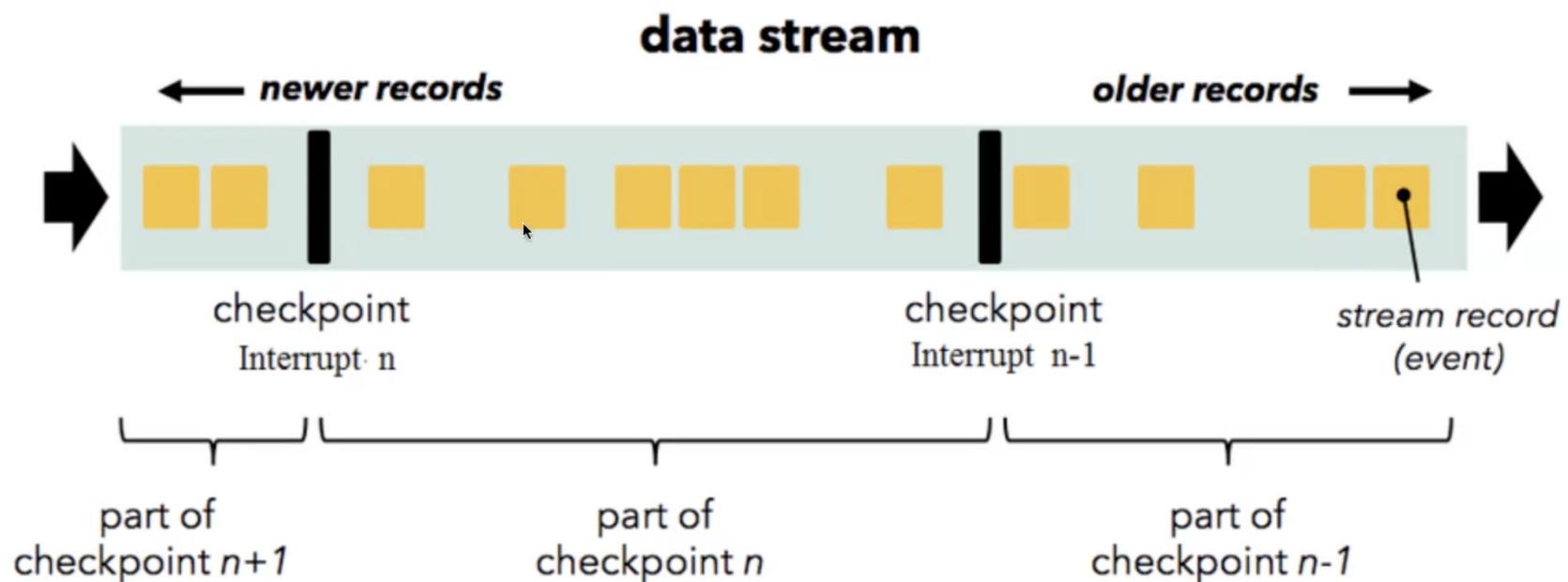


DE



1. Основные признаки того, что данные "большие"
2. Scala
 - a. Принципы работы, откуда пошёл
 - b. Хвостовая рекурсия
 - c. Pattern matching
3. Map Reduce
4. Apache Spark
 - a. Основные принципы работы
 - b. Отношение master-worker, YARN
 - c. Работа с RDD
 - d. Работа с Spark DataFrame
 - e. Основные функции DataFrame
 - f. Основные функции DataFrame
 - g. Spark Streaming
 - Общий пайплайн
 - h. MLlib
 - Коллаборативная фильтрация
 - i. GraphLib
 - j. RDD vs DataFrame
 - k. Transformer
 - l. Hashing trick
5. Apache Kafka
 - a. Основные понятия
 - b. Репликация данных
 - c. Принцип работы consumer group
 - d. Гарантия доставки данных
 - e. Kafka streaming – кейсы применения, внутреннее устройство
6. Apache Flink
 - a. Принципы работы
 - b. Устройство пайплайн
 - c. Flink vs Spark streaming
 - d. Checkpointing
 - Watermark
 - e. Кейсы применения
7. Принципы работы с streaming data
 - a. Разные семантики доставки данных
 - Семантика at least once («хотя бы один раз»).
 - Семантика at most once («не более одного раза»).
 - Семантика exactly once («строго однократная доставка»).
 - b. DataFlow model
- 8.* Вероятностные структуры данных
 - a. Фильтр Блума
 - b. Hyper log log
 - c. T-Digest

1. Основные признаки того, что данные "большие"

3V: VVV - Volume, Velocity, Variety

Из-за разных форматов и путей возникновения Big Data отличаются рядом характеристик:

1. Volume. (Большие про то что много и нужно хранить в большом объеме) Огромные «объемы» данных, которые организации получают из бизнес-транзакций, интеллектуальных (IoT) устройств, промышленного оборудования, социальных сетей и других источников, нужно где-то хранить. В прошлом это было проблемой, но развитие систем хранения информации облегчило ситуацию и сделало информацию доступнее.

2. Velocity. (про скорость генерации данных, не можем за длинное время писать много данных) Чаще всего этот пункт относится к скорости прироста, с которой данные поступают в реальном времени. В более широком понимании характеристика объясняет необходимость высокоскоростной обработки из-за темпов изменения и всплесков активности.

(где-то долго пишутся данные, много event-loop-ов для пользователя и т.д.)

3. Variety. Разнообразие больших данных проявляется в их форматах: структурированные цифры из клиентских баз, неструктурированные текстовые, видео- и аудиофайлы, а также полустроктурированная информация из нескольких источников. Если раньше данные можно было собирать только из электронных таблиц, то сегодня данные поступают в разном виде: от электронных писем до голосовых сообщений

4*. veracity — достоверность, использовалась в рекламных материалах IBM[28]),

Кейсы:

Датчики IoT, действия клиентов приложений, транзакции банков, трафик города и т.п.

ETL = Extract/Transfrom/Load процесс преобразования данных из Data Warehouse(хранилище).

Data Warehouse - больше про структурированные данные, их достают легко.

Data Lake - данные сырье, для каждого запроса долго процессинг идет.

Разница между warehouse и lake

	Data warehouse	Data lake
Хранение	Трансформированные данные	Сыре данные
Вид данных	Structured	Structured, semi-structured, unstructured
Схема данных	Определяется до сохранения	Определяется после сохранения
Процессинг	ETL	ELT
Стоимость хранения	Растёт быстро с увеличением объема	Низкая стоимость
Основные пользователи	Business Analytics	Data Scientists/Data Engineers

2. Scala

<https://docs.scala-lang.org/overviews/>

a. Принципы работы, откуда пошёл

Надстройка над java, работает на JVM, scalac компилирует Scala → Java bytecode

Включает functional и imperative парадигмы, ООП. Типизированный язык.

val - immutable, var - mutable.

Int, Double, String, Char, BigInt и др. типы.

Functional programming

- Functions are first class citizens
- Immutability
- Tuples
- Currying
- Recursion
- Monads

RDD operations

- **groupByKey** – по RDD (K, V) возвращает (K, Iterable<V>)
- **reduceByKey** – агрегация значения
- **aggregateByKey** – тоже агрегация, но позволяет выдавать другой тип данных
- **join** – джойнит данные по ключу. (K,V).join(K,W) = (K, (V,W))
- **cartesian** – декартово произведение
- **union** – объединение с другим RDD
- **intersection** – пересечение с другим RDD
- **distinct** – уникальные значения
- **coalesce** – уменьшить количество партиций в данных
- **repartition** – выставить количество партиций

Pattern matching

```
def flatten(list: List[Any]): List[Any] =  
  list match {  
    case (x: List[Any]) :: xs =>  
      flatten(x) :: flatten(xs)  
    case x :: xs => x :: flatten(xs)  
    case Nil => Nil  
  }  
  
val nested = List(1, List(2, 3), 4);  
val flat = flatten(nested); // List(1, 2, 3, 4)
```

Static typing

- Type checking done at compile time
- Type associated with variable, not value
- Better tools possible
- More verbose code compared to dynamic language
- Can't add methods to class at runtime
- No duck typing – really?

b. Хвостовая рекурсия

Любой рекурсивный вызов является последней операцией перед возвратом из функции.. Оптимизация хвостовой рекурсии путём преобразования её в плоскую итерацию реализована во многих оптимизирующих компиляторах. В некоторых функциональных языках программирования спецификация гарантирует обязательную оптимизацию хвостовой рекурсии.

c. Pattern matching

Что это такое и кейсы применения

Работает как switch, чуть больше функционала.

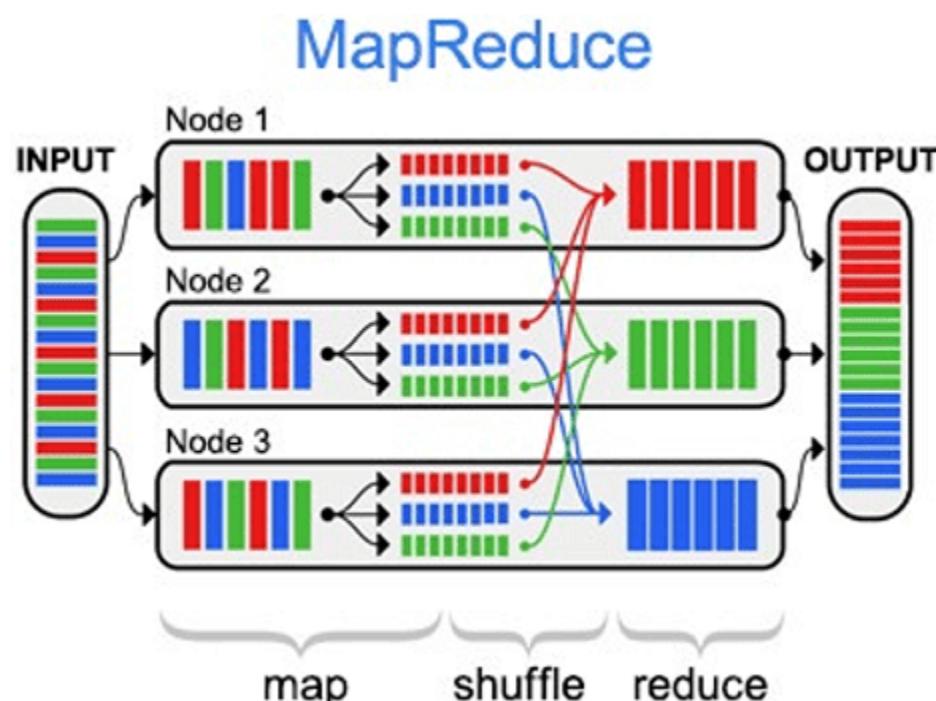
```
val x: Int = Random.nextInt(10)

x match {
  case 0 => "zero"
  case 1 => "one"
  case 2 => "two"
  case _ => "other"
}
```

3. Map Reduce

Водель распределённых вычислений используемая для параллельных вычислений над очень большими, вплоть до нескольких петабайт, наборами данных в компьютерных кластерах.

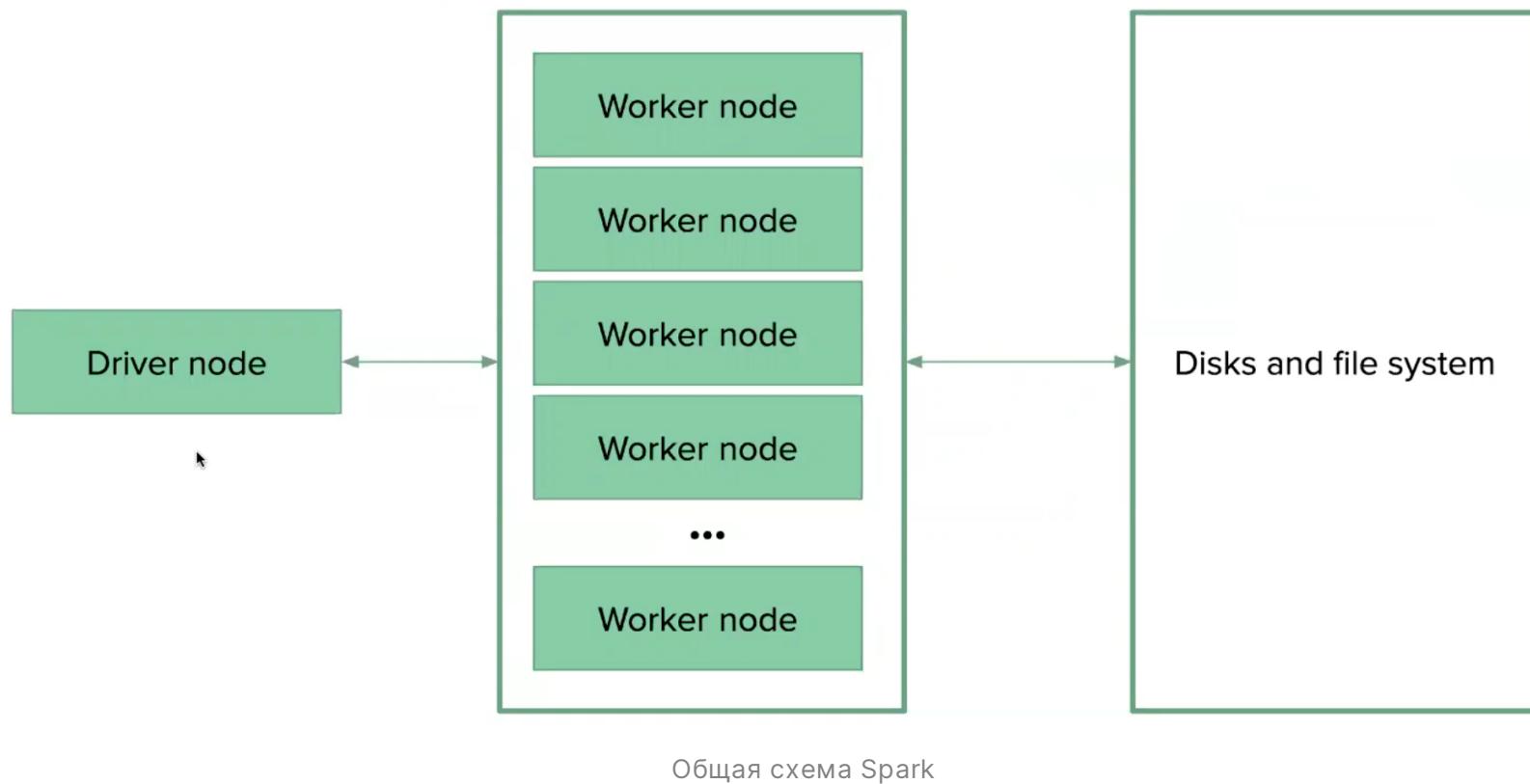
1. **Map** – предварительная обработка входных данных в виде большого списка значений. При этом главный узел кластера (master node) получает этот список, делит его на части и передает рабочим узлам (worker node). Далее каждый рабочий узел применяет функцию Map к локальным данным и записывает результат в формате «ключ-значение» во временное хранилище.
2. **Shuffle** - когда рабочие узлы перераспределяют данные на основе ключей, ранее созданных функцией Map, таким образом, чтобы все данные одного ключа лежали на одном рабочем узле.
3. **Reduce** – параллельная обработка каждым рабочим узлом каждой группы данных по порядку следования ключей и «склейка» результатов на master node. Главный узел получает промежуточные ответы от рабочих узлов и передаёт их на свободные узлы для выполнения следующего шага. Получившийся после прохождения всех необходимых шагов результат – это и есть решение исходной задачи.



4. Apache Spark

<https://medium.com/better-programming/high-level-overview-of-apache-spark-c225a0a162e9>

Apache Spark - фреймворк с открытым исходным кодом для реализации распределённой обработки неструктурированных и слабоструктурированных данных, входящий в экосистему проектов Hadoop.



a. Основные принципы работы

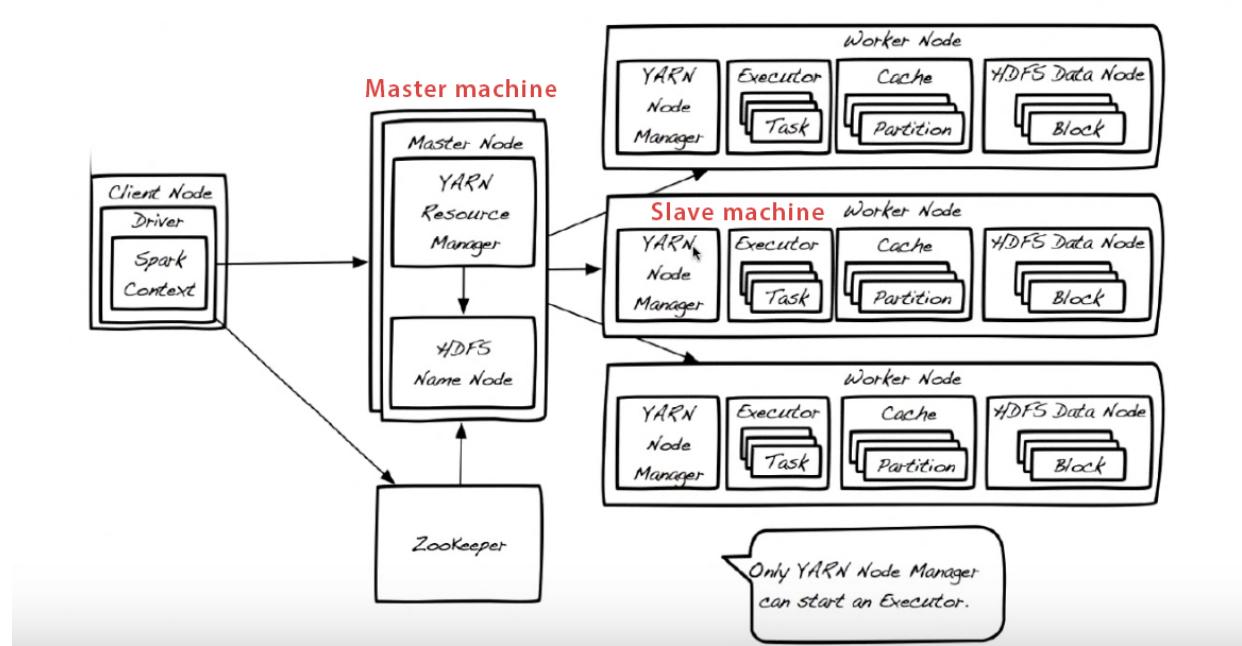
Apache Spark™ is a unified analytics engine for large-scale data processing

- Двигок для процессинга данных
- API для Python, Scala, Java, R
- Запросы на SQL
- Просто разворачивается на разных кластерах: свой кластер, EC2(сервера Amazon), Hadoop YARN, etc
- Работает с разными источниками данных: HDFS, Cassandra, Hive, etc
- Широкое коммьюнити даёт возможность для stackoverflow based engineering
- Одна из колон в BigData инфраструктуре

b. Отношение master-worker, YARN

По сути это внедрение архитектуры **Apache yarn hadoop** в **spark**. Плаиорвщик который обеспечивает справедливость раздачи ресурсов

YARN-based Architecture



YARN – Yet Another Resource Negotiator

- Отделяет логику планировщика от Spark, FAIR вместо FIFO, то есть планировкой yarn займется, а не spark scheduler

c. Работа с RDD

1. Распределенная **immutable** коллекция данных (примитив для работы с произвольными данными)
2. любой распределенный источник
3. in-memory(на node-ах то есть), но при нехватке spill на диск
4. Lazy evaluation - пока не произойдет действие, не будет вычисляться ничего (после трансформаций)
5. map, filter, flatmap, sample - основные операции

Основные операции:

- **map** – преобразует значение с помощью переданной функции
 - lines.map(s ⇒ (s,1))
- **filter** – фильтрует значения по переданной функции
 - lines.filter(s ⇒ (s % 3 == 0))
- **flatMap** – как map, но делает “плоский” результат
 - lines.flatMap(s ⇒ (s,1))
- **sample** – получить небольшой сэмпл данных
 - lines.sample(10)
- **groupByKey** – по RDD (K, V) возвращает (K, Iterable<V>)
- **reduceByKey** – агрегация значения
- **aggregateByKey** – тоже агрегация, но позволяет выдавать другой тип данных
- **join** – джойнит данные по ключу. (K,V).join(K,W) = (K, (V,W))
- **cartesian** – декартово произведение
- **union** – объединение с другим RDD
- **intersection** – пересечение с другим RDD
- **distinct** – уникальные значения
- **coalesce** – уменьшить количество партиций в данных
- **repartition** – выставить количество партиций

d. Работа с Spark DataFrame

Через модуль SparkSQL: join, select, groupBy(), agg и др.

Датафрейм - распределенная коллекция данных

- Позволяет переписать запросы в функциональном стиле
- Сильная типизация – ловим ошибки на этапе компиляции (но не все)
- Все оптимизации от Spark SQL используются и здесь
- DataFrame = Dataset[Row] – удобный alias, работать будем с DataFrame

Различия RDD и DataFrame

Aa Basis of Difference	≡ Spark RDD	≡ Spark DataFrame
<u>What is it?</u>	Low-level API	High-level abstraction
<u>Execution</u>	Lazy evaluation	Lazy evaluation

Aa Basis of Difference	Spark RDD	Spark DataFrame
Data types	unstructured	Both unstructured and structured
Benefit	Simple API	Gives schema to distributed data
Limitation	Limited performance	Possibility of failure during the run time

f. Основные функции DataFrame

show, printschema, filter, select, groupby agg

g. Spark Streaming

Не классический(трушный) stream, весь вход разбирается в batches и прогоняется через spark engine.

DStream - батч RDD из данных

Здесь понятие *windows duration* (чз сколько секунд закончится окно)

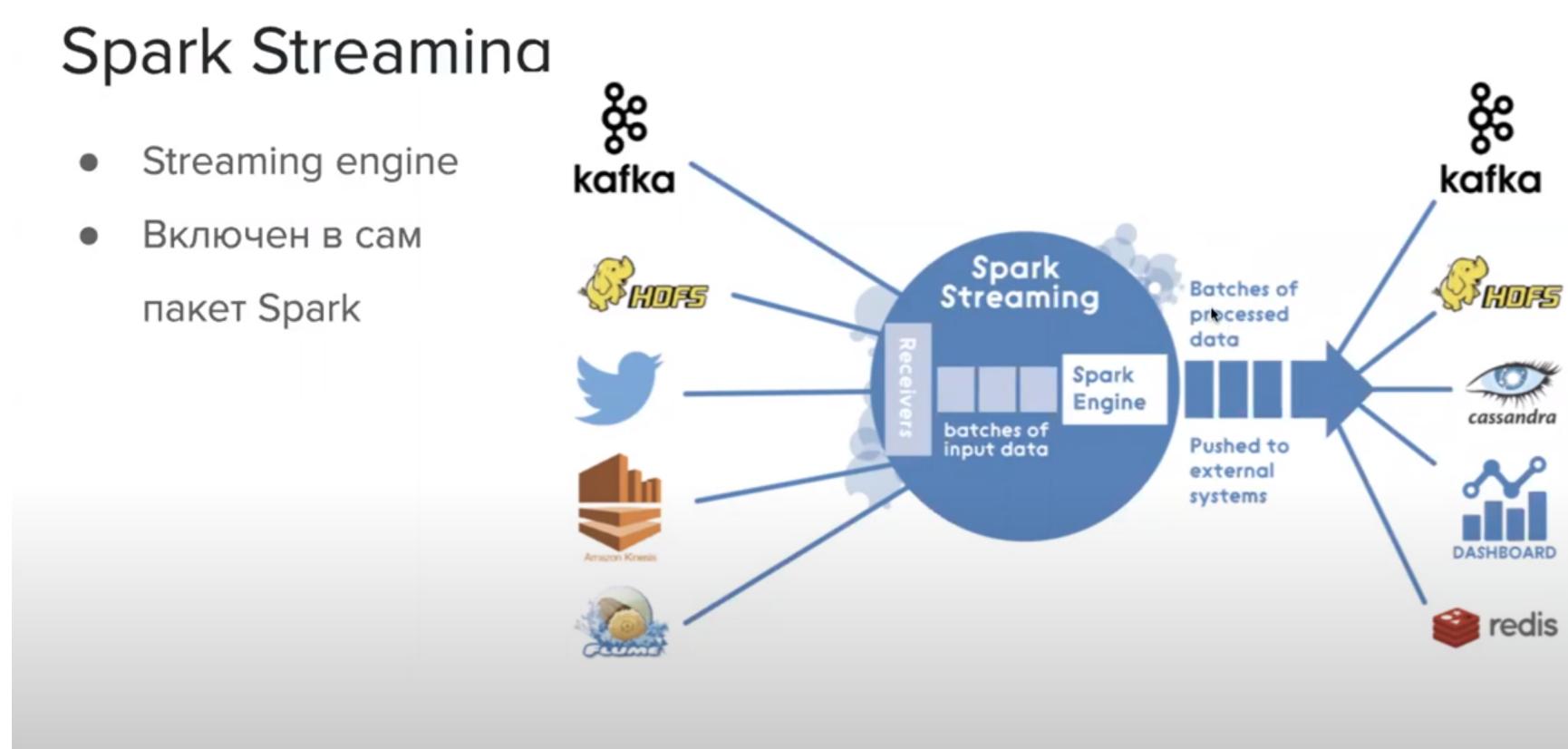
slide duration - как часто задаётся окно



Общий пайплайн

Pipeline обработки данные спарк стримингом

1. Создать источники данных через DStream
2. Определить операции над стримом, которые принимают и выдают DStream
3. Начать получение данных с помощью streamingContext.start().
4. Дождаться окончания процессинга (ручная остановка или ошибка в программе) с помощью streamingContext.awaitTermination().
5. Можно вручную остановить с помощью streamingContext.stop().



h. MLlib

i. Основной функционал

Либы внутри spark для машинного обучения

ii. Как реализовано обучение колаборативной фильтрации

Колаборативная фильтрация

Учитывая, что самый известный пример применения Больших Данных — это рекомендательная система, было бы странным, если бы самые простейшие алгоритмы не были реализованы во многих пакетах. Это касается и Spark'a. В нем реализован алгоритм

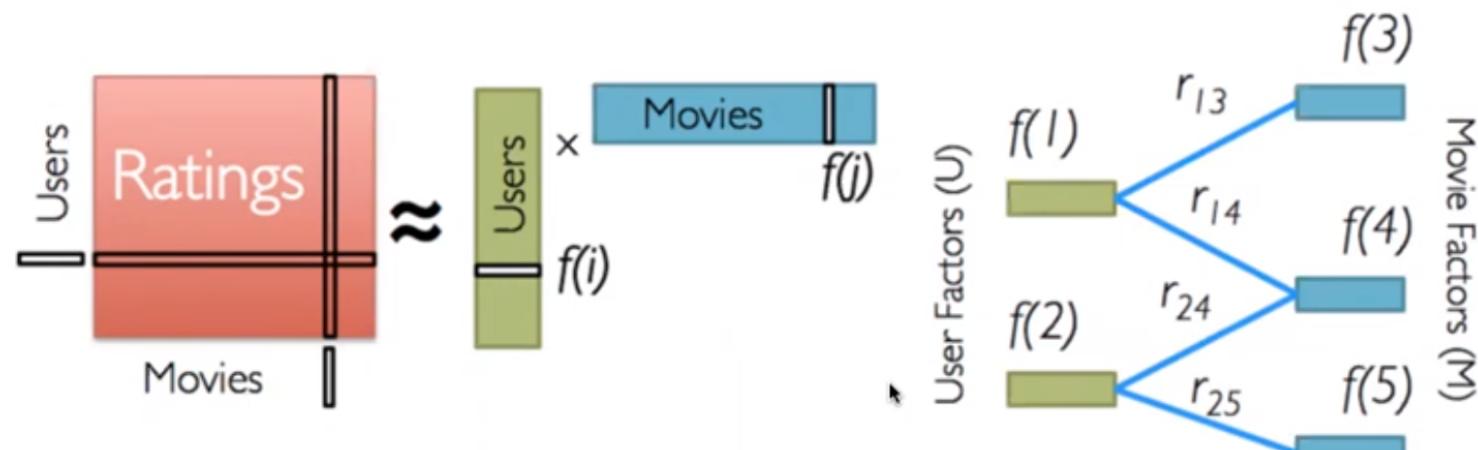
ALS (Alternative Least Square) — пожалуй, один из самых известных алгоритмов колаборативной фильтрации. Описание самого алгоритма заслуживает отдельной статьи. Здесь только скажем в двух словах, что алгоритм фактически занимается разложением матрицы отзывов (строки которой — это пользователи, а столбцы — продукты) — на матрицы **продукт — топик и топик-пользователь**, где топики — это некоторые скрытые переменные, смысл которых зачастую не понятен (вся прелесть алгоритма **ALS** как раз в том, чтобы сами топики и их значения найти). Суть этих топиков в том, что каждый пользователь и каждый фильм теперь характеризуются набором признаков, а скалярное произведение этих векторов — это и есть оценка фильма конкретного пользователя. Обучающая выборка для этого алгоритма задается в виде таблицы

userID → productID → rating

После чего делается обучение модели с помощью ALS :

```
from pyspark.mllib.recommendation import ALS
model = ALS.train (ratings, 20, 60)
predictions = model.predictAll(ratings.map (lambda x: (x[0],x[1])))
```

Alternating least squares (ALS)



iii. Линейная регрессия

iv. RDD vs DataFrame

i. GraphLib

Либа про графы в спарке

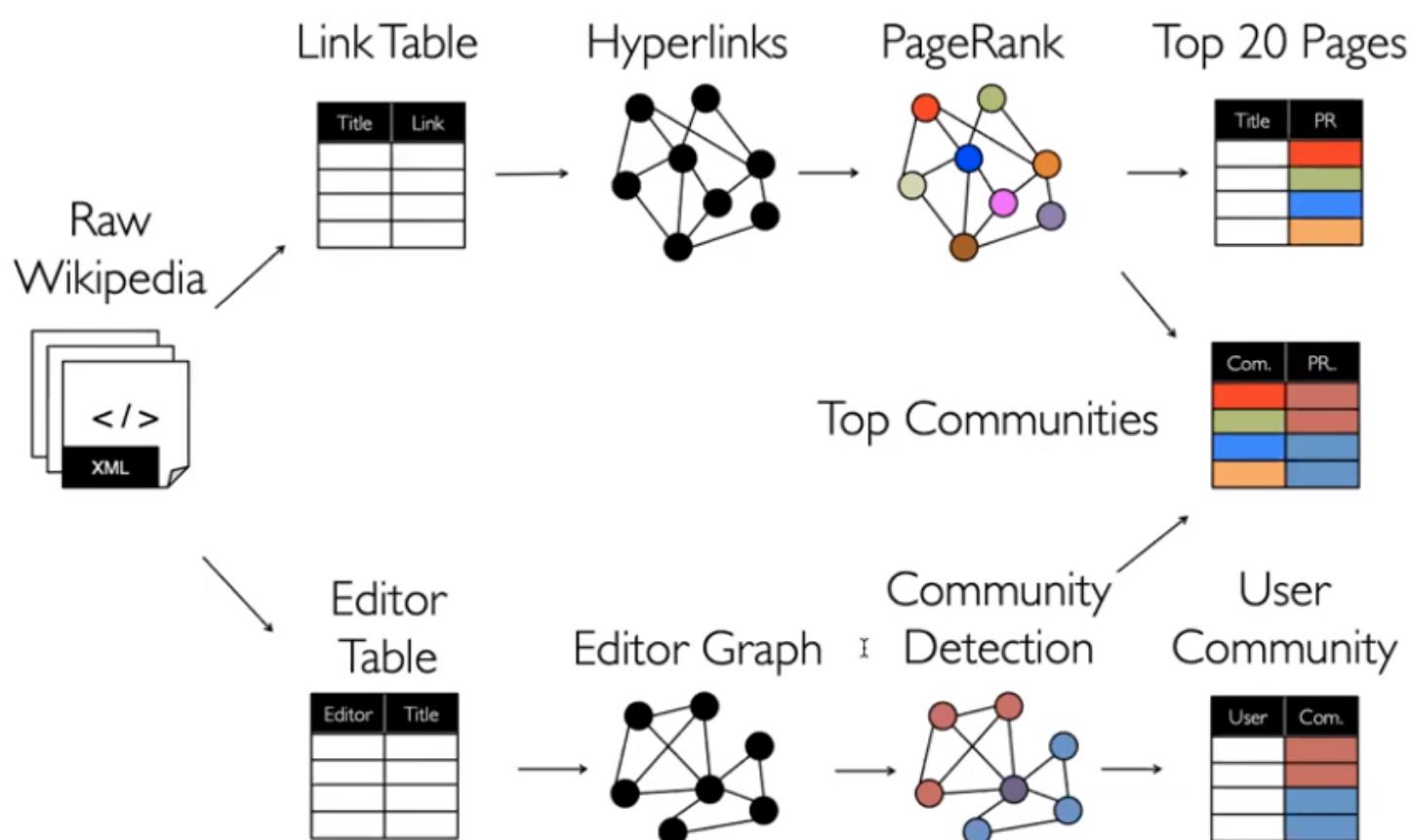
Complex Pipelines

Solution: Embed graph processing within a table-oriented system (Spark)

Challenges:

1. Storage: How to store graphs as tables?
2. Computation: How to express graph ops as table ops (map, reduce, join, etc.)?
3. API: How to present the two views to the user?

Complex Pipelines



Graph Operations (Scala)

```
class Graph[VD, ED] {  
    // Table Views -----  
    def vertices: RDD[(VertexId, VD)]  
    def edges: RDD[Edge[ED]]  
    def triplets: RDD[EdgeTriplet[VD, ED]]  
    // Transformations -----  
    def mapVertices[VD2](f: (VertexId, VD) => VD2): Graph[VD2, ED]  
    def mapEdges[ED2](f: Edge[ED] => ED2): Graph[VD2, ED]  
    def reverse: Graph[VD, ED]  
    def subgraph(epred: EdgeTriplet[VD, ED] => Boolean,  
                vpred: (VertexId, VD) => Boolean): Graph[VD, ED]  
    // Joins -----  
    def outerJoinVertices[U, VD2](  
        tbl: RDD[(VertexId, U)])  
        (f: (VertexId, VD, Option[U]) => VD2): Graph[VD2, ED]  
    // Computation -----  
    def aggregateMessages[A](  
        sendMsg: EdgeContext[VD, ED, A] => Unit,  
        mergeMsg: (A, A) => A): RDD[(VertexId, A)]
```

i. RDD vs DataFrame

сравнение Spark RDD со Spark df

Basis of Difference	Spark RDD	Spark DataFrame
What is it?	Low-level API	High-level abstraction
Execution	Lazy evaluation	Lazy evaluation
Data types	unstructured	Both unstructured and structured
Benefit	Simple API	Gives schema to distributed data
Limitation	Limited performance	Possibility of failure during the run time

проще API

Сравнение Spark RDD со Spark df

j. Transformer

Про трансформацию слов в вектора

- **Tokinizer** - токианайзер
- **CountVectorizer**
- **HashingTf**
 - не имеет обратной операции
 - работает за $T = O(n)$ и $M = O(1)$
 - depends on a size of the vector

HashingTF Transformer

```
val tokenizer = new Tokenizer().setInputCol("text").setOutputCol("terms")
val hashingTf = new HashingTF().setInputCol("terms").setOutputCol("features")
val pipeline = new Pipeline().setStages(Array(tokenizer, hashingTf))
val model = pipeline.fit(df)
```

text	terms	features
The quick brown fox	[the, quick, brown, fox]	(262144, [22323, 38208, 103838, 129637], [1.0, 1.0, 1.0, 1.0])
jumps over	[[jumps, over]]	(262144, [179832, 252565], [1.0, 1.0])
the lazy dog	[[the, lazy, dog]]	(262144, [51504, 75919, 103838], [1.0, 1.0, 1.0])



#EUds15

13

k. Hashing trick

Проблема в том, что иногда BOW или One-Hot не работают. Например sparse данные, много пустых ячеек.

The “Hashing Trick”

- Use a *hash function* to map feature values to indices in the feature vector

Dublin → Hash "city=dublin" → Modulo hash value to vector size to get index of feature → [0 0 1 0 0]

Stock Price → Hash "stock_price" → Modulo hash value to vector size to get index of feature → [0 2.60 ... 0 0]

SPARK SUMMIT EUROPE 2017 #EUds15

Feature Hashing: Pros

- Fast & Simple
- Preserves sparsity
- Memory efficient
 - Limits feature vector size
 - No need to store mapping feature name -> index
- Online learning
- Easy handling of missing data
- Feature engineering

-s : collisions and no inverse ⇒ no idea about features impact

5. Apache Kafka

<https://kafka.apache.org/intro>

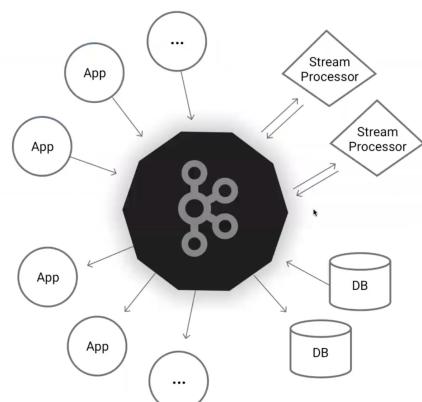
Технология-сервис которая используется для обработки сообщение(message broker). Основное назначение: приём\передача сообщений.

Применяется в сложной системой в которой присутствует много разных независимых частей и мы хотим обеспечить их взаимодействие. Много сервисов, хотим их связать.

Streaming semantics: (про стратегию репликации)

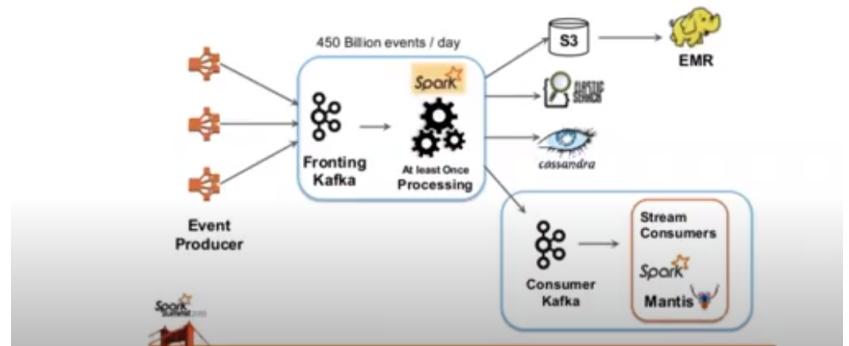
- At most once
- At least once
- Exactly once - строго однократная доставка

Популярная связка: Apache Kafka + Streaming service



Общий пример применения Kafka

Netflix example



Пример Kafka in Netflix

a. Основные понятия

Сущности из которых состоит кафка : Consumer, producer, broker, topic, offset, partition, consumer group, leader/follower replica, cluster

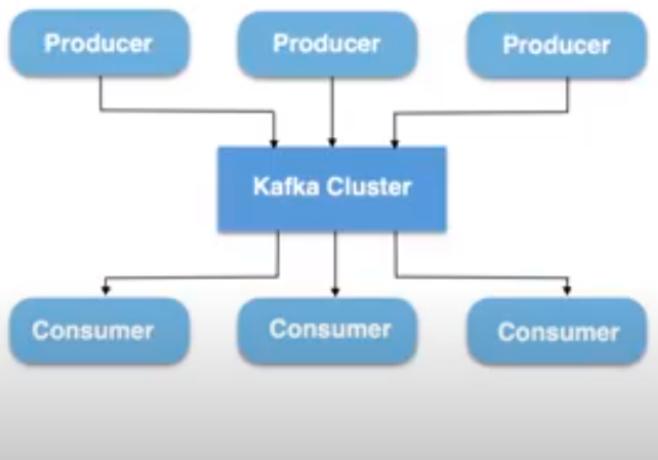
Topic

- Конкретный стрим данных
- Producer пишет данные в конкретный топик
- Consumer читает из конкретного топика
- Партицирован
- Имеет некий offset

topic: воспринимать как табличку в БД.

Kafka Broker

- Собственно один сервер Kafka
- Дает возможность пользователям писать и читать сообщения, не взаимодействуя между собой
- Развернут на кластере – может быть как один, так и несколько



broker: 1 сервер кафки через который можно подключаться с помощью consumer или producer. Их может быть много

Offset

Immutable номер записи внутри конкретной партиции

То есть, чтобы найти конкретное сообщение, необходимы:

- Topic name
- Partition number
- Offset

offset - используется для задачи указателя на сообщение

Consumer

- Любое приложение, которое читает сообщения из Kafka
- Читает с сервера Kafka, никак не взаимодействуя с producer'ами

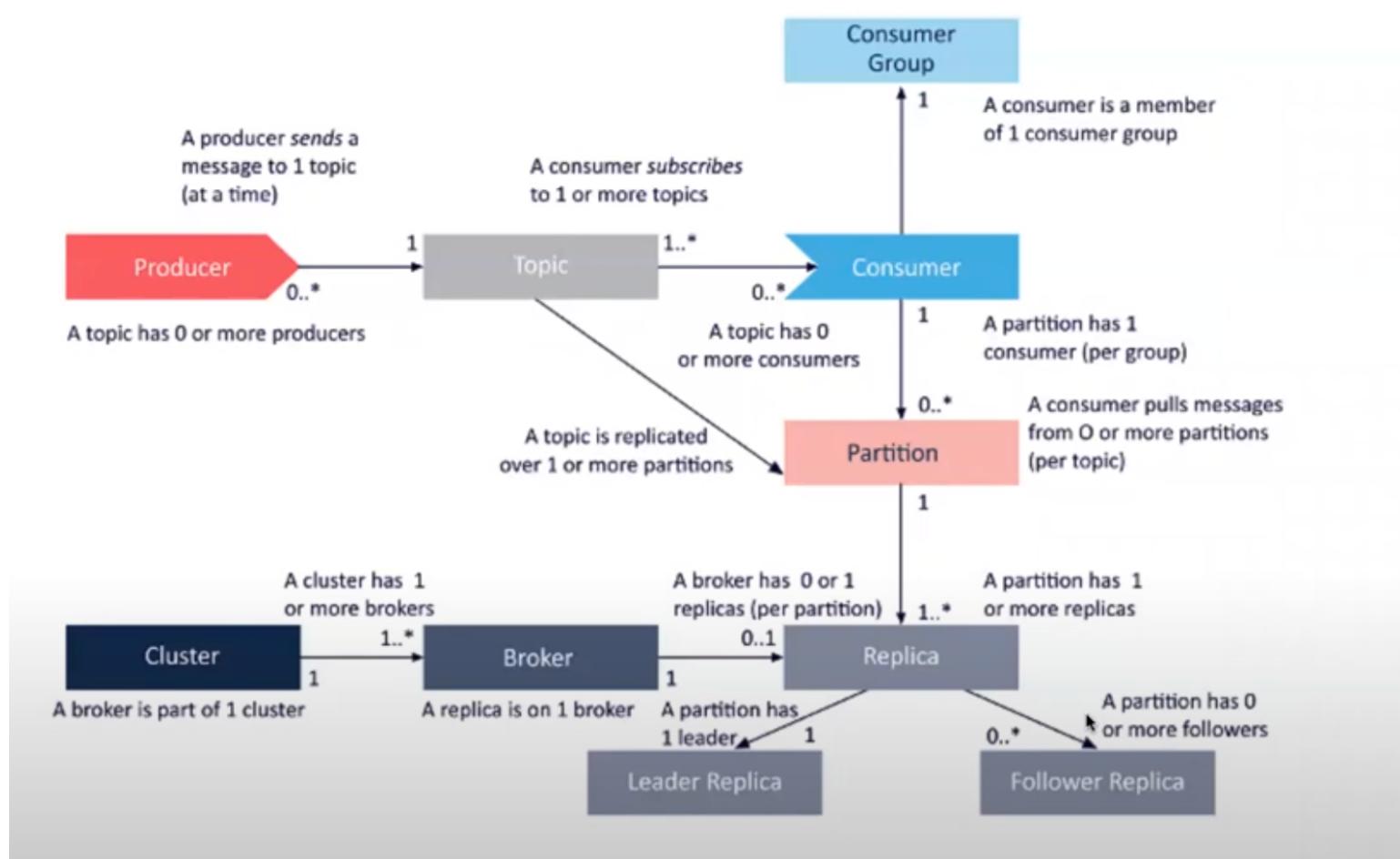
Consumer

Producer

- Любое приложение, которое шлёт сообщения
- Сообщение - достаточно маленький кусок информации
- Пишет на сервер Kafka, а не конкретному конечному пользователю
- Примеры:
 - Web Application
 - Spark Connector – пишем сообщения прямо из DataFrame'a Spark
 - Flink – об этом позже
 - Etc – все, под что написан коннектор

Producer

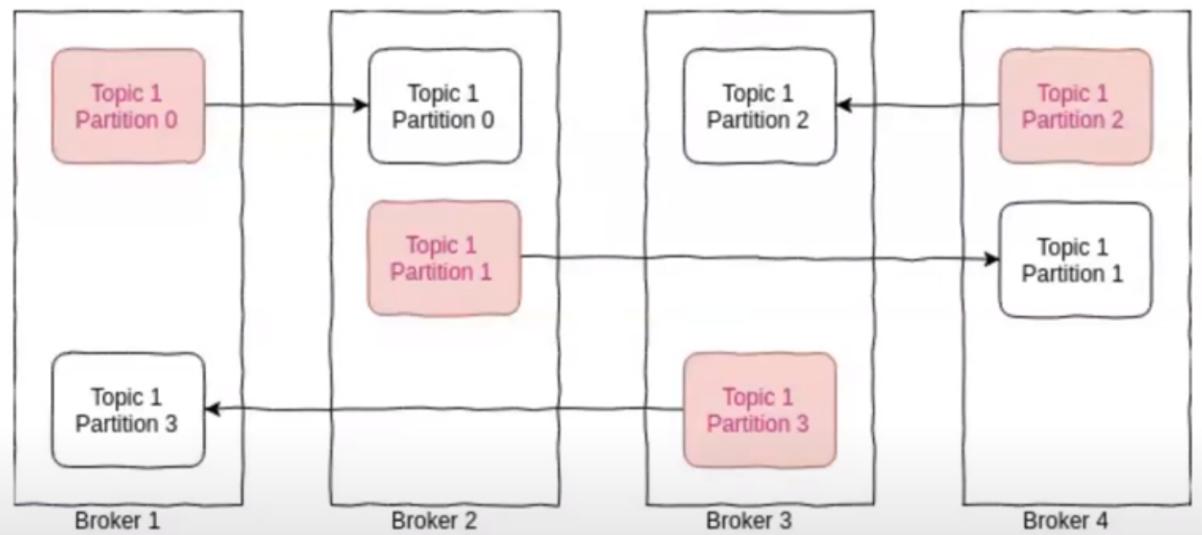
topic + partition + offset - задают систему координат



b. Репликация данных

Репликация данных

Пишем данные второй раз чтобы они оставались доступными при потере одного брокера



Механизм для того, чтобы обеспечить не потерю данных. Чтобы при потере брокера все было ок

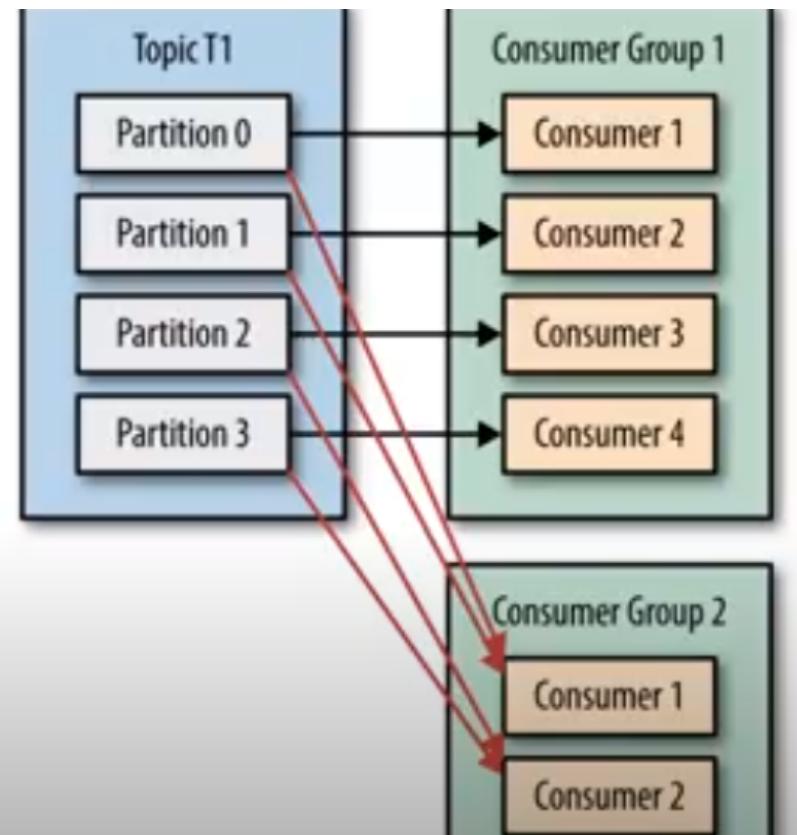
c. Принцип работы consumer group

У партиции ровно один консьюмер внутри группы

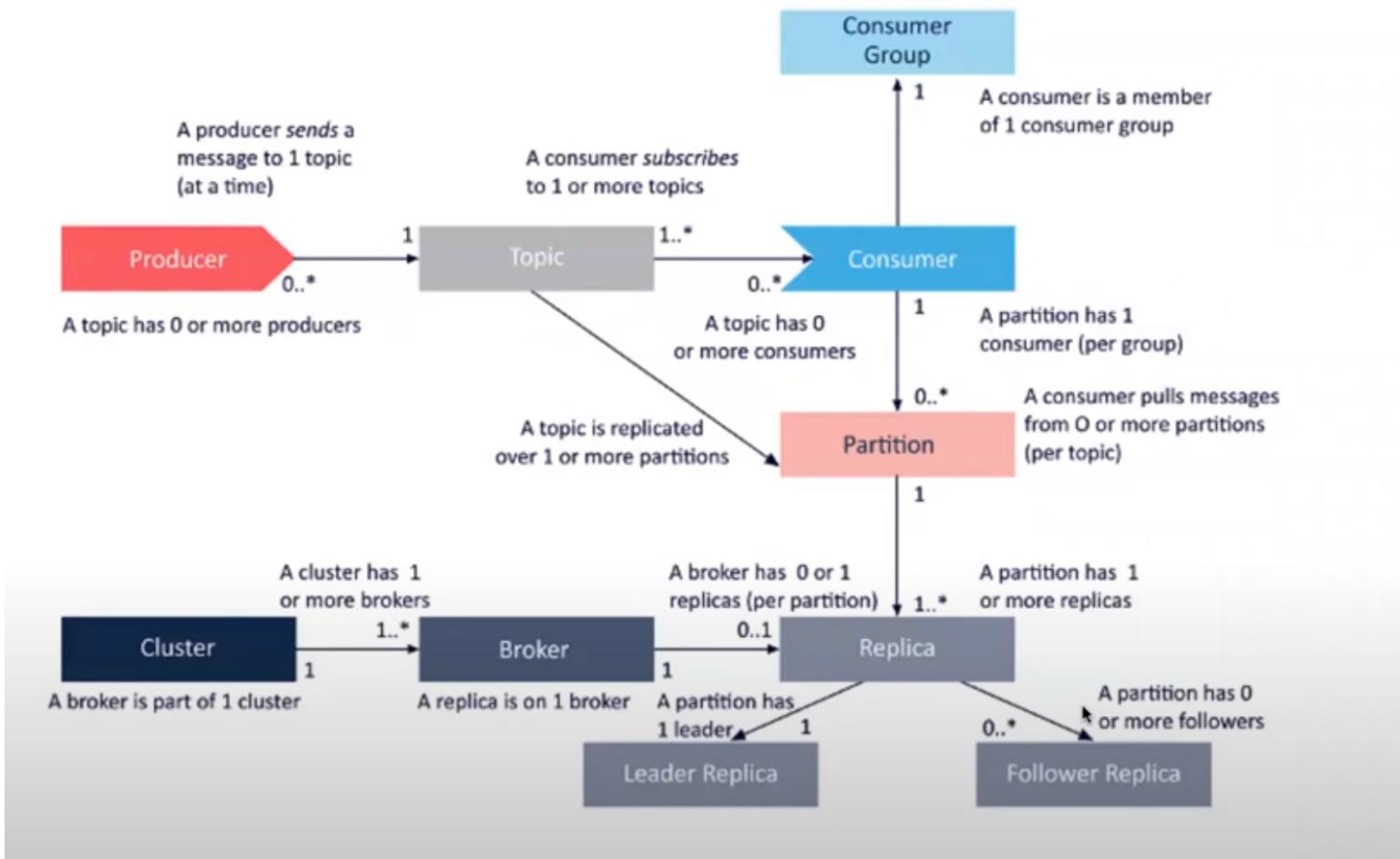
При этом один консьюмер подписан на сколько угодно партиций

Consumer group

Позволяет объединять чтение Consumer'ов так, чтобы они параллельно читали разные партиции



Consumer group



d. Гарантия доставки данных

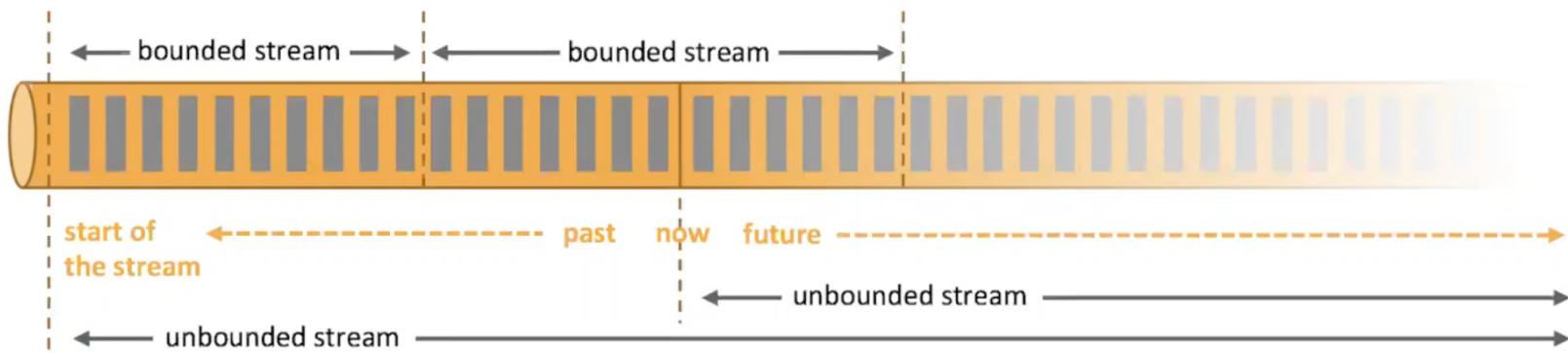
Если ожидать подтверждения от всех репликаций, то не будет data loss. Если от лидера – то возможно, а если не ждать вообще то обязательно будет data loss.

Мы сами выбираем какую стратегию используем

e. Kafka streaming – кейсы применения, внутреннее устройство

Stream: поток данных который генерируется во времени и куда-то приходит. 2 вида стримов:

- *bounded stream:* как с батчами
- *unbounded stream:* бесконечный процессинг



- Kafka содержит в себе пакет, который позволяет реализовывать стриминг на своих данных
- Позволяет добавлять стриминговый процессинг
- Per-record
- **Stateless, stateful, windowing** - виды операторов (считаем сумму и от типа стрима будет многое зависеть); windowing - берем окно длиной времени t
- Не требует отдельного кластера: не надо поднимать отдельный сервис
- Есть exactly-once семантика (*строго однократная доставка*)

Kafka Streams: Pros & Cons



- Легковесный фреймворк, хорошо подойдет для микросервисов
- Не нужен отдельный кластер
- Наследует все хорошее из самой Kafka
- Есть join'ы, хранит внутреннее состояние в RocksDB
- Exactly Once (Kafka 0.11 onwards).
- Невозможно использование без Kafka
- Достаточно новая технология, мало поддержки сообщества
- Изначально не предназначен для тяжелых вычислений

плюсы и минусы kafka

Вид стриминга который используется в кафке: **exactly once**

At most once	At least once	Exactly once
Message pulled once	Message pulled one or more times; processed each time	Message pulled one or more times; processed once
May or may not be received	Receipt guaranteed	Receipt guaranteed
No duplicates	Likely duplicates	No duplicates
Possible missing data	No missing data	No missing data

6. Apache Flink

Мастрид эти ссылки, но другие пункты в документации тоже важны:

<https://flink.apache.org/flink-architecture.html>

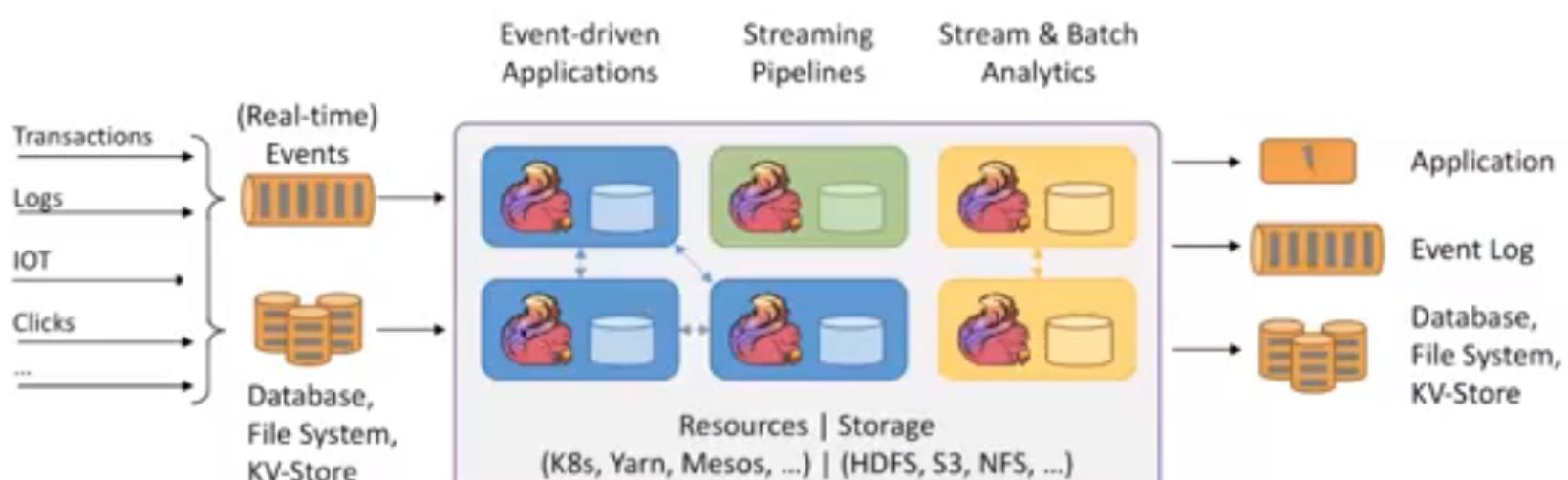
<https://flink.apache.org/usecases.html>

a. Принципы работы

Движок процессинга данных(честный стриминг). Более честные взаимодействия между разными пайплайнами. Pipelinis взаимодействуют друг с другом: **Event-driven applications, Streaming pipelines, Stream & batch analytics** (про супер адаптивные системы, которые умеют в быстрый фидбек для пользователей. Есть моментальный фидбек луп: т.е можем писать в наше приложение

Apache Flink

Движок для процессинга стримов данных

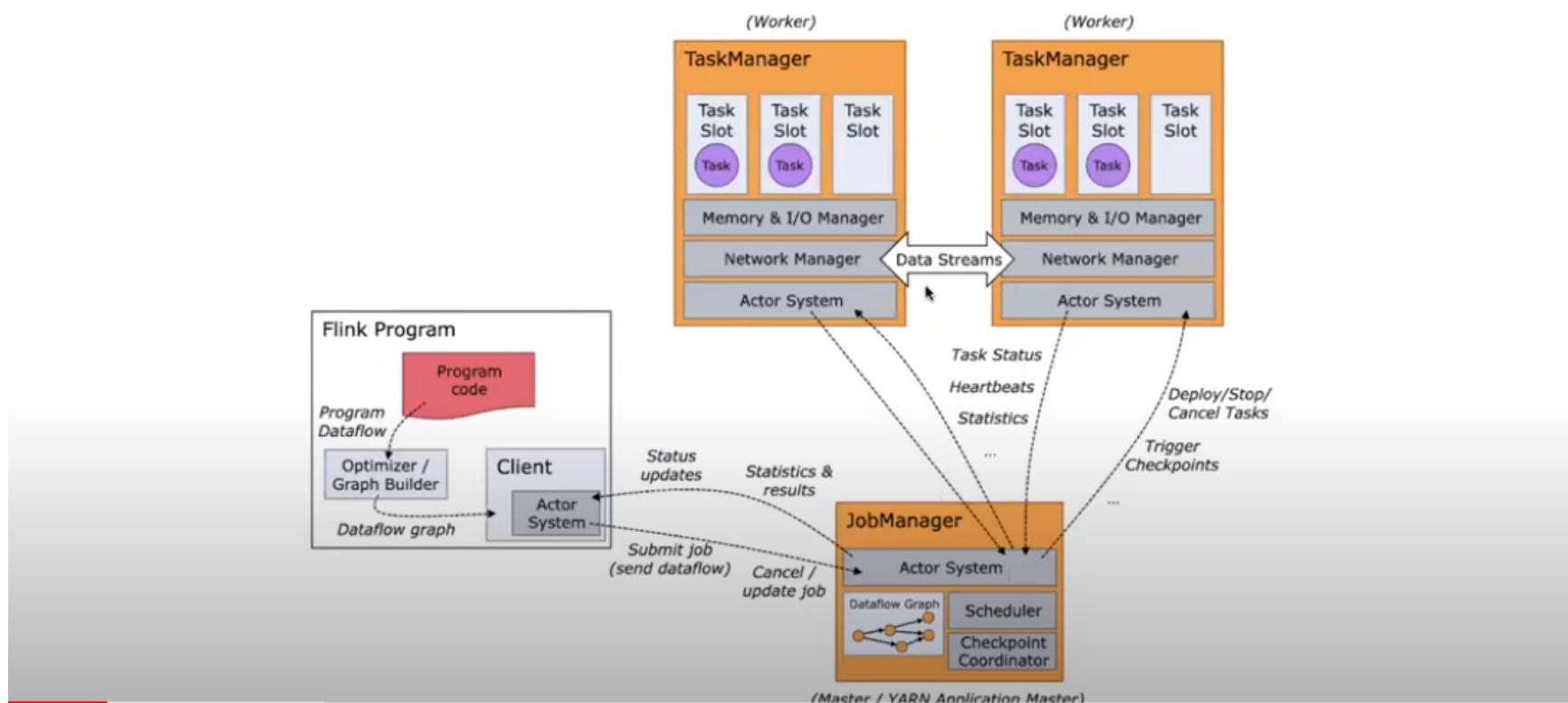


b. Устройство пайплайн

Есть основная нода которая управляет другими

+ появляются чекпоинты

Внутреннее устройство Flink



c. Flink vs Spark streaming

оба высокоуровневые

один трущийся другой нет

exactly one и там и там

Различия Spark Streaming и Flink

Оба фреймворка созданы
для стриминга, но
качественно различаются



Streaming	mini batches	"true"
API	high-level	high-level
Fault tolerance	RDD-based (lineage)	coarse checkpointing
State	external	internal
Exactly once	exactly once	exactly once
Windowing	restricted	flexible
Latency	medium	low
Throughput	high	high

sd. Checkpointing

Metadata checkpointing - определение самого приложения:

- **Configuration** - конфигурация streaming application
- **DStream operations** - операции над DStreams, которые определяют приложение
- **Incomplete batches** - батчи, на которых не завершился процессинг

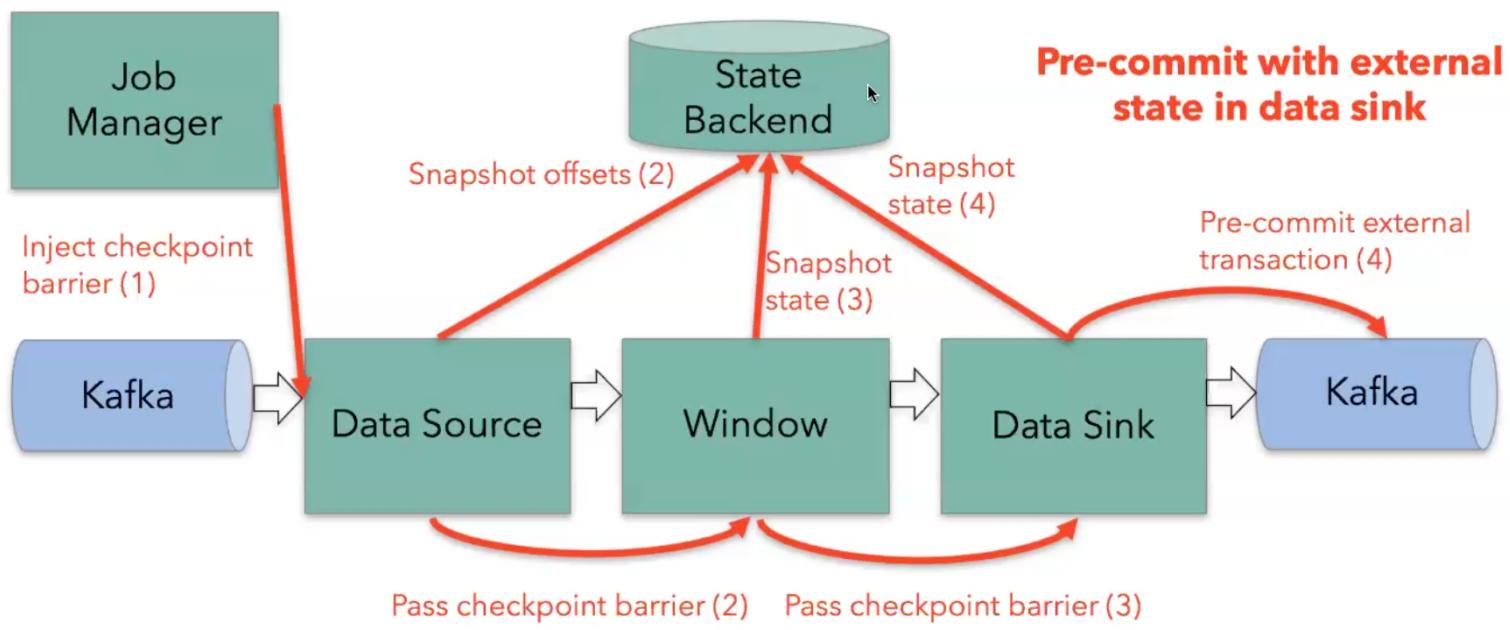
Data checkpointing - saving of the generated RDDs to reliable storage. Mainly state

Чем хорошо – если отвалится, запустим с чекпоинта и минимальные усилия уйдут и время для рестарта.

Job manager делает Checkpointing и в стрим прилетает сигнал что надо делать чекпоинт.

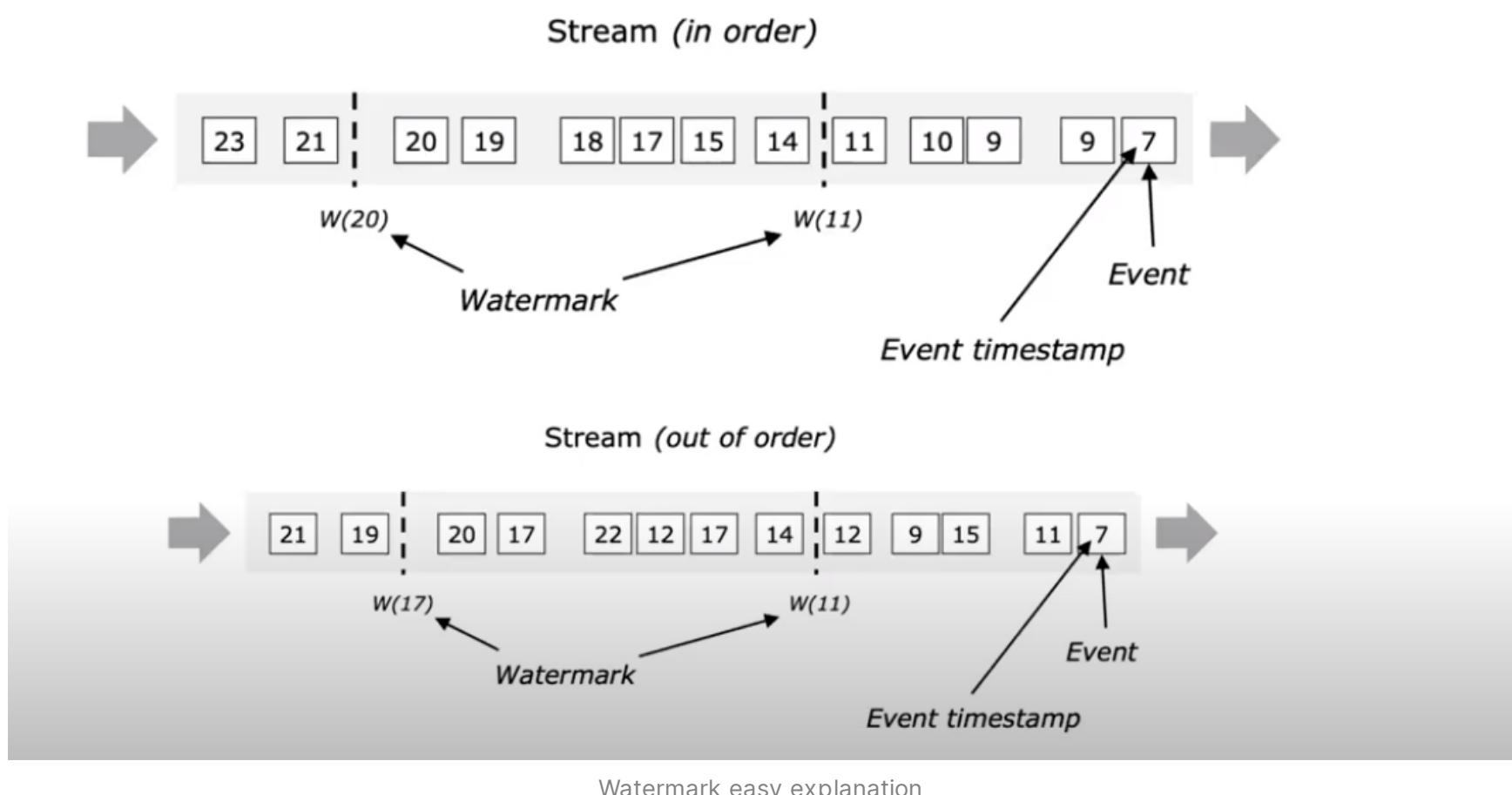
Exactly-once kafka + flink

Exactly-once two-phase commit



Watermark

Стрим данных может быть не упорядочен (к нам ивенты могут прийти не в том порядке). Поток ивентов идет не равномерно по времени. Брать максимальный таймстемп который у нас уже встречался и брать минуту. То есть если событие опаздывает не больше чем на минуту, то я его обработаю. Делим событие на опоздавшие и не опоздавшие

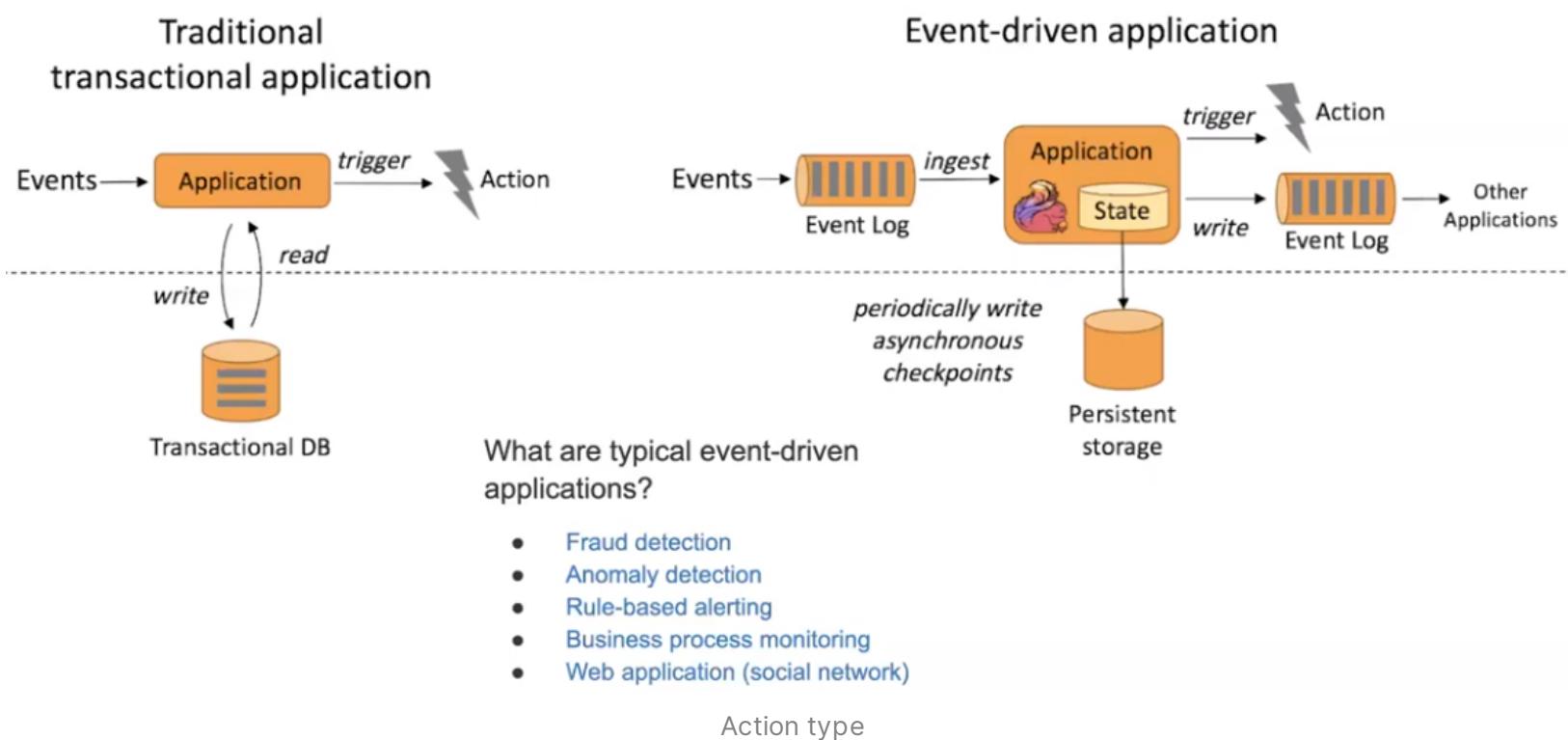


https://s3-us-west-2.amazonaws.com/secure.notion-static.com/1d25fbed-ed3a-441d-8061-7ac2e6b77e70/L7._Apache_beam.pdf

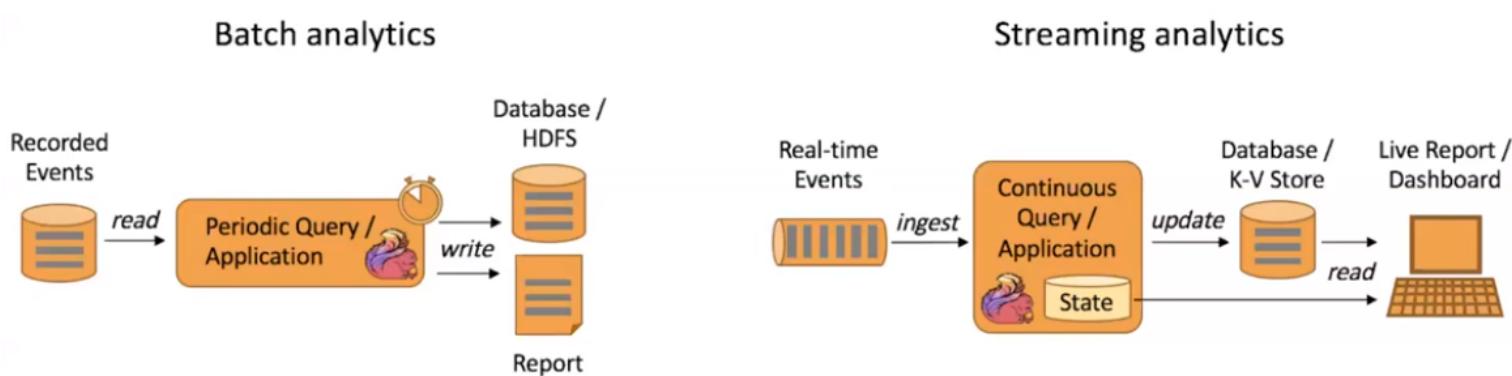
е. Кейсы применения

Спокойно себе читаем пишу в БД, если происходит какое-то особое событие вылетает триггер который позволяет что-то сделать (Fraud detection). + есть логи всех ивентов

Flink use cases: Event-driven application



Flink use cases: Analytics



7. Принципы работы с streaming data

(презентация Apache Beam и Dataflow)

а. Разные семантики доставки данных

Семантика at least once («хотя бы один раз»).

Если продюсер получает подтверждение от брокера Kafka, и при этом `acks=all`, это означает, что сообщение было записано в топик Kafka строго однократно. Но если продюсер не получает подтверждение по истечении тайм-аута или получает ошибку, то он может попробовать снова отправить сообщение, считая, что оно не было записано в топик Kafka. Если брокер дал сбой непосредственно перед отправкой подтверждения, но после того, как сообщение было успешно записано в топик Kafka, эта повторная попытка отправки приведёт к тому, что сообщение будет записано и отправлено конечному потребителю дважды. Такой подход приводит к дублированию работы и некорректным результатам.

Семантика at most once («не более одного раза»).

Если продюсер не производит повторную отправку сообщения по истечении тайм-аута или получения ошибки, то сообщение может не записаться в топик Kafka, и, следовательно, оно не будет доставлено потребителю. В большинстве случаев сообщения будут доставляться, но, чтобы избежать вероятности дублирования, мы допускаем, что иногда сообщения не доходят.

Семантика **exactly once** («строго однократная доставка»).

Даже при повторной попытке продюсера отправить сообщение, сообщение доставляется строго один раз. Семантика **exactly once** – наиболее желаемая гарантия, но при этом наименее понимаемая. Причина в том, что она требует взаимодействия между самой системой передачи сообщений и приложением, генерирующим и получающим сообщения. Например, если после удачного получения сообщения вы перемотаете Kafka-потребитель на предыдущее положение, то снова получите оттуда все до последнего сообщения. Это наглядно показывает, почему система обмена сообщениями и клиентское приложение должны взаимодействовать друг с другом, чтобы работала семантика exactly-once.

At most once	At least once	Exactly once
Message pulled once	Message pulled one or more times; processed each time	Message pulled one or more times; processed once
May or may not be received	Receipt guaranteed	Receipt guaranteed
No duplicates	Likely duplicates	No duplicates
Possible missing data	No missing data	No missing data

Семантики доставки

b. DataFlow model

(не было на лекциях)

Модель потока данных – это схематическое представление потока и обмена информацией внутри системы. Модели потоков данных используются для графического представления потока данных в информационной системе путем описания процессов, связанных с передачей данных от входных данных к файловому хранилищу и формированием отчетов.

8.* Вероятностные структуры данных

a. Фильтр Блума

Фильтр Блума представляет собой битовый массив из m бит. Изначально, когда структура данных хранит пустое множество, все m бит обнулены. Пользователь должен определить k независимых хеш-функций h_1, \dots, h_k , каждая из которых отображает множество элементов в множество мощностью m . (Иными словами, каждому элементу хеш-функция сопоставляется число от 1 до m .) Для каждого элемента e биты массива с номерами $h_1(e), \dots, h_k(e)$ равными значениям хеш-функций устанавливаются в 1.

Для проверки принадлежности элемента e к множеству хранимых элементов необходимо проверить состояние битов $h_1(e), \dots, h_k(e)$. Если хотя бы один из них равен нулю, элемент не может принадлежать множеству (иначе бы при его добавлении все эти биты были установлены). Если все они равны единице, то структура данных сообщает, что e принадлежит множеству. При этом может возникнуть две ситуации: либо элемент действительно принадлежит множеству, либо все эти биты оказались установлены по случайности при добавлении других элементов, что и является источником ложных срабатываний в этой структуре данных.

Независимость хеш-функций обеспечивает минимальную вероятность повторения индексов $h_k(e)$, минимизируя число бит установленных в 1 несколько раз. (А это главный источник ложноположительных срабатываний.)

b. Hyper log log

c. T-Digest