Contents

Chapter 1

Apache Spark

Agence nationale de la Statistique et de la Démographie (ANSD)

École nationale de la Statistique et de l'Analyse économique Pierre Ndiaye (ENSAE)

Présentation du projet du groupe 7 – Année académique 2025-2026

Spark : Traitement des données massives en temps réel

Rédigé par

COMPAORE Bassirou DIAKHATE Khadidiatou DIALLO Aissatou FOGWOUNG DJOUFACK Sarah-Laure FOUMSOU Lawa Prosper

Élèves ingénieurs statisticiens économistes (ISE2)

Sous la supervision de

Mme DIAW Mously

Freelance Senior Data Scientist / ML Engineer

Livre réalisé dans le cadre du cours d'Initiation au Big Data & Cloud Computing.

Chapter 2

Introduction

2.1 Contexte général : l'explosion des données

Au cours des dernières années, la production de données à l'échelle mondiale a connu une croissance exponentielle. Cette évolution est principalement portée par la généralisation des objets connectés (Internet of Things – IoT), l'essor massif des réseaux sociaux, la digitalisation des transactions économiques ainsi que la multiplication des capteurs et systèmes automatisés dans des secteurs variés tels que l'énergie, la santé, la finance ou encore les transports.

Contrairement aux modèles traditionnels, les données ne sont plus produites de manière ponctuelle ou quotidienne, mais de façon **continue et en temps réel**. Chaque seconde, des millions d'événements sont générés : messages, clics, paiements, mesures de capteurs, flux applicatifs, journaux systèmes (logs), etc. Cette transformation marque un changement profond dans la nature des données, désormais caractérisées par des **volumes très élevés**, une **grande diversité de formats** (texte, images, vidéos, flux JSON, données semi-structurées) et une **vitesse de génération particulièrement élevée**.

Face à cette nouvelle réalité, les systèmes classiques de gestion et de traitement des données atteignent rapidement leurs limites, tant en termes de performance que de capacité d'adaptation.

2.2 Limites des approches traditionnelles : SQL et Hadoop

Les bases de données relationnelles classiques telles que MySQL, PostgreSQL ou Oracle ont longtemps constitué la solution de référence pour le stockage et l'analyse des données. Conçues pour des données essentiellement structurées et organisées sous forme de tables, elles reposent généralement sur des architectures centralisées ou faiblement distribuées. Dans ce contexte, leur capacité de montée en charge demeure limitée lorsqu'il s'agit de traiter des volumes massifs de données ou des flux continus en temps réel.

Dans un second temps, l'écosystème Hadoop a apporté une réponse partielle à ces problématiques. Grâce au système de fichiers distribué HDFS, Hadoop permet le stockage de données sur plusieurs machines, tandis que le modèle de calcul MapReduce offre une bonne tolérance aux pannes et une certaine robustesse. Toutefois, Hadoop présente également plusieurs limites importantes :

- des temps de traitement élevés dus aux écritures fréquentes sur disque ;
- une orientation principalement vers le traitement **batch** ;
- une complexité de développement et de configuration non négligeable ;
- une faible adaptation aux besoins de traitement interactif et temps réel.

Ces contraintes ont progressivement mis en évidence la nécessité de solutions plus rapides, plus flexibles et mieux adaptées aux exigences modernes du Big Data.

2.3 Apache Spark : une réponse aux besoins modernes du Big Data

C'est dans ce contexte qu'**Apache Spark** s'est imposé comme une technologie centrale du Big Data. Conçu pour dépasser les limitations du modèle Hadoop MapReduce, Spark repose sur un **modèle de calcul en mémoire**, permettant d'améliorer significativement les performances, en particulier pour les traitements analytiques et itératifs.

Apache Spark se distingue notamment par :

- sa **rapidité**, grâce à l'exécution en mémoire et au parallélisme sur plusieurs cœurs ou plusieurs machines ;

- sa **flexibilité**, en prenant en charge des données structurées, semi-structurées et non structurées ;
- sa **scalabilité**, pouvant fonctionner aussi bien sur une machine unique que sur un cluster distribué ;
- son **écosystème riche**, intégrant le traitement batch, le streaming en temps réel, le machine learning et l'analyse SQL ;
- sa **compatibilité avec les architectures Cloud modernes**, notamment dans les environnements de type Lakehouse.

Dans le cadre de ce projet d'initiation au Big Data et au Cloud Computing, Apache Spark constitue la technologie centrale étudiée. L'objectif est de comprendre, à travers des cas concrets, les principes fondamentaux du traitement distribué de données massives, notamment dans des contextes batch et temps réel.

La suite de ce document présentera successivement les concepts fondamentaux d'Apache Spark (architecture, fonctionnement et principaux modules), ses méthodes de traitement, ainsi que ses forces et limites dans un contexte Big Data moderne.

Chapter 3

Présentation générale d'Apache Spark

3.1 Qu'est-ce qu'Apache Spark ?

Apache Spark est un **framework open source de calcul distribué**, conçu pour le **traitement efficace de données massives (Big Data)**.
Il permet d'analyser de très grands volumes de données en exploitant le **calcul parallèle** sur plusieurs machines, tout en offrant une **exécution rapide grâce au traitement en mémoire**.

Contrairement aux approches classiques basées sur des traitements sur disque, Spark a été pensé pour répondre aux besoins modernes du Big Data : **vitesse**, **flexibilité** et **scalabilité**.

> *Apache Spark unifie le traitement batch, le streaming temps réel, l'analyse SQL et le machine learning au sein d'un même moteur.*

3.2 Philosophie et objectifs de Spark

La conception de Spark repose sur trois principes fondamentaux :

3.2.1 Vitesse

Spark est capable d'exécuter certains traitements **jusqu'à 100 fois plus rapidement que Hadoop MapReduce**, notamment grâce à : - l'exécution des

calculs **en mémoire (in-memory computing)**, - la réduction des écritures disque, - l'optimisation automatique des plans d'exécution.

3.2.2 Facilité d'utilisation

Spark propose des **API simples et expressives** dans plusieurs langages : - **Scala**, **Python**, **Java**, **R** et **SQL**.
Cela permet aux utilisateurs de développer des applications distribuées sans gérer directement la complexité du parallélisme.

3.2.3 Généralité

Spark est une plateforme **polyvalente**, capable de gérer : - le traitement **batch**, - le **streaming temps réel**, - l'analyse **SQL**, - le **machine learning**, - l'analyse de **graphes**.

3.3 Concepts fondamentaux de Spark

3.3.1 Environnement distribué : le cluster

Un **cluster** est un ensemble de machines (nœuds) qui collaborent pour stocker et traiter les données.
Spark fonctionne dans cet environnement distribué afin de répartir le calcul et d'augmenter les performances.

3.3.2 Partitions

Les données sont découpées en **partitions**, chacune étant traitée indépendamment.
Ce découpage permet : - le **traitement parallèle**, - une meilleure utilisation des ressources, - une montée en charge efficace.

3.3.3 RDD et DataFrames

- Les **RDD (Resilient Distributed Datasets)** sont la structure de données historique de Spark, distribuée et tolérante aux pannes.
- Aujourd'hui, les **DataFrames** et **Datasets** sont privilégiés car plus optimisés et plus simples à utiliser, tout en reposant sur les mêmes principes.

3.3.4 Transformations et actions

- Une **transformation** prépare un traitement (ex. `filter`, `map`) mais **n'exécute pas immédiatement** le calcul.
- Une **action** déclenche réellement l'exécution (ex. `count`, `collect`).

Ce mécanisme repose sur le principe de **lazy evaluation**, qui permet à Spark d'optimiser le plan d'exécution.

3.3.5 DAG (Directed Acyclic Graph)

Le **DAG (graphe orienté acyclique)** est un **plan logique d'exécution** construit automatiquement par Spark à partir des transformations définies par l'utilisateur.

Il représente : - l'ordre des opérations à effectuer, - les dépendances entre les transformations, - les étapes nécessaires avant toute exécution réelle.

Le DAG permet à Spark : - d'optimiser l'enchaînement des opérations, - de retarder l'exécution grâce au principe de *lazy evaluation*, - de préparer efficacement la création des stages et des tasks.

Aucune donnée n'est réellement traitée tant qu'une **action** n'est pas appelée.

3.3.6 Shuffle

Le **shuffle** correspond à un **échange de données entre les différentes machines du cluster**.

Il intervient lors d'opérations nécessitant un regroupement ou une redistribution des données, telles que : - `groupBy`, - `join`, - `reduceByKey`, - `orderBy`.

Le shuffle est une opération : - coûteuse en temps, - consommatrice de ressources (réseau et disque).

Spark cherche donc à **minimiser le nombre de shuffles**, car ils ont un impact direct sur les performances globales de l'application.

3.3.7 Stages

Les **stages** sont des **étapes d'exécution** dérivées du DAG.

Un stage regroupe : - un ensemble d'opérations pouvant être exécutées **sans échange de données entre les machines**, - des transformations successives qui ne nécessitent pas de shuffle.

Chaque fois qu'un shuffle est requis, Spark **sépare le DAG en plusieurs stages**.
Ainsi, le nombre de stages dépend directement de la structure des opérations et de la présence de shuffles.

3.3.8 Tasks

Les **tasks** représentent la **plus petite unité d'exécution** dans Spark.

Chaque stage est découpé en plusieurs tasks, et : - chaque task traite **une partition de données**, - les tasks s'exécutent **en parallèle** sur les executors du cluster.

Ce mécanisme permet à Spark : - d'exploiter pleinement le parallélisme, - d'améliorer les performances, - d'assurer une bonne tolérance aux pannes.

3.4 Fonctionnement général d'Apache Spark

Le fonctionnement de Spark peut être compris comme une succession d'étapes logiques allant de l'écriture du programme à l'exécution effective des calculs sur le cluster.

3.4.1 Ecriture du programme

L'utilisateur écrit un programme Spark en utilisant une API (Python, Scala, SQL, etc.) et définit une suite de transformations sur les données.

À ce stade : - aucune donnée n'est encore traitée, - Spark se contente d'enregistrer les opérations demandées.

3.4.2 Construction du DAG

Lorsque l'utilisateur appelle une **action**, Spark analyse l'ensemble des transformations et construit un **DAG** représentant le plan logique d'exécution.

Ce DAG permet à Spark : - d'identifier les dépendances entre les opérations, - d'optimiser l'ordre des calculs, - de détecter les points nécessitant des échanges de données (shuffles).

3.4.3 Découpage en stages

À partir du DAG, Spark divise le plan d'exécution en **stages**.

Chaque stage correspond à : - un ensemble d'opérations pouvant être exécutées localement, - une phase sans échange de données entre machines.

Les frontières entre les stages sont généralement définies par les opérations de shuffle.

3.4.4 Création et exécution des tasks

Chaque stage est ensuite découpé en **tasks**, correspondant aux partitions des données.

Les tasks sont : - distribuées aux executors, - exécutées en parallèle, - supervisées par le driver.

Cette exécution parallèle permet à Spark de traiter efficacement de très grands volumes de données.

3.4.5 Collecte des résultats

Une fois les tasks terminées : - les résultats sont agrégés, - renvoyés au driver si nécessaire, - ou stockés dans un système externe (HDFS, S3, base de données, etc.).

Le traitement Spark est alors considéré comme terminé.

3.5 Architecture de Spark

3.5.1 Le Driver Program

Le **Driver** est le cerveau de l'application Spark. Il : - contient le code principal, - crée la **SparkSession**, - planifie les opérations, - distribue les tâches aux executors, - récupère les résultats.

Le driver **ne traite pas directement les données**, il coordonne l'exécution.

3.5.2 Les Executors

Les **Executors** sont des processus lancés sur les machines du cluster. Ils : - exécutent les tasks envoyées par le driver, - stockent temporairement les données en mémoire, - renvoient les résultats.

Plus le nombre d'executors est élevé, plus le traitement est **parallèle et performant**.

3.5.3 Le Cluster Manager

Le **Cluster Manager** gère l'allocation des ressources (CPU, mémoire).
Spark peut fonctionner avec différents gestionnaires : - **Standalone**, - **YARN**, - **Mesos**, - **Kubernetes**, très utilisé dans les environnements Cloud.

3.6 Principaux modules de Spark

3.6.1 Spark SQL

Module dédié aux données structurées : - exécution de requêtes SQL distribuées, - manipulation de DataFrames/Datasets, - optimisation automatique via le **Catalyst Optimizer**, - lecture de formats comme Parquet, ORC, JSON, CSV.

3.6.2 Spark Streaming

Module de traitement de flux de données en continu : - ingestion depuis Kafka, Kinesis, fichiers streamés, - fonctionnement par **micro-batches**, - adapté aux applications temps réel (monitoring, détection de fraude).

3.6.3 MLlib

Bibliothèque de machine learning distribué : - régression, classification, clustering, - systèmes de recommandation, - pipelines de machine learning à grande échelle.

3.6.4 GraphX

Module spécialisé dans l'analyse de graphes : - représentation de graphes distribués, - algorithmes comme PageRank ou Connected Components, - utile pour l'analyse de réseaux complexes.

Chapter 4

Méthodologie de traitement avec Apache Spark

Apache Spark propose deux grandes approches complémentaires pour le traitement des données massives :
le **traitement Batch (par lots)** et le **traitement Streaming (par flux)**.
Ces deux méthodes répondent à des besoins différents en fonction de la nature des données et des contraintes de temps.

4.1 Traitement Batch (par lots)

Le traitement **Batch** correspond au traitement de **grands volumes de données déjà stockées**.
Les données sont collectées sur une période donnée, puis traitées **en une seule fois**, sous forme de lots complets.

4.1.1 Principe général

- Les données sont préalablement stockées dans des systèmes de stockage distribués
  (HDFS, Amazon S3, bases de données, fichiers CSV/Parquet, etc.).
- Spark lit l'ensemble des données, applique une série de transformations, puis produit un résultat final.
- Le traitement n'est **pas continu** : il démarre à un instant donné et s'arrête une fois le calcul terminé.

4.1.2 Fonctionnement avec Spark

- Spark construit un **DAG (Directed Acyclic Graph)** représentant la chaîne logique des transformations.
- Ce DAG est découpé en **stages**, eux-mêmes composés de **tasks** exécutées en parallèle.
- L'exécution repose sur le **calcul en mémoire**, ce qui améliore fortement les performances par rapport aux approches classiques basées uniquement sur le disque.

4.1.3 Cas d'usage typiques

Le traitement Batch est particulièrement adapté pour : - les pipelines **ETL** (Extraction, Transformation, Loading), - le nettoyage massif de données, - les agrégations statistiques lourdes, - l'entraînement de modèles de **machine learning**, - les analyses historiques où la latence n'est pas critique.

4.2 Traitement Streaming (par flux)

Le traitement **Streaming** est utilisé lorsque les données arrivent **en continu** et doivent être traitées **quasi en temps réel**.

4.2.1 Principe général

- Les données sont produites sous forme de flux continus
  (Kafka, capteurs IoT, logs applicatifs, transactions, événements systèmes).
- Spark traite les données dès leur arrivée, sans attendre la fin d'un lot complet.
- L'objectif principal est de **réagir rapidement** à de nouveaux événements.

4.2.2 Streaming dans Spark : le micro-batch

- Spark Streaming repose sur un modèle de **micro-batch** :
  - le flux continu est découpé en **petits blocs temporels successifs**,
  - chaque micro-lot est traité comme un job batch très rapide.
- Ce modèle permet de combiner :
  - la simplicité du batch,
  - et la réactivité du temps réel.

4.2.3 Capacités offertes

Le traitement Streaming permet notamment : - des **agrégations continues**, - des calculs glissants via des **fenêtres temporelles**, - des mises à jour en temps réel, - la génération d'alertes ou de tableaux de bord dynamiques.

4.2.4 Cas d'usage typiques

Le streaming est indispensable pour : - la détection de fraude, - la surveillance de systèmes, - l'analyse de logs en temps réel, - les systèmes d'alerte, - les applications nécessitant une **faible latence**.

4.3 Comparaison Batch vs Streaming

Critère	Batch	Streaming
Nature des données	Données stockées	Données en continu
Mode de traitement	Par lots complets	Flux (micro-batch)
Latence	Minutes à heures	Millisecondes à secondes
Complexité	Moins élevée	Plus exigeante
Cas d'usage	ETL, analyses lourdes, rapports	Alertes, monitoring, temps réel

Chapter 5

Cadre pratique et description des données

5.1 Justification du choix des données de cryptomonnaies

Dans le cadre de ce projet d'initiation au Big Data et au Cloud Computing, les données issues des marchés de cryptomonnaies ont été retenues comme cas d'étude. Ce choix repose sur des considérations à la fois techniques et méthodologiques.

Les marchés de cryptomonnaies constituent un environnement particulièrement adapté à l'étude des systèmes Big Data, en raison de la nature **continue, volumineuse et fortement dynamique** des données produites. Contrairement aux marchés financiers traditionnels, ces plateformes fonctionnent sans interruption, générant des flux de données en temps réel 24 heures sur 24 et 7 jours sur 7.

Les données de cryptomonnaies présentent ainsi les principales caractéristiques du Big Data (les 3V) : - un **volume élevé**, lié au nombre important de transactions, - une **vélocité importante**, avec des mises à jour fréquentes des prix et volumes, - une **variété des formats**, incluant des données numériques, temporelles et semi-structurées.

Ce contexte en fait un support pertinent pour illustrer les problématiques de traitement distribué, ainsi que la distinction entre traitements batch et traitements streaming.

5.2 Collecte des données : connexion à la plateforme Binance

Pour la collecte des données, nous nous sommes connectés à la plateforme **Binance**, l'une des plus grandes plateformes d'échange de cryptomonnaies au monde, via ses **API publiques**.

Techniquement, Binance met à disposition des **flux de données en temps réel (WebSocket)** permettant de recevoir en continu les informations de marché pour différentes paires de cryptomonnaies. Une fois la connexion établie, les données sont transmises **événement par événement**, sans interruption, sous forme de messages structurés.

Ces messages sont ensuite publiés sur un **topic Kafka**, jouant le rôle de couche d'ingestion et de mise en file des données. Kafka permet ainsi de : - découpler la phase de collecte de la phase de traitement, - assurer une tolérance aux pannes, - gérer efficacement des flux de données continus à forte fréquence.

Les flux Kafka sont intégrés dans un traitement où **Apache Spark** intervient pour : - ingérer les données en continu ou par lots, - structurer les messages reçus, - appliquer des transformations et nettoyages.

5.3 Description des variables collectées

Les données collectées depuis la plateforme Binance sont composées des variables suivantes :

- **symbol** : identifiant de la paire de cryptomonnaies (par exemple BTCUSDT), permettant de distinguer les actifs analysés.
- **open_price** : prix d'ouverture sur l'intervalle de temps considéré.
- **close_price** : prix de clôture sur l'intervalle, souvent utilisé pour l'analyse des tendances.
- **high_price** : prix maximum atteint durant l'intervalle.
- **low_price** : prix minimum atteint durant l'intervalle.
- **volume** : volume total de l'actif échangé sur la période.
- **quote_volume** : volume échangé exprimé dans la devise de cotation (par exemple en USDT).
- **timestamp_ts** : horodatage associé à chaque observation, essentiel pour les analyses temporelles (par la suite mis sous format date).
- **spread** : écart entre les prix acheteur et vendeur, indicateur de la liquidité du marché.
- **mid_price** : prix moyen calculé à partir des valeurs extrêmes, servant de référence analytique.

Ces variables constituent une base de données cohérente pour l'analyse quantitative des marchés de cryptomonnaies.

5.4 Métriques analysées et intérêt du traitement distribué

À partir des données collectées, plusieurs métriques ont été calculées afin de caractériser le comportement du marché, notamment : - les rendements: mesure de l'évolution relative des prix sur une période donnée, - les mesures de volatilité: - les volumes agrégés par période, - les indicateurs statistiques descriptifs (moyennes, écarts-types, minimums et maximums).

L'analyse de ces métriques permet de mieux comprendre la dynamique des marchés, d'identifier les périodes de forte activité ou de volatilité, et de fournir des éléments d'aide à la décision.

Apache Spark joue ici un rôle central en permettant : - le traitement de volumes importants de données, - l'exécution de calculs analytiques distribués, - la gestion efficace des fenêtres temporelles en streaming, - l'unification des traitements batch et temps réel au sein d'un même environnement.

Chapter 6

Cas de traitement batch

Cette section présente de manière progressive et structurée la mise en œuvre du traitement batch appliqué aux données de marché issues de la plateforme Binance. L'objectif est de décrire clairement l'architecture mise en place, les choix techniques réalisés et les différentes étapes du traitement, depuis la préparation de l'environnement jusqu'à l'automatisation complète.

6.1 Mise en place de l'environnement de travail

Le traitement batch a été réalisé sur une **machine unique**, configurée de manière à reproduire une chaîne complète de traitement Big Data à échelle locale. Cette étape vise à garantir que l'ensemble du pipeline puisse fonctionner de manière cohérente, reproductible et autonome.

6.1.1 Outils et logiciels nécessaires

Plusieurs composants ont été installés afin de couvrir l'ensemble des besoins du projet.

- **Python**
  Python constitue le langage de pilotage du projet. Il est utilisé pour orchestrer les scripts, interagir avec Apache Spark via PySpark, générer les rapports HTML et automatiser l'exécution du pipeline.
- **PostgreSQL**
  PostgreSQL est utilisé comme système de gestion de base de données relationnelle. Il assure le stockage persistant et structuré des données de

marché collectées depuis Binance.
Dans le cadre du projet, PostgreSQL a été téléchargé depuis le site officiel (*PostgreSQL: Downloads*), puis une base de données nommée `crypto_db` a été créée, cette base contient une table `binance_tickers` destinée à recevoir les observations de marché. PostgreSQL joue ici le rôle de **couche de stockage intermédiaire** entre la collecte et l'analyse.

- **Java**
  Apache Spark étant développé en Scala et exécuté sur la **Java Virtual Machine (JVM)**, l'installation de Java est indispensable. Même si les traitements sont écrits en Python (PySpark), la présence de Java est indispensable pour :
  - lancer Spark,
  - gérer l'exécution distribuée,
  - assurer la communication entre les différents composants internes de Spark.

Sans Java, Spark ne peut tout simplement pas fonctionner.

6.1.2 Architecture générale et composants intégrés

L'architecture retenue repose sur une séparation claire entre la **collecte**, le **stockage** et le **traitement analytique**.

Les données de marché ne sont pas traitées directement lors de leur réception. Elles sont d'abord stockées dans PostgreSQL, ce qui permet de conserver un historique fiable et de découpler la phase de collecte de la phase d'analyse.

- **Script d'ingestion des données**
  Un script d'ingestion indépendant, nommé `API_Postgres`, est chargé de :
  - se connecter à l'API de Binance,
  - récupérer les données de marché,
  - insérer ces données dans la table `binance_tickers`.
- **Connexion entre Spark et PostgreSQL (driver JDBC)**
  Apache Spark n'accède pas directement aux bases de données relationnelles. Pour établir la communication avec PostgreSQL, un **driver JDBC PostgreSQL** a été téléchargé et ajouté au projet sous la forme d'un fichier `.jar`, placé dans le dossier `jars`.

  Ce driver joue le rôle d'interface :
  - Spark envoie des requêtes via JDBC,

  - PostgreSQL exécute ces requêtes,
  - les résultats sont renvoyés à Spark sous forme de DataFrame.

Grâce à ce mécanisme, Spark peut lire les données de la table binance_tickers comme s'il s'agissait de données natives.

- **Centralisation des paramètres de connexion**
  Les paramètres de connexion à PostgreSQL (hôte, port, nom de la base, utilisateur, mot de passe) ont été regroupés dans un module Python distinct, nommé config_binance. Cette organisation présente plusieurs avantages :
- elle évite de dupliquer les informations de connexion dans plusieurs scripts,
- elle améliore la lisibilité du code,
- elle facilite la maintenance,
- elle rend le projet facilement reproductible sur une autre machine.

Ainsi, un nouvel utilisateur souhaitant reproduire le projet n'a besoin que : - d'installer PostgreSQL, - de créer la table binance_tickers, - de renseigner ses propres paramètres de connexion dans ce module, - puis d'exécuter les scripts Spark sans autre modification.

6.2 Connexion aux données et lecture du dernier batch disponible

6.3 Connexion aux données et sélection du dernier batch

Le cœur du traitement batch repose sur l'exploitation des données de marché issues de Binance et stockées de manière persistante dans la base de données **PostgreSQL**.

6.3.1 Lecture des données avec Apache Spark

Dans un premier temps, une **session Spark** est créée et configurée afin d'utiliser le **driver JDBC PostgreSQL**. Cette configuration permet à Spark d'établir une connexion directe avec la base relationnelle.

À partir de cette session, la table `binance_tickers` est lue et chargée dans un **DataFrame Spark**, qui constitue la structure centrale pour l'ensemble des traitements analytiques réalisés par la suite.

6.3.2 Notion de batch et rôle de l'horodatage `run_ts`

La table `binance_tickers` est alimentée de manière régulière par le script d'ingestion.
Chaque insertion de données correspond à un **lot de données (batch)** et est associée à un horodatage spécifique, stocké dans la colonne `run_ts`.

Cet horodatage joue un rôle clé : il permet d'identifier précisément à quel batch appartient chaque enregistrement.

Afin de garantir un reporting à jour tout en limitant les coûts de calcul, le traitement Spark ne s'applique pas à l'ensemble de l'historique, mais uniquement au **dernier batch disponible**. Pour cela, la démarche suivante est adoptée :

- identification de la valeur maximale de `run_ts` dans la table ;
- filtrage du DataFrame afin de ne conserver que les lignes correspondant à ce `run_ts` maximal.

Ainsi, chaque exécution du script travaille uniquement sur la **fenêtre de données la plus récente**, sans retraiter l'historique complet des observations.

6.3.3 Notion de ticker et structure des données de marché

Dans le cadre de ce projet, un **ticker** désigne une paire de trading fournie par l'API Binance et identifiée par le champ `symbol` (par exemple : `BTCUSDT`, `ETHUSDT`).

Chaque `symbol` représente un **marché unique**, correspondant à l'échange d'une cryptomonnaie contre une devise de référence (généralement l'USDT). Une même cryptomonnaie (par exemple le Bitcoin) peut donc apparaître dans plusieurs tickers, mais chaque ticker reste distinct.

Au sein d'un même batch, un ticker peut apparaître **plusieurs fois** sur la période observée, ce qui justifie la construction d'indicateurs agrégés (rendement moyen, volatilité, volumes échangés, etc.) afin de résumer et d'interpréter correctement le comportement du marché.

6.4 Enrichissement et construction des indicateurs de marché

Les données brutes issues de Binance (prix d'ouverture, de clôture, plus haut, plus bas, volumes) sont ensuite enrichies afin de produire des indicateurs plus interprétables.

Les principales transformations réalisées sont :

- calcul du **spread de prix** (différence entre le plus haut et le plus bas) ;
- calcul du **prix médian (mid price)**, défini comme la moyenne du prix maximum et du prix minimum ;
- extraction d'une **date au format AAAA-MM-JJ** à partir du timestamp ;
- conversion du temps d'événement en secondes depuis 1970.

Les volumes sont exprimés en **millions d'USDT** afin de faciliter la lecture des résultats.

À partir de ces variables, trois indicateurs analytiques sont calculés :

- le **rendement relatif**, basé sur la variation entre le prix d'ouverture et le prix de clôture ;
- la **volatilité absolue**, mesurant l'amplitude des variations de prix ;
- la **volatilité relative**, obtenue en rapportant la volatilité absolue au prix médian.

Les cas de division par zéro sont explicitement traités afin d'éviter toute erreur de calcul.

6.5 Agrégation des résultats et classements « Top 5 »

Deux niveaux d'agrégation sont ensuite réalisés.

- **Indicateurs globaux de marché** :
  - nombre d'échanges ;
  - nombre total d'observations ;
  - volume total échangé ;
  - rendement moyen du marché ;

  - volatilité moyenne du marché.

- **Statistiques par échange** :
  - rendement moyen ;
  - volatilité moyenne ;
  - volume total échangé.

À partir de ces résultats, plusieurs classements « Top 5 » sont établis : - meilleures performances ; - plus faibles performances ; - volumes les plus élevés ; - volatilités les plus fortes.

Ces classements offrent une lecture synthétique et comparative du marché.

6.6 Génération automatisée d'un rapport HTML

Les résultats sont ensuite convertis en un **rapport HTML automatisé**, comprenant :

- des graphiques illustrant les classements ;
- des tableaux formatés ;
- une synthèse textuelle du comportement global du marché ;
- une section d'aide à la décision, présentée à titre informatif et non comme une recommandation d'investissement.

Chaque rapport est sauvegardé dans un dossier dédié avec un nom horodaté.

6.7 Envoi automatique du rapport par courrier électronique

Afin de faciliter le partage des résultats, un module d'envoi de courriels a été intégré au pipeline.

Le principe repose sur :

- l'utilisation d'un compte Gmail configuré avec un mot de passe d'application ;

- la construction d'un message électronique dont le contenu reprend directement le rapport HTML ;
- l'envoi du message à une liste de destinataires prédéfinie.

Ainsi, à chaque exécution du traitement batch, un nouveau rapport est généré puis immédiatement transmis, sans intervention manuelle.

6.8 Reproductibilité du traitement

Le fichier `config_binance.py` centralise l'ensemble des paramètres nécessaires au bon fonctionnement du pipeline **Binance → PostgreSQL → Spark → Rapport**, en les séparant clairement du code métier.
Ce fichier est **propre à chaque machine** : avant toute exécution, l'utilisateur doit l'adapter à sa configuration locale.

6.8.1 Configuration PostgreSQL

La première partie du fichier concerne la base de données PostgreSQL et regroupe les paramètres suivants :

- `PG_HOST` : adresse de l'hôte PostgreSQL
- `PG_PORT` : port d'écoute
- `PG_SUPER_DB` : base « super » (souvent `postgres`) utilisée pour les opérations d'administration
- `PG_USER` et `PG_PWD` : identifiants de connexion
- `PG_TARGET_DB` : base de données du projet (ici `crypto_db`)

Ces paramètres permettent au script d'ingestion de : - se connecter à PostgreSQL, - créer la base du projet si nécessaire, - alimenter automatiquement les tables cibles,

sans modifier le code principal.

6.8.2 Configuration de l'envoi d'e-mails

La section suivante définit les paramètres nécessaires à l'envoi automatique des rapports :

- `EMAIL_USER` : adresse Gmail expéditrice
- `EMAIL_PWD` : mot de passe d'application associé

Ces informations sont utilisées pour transmettre automatiquement un rapport HTML synthétique par e-mail à la fin de chaque exécution batch.

6.8.3 Paramètres du batch Spark

Enfin, la configuration du batch contrôle le comportement du traitement Spark :

- `BATCH_INTERVAL_SECONDS` : intervalle entre deux exécutions complètes (par exemple toutes les 6 heures)
- `SPARK_MASTER = "local[*]"` : exécution de Spark en mode local en exploitant tous les cœurs disponibles de la machine

Ainsi, le fichier `config_binance.py` constitue un point d'entrée configuration, rendant le projet à la fois **portable** (chaque utilisateur adapte ce fichier) et **reproductible** (même pipeline, environnement différent).

6.9 Exécution du batch

Dans le dossier du projet, l'exécution du batch se fait en deux commandes simples via le terminal / cmd :

' ‘'bash python -m pip install -r requirements.txt python Analyse_Binance.py' ' '

- La première commande python -m pip install -r requirements.txt installe automatiquement toutes les bibliothèques Python nécessaires au projet (pandas, requests, psycopg2, pyspark, etc.) à partir du fichier requirements.txt. Cela garantit que l'environnement contient exactement les dépendances attendues pour que le script tourne correctement.
- La deuxième commande python Analyse_Binance.py lance le pipeline batch complet :
  - appel de l'API Binance et récupération des tickers,
  - insertion/actualisation des données dans PostgreSQL (crypto_db),

- lancement de Spark en mode local pour analyser les données (gagnants/perdants, volumes, volatilité, etc.),
- génération périodique du rapport et envoi par email.

L'ensemble du processus est ainsi entièrement automatisé et peut être relancé de manière identique sur toute machine correctement configurée.

Chapter 7

Cas de traitement streaming

Cette section présente de manière détaillée la mise en œuvre du **traitement en streaming** appliqué aux données de marché issues de la plateforme Binance. Contrairement au traitement batch, le streaming vise à traiter les données **au fil de leur arrivée**, avec une faible latence, afin de réagir quasi instantanément aux nouveaux événements de marché.

7.1 Mise en place de l'environnement de travail

La mise en œuvre du traitement streaming repose sur une **architecture hybride**, combinant des services conteneurisés et un moteur de calcul exécuté localement.

Plus précisément :

- **Docker** est utilisé pour exécuter les services d'infrastructure (Kafka, Zookeeper, PostgreSQL) dans des environnements isolés et reproductibles ;
- **Apache Spark** est exécuté en local (hors conteneur), afin de faciliter le développement, le débogage et l'observation des traitements.

Cette approche permet de bénéficier à la fois : - de la **reproductibilité** offerte par Docker, - et de la **simplicité de développement** d'un Spark local.

7.2 Installation et configuration d'Apache Spark en local

Le traitement streaming est réalisé à l'aide d'Apache Spark exécuté directement sur une machine Windows.

7.2.1 Installation d'Apache Spark

La version suivante a été utilisée :

- **Apache Spark 4.0.1** (`spark-4.0.1-bin-hadoop3.tgz`)

Apache Spark est développé en **Scala** et repose sur l'écosystème Hadoop. Même lorsque les traitements sont écrits en Python (via PySpark), Spark s'exécute sur la **Java Virtual Machine (JVM)**.

7.2.2 Spécificités Windows

Sous Windows, certaines dépendances Hadoop ne sont pas disponibles nativement. Afin d'assurer le bon fonctionnement de Spark, les éléments suivants ont été ajoutés :

- `winutils.exe`
- `hadoop.dll`

Ces fichiers sont nécessaires pour : - la gestion correcte du système de fichiers, - certaines opérations internes de Spark et Hadoop.

Sans ces composants, Spark peut démarrer mais échouer lors de traitements plus avancés.

7.3 Rôle de Docker dans le projet

7.3.1 Pourquoi utiliser Docker ?

Docker est une plateforme de **conteneurisation** permettant d'exécuter des applications dans des environnements isolés, incluant toutes leurs dépendances.

Dans le cadre de ce projet, Docker permet de :

- déployer rapidement les services nécessaires au streaming,
- éviter les conflits de versions entre machines,
- garantir un environnement d'exécution reproductible.

Contrairement à une machine virtuelle complète, Docker :

- partage le noyau du système hôte,
- est plus léger,
- démarre plus rapidement.

7.3.2 Docker Desktop sous Windows

Docker repose sur des mécanismes propres au noyau Linux. Sous Windows, Docker Desktop s'appuie donc sur **WSL2** ou **Hyper-V** afin d'exécuter une machine Linux légère permettant au moteur Docker de fonctionner correctement.

7.4 Orchestration des services avec *docker-compose*

Afin de gérer plusieurs services simultanément, un fichier `docker-compose.yml` a été utilisé.

Ce fichier permet de :

- définir plusieurs services dans un seul document,
- configurer les ports, volumes et réseaux,
- lancer ou arrêter l'ensemble de l'infrastructure avec une seule commande.

7.4.1 Services déployés

Dans le cadre de ce projet, `docker-compose` permet de lancer :

- **Zookeeper**, nécessaire à la coordination de Kafka ;
- **Kafka**, utilisé pour la gestion des flux de données en temps réel ;
- **PostgreSQL**, utilisé pour le stockage persistant des données traitées.

Apache Spark n'est pas conteneurisé et reste exécuté localement.

7.4.2 Image Docker et conteneur Docker

Il convient de distinguer :

- une **image Docker**, qui correspond à un modèle statique contenant l'application et ses dépendances ;
- un **conteneur Docker**, qui est une instance active créée à partir de cette image.

7.4.3 Lancement des services

Les services sont démarrés à l'aide de la commande suivante :

```
docker compose up -d
```

Une fois cette commande exécutée, l'ensemble des services devient accessible via les ports définis dans le fichier de configuration.

7.5 Kafka : organisation du streaming des données

Le cœur de l'architecture de streaming repose sur **Apache Kafka**, qui joue le rôle de système de messagerie distribué et tolérant aux pannes. Kafka permet de découpler la **production des données** de leur **traitement**, tout en garantissant un débit élevé et une faible latence.

7.5.1 Notion de *topic* Kafka

Dans Kafka, les données sont organisées autour du concept de **topic**.
Un topic peut être vu comme un **canal de diffusion** dans lequel circulent des messages.

Un topic met en relation deux types d'acteurs :

- un **producer**, qui publie (écrit) les messages dans le topic ;
- un **consumer**, qui s'abonne au topic pour lire les messages.

Chaque topic est découpé en **partitions**. Cette organisation permet :

- le **parallélisme**, car plusieurs consommateurs peuvent lire différentes partitions en parallèle ;
- la **montée en charge**, en répartissant les données sur plusieurs partitions ;
- la **conservation temporaire des messages**, ce qui permet de relire les données si nécessaire.

Kafka garantit ainsi un stockage fiable des flux entrants, même en cas de pic de charge ou de ralentissement du système de traitement.

7.5.2 Topic utilisé dans le projet

Dans le cadre de ce projet, un seul topic Kafka a été créé :

- `projet-bdcc`

Ce topic centralise l'ensemble des messages de marché transmis en temps réel depuis la plateforme Binance.
Il constitue le point d'entrée unique des données streaming avant leur traitement par Apache Spark.

7.6 Extraction des données en temps réel : WebSocket Binance vers Kafka

7.6.1 Principe du WebSocket

Un **WebSocket** est un protocole de communication permettant d'établir une connexion persistante entre un client et un serveur.
Contrairement aux requêtes HTTP classiques, la connexion reste ouverte et permet l'échange de données **en continu**, sans avoir à relancer de nouvelles requêtes.

Ce mécanisme est particulièrement adapté aux cas d'usage temps réel, où les données sont produites de manière fréquente et imprévisible.

7.6.2 Flux Binance utilisé

Les données de marché sont récupérées à partir du flux WebSocket public fourni par Binance : wss://stream.binance.com:9443/ws/!miniTicker@arr

Ce flux diffuse en continu des informations de marché pour un grand nombre de paires de cryptomonnaies.
Les messages sont transmis sous forme **JSON**, chaque message correspondant à un événement de marché (mise à jour de prix, volumes, etc.).

7.6.3 Envoi des données vers Kafka

Une fois la connexion WebSocket établie :

- chaque message reçu est immédiatement pris en charge par un **Kafka Producer** ;
- le message est publié dans le topic `projet-bdcc`.

Kafka joue ainsi le rôle de **tampon fiable** entre la source de données (Binance) et le moteur de calcul (Apache Spark).
Cette architecture permet de :

- absorber les variations de débit des données entrantes ;
  - garantir qu'aucune donnée ne soit perdue ;
- découpler la collecte des données de leur traitement analytique.

7.7 Traitement des flux avec Spark Structured Streaming

Les messages stockés dans Kafka sont ensuite consommés par Apache Spark à l'aide du module **Structured Streaming**.

Spark Structured Streaming permet de traiter des flux de données continus en s'appuyant sur le modèle des **DataFrames**, déjà utilisé pour les traitements batch.

Concrètement, Spark assure les opérations suivantes :

- lecture continue des messages depuis le topic Kafka ;
- parsing et désérialisation des messages JSON ;
- normalisation des champs (prix, volumes, horodatage, symboles, etc.) ;
- application de transformations et de calculs analytiques en temps réel.

Grâce à cette approche, les traitements streaming restent proches des traitements batch classiques, ce qui facilite :

- la compréhension du code,
- la maintenance,
- et l'évolution du pipeline analytique.

7.8 Stockage des données traitées dans PostgreSQL

Une fois les données traitées par Spark, elles sont stockées dans une base de données relationnelle afin d'assurer leur persistance et leur exploitation ultérieure.

Dans l'architecture mise en place :

- **PostgreSQL** s'exécute dans un conteneur Docker ;
- Apache Spark écrit les données via une connexion **JDBC** ;
- les données sont insérées ou mises à jour selon la logique définie dans le projet.

7.9 Dashboard

Un tableau de bord interactif a été développé avec **Streamlit** afin de visualiser, en quasi temps réel, les données de marché issues de Binance et traitées par le pipeline de streaming.

7.9.1 Architecture et flux de données

Le dashboard se connecte directement à la base **PostgreSQL** (via `SQLAlchemy`) contenant une table `projet_bdcc`. À chaque rafraîchissement, l'application exécute une requête SQL filtrant les observations selon :

- une sélection de cryptomonnaies (`symbol`) choisie dans la barre latérale ;
- une limite d'observations (`LIMIT`) afin de contrôler le volume chargé en mémoire ;
- un tri temporel sur `timestamp_ts` afin de reconstruire correctement les séries temporelles.

Les données sont ensuite converties en `DataFrame` Pandas pour permettre la production rapide de métriques et de graphiques.

7.9.2 Rafraîchissement automatique et logique de cache

Le suivi temps réel est assuré par un **auto-refresh** (`streamlit_autorefresh`) déclenché toutes les `REFRESH_INTERVAL = 5` secondes. Ce mécanisme permet de recharger les données en continu sans intervention manuelle.

Pour éviter des requêtes inutiles et améliorer la performance, le chargement est encapsulé dans une fonction décorée par `@st.cache_data` avec un *TTL* (Time To Live). Ainsi, sur un court intervalle, Streamlit réutilise le résultat en cache plutôt que de relancer systématiquement la requête.

7.9.3 Enrichissement des données : indicateurs calculés côté dashboard

Après chargement, l'application calcule plusieurs indicateurs dérivés afin d'obtenir des mesures directement interprétables :

- **Variation relative (%)** : Cet indicateur mesure le rendement sur la période observée.
- **Volatilité relative** : Il donne une approximation de l'amplitude des fluctuations, normalisée par le prix.
- **Spread** : Il représente l'écart absolu entre le maximum et le minimum observés sur la période.

Ces calculs sont effectués à l'affichage afin de rendre l'interface autonome, tout en laissant la persistance se concentrer sur les variables brutes essentielles.

7.9.4 Paramétrage utilisateur (Sidebar)

La barre latérale permet :

- de sélectionner les cryptomonnaies à suivre (`multiselect`) à partir de la liste des symboles disponibles en base ;

- de filtrer la période d'observation (`1H`, `24H`, `7J`, `30J`, `Tout`) via un filtre temporel appliqué sur `timestamp_ts` ;

Ce choix d'interface permet à l'utilisateur d'adapter dynamiquement le périmètre d'analyse sans modifier le code.

7.9.5 Organisation fonctionnelle du dashboard (onglets)

Le dashboard est structuré en cinq onglets principaux, chacun répondant à un besoin analytique complémentaire.

7.9.5.1 *Overview* : résumé par actif

L'onglet **Overview** fournit, pour chaque crypto sélectionnée :

- le dernier prix de clôture ;
- la variation absolue et relative sur la période filtrée ;
- des statistiques agrégées : prix moyen, maximum, minimum, volatilité moyenne et volume cumulé.

Ces métriques sont affichées sous forme de *cards* et de blocs synthétiques afin de faciliter une lecture rapide de l'état du marché.

7.9.5.2 *Graphiques* : volume et tendances

L'onglet **Graphiques** propose, pour chaque crypto :

- un graphique *candlestick* (Open/High/Low/Close) ;
- deux **moyennes mobiles** (`MA7`, `MA25`) pour visualiser la tendance ;
- un histogramme de **volume** superposé.

L'objectif est de combiner sur une même vue :

- la dynamique des prix,
- les zones de volatilité,
- et l'intensité des échanges (liquidité via le volume).

7.9.5.3 *Top Cryptos* : classements et comparaison

L'onglet **Top Cryptos** construit des classements à partir de la dernière observation de chaque symbole :

- **Top 5 par volume** (actifs les plus échangés) ;
- **Top Gainers** (plus fortes hausses, via `pct_change`) ;
- **Top Losers** (plus fortes baisses).

Deux visualisations complémentaires sont proposées :

- évolution des *gainers* sous forme d'indice base 100 (comparabilité des trajectoires) ;
- barres de variation (%) pour les *losers* (lecture immédiate des chutes).

7.9.5.4 *Anomalies* : alertes de marché

L'onglet **Anomalies** détecte des comportements atypiques sur la dernière observation disponible :

- **Variations extrêmes** : $|pct\_change| > 5\%$
- **Volatilité élevée** : sélection des symboles au-dessus du 3e quartile de `volatility`

Les anomalies sont accompagnées :

- de tableaux stylisés (gradients),
- et de graphiques en barres pour synthétiser les alertes.

Cette partie vise à fournir un mécanisme simple de *surveillance* du marché, utile pour identifier rapidement les actifs "instables" ou "en mouvement".

7.9.5.5 *Détails Crypto* : focus sur un actif

L'onglet **Détails Crypto** permet une analyse approfondie d'un actif choisi :

- tableau récapitulatif des dernières valeurs (Open, Close, High, Low, spread, variation, volatilité, volume) ;
- graphique High/Low + moyennes mobiles ;
- graphique de volume ;
- historique complet formaté (table interactive) avec mise en évidence de la variation et de la volatilité.

Cette vue détaillée permet à l'utilisation, après une observation globale, d'explorer un actif en profondeur.

7.9.5.6 Apport du dashboard dans le pipeline streaming

Le dashboard Streamlit matérialise la dernière étape du pipeline en offrant :

- une **visualisation quasi temps réel** des données ingérées en streaming,
- une capacité de **monitoring** via des indicateurs et alertes,
- une **interface interactive** facilitant l'exploration des séries temporelles et des anomalies,
- une exploitation directe des données persistées dans PostgreSQL, sans dépendre d'un environnement de développement.

En pratique, il constitue un composant clé pour démontrer la valeur du streaming : au lieu d'attendre un traitement batch, l'information est consultable dès qu'elle est disponible dans la base, avec un rafraîchissement automatique.

Chapter 8

Forces et limites d'Apache Spark

Cette section propose une synthèse critique des principaux **atouts** et **limites** d'Apache Spark. L'objectif est de mettre en évidence les situations dans lesquelles Spark constitue une solution particulièrement adaptée, ainsi que les contraintes techniques qu'il convient de prendre en compte lors de son utilisation.

8.1 1. Forces d'Apache Spark

8.1.1 1.1 Performances élevées grâce au calcul en mémoire

L'un des principaux atouts d'Apache Spark réside dans son modèle de calcul **in-memory**.
Contrairement à Hadoop MapReduce, qui écrit systématiquement les résultats intermédiaires sur disque, Spark conserve les données en mémoire vive (RAM) lorsque cela est possible.

Ce mécanisme permet : - une réduction significative des temps de latence, - des performances très élevées pour les traitements itératifs, - une exécution rapide des algorithmes analytiques et de machine learning.

Dans les cas pratiques étudiés, ce modèle est particulièrement adapté : - au calcul fréquent d'indicateurs de marché, - à l'analyse répétée de fenêtres temporelles, - aux agrégations complexes sur des volumes importants de données.

8.1.2 1.2 Un moteur unifié pour différents types de traitements

Apache Spark se distingue par sa **polyvalence**. Il repose sur un moteur unique capable de gérer : - le traitement **batch** de grands volumes de données, - le **streaming** quasi temps réel (Structured Streaming), - les requêtes **SQL** (Spark SQL), - les algorithmes de **machine learning** (MLlib), - le traitement de graphes (GraphX).

Cette unification présente plusieurs avantages : - un même modèle de programmation pour des cas d'usage variés, - une réduction de la complexité logicielle, - une meilleure cohérence dans les pipelines de données.

Dans le cadre de ce projet, Spark a permis d'implémenter à la fois : - des traitements batch pour l'analyse consolidée des données, - des traitements streaming pour l'analyse en continu des flux temps réel.

8.1.3 1.3 Scalabilité et parallélisme

Spark est conçu pour fonctionner : - sur une **machine unique** (mode local), - ou sur un **cluster distribué** composé de plusieurs nœuds.

Grâce au découpage automatique des données en partitions et à l'exécution parallèle des tâches, Spark permet : - une montée en charge progressive, - une exploitation efficace des ressources CPU et mémoire, - une adaptation à des volumes de données croissants.

Cette caractéristique rend Spark particulièrement adapté aux environnements Big Data et aux architectures Cloud.

8.1.4 1.4 Résilience et tolérance aux pannes

Apache Spark intègre des mécanismes avancés de **tolérance aux pannes**.
Les données sont représentées sous forme de structures immuables (RDDs ou DataFrames), associées à une logique de reconstruction basée sur le **DAG (Directed Acyclic Graph)**.

En cas de défaillance : - Spark est capable de recalculer automatiquement les partitions perdues, - l'exécution peut se poursuivre sans redémarrage complet du job.

Cette résilience est essentielle dans des environnements distribués, où les pannes matérielles ou logicielles sont fréquentes.

8.2 2. Limites d'Apache Spark

8.2.1 2.1 Forte consommation de ressources mémoire

La principale contrepartie du calcul en mémoire est la **consommation élevée de RAM**.
Pour fonctionner efficacement, Spark nécessite : - une quantité suffisante de mémoire, - un dimensionnement précis des ressources, - une configuration adaptée (gestion des partitions, cache, persistence).

Dans les environnements Cloud ou sur des machines limitées, cela peut entraîner : - un coût matériel ou financier élevé, - des problèmes de performance si la mémoire est insuffisante.

8.2.2 2.2 Streaming basé sur le micro-batching

Bien que Spark Structured Streaming permette le traitement de flux continus, il repose sur un modèle de **micro-batching**.
Les données sont traitées par petits lots successifs, et non strictement événement par événement.

Cela implique : - une latence faible mais non nulle, - une légère différence avec le "vrai" temps réel.

Pour des applications nécessitant une latence extrêmement faible (par exemple, trading haute fréquence ou détection instantanée d'événements critiques), d'autres solutions comme **Apache Flink** peuvent être plus adaptées.

8.2.3 2.3 Complexité de l'optimisation des performances

Bien que l'API Spark soit relativement accessible, l'optimisation fine des performances peut s'avérer complexe, notamment pour : - le choix du nombre de partitions, - la gestion du cache mémoire, - l'équilibrage des ressources, - la compréhension des plans d'exécution (DAG, stages, tasks).

Cette complexité peut constituer une barrière pour les débutants et nécessite une bonne compréhension des mécanismes internes de Spark.

8.2.4 2.4 Moins adapté à la gestion d'un très grand nombre de petits fichiers

Spark peut être moins performant lorsqu'il est confronté à : - un très grand nombre de fichiers de petite taille, - des accès disque fragmentés.

Dans ce cas, des solutions complémentaires (compaction des fichiers, formats optimisés comme Parquet, ou systèmes spécialisés) sont souvent nécessaires.

Chapter 9

Conclusion générale

Ce projet d'initiation au Big Data et au Cloud Computing a permis de mettre en œuvre, de manière concrète, les principaux concepts abordés en cours à travers l'étude d'un cas pratique réel sur l'analyse des données de marché des cryptomonnaies.

Dans un premier temps, le travail a mis en évidence les limites des approches traditionnelles de traitement des données face à des volumes importants, des flux continus et des exigences de réactivité accrues. Ces constats ont conduit à l'utilisation d'Apache Spark, une technologie Big Data moderne, capable de répondre à ces enjeux grâce à son modèle de calcul distribué et en mémoire.

La première partie pratique du projet s'est concentrée sur le **traitement batch**. À partir des données de marché collectées depuis la plateforme Binance et stockées dans une base PostgreSQL, Apache Spark a été utilisé pour construire des indicateurs analytiques pertinents (rendement, volatilité, volume, classements Top 5). Cette approche a permis de produire des analyses consolidées, synthétisées sous la forme de rapports HTML automatisés, facilitant l'interprétation des résultats et leur diffusion. Le traitement batch s'est révélé particulièrement adapté aux analyses globales et aux besoins de reporting périodique.

La seconde partie du projet a porté sur le **traitement streaming**, illustrant la capacité de Spark à gérer des flux de données quasi temps réel. Grâce à l'intégration de Kafka et à l'utilisation de WebSockets pour la récupération des données Binance, un pipeline de streaming complet a été mis en place. Apache Spark Structured Streaming a permis de consommer, transformer et stocker les données en continu, tout en conservant un modèle de programmation proche de celui des traitements batch. Cette approche a mis en évidence l'intérêt du streaming pour des cas d'usage nécessitant une réactivité accrue et une surveillance continue des données de marché.

L'architecture hybride adoptée, combinant Spark en local et des services d'infrastructure conteneurisés via Docker (Kafka, Zookeeper, PostgreSQL),

a également permis de souligner l'importance de la reproductibilité et de la modularité dans les projets Big Data. L'utilisation de Docker a facilité le déploiement des services, tandis que Spark a assuré la cohérence des traitements analytiques.

Au-delà des résultats obtenus, ce projet a permis de mieux comprendre les **forces et limites d'Apache Spark**. Si Spark offre des performances élevées, une grande polyvalence et une forte capacité de montée en charge, son utilisation efficace nécessite une bonne gestion des ressources et une compréhension des compromis techniques, notamment en matière de mémoire et de latence en streaming.

En conclusion, ce travail illustre la manière dont les technologies Big Data et Cloud peuvent être mobilisées pour construire des pipelines de données complets, allant de la collecte à l'analyse, en passant par le stockage et la visualisation. Il constitue une base solide pour aborder des architectures plus avancées et des cas d'usage industriels, tout en offrant une compréhension concrète des enjeux actuels du traitement des données massives.

Chapter 10

Ressources bibliographiques

10.1 Apache Spark et Big Data

- **AWS** — *Qu'est-ce qu'Apache Spark ?*
  Présentation générale d'Apache Spark, de ses principes fondamentaux et de ses cas d'usage dans le Big Data et le Cloud.
  https://aws.amazon.com/fr/what-is/apache-spark/

10.2 Streaming de données et Kafka

- **Confluent** — *Kafka Basics*
  Introduction aux concepts clés de Kafka (topics, producers, consumers, partitions), utile pour comprendre les architectures de streaming.
  https://developer.confluent.io/learn-kafka/

- **Adaltas** — *Spark Streaming Data Pipelines with Structured Streaming*
  Article détaillant la conception de pipelines de données temps réel à l'aide de Spark Structured Streaming.
  https://www.adaltas.com/fr/2019/04/18/spark-streaming-data-pipelines-structured-streaming/

10.3 Conteneurisation et Docker

- **Docker Inc.** — *Docker Documentation*
  Documentation officielle sur Docker, Docker Desktop et les principes de la conteneurisation.
  https://docs.docker.com/

- **Subham Kharwal** — *Docker Images Repository*
  Exemples d'images Docker et bonnes pratiques de structuration de conteneurs.
  https://github.com/subhamkharwal/docker-images?tab=readme-ov-file

---

10.4 Bases de données relationnelles

- **PostgreSQL Global Development Group** — *PostgreSQL Documentation*
  Documentation officielle du système de gestion de base de données PostgreSQL, utilisée pour le stockage des données du projet.
  https://www.postgresql.org/docs/

10.5 Données financières et API Binance

- **Binance** — *Plateforme de trading Spot*
  Source des données financières exploitées dans le projet.
  https://www.binance.com/fr

- **Binance Developers** — *Binance Spot API Documentation*
  Documentation officielle des API REST et WebSocket permettant l'accès programmatique aux données de marché.
  https://developers.binance.com/docs/binance-spot-api-docs