Frameworks Comparison: Credit Scoring

Frameworks Comparison: The Data Analyst’s Toolkit

Python (SQLAlchemy + Pandas) vs Java (JPA/JDBC + DFlib)

Phase: Études — Exploration des Frameworks **Contexte:** Banking Credit Scoring System

---

Table des Matières

---

1. Vue d’ensemble architecturale

Le workflow d’un data analyst suit toujours le même schéma, quel que soit le langage :

`[Base de données] → [Couche ORM/Query] → [DataFrame] → [Analyse/Visualisation]`

Voici comment chaque brique se mappe entre les deux écosystèmes :

Rôle	Python	Java
ORM complet	SQLAlchemy (Core + ORM)	JPA (Jakarta Persistence API)
Query builder	SQLAlchemy Core `select()`	JDBC + SQL natif
Driver bas niveau	`pymysql` / `asyncmy`	JDBC (`mysql-connector-java`)
DataFrame	Pandas (`pd.DataFrame`)	DFlib (`DataFrame`)

Rôle	Python	Java
SQL → DataFrame	`pd.read_sql()`	`Jdbc.connector().read()`
Migration	Alembic	Flyway
Gestion deps	uv (`pyproject.toml`)	Maven (`pom.xml`)

Différences fondamentales de philosophie

SQLAlchemy est un framework à deux niveaux. Le **Core** est un query builder SQL pur (on écrit du SQL de manière programmatique). L'**ORM** ajoute une couche d'abstraction objet par-dessus. En tant que data analyst, tu utilises souvent les deux : l'ORM pour définir le schéma, le Core pour écrire des requêtes analytiques complexes.

JPA (Jakarta Persistence API) est une **spécification**, pas une implémentation. Hibernate est l'implémentation la plus répandue. JPA est purement ORM : on pense en objets, on écrit en JPQL (Java Persistence Query Language). Pour les requêtes complexes, on descend vers du **Native SQL** ou on utilise le **Criteria API**.

JDBC est le niveau le plus bas en Java — l'équivalent direct de `pymysql`. On écrit du SQL brut, on récupère des `ResultSet`. C'est ce que DFlib utilise directement pour charger les données.

---

2. Le Schéma Commun — Credit Scoring Model

Pour toutes les comparaisons, nous utilisons ce schéma relationnel bancaire :

```
clients (id PK, first_name, last_name, email, annual_income DECIMAL,
         employment_status ENUM, credit_history_years INT, registration_date)
    │
    └──< credit_applications (id PK, client_id FK, product_id FK,
    │         requested_amount DECIMAL, credit_score INT, risk_level ENUM,
    │         status ENUM, interest_rate DECIMAL, application_date)
    │
loan_products (id PK, code, name, max_amount DECIMAL,
               base_rate DECIMAL, term_months INT)
```

Relations : - Un **client** dépose plusieurs **demandes de crédit** (credit_applications) - Un **produit de prêt** (loan_product) est référencé dans plusieurs **demandes** - **credit_applications** est la table de jonction enrichie

— elle porte le `credit_score`, le `risk_level`, le `status`, et le `interest_rate` calculé

Contexte métier

Le **credit scoring** est le processus par lequel une banque évalue le risque d'un emprunteur. Le score (typiquement 300-850, calqué sur FICO) détermine : - Si la demande est **approuvée**, **rejetée**, ou **en révision** - Le **niveau de risque** (LOW, MEDIUM, HIGH, CRITICAL) - Le **taux d'intérêt** appliqué (plus le risque est élevé, plus le taux monte)

En tant que data analyst bancaire, les questions typiques sont : - Quel est le taux d'approbation par produit ? - Quelle est la distribution des scores par tranche de revenu ? - Quel est le taux moyen accordé par niveau de risque ? - Quels clients à haut revenu ont un score étonnamment bas ? (anomalies)

---

3. Couche ORM : Définition des Modèles

3.1 — SQLAlchemy 2.0 (Nouvelle API avec `Mapped` et `mapped_column`)

SQLAlchemy 2.0 a introduit un changement majeur : le passage de l'ancienne syntaxe `Column()` vers le système de type annotations avec `Mapped[]` et `mapped_column()`. Ce nouveau style apporte la **type safety** native — Pydantic et les IDE comprennent les types directement.

Points clés de la nouvelle API : - `DeclarativeBase` remplace `declarative_base()` - `Mapped[type]` déclare le type Python de la colonne - `mapped_column()` remplace `Column()` et infère le type SQL depuis le type Python - `relationship()` reste identique mais profite du typage `Mapped[list[...]]` - Les `Enum` Python se mappent directement vers des `ENUM` MySQL

```
# models.py — SQLAlchemy 2.0 (Sync) — Banking Credit Scoring
import enum
from datetime import date
from decimal import Decimal
from sqlalchemy import ForeignKey, String, DECIMAL, Enum, Integer, create_engine
from sqlalchemy.orm import (
    DeclarativeBase, Mapped, mapped_column,
    relationship, sessionmaker
)


# --- Enums métier ---
class EmploymentStatus(enum.Enum):
    """Statut d'emploi du client.
    Impacte directement le scoring : un CDI est plus favorable
```

```
qu'un indépendant.
            """
            SALARIED = "SALARIED"           # CDI / salarié
            SELF_EMPLOYED = "SELF_EMPLOYED"  # Indépendant / freelance
            UNEMPLOYED = "UNEMPLOYED"        # Sans emploi
            RETIRED = "RETIRED"              # Retraité
            STUDENT = "STUDENT"              # Étudiant


        class RiskLevel(enum.Enum):
            """Niveau de risque calculé à partir du credit_score.
            LOW: score >= 750 | MEDIUM: 650-749 | HIGH: 550-649 |
CRITICAL: < 550
            """
            LOW = "LOW"
            MEDIUM = "MEDIUM"
            HIGH = "HIGH"
            CRITICAL = "CRITICAL"


        class ApplicationStatus(enum.Enum):
            """Statut du dossier de demande de crédit."""
            PENDING = "PENDING"      # En attente d'analyse
            APPROVED = "APPROVED"    # Approuvé
            REJECTED = "REJECTED"    # Rejeté
            UNDER_REVIEW = "UNDER_REVIEW"  # En révision manuelle


        class Base(DeclarativeBase):
            """Classe de base pour tous les modèles.
            Remplace l'ancien declarative_base().
            """
            pass


        class Client(Base):
            __tablename__ = "clients"


# Mapped[int] → INTEGER, mapped_column(primary_key=True) → PK
auto-increment
            id: Mapped[int] = mapped_column(primary_key=True,
autoincrement=True)
            first_name: Mapped[str] = mapped_column(String(100))
            last_name: Mapped[str] = mapped_column(String(100))
            email: Mapped[str] = mapped_column(String(255), unique=True)
            annual_income: Mapped[Decimal] = mapped_column(DECIMAL(12,
2))
```

```
            employment_status: Mapped[EmploymentStatus] = mapped_column(
                Enum(EmploymentStatus, native_enum=True)
            )
            credit_history_years: Mapped[int] = mapped_column(Integer,
default=0)
            registration_date: Mapped[date] =
mapped_column(default=date.today)

            # Relationship: un client a plusieurs demandes de crédit
            # back_populates crée la liaison bidirectionnelle
            applications: Mapped[list["CreditApplication"]] =
relationship(
                back_populates="client"
            )

            def __repr__(self) -> str:
                return f"Client(id={self.id}, name={self.first_name}
{self.last_name}, income={self.annual_income})"


        class LoanProduct(Base):
            __tablename__ = "loan_products"

            id: Mapped[int] = mapped_column(primary_key=True,
autoincrement=True)
            code: Mapped[str] = mapped_column(String(20), unique=True)
            name: Mapped[str] = mapped_column(String(200))
            max_amount: Mapped[Decimal] = mapped_column(DECIMAL(14, 2))
            base_rate: Mapped[Decimal] = mapped_column(DECIMAL(5, 2))  #
taux de base en %
            term_months: Mapped[int] = mapped_column(Integer)

            applications: Mapped[list["CreditApplication"]] =
relationship(
                back_populates="product"
            )

            def __repr__(self) -> str:
                return f"LoanProduct(id={self.id}, code={self.code},
rate={self.base_rate}%)"


        class CreditApplication(Base):
            __tablename__ = "credit_applications"

            id: Mapped[int] = mapped_column(primary_key=True,
autoincrement=True)
            client_id: Mapped[int] =
```

```
mapped_column(ForeignKey("clients.id"))
            product_id: Mapped[int] =
mapped_column(ForeignKey("loan_products.id"))

            # Données de la demande
            requested_amount: Mapped[Decimal] =
mapped_column(DECIMAL(14, 2))
            credit_score: Mapped[int] = mapped_column(Integer)
# 300-850 (échelle FICO)
            risk_level: Mapped[RiskLevel]
= mapped_column(Enum(RiskLevel, native_enum=True))
            status: Mapped[ApplicationStatus] = mapped_column(
                Enum(ApplicationStatus, native_enum=True),
                default=ApplicationStatus.PENDING
            )
            interest_rate: Mapped[Decimal] = mapped_column(DECIMAL(5,
2))  # taux accordé
            application_date: Mapped[date] =
mapped_column(default=date.today)

            # Relations inverses
            client: Mapped["Client"] =
relationship(back_populates="applications")
            product: Mapped["LoanProduct"] =
relationship(back_populates="applications")

            def __repr__(self) -> str:
                return (
                    f"CreditApplication(client_id={self.client_id}, "
                    f"score={self.credit_score},
risk={self.risk_level.value}, "
                    f"status={self.status.value})"
                )


        # --- Engine et Session Factory ---
        DATABASE_URL = "mysql+pymysql://analyst:analyst@mysql:3306/
credit_scoring_db"
        engine = create_engine(DATABASE_URL, echo=False)
        SessionLocal = sessionmaker(bind=engine)
```

3.2 — SQLAlchemy 2.0 (Async)

La version asynchrone change uniquement la couche connexion. Les modèles restent identiques. Ce qui change : `create_async_engine`, `async_sessionmaker`, et `AsyncSession`.

```
        # models_async.py — SQLAlchemy 2.0 (Async) — Banking Credit
Scoring
```

```python
from sqlalchemy.ext.asyncio import (
    create_async_engine,
    async_sessionmaker,
    AsyncSession
)

# Le driver change : pymysql → asyncmy (ou aiomysql)
ASYNC_DATABASE_URL = "mysql+asyncmy://analyst:analyst@mysql:3306/
credit_scoring_db"

async_engine = create_async_engine(ASYNC_DATABASE_URL,
echo=False)
AsyncSessionLocal = async_sessionmaker(
    bind=async_engine,
    class_=AsyncSession,
    expire_on_commit=False  # Important: évite le lazy-loading
implicite en async
)

# Les modèles (Client, LoanProduct, CreditApplication) restent
EXACTEMENT les mêmes.
# Seule la manière de créer les tables et d'exécuter les requêtes
change.

# Création des tables en async :
async def init_db():
    async with async_engine.begin() as conn:
        await conn.run_sync(Base.metadata.create_all)
```

3.3 — JPA (Jakarta Persistence avec Hibernate)

JPA utilise des **annotations** sur les classes Java. La philosophie est similaire à SQLAlchemy ORM, mais plus rigide : tout passe par des entités, le schéma est décrit exclusivement via les annotations.

Points clés : - `@Entity` marque la classe comme table - `@Id` + `@GeneratedValue` = clé primaire auto-incrémentée - `@ManyToOne` / `@OneToMany` = relations avec lazy/eager loading - `@Enumerated(EnumType.STRING)` pour mapper les enums Java vers des ENUM SQL - `persistence.xml` configure la connexion (équivalent de `create_engine`)

```java
// --- Enums métier ---

// EmploymentStatus.java
package com.creditscoring.model;

public enum EmploymentStatus {
    SALARIED,
```

```java
    SELF_EMPLOYED,
    UNEMPLOYED,
    RETIRED,
    STUDENT
}

// RiskLevel.java
package com.creditscoring.model;

public enum RiskLevel {
    LOW,        // score >= 750
    MEDIUM,     // 650-749
    HIGH,       // 550-649
    CRITICAL    // < 550
}

// ApplicationStatus.java
package com.creditscoring.model;

public enum ApplicationStatus {
    PENDING,
    APPROVED,
    REJECTED,
    UNDER_REVIEW
}

// Client.java — JPA Entity
package com.creditscoring.model;

import jakarta.persistence.*;
import java.math.BigDecimal;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

@Entity
@Table(name = "clients")
public class Client {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "first_name", length = 100, nullable = false)
    private String firstName;

    @Column(name = "last_name", length = 100, nullable = false)
    private String lastName;
```

```java
    @Column(length = 255, unique = true, nullable = false)
    private String email;

    @Column(name = "annual_income", precision = 12, scale = 2,
nullable = false)
    private BigDecimal annualIncome;

    // @Enumerated(EnumType.STRING) stocke le NOM de l'enum
("SALARIED")
    // EnumType.ORDINAL stockerait l'index (0, 1, 2) — à éviter
absolument
    @Enumerated(EnumType.STRING)
    @Column(name = "employment_status", nullable = false)
    private EmploymentStatus employmentStatus;

    @Column(name = "credit_history_years", nullable = false)
    private Integer creditHistoryYears = 0;

    @Column(name = "registration_date")
    private LocalDate registrationDate = LocalDate.now();

    // OneToMany: un client a plusieurs demandes de crédit
    // mappedBy pointe vers le champ "client" dans
CreditApplication
    @OneToMany(mappedBy = "client", cascade = CascadeType.ALL,
fetch = FetchType.LAZY)
    private List<CreditApplication> applications = new
ArrayList<>();

    // Constructeur vide obligatoire pour JPA
    public Client() {}

    public Client(String firstName, String lastName, String
email,
                  BigDecimal annualIncome, EmploymentStatus
employmentStatus,
                  int creditHistoryYears) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
        this.annualIncome = annualIncome;
        this.employmentStatus = employmentStatus;
        this.creditHistoryYears = creditHistoryYears;
    }

    // Getters
    public Long getId() { return id; }
```

```java
        public String getFirstName() { return firstName; }
        public String getLastName() { return lastName; }
        public String getEmail() { return email; }
        public BigDecimal getAnnualIncome() { return annualIncome; }
        public EmploymentStatus getEmploymentStatus() { return
employmentStatus; }
        public Integer getCreditHistoryYears() { return
creditHistoryYears; }
        public LocalDate getRegistrationDate() { return
registrationDate; }
        public List<CreditApplication> getApplications() { return
applications; }
    }

    // LoanProduct.java — JPA Entity
    package com.creditscoring.model;

    import jakarta.persistence.*;
    import java.math.BigDecimal;
    import java.util.ArrayList;
    import java.util.List;

    @Entity
    @Table(name = "loan_products")
    public class LoanProduct {

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;

        @Column(length = 20, unique = true, nullable = false)
        private String code;

        @Column(length = 200, nullable = false)
        private String name;

        @Column(name = "max_amount", precision = 14, scale = 2,
nullable = false)
        private BigDecimal maxAmount;

        @Column(name = "base_rate", precision = 5, scale = 2,
nullable = false)
        private BigDecimal baseRate;

        @Column(name = "term_months", nullable = false)
        private Integer termMonths;

        @OneToMany(mappedBy = "product", cascade = CascadeType.ALL,
```

```java
fetch = FetchType.LAZY)
            private List<CreditApplication> applications = new
ArrayList<>();

            public LoanProduct() {}

            public LoanProduct(String code, String name, BigDecimal
maxAmount,
                               BigDecimal baseRate, int termMonths) {
                this.code = code;
                this.name = name;
                this.maxAmount = maxAmount;
                this.baseRate = baseRate;
                this.termMonths = termMonths;
            }

            public Long getId() { return id; }
            public String getCode() { return code; }
            public String getName() { return name; }
            public BigDecimal getMaxAmount() { return maxAmount; }
            public BigDecimal getBaseRate() { return baseRate; }
            public Integer getTermMonths() { return termMonths; }
        }

        // CreditApplication.java — JPA Entity
        package com.creditscoring.model;

        import jakarta.persistence.*;
        import java.math.BigDecimal;
        import java.time.LocalDate;

        @Entity
        @Table(name = "credit_applications")
        public class CreditApplication {

            @Id
            @GeneratedValue(strategy = GenerationType.IDENTITY)
            private Long id;

            // ManyToOne: plusieurs demandes appartiennent à un client
            @ManyToOne(fetch = FetchType.LAZY)
            @JoinColumn(name = "client_id", nullable = false)
            private Client client;

            @ManyToOne(fetch = FetchType.LAZY)
            @JoinColumn(name = "product_id", nullable = false)
            private LoanProduct product;
```

```
            @Column(name = "requested_amount", precision = 14, scale =
2, nullable = false)
            private BigDecimal requestedAmount;

            @Column(name = "credit_score", nullable = false)
            private Integer creditScore;  // 300-850

            @Enumerated(EnumType.STRING)
            @Column(name = "risk_level", nullable = false)
            private RiskLevel riskLevel;

            @Enumerated(EnumType.STRING)
            @Column(nullable = false)
            private ApplicationStatus status = ApplicationStatus.PENDING;

            @Column(name = "interest_rate", precision = 5, scale = 2,
nullable = false)
            private BigDecimal interestRate;

            @Column(name = "application_date")
            private LocalDate applicationDate = LocalDate.now();

            public CreditApplication() {}

            public CreditApplication(Client client, LoanProduct product,
                                     BigDecimal requestedAmount, int
creditScore,
                                     RiskLevel riskLevel, BigDecimal
interestRate) {
                this.client = client;
                this.product = product;
                this.requestedAmount = requestedAmount;
                this.creditScore = creditScore;
                this.riskLevel = riskLevel;
                this.interestRate = interestRate;
            }

            public Long getId() { return id; }
            public Client getClient() { return client; }
            public LoanProduct getProduct() { return product; }
            public BigDecimal getRequestedAmount() { return
requestedAmount; }
            public Integer getCreditScore() { return creditScore; }
            public RiskLevel getRiskLevel() { return riskLevel; }
            public ApplicationStatus getStatus() { return status; }
            public BigDecimal getInterestRate() { return interestRate; }
            public LocalDate getApplicationDate() { return
```

```
applicationDate; }
        }
```

3.4 — Comparaison directe : Mapping des concepts ORM

Concept	SQLAlchemy 2.0	JPA / Hibernate
Déclarer un modèle	`class Client(Base):`	`@Entity class Client {}`
Clé primaire	`Mapped[int] = mapped_column(primary_key=True)`	`@Id @GeneratedValue`
Colonne typée	`Mapped[str] = mapped_column(String(100))`	`@Column(length = 100)`
Colonne décimale	`Mapped[Decimal] = mapped_column(DECIMAL(12,2))`	`@Column(precision=12, scale=2)`
Enum	`Mapped[RiskLevel] = mapped_column(Enum(RiskLevel))`	`@Enumerated(EnumType.STR`
Clé étrangère	`mapped_column(ForeignKey("clients.id"))`	`@JoinColumn(name = "client_id")`
Relation 1-N	`Mapped[list["CreditApplication"]] = relationship()`	`@OneToMany(mappedBy = "client")`
Relation N-1	`Mapped["Client"] = relationship()`	`@ManyToOne @JoinColumn`
Lazy loading	Défaut pour les relations	`fetch = FetchType.LAZY`
Session / Transaction	`Session()` / `with session:`	`EntityManager` / `@Transactional`
Config connexion	`create_engine(url)`	`persistence.xml`

---

4. Couche Query : Écrire des Requêtes

C'est ici que la différence est la plus significative pour un data analyst bancaire. Tu écris des requêtes analytiques (JOINs, GROUP BY, agrégations, CASE WHEN) et tu veux récupérer des données tabulaires.

4.1 — SQLAlchemy 2.0 Core (Nouvelle API `select()`)

La nouvelle API abandonne l'ancien style `session.query(Model)` au profit de `select()` combiné avec `session.execute()`. C'est plus explicite et plus composable.

```
        # queries_sync.py — SQLAlchemy 2.0 Select API — Credit Scoring
        from sqlalchemy import select, func, desc, case, and_, cast,
Float

        # ---------- QUERY 1: Toutes les demandes avec détails client +
produit ----------
```

```
        # Ancien style (déprécié) :
session.query(CreditApplication).all()
        # Nouveau style :
        stmt_all = (
            select(
                Client.first_name,
                Client.last_name,
                Client.annual_income,
                Client.employment_status,
                LoanProduct.name.label("product_name"),
                LoanProduct.base_rate,
                CreditApplication.requested_amount,
                CreditApplication.credit_score,
                CreditApplication.risk_level,
                CreditApplication.status,
                CreditApplication.interest_rate,
                CreditApplication.application_date
            )
            .join(CreditApplication.client)
            .join(CreditApplication.product)
            .order_by(desc(CreditApplication.application_date))
        )

        with SessionLocal() as session:
            result = session.execute(stmt_all)
            rows = result.all()  # Liste de Row tuples (named tuples)
            # Chaque row: row.first_name, row.credit_score,
row.risk_level, etc.


        # ---------- QUERY 2: Taux d'approbation par produit ----------
        # SELECT p.name, COUNT(*), SUM(CASE WHEN status='APPROVED' THEN 1
ELSE 0 END),
        #        SUM(...) / COUNT(*) * 100 AS approval_rate
        stmt_approval = (
            select(
                LoanProduct.code,
                LoanProduct.name,

func.count(CreditApplication.id).label("total_applications"),
                func.sum(
                    case(
                        (CreditApplication.status ==
ApplicationStatus.APPROVED, 1),
                        else_=0
                    )
                ).label("approved_count"),
                (
```

```
                    func.sum(
                        case(
                            (CreditApplication.status ==
ApplicationStatus.APPROVED, 1),
                            else_=0
                        )
                    ) * 100.0 / func.count(CreditApplication.id)
                ).label("approval_rate_pct")
            )
            .join(CreditApplication.product)
            .group_by(LoanProduct.id)
            .order_by(desc("approval_rate_pct"))
        )


        # ---------- QUERY 3: Statistiques par niveau de risque
----------
        stmt_risk = (
            select(
                CreditApplication.risk_level,

func.count(CreditApplication.id).label("num_applications"),

func.avg(CreditApplication.credit_score).label("avg_score"),

func.avg(CreditApplication.interest_rate).label("avg_rate"),

func.avg(CreditApplication.requested_amount).label("avg_amount"),

func.min(CreditApplication.credit_score).label("min_score"),

func.max(CreditApplication.credit_score).label("max_score")
            )
            .group_by(CreditApplication.risk_level)
            .order_by(CreditApplication.risk_level)
        )


        # ---------- QUERY 4: Clients haut revenu avec score bas
(anomalies) ----------
        # Subquery: moyenne globale des scores
        global_avg_score =
select(func.avg(CreditApplication.credit_score)).scalar_subquery()

        stmt_anomalies = (
            select(
                Client.first_name,
                Client.last_name,
```

```
                Client.annual_income,
                Client.employment_status,

func.avg(CreditApplication.credit_score).label("avg_score")
            )
            .join(CreditApplication.client)
            .group_by(Client.id)
            .having(
                and_(
                    Client.annual_income > 100000,
# Haut revenu
                    func.avg(CreditApplication.credit_score) <
global_avg_score  # Score sous la moyenne
                )
            )
            .order_by(Client.annual_income.desc())
        )


        # ---------- QUERY 5: Montant total de crédit exposé par statut
----------
        stmt_exposure = (
            select(
                CreditApplication.status,
                func.count(CreditApplication.id).label("count"),

func.sum(CreditApplication.requested_amount).label("total_exposure"),

func.avg(CreditApplication.requested_amount).label("avg_amount")
            )
            .group_by(CreditApplication.status)
        )
```

4.2 — SQLAlchemy 2.0 Async

```
        # queries_async.py — SQLAlchemy 2.0 Async — Credit Scoring
        import asyncio
        from sqlalchemy import select, func, desc
        from sqlalchemy.orm import selectinload

        # Même statements que le sync — seule l'exécution change
        async def get_risk_stats():
            stmt = (
                select(
                    CreditApplication.risk_level,

func.count(CreditApplication.id).label("num_applications"),
```

```
func.avg(CreditApplication.credit_score).label("avg_score"),

func.avg(CreditApplication.interest_rate).label("avg_rate")
                )
                .group_by(CreditApplication.risk_level)
            )

            async with AsyncSessionLocal() as session:
                result = await session.execute(stmt)
                return result.all()

        # Pour les relations (lazy loading interdit en async) :
        async def get_client_with_applications(client_id: int):
            stmt = (
                select(Client)
                .options(selectinload(Client.applications))  # Eager
loading explicite
                .where(Client.id == client_id)
            )
            async with AsyncSessionLocal() as session:
                result = await session.execute(stmt)
                return result.scalar_one_or_none()

        # Exécution
        # asyncio.run(get_risk_stats())
```

Point critique async : En mode asynchrone, le **lazy loading implicite** est **interdit**. Si tu accèdes à `client.applications` sans avoir fait un eager load, SQLAlchemy lève une `MissingGreenlet` exception. Tu dois toujours utiliser `selectinload()`, `joinedload()`, ou `subqueryload()` explicitement.

4.3 — JPA (JPQL et Criteria API)

JPQL est le langage de requête de JPA. Il ressemble à SQL mais opère sur les **entités** (noms de classes) plutôt que sur les **tables**.

```
        // queries_jpa.java — JPA JPQL Queries — Credit Scoring
        import jakarta.persistence.*;
        import java.util.List;

        EntityManagerFactory emf =
Persistence.createEntityManagerFactory("credit-scoring-pu");
        EntityManager em = emf.createEntityManager();


        // ---------- QUERY 1: Toutes les demandes avec détails
----------
        TypedQuery<Object[]> q1 = em.createQuery(
            """
```

```
            SELECT c.firstName, c.lastName, c.annualIncome,
                   p.name, ca.requestedAmount, ca.creditScore,
                   ca.riskLevel, ca.status, ca.interestRate
            FROM CreditApplication ca
            JOIN ca.client c
            JOIN ca.product p
            ORDER BY ca.applicationDate DESC
            """, Object[].class
        );
        List<Object[]> allApps = q1.getResultList();


        // ---------- QUERY 2: Taux d'approbation par produit ----------
        TypedQuery<Object[]> q2 = em.createQuery(
            """
            SELECT p.code, p.name,
                   COUNT(ca),
                   SUM(CASE WHEN ca.status = 'APPROVED' THEN 1 ELSE 0
END),
                   SUM(CASE WHEN ca.status = 'APPROVED' THEN 1.0 ELSE 0.0
END)
                       / COUNT(ca) * 100
            FROM CreditApplication ca
            JOIN ca.product p
            GROUP BY p.id
            ORDER BY SUM(CASE WHEN ca.status = 'APPROVED' THEN 1.0 ELSE
0.0 END)
                       / COUNT(ca) DESC
            """, Object[].class
        );


        // ---------- QUERY 3: Stats par niveau de risque ----------
        TypedQuery<Object[]> q3 = em.createQuery(
            """
            SELECT ca.riskLevel,
                   COUNT(ca),
                   AVG(ca.creditScore),
                   AVG(ca.interestRate),
                   AVG(ca.requestedAmount),
                   MIN(ca.creditScore),
                   MAX(ca.creditScore)
            FROM CreditApplication ca
            GROUP BY ca.riskLevel
            """, Object[].class
        );
```

```
        // ---------- QUERY 4: Anomalies — haut revenu / score bas
----------
        TypedQuery<Object[]> q4 = em.createQuery(
            """
            SELECT c.firstName, c.lastName, c.annualIncome,
AVG(ca.creditScore)
            FROM CreditApplication ca
            JOIN ca.client c
            GROUP BY c.id
            HAVING c.annualIncome > 100000
               AND AVG(ca.creditScore) < (SELECT AVG(ca2.creditScore)
FROM CreditApplication ca2)
            ORDER BY c.annualIncome DESC
            """, Object[].class
        );


        em.close();
        emf.close();
```

4.4 — JDBC (SQL Natif)

JDBC est le plus bas niveau. Tu écris du SQL pur, tu gères les connexions, tu itères sur les `ResultSet`. C'est ce que DFlib utilise en interne.

```
        // queries_jdbc.java — JDBC Raw SQL — Credit Scoring
        import java.sql.*;

        String url = "jdbc:mysql://mysql:3306/credit_scoring_db";
        String user = "analyst";
        String password = "analyst";

        // ---------- QUERY 1: Toutes les demandes ----------
        String sqlAll = """
            SELECT c.first_name, c.last_name, c.annual_income,
c.employment_status,
                   p.name AS product_name, p.base_rate,
                   ca.requested_amount, ca.credit_score, ca.risk_level,
                   ca.status, ca.interest_rate, ca.application_date
            FROM credit_applications ca
            JOIN clients c ON ca.client_id = c.id
            JOIN loan_products p ON ca.product_id = p.id
            ORDER BY ca.application_date DESC
            """;

        try (Connection conn = DriverManager.getConnection(url, user,
password);
             PreparedStatement pstmt = conn.prepareStatement(sqlAll);
```

```java
             ResultSet rs = pstmt.executeQuery()) {

            while (rs.next()) {
                String name = rs.getString("first_name") + " " +
rs.getString("last_name");
                int score = rs.getInt("credit_score");
                String risk = rs.getString("risk_level");
                double rate = rs.getDouble("interest_rate");
            }
        }


        // ---------- QUERY 2: Taux d'approbation par produit ----------
        String sqlApproval = """
            SELECT p.code, p.name,
                   COUNT(*) AS total,
                   SUM(CASE WHEN ca.status = 'APPROVED' THEN 1 ELSE 0
END) AS approved,
                   SUM(CASE WHEN ca.status = 'APPROVED' THEN 1 ELSE 0
END) * 100.0
                      / COUNT(*) AS approval_rate
            FROM credit_applications ca
            JOIN loan_products p ON ca.product_id = p.id
            GROUP BY p.id
            ORDER BY approval_rate DESC
            """;
        // Même pattern: prepareStatement → executeQuery → iterate
ResultSet
```

4.5 — Comparaison directe : Style de requêtes

Aspect	**SQLAlchemy 2.0 Core**	**JPQL (JPA)**	**JDBC**
Langage	Python DSL (`select()`)	String JPQL	String SQL natif
JOIN syntax	`.join(CreditApplication.client)`	`JOIN ca.client c`	`JOIN clients c ON ...`
CASE WHEN	`case((condition, val), else_=)`	`CASE WHEN ... THEN ... END`	SQL standard
Enum handling	Comparaison Python native	String comparison en JPQL	String comparison en SQL
Noms utilisés	Classes + attributs Python	Classes + champs Java	Tables + colonnes SQL
Type safety	Oui (IDE autocomplete)		

Aspect	SQLAlchemy 2.0 Core	JPQL (JPA)	JDBC
		Non (strings)	Non (strings)
Résultat	Row named tuples	`Object[]`	`ResultSet`
Composabilité	Excellente (chaîner)	Limitée	Aucune
CTE / Window funcs	Supporté nativement	Non en JPQL, Native Query	SQL complet

---

5. Couche DataFrame : Analyse de Données

C’est le cœur du workflow data analyst bancaire. On a nos données de scoring, on veut les analyser en mémoire.

5.1 — Pandas : SQL → DataFrame → Analyse

```python
# analysis_pandas.py — Pandas Workflow — Credit Scoring
import pandas as pd
from sqlalchemy import create_engine

engine = create_engine("mysql+pymysql://analyst:analyst@mysql:3306/credit_scoring_db")


# ========== CHARGEMENT DES DONNÉES ==========

df = pd.read_sql(
    """
    SELECT c.first_name, c.last_name, c.annual_income,
           c.employment_status, c.credit_history_years,
           p.code AS product_code, p.name AS product_name,
           p.base_rate, p.term_months,
           ca.requested_amount, ca.credit_score, ca.risk_level,
           ca.status, ca.interest_rate, ca.application_date
    FROM credit_applications ca
    JOIN clients c ON ca.client_id = c.id
    JOIN loan_products p ON ca.product_id = p.id
    """,
    engine,
    parse_dates=["application_date"]
)


# ========== ANALYSE ==========

# --- Taux d'approbation par produit ---
```

```
        approval_by_product = (
            df.groupby("product_name")
            .agg(
                total=("status", "count"),
                approved=("status", lambda x: (x == "APPROVED").sum()),
            )
        )
        approval_by_product["approval_rate"] = (
            (approval_by_product["approved"] /
approval_by_product["total"] * 100).round(2)
        )
        approval_by_product =
approval_by_product.sort_values("approval_rate", ascending=False)


        # --- Statistiques par niveau de risque ---
        risk_stats = (
            df.groupby("risk_level")["credit_score"]
            .agg(["mean", "std", "min", "max", "count"])
            .rename(columns={"mean": "avg_score", "count":
"num_applications"})
            .round(2)
        )


        # --- Distribution des scores (binning calqué sur FICO) ---
        df["score_bracket"] = pd.cut(
            df["credit_score"],
            bins=[300, 550, 650, 750, 850],
            labels=["CRITICAL (300-549)", "HIGH (550-649)", "MEDIUM
(650-749)", "LOW (750-850)"]
        )
        score_distribution =
df["score_bracket"].value_counts().sort_index()


        # --- Spread: différence entre taux accordé et taux de base ---
        df["rate_spread"] = df["interest_rate"] - df["base_rate"]

        spread_by_risk = (
            df.groupby("risk_level")["rate_spread"]
            .agg(["mean", "min", "max"])
            .round(2)
        )


        # --- Exposition totale par statut ---
        exposure = (
```

```
        df.groupby("status")
        .agg(
            count=("requested_amount", "count"),
            total_exposure=("requested_amount", "sum"),
            avg_amount=("requested_amount", "mean")
        )
        .round(2)
    )


    # --- Pivot: Produit × Risque → montant moyen demandé ---
    pivot = df.pivot_table(
        values="requested_amount",
        index="product_name",
        columns="risk_level",
        aggfunc="mean"
    ).round(2)


    # --- Anomalies: haut revenu, score bas ---
    global_avg = df["credit_score"].mean()
    anomalies = (
        df[
            (df["annual_income"] > 100_000) &
            (df["credit_score"] < global_avg)
        ]
        [["first_name", "last_name", "annual_income",
"credit_score", "risk_level"]]
        .drop_duplicates()
        .sort_values("annual_income", ascending=False)
    )


    # --- Top 10 clients par score moyen ---
    top_clients = (
        df.groupby(["first_name", "last_name"])
        .agg(
            avg_score=("credit_score", "mean"),
            num_applications=("status", "count"),
            total_requested=("requested_amount", "sum")
        )
        .sort_values("avg_score", ascending=False)
        .head(10)
        .round(2)
    )
```

5.2 — DFlib : SQL → DataFrame → Analyse

DFlib est la réponse Java à Pandas. La philosophie est similaire : des DataFrames immuables avec des opérations chaînées. La différence majeure est que DFlib est **immuable** — chaque opération retourne un nouveau DataFrame.

```
        // analysis_dflib.java — DFlib Workflow — Credit Scoring
        import org.dflib.*;
        import org.dflib.jdbc.connector.JdbcConnector;
        import org.dflib.Exp;
        import org.dflib.agg.Agg;

        // ========== CONNEXION JDBC ==========
        JdbcConnector connector = new JdbcConnector(
            "jdbc:mysql://mysql:3306/credit_scoring_db",
            "analyst",
            "analyst"
        );


        // ========== CHARGEMENT DES DONNÉES ==========

        DataFrame df = connector
            .sqlLoader("""
                SELECT c.first_name, c.last_name, c.annual_income,
                       c.employment_status, c.credit_history_years,
                       p.code AS product_code, p.name AS product_name,
                       p.base_rate, p.term_months,
                       ca.requested_amount, ca.credit_score,
ca.risk_level,
                       ca.status, ca.interest_rate, ca.application_date
                FROM credit_applications ca
                JOIN clients c ON ca.client_id = c.id
                JOIN loan_products p ON ca.product_id = p.id
            """)
            .load();

        System.out.println(df.head(5));
        System.out.println("Shape: " + df.height() + " rows x " +
df.width() + " cols");


        // ========== ANALYSE ==========

        // --- Statistiques par niveau de risque ---
        DataFrame riskStats = df
            .group("risk_level")
            .agg(
```

```
                Exp.$int("credit_score").avg().as("avg_score"),
                Exp.$int("credit_score").min().as("min_score"),
                Exp.$int("credit_score").max().as("max_score"),
                Exp.$decimal("interest_rate").avg().as("avg_rate"),
                Exp.$decimal("requested_amount").avg().as("avg_amount"),
                Exp.count().as("num_applications")
            )
            .sort(Exp.$decimal("avg_score").desc());


        // --- Colonne calculée: spread (taux accordé - taux de base) ---
        DataFrame withSpread = df.addColumn(
            "rate_spread",
            Exp.$decimal("interest_rate").sub(Exp.$decimal("base_rate"))
        );

        // --- Spread moyen par risque ---
        DataFrame spreadByRisk = withSpread
            .group("risk_level")
            .agg(
                Exp.$decimal("rate_spread").avg().as("avg_spread"),
                Exp.$decimal("rate_spread").min().as("min_spread"),
                Exp.$decimal("rate_spread").max().as("max_spread")
            );


        // --- Filtrage: demandes à haut risque rejetées ---
        DataFrame highRiskRejected = df
            .rows(
                Exp.$str("risk_level").eq("CRITICAL")
                    .and(Exp.$str("status").eq("REJECTED"))
            )
            .select();


        // --- Exposition par statut ---
        DataFrame exposure = df
            .group("status")
            .agg(
                Exp.count().as("count"),
                Exp.
$decimal("requested_amount").sum().as("total_exposure"),
                Exp.$decimal("requested_amount").avg().as("avg_amount")
            );


        // --- Top 10 clients par score moyen ---
        DataFrame topClients = df
```

```
            .group("first_name", "last_name")
            .agg(
                Exp.$int("credit_score").avg().as("avg_score"),
                Exp.count().as("num_applications"),
                Exp.
$decimal("requested_amount").sum().as("total_requested")
            )
            .sort(Exp.$decimal("avg_score").desc())
            .head(10);


        // --- Export CSV ---
        Csv.saver().save(riskStats, "risk_analysis.csv");
```

5.3 — Comparaison directe : Pandas vs DFlib

Opération	Pandas	DFlib
Charger depuis SQL	`pd.read_sql(sql, engine)`	`connector.sqlLoader(sql).loa`
Voir les premières lignes	`df.head(5)`	`df.head(5)`
Sélectionner des colonnes	`df[["col1", "col2"]]`	`df.cols("col1", "col2").sele`
Filtrer les lignes	`df[df["status"] == "APPROVED"]`	`df.rows(Exp. $str("status").eq("APPROVED"`
Filtrer combiné (AND)	`df[(cond1) & (cond2)]`	`df.rows(Exp.$str("x").eq("A" $int("y").gt(5))).select()`
Ajouter une colonne	`df["spread"] = df["rate"] - df["base"]`	`df.addColumn("spread", Exp. $decimal("rate").sub(Exp. $decimal("base")))`
Group by + agg	`df.groupby("risk").agg({"score": "mean"})`	`df.group("risk").agg(Exp. $int("score").avg().as("mean`
Lambda / custom agg	`.agg(lambda x: (x == "APPROVED").sum())`	Pas de lambda — SQL-side ou po
Binning (pd.cut)	`pd.cut(df["score"], bins=[...])`	Pas de cut natif — condition chai
Pivot table	`df.pivot_table(values, index, columns)`	Pas de pivot natif — group + resh
Trier	`df.sort_values("col", ascending=False)`	`df.sort(Exp.$decimal("col").`
Merge/Join	`pd.merge(df1, df2, on="key")`	`df1.join(df2).on("key").sele`
Mutabilité	**Mutable** par défaut	**Immuable** — chaque op retourn nouveau DF
Type system	Dynamique (dtype inféré)	Fort (`$int()`, `$decimal()`, `$str(`

6. Le Workflow Complet du Data Analyst

Python — Le workflow que tu connais

```
# workflow_python.py — Credit Scoring Notebook
from sqlalchemy import create_engine
import pandas as pd

# 1. Connexion
engine = create_engine("mysql+pymysql://analyst:analyst@mysql:
3306/credit_scoring_db")

# 2. Query custom
sql = """
    SELECT c.first_name, c.last_name, c.annual_income,
c.employment_status,
           p.code, p.name AS product, p.base_rate, p.term_months,
           ca.requested_amount, ca.credit_score, ca.risk_level,
           ca.status, ca.interest_rate, ca.application_date
    FROM credit_applications ca
    JOIN clients c ON ca.client_id = c.id
    JOIN loan_products p ON ca.product_id = p.id
"""

# 3. Chargement en DataFrame
df = pd.read_sql(sql, engine, parse_dates=["application_date"])

# 4. Analyse rapide
print(df.describe())
print(df.dtypes)
print(df.shape)

# 5. Transformations
df["rate_spread"] = df["interest_rate"] - df["base_rate"]
df["income_bracket"] = pd.cut(
    df["annual_income"],
    bins=[0, 30000, 60000, 100000, 200000, float("inf")],
    labels=["<30K", "30-60K", "60-100K", "100-200K", "200K+"]
)

# 6. Agrégation
risk_report = (
    df.groupby("risk_level")
    .agg(
        avg_score=("credit_score", "mean"),
        avg_rate=("interest_rate", "mean"),
        total_exposure=("requested_amount", "sum"),
        count=("status", "count")
```

```
        )
        .round(2)
    )

    # 7. Export
    risk_report.to_csv("credit_risk_report.csv")
```

Java — Le même workflow avec DFlib

```
    // workflow_java.java — Credit Scoring Notebook (IJava kernel)
    import org.dflib.*;
    import org.dflib.jdbc.connector.JdbcConnector;

    // 1. Connexion
    JdbcConnector connector = new JdbcConnector(
        "jdbc:mysql://mysql:3306/credit_scoring_db", "analyst",
"analyst"
    );

    // 2 + 3. Query + Chargement en DataFrame
    DataFrame df = connector.sqlLoader("""
        SELECT c.first_name, c.last_name, c.annual_income,
c.employment_status,
               p.code, p.name AS product, p.base_rate, p.term_months,
               ca.requested_amount, ca.credit_score, ca.risk_level,
               ca.status, ca.interest_rate, ca.application_date
        FROM credit_applications ca
        JOIN clients c ON ca.client_id = c.id
        JOIN loan_products p ON ca.product_id = p.id
    """).load();

    // 4. Analyse rapide
    System.out.println(df.head(5));
    System.out.println("Shape: " + df.height() + " rows x " +
df.width() + " cols");
    System.out.println("Columns: " + df.getColumnsIndex());

    // 5. Transformation
    DataFrame enriched = df.addColumn(
        "rate_spread",
        Exp.$decimal("interest_rate").sub(Exp.$decimal("base_rate"))
    );

    // 6. Agrégation
    DataFrame riskReport = enriched
        .group("risk_level")
        .agg(
            Exp.$int("credit_score").avg().as("avg_score"),
```

```
                Exp.$decimal("interest_rate").avg().as("avg_rate"),
                Exp.
$decimal("requested_amount").sum().as("total_exposure"),
                Exp.count().as("count")
            )
            .sort(Exp.$decimal("avg_score").desc());

        // 7. Export
        Csv.saver().save(riskReport, "credit_risk_report.csv");
```

---

7. Migrations : Alembic vs Flyway

7.1 — Alembic (Python)

```
        # Installation et initialisation
        uv add alembic
        alembic init migrations
        # Configurer alembic.ini : sqlalchemy.url = mysql+pymysql://
analyst:analyst@mysql:3306/credit_scoring_db
        # Configurer migrations/env.py : target_metadata = Base.metadata

        # $ alembic revision --autogenerate -m "create credit scoring
tables"
        # Fichier généré: migrations/versions/
xxxx_create_credit_scoring_tables.py
        def upgrade():
            op.create_table('clients',
                sa.Column('id', sa.Integer(), autoincrement=True,
nullable=False),
                sa.Column('first_name', sa.String(100), nullable=False),
                sa.Column('last_name', sa.String(100), nullable=False),
                sa.Column('email', sa.String(255), nullable=False),
                sa.Column('annual_income', sa.DECIMAL(12, 2),
nullable=False),
                sa.Column('employment_status',
sa.Enum('SALARIED','SELF_EMPLOYED','UNEMPLOYED','RETIRED','STUDENT'),
nullable=False),
                sa.Column('credit_history_years', sa.Integer(),
nullable=False),
                sa.Column('registration_date', sa.Date(),
nullable=False),
                sa.PrimaryKeyConstraint('id'),
                sa.UniqueConstraint('email')
            )
            op.create_table('loan_products',
                sa.Column('id', sa.Integer(), autoincrement=True,
nullable=False),
```

```
                sa.Column('code', sa.String(20), nullable=False),
                sa.Column('name', sa.String(200), nullable=False),
                sa.Column('max_amount', sa.DECIMAL(14, 2),
nullable=False),
                sa.Column('base_rate', sa.DECIMAL(5, 2), nullable=False),
                sa.Column('term_months', sa.Integer(), nullable=False),
                sa.PrimaryKeyConstraint('id'),
                sa.UniqueConstraint('code')
            )
            op.create_table('credit_applications',
                sa.Column('id', sa.Integer(), autoincrement=True,
nullable=False),
                sa.Column('client_id', sa.Integer(), nullable=False),
                sa.Column('product_id', sa.Integer(), nullable=False),
                sa.Column('requested_amount', sa.DECIMAL(14, 2),
nullable=False),
                sa.Column('credit_score', sa.Integer(), nullable=False),
                sa.Column('risk_level',
sa.Enum('LOW','MEDIUM','HIGH','CRITICAL'), nullable=False),
                sa.Column('status',
sa.Enum('PENDING','APPROVED','REJECTED','UNDER_REVIEW'),
nullable=False),
                sa.Column('interest_rate', sa.DECIMAL(5, 2),
nullable=False),
                sa.Column('application_date', sa.Date(), nullable=False),
                sa.ForeignKeyConstraint(['client_id'], ['clients.id']),
                sa.ForeignKeyConstraint(['product_id'],
['loan_products.id']),
                sa.PrimaryKeyConstraint('id')
            )

        def downgrade():
            op.drop_table('credit_applications')
            op.drop_table('loan_products')
            op.drop_table('clients')
```

```
alembic upgrade head    # Appliquer
alembic history         # Voir l'historique
alembic downgrade -1    # Rollback
```

7.2 — Flyway (Java)

```
-- V1__create_clients.sql
CREATE TABLE clients (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(100) NOT NULL,
    last_name VARCHAR(100) NOT NULL,
    email VARCHAR(255) NOT NULL UNIQUE,
```

```
            annual_income DECIMAL(12,2) NOT NULL,
            employment_status
ENUM('SALARIED','SELF_EMPLOYED','UNEMPLOYED','RETIRED','STUDENT')
NOT NULL,
            credit_history_years INT NOT NULL DEFAULT 0,
            registration_date DATE NOT NULL DEFAULT (CURRENT_DATE)
        );

        -- V2__create_loan_products.sql
        CREATE TABLE loan_products (
            id BIGINT AUTO_INCREMENT PRIMARY KEY,
            code VARCHAR(20) NOT NULL UNIQUE,
            name VARCHAR(200) NOT NULL,
            max_amount DECIMAL(14,2) NOT NULL,
            base_rate DECIMAL(5,2) NOT NULL,
            term_months INT NOT NULL
        );

        -- V3__create_credit_applications.sql
        CREATE TABLE credit_applications (
            id BIGINT AUTO_INCREMENT PRIMARY KEY,
            client_id BIGINT NOT NULL,
            product_id BIGINT NOT NULL,
            requested_amount DECIMAL(14,2) NOT NULL,
            credit_score INT NOT NULL,
            risk_level ENUM('LOW','MEDIUM','HIGH','CRITICAL') NOT NULL,
            status ENUM('PENDING','APPROVED','REJECTED','UNDER_REVIEW')
NOT NULL DEFAULT 'PENDING',
            interest_rate DECIMAL(5,2) NOT NULL,
            application_date DATE NOT NULL DEFAULT (CURRENT_DATE),
            FOREIGN KEY (client_id) REFERENCES clients(id),
            FOREIGN KEY (product_id) REFERENCES loan_products(id)
        );

        mvn flyway:migrate    # Appliquer
        mvn flyway:info       # Statut
        mvn flyway:clean      # DROP ALL (dangereux)
```

7.3 — Comparaison : Alembic vs Flyway

Aspect	Alembic	Flyway
Autogenerate	Oui (compare modèles vs DB)	Non (SQL écrit à la main)
Format migration	Python (opérations op.)	SQL pur
Nommage	Hash + message	`V{n}__{desc}.sql` strict
Rollback	`downgrade()` custom	Payant (Community: forward only)
Table de tracking	`alembic_version`	`flyway_schema_history`

Aspect	Alembic	Flyway
Intégration ORM	SQLAlchemy natif	Indépendant de l'ORM

8. Environnement Docker

```yaml
# docker-compose.yml
version: "3.9"

services:
  mysql:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: credit_scoring_db
      MYSQL_USER: analyst
      MYSQL_PASSWORD: analyst
    ports:
      - "3306:3306"
    volumes:
      - mysql_data:/var/lib/mysql
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
      interval: 10s
      retries: 5

  adminer:
    image: adminer:latest
    ports:
      - "8080:8080"
    depends_on:
      - mysql

  jupyter:
    build:
      context: .
      dockerfile: Dockerfile.jupyter
    ports:
      - "8888:8888"
    volumes:
      - ./notebooks:/home/jovyan/work
      - ./src:/home/jovyan/src
    depends_on:
      mysql:
        condition: service_healthy
    environment:
      - DATABASE_URL=mysql+pymysql://analyst:analyst@mysql:3306/
credit_scoring_db
```

```
volumes:
  mysql_data:
```

```dockerfile
# Dockerfile.jupyter — Python + Java kernels
FROM jupyter/minimal-notebook:latest

USER root

# Java 21 + Maven
RUN apt-get update && \
    apt-get install -y openjdk-21-jdk maven && \
    apt-get clean

# IJava kernel (Java dans Jupyter)
RUN curl -L https://github.com/SpencerPark/IJava/releases/
download/v1.3.0/ijava-1.3.0.zip \
    -o /tmp/ijava.zip && \
    unzip /tmp/ijava.zip -d /tmp/ijava && \
    cd /tmp/ijava && python3 install.py --sys-prefix && \
    rm -rf /tmp/ijava*

USER $NB_UID

# Python deps via uv
COPY pyproject.toml /home/jovyan/
RUN pip install uv && \
    cd /home/jovyan && \
    uv pip install --system -r pyproject.toml
```

```toml
# pyproject.toml
[project]
name = "credit-scoring-analyst"
version = "0.1.0"
requires-python = ">=3.12"
dependencies = [
    "sqlalchemy[asyncio]>=2.0",
    "pymysql",
    "asyncmy",
    "pandas>=2.2",
    "alembic",
    "jupyterlab",
    "matplotlib",
    "cryptography",
]
```

```xml
<!-- pom.xml -->
<project>
    <modelVersion>4.0.0</modelVersion>
```

```xml
    <groupId>com.creditscoring</groupId>
    <artifactId>credit-scoring-analyst</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <java.version>21</java.version>
        <dflib.version>1.0.0-M22</dflib.version>
    </properties>

    <dependencies>
        <!-- DFlib Core + JDBC + CSV -->
        <dependency>
            <groupId>org.dflib</groupId>
            <artifactId>dflib</artifactId>
            <version>${dflib.version}</version>
        </dependency>
        <dependency>
            <groupId>org.dflib</groupId>
            <artifactId>dflib-jdbc</artifactId>
            <version>${dflib.version}</version>
        </dependency>
        <dependency>
            <groupId>org.dflib</groupId>
            <artifactId>dflib-csv</artifactId>
            <version>${dflib.version}</version>
        </dependency>

        <!-- MySQL JDBC Driver -->
        <dependency>
            <groupId>com.mysql</groupId>
            <artifactId>mysql-connector-j</artifactId>
            <version>8.3.0</version>
        </dependency>

        <!-- JPA + Hibernate -->
        <dependency>
            <groupId>jakarta.persistence</groupId>
            <artifactId>jakarta.persistence-api</artifactId>
            <version>3.1.0</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate.orm</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>6.4.0.Final</version>
        </dependency>

        <!-- Flyway -->
        <dependency>
```

```xml
                <groupId>org.flywaydb</groupId>
                <artifactId>flyway-core</artifactId>
                <version>10.0.0</version>
            </dependency>
            <dependency>
                <groupId>org.flywaydb</groupId>
                <artifactId>flyway-mysql</artifactId>
                <version>10.0.0</version>
            </dependency>
        </dependencies>
    </project>
```

Récapitulatif — Ce qu'il faut retenir

En tant que data analyst bancaire venant de Python :

1. **Ton workflow ne change pas** : SQL → DataFrame → Analyse → Export. Le domaine change (credit scoring au lieu d'académique), mais le pattern reste identique.

2. **SQLAlchemy 2.0** gère nativement les **Enum Python** (`RiskLevel`, `ApplicationStatus`) qui se mappent vers des `ENUM` MySQL. Les colonnes `DECIMAL(12,2)` pour les montants financiers sont typées avec `Mapped[Decimal]`.

3. **Les requêtes bancaires** utilisent beaucoup de `CASE WHEN` (taux d'approbation), de subqueries (anomalies vs moyenne globale), et de GROUP BY multidimensionnels (risque × produit). SQLAlchemy Core exprime tout cela de manière composable avec `case()`, `scalar_subquery()`, et `func`.

4. **DFlib** ne supporte pas nativement les opérations de binning (`pd.cut`) ni les lambda custom dans `agg()`. Pour ces cas, soit on fait le calcul côté SQL, soit on post-traite le DataFrame avec des filtres conditionnels chaînés.

5. **JPA** gère les enums avec `@Enumerated(EnumType.STRING)` — toujours utiliser `STRING`, jamais `ORDINAL` (qui stocke l'index et casse si on réordonne l'enum).

6. **Flyway** : les migrations contiennent les `ENUM()` MySQL en dur dans le DDL. Si on ajoute une valeur d'enum, il faut une nouvelle migration `ALTER TABLE ... MODIFY COLUMN`.