Le Problème N+1 Query - Guide Complet pour Data Analyst

Vue d'Ensemble

Le **problème N+1** est l'un des problèmes de performance les plus courants et les plus coûteux dans les applications qui interagissent avec des bases de données. En tant que Data Analyst, comprendre ce problème est crucial pour: - Optimiser vos scripts de data extraction - Diagnostiquer les lenteurs dans les pipelines ETL - Évaluer la qualité du code backend - Améliorer vos propres requêtes

---

Qu'est-ce que le Problème N+1?

Définition Simple

Le problème N+1 se produit quand votre code exécute: - **1 requête** pour récupérer une liste d'éléments principaux - **N requêtes supplémentaires** (une par élément) pour récupérer les données liées

Total: N+1 requêtes au lieu d'une seule requête optimisée.

Exemple Concret - Contexte Bancaire

Supposons que vous voulez analyser les transactions de tous les clients:

```
# MAUVAIS - Problème N+1
clients = session.query(Client).all()  # 1 requête

for client in clients:  # Si 1000 clients
    transactions = client.transactions  # 1000 requêtes supplémentaires!
    # Analyse...
```

Résultat: 1001 requêtes au lieu de 1 ou 2!

Pourquoi C'est Grave?

Impact sur la Performance

Scénario	Requêtes	Temps (estimé)
100 clients	101	~500ms
1,000 clients	1,001	~5 secondes
10,000 clients	10,001	~50 secondes
100,000 clients	100,001	~8 minutes

Problèmes Associés

1. **Latence réseau:** Chaque requête = aller-retour vers la DB
2. **Charge serveur DB:** Surcharge du serveur de base de données
3. **Timeouts:** Risque de dépassement des délais
4. **Coûts cloud:** Plus de requêtes = plus de coûts (BigQuery, etc.)
5. **Blocages:** Risque de locks prolongés

SQLAlchemy / SQLModel (Python)

Configuration de Base

```python
from sqlalchemy import create_engine, Column, Integer, String, ForeignKey, Float
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship, sessionmaker, joinedload, selectinload
from sqlmodel import SQLModel, Field, Relationship

# SQLAlchemy classique
Base = declarative_base()

class Client(Base):
    __tablename__ = 'clients'

    id = Column(Integer, primary_key=True)
    nom = Column(String(100))
    email = Column(String(100))

    # Relation
    transactions = relationship("Transaction", back_populates="client")
    comptes = relationship("Compte", back_populates="client")

class Transaction(Base):
    __tablename__ = 'transactions'

    id = Column(Integer, primary_key=True)
    client_id = Column(Integer, ForeignKey('clients.id'))
    montant = Column(Float)
    date = Column(DateTime)

    client = relationship("Client", back_populates="transactions")
```

Le Problème - Code Naïf

```python
#  MAUVAIS - Génère N+1 requêtes
def get_client_transactions_bad():
    clients = session.query(Client).all()  # 1 requête

    results = []
    for client in clients:
        # Chaque accès à client.transactions déclenche une requête!
        total = sum(t.montant for t in client.transactions)  # N requêtes
        results.append({
            'client': client.nom,
            'total': total
        })

    return results

# Logs SQL générés:
# SELECT * FROM clients;
# SELECT * FROM transactions WHERE client_id = 1;
# SELECT * FROM transactions WHERE client_id = 2;
```

```
# SELECT * FROM transactions WHERE client_id = 3;
# ... (répété N fois)
```

Solution 1: Eager Loading avec `joinedload`

```
from sqlalchemy.orm import joinedload

#  BON - Une seule requête avec JOIN
def get_client_transactions_good_v1():
    clients = session.query(Client)\
        .options(joinedload(Client.transactions))\
        .all()  # 1 seule requête avec LEFT JOIN

    results = []
    for client in clients:
        # Données déjà chargées - pas de requête supplémentaire!
        total = sum(t.montant for t in client.transactions)
        results.append({
            'client': client.nom,
            'total': total
        })

    return results

# SQL généré:
# SELECT clients.*, transactions.*
# FROM clients
# LEFT OUTER JOIN transactions ON clients.id = transactions.client_id;
```

Solution 2: Eager Loading avec `selectinload`

```
from sqlalchemy.orm import selectinload

#  BON - Deux requêtes (mieux pour grandes collections)
def get_client_transactions_good_v2():
    clients = session.query(Client)\
        .options(selectinload(Client.transactions))\
        .all()  # 2 requêtes: 1 pour clients, 1 pour toutes les transactions

    results = []
    for client in clients:
        total = sum(t.montant for t in client.transactions)
        results.append({
            'client': client.nom,
            'total': total
        })

    return results

# SQL généré:
# SELECT * FROM clients;
# SELECT * FROM transactions WHERE client_id IN (1, 2, 3, 4, ...);
```

Solution 3: Agrégation directe en SQL

```python
from sqlalchemy import func

#  MEILLEUR - Calcul côté base de données
def get_client_totals_best():
    results = session.query(
        Client.nom,
        func.sum(Transaction.montant).label('total')
    )\
    .join(Transaction)\
    .group_by(Client.id, Client.nom)\
    .all()

    return [{'client': r.nom, 'total': r.total} for r in results]

# SQL généré:
# SELECT clients.nom, SUM(transactions.montant) as total
# FROM clients
# JOIN transactions ON clients.id = transactions.client_id
# GROUP BY clients.id, clients.nom;
```

Comparaison des Approches

Méthode	Requêtes	Quand utiliser
Sans optimisation	N+1	❌ Jamais
joinedload	1	Peu de données liées par parent
selectinload	2	Beaucoup de données liées
subqueryload	2	Alternative à selectinload
Agrégation SQL	1	Quand on veut seulement des agrégats

SQLModel (Modern Python)

SQLModel utilise SQLAlchemy sous le capot, donc les mêmes solutions s'appliquent:

```python
from sqlmodel import SQLModel, Field, Relationship, Session, select
from sqlalchemy.orm import selectinload

class Client(SQLModel, table=True):
    id: int | None = Field(default=None, primary_key=True)
    nom: str
    email: str

    transactions: list["Transaction"] = Relationship(back_populates="client")

class Transaction(SQLModel, table=True):
    id: int | None = Field(default=None, primary_key=True)
    client_id: int = Field(foreign_key="client.id")
    montant: float

    client: Client | None = Relationship(back_populates="transactions")
```

```python
# Solution avec SQLModel
def get_clients_with_transactions(session: Session):
    statement = select(Client).options(selectinload(Client.transactions))
    clients = session.exec(statement).all()
    return clients
```

Spring Data JPA (Java) - Pour Référence

Même si tu n'es pas familier avec Java, voici comment Spring JPA gère ce problème:

Le Problème en JPA

```java
// Entity
@Entity
public class Client {
    @Id
    private Long id;
    private String nom;

    @OneToMany(mappedBy = "client", fetch = FetchType.LAZY)  // LAZY par défaut
    private List<Transaction> transactions;
}

//  MAUVAIS - N+1
List<Client> clients = clientRepository.findAll();
for (Client client : clients) {
    client.getTransactions().size();  // Déclenche une requête!
}
```

Solutions JPA

```java
// Solution 1: JPQL avec JOIN FETCH
@Query("SELECT c FROM Client c JOIN FETCH c.transactions")
List<Client> findAllWithTransactions();

// Solution 2: EntityGraph
@EntityGraph(attributePaths = {"transactions"})
List<Client> findAll();

// Solution 3: Batch fetching (dans application.properties)
spring.jpa.properties.hibernate.default_batch_fetch_size=100
```

Équivalences Python ↔ Java

Python (SQLAlchemy)	Java (JPA/Hibernate)
`joinedload()`	`JOIN FETCH` ou `@EntityGraph`
`selectinload()`	`@BatchSize` ou `IN` query
`lazy='dynamic'`	`FetchType.LAZY`
`lazy='joined'`	`FetchType.EAGER` avec JOIN

Comment Détecter le Problème N+1 (Data Analyst)

1. Activer le Logging SQL

```python
# SQLAlchemy - Activer les logs
import logging
logging.basicConfig()
logging.getLogger('sqlalchemy.engine').setLevel(logging.INFO)

# Ou dans la création de l'engine
engine = create_engine(
    'postgresql://user:pass@localhost/db',
    echo=True  # Affiche toutes les requêtes
)
```

2. Analyser les Patterns dans les Logs

Symptômes du N+1:

```
INFO: SELECT * FROM clients
INFO: SELECT * FROM transactions WHERE client_id = 1
INFO: SELECT * FROM transactions WHERE client_id = 2
INFO: SELECT * FROM transactions WHERE client_id = 3
# ... pattern répétitif!
```

Requête optimisée:

```
INFO: SELECT * FROM clients LEFT JOIN transactions ON ...
# Une seule requête!
```

3. Utiliser des Outils de Profiling

```python
# SQLAlchemy-Utils pour détecter
from sqlalchemy_utils import get_query_entities

# Ou utiliser le profiler intégré
from sqlalchemy import event

query_count = 0

@event.listens_for(engine, "before_cursor_execute")
def receive_before_cursor_execute(conn, cursor, statement, parameters, context, executemany):
    global query_count
    query_count += 1
    print(f"Query #{query_count}: {statement[:100]}...")

# Après votre code
print(f"Total queries: {query_count}")
```

4. Script de Détection Automatique

```python
def detect_n_plus_1(func, session):
    """Détecte si une fonction génère un pattern N+1"""
```

```
    queries = []

    @event.listens_for(session.bind, "before_cursor_execute")
    def log_query(conn, cursor, statement, parameters, context, executemany):
        queries.append(statement)

    # Exécuter la fonction
    result = func()

    # Analyser les patterns
    if len(queries) > 10:
        # Chercher des requêtes répétitives
        from collections import Counter
        patterns = Counter()
        for q in queries:
            # Extraire le pattern de base (sans les valeurs spécifiques)
            pattern = re.sub(r'\d+', 'N', q)
            patterns[pattern] += 1

        for pattern, count in patterns.most_common(5):
            if count > 5:
                print(f"   POSSIBLE N+1: {count} requêtes similaires")
                print(f"    Pattern: {pattern[:100]}...")

    print(f"Total: {len(queries)} requêtes")
    return result
```

5. Monitoring en Production

```
# Avec des métriques (ex: Prometheus)
from prometheus_client import Counter, Histogram

query_counter = Counter('db_queries_total', 'Total database queries')
query_duration = Histogram('db_query_duration_seconds', 'Query duration')

# Dans votre code
@event.listens_for(engine, "before_cursor_execute")
def before_query(*args):
    query_counter.inc()
```

Checklist pour Data Analyst

Avant d'Écrire du Code

```
Identifier les relations entre entités
Estimer le volume de données
Planifier les agrégations nécessaires
Choisir la stratégie de chargement appropriée
```

Pendant le Développement

```
Activer les logs SQL
```

```
Utiliser eager loading pour les relations accédées
Préférer les agrégations SQL aux calculs Python
Tester avec des volumes réalistes
```

Code Review

```
Vérifier l'absence de boucles avec accès aux relations
S'assurer que les options de chargement sont spécifiées
Valider le nombre de requêtes avec les logs
Comparer les temps d'exécution
```

Cas Pratiques Bancaires

Cas 1: Rapport des Transactions par Client

```python
#  MAUVAIS
def rapport_transactions_bad():
    clients = session.query(Client).all()
    rapport = []
    for client in clients:
        for tx in client.transactions:  # N+1!
            rapport.append({
                'client': client.nom,
                'montant': tx.montant,
                'date': tx.date
            })
    return pd.DataFrame(rapport)

#  BON
def rapport_transactions_good():
    query = session.query(
        Client.nom,
        Transaction.montant,
        Transaction.date
    ).join(Transaction)

    return pd.read_sql(query.statement, session.bind)
```

Cas 2: Calcul NPL par Secteur

```python
#  MAUVAIS
def npl_par_secteur_bad():
    secteurs = session.query(Secteur).all()
    results = {}
    for secteur in secteurs:
        prets = secteur.prets  # N+1!
        npl = sum(1 for p in prets if p.jours_retard > 90)
        total = len(prets)
        results[secteur.nom] = npl / total if total > 0 else 0
    return results

#  BON
```

```python
def npl_par_secteur_good():
    query = session.query(
        Secteur.nom,
        func.count(case((Pret.jours_retard > 90, 1))).label('npl'),
        func.count(Pret.id).label('total')
    ).join(Pret).group_by(Secteur.nom)

    return {r.nom: r.npl / r.total if r.total > 0 else 0 for r in query.all()}
```

Cas 3: Export pour Power BI

```python
#  BON - Extraction optimisée pour BI
def export_for_powerbi():
    # Une seule requête avec toutes les jointures
    query = """
    SELECT
        c.id as client_id,
        c.nom as client_nom,
        c.segment,
        t.id as transaction_id,
        t.montant,
        t.date,
        t.type,
        a.code as agence_code,
        a.region
    FROM clients c
    LEFT JOIN transactions t ON c.id = t.client_id
    LEFT JOIN agences a ON c.agence_id = a.id
    WHERE t.date >= :start_date
    """

    df = pd.read_sql(query, engine, params={'start_date': '2024-01-01'})
    df.to_parquet('export_powerbi.parquet')
    return df
```

Résumé des Solutions

Problème	Solution Python	Solution Java
Chargement de relations	`joinedload()` / `selectinload()`	`JOIN FETCH` / `@EntityGraph`
Boucle avec accès relation	Refactorer avec JOIN	Refactorer avec JPQL
Agrégations	`func.sum()`, `group_by()`	`@Query` avec agrégation
Export massif	`pd.read_sql()` directement	Native query

Points Clés à Retenir

1. **Le N+1 est silencieux** - Le code fonctionne, mais lentement

2. **Toujours activer les logs SQL** pendant le développement
3. **Préférer les calculs côté base de données** aux boucles Python
4. **Utiliser eager loading** pour les relations que vous allez accéder
5. **Tester avec des volumes réalistes** - Le problème n'apparaît pas avec 10 lignes

---

Questions d'Entretien Fréquentes

Q: Qu'est-ce que le problème N+1? > Exécution de 1 requête pour la liste principale + N requêtes pour les détails, au lieu d'une seule requête optimisée.

Q: Comment le détecter? > Activer les logs SQL et chercher des patterns de requêtes répétitives.

Q: Comment le résoudre en SQLAlchemy? > Utiliser `joinedload()` ou `selectinload()`, ou refactorer avec des JOIN explicites.

Q: Quelle est la différence entre joinedload et selectinload? > joinedload = 1 requête avec JOIN (bon pour peu de données liées) > selectinload = 2 requêtes avec IN clause (bon pour beaucoup de données liées)