BIGDATA

Лекция 7

ПЛАН

- Spark RDD
- Spark DataFrame/SQL
- Kafka

Spark

- Предлагает несколько моделей
- Чуть более историческая - RDD
- Resilient Distributed Dataset
- Это такая распределенная коллекция
- С параллельной обработкой
- И устойчивостью к падениям

RDD

- Из чего можно создать RDD
  - Из обычной программной коллекции
  - Из обычного файла
  - Из HDFS-файла
  - Из всего, что выражается через InputFormat
  - Из HBase-таблицы
  - Из Hive-таблицы
  - Из всякого разного: Cassandra, S3 и т.п.

RDD

- Два типа операций: преобразования (transformations) и действия(actions)
- Пример преобразования - метод map
- Это не тот, который из MapReduce
- А в смысле FP
- Поэлементное преобразование
- Результат преобразования - новый RDD

RDD

- Есть еще метод reduceByKey
- Вот он больше похож на reduce из MR
- Редукция по ключам
- И результат - RDD
- Как и в MR
- То есть reduceByKey - преобразование

Преобразования

- Все преобразования - ленивые
- Действия являются триггером для материализации преобразований
- Если сохранить преобразование в переменной
- И применить отдельно два действия
- Оно может быть пересчитано
- Но можно явно заказать его переиспользование
- Методы cache и persist

Пример на Scala

```scala
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

Пример на Python

```python
lines = sc.textFile("data.txt")
lineLengths = lines.map(lambda s: len(s))
totalLength = lineLengths.reduce(lambda a, b: a + b)
```

Тонкости с функциями

- Параметрами преобразований и действий являются функции
- Функции работают удаленно - вообще говоря
- В духе map-reduce
- Не любой синтаксически корректный код будет корректен по смыслу
- Или будет корректен, но не оптимален

Рекомендации

- Для Scala
  - Анонимные функции
  - Статические методы в глобальном синглтоне
- Для Python
  - lambda
  - Локальные def в той же функции, где вызываем spark-метод
  - Функции верхнего уровня в том же модуле

Как оно работает

- Spark планирует каждое задание (job)
- И разбивает его на задачи (task)
- Задачи назначает исполнителям (executor)
- Учитывая, где находятся данные
- Их ожидаемый размер и т.п.

Как оно работает

- И отправляет код к месту работы
- Как правило - поближе к данным
- В частности, при наличии замыкания
- Оно корректно отправится к месту работы
- И если код в замыкании что-то поменяет
- То оно поменяется в удаленном слепке

Варианты преобразований

- map, filter - в классическом понимании
- flatMap - отображаем элемент в последовательность, не добавляя размерности
- flatMapValues - вариация на парах
- Отображаем второй элемент
- И разбиваем получившийся список на элементы
- Сохраняя первый элемент

Пример

```
1 rdd = sc.parallelize([2, 3, 4])
2 sorted(rdd.flatMap(lambda x: range(1, x)).collect())
3 # [1, 1, 1, 2, 2, 3]
4
5 sorted(rdd.flatMap(lambda x: [(x, x), (x, x)]).collect())
6 # [(2, 2), (2, 2), (3, 3), (3, 3), (4, 4), (4, 4)]
```

Пример

```
1 rdd = sc.parallelize([("a", ["x", "y", "z"]),
2                       ("b", ["p", "r"])])
3 def f(x): return x
4
5 rdd.flatMapValues(f).collect()
6 # [('a', 'x'), ('a', 'y'), ('a', 'z'),
7 #  ('b', 'p'), ('b', 'r')]
```

Варианты преобразований

- mapValues - почти как map
- работает над RDD пар
- И отображает вторые элементы

Пример

```
1 rdd = sc.parallelize([("a", ["apple", "banana", "lemon"]),
2                       ("b", ["grapes"])])
3 def f(x): return len(x)
4 rdd.mapValues(f).collect()
5 # [('a', 3), ('b', 1)]
```

Пример

```
1 rdd = sc.parallelize([("a", "x",), ("b", "p")])
2 def f(x): return [x, x]
3 rdd.flatMapValues(f).collect()
4 # [('a', 'x'), ('a', 'x'), ('b', 'p'), ('b', 'p')]
```

Группировки

- groupBy - группируем по значению функции от элемента
- Получаем RDD пар
- Первый элемент - значение, по которому группируемся
- Второй - итерируемый элемент по значениям
- Может быть очень большим

Пример

```
1 rdd = sc.parallelize([1, 1, 2, 3, 5, 8])
2 result = rdd.groupBy(lambda x: x % 2).collect()
3
4
5 sorted([(x, sorted(y)) for (x, y) in result])
6 [('a', 3), ('b', 1)]
```

Группировки

- groupByKey - на входе список пар
- На выходе - тоже список пар
- С уникальным первым элементом
- И итерируемым вторым

Пример

```
1 rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
2 sorted(rdd.groupByKey().mapValues(len).collect())
3 # [('a', 2), ('b', 1)]
4
5 sorted(rdd.groupByKey().mapValues(list).collect())
6 # [('a', [1, 1]), ('b', [1])]
```

Группировки

- cogroup - скорее по названию
- По сути это недоделанный OUTER JOIN
- Получаем RDD с уникальными ключами, встречавшимися хотя бы в одной RDD
- И парой списков во втором элементе пары
- В списке - встречавшиеся вторыми элементами при данном ключе

Пример

```
1 rdd1 = sc.parallelize([("a", 1), ("b", 4)])
2 rdd2 = sc.parallelize([("a", 2)])
3 [(x, tuple(map(list, y)))
4     for x, y in sorted(list(rdd1.cogroup(rdd2).collect()))]
5 # [('a', ([1], [2])), ('b', ([4], []))]
```

Обобщение

- groupWith - почти как cogroup
- Только можно вовлекать несколько групп

Пример

```
1 rdd1 = sc.parallelize([("a", 5), ("b", 6)])
2 rdd2 = sc.parallelize([("a", 1), ("b", 4)])
3 rdd3 = sc.parallelize([("a", 2)])
4 rdd4 = sc.parallelize([("b", 42)])
5 [(x, tuple(map(list, y))) for x, y in
6     sorted(list(rdd1.groupWith(rdd2, rdd3, rdd4)
7                 .collect()))]
8 # [('a', ([5], [1], [2], [])), ('b', ([6], [4], [], [42]))]
```

Свертка

- foldByKey - свертка по ключам
- Для значений одного ключа выполняем классическую свертку
- Явно указывая начальный элемент
- Который должен быть нулем/единицей в смысле теории групп
- Тип аккумулятора/результата равен типу элемента

Пример

```
1 rdd = sc.parallelize([("a", 1), ("b", 7), ('b', 2),
2                       ("a", 3), ('b', -1)])
3 from operator import add
4 sorted(rdd.foldByKey(0, add).collect())
5 # [('a', 4), ('b', 8)]
```

Свертка

- keyBy - порождает ключ
- На входе - RDD с элементами типа T
- На выходе - RDD с элементами типа (K, T)
- K - тип ключа

Пример

```
1 rdd1 = sc.parallelize(range(0,3)).keyBy(lambda x: x*x)
2 rdd2 = sc.parallelize(zip(range(0,5), range(0,5)))
3 [(x, list(map(list, y)))
4     for x, y in sorted(rdd1.cogroup(rdd2).collect())]
5
6 # [(0, [[0], [0]]), (1, [[1], [1]]), (2, [[], [2]]),
7 #  (3, [[], [3]]), (4, [[2], [4]])]
```

Join

- join - сцепление по ключам
- Каждый совпавший ключ порождает n1 * n2 элементов в результате
- Значения - пары значений из аргументов
- leftOuterJoin, rightOuterJoin, fullOuterJoin

Пример

```
1 rdd1 = sc.parallelize([("a", 1), ("b", 4)])
2 rdd2 = sc.parallelize([("a", 2), ("a", 3)])
3 sorted(rdd1.join(rdd2).collect())
4 # [('a', (1, 2)), ('a', (1, 3))]
```

Пример

```
1 rdd1 = sc.parallelize([("a", 1), ("b", 4)])
2 rdd2 = sc.parallelize([("a", 2), ("c", 8)])
3 sorted(rdd1.fullOuterJoin(rdd2).collect())
4 # [('a', (1, 2)), ('b', (4, None)), ('c', (None, 8))]
```

Пример

```
1 rdd1 = sc.parallelize([("a", 1), ("b", 4)])
2 rdd2 = sc.parallelize([("a", 2)])
3 sorted(rdd1.leftOuterJoin(rdd2).collect())
4 # [('a', (1, 2)), ('b', (4, None))]
```

Преобразования по разделам

- mapPartitions - отображение по разделам
- Параметр - функция, отображающая Iterable в Iterable
- Другой тип преобразований по сравнению с map
- Объект преобразования - совокупность данных с раздела
- mapPartitionsWithIndex - с нумерацией передаваемых элементов

Пример

```
1 rdd = sc.parallelize([1, 2, 3, 4], 2)
2 def f(iterator): yield sum(iterator)
3
4 rdd.mapPartitions(f).collect()
```

Пример

```
1 rdd = sc.parallelize([1, 2, 3, 4], 4)
2 def f(splitIndex, iterator): yield splitIndex
3 rdd.mapPartitionsWithIndex(f).sum()
4 def f(splitIndex, iterator):
5     yield (splitIndex, sum(iterator))
6 rdd.mapPartitionsWithIndex(f).collect()
7 # [(0, 1), (1, 2), (2, 3), (3, 4)]
```

Варианты преобразований

- aggregateByKey - посложнее, на мотив MapReduce
- Три функции-параметра
- partitionFunc - разбиение на разделы по ключу
- seqFunc - свертка на разделе
- combFunc - редукция результатов разделов

Пример

```
1 rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 2)])
2 seqFunc = (lambda x, y: (x[0] + y, x[1] + 1))
3 combFunc = (lambda x, y: (x[0] + y[0], x[1] + y[1]))
4 sorted(rdd.aggregateByKey((0, 0), seqFunc,
5         combFunc).collect())
```

Действия

- foreach/forEachPartition
- Выполняет функцию над каждым элементом
- Или над каждым разделом
- Данные продолжают свое существование

Пример

```
1 d = sc.parallelize(range(1000), 20)
2 d.foreachPartition(lambda v: print(list(v)))
3 d.foreachPartition(lambda v: print(list(v)))
```

Действия

- count - понятно
- countAprox - примерная точность
- Пытаемся посчитать
- "maximum time", "desired confidense"
- countApproxDistinct - оценка через HyperLogLog

Пример

```
>>> n = sc.parallelize(range(1_000_000)).map(str)
           .countApproxDistinct()
>>> n
1005571
>>> n = sc.parallelize(range(1_000_000)).map(str)
            .countApproxDistinct(0.01)
>>> n
1003494
>>> n = sc.parallelize(range(1_000_000)).map(str)
             .countApproxDistinct(0.001)
>>> n
1000075
```

KAFKA

- Брокер сообщений
- Фактически - база данных
- Потоковый key-value storage
- Производный продукт - Kafka streams

ИДЕЯ

- Данные хранятся в виде лога
- Запись идет в хвост последовательности
- Чтение последовательностей целиком
- Взамен получаем высокую пропускную способность

EVENT SOURCING

- Event Sourcing - сопутствующая парадигма проектирования
- CQRS - Command and Query Responsibility Segregation
- Классическое проектирование базируется на сущности (Entity)
- И на ее состоянии

EVENT SOURCING

- Ключевая задача - обновлять состояние и одновременно читать
- Возможно, для разных целей и с разным шаблоном нагрузки
- Обеспечивать консистентность
- Какие-то логи могут присутствовать (WAL, триггеры)
- Но скорее как ограниченное решение частных проблем

EVENT SOURCING

- ES делает поток событий первичным источником истины
- Первая задача - зафиксировать его
- Из него можно порождать состояние
- Или разные наборы состояний
- Или не из него, а из его проекции

ТЕРМИНЫ

- event - базовая единица хранения
- Пара ключ-значение + timestamp
- Интерпретация значения - дело клиентов
- Часто соответствуют событиям реального мира
- Но не обязательно

ТЕРМИНЫ

- producer - клиент, пишущий event-ы
- consumer - клиент, подписывающийся на event-ы и читающий их
- Взаимодействие producer-ов и consumer-ов в целом асинхронное
- Но возможны нюансы

ТЕРМИНЫ

- topic - смысловой раздел, в который записывается event
- Любой event пишется в какой-то topic
- Можно копии event-а записать в разные topic-и
- Для kafka это разные event-ы
- Не бывает event-а вне топика

ТЕРМИНЫ

- partition - элемент разбиения topic-а
- И физического, и логического
- Каждый event попадает ровно в один partition
- По умолчанию partion определяется по хешу ключа

ТЕРМИНЫ

- Ключа может не быть
- Тогда round robin
- Можно произвольно писать event в какой-либо partition
- Порядок чтения из partition-а совпадает с порядком записи

ТЕРМИНЫ

- Порядок чтения из topic-а отдельно не гарантируется
- Если один partition - то порядок чтения повторяет порядок записи
- partition-ы могут находиться на разных машинах
- topic-и могут реплицироваться
- Физически реализуется через репликацию partition-ов

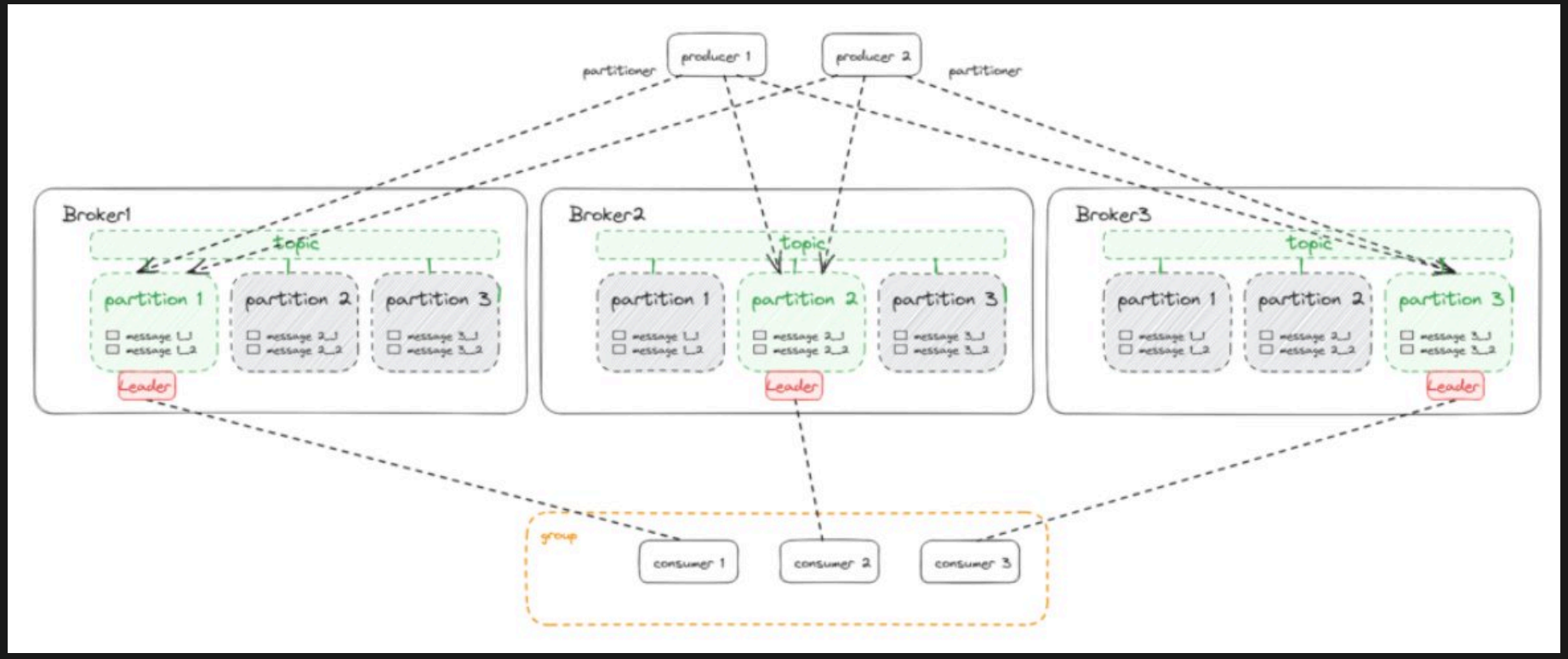
ТЕРМИНЫ

- broker - физическая машина, элемент кластера
- broker физически хранит partition-ы
- В идеале - все partition-ы topic-а должны жить на разных broker-ах

ТЕРМИНЫ

- replica - физическая копия partition-а
- Используется исключительно как резервная копия
- В любой момент времени и чтение, и запись идут в лидирующей реплике
- Она называется partition leader

КАРТИНКА


producer 1
producer 2
partitioner
partitioner
Broker1
Broker2
Broker3
topic
topic
topic
partition 1
partition 2
partition 3
partition 1
partition 2
partition 3
partition 1
partition 2
partition 3
Leader
Leader
Leader
group
consumer 1
consumer 2
consumer 3

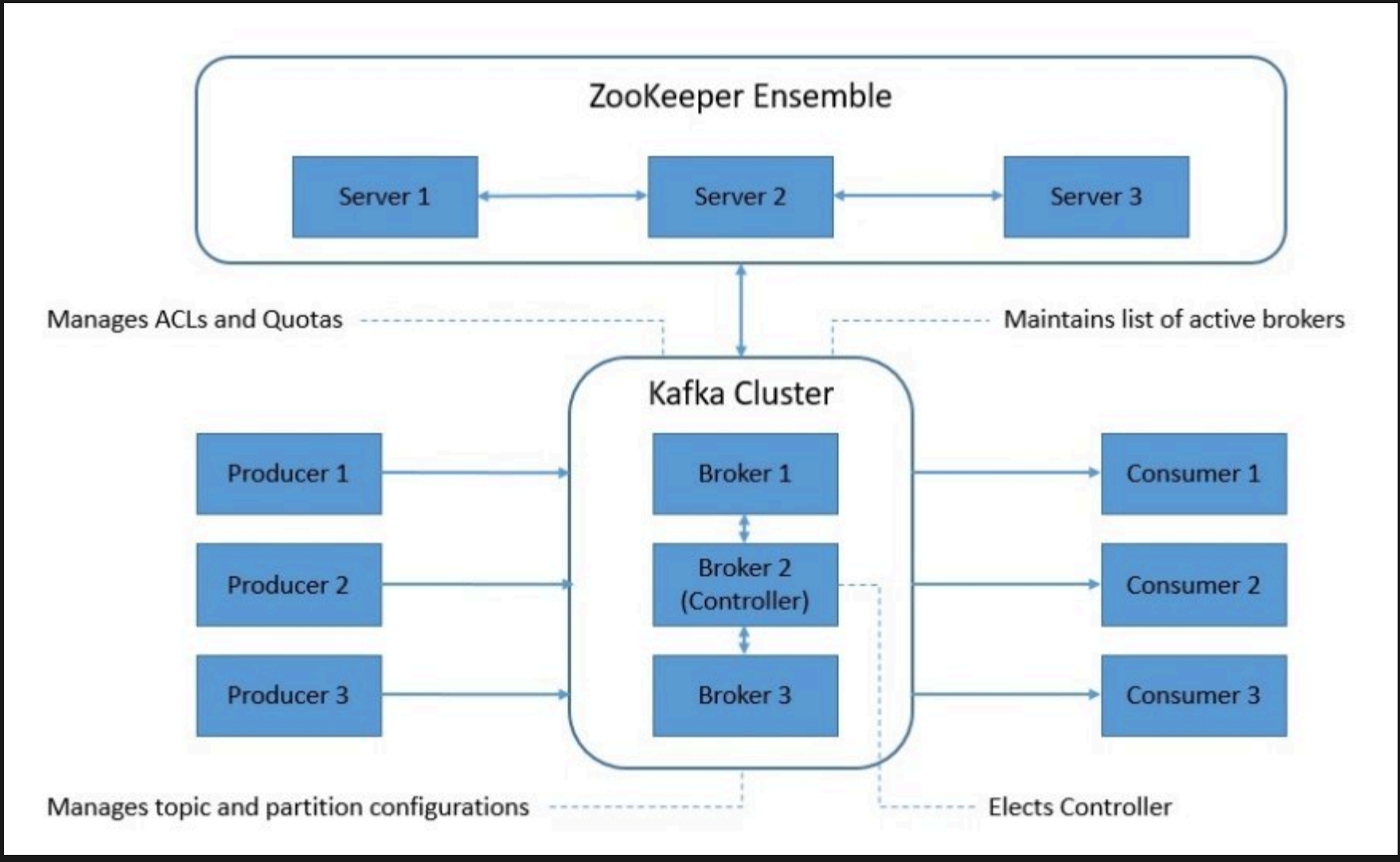
РАСПРЕДЕЛЕННОСТЬ

- Kafka-кластер спроектирован под широкую масштабируемость
- Одна из предпосылок - в нем нет лидера кластера в целом
- Есть лидер у каждого partition-а
- Клиент, пишущий в partition, общается непосредственно с его лидером

РАСПРЕДЕЛЕННОСТЬ

- Но нужно знать, где находится лидер конкретного partition-а
- И лидера надо как-то определять
- И то, и другое умеет делать Zookeeper
- Это внешнее по отношению к Kafka средство
- Очень консистентный мини-кластер

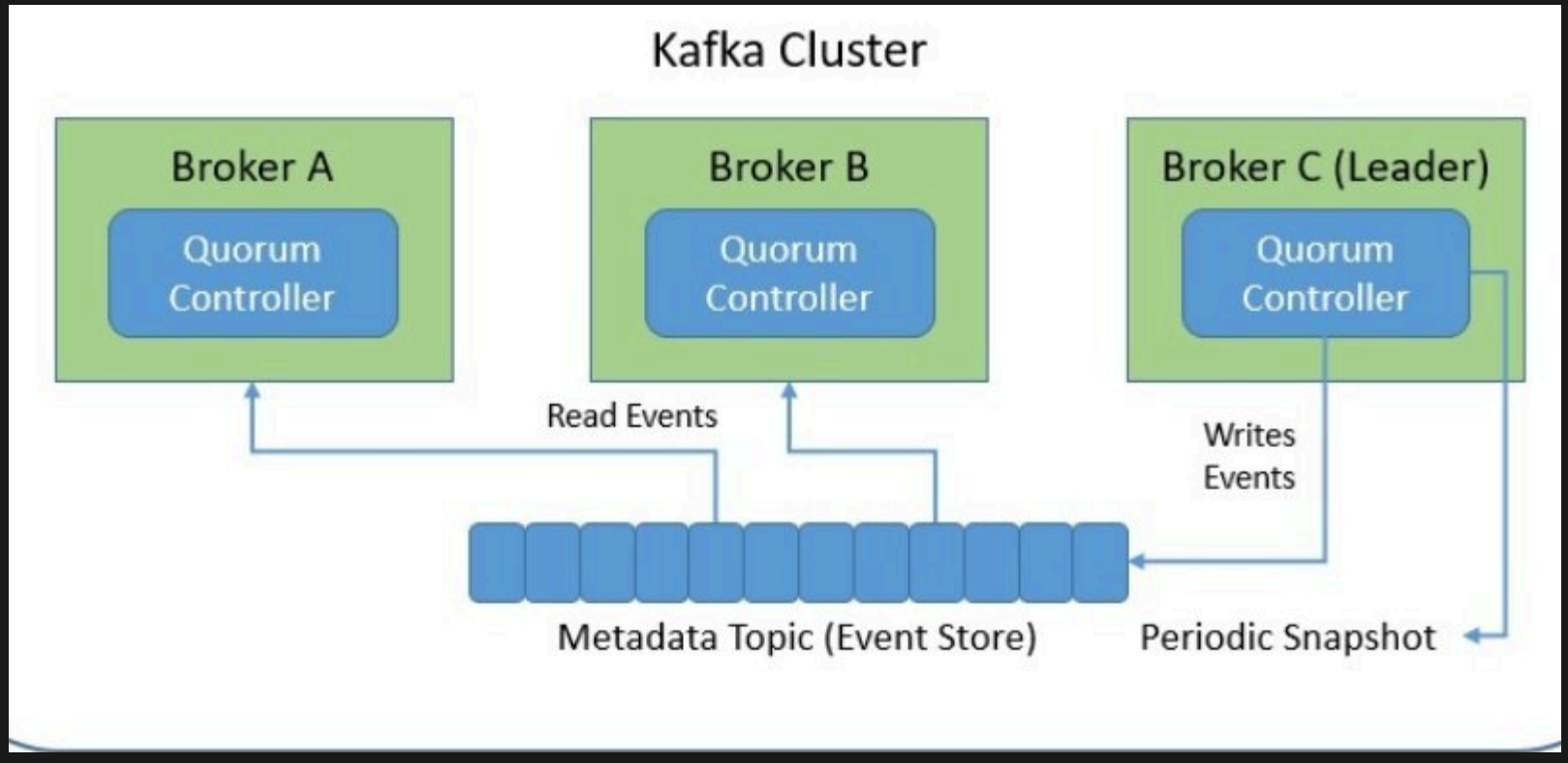
КАРТИНКА


ZooKeeper Ensemble
Server 1
Server 2
Server 3
Manages ACLs and Quotas
Maintains list of active brokers
Kafka Cluster
Producer 1
Producer 2
Producer 3
Broker 1
Broker 2
(Controller)
Broker 3
Consumer 1
Consumer 2
Consumer 3
Manages topic and partition configurations
Elects Controller

KRAFT

- Kafka реализует свой аналог - протокол KRaft
- Одна из целей - упростить структуру системы
- Другая - использования event-sourcing для метаданных
- KRaft организует переливку лога метаданных от активного контроллера к другим контроллерам

КАРТИНКА


Kafka Cluster
Broker A
Quorum Controller
Broker B
Quorum Controller
Broker C (Leader)
Quorum Controller
Read Events
Writes Events
Metadata Topic (Event Store)
Periodic Snapshot

ВЗАИМОДЕЙСТВИЕ

- Базовое API - Java
- Клиентские библиотеки для других языков
- Есть клиентские приложения в дистрибутиве
- Позволяют экспериментировать и базово мониторить состояние

ЗАПУСК СЕРВЕРА

```
export  KAFKA_CLUSTER_ID="$(bin/kafka-storage.sh random-uuid)"

./bin/kafka-storage.sh format -t $KAFKA_CLUSTER_ID -c config/kraft

./bin/kafka-server-start.sh config/kraft/server.properties
```

БАЗОВЫЕ ОПЕРАЦИИ

```
./bin/kafka-topics.sh --create --topic quickstart-events --bootstra

./bin/kafka-topics.sh --describe --topic quickstart-events --bootst

./bin/kafka-console-producer.sh --topic quickstart-events --bootstr
>first
>second

.bin/kafka-console-consumer.sh --topic quickstart-events --from-beg
```

ПАРАДИГМЫ ОБМЕНА СООБЩЕНИЯМИ

- publish-subscribe: отправитель пишет сообщения
- Желающие читают из topic-а
- Независимо друг от друга
- Kafka позволяет такой режим по желанию читателя

ПАРАДИГМЫ ОБМЕНА СООБЩЕНИЯМИ

- queue: отправитель пишет сообщения
- Хотим распараллелить обработку
- Чтобы работало несколько consumer-ов
- И они совместно обрабатывали события topic-а

ПАРАДИГМЫ ОБМЕНА СООБЩЕНИЯМИ

- Для этого они объединяются в группу - consumer group
- Группа распознается по общему идентификатору группы
- В момент подписки на topic consumer сообщает о желании быть членом группы
- Дальше все зависит от количества partition-ов в topic-е
- И от количества consumer-ов в группе

CONSUMER GROUP

- У каждой группы есть координатор
- Это один из брокеров
- Он отслеживает сердцебиение всех членов группы
- И информирует лидера группы, если обнаруживаются проблемы

CONSUMER GROUP

- Лидер группы - один из клиентов
- Он распределяет partition-ы по consumer-ам
- Кто первый пришел в пустую группу - становится лидером
- И сам себе назначает все partition-ы

CONSUMER GROUP

- По мере пополнения группы лидер перераспределяет partition-ы
- Существуют несколько конфигурируемых стратегий распределения
- Общий момент - из одного partition-а читает только один consumer
- Количество partition-ов задает лимит на распараллеливание чтения topic-а

CONSUMER GROUP

- Могут быть и косвенные лимиты
- Например, можно создать topic на 20 partition-ах
- И разместить на одном broker-е
- Не факт, что 20 consumer-ов дадут близкое к 20х ускорение

CONSUMER GROUP

- Для каждой partition-а хранится свой offset
- Он хранится в отдельном служебном topic-е
- Мы персистентно знаем, на какой записи остановилась consumer group
- Но есть нюанс

УДАЛЕНИЕ ДАННЫХ

- Нет явного удаления
- Но возможно удаление или уплотнение
- Настраивается на уровне topic-а
- Конфигурация указывается при создании
- Может быть изменена впоследствии

УДАЛЕНИЕ ДАННЫХ

- Удалять можно по таймауту или по размеру
- Размер - применяется к partition-у
- Гранулярность - log segment
- Поэтому возможна задержка с реальным удалением

УПЛОТНЕНИЕ ДАННЫХ

- Уплотнение - проход по всем ключам
- С целью гарантированно оставить последнее упоминание каждого ключа в каждом topic-е
- Корректность уплотнения как операции зависит от семантики данных
- Удаление может совмещаться с уплотнением

CONNECT

- Kafka Connect: инфраструктура для взаимодействия с другими хранилищами
- Абстракция зачитывания чего-либо в Kafka topic
- Абстракция записи чего либо из Kafka topic-а
- Конкретные коннекторы, реализующие абстракции
- Существуют коннекторы для всех мыслимых СУБД

STREAMS

- Kafka Streams - клиентская библиотека на Java/Scala
- Абстракция в духе функционально-декларативного подхода
- topic рассматривается как абстрактный поток
- Над потоком можно выполнять операции