Streaming (Big) Data et Architectures de Données

- Étude de cas avec Apache Spark Structured Streaming, Apache Kafka et Delta Lake -

Khaled Jouini
j.khaled@gmail.com

ISITCom, University of Sousse

Ce document peut comporter des erreurs, à utiliser donc en connaissance de cause!

Plan du cours (1/2)

Plan du cours (2/2)

Lectures intéressantes (1/2)

- *Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics*.
  Armbrust, M., Ghodsi, A., Xin, R. S., Zaharia, M., & Das, T.
  Conference on Innovative Data Systems Research (CIDR). 2021.
- *Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores*.
  Xin, R., Armbrust, M., Das, T., et al.
  Proceedings of the VLDB Endowment, 13(12), 3411{3424. 2020.
- *Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark*.
  Armbrust, M., Das, T., Xin, R. S., Yavuz, B., Zhu, S., Gonzalez, J. E. & Zaharia, M.
  Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18), 601-613. 2018.
- *Kafka: A Distributed Messaging System for Log Processing*.
  Kreps, J., Narkhede, N., & Rao, J.
  In Proceedings of the NetDB (pp. 1-7). 2011.

Lectures intéressantes (2/2)

- *In Search of an Understandable Consensus Algorithm*.
  Ongaro, D., & Ousterhout, J.
  In Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 2014) (pp. 305{319). USENIX Association. 2014.

- *Concurrency Control in Distributed Database Systems*.
  Bernstein, P. A., & Goodman, N.
  ACM Computing Surveys (CSUR), 13(2), 185-221. 1983.

- *Delta Lake: The Definitive Guide*.
  Lee D., Wentling T., Haines S. & Babu P.
  eBook, O'Reilly, 978-1-098-15195-9. November 2024.
  Disponible sur `https://delta.io/pdfs/dldg_databricks.pdf`.

- *Documentation officielle de Delta Lake (`https://delta.io/`), Apache Spark et d'Apache Kafka*

Section 1 - 1. Stream Processing et Streaming Data - Étude de cas avec Apache Spark

1.1. Stream Processing vs. Batch Processing

Stream Processing : méthode de traitement des données en **temps réel**, où les données sont traitées au fur et à mesure qu'elles arrivent, plutôt qu'en lots.

Comparaison avec le traitement par lots :

- **Batch Processing** : Les données sont collectées sur une période, puis traitées en une seule fois.
- **Stream Processing** : Les données sont traitées **en continu** dès leur réception, réduisant ainsi la latence entre l'apparition de l'événement et son traitement.

1.1. Stream Processing vs. Batch Processing

- **Réactivité en temps réel** : Permet de prendre des décisions immédiates (ex : détection de fraude).
- **Traitement continu des données** : Traite les données non structurées ou semi-structurées à mesure qu'elles arrivent (ex : flux de capteurs, logs).
- **Adapté aux grandes volumétries** : Supporte des millions d'événements par seconde avec une scalabilité horizontale.

1.1. Comparaison : Data at Rest vs Data in Motion

Caractéristiques	Data at Rest	Data in Motion
Données	Statiques	Dynamiques, en temps réel
Stockage	Bases de données, entrepôts de données, Data Lakes	En transit sur les réseaux ou dans un pipeline
Traitement	Par lots (*batch processing*)	Continu ou en temps réel
Latence	Haute (analyse différée)	Faible (analyse immédiate)
Volume	Volume important mais fixe	Volume potentiellement infini
Exemples	Analyse de logs, rapports financiers	Transactions financières, IoT, réseaux sociaux
Cas d'utilisation	Archivage, Data Warehousing	Détection de fraudes, monitoring temps réel
Technologies	HDFS, bases de données, Data Lakes	Kafka, Spark Streaming, Flink

Table: Comparaison entre Data at Rest et Data in Motion

1.1. Cas d'utilisation du Stream Processing

- **Surveillance des systèmes IoT** : Analyse des données de capteurs en temps réel pour détecter des anomalies.
- **Systèmes financiers** : Surveillance des transactions pour détecter la fraude instantanément.
- **Analyse des logs en temps réel** : Détection des erreurs système et des incidents de sécurité en direct.
- **Publicité en ligne** : Analyse des clics en temps réel pour ajuster les campagnes publicitaires.

1.1. Défis du Stream Processing

- **Faible Latence** : Traitement des données presque instantané pour offrir une réactivité élevée.
- **Tolérance aux pannes** : Assurer que les données ne sont pas perdues ou doublées en cas de panne du système.
- **Consistance des données** : Garantir des résultats exacts même en cas d'échec ou de perturbation (**exactly-once processing**).
- **Gestion des données tardives** : Intégrer les données arrivant en retard tout en assurant la précision des résultats.

1.2. Apache Spark Structured Streaming

- **Structured Streaming**
  - Framework de traitement de flux de données, **scalable** et **tolérant aux pannes** construit sur le moteur d'exécution Spark SQL
  - **Avantages** : **expression des traitements avec les mêmes opérateurs utilisés pour le traitement batch**.
  - Possibilité de faire la jointure entre données en flux et données batch.

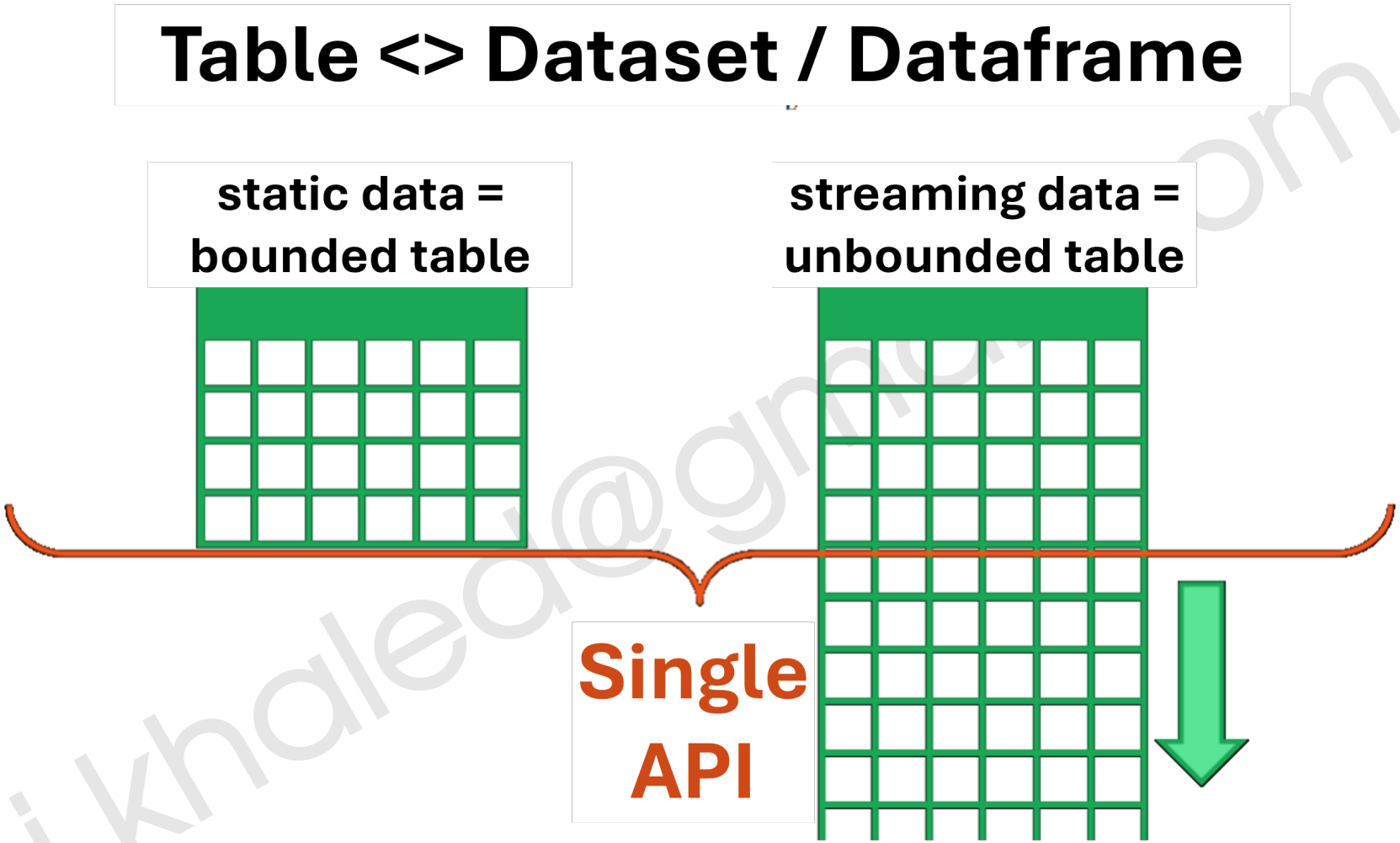
- Traitement batch avec Spark :

```
inputDF = spark.read.format("json").load("source-path")
resultDF = inputDF.select("device", "signal").where("signal > 15")
resDF.write.format("parquet").save("dest-path")
```

- Traitement en streaming avec Spark :

```
iStreamDF= spark.readStream.format("json").load("source-path")
rStreamDF= iStreamDF.select("device","signal").where("signal > 15")
rStreamDF.writeStream.format("parquet").start("dest-path")
```

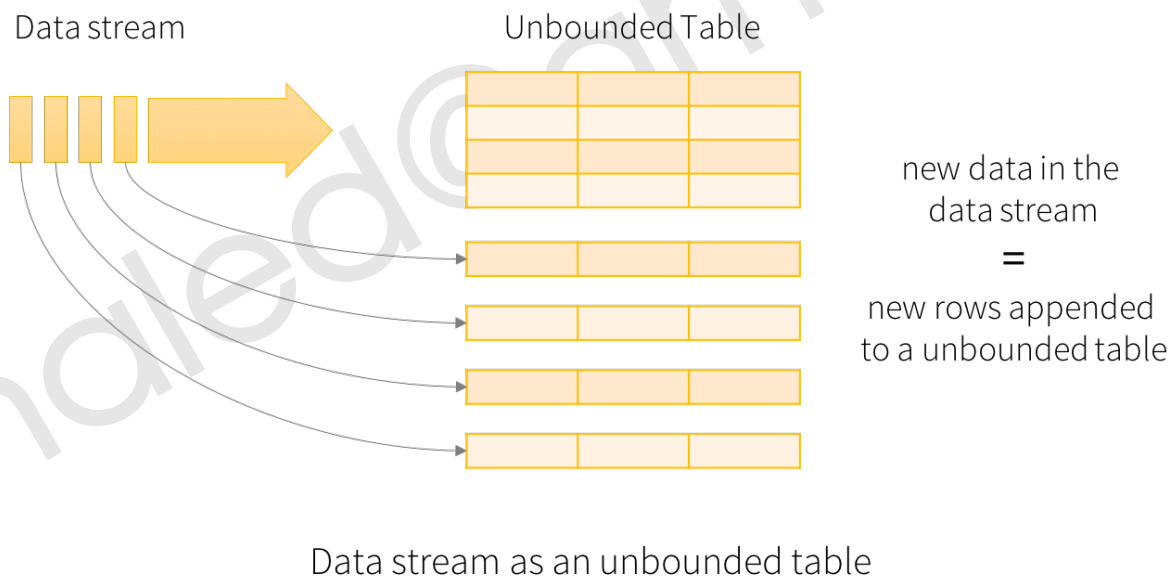
1.2. Modèle de données



1.2. Modèle de données

Concept clé : Le modèle de **Spark Structured Streaming** repose sur une abstraction simple où un **flux de données** est traité comme une **table dynamique**.

Chaque donnée arrivant dans le flux est vue comme une **nouvelle ligne** ajoutée à une table d'entrée.



Data stream as an unbounded table

(Source : Documentation officielle d'Apache Structured Streaming)

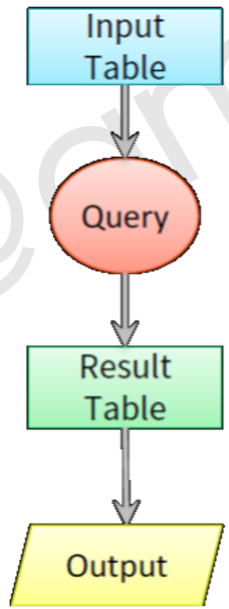
1.2. Single API

Les transformations sur ces données sont exécutées sous forme de requêtes SQL ou DataFrame, comme pour des tables statiques.

```
input = spark.read
          .format("json")
          .load("source-path")

result = input
          .select("device", "signal")
          .where("signal > 15")

result.write
          .format("parquet")
          .save("dest-path")
```




```
input = spark.readStream
          .format("json")
          .load("source-path")

result = input
          .select("device", "signal")
          .where("signal > 15")

result.writeStream
          .format("parquet")
          .start("dest-path")
```

1.2. Triggers et Micro-batchs

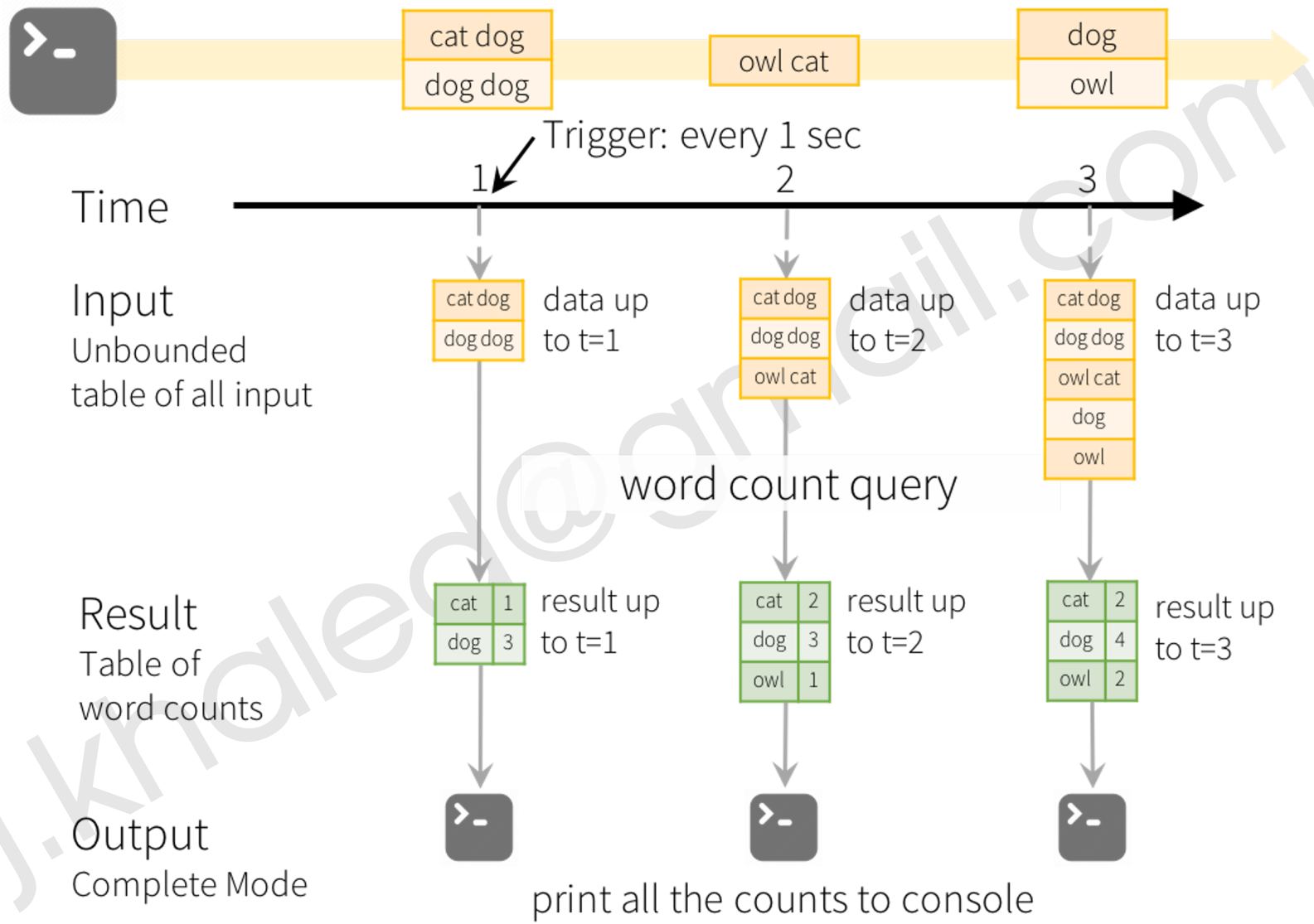
Traitement incrémental

- **Important** : **une seule passe sur les données** (on ne peut pas tout garder, c'est potentiellement infini!)
- À chaque intervalle de temps (ou **trigger**), seuls les **nouveaux éléments** sont traités.
- Les nouveaux éléments forment un **micro-batch** : le micro-batch est l'**unité de travail** de Spark Streaming

$\Rightarrow$ Les triggers spécifient la fréquence à laquelle les données sont lues et traitées (par exemple, toutes les 5 secondes, toutes les minutes).

- Les triggers peuvent être basés sur le *processing Time* ou sur l'*event Time* (nous y reviendrons).
- Cela permet d'**optimiser** l'utilisation de la mémoire et de réduire le temps de traitement.

1.2. Triggers et Micro-batchs



(Source : Documentation officielle d'Apache Structured Streaming)

1.2. Triggers et Micro-batchs

- Les **triggers** contrôlent la fréquence de traitement des données du flux (lecture, traitement, enregistrement du résultat).
- Trois types de triggers
  - **Processing Time** (`trigger(processingTime='10 seconds')`): Traitement déclenché toutes les X secondes.
  - **Once** (`trigger(once=True)`): Un traitement unique de toutes les données disponibles, puis arrêt du flux.
  - **Continuous** (`trigger(continuous='1 second')`): Traitement en continu avec la latence la plus faible possible. .

```
wordCounts.writeStream \
    .trigger(processingTime='10 seconds') \
    .format("console") \
    .start()
```

1.2. Micro-batchs et traitement incrémental

- Le dataframe n'est pas conservé dans son intégralité en mémoire.
- Seul le dernier micro-batch est gardé en mémoire
- Une fois traité, un micro-batch est **éliminé** pour libérer de la mémoire.
- Spark conserve uniquement l'**état minimal** nécessaire pour les agrégations ou les calculs de longue durée.

1.2. Modes de sortie

Trois modes de sortie sont utilisés pour déterminer comment les résultats d'un traitement en streaming sont écrits dans un stockage externe

- **Complete Mode**
  - Tout le résultat est réécrit à chaque intervalle (même si seules quelques lignes ont changé).
  - Utile les calculs cumulatifs (sommes totales, moyennes sur toute la période).
  - Scénarios où l'état complet est nécessaire à chaque instant.
  - Coûteux si la table est très grande.
- **Append Mode**
  - Seules les nouvelles lignes sont ajoutées.
  - Utile par exemple pour la création d'historiques.
- **Update Mode**
  - Seules les lignes modifiées sont écrites (upsert).
  - Exemple : mises à jour en temps réel de métriques ou de tableaux de bord.

1.2. Modes de sortie - Exemple

Flux de mots

- intervalle 1 : "chat", "chien", "oiseau" ;
- intervalle 2 : "chat", "chien", "poisson"

Agrégation : Comptage des occurrences de chaque mot

Mode Append

- Sortie **Premier intervalle** : chat: 1, chien: 1, oiseau: 1
- Sortie **Deuxième intervalle** : chat: 2, chien: 2, poisson: 1

Sortie totale en mode Append

Mot	Occurrences
chat	1
chien	1
oiseau	1
chat	2
chien	2
poisson	1

1.2. Modes de sortie - Exemple (suite)

Flux de mots

- intervalle 1 : "chat", "chien", "oiseau" ;
- intervalle 2 : "chat", "chien", "poisson"

Mode Update

- Sortie **Premier intervalle** : chat: 1, chien: 1, oiseau: 1
- Sortie **Deuxième intervalle** : chat: 2, chien: 2, poisson: 1

Sortie totale en mode Update

Mot	Occurrences
chat	2
chien	2
oiseau	1
poisson	1

Seules les lignes mises à jour sont écrites. Les lignes inchangées ne sont pas réécrites.

1.2. Modes de sortie - Exemple (suite)

Flux de mots

- intervalle 1 : "chat", "chien", "oiseau" ;
- intervalle 2 : "chat", "chien", "poisson"

Mode Complete

- Sortie **Premier intervalle** : chat: 1, chien: 1, oiseau: 1
- Sortie **Deuxième intervalle** : chat: 2, chien: 2, oiseau: 1, poisson: 1

Sortie totale en mode Complete

Mot	Occurrences
chat	2
chien	2
oiseau	1
poisson	1

Le mode complete écrit l'ensemble des résultats à chaque intervalle pour un snapshot complet de l'état.

1.2. Comparaison des Modes de Sortie

Quand utiliser quel mode ?

Mode	Description	Utilisation typique
Complete	Réécrit toute la table	Calculs cumulatifs
Append	Ajoute de nouvelles lignes	Historiques, enrichissement
Update	Met à jour les lignes existantes	Tableaux de bord

Choisir le bon mode

- Type de requête (agrégation, enrichissement).
- Contraintes de performance (débit, latence).
- Exigences de stockage.

1.3. Watermarking et Données Tardives

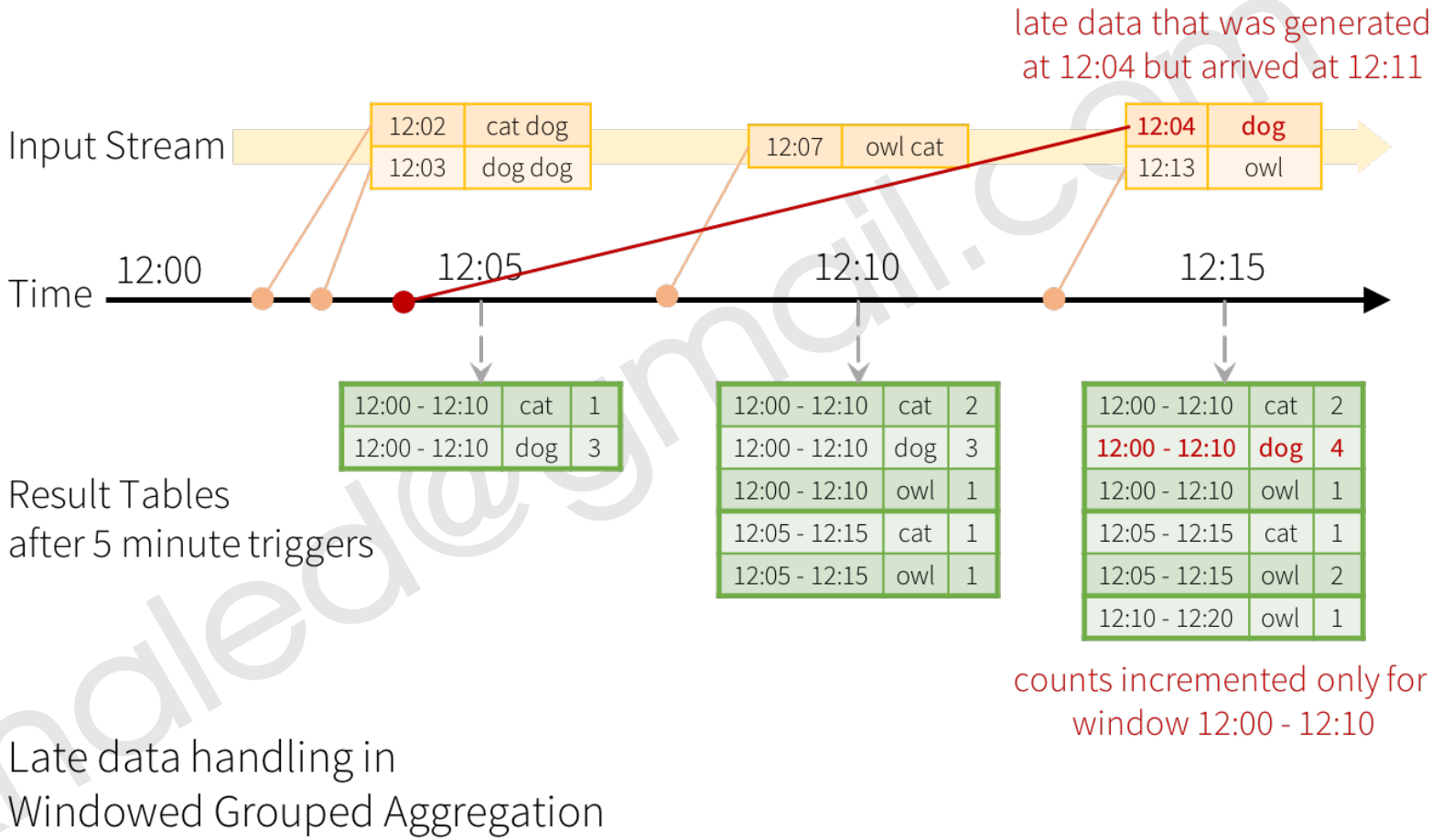
Problème des **données tardives** : Dans de nombreuses applications (ex: IoT), les données peuvent arriver en **retard**.

Watermarking

- Le **watermark** est un **seuil glissant** qui indique combien de **retard** est acceptable pour un événement.
- Les **données arrivant après ce seuil** sont ignorées pour éviter que l'état ne croisse indéfiniment.

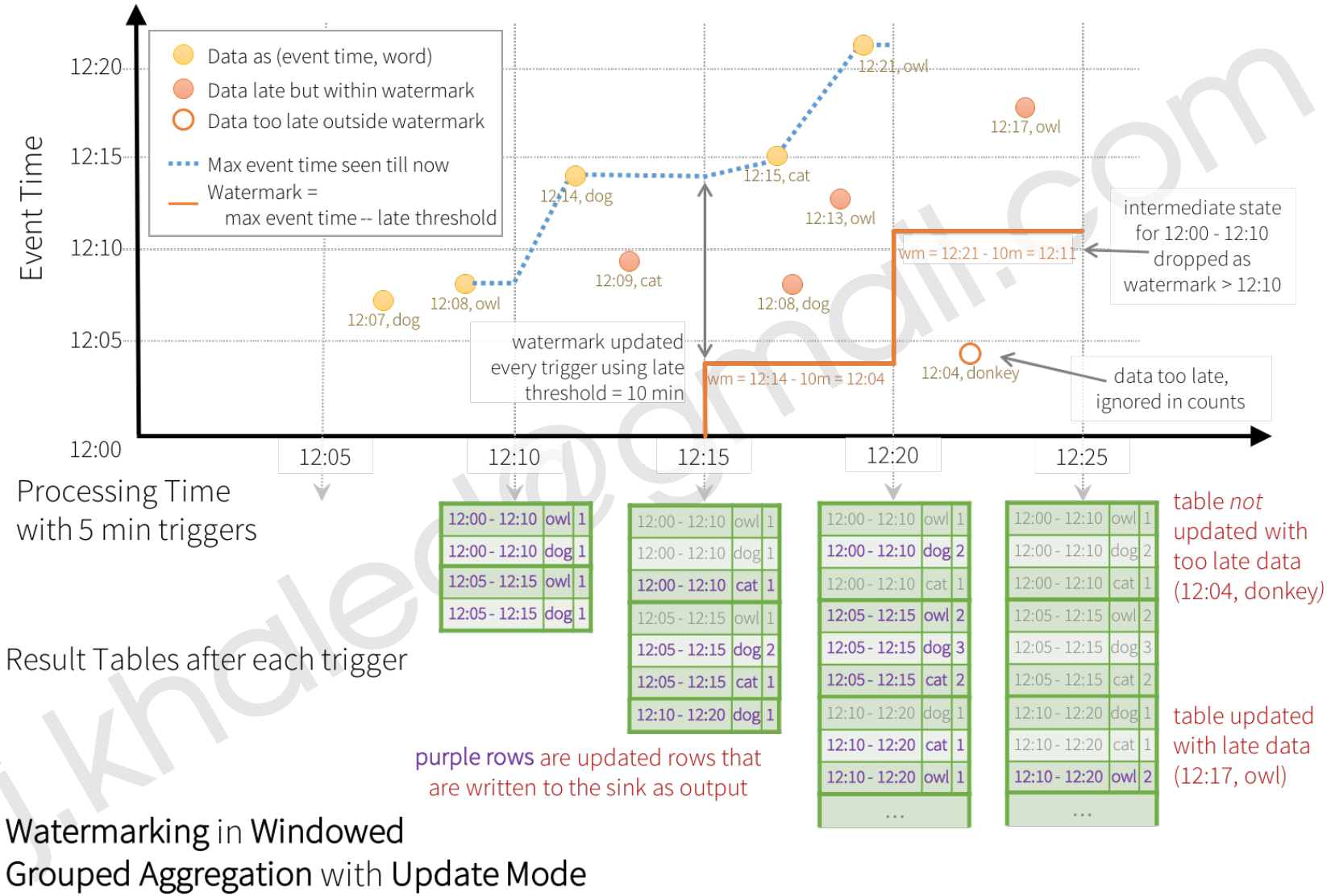
```
# timestamp ici est l'estampille temporelle associ e    la donn e
watermarkedDF = df.withWatermark("timestamp", "10 minutes")
```

1.3. Watermarking et Données Tardives



(Source : Documentation officielle d'Apache Structured Streaming)

1.3. Watermarking et Données Tardives



(Source : Documentation officielle d'Apache Structured Streaming)

1.3. Exactly-Once Processing

- **Tolérance aux pannes** : Le système continue à fonctionner correctement même après une panne partielle ou complète, sans que les données ne soient ni **perdues**, ni **corrompues**.
- Les sources de données comme **Kafka** ou **Kinesis** permettent de rejouer les données depuis un point donné (**offset**).
- Cela garantit que les données peuvent être relues si une panne survient avant la fin du traitement.
- Le **Traitement exactly-once** signifie que chaque élément de données est traité une seule fois, sans être omis ni traité deux fois, **même en cas de panne**.
- Essentiel pour assurer la **consistance des résultats** dans les systèmes de traitement en streaming, où les erreurs ou interruptions sont fréquentes.

1.3. Tolérance aux Pannes

Mécanismes Clés pour le *Exactly-Once Processing*

1. **Checkpointing** : enregistrement régulier de l'état intermédiaire du traitement dans un **répertoire de checkpoint**.

```
resultStreamDF.writeStream.format("parquet")
    .option("checkpointLocation", "/path/to/checkpoint/dir")
    .start("/path/to/output/dir")
```

2. **Write-Ahead Logging (WAL)** (journalisation anticipée) : les modifications de l'état sont écrites dans un journal avant d'être appliquées.

3. **Offset Tracking**
   - Spark suit les **offsets** (positions des données) des éléments traités.
   - En cas de panne, Spark peut reprendre exactement au dernier **offset validé**, sans perdre ou retraiter des données.

1.3. Processus de Reprise après Panne

Reprise sur panne

- Lors d'une panne, Spark redémarre la requête à partir du **dernier checkpoint** enregistré.

- Grâce au **journal WAL** et aux **offsets** conservés, Spark peut rejouer les événements non traités avant la panne.

1.3. Processus de Reprise après Panne

Reprise sur panne - Exemple Word Count

- **Checkpointing** assure que l'état (le total des mots comptés) est sauvegardé périodiquement, permettant une reprise à partir du dernier état enregistré.
- **Offset Tracking** garde la trace des positions dans le flux de données (Kafka), assurant qu'aucun message n'est relu ni perdu après une panne.
- **WAL** garantit que toutes les modifications planifiées (incrémenter le compteur d'un mot) sont enregistrées avant d'être appliquées, assurant une reprise correcte.
- **Reprise après panne** permet à Spark de reprendre de manière transparente à partir de l'endroit exact où il s'était arrêté, sans affecter la consistance du calcul.

1.4. Concepts Clés de l'Agrégation Temporelle

1. Qu'est-ce que l'Agrégation Temporelle ?

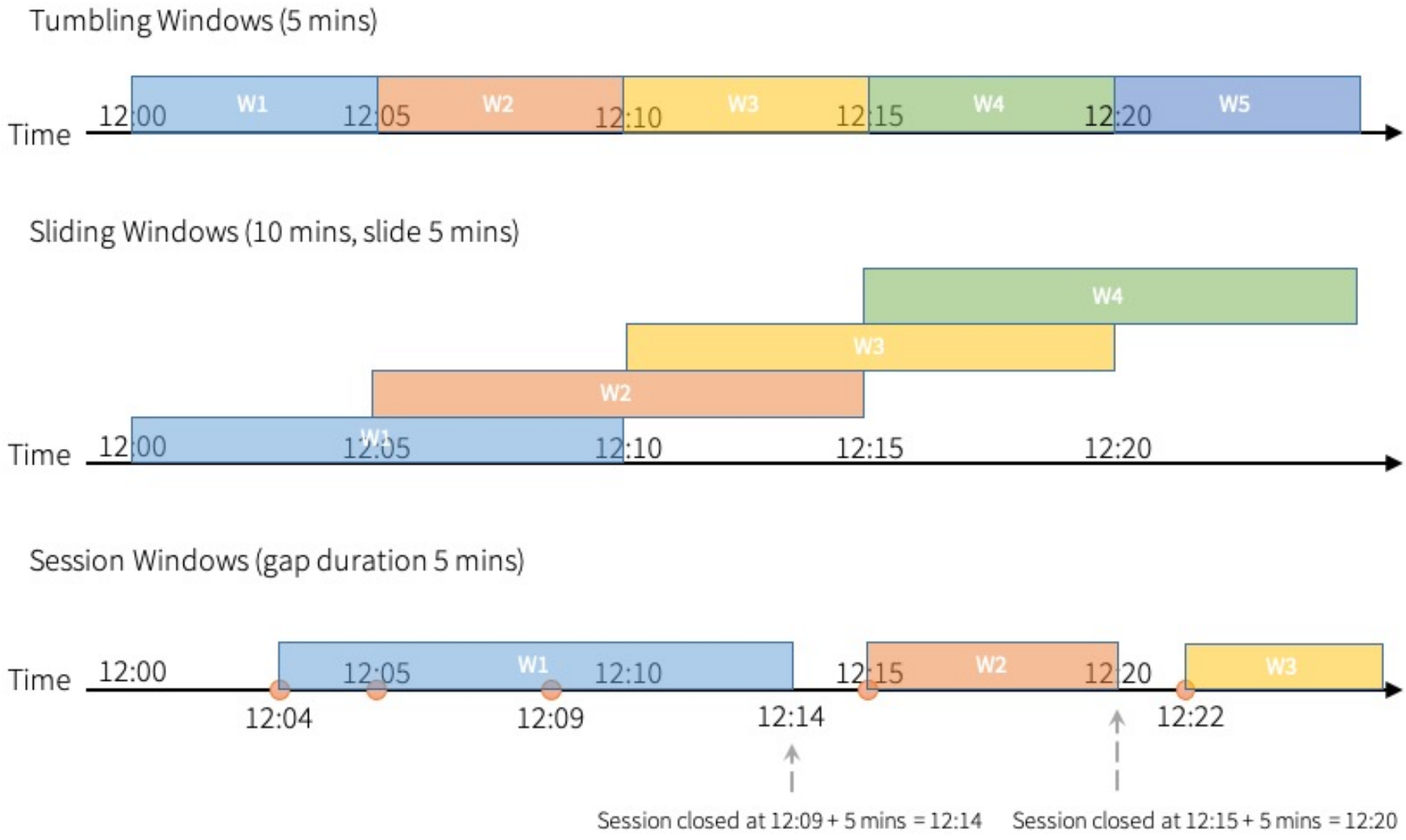
- Regrouper des données en fonction du temps pour obtenir une vue synthétique.
- Très fréquente dans les requêtes sur les flux.
- Exemple: Nombre de clics par heure.
- Pourquoi ? Simplifier, identifier des tendances, optimiser, décider.

2. Types d'Agrégation Temporelle :

- **Fenêtres sautantes** (*Tumbling Windows*) : Périodes de temps fixes, **sans chevauchement**, chaque événement appartient à une seule fenêtre.
- **Fenêtres Glissantes** (*Sliding Windows*) : Fenêtres qui se chevauchent, chaque événement peut appartenir à plusieurs fenêtres.
- **Fenêtres de Session** : Basées sur l'activité des événements avec une période d'inactivité définie. Exemple : Suivre les sessions utilisateurs sur un site web.

1.4. Agrégation temporelle

Types de fenêtre



(Source : Documentation officielle d'Apache Structured Streaming)

1.4. Fenêtrage PySpark SQL-Like

Tumbling Window

```
wordCounts = spark.sql("
SELECT window.start AS window_start, window.end AS window_end,
    word, COUNT(*) AS word_count FROM words
GROUP BY window(timestamp, '10 seconds'), word")
wordCounts.writeStream.start()
```

Sliding Window

```
wordCounts = spark.sql("
SELECT window.start AS window_start, window.end AS window_end,
    word, COUNT(*) AS word_count FROM words
GROUP BY window(timestamp, '10 seconds', '5 seconds'), word")
wordCounts.writeStream.start()
```

Session Windows

```
wordCounts = spark.sql("
SELECT session_window.start AS session_start, session_window.end
     AS session_end, word, COUNT(*) AS word_count FROM words
GROUP BY session_window(timestamp, '30 minutes'), word")
wordCounts.writeStream.start()
```

1.5. Utilisation d'un modèle prédictif sur un flux de données - Exemple Spark ML

Important : l'**apprentissage n'est pas incrémental**, mais par lots! **seule l'inférence est en ligne**.

Même API que l'inférence batch.

```
# Charger le mod le pr -enregistr
model = LinearRegressionModel.load("/path/to/saved/model")

# Lire les donn es en flux    partir d'un r pertoire de
    fichiers JSON
streamingDF = spark.readStream.format("json").schema("date DATE,
     country STRING, cases INT").load("/path/to/stream/data")

# Appliquer le mod le au flux
predictions = model.transform(streamingDF)
```

- Le modèle est appliqué aux données de flux en temps réel pour faire des prédictions.
- Bibliothèques d'apprentissage incrémental : River, Scikit-multiflow, MOA, etc.

1.6. Exemple

- La **source d'un flux** peut être : **fichiers** ou **répertoires de fichiers** (text, csv, json, parquet, etc.), **Kafka** (nous y reviendrons), socket, etc.

- Dans notre **exemple** nous allons utiliser un Netbook Databriks et lire les données à partir d'un **répertoire DBFS contenant des fichiers JSON**.

```
%fs ls /databricks-datasets/structured-streaming/events/
```

path	name	size
dbfs:/databricks-datasets/structured-streaming/events/file-0.json	file-0.json	72530
dbfs:/databricks-datasets/structured-streaming/events/file-1.json	file-1.json	72961
dbfs:/databricks-datasets/structured-streaming/events/file-10.json	file-10.json	73025

- Les fichiers contiennent des **évènements "Open" / "Close" d'objets connectés**.

```
%fs head /databricks-datasets/structured-streaming/events/file-0.json
```

```
{"time":1469501107,"action":"Open"}
{"time":1469501147,"action":"Open"}
{"time":1469501202,"action":"Open"}
{"time":1469501219,"action":"Open"}
```

Note : les fichiers sont lus dans l'ordre de leur création et il est possible d'ajouter de nouveau fichiers au répertoire après le lancement du traitement

1.6. Exemple batch 1/2

https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/5465058377779245/4000060192685969/2669555599124617/latest.html

1. Définition du dataframe (et de son schéma)

```
from pyspark.sql.functions import *
from pyspark.sql.types import *

inputPath = "/databricks-datasets/structured-streaming/events/"

jsonSchema = StructType().add("time",TimestampType()).add("action",StringType())

staticInputDF = spark.read.json(inputPath,jsonSchema)
```

- Si des données (des fichiers) sont ajoutés ultérieurement à la source de données (au répertoire), elles ne seront pas prises en compte (à moins de réexécuter la lecture manuellement

2. Interrogation du dataframe avec les opérateurs SQL

```
totalCountsDF = staticInputDF.groupBy(staticInputDF.action).count()
```

2. Ou bien alternativement, interrogation à travers une vue

```
staticInputDF.createOrReplaceTempView("staticCounts")
```

```
%sql
select action, count(*) from staticCounts group by staticCounts.action
```

1.6. Exemple batch 2/2

https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/5465058377779245/4000060192685969/2669555599124617/latest.html

3. Utilisation du fenêtrage sur des données batch : nombre d'open/close par heure

```
staticWindowedCountsDF = staticInputDF
    groupBy(staticInputDF.action,window(staticInputDF.time, "1 hour"))
    .count()
```

4. Interrogation

```
staticWindowedCounts.createOrReplaceTempView("windowedCounts")
```

```
%sql
select action, date_format(window.end, "MMM-dd HH:mm") as time, count
                    from windowedCounts
                    order by window.end, action
```

5. Il est également possible d'écrire le contenu du dataframe vers un sink

```
staticWindowedCounts.write...
```

1.6. Exemple streaming sans fenêtrage

1. Définition du flux (et de son schéma)

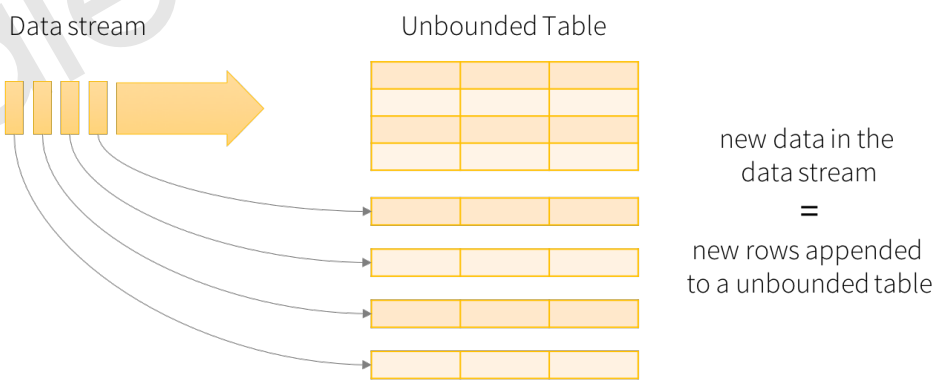
```
from pyspark.sql.functions import *
from pyspark.sql.types import *

inputPath = "/databricks-datasets/structured-streaming/events/"

jsonSchema = StructType().add("time",TimestampType()).add("action",StringType())

streamingInputDF = spark.readStream.option("maxFilesPerTrigger", 1)
                        .json(inputPath,jsonSchema)
```

- Le flux est représenté sous la forme d'un Dataframe ou d'une table infinie.
- La seule différence avec un traitement batch est qu'à la différence de read, **readStream** va constamment consulter la source de données (le répertoire dans notre cas) à la recherche de nouvelles données (fichiers dans notre cas)



Data stream as an unbounded table

1.6. Exemple streaming sans fenêtrage

https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/5465058377779245/4000060192685989/2669555599124617/latest.html

2. Interrogation du dataframe avec les opérateurs SQL de la même manière qu'avec un DataFrame ordinaire. La seule différence est que le résultat de la requête évolue au fur et à mesure de la lecture du flux

```
totalCountsDF = streamingInputDF.groupBy(streamingInputDF.action).count()
```

2. Ou bien alternativement, interrogation à travers une vue

```
streamingInputDF.createOrReplaceTempView("streamingCounts")
```

```
%sql
select action, count(*) from streamingCounts group by streamingCounts.action
```

1.6. Exemple streaming avec fenêtrage

2. Découpage du flux et regroupent des évènements

- Avant de démarrer la lecture du flux nous avons besoin de définir la manière avec laquelle le flux sera découpé en groupes (le type et la durée de la fenêtre temporelle)

- Notre requête consiste à grouper les évènements par action (open/close) et par fenêtre d'une heure. On comptabilise ensuite les évènements d'un même groupe.

```
streamingCountsDF = (
    streamingInputDF
      .groupBy(streamingInputDF.action, window(streamingInputDF.time, "1 hour"))
      .count()
)
```

- La fenêtre que nous venons de définir est une tumbling window. On aurait pu définir une **sliding window** comme suit

```
.groupBy(.., window(streamingInputDF.time, "1 hour", "30 minutes"))
```

1.6. Exemple écriture de l'output du flux

3. De la même que pour le batch, le résultat d'une requête continue peut être écrit dans un sink

```
query = (streamingCountsDF
        .writeStream.format("memory")
        .queryName("counts")
        .outputMode("complete")
        .trigger(processingTime='5 seconds')
        .start())
```

- `writeStream.format("sink")` : "sink" ou là où l'output du flux sera redérigé (peut également être une base, un flux Kafka, etc.)
- `queryName("counts")` : nom de la table ou du dataframe où l'output est écrit. Nécessaire de lui affecter un nom pour pouvoir l'interroger par la suite
- `outputMode("complete")` : tout l'output est retransmis à la table "counts". Il y a également un mode update et un mode append (nous y reviendrons)
- `trigger(processingTime='5 seconds')` : fréquence de rafraîchissement de la table (recalcule de count). Par défaut "0", signifiant traiter les évènements aussi rapidement que possible.

1.6. Exemple

4. Interrogation de l'output du flux

```
%sql
select action, date_format(window.end, "MMM-dd HH:mm") as time, count
          from counts order by time, action
```

- Les partitions du flux peuvent être interrogés comme n'importe quelle table Spark

Quiz

1. Quelle est la principale différence entre le **stream processing** et le **batch processing** ?

2. Quelle est la différence entre les modes **Append** et **Update** ?

3. Qu'est-ce que l'**agrégation temporelle** dans le stream processing ?

4. Qu'est ce qu'un micro-batch? C'est quoi le lien entre les **triggers** et les **micro-batchs** ?

5. Expliquez en quoi consiste le **Watermarking** dans le stream processing ?

6. Expliquez brièvement ce qu'est le traitement "**Exactly-once**" et les mécanismes permettant de l'assurer.

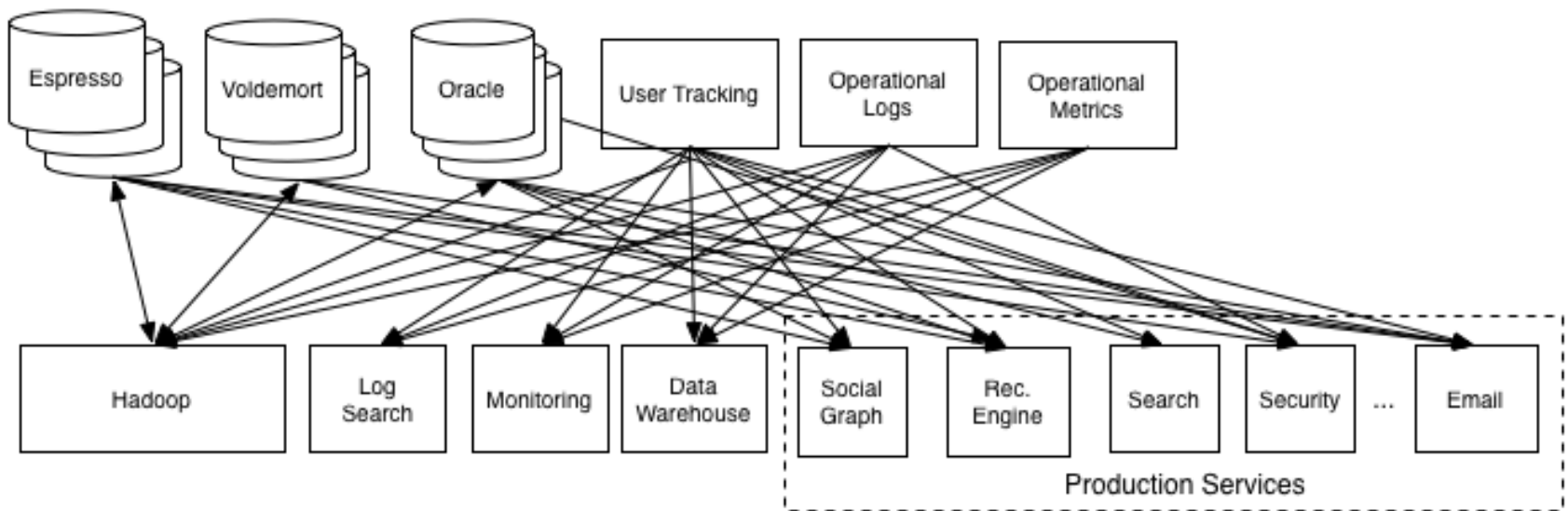
7. Qu'est ce que le **checkpointing** et le **write Ahead Log**? Quels rôles ont ils dans le traitement des flux.

8. Expliquez comment se fait la **reprise sur panne** dans un système de traitement de flux? Pourquoi est il nécessaire de faire de l'**offset tracking**?

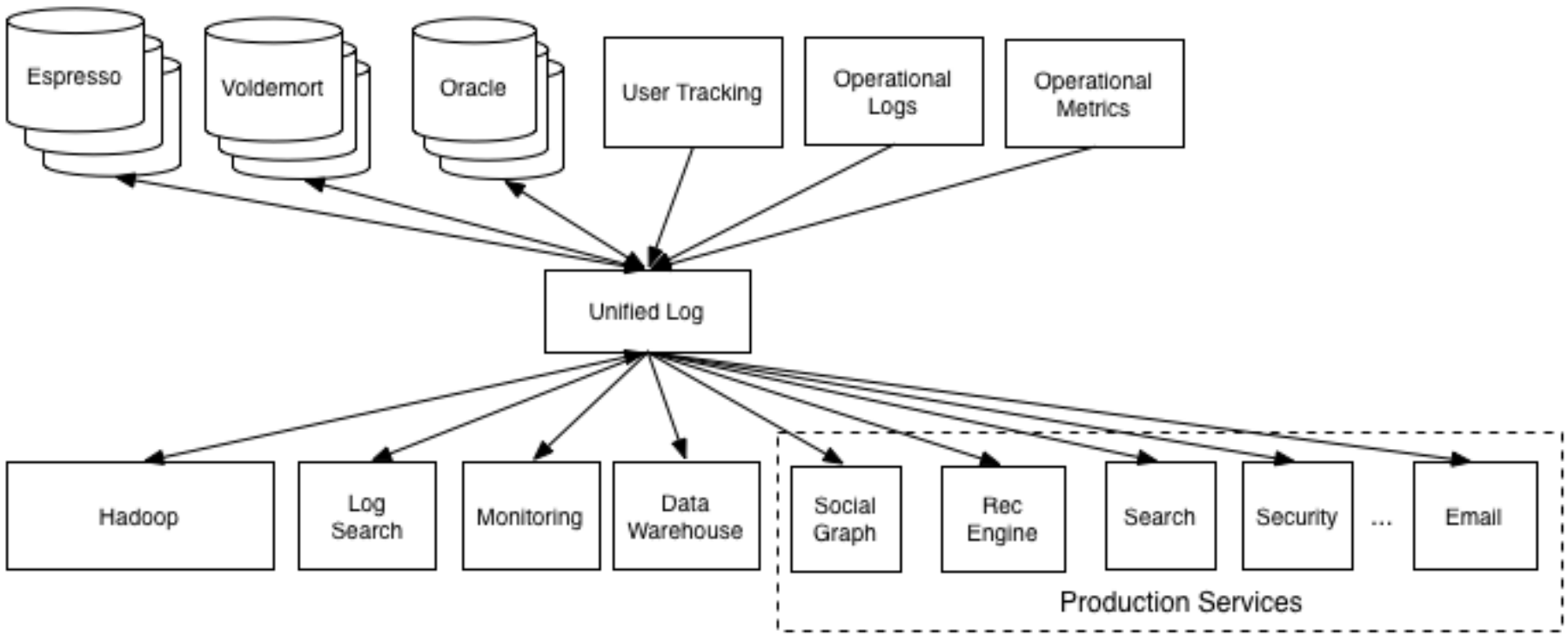
Section 2 - 2. Streaming Pipeline - Étude de cas avec Kafka

2. Apache Kafka, quésaco?

- Plateforme **distribuée** de streaming en **temps réel** développée initialement par LinkedIn.
- Conçu pour transmettre et gérer des **flux massifs de données** provenant de **multiples sources**.
- Autres systèmes : Apache Pulsar, Amazon Kinesis, Google Cloud Pub/Sub, RabbitMQ,



2.1. Streaming pipeline avec Kafka



2.1. Cas d'utilisation courants de Kafka

- **Pipelines de traitement de données** : Acheminer des données en temps réel entre les systèmes (bases de données, systèmes de fichiers, etc.).

- **Monitoring en temps réel** : Analyser des logs, des événements, ou des métriques à la volée pour la supervision des systèmes.

- **Applications événementielles** : Kafka permet la gestion des événements pour des architectures orientées événements, assurant une communication fluide entre les microservices.

2.2. Les Composants de Kafka

Topic : les records publiés dans Kafka sont groupés par sujet (similaire aux tables).

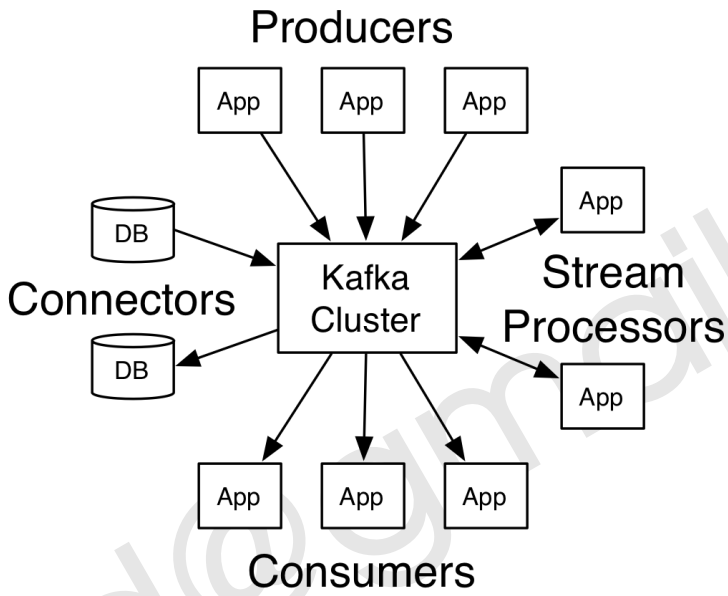
Producer : Tout système qui écrit des messages (enregistrements) dans un **topic**.

Consumer : Tout système qui lit les messages d'un ou plusieurs **topics**.

Broker : Serveur Kafka qui stocke les données et envoie les messages aux consumers. Plusieurs brokers forment un **cluster Kafka**.

Partition : Les topics sont divisés en **partitions** pour améliorer le parallélisme.

2.3. Partitionnement des topics



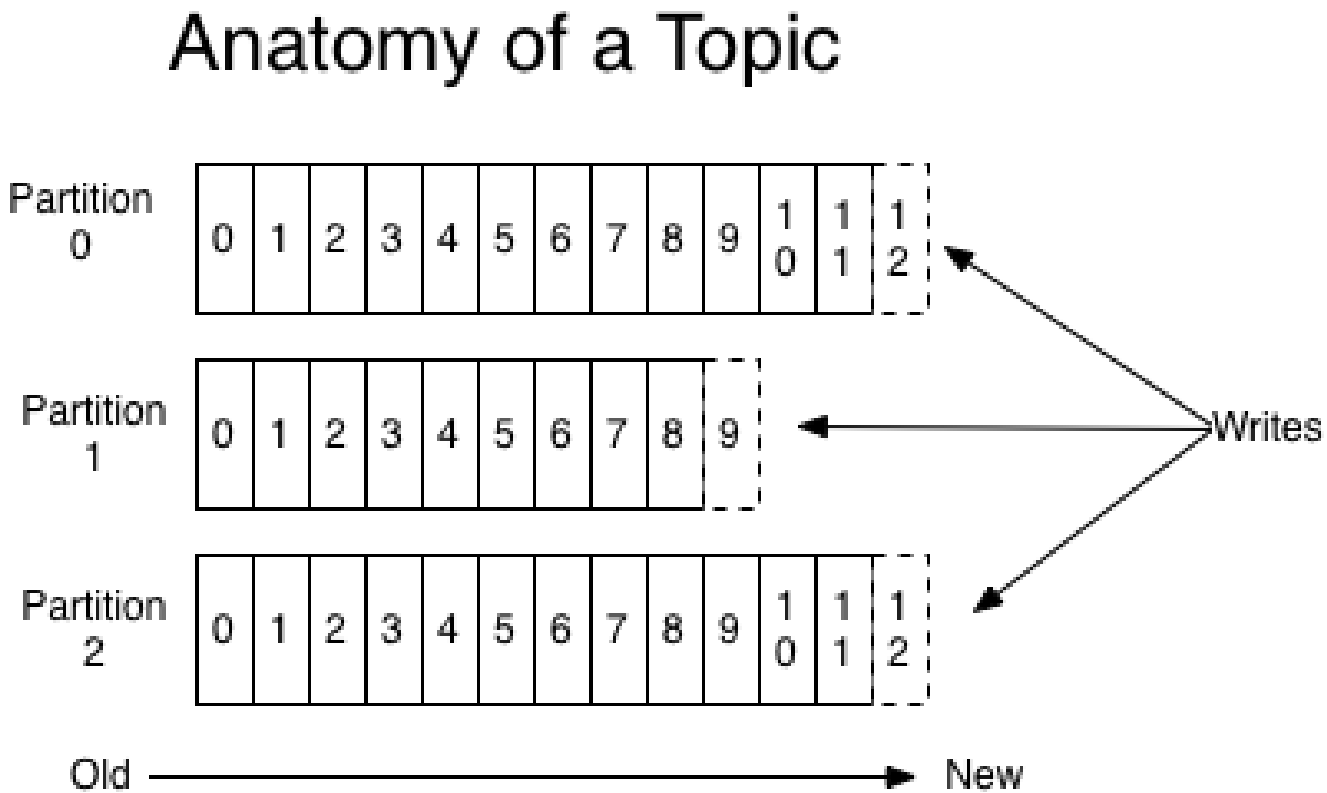
Différentes manières pour ajouter les records aux partitions d'un topic

- **Aléatoirement** : pour chaque message, une partition est choisie au hasard.
- **Round robin** : le producer itére sur les partitions les unes après les autres pour distribuer un nombre de message égal sur chaque partition.
- **Hashage** : le producer peut choisir une partition en fonction du contenu du message.

2.3. Partitionnement des topics

Offset : chaque record est identifié par un numéro de séquence dans la partition où il est enregistré

Les partitions sont des séquences **immuables** de records. On ne peut qu'ajouter des records à la fin de la partition (ou y lire).



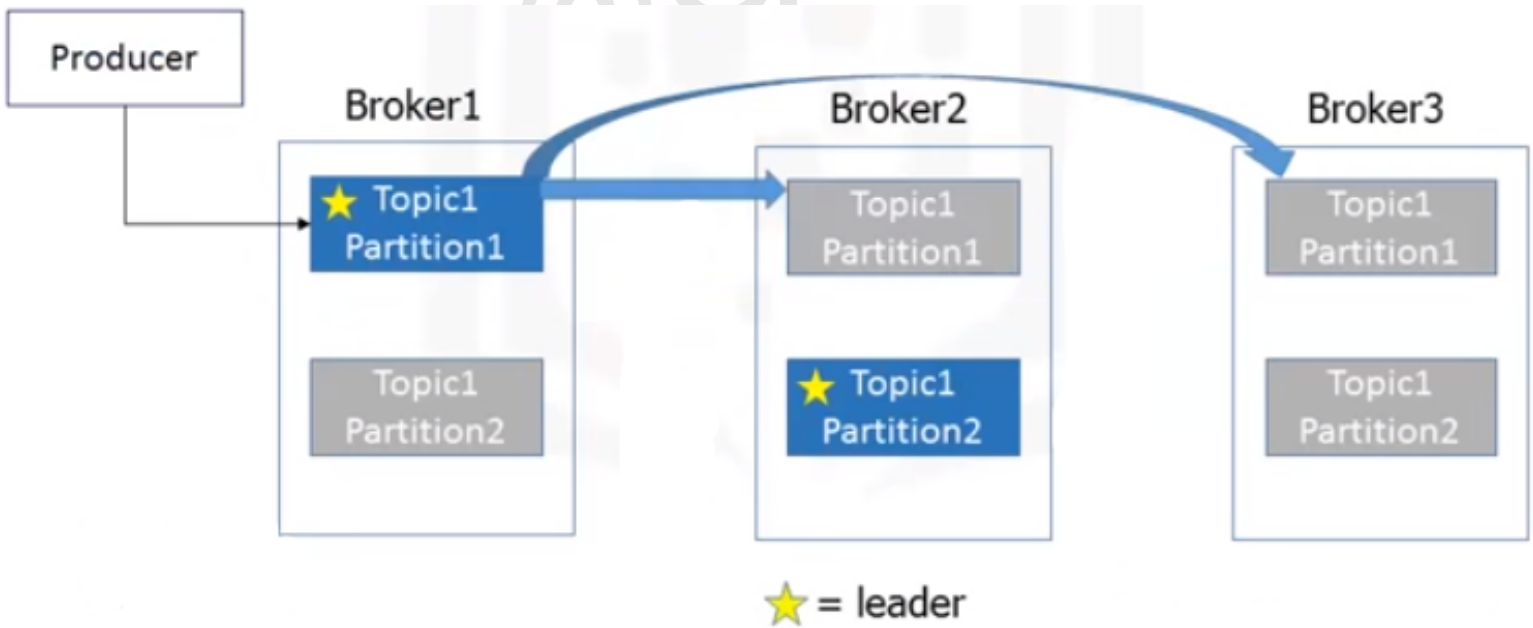
2.3. Réplication des topics

Kafka **réplique** les partitions sur plusieurs brokers pour garantir la **disponibilité** et la **tolérance aux pannes**. La réplication est **master/slave asynchrone**.

Chaque partition a un **leader** (master) et un ou plusieurs **followers** (slaves).

Le **leader** gère toutes les opérations de lecture et d'écriture, tandis que les **followers** répliquent les données du leader (des stand-by).

En cas de panne du leader, un follower est élu nouveau leader (**RAFT**).



2.3. In-Sync Replicas (ISR)

Exemple de création d'un topic avec un facteur de réplication dans un cluster Kafka :

```
kafka-topics.sh --create --topic example-topic \
--replication-factor 3 --partitions 3 \
--bootstrap-server localhost:9092
```

Kafka maintient une liste de répliques synchronisées avec le leader appelée **In-Sync Replicas** (ISR).

Les **ISR** sont les répliques actives qui sont à jour avec le leader. Si un follower prend trop de retard, il est retiré de l'ISR.

Lorsqu'un leader tombe en panne, un des followers de l'ISR est élu comme nouveau leader.

2.4. Modes de livraison

At-most-once

- Livraison unique : les messages peuvent être perdus mais jamais redélivrés.
- L'offset est enregistré avant traitement. En cas de panne, le message est perdu.

At-least-once

- Livraison garantie avec risques de **doublons**.
- L'offset est mis à jour après traitement. En cas de panne, le message est retraité.

Exactly-once

- Livraison et traitement **une seule fois**, même en cas de panne.
- Utilise des **transactions** pour lier traitement et mise à jour de l'offset. Si la transaction échoue, le message sera retraité.
- Les producteurs idempotents évitent les doublons. Offre une fiabilité maximale avec une légère latence accrue.

2.4. Modes de livraison - Exemple At-Most-Once

Kafka produit un message : Un producteur envoie un message dans un topic Kafka.

Structured Streaming consomme le message : Le moteur de streaming lit le message depuis Kafka.

At-Most-Once - Commit de l'offset avant traitement

- L'offset est validé à Kafka avant le traitement du message.
- Kafka peut le cas échéant supprimer le message pour libérer la mémoire.

Traitement : Le moteur de streaming applique des transformations ou agrégations au message. Si une panne survient, le message ne sera pas relu, car l'offset est déjà validé.

Risques : Ce mode peut entraîner une perte de message si une erreur survient pendant le traitement.

2.4. Modes de livraison - Exemple At-Least-Once

1. **Structured Streaming consomme le message** : Le moteur de streaming lit le message depuis Kafka.

2. **Traitement** : Le moteur de streaming traite le message.

3. **At-Least-Once - Commit de l'offset après traitement**

   - L'offset du message est commit à Kafka seulement après un traitement réussi.

   - Si une panne survient avant la validation, le message sera relu au redémarrage du moteur, entraînant des doublons possibles.

2.4. Modes de livraison - Exemple Exactly-Once

1. **Lecture** : Spark Structured Streaming consomme le message depuis Kafka.

2. **Traitement** : Le moteur applique les transformations ou agrégations nécessaires sur le message.

3. **Exactly-once - Commit de l'offset** : toutes les opérations liées à ce message (lecture, traitement, mise à jour de l'offset) sont **effectuées soit toutes, soit annulées en bloc.**

Résultat : Cette gestion par transaction assure qu'un message est livré et traité exactement une fois, même en cas de panne.

2.5. RAFT dans Kafka (KRaft)

KRaft est la nouvelle architecture de Kafka, introduite à partir de Kafka 2.8.0, qui remplace **Zookeeper**.

Zookeeper était utilisé pour gérer les métadonnées de Kafka et coordonner les nœuds du cluster. Avec **KRaft**, cette gestion est internalisée grâce à l'algorithme **RAFT**.

RAFT permet à Kafka de gérer les **élections de leader** et la **réplication des métadonnées**.

Papier introduisant RAFT (https://raft.github.io/) (0014) **Ongaro, D., & Ousterhout, J. (2014). In Search of an Understandable Consensus Algorithm (Extended Version). In Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 2014) (pp. 305{319). USENIX Association.**

2.5. Algorithme de Consensus RAFT - Élection du leader

Détection des pannes

- Chaque follower dispose d'un **timeout** (*e.g.* entre 150 et 300 ms).
- Si le follower ne reçoit pas de heartbeat pendant ce délai, il considère que le leader est défaillant et doit être remplacé.
- Le follower se transforme alors en **candidat** et lance une **élection** pour tenter de devenir le nouveau leader.
- Un candidat lance une élection en augmentant son **terme** et en envoyant des **requêtes de vote** (RequestVote) à tous les nœuds.

2.5. Algorithme de Consensus RAFT - Élection du leader

Détection des pannes

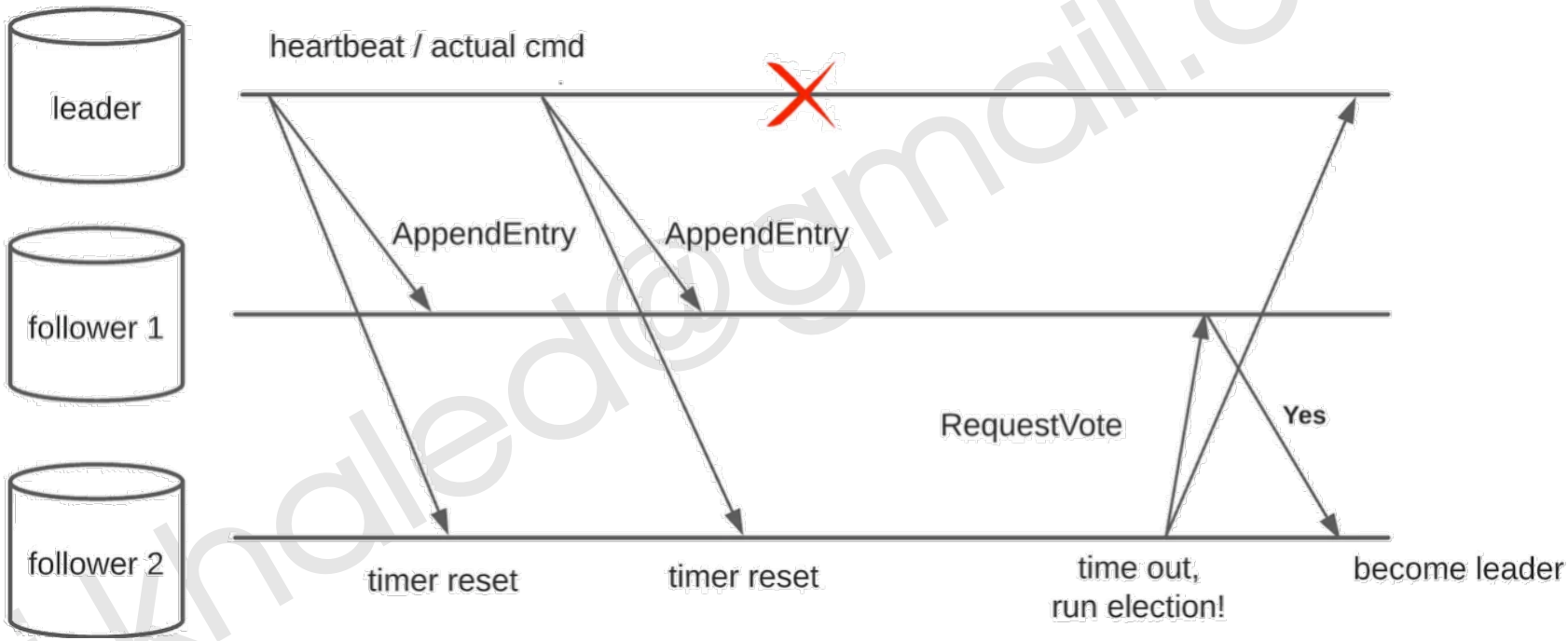


Figure reprise de (OO14)

2.5. Algorithme de Consensus RAFT - Élection du leader

Vote d'un nœud : Un nœud recevant une **requête de vote**

- Vérifie si le **terme du candidat est supérieur** au sien. Si c'est le cas, il vote pour le candidat.
- Vérifie si son **log est plus à jour** que celui du candidat. Si c'est le cas, il refuse de voter.
- Ne **vote qu'une seule fois par terme**.

Atteinte du quorum : Un candidat devient leader lorsqu'il reçoit des votes de la **majorité des nœuds** (quorum).

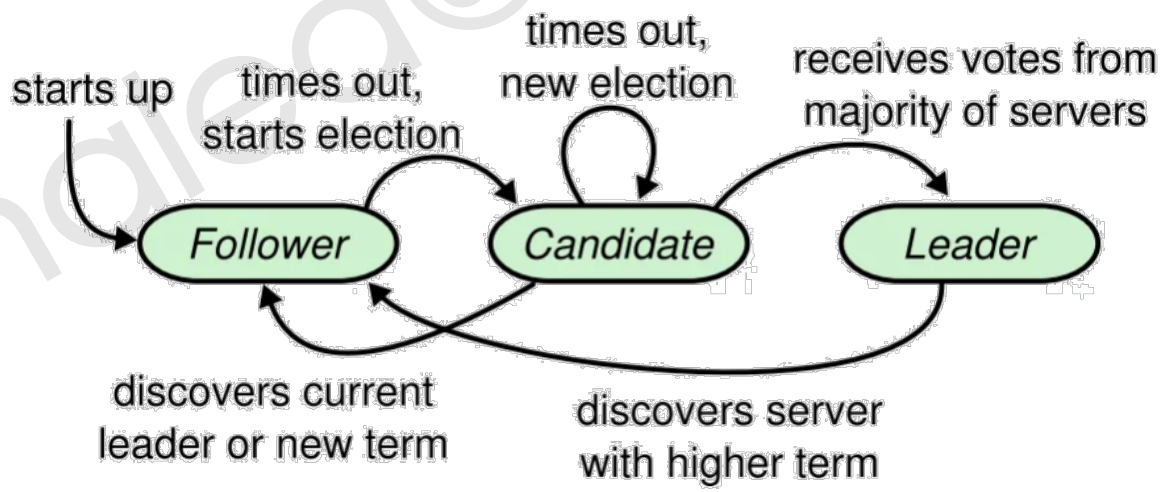


Figure reprise de (OO14)

2.5. Algorithme de Consensus RAFT - Gestion des Termes

- **Définition d'un terme :** Un terme dans Raft représente une unité de temps logique pendant laquelle un leader est élu. Chaque terme commence soit par une élection, soit par la continuité du leader précédent.
- **Mise à jour du terme :** Chaque nœud conserve un numéro de terme qui est incrémenté lors d'une élection ou lorsqu'il reçoit un message d'un nœud avec un terme supérieur. Le leader diffuse son terme lors de la réplication du log, permettant aux autres nœuds de se synchroniser.
- **Vote unique par terme :** Pour prévenir les conflits de leadership, chaque nœud ne peut voter qu'une seule fois par terme. Un nœud vote pour le candidat ayant le terme le plus élevé et un log au moins aussi à jour.
- **Cohérence des journaux :** Un leader élu dans un terme plus élevé dispose d'un journal plus récent et plus complet, assurant ainsi la cohérence des données dans tout le cluster. Avant d'être élu, un candidat doit démontrer qu'il a appliqué toutes les entrées du log des autres nœuds jusqu'à un certain index.

2.5. Exemple d'Élection RAFT - Configuration Initiale et Détection de la Panne

Configuration initiale :

- **Nœud A :** Leader, terme 1
- **Nœuds B, C, D, E :** Followers, terme 1

Défaillance du leader : Le nœud A tombe en panne.

Détection de la panne :

- **Nœud B** détecte en premier l'absence de heartbeat de A.
- **Nœud C** détecte ensuite la défaillance de A.

Important : les **timeouts sont différents** d'un nœud à un autre dans RAFT (randomized timeouts)

2.5. Exemple d'Élection RAFT - Scénario 1 : un candidat obtient le quorum

Étape	Nœud A	Nœud B	Nœud C	Nœud D	Nœud E
Initial	Leader (terme 1)	Follower (terme 1)	Follower (terme 1)	Follower (terme 1)	Follower (terme 1)
Détection de la panne par B	-	**Terme 2, candidat**	Follower (terme 1)	Follower (terme 1)	Follower (terme 1)
Détection de la panne par C	-	Terme 2, candidat	Terme 2, candidat	Follower (terme 1)	Follower (terme 1)
Votes de D et E pour B	-	**Terme 2, leader**	Terme 2, follower	Terme 2, a voté pour B	Terme 2, a voté pour B
Requête de vote de C	-	Leader	Terme 2, follower	Terme 2, a voté pour B	Terme 2, a voté pour B

2.5. Exemple d'Élection RAFT - Scénario 2 : Vote divisé

Étape	Nœud A	Nœud B	Nœud C	Nœud D	Nœud E
Initial	Leader (terme 1)	Follower (terme 1)	Follower (terme 1)	Follower (terme 1)	Follower (terme 1)
Détection de la panne par B	-	Terme 2, **candidat**	Follower (terme 1)	Follower (terme 1)	Follower (terme 1)
Détection de la panne par C	-	Terme 2, candidat	Terme 2, **candidat**	Follower (terme 1)	Follower (terme 1)
Vote de D pour B	-	Terme 2, candidat	Terme 2, candidat	Terme 2, a voté pour B	Follower (terme 1)
Vote de E pour C	-	Terme 2, candidat	Terme 2, candidat	Terme 2, **a voté pour B**	Terme 2, **a voté pour C**

Résolution du vote divisé : Dans ce cas, aucun candidat n'a obtenu le quorum (3 votes). RAFT résout cette situation en utilisant des **timeouts d'élection aléatoires**, qui permettent à un candidat de déclencher la prochaine élection légèrement avant l'autre.

2.6. Session Illustrative - Télécharger l'image Kafka et lancer un broker

- Il existe une image Docker officielle de Kafka avec laquelle vous pouvez tester les différentes opérations.
  https://hub.docker.com/r/apache/kafka

```
docker pull apache/kafka
```

Listing 1: Téléchargement de l'image Kafka

- Une fois l'image téléchargée, vous pouvez démarrer un *broker Kafka* dans un conteneur Docker.

```
docker run -d --name broker apache/kafka:latest
```

Listing 2: Démarrer un broker Kafka

- Pour interagir avec Kafka, vous devez accéder à une console dans le conteneur où Kafka est exécuté.

```
docker exec --workdir /opt/kafka/bin/ -it broker sh
```

Listing 3: Ouvrir une console dans le conteneur

2.6. Session Illustrative - Créer un topic Kafka

- Depuis la console ouverte dans le conteneur, créez un topic appelé `test-topic` avec la commande suivante :

```
./kafka-topics.sh --bootstrap-server localhost:9092 \
    --create --topic test-topic
```

Listing 4: Créer un topic Kafka

- Vous pouvez maintenant produire des événements dans le topic `test-topic` à l'aide du producer Kafka.
- Exécutez la commande suivante pour lancer un producer Kafka :

```
./kafka-console-producer.sh --bootstrap-server localhost:9092 \
    --topic test-topic
```

Listing 5: Lancer un producer Kafka

- Le prompt `>` apparaîtra. Tapez `hello`, appuyez sur Entrée, puis tapez `world` et appuyez à nouveau sur Entrée.

2.6. Session Illustrative - Consommer les événements du topic

- Pour lire les messages produits dans le topic `test-topic`, utilisez le consumer Kafka.
- Exécutez la commande suivante pour lire les messages depuis le début du journal :

```
docker exec --workdir /opt/kafka/bin/ -it broker sh
```

Listing 6: Lancer un shell pour le consumer

```
./kafka-console-consumer.sh --bootstrap-server localhost:9092 \
    --topic test-topic --from-beginning
```

Listing 7: Consommer les messages du topic

- Vous verrez les deux messages produits : `hello` et `world`.
- Une fois que vous avez terminé d'utiliser Kafka, vous pouvez arrêter et supprimer le conteneur Docker pour libérer des ressources.
- Exécutez la commande suivante sur votre machine hôte pour arrêter et supprimer le conteneur Kafka :

```
docker rm -f broker
```

Listing 8: Supprimer le conteneur Kafka

2.6. Session Illustrative - Intégration de Kafka et Spark Structured Streaming

- Spark Structured Streaming peut consommer directement des flux de données en provenance de Kafka.
- Cela permet de traiter et d'analyser des données en temps réel tout en profitant des capacités de scalabilité de Kafka et de Spark.
- **Avantages de l'intégration** :
  - Scalabilité pour traiter des volumes massifs de données.
  - Faible latence pour des pipelines de données temps réel.
  - Tolérance aux pannes grâce à Kafka (réplication) et à Spark (checkpointing).

2.6. Session Illustrative - Consommer des données Kafka avec Spark

- Exemple de code pour lire un flux de données Kafka dans Spark Structured Streaming :

```
df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "test-topic") \
    .load()

kafkaDF = df.selectExpr("CAST(key AS STRING)", "CAST(value AS
    STRING)")

query = kafkaDF.writeStream \
    .outputMode("append") \
    .format("console") \
    .start()

query.awaitTermination()
```

2.6. Session Illustrative - Étapes principales pour intégrer Kafka avec Spark

- **1. Configurer Kafka** : Définir les brokers Kafka et les topics.
- **2. Lire depuis Kafka** : Utiliser le connecteur Kafka de Spark pour lire des flux de données.
- **3. Traiter les données** : Utiliser les transformations Spark comme `select`, `filter`, ou `groupBy` pour analyser les données en temps réel.
- **4. Écrire les résultats** : Les résultats peuvent être écrits dans différents sinks comme la console, les fichiers, ou les bases de données.
- **5. Checkpointing** : Configurer le checkpointing pour garantir la tolérance aux pannes.

Quiz

1. Qu'est-ce que Kafka et pourquoi est-il utilisé dans les pipelines de traitement de données en streaming ?

2. Quels sont les composants principaux de Kafka et quel rôle joue chaque composant ?

3 Expliquez le concept de **partitionnement**. Pourquoi le partitionnement est-il important pour la **scalabilité** ?

4. Qu'est-ce que la **réplication** dans Kafka et comment garantit-elle la tolérance aux pannes ?

5. Expliquez la différence entre **at-most-once**, **at-least-once**, et **exactly-once**.

6. Quelle est la différence entre un **topic** et une **partition** ?

7. Qu'est-ce qu'un **offset** et quelle est son utilité ?

8. Qu'est-ce qu'une **In-Sync Replica (ISR)** et comment fonctionne-t-elle ?

9. Comment fonctionne l'**algorithme de consensus RAFT**? Quel rôle jouent les termes?

Section 3 - 3. Architecture Lambda

3. Architecture Lambda
   - 3.1. Principes
   - 3.2. Limitations de l'Architecture Lambda
   - 3.3. Transition vers l'Architecture Lakehouse

3.1. Qu'est-ce que l'Architecture Lambda ?

- L'**architecture Lambda** est un modèle de traitement de données conçu pour gérer des **grandes quantités de données** à la fois en **batch** et en **temps réel**.

- Elle est souvent utilisée dans des systèmes de traitement de données distribués où les **données en streaming** et les **données historiques** doivent être traitées simultanément.

- Son objectif est de combiner les avantages du traitement par lot pour la fiabilité et le traitement en temps réel pour la latence réduite.

3.1. Fonctionnement de l'Architecture Lambda

Les données entrantes sont envoyées à deux couches distinctes :

- **Batch Layer** : Traite les données par lot et construit des vues à long terme.
- **Speed Layer** : Traite les données en temps réel pour des réponses rapides.

La couche batch est plus précise mais a une latence plus élevée, tandis que la couche speed offre une faible latence mais peut être moins précise.

Serving Layer

- Stocke des données agrégées, transformées et prêtes à être consommées par les applications et les requêtes ad hoc.
- Exemples : NoSQL, Elasticsearch, etc.

3.1. Exemple de Flux Lambda

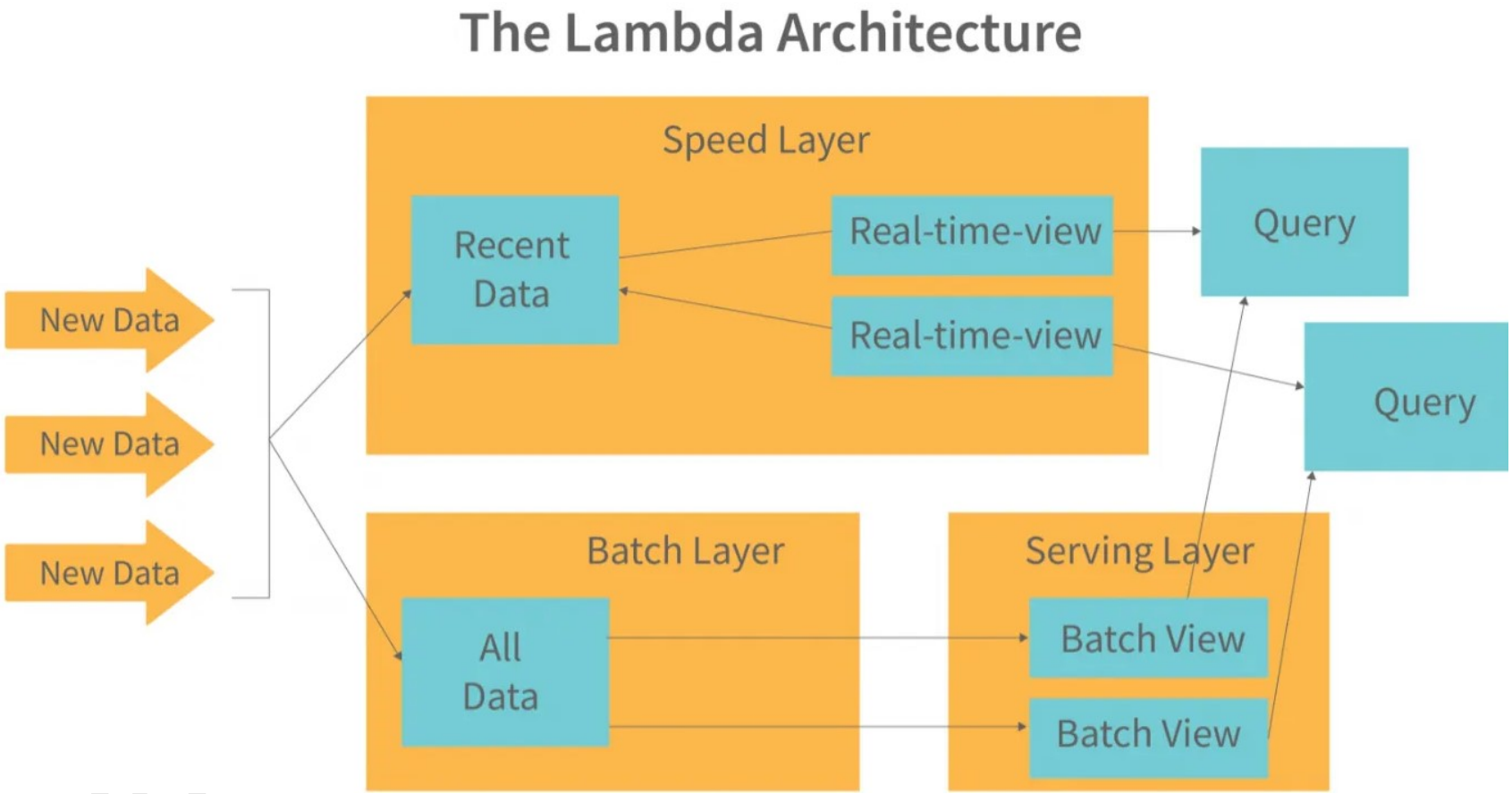


Figure: Schéma de l'architecture Lambda montrant la couche batch, la couche speed et la couche serving

3.1. Exemple - Détection de Fraude

- Détection de Fraude : Architecture Lambda utilisée pour détecter des anomalies dans les transactions financières en utilisant à la fois les données historiques et en temps réel.

- Analyse des transactions passées pour créer un modèle de référence (apprentissage) tout en surveillant les transactions en temps réel pour détecter des comportements anormaux (inférence).

- Couches Utilisées :

  - **Batch Layer** : Analyse des données historiques des transactions pour identifier des modèles de comportement normal ou suspect.

  - **Speed Layer** : Détection des anomalies dans les transactions en temps réel pour répondre immédiatement aux fraudes potentielles.

3.1. Exemple - Traitement des Données IoT

- Traitement des Données IoT : Les systèmes IoT génèrent des volumes massifs de données en temps réel. L'architecture Lambda permet de traiter ces données tout en offrant des capacités analytiques à long terme.
- Couches Utilisées :
  - **Batch Layer** : Agrégation et analyse des données historiques provenant des capteurs IoT pour identifier les tendances à long terme.
  - **Speed Layer** : Traitement des données en temps réel pour détecter des événements critiques ou des alertes (par exemple, la température d'un moteur qui dépasse un seuil critique).
- Exemple : Une usine connectée où des capteurs IoT surveillent la production en temps réel pour identifier les pannes immédiates, tout en analysant les données historiques pour optimiser l'efficacité des machines.

3.2. Limitations de l'Architecture Lambda

- **Complexité** : Maintenir deux pipelines de traitement séparés (batch et speed) double la charge de travail de développement et de maintenance.
- **Redondance** : Les mêmes données doivent être traitées deux fois (une dans la couche batch et une dans la couche speed), entraînant un effort de calcul et de stockage supplémentaire.
- **Latence de la couche batch** : Bien que les résultats de la couche speed soient rapides, les résultats finaux, les plus précis, dépendent du recalcul par la couche batch, entraînant une latence plus élevée.
- **Difficulté à unifier les vues** : Fusionner les résultats des deux couches peut être complexe, surtout lorsque les résultats de la couche speed ne correspondent pas parfaitement aux résultats de la couche batch.

3.3. Introduction à l'Architecture Lakehouse

- L'**architecture Lakehouse** est une approche unifiée qui combine les avantages des **Data Lakes** (felxibilité, variété de données et de formats) et des **Data Warehouses** (ACID, DML SQL, etc.).

- Contrairement à l'architecture Lambda, un système Lakehouse n'a pas besoin de séparer les pipelines batch et streaming, ce qui réduit la complexité.

- Il permet de stocker à la fois des données structurées et non structurées dans un seul système et prend en charge le traitement en temps réel avec les garanties ACID.

3.3. Du Datawarehouse au Lakehouse

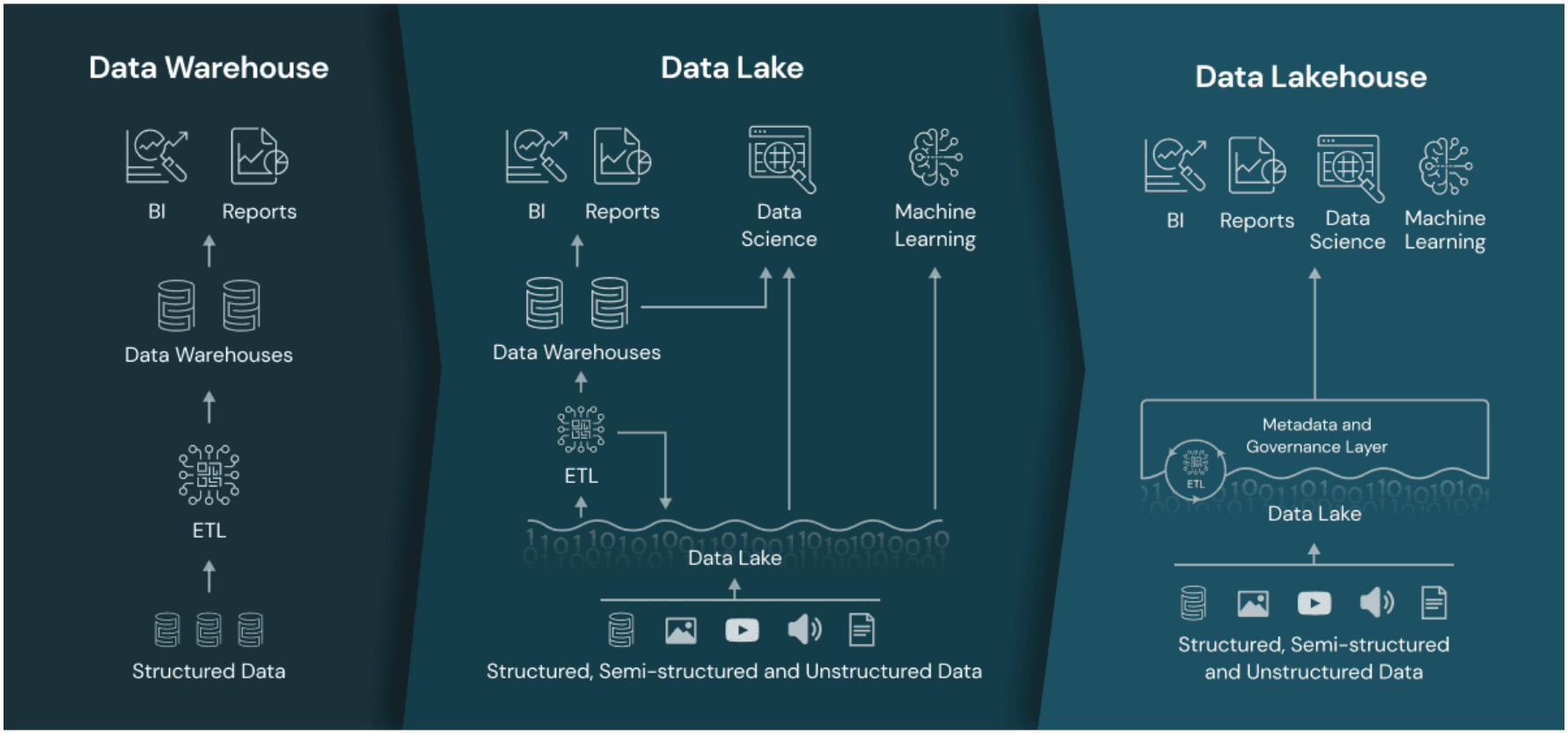


Figure reprise de https://docs.databricks.com/en/migration/warehouse-to-lakehouse.html

3.3. Avantages de l'Architecture Lakehouse

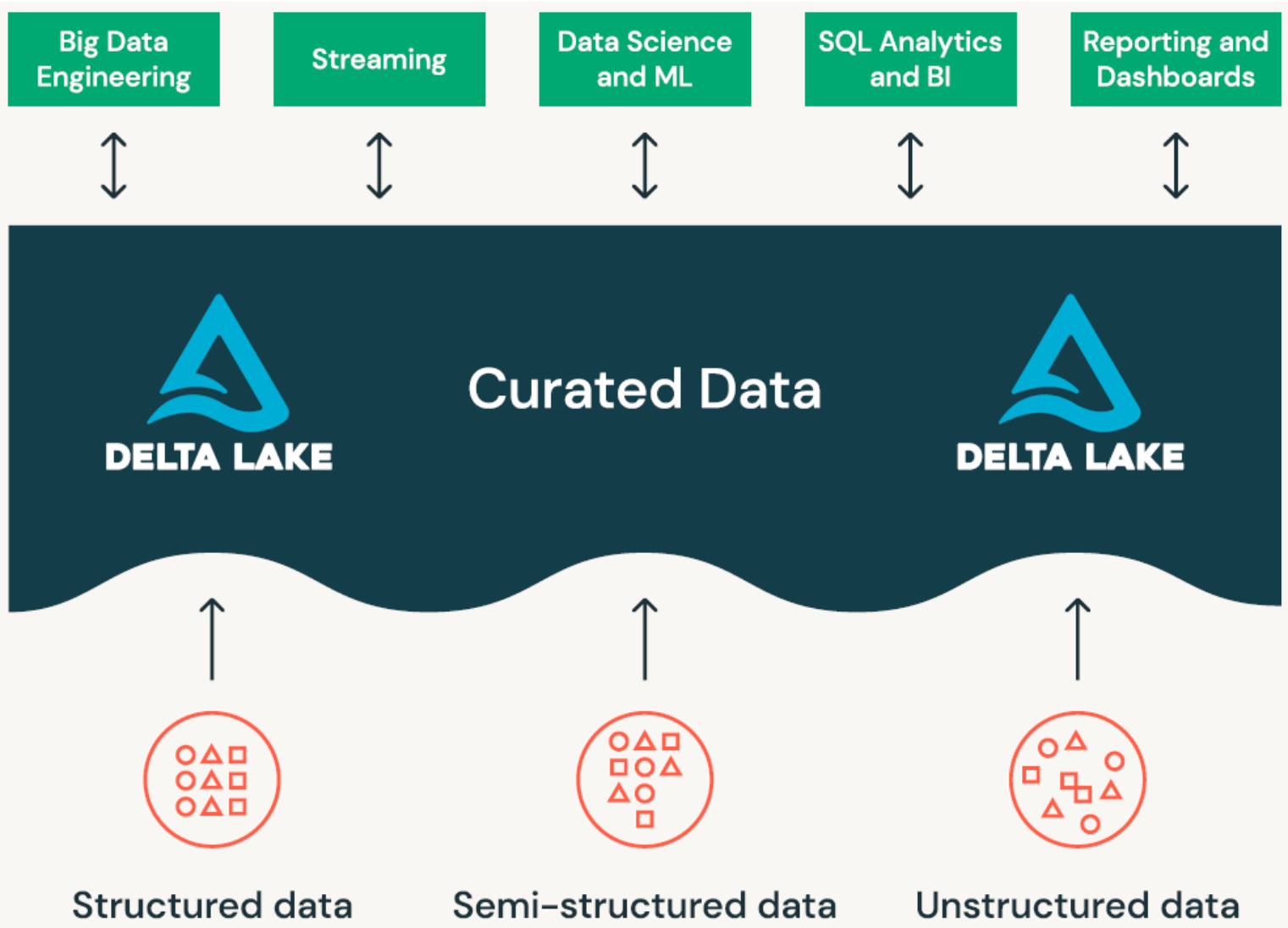
- **Simplification des pipelines** : Un seul pipeline de données est utilisé pour le traitement batch et streaming, éliminant la redondance et simplifiant la maintenance.
- **Traitement en temps réel et batch unifié** : Le traitement en temps réel et par lots peut se faire dans le même système sans double traitement.
- **Support des données non structurées** : Contrairement aux entrepôts de données traditionnels, un Lakehouse peut gérer des données non structurées comme des images, des vidéos, ou des fichiers logs.
- **Performances améliorées** : Les formats optimisés (comme Delta Lake) permettent de traiter de grandes quantités de données tout en garantissant les transactions ACID.

Section 4 - 4. Lake House et Delta Lake (Delta Table)

4.1. Delta Lake, quésaco?

- **Delta Lake** : **standard** de-facto pour l'implémentation des **lakehouses** (`delta.io`).
- Initié par Databricks, mais open-source et agnostique au format, compatible avec des moteurs d'exécution Spark, Flink, Snowflake, Google BigQuery, Redshift, etc.
- Typiquement composé d'un ensemble de Delta Tables
- **Table Delta** : Format de stockage unifié qui permet d'écrire à la fois des données en flux (streaming) et des données par lots (**Support Streaming + Batch**).
- Cette flexibilité permet de gérer à la fois les écritures incrémentielles typiques du streaming et les traitements par lots, tout en assurant des transactions ACID (sur une seule opération et une seule table).
- Delta Table est conçu pour être compatible avec des outils comme Spark Structured Streaming pour le traitement de flux, Spark SQL pour le traitement par lots et Spark ML pour le machine learning.

4.1. Delta Lake, quésaco?



(Figure reprise de la documentation officielle de Delta Lake)

4.1. Options de Stockage d'une Delta Table

- Une Delta Table est généralement stockée dans plusieurs fichiers **Parquet** (par défaut, mais pas que)
- Ces fichiers sont typiquement stockés sur un système de fichiers distribué comme :
  - **Stockage Cloud** (Google Cloud Storage, AWS S3, Azure Blob Storage).
  - **HDFS** (Hadoop Distributed File System) :
  - **DBFS** (Databricks File System) : système de fichiers natif de Databricks, optimisé pour le traitement de données à grande échelle.
  - Fichier Local pour des environnements locaux ou de tests.
- Format **Parquet** : format **orienté colonne**, ce qui permet des optimisations de requêtes et une meilleure compression.
- Delta Lake s'appuie sur Parquet et l'enrichit avec des fonctionnalités supplémentaires telles que la **gestion des versions**, les **transactions ACID** et la prise en charge de la **gestion des métadonnées**.

4.1. Avantages de l'Approche Delta Lake

- Consolidation des données : Delta Lake stocke à la fois les données en temps réel et historiques dans un même espace, simplifiant la gestion et l'analyse des données.

- Flexibilité : Les utilisateurs peuvent choisir le mode d'écriture (batch ou streaming) qui correspond le mieux aux besoins de leurs cas d'utilisation.

- Scalabilité : Conçu pour gérer de grands volumes de données et des taux d'écriture élevés, Delta Lake permet une scalabilité horizontale pour des pipelines de données robustes.

- Fiabilité : Les mécanismes de transactions ACID et de contrôle de version assurent la fiabilité des données, évitant les corruptions ou pertes lors des écritures simultanées.

4.2. Création d'une Table Delta à partir d'un DataFrame Batch

```
data = [(1,"TUN","2020-01-01",100), (2,"ITA","2020-01-02",50)]
schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("country", StringType(), True),
    StructField("date", DateType(), True),
    StructField("cases", IntegerType(), True)
])

df = spark.createDataFrame(data, schema)

# Enregistrement du DataFrame dans une table Delta
df.write.format("delta").saveAsTable("dbfs:/delta/covid_data")
```

- Les données sont enregistrées dans une table Delta via la commande `saveAsTable`, ce qui permet de bénéficier des avantages du format Delta
- Contrairement à save(), saveAsTable() permet d'utiliser directement la delta table dans les requêtes SQL.

4.2. Ajout de nouvelles Données batch

```
# Nouvelles donn es par lots
new_data = [
    (3, "ESP", "2020-01-03", 80),
    (4, "DEU", "2020-01-04", 120)
]

new_df = spark.createDataFrame(new_data, schema)

# Ajout des nouvelles donn es    la table Delta
new_df.write.format("delta").mode("append").saveAsTable("dbfs:/
    delta/covid_data")
```

- De nouvelles données sont ajoutées sous forme de batch.
- Le mode `append` permet d'insérer ces nouvelles lignes sans supprimer ni remplacer les données déjà existantes.

4.2. Ajout de Données en flux à la Table Delta

```
# On peut egalement ins rer des donn es en flux dans la m me table
streamDF = spark.readStream.format("kafka")
    .option("kafka.bootstrap.servers","..:9092")
    .option("subscribe", "covTopic").load()

# Convertir la valeur du message Kafka en type String
sDF = streamDF.selectExpr("CAST(value AS STRING)")

# Transformation des donn es (r cup ration et transtypage)
parsedDF = sDF
    .withColumn("id",split(sDF["value"],",").getItem(0).cast("int"))
    .withColumn("country",split(sDF["value"],",").getItem(1))
    .withColumn("date",split(sDF["value"],",").getItem(2).cast("date"))
    .withColumn("cases",split(sDF["value"],",").getItem(3).cast("int"))

# Ecriture des donn es de flux dans la table Delta sur DBFS
query = parsedDF.writeStream.format("delta")
    .option("checkpointLocation", "dbfs:/delta/checkpoints/covid_data").
     option("mergeSchema", "true").start("dbfs:/delta/covid_data")
```

- Le message Kafka est traité pour extraire les colonnes nécessaires.
- Les données sont ensuite insérées en continu dans la table Delta.
- **mergeSchema** : permet de gérer les **schémas évolutifs**

4.3. Opérations DML sur la Delta table et sémantique ACID

- **Mise à jour des données**

```
INSERT INTO covid_data
VALUES (1000, 'ESP', '2024-10-15', 3000)
```

- **Mise à jour des données**

```
UPDATE covid_data
SET cases = cases + 50
WHERE country = 'TUN'
```

- **Suppression des données**

```
DELETE FROM covid_data
WHERE date < '2024-01-01'
```

- Les opérations DML sur une table Delta respectent les propriétés ACID

4.3. Transactions sur les Delta Tables

- Les transactions portent sur une **seule opération** et sur **une seule table**
- Delta Lake utilise le protocole **MVCC (Multi-Version Concurrency Control)** pour gérer la concurrence d'accès.
- Le **MVCC** permet à plusieurs transactions d'accéder à une même table en même temps, en utilisant des versions différentes des données. Cela évite les blocages et améliore la concurrence.
- Chaque modification crée une nouvelle version de la table, permettant de revenir à un état antérieur sans affecter les transactions en cours.

4.4. MultiVersion Concurrency Control

Principe du protocole MVCC

- Lorsqu'un enregistrement est modifié à un instant t, une nouvelle version de l'enregistrement estampillée par t est générée sans écraser l'ancienne version
- Un même enregistrement peut ainsi avoir plusieurs versions
- Lorsque la transaction ayant généré la version la plus récente n'est pas encore validée, la version la plus récente est verrouillée et n'est pas visible par les autres transactions.
- Une transaction T_i lancé à un instant $TS(T_i)$ a un accès en lecture aux versions de enregistrements ayant la plus grande estampille temporelle inférieure ou égale à $TS(T_i) \Rightarrow T_i$ voit les données, telles qu'elles étaient à $TS(T_i)$.

4.4. MultiVersion Concurrency Control

Principe du protocole MVCC

- Les lectures ne sont jamais bloquées dans ce protocole. Une transaction voit toujours les données valides par rapport au temps de son lancement.
- L'état cohérent d'une base de données à un instant *t* est appelé **snapshot**, **état** ou **version**
- Lorsqu'une transaction se termine, elle est validée uniquement si les valeurs qu'elle a modifiées n'ont pas été entre temps modifiées par une autre transaction. Ce conflit write-write est résolu par l'annulation de la transaction.
- Ce type d'isolation est appelé **Snapshot Isolation** (appelé également Serializable). Comme le niveau d'isolation Serializable, il évite les dirty reads, les lectures non-reproductibles et les lectures fantômes.
- Contrairement à Serializable, il fait apparaître une anomalie dite *write skew*, non mentionnée dans le standard ANSI.

4.4. Exemple MVCC - Contexte Initial

Contexte : Table Delta **Version 0**, deux produits avec leurs prix et timestamps.

id	prix	timestamp
1	20	2024-11-20 09:00:00
2	30	2024-11-20 09:00:00

Table: Version initiale de la table Delta

Scénarios :

- Scénario 1 : Lecture sans conflit pendant une modification.
- Scénario 2 : Conflit détecté lors d'une modification concurrente.

4.4. Scénario 1 : Lecture sans conflit

Étape 1 : T1 modifie le produit `id = 1`

- Nouveau prix : 25.
- **Copy-On-Write :** Une nouvelle version de la ligne est ajoutée avec un timestamp différent.

Table Delta après validation (Version 1) :

id	prix	timestamp
1	20	2024-11-20 09:00:00
1	25	2024-11-20 09:05:00
2	30	2024-11-20 09:00:00

Table: Table après validation de T1

Étape 2 : T2 lit le produit `id = 1` :

- Lecture effectuée avant validation de T1.
- Résultat : **prix = 20 (Version 0)**.

4.4. Scénario 2 : Conflit détecté

Étape 1 : T1 modifie le produit `id = 1`

- Nouveau prix : 25.
- Nouvelle version ajoutée : **timestamp = 2024-11-20 09:05:00**.

Étape 2 : T2 tente de modifier le même produit (`id = 1`) :

- Nouveau prix : 22.
- Basé sur **Version 0** (`timestamp = 2024-11-20 09:00:00`).

Étape 3 : T1 valide sa modification :

id	prix	timestamp
1	20	2024-11-20 09:00:00
1	25	2024-11-20 09:05:00
2	30	2024-11-20 09:00:00

Table: Table après validation de T1 (Version 1)

Étape 4 : Conflit détecté pour T2 :

- T2 détecte que le timestamp de `id = 1` a changé (`09:05:00`).

4.4. Résolution de Conflits dans Delta Lake (1/2)

Delta Lake propose plusieurs stratégies pour gérer les conflits entre transactions concurrentes :

- **Ignorer les conflits (Last-write-wins)** :
  - La dernière transaction validée remplace les modifications précédentes.
  - **Avantages :** Simple et rapide.
  - **Limites :** Risque de perte de données critiques.

- **Fusion des modifications (Merge)** :
  - Les modifications concurrentes sont combinées selon des règles prédéfinies.

- **Rejouer des transactions échouées** :
  - T2 est réexécutée après la validation de T1, prenant en compte les nouvelles données.

- **Abandon des transactions conflictuelles** :
  - T2 est rejetée, seules les modifications de T1 sont appliquées.
  - **Avantages :** Cohérence stricte garantie.
  - **Limites :** Nécessite une intervention manuelle.

4.5. Time Travel, Politique de Rétention, et MVCC

Time Travel : Permet de revenir à une version précédente de la table pour examiner ou restaurer des données anciennes.

Politique de rétention : Détermine la durée pendant laquelle les anciennes versions des données sont conservées.

VACUUM : Utilisé pour supprimer les versions de données non référencées après la durée de rétention spécifiée.

4.5. Time Travel, Politique de Rétention

```
# Afficher l'historique des versions
spark.sql("DESCRIBE HISTORY covid_data")

# Charger une version sp cifique de la table
spark.sql("SELECT * FROM covid_data VERSION AS OF 1")

# Configurer une politique de r tention pour 30 jours
spark.sql("""
    ALTER TABLE covid_data
    SET TBLPROPERTIES ('delta.retentionDuration' = '30 days')
""")

# Nettoyer les anciennes versions qui ont plus de 30 jours
spark.sql("VACUUM covid_data RETAIN 30 DAYS")
```

4.5. MVCC, Time Travel et Politique de Rétention Avancée

Autres aspects du Time Travel

- **Restauration** : Il est possible de restaurer une version complète de la table à un moment donné.
- **Clonage** : On peut créer une nouvelle table à partir d'une version spécifique, permettant de faire des copies de sauvegarde ou des analyses sur des données historiques.

Politique de rétention

- **Granularité** : La politique de rétention peut être configurée avec une grande granularité, en spécifiant un nombre précis de versions à conserver ou une date de rétention.

4.6. Quiz

1. Expliquez comment le protocole MVCC (Multi-Version Concurrency Control) améliore la concurrence dans Delta Lake.

2. Pourquoi Delta Lake s'appuie-t-il sur le format Parquet ? Quels sont les avantages de ce format pour le stockage et l'analyse des données ?

3. Quelles sont les principales caractéristiques d'une transaction ACID dans Delta Lake ? Pourquoi ces propriétés sont-elles importantes pour la gestion des données ?

4. En quoi la commande VACUUM est-elle essentielle pour la gestion des données historiques dans Delta Lake ?

5. Expliquez la politique de rétention dans Delta Lake. Comment peut-on la configurer et pourquoi est-elle importante ?

6. Quel est le rapport entre le protocole MVCC et le Time Travel?

Section 5 - 5. Architecture en Médaillon

5.1. Vue d'ensemble de l'Architecture Médaillon

L'architecture **Médaillon** organise les données en trois couches :

- **Couche Bronze** : Données brutes, non traitées, provenant de sources en streaming ou batch.
- **Couche Silver** : Données nettoyées, dédupliquées, et normalisées.
- **Couche Gold** : Données consolidées, nettoyées et structurées selon les besoins métier.

Objectif : Rationaliser le traitement des données pour l'analyse et le machine learning.

5.1. Vue d'ensemble

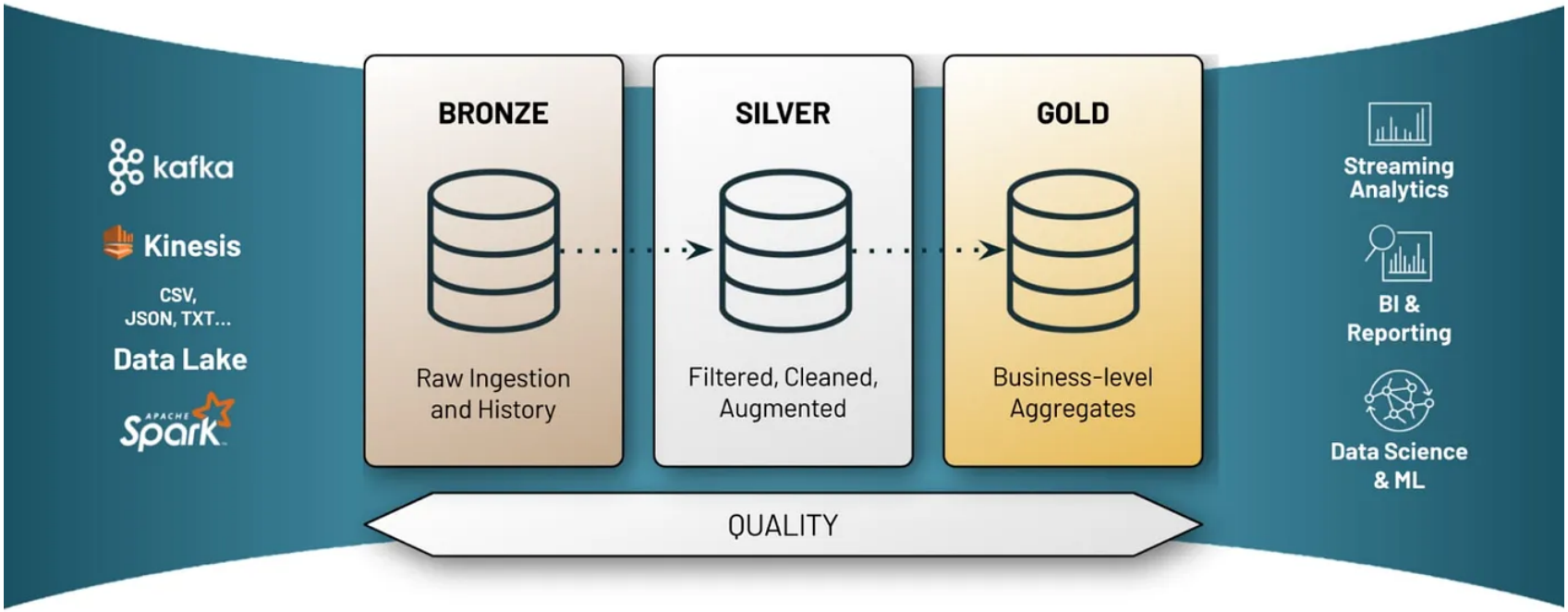


Figure reprise de https://docs.databricks.com/en/migration/warehouse-to-lakehouse.html

5.2. Couche Bronze : Données brutes

- La **Couche Bronze** stocke les données brutes non traitées.
- Données ingérées directement à partir de sources comme des fichiers JSON, des journaux, ou des données en streaming.
- **Cas d'utilisation** : Capturer tous les événements tels quels, en préservant leur forme brute pour des audits ou des analyses historiques.
- **Exemple** : Ingestion en streaming de cas COVID-19 bruts en format JSON.

5.2. Couche Silver : Données nettoyées

- La **Couche Silver** affine les données brutes :
- Les données sont **nettoyées** (ex : filtrer uniquement les cas confirmés de COVID-19) et **dédupliquées**.
- **Cas d'utilisation** : Préparer les données pour une analyse approfondie et les rendre prêtes pour des agrégations.
- **Exemple** : Filtrage des cas confirmés de COVID-19 et suppression des doublons.

5.2. Couche Gold : Données agrégées

- La **Couche Gold** contient des données prêtes à être utilisées pour les entreprises :
- Les données sont **agrégées** par des métriques comme la région, le temps ou d'autres dimensions.
- **Cas d'utilisation** : Business intelligence, rapports, ou entraînement de modèles.
- **Exemple** : Comptage agrégé des cas de COVID-19 par région, prêt pour les rapports ou l'entraînement de modèles.

5.3. Exemple d'Architecture Médaillon en PySpark

Couche Bronze - Ingestion

```
bronzeDF = spark.readStream.format("json").load("/data/bronze_raw")
bronzeDF.writeStream.format("delta").option("checkpointLocation", "/delta/checkpoints/bronze") \
    .start("/delta/tables/bronze_data")
```

Couche Silver - Données nettoyées et dédupliquées

```
silverDF = spark.read.format("delta").load("/delta/tables/bronze_data")
silverCleanedDF = silverDF.filter(col("status") == "confirmed").dropDuplicates()
silverCleanedDF.write.format("delta").mode("overwrite").save("/delta/tables/silver_data")
```

5.3. Exemple d'Architecture Médaillon en PySpark

Couche Gold - Données Agrégées

```
goldDF = spark.read.format("delta").load("/delta/tables/silver_data")
goldAggregatedDF = goldDF.groupBy("region").agg({"cases": "sum"})
goldAggregatedDF.write.format("delta").mode("overwrite").save("/delta/tables/gold_data")
```

- La **Couche Gold** agrège les données par région, fournissant des informations exploitables.

5.3. Conclusion

- L'architecture **Médaillon** avec Delta Lake organise les données en couches, permettant un traitement efficace des données et des analyses.
- Le **Machine Learning** peut être appliqué directement sur la couche Gold pour des analyses prédictives.
- Les fonctionnalités de Delta Lake (transactions ACID, enforcement de schéma, Time Travel) assurent la fiabilité des pipelines.

5.4. Récapitulatif architectures de données

Data Silos : Ensembles de données isolés, ne communiquant pas entre eux (BD, ERP isolés, etc.).

- **Problème** : Fragmentation des données, duplication, analyse globale limitée.

Data Lake : Entrepôt centralisé où les données brutes sont stockées dans leur format natif (Amazon S3, HDFS, etc.).

- **Avantages** : Économique, flexibilité pour des données hétérogènes.
- **Limites** : Complexité d'exploitation, risque de devenir un "data swamp".

5.4. Contexte des architectures de données

Architecture Lambda : Combine traitement batch et en streaming.

- **Avantages** : Fiabilité du batch + faible latence du streaming.
- **Limites** : Duplication des pipelines, coûts élevés.

Lake House : Combine les avantages des Data Lakes et des entrepôts de données.

- **Avantages** : Unification des pipelines, traitement simplifié.
- **Limites** : Dépendance à des outils spécifiques, complexité initiale.
- **Exemple** : Delta Lake (ACID), Snowflake, Google BigQuery.

Comparaison des Concepts d'Architectures de Données

Concept	**Stockage**	**Traitement**	**Modèle de données**	**Intégration**	**Outils typiques**	**Cas d'utilisation**	**Langages typiques**
Data Silos	Fragmenté	Isolé	Relationnel	ETL	BDs relationnelles, ERP	Applications métier spécifiques	SQL
Data Warehouse	Centralisé	Structuré	Dimensionnel (*e.g.* étoile, flocon)	ETL (*e.g.* Talend)	Snowflake, BigQuery	Analyses décisionnelles, reporting	SQL, PL/SQL
Data Lake	Centralisé	Non structuré	Schema-on-read	ELT	S3, HDFS	Stockage de données brutes, IA, ML	Python, Scala, R
Lake House	Centralisé	Batch + streaming	Hybride (structuré, semi-structuré et non-structuré)	ELT, streaming	Delta Lake, Iceberg	Unification des données et des analyses, ML	SQL, Spark SQL, Spark ML

Quiz

1. Quel est l'objectif principal de l'architecture Médaillon ? Expliquez en quoi cette organisation est bénéfique pour l'analyse des données.
2. Quels sont les rôles respectifs des couches Bronze, Silver et Gold dans l'architecture Médaillon ?
3. Quelles transformations sont typiquement effectuées dans la couche Silver, et pourquoi sont-elles nécessaires ?
4. Comment les données de la couche Gold se distinguent-elles de celles des couches Bronze et Silver ?
5. Donnez un exemple concret d'un pipeline qui utilise les trois couches de l'architecture Médaillon.
6. Comment l'architecture Médaillon contribue-t-elle à améliorer la traçabilité des données et à diagnostiquer les erreurs dans les pipelines de données ?
7. Quels sont les défis potentiels liés à la mise en place et à la maintenance de l'architecture Médaillon ?