CQRS** (Command Query Responsibility Segregation), où les opérations de lecture et d'écriture sont physiquement séparées pour optimiser les performances.

Configuration et optimisation des SGBD

Paramètres réseau et connectivité

Configuration de l'adresse de liaison

Le paramètre **bind address** détermine les interfaces réseau sur lesquelles le SGBD accepte les connexions. Une configuration sur **localhost** limite l'accès à la machine locale, offrant une sécurité maximale mais une flexibilité réduite. Une configuration sur **0.0.0.0** permet l'accès depuis n'importe quelle adresse IP, offrant une flexibilité maximale mais nécessitant une vigilance sécuritaire accrue. Le choix dépend fondamentalement de l'architecture : développement local versus accès multi-machines.

Dimensionnement des connexions : coordination critique

La configuration du nombre maximum de connexions nécessite une **coordination étroite** entre administrateurs de bases de données et équipes applicatives. Les problématiques courantes incluent le sous-dimensionnement applicatif (application configurée pour 10 connexions maximum, base acceptant 200 connexions) créant un goulot d'étranglement au niveau applicatif, ou le sous-dimensionnement base (application configurée pour 2000 connexions, base limitée à 200) provoquant des erreurs de connexion. Les bonnes pratiques recommandent d'inclure les connexions de service (backup, maintenance) dans le dimensionnement, de surveiller les métriques de pool de connexions, et d'ajuster dynamiquement selon la charge observée.

Optimisation mémoire : shared buffers et stratégies de cache

Les **shared buffers** constituent le cache principal du SGBD, où les blocs de données fréquemment accédés restent en mémoire pour réduire drastiquement les accès disque. L'amélioration des performances est directement proportionnelle à la taille du cache, dans la limite des ressources disponibles. La configuration peut adopter une approche de **pool unique** pour la simplicité de gestion avec allocation globale, ou de **pools multiples** permettant une optimisation fine par type d'opération.

Espaces de travail pour jointures et stratégies d'optimisation

La mémoire de travail dédiée aux jointures influence directement les **stratégies d'optimisation** choisies par le SGBD. Deux approches principales existent : Les **Nested Loops** (boucles imbriquées) suivent une logique simple de double boucle, avec une complexité O(n×m), une faible consommation mémoire, mais des performances dégradées sur gros volumes. Les **Hash Join** créent des tables de hachage en mémoire, permettent une parallélisation sur plusieurs cœurs, offrent des performances supérieures mais avec une consommation mémoire importante. L'optimiseur choisit automatiquement la stratégie en fonction des ressources disponibles et des caractéristiques des données.

Optimisation des requêtes : l'art de la performance

L'optimiseur automatique : intelligence du SGBD

L'**optimiseur de requêtes** constitue le cerveau du SGBD, responsable de transformer une requête SQL en plan d'exécution optimal. Ses critères d'optimisation incluent la taille des tables impliquées, les statistiques de distribution des données, les index disponibles, et les ressources système disponibles.

Intervention manuelle : hints et optimisation ciblée

Lorsque l'optimiseur automatique produit des plans sous-optimaux, les **hints** permettent d'orienter ses décisions. Ces indications peuvent porter sur l'ordre de traitement des tables, la sélection d'index spécifiques, ou le choix de stratégie de jointure. Le processus d'optimisation suit généralement ces étapes : analyse du plan d'exécution initial, identification des goulots d'étranglement, application de hints ciblés, et comparaison des métriques de performance.

Outils d'analyse avancés

Les SGBD modernes intègrent des **advisors automatiques** qui détectent proactivement les requêtes problématiques, proposent des suggestions d'optimisation chiffrées, et offrent des interfaces graphiques pour l'aide à la décision. Cette sophistication justifie économiquement l'investissement dans des solutions commerciales pour les charges complexes impliquant des requêtes de 60-80 lignes sur des centaines de tables.

Gestion des utilisateurs et contrôle d'accès

Architecture hiérarchique des permissions

Fondements conceptuels

La gestion des droits dans les systèmes de bases de données s'organise selon une hiérarchie logique qui répond à des impératifs pratiques de simplification administrative. Au niveau le plus basique, les **droits unitaires** constituent les autorisations élémentaires d'effectuer une action spécifique sur un objet particulier. Bien que cette granularité fine offre un contrôle précis, elle devient rapidement ingérable dans des environnements complexes. Les **rôles** regroupent des ensembles cohérents de droits selon une logique fonctionnelle ou métier. Un rôle "bibliothécaire" pourrait inclure des droits en lecture sur toutes les tables de livres, des droits en écriture sur la table des emprunts, et des droits spécifiques sur la gestion des adhérents. Les **groupes d'utilisateurs** permettent une gestion collective, où les rôles peuvent être assignés à des groupes, bénéficiant automatiquement à tous leurs membres.

Implémentation technique

La syntaxe SQL standardisée suit des principes similaires dans tous les SGBD. La création d'entités utilise les commandes `CREATE USER` et `CREATE ROLE`. L'attribution des droits s'effectue avec la commande `GRANT` qui associe systématiquement privilège, objet et bénéficiaire. La fonctionnalité `WITH GRANT OPTION` permet la délégation de l'attribution des droits, particulièrement utile pour la gestion décentralisée. La commande `REVOKE` permet l'annulation des privilèges précédemment accordés.

Intégration avec les systèmes d'authentification d'entreprise

Modes d'authentification multiples

Les bases de données d'entreprise supportent généralement plusieurs modes d'authentification. L'**authentification native** utilise les mécanismes propres au SGBD, offrant un contrôle total mais nécessitant une administration dédiée. L'**intégration Active Directory** permet de centraliser la gestion des identités, avec les comptes définis dans AD et l'authentification déléguée au contrôleur de domaine. L'**authentification intégrée Windows/Kerberos** représente le niveau d'intégration le plus poussé, offrant une expérience Single Sign-On transparente mais avec des défis techniques importants concernant la propagation des tokens et la gestion des erreurs.

Considérations économiques : impact des licences

Les modèles de licencing influencent significativement les choix d'architecture. Le **licencing par utilisateur nommé** impose un coût pour chaque utilisateur accédant à la base, tandis que le **licencing par cœur de processeur** facture selon la puissance du serveur, indépendamment du nombre d'utilisateurs. Face aux coûts élevés des licences propriétaires, les entreprises adoptent souvent des stratégies de **comptes de service applicatifs**, où une application utilise un compte unique et gère elle-même les autorisations par utilisateur, réduisant drastiquement le nombre de licences nécessaires.

Stratégies de sauvegarde et de restauration

Principe du moindre privilège : prévention fondamentale

Le **principe du moindre privilège** constitue un pilier fondamental de la sécurité informatique. Il stipule qu'un utilisateur ne doit disposer que des privilèges strictement nécessaires à l'accomplissement de ses fonctions légitimes. Dans le contexte des bases de données, cela signifie éviter de se connecter systématiquement avec des comptes administrateurs pour des tâches courantes. L'exemple classique d'une commande Linux illustre l'importance de ce principe : la différence entre `rm -rf toto/*` et `rm -rf toto/ *` (un simple espace) peut avoir des conséquences dramatiques si exécutée avec des privilèges root. Les statistiques montrent que près de la moitié des incidents en production résultent de commandes exécutées avec des privilèges excessifs.

Types de sauvegarde selon la disponibilité

Sauvegarde à froid vs. sauvegarde à chaud

La **sauvegarde à froid** nécessite l'arrêt complet du système, garantissant une cohérence parfaite mais avec une interruption de service. Elle convient aux entreprises disposant de fenêtres de maintenance régulières. La **sauvegarde à chaud** maintient la disponibilité du service mais introduit un risque de perte de données ou d'incohérence. Pour les grandes plateformes comme Netflix, la sauvegarde est un processus continu créant un cycle perpétuel de protection des données.

Stratégies techniques pour la sauvegarde à chaud

Le **snapshot** capture l'état des données à un instant T, offrant une cohérence parfaite si la capture s'effectue entre deux transactions. La **sauvegarde par réplication** utilise une architecture maître-esclave où la réplication est interrompue sur l'esclave pour effectuer la sauvegarde, puis reprend pour rattraper les modifications manquantes.

Approches de sauvegarde : logique vs. physique

Sauvegarde logique (dump)

La sauvegarde logique exporte les données sous forme de requêtes SQL reconstituables. Elle offre une flexibilité maximale, une lisibilité du format, une portabilité entre versions et plateformes, et une granularité permettant la restauration sélective. Cependant, elle présente des limitations en termes de performance et de volumétrie.

Sauvegarde physique

La sauvegarde physique implique la copie directe des fichiers de données et des journaux de transactions. Elle nécessite un **mode backup** pour garantir la cohérence, figeant les datafiles en lecture seule pendant la copie et appliquant ensuite les logs pour la mise à jour.

Sauvegarde par snapshot

Le snapshot utilise les capacités de l'infrastructure sous-jacente pour créer une image instantanée au niveau du système de fichiers. Cette approche offre un impact minimal sur la base en fonctionnement, une rapidité de création quasi-instantanée, et pas d'interruption de service.

Stratégies de sauvegarde selon le contenu

Types de sauvegarde

La **sauvegarde complète** capture l'intégralité des données, constituant la référence pour toute stratégie. La **sauvegarde différentielle** capture tous les changements depuis la dernière sauvegarde complète, facilitant la restauration mais avec une taille croissante. La **sauvegarde incrémentale** capture uniquement les changements depuis la dernière sauvegarde, réduisant la taille mais complexifiant la restauration.

Point-in-Time Recovery (PITR)

Le PITR permet de restaurer une base de données à un moment précis en combinant une sauvegarde physique et l'application des journaux de transactions jusqu'au moment souhaité. Cette technique s'avère particulièrement utile pour les investigations forensiques et la récupération après incidents.

Métriques et objectifs

Indicateurs de performance

Le **RPO** (Recovery Point Objective) définit la perte de données maximale acceptable, influençant directement la fréquence des sauvegardes. Le **RTO** (Recovery Time Objective) spécifie le temps maximal acceptable pour restaurer le service, déterminant le choix de la stratégie de restauration.

Politiques de rétention

La politique de rétention définit la durée de conservation des sauvegardes et doit prendre en compte les besoins métier, les contraintes légales (RGPD, enquêtes fiscales), et les coûts de stockage. Elle nécessite une collaboration étroite entre équipes techniques et métier.

Optimisation des données historisées : le processus de freeze

Principe fondamental

L'optimisation des données historisées représente un enjeu crucial dans la gestion moderne des bases de données. Le processus de **freeze** (gel) constitue une technique d'optimisation particulièrement efficace pour traiter les données anciennes qui ne subissent plus de modifications.

Mécanisme de transformation

Le freeze repose sur la **transformation de données mutables en données immuables**. Lorsque des données atteignent un certain âge ou que leur période de modification active se termine, elles sont marquées comme étant en **lecture seule**. Cette transformation entraîne plusieurs optimisations automatiques : élimination des logs de transaction puisqu'aucune modification future n'est attendue, optimisation du cache avec mise en cache agressive des données immuables, et restructuration physique pour optimiser les accès en lecture.

Applications pratiques et bénéfices

Dans le secteur bancaire, les transactions de plus d'un exercice fiscal peuvent être gelées, permettant une consultation rapide de l'historique sans impacter les performances des opérations courantes. Les logs d'activité système, une fois consolidés, bénéficient grandement de cette optimisation. Cette technique s'inscrit dans une approche de **séparation des charges OLTP et OLAP**, où les données gelées se rapprochent naturellement du modèle OLAP optimisé pour l'analyse et la consultation.

Évolutions et perspectives

Impact du cloud et de l'intelligence artificielle

Les environnements cloud modernes permettent d'optimiser cette approche en déplaçant automatiquement les données gelées vers des supports de stockage moins coûteux mais toujours accessibles. L'intelligence artificielle peut aider à déterminer automatiquement le moment optimal pour geler des données, en analysant les patterns d'accès et les contraintes métier.

Défis et limitations

La gestion de la transition entre états mutable et gelé doit être soigneusement orchestrée. Il faut prévoir des procédures pour les cas exceptionnels où des données gelées doivent être modifiées, impliquant généralement un "dégel" temporaire contrôlé. L'état des données doit être tracé et auditable, particulièrement dans des contextes réglementés.

Schéma récapitulatif

https://archive.qoyri.fr/mindmaps/cours06.11.25.2.html

Concepts architecturaux principaux

Architecture des systèmes d'information :

- Systèmes centraux (Core Banking, ERP) et bases périphériques
- Approche distribuée et spécialisée
- Séparation fonctionnelle des données **Architecture interne des SGBD :**
- Structure logique : Base → Tablespaces → Segments → Extents → Blocks
- Structure physique : Datafiles et Raw devices
- Stratégies d'optimisation Hot/Warm/Cold

Rôles et responsabilités

Database Administrator (DBA) :

- Installation et configuration (native, paquets, conteneurs)
- Sécurité et conformité (RGPD, accès granulaire, audit)
- Haute disponibilité (sauvegardes, réplication, astreintes)
- Optimisation (requêtes, infrastructure, stockage) **Gestion des utilisateurs :**
- Hiérarchie Droits → Rôles → Groupes
- Commandes GRANT/REVOKE avec WITH GRANT OPTION
- Intégration Active Directory et authentification Kerberos

Stratégies de sauvegarde et performance

Types de sauvegarde :

- Par disponibilité : Cold backup vs Hot backup
- Par contenu : Complète, Différentielle, Incrémentale
- Par méthode : Logique (dump), Physique, Snapshot **Métriques et objectifs :**
- RPO (Recovery Point Objective) : perte de données acceptable
- RTO (Recovery Time Objective) : temps de restauration maximal
- PDMA (Perte de Données Maximale Admissible)

Technologies et optimisation

Gestion transactionnelle :

- Logs de transaction (WAL, bin log, transaction logs)
- Mécanismes de recovery (Redo/Undo)
- Propriétés ACID garanties **Optimisation avancée :**
- Freeze des données historisées
- Séparation OLTP/OLAP
- Point-in-Time Recovery (PITR)

Mots-clés essentiels

`DBA`, `ERP`, `Core Banking`, `Tablespace`, `Datafiles`, `Raw devices`, `WAL`, `RGPD`, `Hot/Warm/Cold storage`, `ACID`, `Recovery`, `High Availability`, `Shared Buffers`, `Connection Pool`, `Nested Loops`, `Hash Join`, `Hints`, `Optimiseur`, `Rollback`, `Commit`, `RBAC`, `Information Schema`, `PITR`, `RPO`, `RTO`, `Freeze`, `OLTP/OLAP`, `Snapshot`, `Réplication`, `Principe du moindre privilège

21-05 cours : Chapitre 5 : Optimisation, Maintenance et Architectures Avancées des Bases de Données

Introduction : Au-delà de l'Écriture des Requêtes

Après avoir maîtrisé la syntaxe du langage SQL, la modélisation des données (MCD, MLD, MPD) et les principes fondamentaux des systèmes de gestion de bases de données (SGBD), une nouvelle étape cruciale se présente pour tout professionnel des données : l'optimisation des performances. Une requête fonctionnelle n'est pas nécessairement une requête performante. Dans des environnements de production, où les volumes de données peuvent atteindre des milliards d'enregistrements, une requête mal optimisée peut saturer les ressources du serveur (CPU, RAM, I/O disque) et dégrader l'expérience utilisateur. Ce chapitre couvre un large spectre de l'ingénierie des bases de données, des mécanismes internes d'optimisation des SGBD relationnels à la maintenance proactive, en passant par les architectures distribuées des systèmes NoSQL. Nous explorerons comment analyser et améliorer les performances des requêtes, comment maintenir une base de données saine et réactive, et comment concevoir des systèmes capables de gérer des volumes

de données massifs tout en garantissant haute disponibilité et scalabilité.

Partie I : Optimisation des Requêtes dans les SGBD Relationnels

1. Le Cœur de la Performance : L'Optimiseur de Requêtes

Lorsqu'un SGBD reçoit une requête SQL, il ne l'exécute pas aveuglément. Entre la réception de la chaîne de caractères `SELECT ...` et l'affichage du résultat, un composant logiciel sophistiqué entre en jeu : l'**optimiseur de requêtes** (*Query Optimizer*). Son rôle est d'analyser la requête et de déterminer le "chemin" le plus efficace pour accéder aux données et les traiter. Ce chemin est appelé **plan d'exécution** (*execution plan*).

1.1. Le Processus d'Exécution d'une Requête

1. **Analyse Syntaxique (*Parsing*)** : Le moteur vérifie que la syntaxe de la requête est correcte.
2. **Analyse Sémantique** : Il vérifie que les tables et colonnes existent et que l'utilisateur a les droits nécessaires.
3. **Génération du Plan d'Exécution** : C'est ici qu'intervient l'optimiseur. Pour une même requête, il existe de multiples manières de l'exécuter (ordre des jointures, utilisation d'index, etc.). L'optimiseur évalue plusieurs stratégies possibles, estime leur "coût" et choisit celle qu'il juge la plus rapide.
4. **Exécution** : Le moteur exécute la requête en suivant pas à pas le plan d'exécution choisi.

1.2. Les Critères de Décision de l'Optimiseur

L'optimiseur base ses décisions sur une multitude de facteurs pour estimer le coût de chaque plan. Ce coût est une métrique abstraite représentant principalement le temps et les ressources (surtout les I/O disque).

- **Présence d'Index** : Un index permet un accès direct à une ligne sans scanner toute la table (*Full Table Scan*). C'est souvent le facteur le plus important.

- **Statistiques et Cardinalité** : Le SGBD maintient des statistiques sur les données : nombre de lignes par table (*cardinalité*), distribution des valeurs, etc. Ces informations sont cruciales. Par exemple, si une requête filtre sur une valeur très rare (haute sélectivité), l'utilisation d'un index est très efficace. Si elle doit retourner 90% de la table (faible sélectivité), un scan complet peut s'avérer plus rapide.
- **Gestion de la Mémoire (Cache)** : L'optimiseur tient compte des données déjà présentes en RAM (*buffer cache*). Il évalue également si les opérations intermédiaires (comme le résultat d'une jointure) peuvent tenir en mémoire ou si elles nécessiteront un stockage temporaire sur disque, une opération très pénalisante.

> **Analogie : Planifier un trajet en ville** Pensez à l'optimiseur comme à une application GPS. Pour aller d'un point A à un point B, il existe plusieurs itinéraires. Le GPS (l'optimiseur) évalue les options en fonction de divers critères : distance, trafic en temps réel (statistiques), péages (coût d'une opération), type de route (index vs scan complet). Il ne choisit pas le chemin le plus court en kilomètres, mais le plus rapide en temps. Le plan d'exécution est l'itinéraire détaillé qu'il vous propose.

2. Dévoiler la Stratégie : La Commande `EXPLAIN`

La commande `EXPLAIN` est l'outil de diagnostic indispensable pour visualiser le plan d'exécution choisi par l'optimiseur. En préfixant une requête par `EXPLAIN`, on demande au SGBD non pas de l'exécuter, mais de nous retourner sa stratégie.

2.1. `EXPLAIN` vs `EXPLAIN ANALYZE` : Théorie vs Pratique

- **`EXPLAIN` (Le Plan Théorique)**
  - **Fonctionnement** : Le SGBD génère le plan d'exécution mais **n'exécute pas** la requête. Il se base sur ses statistiques internes pour *estimer* le coût et le nombre de lignes traitées.
  - **Avantage** : C'est **instantané**. Idéal pour une première analyse rapide, surtout sur des requêtes très lentes ou en production.
  - **Cas d'usage** : Premier réflexe pour déceler des problèmes évidents (ex: un scan de table non désiré).
- **`EXPLAIN ANALYZE` (Le Plan Exécuté et Mesuré)**
  - **Fonctionnement** : Le SGBD génère le plan, **exécute réellement la requête**, mesure les temps et les ressources consommés, puis retourne le plan enrichi de ces **valeurs réelles**.
  - **Avantage** : Fournit des informations beaucoup plus précises. Il compare les estimations (*cost*) avec la réalité (*actual time*). Un grand écart entre les deux peut indiquer que les statistiques de la base sont obsolètes.

- **Inconvénient** : Le temps d'exécution est celui de la requête elle-même. À utiliser avec prudence sur des requêtes modifiant des données.
- **Cas d'usage** : Indispensable pour une optimisation fine et pour valider l'efficacité d'une modification (ex: ajout d'un index).

2.2. Anatomie d'un Plan d'Exécution : Opérations Clés

L'analyse d'un plan passe par la compréhension de ses opérations.

- **Méthodes d'Accès aux Données** :
  - `Sequential Scan` **(ou** `Full Table Scan`**)** : Lecture de la table entière. Efficace sur de petites tables ou pour des requêtes peu sélectives.
  - `Index Scan` : Utilisation d'un index pour localiser rapidement les lignes. L'objectif principal de l'optimisation pour les requêtes sélectives.
  - `Index-Only Scan` : Si toutes les colonnes demandées sont dans l'index, le moteur n'accède même pas à la table, ce qui est extrêmement rapide.
- **Méthodes de Jointure** :
  - `Nested Loop Join` **(Boucles Imbriquées)** : Pour chaque ligne de la table A, parcourt toute la table B. Efficace si l'une des tables est très petite.
  - `Hash Join` : Construit une table de hachage en mémoire avec la plus petite table, puis la sonde avec la plus grande. Très rapide pour les grands volumes, mais gourmand en mémoire.
  - `Merge Join` **(Jointure par Fusion)** : Requiert que les deux tables soient triées sur la clé de jointure. Le moteur parcourt alors les deux tables en parallèle. Très efficace si les données sont déjà triées (via un index par exemple).

3. L'Art de l'Indexation : Le Levier Principal de l'Optimisation

Un index est une structure de données séparée, conçue pour accélérer la recherche de lignes. Il contient les valeurs d'une ou plusieurs colonnes et un pointeur vers l'emplacement physique de la ligne. **Avantages :** Accélération massive des lectures (`SELECT`). **Inconvénients :** Ralentissement des écritures (`INSERT`, `UPDATE`, `DELETE`) car l'index doit aussi être mis à jour, et consommation d'espace disque.

3.1. Quelles Colonnes Indexer ?

- **Clés primaires (`PRIMARY KEY`)** : Automatiquement indexées.
- **Clés étrangères (`FOREIGN KEY`)** : **Essentiel**. Accélère drastiquement les jointures.
- **Colonnes des clauses `WHERE`** : Surtout si elles sont très sélectives (ex: `email`, `numero_commande`).

- **Colonnes des clauses `ORDER BY` et `GROUP BY`** : Un index peut fournir les données déjà triées, évitant une opération de tri coûteuse.

3.2. Les Différents Types d'Index

- **B-Tree (Arbre B)** : Le type par défaut, le plus polyvalent. Efficace pour les égalités (`=`), les comparaisons (`<`, `>`, `BETWEEN`) et les préfixes (`LIKE 'prefixe%'`).
- **Hash** : Extrêmement rapide, mais uniquement pour les recherches d'égalité stricte (`=`).
- **Index Spécialisés** :
  - **GIN/GiST (PostgreSQL)** : Pour les types de données complexes comme les `JSONB` (recherche dans un document) ou les données géospatiales (PostGIS).
  - **Vectoriels (ex: HNSW)** : Pour la recherche par similarité sur des vecteurs numériques (*embeddings*), au cœur des applications d'IA comme la recherche sémantique et les systèmes RAG (Retrieval-Augmented Generation).

Partie II : Maintenance et Gestion de la Concurrence

Une base de données est un système vivant qui nécessite un entretien régulier pour maintenir ses performances.

1. Maintenance Proactive : Statistiques et Fragmentation

1.1. Mise à Jour des Statistiques

Comme vu précédemment, l'optimiseur s'appuie sur les **statistiques**. Après des modifications massives de données (`INSERT`, `DELETE`), ces statistiques deviennent **obsolètes**, conduisant l'optimiseur à choisir de mauvais plans d'exécution.

- **Diagnostic** : Un grand écart entre les estimations de `EXPLAIN` et les mesures réelles de `EXPLAIN ANALYZE`.
- **Solution** : Exécuter périodiquement la commande `ANALYZE nom_de_la_table;` (PostgreSQL) pour recalculer les statistiques. C'est une opération lourde à planifier durant les périodes de faible charge.

1.2. Gestion de la Fragmentation (`VACUUM`)

Les opérations `DELETE` et `UPDATE` créent des "trous" dans les fichiers de données, un phénomène appelé **fragmentation**. Cela ralentit les lectures, qui doivent effectuer des sauts sur le disque (I/O aléatoires) au lieu d'une lecture séquentielle efficace.

- **`VACUUM`** : Marque l'espace libéré comme réutilisable pour de futurs `INSERT`.
- **`VACUUM FULL`** : Opération plus agressive qui réécrit complètement la table pour éliminer les trous et tasser les données. Elle est **extrêmement lente et bloquante** (pose un verrou exclusif sur la table) et doit être utilisée avec une extrême précaution lors de fenêtres de maintenance planifiées. La plupart des SGBD modernes intègrent un processus `autovacuum` qui tente de gérer ces tâches automatiquement, mais sa configuration doit être finement ajustée dans les environnements critiques.

2. Manipulation des Données et Gestion de la Concurrence

2.1. Commandes `UPDATE` et `DELETE`

Ces commandes modifient ou suppriment des lignes. Leur puissance réside dans la clause `WHERE`, qui cible les lignes à affecter. **L'omission de la clause `WHERE` est catastrophique**, car l'opération s'appliquera à toute la table.

2.2. Transactions et Propriétés ACID

Une **transaction** est un bloc d'opérations traité de manière atomique. Elle garantit les propriétés **ACID** :

- **Atomicité** : Tout ou rien.
- **Cohérence** : La base reste dans un état valide.
- **Isolation** : Les transactions concurrentes n'interfèrent pas entre elles.
- **Durabilité** : Une fois validée (`COMMIT`), la modification est permanente.

2.3. Verrouillage (Locking) et Interblocages (Deadlocks)

Pour garantir l'isolation, les SGBD utilisent des **verrous** (*locks*). Une transaction qui modifie une ligne pose un **verrou exclusif**, empêchant les autres de la lire ou de la modifier.

- **`SELECT ... FOR UPDATE`** : Permet à un développeur de poser explicitement un verrou exclusif sur les lignes lues, en prévision d'une mise à jour. Cela évite les conflits où deux transactions liraient la même valeur, puis essaieraient de la mettre à jour séquentiellement, la seconde écrasant le travail de la première.
- **Interblocage (Deadlock)** : Situation où deux transactions s'attendent mutuellement.

    - Transaction A verrouille la Ligne 1 et attend la Ligne 2.
    - Transaction B verrouille la Ligne 2 et attend la Ligne 1. Le SGBD détecte ce cycle et "tue" l'une des transactions (la moins coûteuse à annuler), qui reçoit une erreur et doit être réessayée par l'application.

3. Logique Applicative dans la Base de Données

Les SGBD permettent d'embarquer du code via des langages procéduraux (PL/SQL pour Oracle, T-SQL pour SQL Server, PL/pgSQL pour PostgreSQL).

- **Procédures Stockées** : Blocs de code exécutant une série d'actions (`EXEC ma_procedure(...)`). Utiles pour encapsuler une logique métier complexe, réduisant les allers-retours réseau.
- **Fonctions** : Similaires aux procédures, mais **retournent une valeur unique**. Elles peuvent être utilisées directement dans une requête `SELECT`.
- **Triggers (Déclencheurs)** : Procédures spéciales exécutées **automatiquement** en réponse à un événement (`INSERT`, `UPDATE`, `DELETE`) sur une table. Utiles pour l'audit, la validation de données complexes ou la maintenance de la dénormalisation. Ils doivent être utilisés avec parcimonie car leur logique "cachée" peut rendre le débogage difficile et impacter les performances.

Partie III : Architectures NoSQL pour la Haute Disponibilité et la Scalabilité

L'ère du Big Data a popularisé les bases de données NoSQL, conçues nativement pour la distribution et la scalabilité horizontale. La question centrale devient : "Comment architecturer mon cluster pour que les données soient fiables, accessibles et performantes ?"

1. Le Cadre Théorique : Le Théorème CAP

Le **théorème CAP** stipule qu'un système distribué ne peut garantir simultanément que deux des trois propriétés suivantes :

1. **Cohérence (Consistency - C)** : Tous les nœuds voient la même donnée au même moment.
2. **Disponibilité (Availability - A)** : Chaque requête reçoit une réponse.
3. **Tolérance à la Partition (Partition Tolerance - P)** : Le système fonctionne malgré les pannes réseau. La tolérance à la partition (P) étant une nécessité, le choix se fait entre C et A. La plupart des systèmes NoSQL sont des systèmes **AP**, privilégiant la disponibilité et adoptant un modèle de **cohérence éventuelle (*Eventual Consistency*)**.

2. La Réplication : Garantir la Haute Disponibilité

La réplication consiste à copier les données sur plusieurs serveurs (nœuds). Un ensemble de serveurs contenant les mêmes copies est un **Replica Set**.

- **Objectif** : Assurer la continuité de service en cas de panne d'un serveur (**failover**).
- **Modèle Primaire-Secondaire (MongoDB)** : Un seul nœud **Primaire** accepte les écritures. Les nœuds **Secondaires** répliquent ces écritures et peuvent servir les lectures.
- **Élection et Quorum** : Pour éviter le problème du **"split-brain"** (deux primaires divergents après une coupure réseau), une décision (comme l'élection d'un nouveau primaire) requiert un **quorum**, c'est-à-dire une majorité des nœuds (`(N/2) + 1`). C'est pourquoi les Replica Sets ont presque toujours un nombre **impair** de nœuds (3, 5, ...).

3. Le Sharding : Assurer la Scalabilité Horizontale

Le sharding (ou partitionnement horizontal) consiste à répartir les données sur plusieurs serveurs (ou **shards**). Chaque shard ne stocke qu'un sous-ensemble des données.

- **Objectif** : Gérer des volumes de données qui dépassent la capacité d'un seul serveur et distribuer la charge de travail.
- **Architecture (MongoDB)** :
  1. **Shards** : Stockent les données. Pour la robustesse, **chaque shard est lui-même un Replica Set**.
  2. **Serveurs de Configuration** : Stockent les métadonnées (la "carte" indiquant quelle donnée est sur quel shard).
  3. **Routeurs de Requêtes (`mongos`)** : Points d'entrée pour les applications, qui routent les requêtes vers les bons shards.

- **Clé de Sharding (Shard Key)** : C'est le champ qui détermine sur quel shard un document sera stocké. Le choix de cette clé est **la décision la plus critique** dans la conception d'un cluster shardé. Une bonne clé assure une distribution uniforme des données et de la charge.

4. Modélisation en NoSQL Orienté Document

Dans une base comme MongoDB, la modélisation tourne autour de deux stratégies :

- **Intégration (Embedding)** : Imbriquer des données liées dans un seul document (dénormalisation).
  - **Avantages** : Très performant en lecture (une seule requête), atomicité des écritures sur le document.
  - **Inconvénients** : Limite de taille des documents (16 Mo pour MongoDB), problèmes de concurrence à l'écriture.
- **Liaison (Linking)** : Stocker des références (IDs) vers des documents dans d'autres collections (normalisation).
  - **Avantages** : Pas de limite de taille, flexibilité.
  - **Inconvénients** : Moins performant en lecture (requêtes multiples, équivalent d'une jointure applicative), gestion de la cohérence éventuelle par l'application. Le choix entre ces deux approches dépend des schémas d'accès aux données : si les données sont toujours lues ensemble et que la relation est de type "un-à-peu", l'intégration est souvent préférable. Si la relation est "un-à-beaucoup" ou "plusieurs-à-plusieurs", la liaison est plus adaptée.

Schéma Récapitulatif

https://archive.qoyri.fr/mindmaps/cours21.11.25.html

- **Concepts Clés (SQL)**
  - **Optimiseur de Requêtes** : Cerveau du SGBD qui choisit le **plan d'exécution** le plus efficace en se basant sur le **coût** estimé.
  - **`EXPLAIN`** / **`EXPLAIN ANALYZE`** : Outils pour analyser le plan d'exécution (théorique vs. réel).
  - **Indexation** : Levier principal d'optimisation pour accélérer les lectures (`SELECT`).

- **Maintenance** : `ANALYZE` pour mettre à jour les **statistiques** ; `VACUUM` pour gérer la **fragmentation**.
- **Concurrence** : Gérée par les **transactions (ACID)** et les **verrous (locks)**. Un **deadlock** est une situation de blocage mutuel résolue par le SGBD.
- **Logique Embarquée** : **Procédures**, **fonctions** et **triggers** pour exécuter du code directement dans la base.

- **Concepts Clés (NoSQL)**
  - **Théorème CAP** : Compromis entre **Cohérence (C)**, **Disponibilité (A)** et **Tolérance à la Partition (P)**.
  - **Cohérence Éventuelle** : Compromis privilégiant la disponibilité (systèmes AP).
  - **Réplication** : Copie des données pour la **haute disponibilité** (fiabilité). Gérée via un **Replica Set** et un **quorum**.
  - **Sharding** : Partitionnement des données pour la **scalabilité horizontale** (volume, performance). Repose sur une **clé de sharding**.
  - **Modélisation Document** : Arbitrage entre **Intégration (Embedding)** pour la performance en lecture et **Liaison (Linking)** pour la flexibilité et la gestion de grands volumes de données liées.

- **Mots-Clés**
  - `Optimiseur de requêtes`, `Plan d'exécution`, `Coût`, `EXPLAIN`, `Index Scan`, `Sequential Scan`, `Hash Join`, `Statistiques`, `Fragmentation`, `VACUUM`, `Transaction ACID`, `Deadlock`, `Trigger`, `PL/SQL`, `Théorème CAP`, `Cohérence Éventuelle`, `Réplication`, `Sharding`, `Replica Set`, `Quorum`, `Clé de Sharding`, `Embedding`, `Linking`.

21-05 Cours: Modélisation NoSQL et Sécurité des Systèmes d’Information — schémas, index, chiffrement, audit, monitoring

Modélisation NoSQL et Sécurité des Systèmes d’Information: schémas, accès, chiffrement, audit et monitoring

Introduction générale: une approche intégrée, du modèle de données à la sécurité opérationnelle

La réussite d’un projet data moderne repose sur deux piliers complémentaires: une modélisation NoSQL adaptée aux usages réels et une sécurité “en couches” couvrant

l’accès, le chiffrement, l’audit et le monitoring. Contrairement à la normalisation relationnelle “standardisée”, la modélisation NoSQL (documents, colonnes, clé/valeur) se décide au regard des patrons d’accès, de la volumétrie et des contraintes du moteur choisi (MongoDB, Elasticsearch, DynamoDB, etc.). En parallèle, la sécurité n’est jamais une mesure unique: elle s’organise comme un oignon, par couches successives (réseau, transport, authentification, autorisation, données, opérations). Ce chapitre adopte un ton didactique et pragmatique: il développe les choix de schéma (embedding vs références), les limites physiques (taille des documents, I/O), la gestion des binaires, l’indexation et la performance; puis relie ces décisions au chiffrement (en transit et au repos), aux contrôles d’accès (moindre privilège), à l’audit et au monitoring, jusqu’aux sauvegardes testées et à la posture professionnelle.

Partie I — Modélisation NoSQL: principes, choix et bonnes pratiques

1. Pourquoi “ça dépend” en NoSQL

En NoSQL, le “bon” modèle est celui qui justifie ses compromis au regard:

- Des données: forme, taille, hétérogénéité, volatilité.

- Des patrons d’accès: lecture, écriture, filtres, tri, pagination.
- Des contraintes opérationnelles: latence, réplication, limites de taille, ressources.
- Du moteur: capacités d’indexation, mapping, agrégation, limites de document. Idée clé: on choisit le modèle physique pour simplifier les requêtes critiques, respecter les limites techniques et rester malléable.

2. Schéma-less, mais pas sans discipline

Schema-less n’est pas l’absence totale de structure: c’est la capacité à accepter des documents hétérogènes. Exemple: une spec “couleur” en string, une “taille” en entier/float, ou l’absence de certaines clés d’un produit à l’autre. Cette flexibilité:

- Facilite l’évolution du domaine (ajout de nouvelles specs sans migration lourde).
- Implique une gouvernance applicative (conventions de nommage et types).
- Doit composer avec le mapping des moteurs (Elasticsearch: inflation du mapping si trop de variété). Bonne pratique: définir des conventions internes et des index ciblés sur les attributs réellement exploités.

3. Embedding vs Références: la décision structurante

Deux approches physiques:

- Embedding (imbriquer les sous-documents dans le parent):
  - Avantages: lecture “one-shot”, cohérence locale, simplicité applicative.
  - Limites: taille du document, mises à jour ciblées plus coûteuses, filtrage/pagination compliqués sur de gros tableaux.
- Références (collections séparées + clés étrangères applicatives):
  - Avantages: scalabilité, indexation spécifique, accès ciblé/paginé, contrôle du poids.
  - Limites: jointures applicatives, complexité opérationnelle, cohérence à gérer. Règle d’or:
- Sous-éléments nombreux, évolutifs, accédés indépendamment → collection séparée.
- Petites listes, fortement liées au parent, lues ensemble → embedding.

4. Cardinalité, attributs multi-valués et index

- Relations 1–N: préférer une collection séparée où chaque enfant porte la clé du parent (productId), plutôt qu’une liste d’IDs dans le parent (complexifie requêtes et

maintenance).
- Attributs multi-valués: les tableaux peuvent être indexés mais restent coûteux si volumineux; attention aux limites du moteur (multi-keys, nested).
- Choix des types: pour les prix, privilégier des entiers en centimes ou des décimaux avec précision gérée; pour les catégories, enum si stable, string si flexible (standardiser côté appli).

5. La taille contre le nombre: mesurez en octets et en I/O

“Beaucoup” ne signifie pas “mille éléments”—ce qui importe, c’est la taille totale et l’empreinte I/O:

- Plus un document est gros, plus il est long à transférer et traiter (mémoire, cache, réplication).
- Les limites de taille (ex. 16 Mo) protègent la stabilité des moteurs; éviter d’approcher ces seuils.
- Favoriser les accès via index sélectifs plutôt que le chargement massif. Point de vigilance: un tableau de 100 objets avec corps texte volumineux peut suffire à pénaliser les lectures; séparer et paginer.

6. Données binaires: éviter l’inline

Stocker des binaires en base JSON via base64 gonfle la taille et dégrade les performances.

- Alternatives: stockage objet (S3, GCS, Azure Blob) référencé par URL; GridFS (MongoDB) si besoin côté moteur.
- Pratique: conserver les métadonnées en base et pointer vers le binaire.

7. Indexation et performance: filtrer ≠ charger

- Un index peut cibler des champs imbriqués, mais si la requête n’est pas sélective, le moteur chargera quand même des blocs volumineux.
- Les index multi-clés sur des tableaux aident, au prix d’un coût d’écriture augmenté.
- Design des index dès le départ: sur (productId, nom, valeur) pour specs; sur (productId, date) ou (productId, note) pour avis.

8. Étude de cas: Produits, Specs, Avis

- Produits: id, nom, prix, catégorie, métadonnées; éventuellement champs dérivés (dénomralisation).
- Specs:
  - Embedding si peu nombreuses et descriptives (lecture atomique).
  - Collection séparée si filtrage intensif par attributs (couleur, taille, matière) avec index dédiés.
- Avis:
  - Avoid embedding massif: risque de dépasser la taille et surcoût I/O.
  - Collection “Avis” séparée: pagination, index sur date/note, chargements partiels.
  - Dénormalisation partielle dans le produit: moyenne, compteur, derniers N avis “légers” (snippet), pour accélérer l’affichage. Stratégies de bascule: monitorer la taille des documents; au-delà d’un seuil (N avis ou taille), passer à un modèle référencé.

9. Impact du moteur choisi: MongoDB vs Elasticsearch

- MongoDB: schema-less tolérant, indexes sur champs imbriqués, pipeline d’agrégation (match, unwind, group, project); attention à la limite de 16 Mo par document.
- Elasticsearch: orienté recherche, mapping nécessaire; sous-documents “nested” avec requêtes dédiées; trop de variété de clés dégrade le mapping. Conclusion: le moteur dicte aussi le schéma—adapter, tester, mesurer.

10. Exemples de designs

- Design A (embedding modéré):
  - Produit: specs courtes embedded; avisRésumé (moyenne, count, derniersN).
  - Avis complets séparés (pagination).
- Design B (références fortes):
  - Produit minimal; specs et avis en collections dédiées; index composés ciblés.
- Design C (hybride):
  - Quelques specs “pivots” embedded (taille, couleur); reste séparé.
  - Avis séparés + résumé dans le produit.

11. Analogies et cas applicatifs

- Blog et commentaires: embedding pour petite audience; séparation quand le volume explose.

- Catalogue multimédia: métadonnées très variées (codec, durée, résolution) = force du schema-less.
- Commerce à forte charge: séparation fine des avis/specs pour indexation, sharding et mise à l’échelle.

12. Erreurs fréquentes à éviter

- Listes d’IDs multi-valuées au lieu d’une relation 1–N classique.
- Embedding indiscriminé des avis volumineux.
- Oublier les index sur champs de filtrage.
- Croire que schema-less dispense de discipline.
- Ignorer les coûts d’I/O et la pagination réelle.

Partie II — Sécurité des systèmes d’information et des bases: principes, couches et pratiques

1. Défense en profondeur et moindre privilège

La sécurité est une chaîne; elle cède au maillon le plus faible. Approche par couches:

- Réseau: segmentation, allowlists, pare-feu.
- Transport: chiffrement TLS/mTLS.
- Authentification: comptes individuels, MFA, rotation.
- Autorisation: rôles, droits CRUD limités, RLS si disponible.
- Données: chiffrement au repos, politiques de purge maîtrisées.
- Opérations: journalisation, audit externe, sauvegardes testées, procédures d’incident.
  Principe du moindre privilège: ne donner que les droits nécessaires; limiter l’impact d’une compromission.

2. Contrôles d’accès: rôles, permissions, soft delete

- Gérer via rôles (lecteur, éditeur, service, admin); rattacher les comptes aux rôles, pas de droits ad hoc.
- Politique de soft delete: retirer DELETE; marquer “archivé = true”; prévoir vues et audits; respecter RGPD (droit à l’effacement). Modèles d’identité:
- Compte technique unique (simple, mais audit détaillé côté BD plus difficile).
- Propagation d’identité (SSO/Kerberos/JWT) jusque dans la BD (audit fin, complexité accrue).

3. Authentification et hygiène

- Changer impérativement les comptes par défaut (admin/admin).
- Mots de passe forts, rotation, gestionnaire de secrets; offboarding rigoureux.
- Activer et configurer l’authentification/autorisation côté BD (ne pas laisser en mode “ouvert”). Curiosité historique: de nombreuses compromissions “simples” exploitent des identifiants par défaut non modifiés.

4. Contrôle d’accès réseau

- Restreindre aux hôtes connus; différencier dev/test/prod.
- Éviter les connexions directes depuis postes utilisateurs vers BD de production.
- Segmenter (VLAN), ACL, bastion administrateur. Exemple MySQL: lier les comptes à “utilisateur@hôte” avec hôte restreint (éviter “%”).

5. Journalisation et audit

Objectifs: forensique, conformité, accounting.

- Journaliser connexions, DDL, DML critiques, requêtes lentes.
- Protéger les journaux (intégrité; stockage externe immuable/WORM).
- Corréler dans un SIEM (ELK/Opensearch, Splunk). Curiosité: journaux “tamper-evident” enchaînés par hachages, inspirés des blockchains.

6. Chiffrement: en transit et au repos

- En transit (TLS/SSL): prévenir interception et modification; gérer certificats, rotation; mTLS pour microservices.
- Au repos:
  - Chiffrement de disque (FDE: BitLocker, LUKS, FileVault): protection contre vol physique; transparent pour l'application.
  - TDE au niveau moteur (Oracle, SQL Server, extensions PostgreSQL): fichiers de base et backups chiffrés; nécessite KMS/HSM.
  - Chiffrement applicatif (colonnes/blobs): protection forte même avec accès moteur; perte de requêtabilité et complexité de clés. Pragmatisme PKI interne: commencer par la périphérie (reverse proxy/API gateway, VPN), puis étendre le chiffrement interne avec automation (ACME/step-ca/Vault) ou service mesh (Istio, Linkerd).

7. Sauvegardes, transactions et reprise

- Transactions: BEGIN/COMMIT/ROLLBACK; éviter auto-commit pour opérations massives.
- Sauvegardes: full/incrémentales/différentielles/snapshots; isoler et durcir les dépôts; chiffrer les backups.
- Tester les restaurations régulièrement: une sauvegarde non testée n'est pas une sauvegarde; valider RPO/RTO, intégrité, runbooks. Cas réel: backups "réussis" mais vides (0 octet) faute de droits; jour J, rien à restaurer.

8. Décommissionnement des supports

- Effacement cryptographique si disque chiffré (crypto-erase).
- Déchiquetage industriel (DIN 66399), Secure Erase/PSID pour SSD, percage/déformation des plateaux pour HDD.
- Documenter la procédure, séparer HDD/SSD, prestataires certifiés. Curiosité: l'aimant "courant" n'a pas d'effet sur HDD modernes; nul pour SSD.

Partie III — Sécurité des applications web: injections SQL, XSS, validation et WAF

1. Injections SQL: nature et prévention

Définition: injection lorsque des données utilisateur deviennent du code SQL et modifient la requête.

- Exemples: ' OR 1=1 --, UNION, blind/time-based.
- Impacts: bypass d’authentification, exfiltration, modification, DROP selon droits. Prévention incontournable: requêtes préparées (prepared statements).
- Séparent code et données; les paramètres ne sont pas interprétés comme SQL.
- Bénéfices de performance: plan cache, stabilité, moins de parsing. Bonnes pratiques:
- APIs paramétrées (PDO, PreparedStatement, psycopg2/sqlalchemy).
- Typage explicite des paramètres; encodage cohérent (UTF-8).
- Messages d’erreur non verbeux côté utilisateur; journaux détaillés côté SIEM.
- Moindre privilège: même si injection, les dégâts sont limités.

2. Validation des entrées et échappement des sorties

- Input validation: whitelist, longueur, jeu de caractères, normalisation (NFKC); utile mais ne remplace pas les prepared statements.
- Output escaping: protéger contre le XSS (HTML, attribut, JS, URL); outils et CSP; ne pas confondre avec protection SQL.

3. WAF et détection

- WAF niveau 7 (OWASP CRS): détecter patterns d’attaques, journaliser, bloquer/quarantaine.
- Limites: complément, pas substitut au code sécurisé.
- Réponse aux incidents: playbooks, corrélations SIEM, durcissement des règles.

Partie IV — Monitoring, opérations et posture de conseil

1. Logging et transactions: éviter le piège du rollback des logs

- Anti-pattern: logguer “en base” dans la même transaction que le métier — le rollback efface aussi les logs.
- Solutions:
  - Transaction séparée et commit immédiat pour la ligne de log.
  - Logging hors base (stdout, fichiers, bus → collecte centralisée).
  - Audit natif BD pour événements sensibles (pgaudit, profiler). Pratiques:
- Corrélation: request_id/session_id dans chaque log et write BD.
- Séparer log d’événement vs log d’audit; niveaux (ERROR/WARN/INFO); rétention.

2. Observabilité: métriques, traces et alertes

- Time-series: Prometheus, InfluxDB; dashboards (Grafana).
- Logs/traces: ELK/Opensearch, Loki, OpenTelemetry.
- Métriques critiques: connexions, latence, locks, I/O disque, buffer cache, CPU/RAM, taille des index/tables, WAL/redo.
- Alertes: disque >80%, dérive latences, locks longs, croissance anormale des journaux. Checklist santé par SGBD:
- PostgreSQL: pg_stat_activity, pg_stat_statements, autovacuum, pgaudit.
- MySQL/MariaDB: performance_schema, information_schema.
- Oracle: AWR/ASH, OEM.
- MongoDB: profiler, Compass.

3. Accès et réseau: segmentation, bastion, certificats

- Segmentation stricte; deny-by-default; bastion avec MFA pour admins.
- TLS/mTLS entre application et BD; PKI interne; automatiser renouvellement (ACME/Step CA).
- Argumentaire: coût des certificats négligeable face au coût d’une fuite.

4. Sauvegardes et restaurations: vérité opérationnelle

- Stratégie: full + incrémentales; isoler, immutabilité (WORM).
- Tests de restauration périodiques; validation d’intégrité (hash), contenu, performances de reprise.
- Documentation: runbooks d’incident; responsabilités claires.

5. Scénarios d’attaque et erreurs systémiques

- Brute force sur admin évident; élévation aux hyperviseurs; destruction des backups; rançon.
  - Mesures: MFA, segmentation, immutabilité des backups, détection brute force, moindre privilège.
- Risque interne: mécontentement, négligence, collusion.
  - Mesures: audit immuable, revue d’accès périodique, séparation des tâches, culture de sécurité.

6. Posture de conseil et responsabilité

- Devoir de conseil: exposer risques et options; adapter au contexte; formuler les compromis.
- Limites: refuser un engagement manifestement non sécurisé; décharge écrite si nécessaire.
- Traçabilité: ordres de changement signés pour actions sensibles; journaliser qui/quoi/quand/pourquoi.
- Éthique: environnement professionnel respectueux; évolution de responsabilités (de savoir-faire à pilotage des choix).

Interdisciplinarité et liens utiles

- Génie logiciel: DDD pour déterminer agrégats (embedding) vs bounded contexts (références).
- Architecture systèmes: impact schéma sur réplication, sharding, tolérance aux pannes, coûts cloud.
- Sécurité réseau: Zero Trust, mTLS, bastions, segmentation dynamique.
- Data engineering: pipeline ETL, indexation vers moteurs de recherche (Elasticsearch), caches (Redis), data lakes (Parquet/Avro, Spark/Flink).
- Conformité: RGPD (minimisation, anonymisation, droit à l’effacement), PCI-DSS.
- SRE/Opérations: CI/CD avec scanners SAST/DAST, runbooks, game days. Curiosités historiques:
- Essor du NoSQL avec la scalabilité web; MongoDB a popularisé documents JSON-like et pipelines d’agrégation.
- Elasticsearch/Lucene: index inversés, analyzers; mapping strict pour performance de recherche.

- Limites de taille des documents: corrélées aux pages/blocs internes et aux coûts de réplication; compromis de stabilité.

Exemples pratiques intégrés

Cas 1: E-commerce avec filtrage massif de specs et avis nombreux

- Specs: collection séparée avec index sur (productId, nom, valeur) pour “couleur/taille/matière”.
- Avis: collection dédiée, index (productId, date) pour “les 4 plus récents”; produits dénormalisés avec moyenne et compteur.
- Monitoring: latences par requête, taille des collections, croissance index.

Cas 2: Binaires (PDF de factures)

- Métadonnées en base; fichiers en stockage objet (URL, hash, MIME); GridFS si moteur document imposé.
- Sécurité: TDE/FDE pour backups; chiffrement en transit; accès restreints.

Cas 3: Petites listes fortement liées (adresses client)

- Embedding si 1–2 adresses statiques; collection séparée si historique riche (validation, audit, pagination).

Cas 4: Web App et SQLi/XSS

- Prepared statements partout; validation en entrée; escaping en sortie (HTML/JS/URL); CSP; WAF en couche supplémentaire; moindre privilège sur comptes BD.

Pistes méthodologiques: concevoir, tester, ajuster

1. Modèle conceptuel: entités/relations (produits, specs, avis, clients).
2. Patrons d’accès et SLA: requêtes fréquentes, volumes, filtres/tri/pagination.
3. Modèle physique et index:
   - Embedding vs référenciation par cardinalité et volumétrie.
   - Dimensionner la taille des documents; planifier index et partitions.
   - Prévoir pipelines (ETL, caches, search), résilience et resynchronisation.
4. Sécurité et opérations:
   - Rôles et moindres privilèges; TLS/mTLS; audit externe; sauvegardes testées.
   - Observabilité: métriques, logs, traces; alertes actionnables.
5. Itérations: mesurer, profiler, corriger (coût des requêtes, taille des payloads, plans d’exécution).

Points clés et mots-clés (mini-listes contextualisées)

- Modélisation:
  - Embedding vs Références
  - Cardinalité
  - Schema-less discipliné
  - Index multi-clés
  - I/O et taille de document
- Sécurité:
  - Défense en profondeur
  - Moindre privilège
  - TLS/mTLS, TDE/FDE
  - Audit immuable, SIEM
  - Soft delete, RLS
- Web:
  - Prepared statements
  - Input validation, Output escaping
  - XSS, WAF, CSP
- Opérations:
  - Logging hors transaction
  - Monitoring (Prometheus/Grafana)

    - Sauvegardes testées, RPO/RTO
    - Runbooks, game days
- Architecture:
    - DDD, bounded contexts
    - ETL, caches, search
    - PKI interne, service mesh

Schéma riassuntivo: concepts principaux et mots-clés

Concepts principaux:

- Modèle NoSQL centré usages: choisir embedding ou références selon cardinalité, filtrage, volumétrie et limites du moteur (MongoDB, Elasticsearch).
- Taille > nombre: mesurer en octets et en I/O; éviter les documents lourds; indexer les champs de filtrage.
- Binaires hors base: stockage objet + métadonnées; GridFS si nécessaire.
- Sécurité en couches: réseau (segmentation), transport (TLS/mTLS), identité (MFA, rotation), autorisation (rôles, moindre privilège), données (TDE/FDE, chiffrement applicatif), opérations (audit, sauvegardes testées).
- Prévention des injections: requêtes préparées, validation en entrée, escaping en sortie pour XSS, WAF en complément.
- Observabilité: logging hors transaction, audit BD, métriques/traces, alertes; tests de restauration réguliers.
- Gouvernance et posture: devoir de conseil, traçabilité des actions sensibles, refus de risque non maîtrisé. Mots-clés:
- NoSQL, embedding, références, schema-less, cardinalité, index, I/O, dénormalisation, MongoDB, Elasticsearch, mapping/nested, agrégation.
- Défense en profondeur, moindre privilège, rôles, CRUD, soft delete, RLS, TLS/mTLS, TDE/FDE, KMS/HSM.
- Audit, journalisation, WORM, SIEM, prepared statements, plan cache, input validation, output escaping, XSS, WAF, CSP.
- Prometheus, Grafana, ELK/Opensearch, OpenTelemetry, pgaudit, performance_schema.
- Sauvegardes, restauration, RPO/RTO, runbooks, game days, PKI interne, service mesh, ETL, caches, search, RGPD.

https://archive.qoyri.fr/mindmaps/cours21.11.25.2.html