Data Engineering Fundamentals pour Data Analyst

Vue d'Ensemble

En tant que Data Analyst avec un background en backend development et ETL (Apache Beam, BigQuery, MongoDB), tu dois maîtriser les concepts de Data Engineering pour: - Comprendre comment les données arrivent dans tes outils d'analyse - Optimiser les pipelines de données - Collaborer efficacement avec les Data Engineers - Construire des solutions de bout en bout

---

1. Architecture des Données

Data Pipeline - Vue Générale

```
   SOURCES     →       ETL        →     STORAGE      →      ANALYSE
                     PIPELINE
 - Core Bank       - Extract          - Data Lake        - Power BI
 - CRM             - Transform        - Data WH          - Python
 - APIs            - Load             - BigQuery         - SQL
 - Files
```

Batch vs Streaming

Aspect	Batch Processing	Stream Processing
Timing	Périodique (horaire, quotidien)	Temps réel / quasi-réel
Volume	Large volumes	Événements individuels
Latence	Minutes à heures	Secondes à millisecondes
Outils	Apache Beam, Spark, Airflow	Kafka, Flink, Beam
Cas d'usage	Reporting, ML training	Alertes fraude, dashboards live

---

2. ETL vs ELT

ETL (Extract-Transform-Load)

```
Source → [Transform] → Destination

Transformation AVANT chargement
- Nettoyage des données
- Agrégations
- Jointures

Avantages:
- Données propres dans le warehouse
- Moins de stockage nécessaire

Outils: Apache Beam, Informatica, Talend
```

ELT (Extract-Load-Transform)

```
Source → Destination → [Transform]

Chargement brut, transformation APRÈS
- Utilise la puissance du data warehouse
- Transformation en SQL

Avantages:
- Plus rapide à charger
- Flexibilité pour nouvelles transformations
- Exploite BigQuery/Snowflake

Outils: dbt, BigQuery, Snowflake
```

---

3. Apache Beam (ton expérience)

Concepts Clés

```python
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions

# Pipeline basique
with beam.Pipeline(options=PipelineOptions()) as pipeline:
    (
        pipeline
        | 'Read' >> beam.io.ReadFromText('input.csv')
        | 'Parse' >> beam.Map(parse_csv_line)
        | 'Filter' >> beam.Filter(lambda x: x['montant'] > 0)
        | 'Transform' >> beam.Map(calculate_metrics)
        | 'Write' >> beam.io.WriteToBigQuery(
            'project:dataset.table',
            schema='nom:STRING,montant:FLOAT,date:DATE'
        )
    )
```

PCollections et Transforms

```python
# PCollection = Collection distribuée immuable

# Transforms de base
| beam.Map(fn)            # 1 élément → 1 élément
| beam.FlatMap(fn)        # 1 élément → 0..N éléments
| beam.Filter(fn)         # Filtrage conditionnel
| beam.GroupByKey()       # Regroupement par clé
| beam.CombinePerKey(fn) # Agrégation par groupe
| beam.CoGroupByKey()     # Join de plusieurs PCollections
```

Exemple: MongoDB → BigQuery (ton cas d'usage)

```python
import apache_beam as beam
from apache_beam.io.mongodbio import ReadFromMongoDB
```

```python
def transform_document(doc):
    """Transforme un document MongoDB pour BigQuery"""
    return {
        'client_id': str(doc['_id']),
        'nom': doc.get('nom', ''),
        'email': doc.get('email', ''),
        'transactions': len(doc.get('transactions', [])),
        'total_montant': sum(t['montant'] for t in doc.get('transactions', [])),
        'created_at': doc.get('created_at').isoformat() if doc.get('created_at') else None
    }

with beam.Pipeline(options=options) as p:
    (
        p
        | 'ReadMongo' >> ReadFromMongoDB(
            uri='mongodb://localhost:27017',
            db='banking',
            coll='clients'
        )
        | 'Transform' >> beam.Map(transform_document)
        | 'WriteBQ' >> beam.io.WriteToBigQuery(
            'project:dataset.clients',
            schema='client_id:STRING,nom:STRING,email:STRING,'
                   'transactions:INTEGER,total_montant:FLOAT,created_at:TIMESTAMP',
            write_disposition=beam.io.BigQueryDisposition.WRITE_TRUNCATE
        )
    )
```

Windowing (pour Streaming)

```python
# Fenêtrage temporel pour données streaming
from apache_beam import window

(
    pipeline
    | 'Read' >> beam.io.ReadFromPubSub(topic='transactions')
    | 'Parse' >> beam.Map(parse_json)
    | 'Window' >> beam.WindowInto(
        window.FixedWindows(60)  # Fenêtres de 60 secondes
    )
    | 'Sum' >> beam.CombinePerKey(sum)
    | 'Write' >> beam.io.WriteToBigQuery(...)
)
```

---

4. BigQuery (ton expérience)

Optimisation des Requêtes

```sql
--  Utiliser le partitionnement
CREATE TABLE project.dataset.transactions
PARTITION BY DATE(date_transaction)
```

```sql
CLUSTER BY client_id, agence_id
AS SELECT * FROM source_table;

--  Sélectionner seulement les colonnes nécessaires
SELECT client_id, SUM(montant)
FROM transactions
WHERE date_transaction >= '2024-01-01'
GROUP BY client_id;

--  Éviter SELECT *
SELECT * FROM large_table;  -- Coûteux!
```

Bonnes Pratiques BigQuery

```sql
-- 1. Utiliser les colonnes partitionnées dans WHERE
WHERE _PARTITIONDATE = '2024-01-15'

-- 2. Utiliser APPROX_COUNT_DISTINCT pour les estimations
SELECT APPROX_COUNT_DISTINCT(client_id) FROM transactions;

-- 3. Éviter les self-joins massifs, utiliser les window functions
SELECT
    client_id,
    montant,
    SUM(montant) OVER (PARTITION BY client_id ORDER BY date) as cumul
FROM transactions;

-- 4. Utiliser des vues matérialisées pour les agrégations fréquentes
CREATE MATERIALIZED VIEW mv_daily_stats AS
SELECT
    DATE(date_transaction) as jour,
    COUNT(*) as nb_transactions,
    SUM(montant) as total
FROM transactions
GROUP BY jour;
```

Scheduling avec BigQuery

```sql
-- Requête planifiée
-- Via Console ou API
CREATE SCHEDULED QUERY
    daily_aggregation
    SCHEDULE "every day 02:00"
    AS
SELECT
    CURRENT_DATE() as date_rapport,
    COUNT(DISTINCT client_id) as clients_actifs,
    SUM(montant) as volume_total
FROM transactions
WHERE DATE(date_transaction) = CURRENT_DATE() - 1;
```

---

5. Data Warehouse vs Data Lake

Data Warehouse

```
            DATA WAREHOUSE

    Données structurées
    Schéma défini (schema-on-write)
    Optimisé pour requêtes analytiques
    SQL standard

  Exemples: BigQuery, Snowflake, Redshift
```

Data Lake

```
              DATA LAKE

    Données brutes (structurées ou non)
    Schéma flexible (schema-on-read)
    Stockage économique
    Formats: Parquet, JSON, CSV, images

  Exemples: GCS, S3, Azure Data Lake
```

Data Lakehouse (moderne)

Combine les avantages des deux: - Stockage économique du Data Lake - Performance et gouvernance du Data Warehouse - Exemples: Databricks, BigQuery avec GCS

---

6. Formats de Fichiers

Comparaison

Format	Type	Compression	Cas d'usage
CSV	Row-based	Non	Export simple, interopérabilité
JSON	Row-based	Non	APIs, données semi-structurées
Parquet	Column-based	Oui	Analytics, BigQuery, Spark
Avro	Row-based	Oui	Streaming, Kafka
ORC	Column-based	Oui	Hive, analytics

Parquet - Le Standard Analytics

```python
import pandas as pd
import pyarrow.parquet as pq

# Écrire en Parquet
df.to_parquet('data.parquet', compression='snappy')
```

```python
# Lire (seulement les colonnes nécessaires!)
df = pd.read_parquet('data.parquet', columns=['client_id', 'montant'])

# Partitionnement
df.to_parquet(
    'data/',
    partition_cols=['year', 'month'],
    compression='snappy'
)
# Crée: data/year=2024/month=01/part-0.parquet
```

---

7. Orchestration (Airflow)

Concepts de Base

```python
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.providers.google.cloud.operators.bigquery import BigQueryInsertJobOperator
from datetime import datetime, timedelta

default_args = {
    'owner': 'data_team',
    'retries': 3,
    'retry_delay': timedelta(minutes=5),
}

with DAG(
    'daily_etl_banking',
    default_args=default_args,
    description='ETL quotidien des données bancaires',
    schedule_interval='0 2 * * *',  # Tous les jours à 2h
    start_date=datetime(2024, 1, 1),
    catchup=False,
) as dag:

    extract_task = PythonOperator(
        task_id='extract_from_source',
        python_callable=extract_data,
    )

    transform_task = PythonOperator(
        task_id='transform_data',
        python_callable=transform_data,
    )

    load_task = BigQueryInsertJobOperator(
        task_id='load_to_bigquery',
        configuration={
            'load': {
                'sourceUris': ['gs://bucket/data/*.parquet'],
                'destinationTable': {
```

```
                'projectId': 'project',
                'datasetId': 'dataset',
                'tableId': 'table'
            },
            'sourceFormat': 'PARQUET',
        }
    },
)

# Définir les dépendances
extract_task >> transform_task >> load_task
```

---

8. Data Quality

Dimensions de la Qualité

Dimension	Description	Vérification
Complétude	Pas de valeurs manquantes critiques	`COUNT(*)`, `NULL` checks
Exactitude	Valeurs correctes	Règles métier, ranges
Cohérence	Pas de contradictions	Cross-checks, foreign keys
Fraîcheur	Données à jour	Timestamps, latence
Unicité	Pas de doublons	`DISTINCT`, clés primaires

Framework de Data Quality

```python
def data_quality_checks(df, config):
    """Vérifie la qualité des données"""
    issues = []

    # 1. Complétude
    for col in config['required_columns']:
        null_pct = df[col].isnull().mean()
        if null_pct > config['max_null_pct']:
            issues.append(f"COMPLETENESS: {col} has {null_pct:.1%} nulls")

    # 2. Unicité
    for col in config['unique_columns']:
        dup_count = df[col].duplicated().sum()
        if dup_count > 0:
            issues.append(f"UNIQUENESS: {col} has {dup_count} duplicates")

    # 3. Exactitude (ranges)
    for col, (min_val, max_val) in config['ranges'].items():
        out_of_range = ~df[col].between(min_val, max_val)
        if out_of_range.any():
            issues.append(f"ACCURACY: {col} has {out_of_range.sum()} out of range")

    # 4. Fraîcheur
    if 'date_column' in config:
        max_date = df[config['date_column']].max()
```

```
        if (pd.Timestamp.now() - max_date).days > config['max_age_days']:
            issues.append(f"FRESHNESS: Data is {(pd.Timestamp.now() - max_date).days} days old")

    return issues

# Configuration
quality_config = {
    'required_columns': ['client_id', 'montant', 'date'],
    'unique_columns': ['transaction_id'],
    'ranges': {
        'montant': (0, 10000000),
        'taux': (0, 100)
    },
    'date_column': 'date',
    'max_age_days': 1,
    'max_null_pct': 0.05
}

issues = data_quality_checks(df, quality_config)
if issues:
    raise DataQualityError(f"Quality issues: {issues}")
```

Great Expectations (Framework)

```
import great_expectations as gx

# Créer une suite de tests
context = gx.get_context()

validator = context.sources.pandas_default.read_dataframe(df)

# Définir les expectations
validator.expect_column_values_to_not_be_null("client_id")
validator.expect_column_values_to_be_unique("transaction_id")
validator.expect_column_values_to_be_between("montant", min_value=0, max_value=10000000)
validator.expect_column_values_to_match_regex("email", r"^[\w\.-]+@[\w\.-]+\.\w+$")

# Valider
results = validator.validate()
```

---

9. CDC (Change Data Capture)

Concept

CDC capture les modifications (INSERT, UPDATE, DELETE) dans une base source pour les répliquer.

```
Source DB                    CDC                    Target

  INSERT    →    Transaction Log        →    INSERT
  UPDATE    →    (Binary Log/WAL)       →    UPDATE
  DELETE    →                           →    DELETE
```

Patterns CDC

Pattern	Description	Latence
Timestamp	Colonne `updated_at`	Minutes
Trigger	Triggers DB	Temps réel
Log-based	Lecture du WAL/Binlog	Temps réel

Exemple: CDC avec Timestamps

```sql
-- Table source avec colonnes CDC
CREATE TABLE clients (
    id INT PRIMARY KEY,
    nom VARCHAR(100),
    email VARCHAR(100),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    is_deleted BOOLEAN DEFAULT FALSE
);

-- Requête CDC (extraire les changements depuis la dernière synchro)
SELECT *
FROM clients
WHERE updated_at > @last_sync_timestamp
ORDER BY updated_at;
```

---

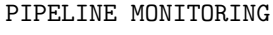
10. Monitoring et Observabilité

Métriques Clés d'un Pipeline

```python
# Métriques à surveiller
metrics = {
    'latency': 'Temps entre source et destination',
    'throughput': 'Nombre de records par minute',
    'error_rate': 'Pourcentage d'erreurs',
    'data_freshness': 'Âge des données les plus récentes',
    'row_count': 'Nombre de lignes traitées',
    'data_quality_score': 'Score de qualité agrégé'
}

# Alertes
alerts = {
    'pipeline_failure': 'Le pipeline a échoué',
    'data_delay': 'Données en retard de plus de X heures',
    'row_count_anomaly': 'Variation anormale du volume',
    'quality_threshold': 'Score qualité sous le seuil'
}
```

Exemple de Dashboard Pipeline



```
Status:   HEALTHY          Last Run: 2024-01-15 02:15


Latency: 15 min        Throughput: 50K rec/min
         75%             85%

Error Rate: 0.02%      Data Freshness: 2h
         5%              60%


Recent Runs
  2024-01-15 02:00 - 1.2M rows - 12 min
  2024-01-14 02:00 - 1.1M rows - 11 min
  2024-01-13 02:00 - 1.3M rows - 25 min (slow)
```

---

11. Questions d'Entretien Data Engineering

Concepts

Q: Quelle est la différence entre ETL et ELT? > ETL transforme avant le chargement (dans le pipeline), ELT charge d'abord les données brutes puis transforme dans le data warehouse (avec SQL/dbt).

Q: Pourquoi utiliser Parquet plutôt que CSV? > Parquet est columnar (lecture partielle), compressé, avec schéma intégré. Plus efficace pour l'analytics.

Q: Qu'est-ce que le CDC? > Change Data Capture - technique pour capturer les modifications (INSERT/UPDATE/DELETE) d'une source pour les répliquer vers une cible.

Apache Beam

Q: Qu'est-ce qu'une PCollection dans Beam? > Une collection distribuée immuable d'éléments, l'abstraction de base pour les données dans un pipeline Beam.

Q: Quelle est la différence entre Map et FlatMap? > Map: 1 élément → 1 élément. FlatMap: 1 élément → 0, 1, ou plusieurs éléments.

BigQuery

Q: Comment optimiser une requête BigQuery? > Utiliser le partitionnement, le clustering, sélectionner seulement les colonnes nécessaires, éviter SELECT *.

Q: Qu'est-ce que le clustering dans BigQuery? > Organisation physique des données par colonnes spécifiées pour accélérer les requêtes filtrées sur ces colonnes.

---

12. Projet Pratique: Pipeline Complet

Architecture

```
MySQL (Source) → Apache Beam → BigQuery → Looker Studio
                      ↓
```

```
Data Quality Checks
        ↓
Alerting (Email/Slack)
```

Code Complet

```python
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions
import mysql.connector
import json
from datetime import datetime

class ExtractFromMySQL(beam.DoFn):
    def __init__(self, config):
        self.config = config

    def setup(self):
        self.conn = mysql.connector.connect(**self.config)

    def process(self, query):
        cursor = self.conn.cursor(dictionary=True)
        cursor.execute(query)
        for row in cursor:
            yield row

    def teardown(self):
        self.conn.close()

class TransformTransaction(beam.DoFn):
    def process(self, row):
        # Nettoyage et transformation
        transformed = {
            'transaction_id': str(row['id']),
            'client_id': str(row['client_id']),
            'montant': float(row['montant']) if row['montant'] else 0.0,
            'date': row['date_transaction'].isoformat() if row['date_transaction'] else None,
            'type': row['type'].upper() if row['type'] else 'UNKNOWN',
            'agence': row['agence_code'],
            'processed_at': datetime.utcnow().isoformat()
        }

        # Validation
        if transformed['montant'] >= 0 and transformed['client_id']:
            yield transformed
        else:
            # Log les erreurs
            yield beam.pvalue.TaggedOutput('errors', {
                'original': row,
                'reason': 'Invalid montant or missing client_id'
            })

class DataQualityCheck(beam.DoFn):
    def process(self, batch):
        total = len(batch)
```

```
        valid = sum(1 for r in batch if r.get('montant', 0) >= 0)
        quality_score = valid / total if total > 0 else 0

        if quality_score < 0.95:
            # Déclencher une alerte
            send_alert(f"Data quality below threshold: {quality_score:.1%}")

        yield from batch

def run_pipeline():
    options = PipelineOptions([
        '--runner=DataflowRunner',
        '--project=my-project',
        '--region=us-central1',
        '--temp_location=gs://my-bucket/temp'
    ])

    mysql_config = {
        'host': 'localhost',
        'user': 'user',
        'password': 'password',
        'database': 'banking'
    }

    with beam.Pipeline(options=options) as p:
        # Extract
        transactions = (
            p
            | 'CreateQuery' >> beam.Create([
                "SELECT * FROM transactions WHERE date_transaction >= CURDATE() - INTERVAL 1 DAY"
            ])
            | 'ExtractMySQL' >> beam.ParDo(ExtractFromMySQL(mysql_config))
        )

        # Transform avec gestion des erreurs
        transformed, errors = (
            transactions
            | 'Transform' >> beam.ParDo(TransformTransaction()).with_outputs('errors', main='valid')
        )

        # Load vers BigQuery
        transformed | 'LoadBigQuery' >> beam.io.WriteToBigQuery(
            'project:dataset.transactions',
            schema='transaction_id:STRING,client_id:STRING,montant:FLOAT,'
                   'date:TIMESTAMP,type:STRING,agence:STRING,processed_at:TIMESTAMP',
            write_disposition=beam.io.BigQueryDisposition.WRITE_APPEND,
            create_disposition=beam.io.BigQueryDisposition.CREATE_IF_NEEDED
        )

        # Log les erreurs
        errors | 'LogErrors' >> beam.io.WriteToText('gs://my-bucket/errors/errors')

if __name__ == '__main__':
    run_pipeline()
```

Résumé des Points Clés

1. **ETL vs ELT:** ETL transforme avant, ELT transforme après le chargement
2. **Apache Beam:** Abstraction unifiée batch/streaming, PCollections immuables
3. **BigQuery:** Partitionnement + Clustering = Performance
4. **Formats:** Parquet pour analytics, Avro pour streaming
5. **Data Quality:** Vérifier complétude, exactitude, cohérence, fraîcheur
6. **CDC:** Capturer les changements pour réplication incrémentale
7. **Monitoring:** Latence, throughput, error rate, freshness