Jakub "morgul" Adamczyk

Analiza Dużych Zbiorów Danych - opracowanie 2021/2022

Big data processing

Introduction to big data

Speedup:

- czas na 1 węźle / czas na N węzłach
- miara zysku z wykorzystania równoległości, pozwala wziąć pod uwagę synchronizację, narzut na komunikację, fragmenty sekwencyjne etc.
- idealnie liniowy, realnie mniejszy

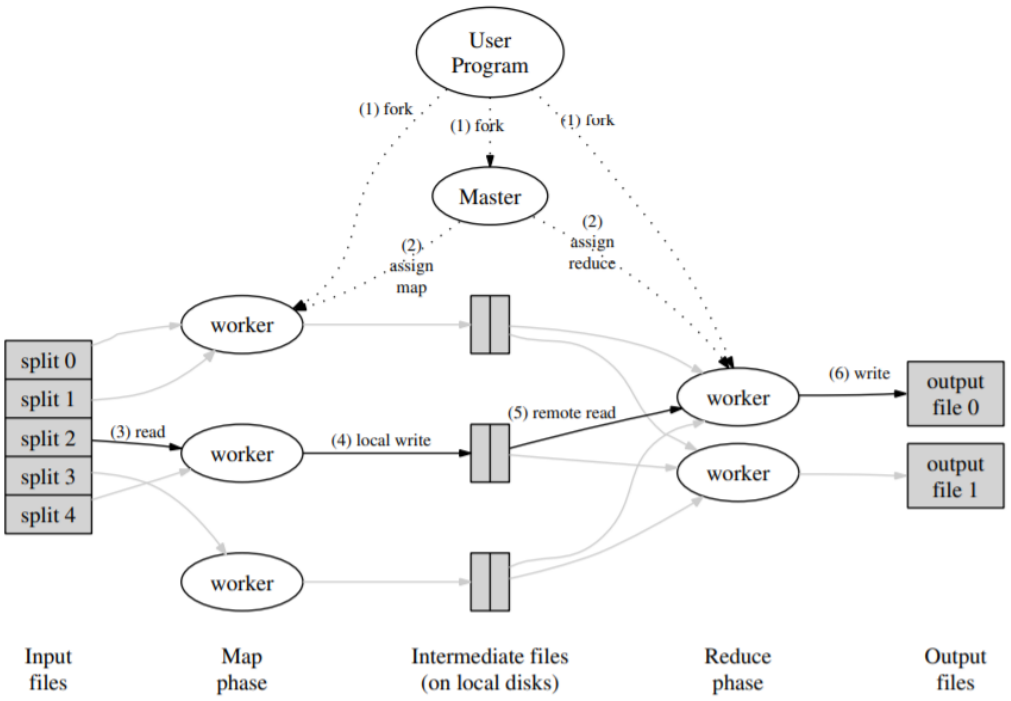
Parallel efficiency:

- speedup / N
- pozwala wyrazić przyspieszenie w %
- idealnie 100%, realnie mniej

Chcemy, żeby jak największa część naszego problemu była współbieżna. Jeżeli całe zadanie można podzielić na niezależne podzadania, to mamy problem **embarrassingly parallel**.

MapReduce:

- model obliczeń rozproszonych
- architektura:

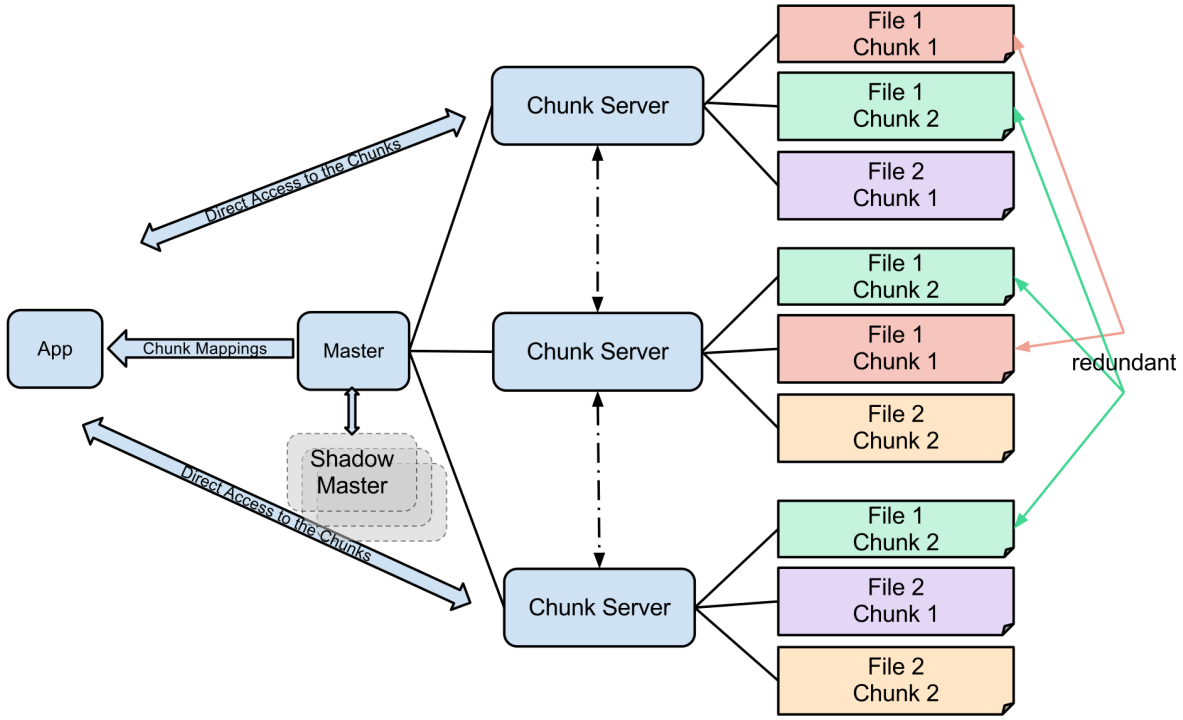


- dobry dla problemów dających się dobrze zrównoleglić i wymagających potem zebrania wyników częściowych obliczonych równolegle

- 2 klasy operacji:
    - map - wykonujemy operację dla każdego elementu
    - reduce - wykonujemy operację dla kilku przetworzonych elementów, redukując je do jednego wyniku
- kiedy oba rodzaje operacji można efektywnie wykonywać współbieżnie, to można oszczędzić czas i pamięć
- może być potrzeba dodatkowej operacji shuffle, która przesyła dane między workerami, szczególnie po map i przed reduce; chcemy tego możliwie unikać, bo jest kosztowna (maksymalizujemy data locality)
- wykonuje się na klastrze, którym zarządza master node, a pracę wykonują worker nodes
- przykłady typowych zadań:
    - NLP, np. zliczanie wystąpień słów w tekście (word count), budowanie macierzy term-vector (wektor częstości słów per użytkownik/host)
    - przetwarzanie logów np. znajdź linie zawierające string (distributed grep), znajdź najczęściej używane adresy URL
    - odwracanie grafu web-link - z grafu [source, list(target)] budujemy listę [target, list(source)]
    - odwracanie indeksu - zamiana mapy {doc_id: list(words)} na {word: list(doc_id)}
- budowa klastra:
    - Just a Bunch of Disks (JBOD)
    - zwykły typowy hardware, żadnych dedykowanych rozwiązań
    - rozproszony storage, z danymi przetwarzanymi bezpośrednio na węźle
    - różni się to znacząco od typowych architektur klastrów HPC, gdzie węzły obliczeniowe są w ogóle bezdyskowe, zamiast tego jest jeden współdzielony centralny system plików

Google File System (GFS):

- rozproszony storage, wykorzystujący partycjonowanie i replikację danych
- architektura:



- duże chunki (64 MB), wykorzystywany do bardzo dużych plików
- minimalizuje udział master node'ów - klient dostaje od mastera lokalizację plików, a serwery z danymi bezpośrednio przesyłają dane do aplikacji
- przeznaczony do komunikacji service-to-service, nie user-to-service (narzędzie programistyczne, a nie zwykła rzecz do używania bezpośrednio)
- na jego podstawie stworzono open source'ową wersję, Hadoop File System (HDFS)

Google Bigtable:

- distributed data storage system, wykorzystuje GFS
- wykorzystywany w usługach Google'a, udostępniany w ramach Google Cloud Datastore w GCP
- wide-column datastore, rodzaj tabelarycznej bazy danych, gdzie kolumny mogą różnić się między wierszami
- na jego podstawie stworzono open source'ową wersję, HBase

Apache Hadoop:

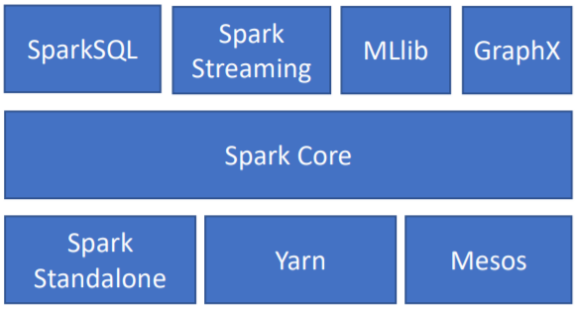
- implementacja modelu MapReduce w Javie
- wykorzystuje HDFS (Hadoop File System) i YARN (Yet Another Resource Manager)
- wzorowany na rozwiązaniach Google'a, ale w pełni open source

COST:

- Configuration that Outperforms a Single Thread
- miara skalowalności systemu, wyznacza jak mocnego sprzętu i równoległości trzeba, żeby był lepszy niż rozsądna prosta implementacja jednowątkowa
- bierze pod uwagę narzut na komunikację
- wiele systemów ma COST rzędu setek rdzeni, więc w wielu przypadkach lepiej je uruchomić w najprostszej, jednowątkowej wersji

Apache Spark:

- framework do przetwarzania w pamięci dużych zbiorów danych
- stworzony jako ewolucja Hadoopa, żeby przyspieszyć jego działanie - Hadoop pozwala co prawda przetworzyć ogromne dane, ale jest przy tym dość wolny
- najważniejsze ulepszenia:
    - przetwarzanie danych w pamięci - Hadoop jest dyskowy
    - użycie RDD (Resilient Distributed Dataframe) - niezawodność
- Spark Core - niskopoziomowe API, działanie bezpośrednio na RDD, niezalecane do bezpośredniego użycia
- architektura:



    - konfiguracja i zarządzanie klastrem - albo customowe rozwiązanie Sparka standalone, albo standardowy YARN (jak w MapReduce), albo Mesos
    - Spark Core, realizujący faktyczne operacje
    - wysokopoziomowe abstrakcje, z których faktycznie korzystamy:
        - Spark SQL - DataFrame API podobne do Pandasa + SQL
        - Spark Streaming - przetwarzanie strumieni
        - MLlib - rozproszone algorytmy ML
        - GraphX - przetwarzanie grafów oparte o framework Pregel

Przetwarzanie chmurowe:

- współczesna architektura big data opiera się na cloud computingu
- rozproszone geograficznie maszyny wirtualne (VM) jako węzły do obliczeń
- idea: dostęp do zdalnych zasobów obliczeniowych jako usług na żądanie
- szczególnie przydatne do zadań batchowych (np. typowy Spark):
    - ekonomiczne - płacimy tylko za zużycie, a duże analizy batchowe odpalamy typowo tylko raz na czas
    - nie przeszkadza nam opóźnienie (latency) między węzłami obliczeniowymi
    - nadają się dobrze do zadań niezależnych
    - model MapReduce został stworzony do takich środowisk

5 cech modelu chmurowego (według NIST):

- na żądanie (on-demand self-service) - klient może w dowolnym momencie zażądać wybranych zasobów obliczeniowych i to bez interakcji z człowiekiem
- szeroki dostęp sieciowy (broad network access) - dostęp do usług odbywa się przez sieć i jest możliwy dla różnorodnych urządzeń, o różnej mocy sieciowej
- pule zasobów (resource pooling) - zasoby obliczeniowe cloud providera są zebrane w pule, żeby obsłużyć wielu klientów w modelu multi-tenant, z dynamicznym przydziałem zasobów wedle zapotrzebowania
- szybka elastyczność (rapid elasticity) - możliwości obliczeniowe mogą być przyznawane (provisioned), skalowane i uwalniane wedle zapotrzebowania, w tym automatycznie, a przy tym często transparentnie
- zapłata za zmierzone zużycie (measured service, pay-per-use) - cloud provider sam kontroluje zużycie zasobów przez klienta, w sposób adekwatny do usługi, raportuje je i obciąża kosztami tylko za realne zużycie

Modele usług chmurowych:

- Infrastructure as a Service (IaaS):
    - cloud provider zapewnia infrastrukturę i zasoby obliczeniowe, klient ma kontrolę nad resztą
    - zapewnione: maszyny wirtualne i wszystko co do nich potrzebne (serwery, dyski, sieć, wirtualizacja)
    - każdy duży cloud provider to zapewnia, np. AWS EC2, Google Compute Engine (GCE)
- Container as a Service (CaaS):
    - zapewnione środowisko do uruchamiania gotowych kontenerów i zarządzania nimi
    - zapewnione: klaster VMek do uruchamiania na nich kontenerów (same VM są wyabstrahowanie) i system do zarządzania nimi (typowo Kubernetes)
    - np. AWS Elastic Container Service (ECS), AWS Elastic Kubernetes Service (EKS), Google Kubernetes Engine (GKE)
- Platform as a Service (PaaS)
    - cloud provider zapewnia pełne środowisko programistyczne, klient wykorzystuje je do programowania, zapewniając dane i kod
    - zapewnione: IaaS + zarządzanie VMkami (system, middleware i komunikacja, zarządzanie działaniem)
    - gotowe środowiska chmurowe, np. AWS Elastic Beanstalk, Google App Engine
- Function as a Service (FaaS):
    - podstawowa usługa serverless, pojedyncza funkcja (operacja) uruchamiająca się w odpowiedzi na konkretne zdarzenie
    - krótkie, pojedyncze operacje, niezależne od innych, bezstanowe (stateless)
    - cloud provider zapewnia:
        - monitoring triggera (eventu włączającego)
        - start
        - wykonanie współbieżne
        - load balancing
        - autoscaling

    - np. AWS Lambda, Google Cloud Function (GCF)
- Software as a Service (SaaS):
    - cloud provider zapewnia wszystko od początku do końca i opakowuje to w gotową do wykorzystania aplikację, klient tylko z niej korzysta
    - zapewnione: wszystko, klient podłącza się do usługi i z niej korzysta
    - gotowe usługi, np. Amazon QuickSight, Cisco WebEx

Model MapReduce

Cechy MapReduce:

- zapewnia:
    - automatyczne zrównoleglenie i rozproszenie obliczeń
    - odporność na błędy (fault tolerance)
    - I/O scheduling
    - monitoring, aktualizacje statusów
- 2 główne cele projektowe:
    - skalowalność
    - cost-efficiency
- ma się skalować do ogromnych danych, przetwarzać masywnie w sposób równoległy, oczywiście działając w chmurze
- wydajność kosztowa ma swoje konsekwencje projektowe:
    - commodity hardware - używamy zwykłego sprzętu serwerowego, nic dedykowanego jak w HPC, sprzęt jest tani, ale niepewny (unrealiable), więc trzeba wbudować fault tolerance
    - commodity network - mamy zwyczajne sieci, nic dedykowanego jak w HPC, przepustowości będą niskie, więc trzeba wykonywać obliczenia tuż przy danych na węzłach
    - automatic fault-tolerance - skoro i tak musimy mieć odporność na awarie, to musi być zautomatyzowana, żeby nie trzeba było zbyt wielu adminów i zarządzania
    - easy to use - programowanie systemów rozproszonych jest trudne, więc dla uproszczenia (i umożliwienia automatyzacji) ograniczamy się tylko do funkcji map() i reduce()

Model programowania MapReduce:

- przetwarzamy dane w postaci rekordów klucz-wartość
- map() przyjmuje parę klucz-wartość i produkuje dla niej listę 1 lub więcej wartości
- reduce() przyjmuje parę (klucz, lista wartości) i zwraca listę wyjściowych rekordów klucz-wartość
- po map trzeba posortować dane po kluczach i przesłać na odpowiednie węzły, ta operacja shuffle & sort może być tą najbardziej kosztowną ze względu na narzut na komunikację
- optymalizacja - funkcja combiner():
    - lokalny reduce, żeby zagregować dane dla powtarzających się kluczy przed shuffle
    - minimalizuje narzut na przesył danych
    - da się zrobić dla operacji łącznych, np. sum, count, max

Przepływ danych w MapReduce:

- input i finalny output są przechowywane w rozproszonym systemie plików, np. GFS albo HDFS
- scheduler uwzględnia lokalność danych, starając się przydzielać zadania węzłom mającym blisko dane do nich
- wyniki pośrednie workerzy przechowują u siebie, we własnym systemie plików
- output jest często wejściem do kolejnego zadania MapReduce, jest przy tym typowo bardzo duży

Koordynacja przetwarzania:

- zadanie mastera
- trzyma się listę zadań, które mogą być idle, in-progress albo completed
- zadania idle czekają na workera, gdy się da to są schedule'owane i wykonywane
- kiedy kończy się zadanie map, to wysyła do mastera lokalizacje i rozmiary plików pośrednich, master wysyła te informacje do reducerów, które potrzebują tych danych
- master pinguje regularnie workerów, żeby wykryć failure'y

Liczba zadań:

- oznaczenia: M zadań map, R zadań reduce
- typowo robi się znacznie więcej M i R niż mamy dostępnych węzłów w klastrze, bo rozmiar zadań jest w praktyce nierówny (np. nierównomierny rozkład kluczy), a chcemy obciążyć wszystkie węzły
- ulepsza to load balancing i przyspiesza recovery po błędzie (utracone mniej obliczonych danych, mniej do przeliczenia)
- rule of thumb: tyle M ile mamy chunków danych w rozproszonym systemie plików (DFS)
- zwykle R < M, bo na wyjściu mamy R plików

Funkcja partycjonująca:

- na wejście dostajemy plik, dzielimy go na M ciągłych kawałków i rozsyłamy jako zadania map
- żeby zmapować pośrednie wyniki na reducerów, definiujemy funkcję partycjonującą (partition function), działającą zasadniczo jak funkcja haszująca
- domyślnie jest to najprostsze możliwe `hash(key)%R`, czasem opłaca się nadpisać, np. kiedy chcemy, żeby dane o określonym kluczu trafiły na jednego reducera, bo będą tam razem przetwarzane (unika dodatkowego przesyłania danych)
- w obrębie danej partycji (i w zadaniach map, i reduce) pary klucz-wartość są przetwarzane w kolejności rosnącej kluczy - **gwarancja kolejności (ordering guarantee)**
- gwarancja kolejności pozwala łatwo generować posortowane wyniki, np. trywialnie zaimplementować rozproszone sortowanie

Failures:

- kiedy padnie nam map worker, to:
    - zadania map, które ma completed albo in-progress są resetowane do idle
    - gotowe zadania trzeba zrobić od nowa, bo dane są przechowywane lokalnie - gdy tracimy workera, to też policzone dane na nim
    - reduce workerzy dostają informację, kiedy zadanie jest na nowo schedule'owane na innego workera, bo muszą wiedzieć, skąd wziąć dalsze dane
- kiedy padnie nam reduce worker, to wystarczy wrócić z jego zadaniami in-progress do idle, bo gotowe wyniki są już zapisane na wynikowym systemie plików
- jeżeli padnie master, to jest źle - trzeba zakończyć całe zadanie MapReduce porażką i poinformować klienta, ew. można zrobić replikowanego mastera, żeby tego uniknąć
- gdy padnie nam pojedynczy węzeł obliczeniowy (1 lub więcej workerów map i/lub reduce), to jego zadania puszczamy od nowa na innych, a wszystkie zadania map z niego robimy od nowa (bo straciliśmy dane)

Stragglers:

- straggler to task, który idzie nam powoli, np. odpalił się duży task na słabym węźle, albo węzeł ma akurat bardzo obciążony dysk
- puszczamy drugą instancję na innym węźle (lub węzłach), bierzemy wynik tego który wykona się szybciej i ubijamy pozostałe
- bardzo ważne na dużych klastrach, bo tam może być dużo takich przypadków

Przykładowe zadania MapReduce:

- wyszukiwanie:
    - wejście: rekordy (numer linii, tekst linii)
    - wyjście: linie spełniający dany pattern (np. regex)
    - map sprawdza czy linia spełnia pattern, jeśli tak to ją daje na wyjście, jak nie to nie
    - nie ma reducera (funkcja identyczności) - typ zadania map-only
- sortowanie:
    - wejście: rekordy (klucz, wartość)
    - wyjście: te same rekordy, ale posortowane po kluczu
    - map i reduce są funkcjami identyczności
    - wykorzystuje gwarancję kolejności - wybieramy taką funkcję partycjonującą `p(k)`, że: `k1 < k2 => p(k1) < p(k2)`
- tworzenie odwróconego indeksu:
    - wejście: rekordy (nazwa pliku, tekst)
    - wyjście: lista par (słowo, lista nazw plików zawierających to słowo)
    - map tnie tekst na słowa i zawraca rekordy (słowo, nazwa pliku)
    - combine robi operację unique
    - reduce zwraca (słowo, sort(lista nazw plików))

- najpopularniejsze słowa:
    - wejście: rekordy (nazwa pliku, tekst)
    - wyjście: 100 słów występujących w największej liczbie plików
    - można zrobić 2 jobami MapReduce:
        - job 1: stworzenie odwróconego indeksu, mamy listę (słowo, lista plików ze słowem)
        - job 2: mapowanie na (liczba wystąpień, słowo) i sortowanie
    - alternatywnie pierwszy job może robić rekordy (word, 1), wtedy wystarczy zsumować
    - można zrobić rozwiązanie przybliżone próbkując rozkłady słów w dokumentach
- całkowanie numeryczne:
    - wejście: lista przedziałów (start, end) do całkowania
    - wyjście: wartość całki na całym przedziale
    - ustalamy jeden wspólny klucz dla wyników pośrednich - dzięki temu każdy map zwróci wynik do jednej puli, a reduce wykona się na wszystkich wynikach
    - każde zadanie map sumuje pewien przedział (start, end)
    - reduce sumuje wszystkie wartości
- odwracanie grafu linków:
    - wejście: rekordy (nazwa strony, treść strony)
    - map zwraca rekordy (target, source) z linków znalezionych na stronie
    - reduce łączy wszystkie URLe znalezione dla danej strony docelowej i zwraca rekordy (target, list(source))

Hadoop:
- implementacja open-source narzędzi big data, które zrobił Google (GFS, MapReduce)
- Hadoop Distributed File System (HDFS):
    - implementacja open source GFSa
    - zapewnia pojedynczy namespace dla plików w całym klastrze
    - replikuje dane 3x dla fault tolerance, zwykle nie używa macierzy RAID
- framework MapReduce - napisany w Javie, poza tym identyczny z oryginalnym (tamten był w C++)
- typowo używa topologii fat tree - buduje drzewiastą hierarchię switchy, na dole liśćmi są node'y z danymi, mają cienkie łącza do switchy, a dalej są mocne łącza w górę hierarchii (bo będą obsługiwać dane z wielu node'ów)

Hadoop Distributed File System (HDFS):
- pliki dzielone na bloki po 128 MB
- bloki są replikowane, typowo 3x (na 3 różnych node'ach)
- master node (namenode) przechowuje metadane, np. nazwy plików, ich lokacje
- zoptymalizowany pod duże pliki i odczyty sekwencyjne dużych części
- do plików można tylko dopisywać (append-only)

Inne narzędzia ekosystemu MapReduce:

- Apache Pig - język skryptowy do tworzenia i łączenia ze sobą jobów MapReduce, trochę podobny do SQLa
- Apache Hive - baza relacyjna zbudowana na Hadoopie, wspiera pełny syntax SQL i rozszerzenia, sporo dodatków względem tradycyjnych baz (np. złożone typy danych)
- interaktywny MapReduce:
    - HaLoop, Twister
    - pętle zadań MapReduce, partycjonowanie danych utrzymywane między iteracjami
    - możliwość trzymania danych w pamięci (Twister)
- Incoop:
    - MapReduce dla obliczeń inkrementalnych
    - przelicza tylko deltę i aktualizuje poprzednie wyniki
- DryadLINQ:
    - rozszerzenie MapReduce z dodatkowymi operacjami
    - operatory bulkowe, np. map, groupByKey, join

Problemy z MapReduce:

- nie nadaje się do interaktywnych i krótkich analiz
- ciężko implementować algorytmy iteracyjne, np. ML

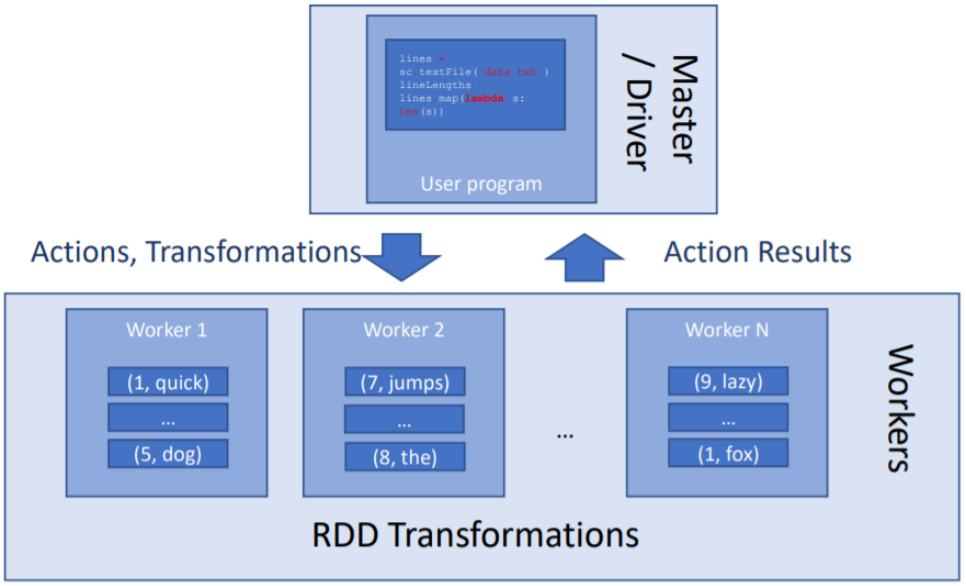
Introduction to Spark

RDD (Resilient Distributed Dataframe):

- podstawowa abstrakcja danych, na której operuje Spark (konkretnie Spark Core)
- niemutowalny, rozproszony zbiór danych
- podstawowe cechy RDD:
    - niemutowalność
    - silnie typowane, np. `RDD[Long]`, `RDD[(Int, String)]`
    - tworzone tylko przez deterministyczne operacje (na RDD lub trwałym źródle danych)
    - utrzymują swój rodowód (lineage) - informacje, jak dany RDD został stworzony, tzn. skąd wczytano bazowe dane i jakie transformacje na nim wykonano
- własności te dają trwałość (resilience) - można zawsze w przypadku błędu (np. padnięcia węzła z danym RDD) go odtworzyć
- uwaga - RDD nie są replikowane!
- optymalizacje:
    - przetwarzanie danych w pamięci
    - leniwa ewaluacja - wynik jest obliczany dopiero wtedy, kiedy trzeba
    - trzyma w pamięci te RDD, które są potrzebne i na ile pozwala pamięć, reszta jest zapisywana na dysk
- 2 rodzaje operacji:
    - transformacje - przekształcenia, które mogą być wykonane leniwie, typowe operacje map/reduce
    - akcje - takie operacje, że trzeba już wykonać obliczenie i zwrócić wynik z workerów do drivera (mastera), np. policzenie liczby wierszy, wypisanie czegoś
- możliwa ręczna kontrola nad:
    - persystencją - można zapisać RDD (np. gdy wiemy, że trzeba go będzie dużo razy używać), w pamięci lub na dysku
    - partycjonowaniem - rozdzielenie RDD między węzły, można wyznaczyć klucz na podstawie którego jest to wykonywane oraz liczbę partycji
- brak optymalizacji zapytań - operacje na RDD są wykonywane as-is, trzeba ręcznie zadbać o efektywność przetwarzania (dlatego lepiej użyć Spark SQL, które ma optymalizator Catalyst)

Architektura Spark Core:

- master / driver - główny węzeł (zwykle 1), z nim się komunikujemy, on wystawia UI, nie wykonuje żadnych obliczeń, używamy dowolnego języka na który pozwala Spark (Python, Java, Scala, R)
- workers - węzły obliczeniowe, typowo bardzo dużo, wykonują obliczenia MapReduce, wysyłają wyniki do mastera, wykonują kod w Scali



Spark Dataset:

- abstrakcja nad RDD, zbiór obiektów które mogą być przetwarzane przez Sparka (równolegle, w rozproszeniu) za pomocą operacji MapReduce
- ma zalety RDD (silne typowanie, funkcje lambda), a przy tym wykorzystuje silnik optymalizacyjny ze SparkSQL
- tylko w Javie i Scali, bo obiekty Datasetu mogą być tworzone z obiektów JVMowych
- pozwalają zastosować operacje funkcyjne implementowane współbieżnie w Sparku na zwykłych obiektach, np. map, flatMap, filter

Spark DataFrame:

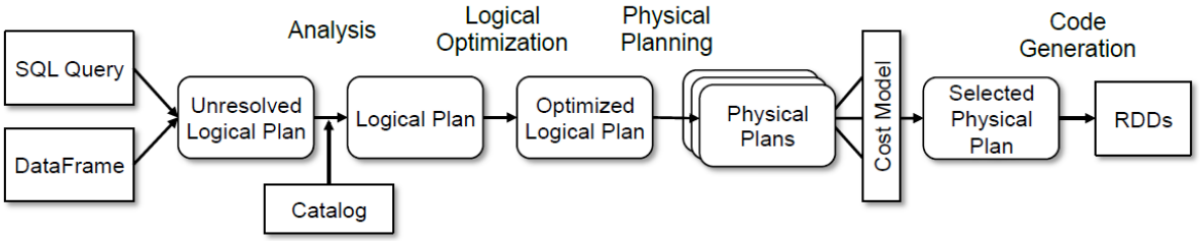
- wysokopoziomowa abstrakcja nad RDD, część Spark SQL, wzorowana na ramkach danych z R / Pandasa
- odpowiada tabeli w bazie danych, ma nazwane kolumny o określonych typach
- mogą być tworzone z wielu źródeł danych:
    - pliki, głównie strukturalne (np. CSV, Parquet), ale nie tylko (np. JSON)
    - zewnętrzne bazy danych, tabele Apache Hive
    - RDD
- dostępne w Pythonie, Javie, Scali i R
- operacje na DataFrame'ach są optymalizowane przez Catalyst (wysokopoziomowy optymalizator zapytań, jak w bazach danych), a potem kompilowane do kodu bajtowego JVMa
- wykorzystuje in-memory caching z kolumnowym formatem danych, bo jest to wydajniejsze od cache'owania obiektów JVMowych
- pozwala używać User Defined Functions (UDFs) - własnych funkcji, napisanych w oryginalnym języku (np. Python)

Przewagi Spark SQL (DataFrame) nad czystym SQLem:

- możliwość rozbicia złożonej logiki na fragmenty kodu w wybranym języku, czytelniejszym od SQLa, np. Python
- możliwość wykonywania różnych przekształceń po kolei na DataFrame'ie, budując logiczny plan wykonania (który jest w całości optymalizowany pod spodem)
- możliwość użycia struktur programistycznych do oddania logiki, np. if/else, pętle
- możliwość wykorzystywania (np. printowania, logowania) wyników pośrednich, np. do debugowania
- gorliwe wartościowanie planu logicznego i leniwe wykonanie, co pozwala sprawdzić kod (np. sprawdzić typy danych, nazwy kolumn) przed długotrwałym wykonaniem

Optymalizator Catalyst:

- optymalizator kosztowy dla Sparka
- etapy:

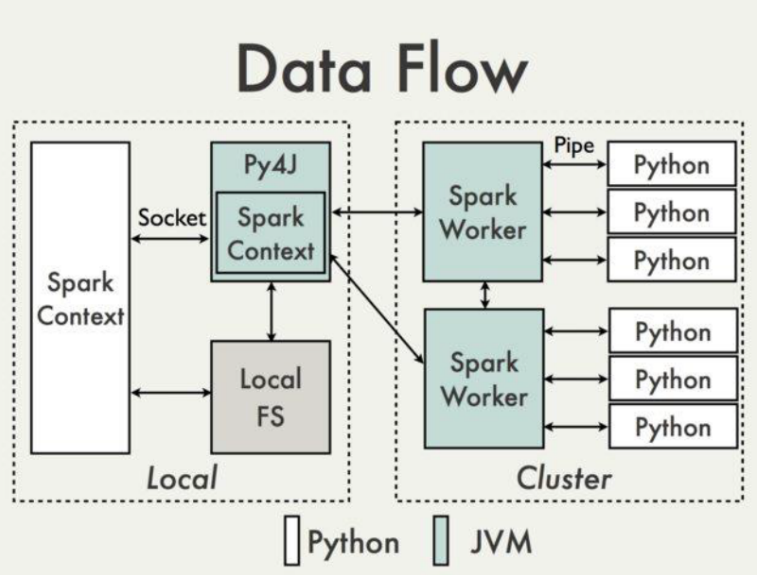


  - analiza - bierzemy dane oraz plan logiczny unresolved, czyli z atrybutami bez określonych typów i ograniczeń, wypełniamy atrybuty i typy analizując dane
  - optymalizacja logiczna - wszelka optymalizacja jak w kompilatorach, np. zwijanie stałych, predicate pushdown
  - fizyczne planowanie - optymalizator kosztowy jak w SQLu, czyli generowanie planów, szacowanie kosztów etc.
  - generacja kodu - kompilacja zapytań do kodu bajtowego JVMa, przy czym kod może być też generowany w runtime'ie
- napisany w Scali, wykorzystuje pattern matching
- dodatkowa optymalizacja w runtime'ie, mechanizm adaptive query execution, dodana w Sparku 3.0

Formaty wejścia:

- Spark przyjmuje bardzo różne formaty wejściowe
- formaty strukturalne mogą zawierać metadane o typach (np. Apache Parquet) albo nie (np. CSV), w razie potrzeby typy są inferowane z danych, ogółem wczytywanie jest proste
- formaty niestrukturalne (np. JSON) są bardziej złożone, mogą zawierać zagnieżdżone pola, Spark czyta takie dane i inferuje typy i złożenia
- wejściem może być też nieplikowe źródło danych, np. baza relacyjna albo NoSQL (np. ElasticSearch), w razie potrzeby Spark też przeprowadzi inferencję typów
- wiele źródeł umożliwia selektywny odczyt, czyli tylko ściągnięcie wybranych danych (np. JSONowe query w ES); Spark wykorzystuje to realizując **predicate pushdown**, czyli tłumaczy zapytanie w DataFrame'ie (lub SQLu) na język wejścia, w ten sposób można np. odpytywać ElasticSearch'a za pomocą SQLa

Budowa PySparka:



- driver to program w Pythonie
- używa socketa, żeby przesłać dane do Py4J do tłumaczenia operacji (transformacji i akcji) na obiekty JVMowe typu PythonRDD, które potem są wysyłane do workerów
- dane są przechowywane jako obiekty pickle - pozwala uniknąć problemów z typami, ale powoduje narzuty na serializację
- wszelkie funkcje (np. lambdy, UDF, funkcje z map/flatMap) są pakowane z użyciem cloudpickle i wysyłane do workerów Pythonowych
- workerzy mają proces Pythona do wykonywania części operacji (np. UDF), ale poza tym dalej używają Scali

Data streaming

Powody wykorzystania strumieniowania danych:

- szybkie, responsywne systemy dla przychodzących zdarzeń
- zbyt duże zbiory danych żeby je przechowywać, chcemy tylko wyników przetworzenia
- dane pochodzące z różnych źródeł, wysyłane przez nie jako strumienie danych
- liczne zastosowania, np. wykrywanie oszustw, analiza kliknięć, monitoring online

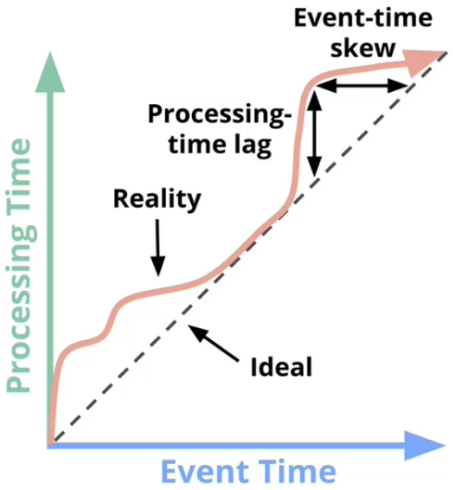
Event time - moment, w którym event faktycznie nastąpił.

Processing time - moment, w którym event został zaobserwowany przez system

Processing time lag - opóźnienie z perspektywy czasu przetwarzania między momentem wystąpienia i przetworzenia eventu.

Event time skew - opóźnienie z perspektywy czasu eventu między momentem wystąpienia i przetworzenia eventu.

Processing time lag = event time skew, różnią się tylko punktem widzenia, ale ilość czasu jest taka sama.



Dane strumieniowe:

- nieograniczone (unbounded) - zasadniczo o nieograniczonym rozmiarze
- dane w ruchu (data in motion) - obserwujemy dane element po elemencie, zbiór danych zmieniający się z czasem, w przeciwieństwie do klasycznego podejścia tabelarycznego (data at rest, pogląd na całe dane w punkcie w czasie)
- nieuporządkowane według czasu eventu - dane o różnych znacznikach czasowych mogą przyjść w dowolnej kolejności
- różnorodne event time skew - dane będą przychodzić z różnym opóźnieniem, nie można założyć żadnej stałej wartości opóźnienia
- typowo każdy event ma znacznik event time, żeby móc analizować kolejność zdarzeń

Podejścia do przetwarzania danych strumieniowych:

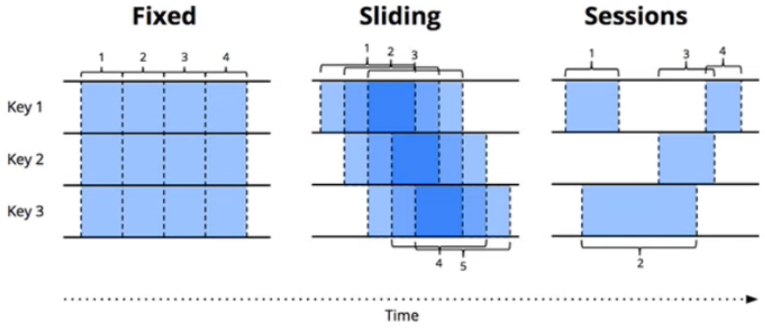
- batch processing:
    - grupujemy zdarzenia w minibatche, a potem przetwarzamy zwykłe tabelki z danymi
    - pozwala wykorzystywać systemy do przetwarzania wsadowego do przy danych streamingowych
    - np. Spark Structured Streaming w trybie micro-batch processing
- stream processing:
    - przetwarzanie faktycznego strumienia eventów, za pomocą systemu zbudowanego konkretnie do streamingu
    - np. Apache Flink, Spark Structured Streaming w trybie continous streaming

Rodzaje stream processingu:

- time-agnostic:
    - ignorujemy całkowicie aspekt czasowy, przetwarzanie eventów jest w dowolnej kolejności
    - nadaje się do problemów gdzie czas eventu tak naprawdę nie ma znaczenia, a logika zależy od samych wartości w danych
    - wymaga od systemu tylko przesyłania strumienia danych, żadna bardziej złożona logika nie jest potrzebna, więc systemy batchowe się do tego nieźle nadają
    - np. filtrowanie, inner join, trening klasycznych algorytmów ML online (out-of-core)
- approximation:
    - nie gwarantujemy respektowania czasu eventu, ale przybliżamy odpowiednią kolejność
    - złożone algorytmy, w gruncie rzeczy też są time-agnostic, przydatne gdy mamy dane nie mieszczące się w pamięci
    - np. top-N, streaming k-means
- windowing - dzielimy dane na okna czasowe, które mogą być tworzone ze względu na czas przetwarzania albo ze względu na czas eventu

Strategie windowingu:

- fixed - okna stałej długości, nienachodzące na siebie
- sliding - okna przesuwne stałej długości i o stałym okresie (period), nachodzące na siebie gdy period < length, np. grupuj co minutę dane z ostatnich 5 minut
- sessions - okna dynamicznej długości, eventy są grupowane według okresów aktywności/nieaktywności, np. sesje użytkowników (stąd nazwa)



Przetwarzanie batchowe w stałych oknach:

- windowing, fixed windows, batch processing
- jeden batch to jedno okno czasowe, więc batche reprezentują ten sam okres czasu, ale są różnej wielkości (różna liczba eventów)
- bardzo proste podejście, ale generuje szereg problemów:
    - opóźnione eventy przez podziały / opóźnienia w sieci
    - dane z batcha trzeba zebrać w centralnej lokalizacji przed przetwarzaniem
- przetwarzanie danych trzeba opóźnić tak, żeby zebrać wszystkie dane z okna

Przetwarzanie batchowe w oknach sesyjnych:

- mamy okna czasowe, przetwarzamy sesja każdego użytkownika osobno
- problem: typowo sesje będą w różnych oknach czasowych, albo pojedyncza sesja będzie podzielona między wiele okien czasowych
- można zwiększać rozmiary batchy (okien czasowych), ale zwiększa to opóźnienie
- sesje podzielone między okna czasowe trzeba łączyć (stitching), co zwiększa złożoność logiki

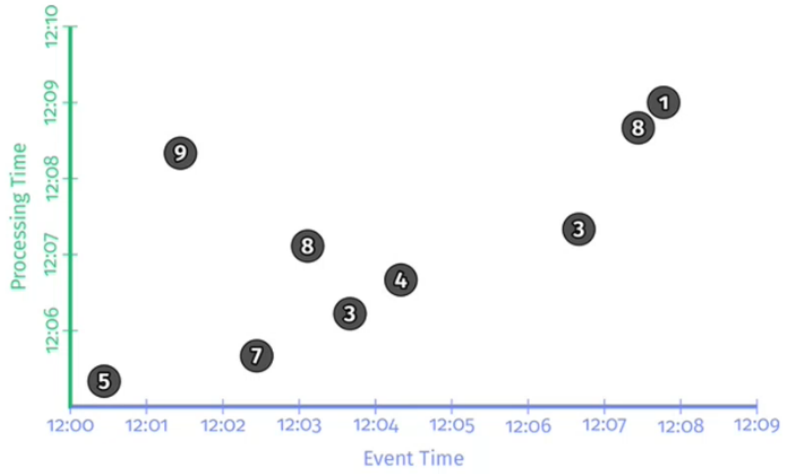
Streaming - windowing by processing time:

- dzielimy dane na okna czasowe według czasu przetwarzania
- buforujemy dane przez określony czas i przesyłamy je jako okno do przetworzenia
- zalety:
    - prostota
    - brak obsługi zdarzeń opóźnionych - de facto wszystko jest opóźnione, ale to ignorujemy, bo nie przejmujemy się czasem zdarzenia
    - dobre dla monitoringu, np. śledzenia liczby requestów na sekundę, bo obserwujemy requesty kiedy do nas przychodzą
- wady:
    - system zależy od czasu przetworzenia, nie faktycznego czasu eventu, więc jeżeli jest on ważny, to słabo działa
    - wszystkie dane przetwarzane ze znacznym opóźnieniem względem faktycznego czasu eventu
    - różne źródła mają różne opóźnienia (event time skew)

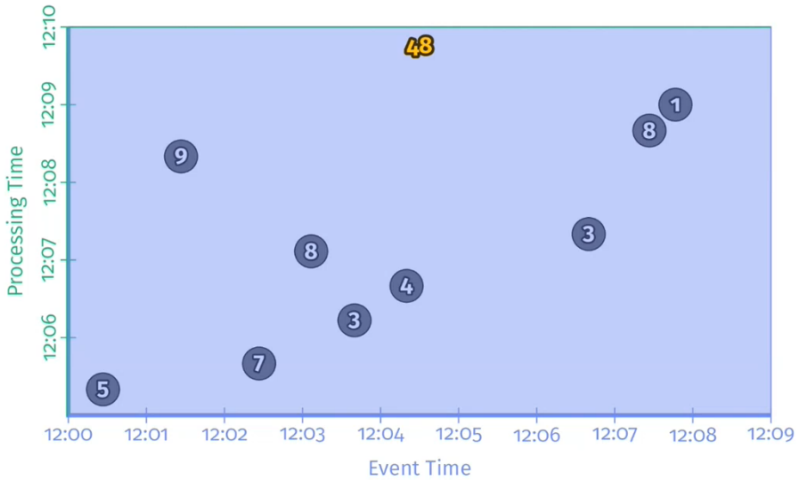
Streaming - windowing by event time:

- staramy się dzielić dane na okna czasowe według czasu eventu
- batche danych odpowiadają czasom, w których faktycznie te wydarzenia nastąpiły
- wymaga operacji "temporal shuffle", czyli posortowania po czasie
- zaleta:
    - faktyczne przetwarzanie zgodnie z czasem eventu
    - pozwala łatwo przetwarzać sesje, bo mamy faktyczne całe, spójne okna czasowe z sesjami
- wady:
    - wymaga więcej buforowania, żeby dostać pełne dane z okna, włącznie z opóźnionymi
    - problem pełności okna (window completeness) - nigdy nie mamy gwarancji, że wszystkie dane z okna już przybyły

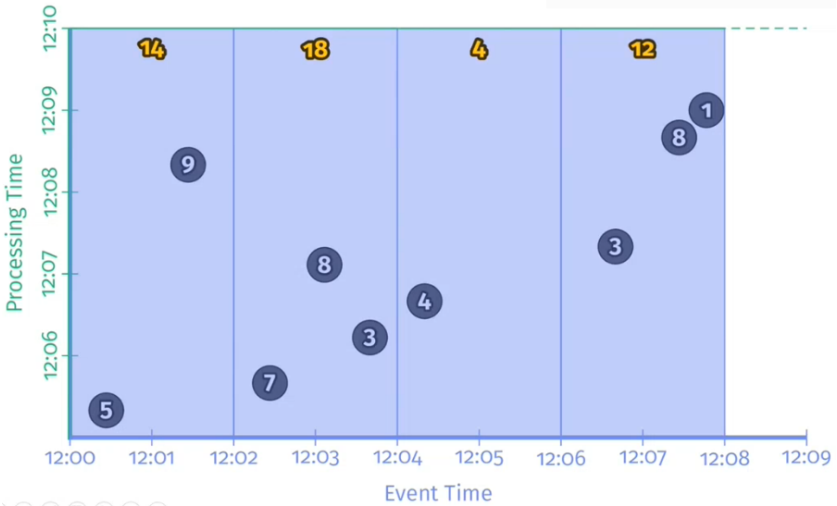
Przykład ilustrujący różnice między powyższymi - liczenie sumy punktów.



Wersja batchowa czeka na wszystkie dane, liczy i zwraca wynik. Powoduje to, że processing time jest większy od największego event time, czyli nie mamy żadnych wyników częściowych w międzyczasie. Nie zadziała też, jeżeli dane mogłyby przychodzić teoretycznie bez końca.



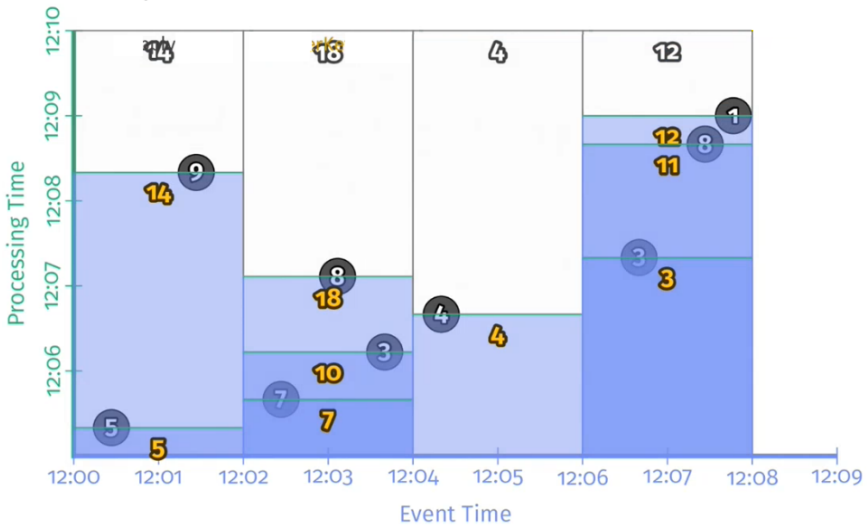
Wersja ze stałymi oknami (fixed windows) zwraca wyniki częściowe np. co 2 minuty. Mamy i wyniki częściowe dla okien, i możemy też akumulować i zwrócić całkowitą sumę.



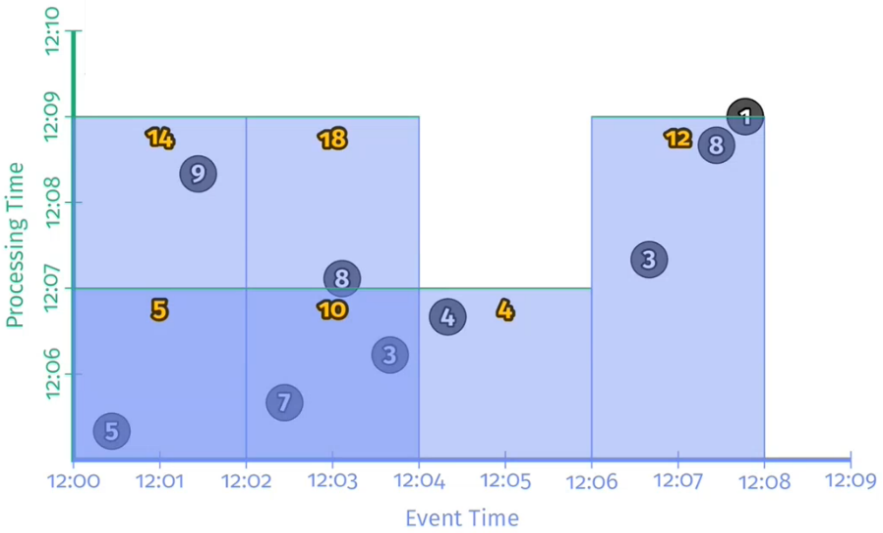
Triggery - wyznaczają, kiedy (w sensie processing time) wyniki są obliczane i zwracane, mogą się odpalać wielokrotnie w ramach okna. Konkretny output w ramach okna to **pane**.

Repeated update triggers:

- regularnie generujemy zaktualizowane pane'y dla danego okna czasowego
- aktualizacje mogą być cykliczne (co jakiś czas), albo kiedy przychodzą nowe rekordy w ramach okna (co 1 rekord albo co batch N rekordów)
- trzeba dobrać częstotliwość aktualizacji - tradeoff opóźnienie vs koszt
- **every new record:**

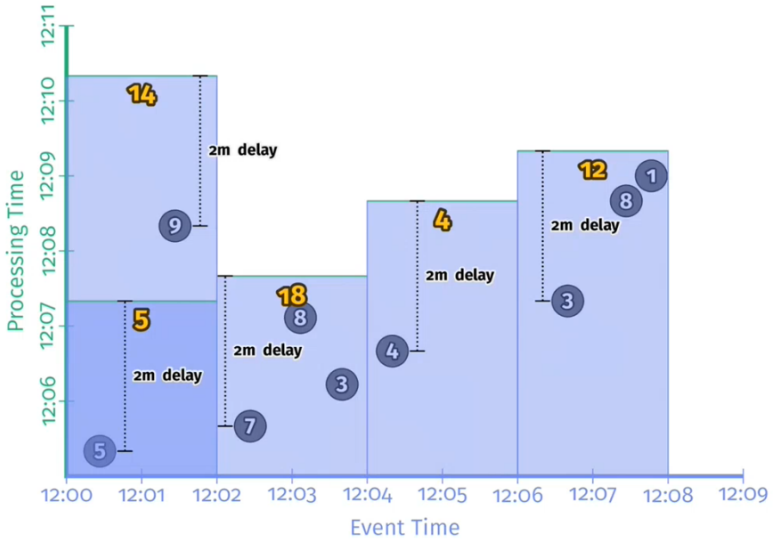


  - zwracamy wynik częściowy dla każdego nowego rekordu w oknie czasowym
  - zbieżność do wartości poprawnej wraz ze zbliżaniem się do końca okna
  - zaleta: maksymalna aktualność danych częściowych
  - wada: zbytnia granularność aktualizacji, szczególnie gdy jest dużo zdarzeń
- **aligned delay-based:**



  - regularnie, co określony czas (liczony w czasie przetwarzania) zwracamy wyniki częściowe dla wszystkich okien czasowych
  - zaleta: dostęp do wyników częściowych, określona częstotliwość
  - wada: dużo aktualizacji naraz (dla wielu okien czasowych), skokowe obciążenie systemu (bursty workloads)

- **unaligned delay-based:**

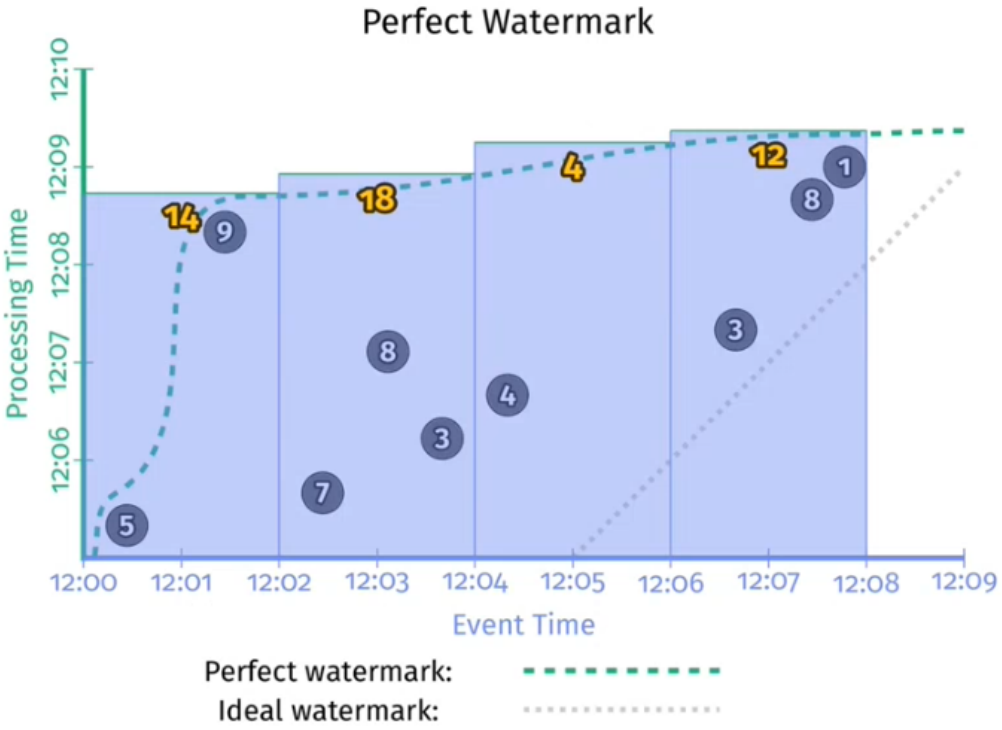


  - częstotliwość jest stała, ale jest osobno liczona dla każdego okna
  - liczymy opóźnienie (delay) dla każdego okna osobno; w każdym oknie od pierwszego eventu czekamy określony delay i zwracamy wynik częściowy
  - zaleta: równomierne rozłożenie obciążenia w porównaniu do aligned delay-based, ilość wyników częściowych zależna od obciążenia konkretnych okien czasowych

Completeness triggers:

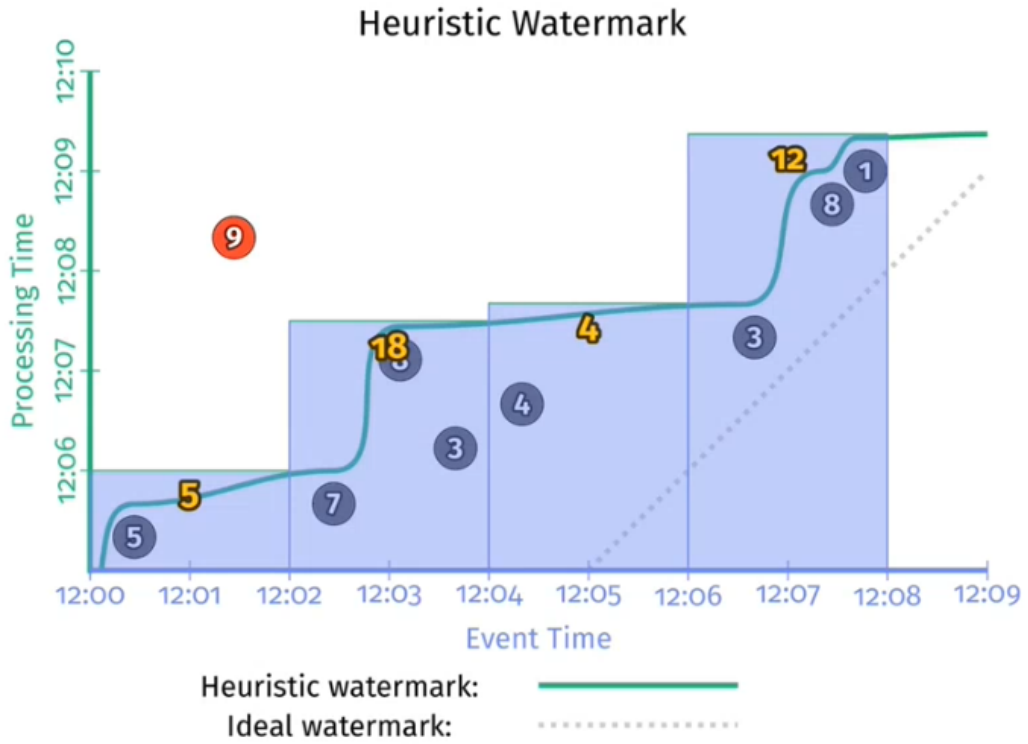
- liczymy pane dla okna dopiero wtedy, kiedy szacujemy, że okno jest kompletne w określonej ilości (np. w całości)
- podobne do batch processingu, bo przetwarzamy dopiero pełne dane (czy przynajmniej uważane za pełne), ale granularność jest na poziomie okien, nie całości danych
- korzysta ze **znaków wodnych (watermarks):**
  - wyznaczają, kiedy okno jest pełne (complete)
  - mapują czas przetwarzania na czas zdarzenia, funkcja `F(P) -> E`
  - dla czasu przetwarzania (processing time) `P` wszystkie dane wejściowe o czasie zdarzenia (event time) `E` już zostały zaobserwowane, czyli nie będzie już zdarzeń wcześniejszych niż `E`
  - mogą mieć różny poziom dokładności, w zależności od tego, jak dużą gwarancję zapewniają
  - mają 2 rodzaje, w zależności od gwarancji i wiedzy o danych: dokładne i heurystyczne
- przydatne przy outer joinach strumieni danych - trzeba tam podjąć decyzję, czy emitować częściowy join (kiedy szacujemy, że dane już są), czy czekać na więcej danych

Dokładne znaki wodne (perfect watermarks):



- mamy perfekcyjną wiedzę o danych wejściowych
- zalety:
  - spełniają dokładnie funkcję
  - gwarantują brak danych opóźnionych
- wady:
  - mocno opóźniają przetworzenie danych, bo czekamy na arbitralnie opóźnione eventy - watermark lag
  - można ich użyć tylko gdy mamy zdarzenia regularne i z góry wiemy ilu eventów się spodziewamy w oknie czasowym

Heurystyczne znaki wodne (heuristic watermarks):



- nie mamy dokładnej wiedzy o danych
- przybliżamy completeness danych, wykorzystując wiedzę o danych, np. partycje, szybkość przyrostu plików
- zalety:
    - można użyć dla dowolnych zdarzeń, szersze spektrum zastosowań niż dla znaków dokładnych
    - szybsze
- wady:
    - opóźnione dane są możliwe
    - mogą być zbyt szybkie - eventy bardzo opóźnione mogą nie zostać przetworzone, mamy niedokładne dane na wyjściu
    - trzeba dobrać czas czekania, wymaga odpowiedniej wiedzy

Early/on-time/late triggers:

- połączenie triggerów typu repeated update trigger oraz completeness trigger
- zasadniczo po prostu watermark trigger, ale dodatkowo wspiera repeated update triggering przed i po triggerze ze znaku wodnego
- 3 rodzaje wyników (panes):
    - early panes:
        - 0 lub więcej
        - emitowane periodycznie za pomocą repeated update trigger, zanim watermark dojdzie do końca okna
        - emituje wyniki częściowe, pozwala obserwować ewolucję wyniku w ramach okna
    - on-time panes:
        - co najwyżej 1
        - emituje go watermark trigger, kiedy dojdzie do końca okna (dokładnego lub szacowanego, w zależności od watermarku)
    - late panes:
        - 0 lub więcej
        - emitowane przy przyjściu nowego opóźnionego rekordu, po watermarku
        - tylko dla heurystycznych watermarków, bo dla perfekcyjnych nigdy nie ma danych opóźnionych
- pozwala połączyć zalety obu podejść, emitując i wyniki częściowe, i sumaryczny dla okna, i uwzględniając dane opóźnione (jeżeli korzystamy z heurystycznego znaku wodnego)

Dozwolone opóźnienia:

- problem w heurystycznych znakach wodnych i triggerach early/on-time/late
- jak długo czekać na dane opóźnione?
- **lateness horizon** - górne ograniczenie na czas czekania na dane - późniejsze są ignorowane i usuwane; typowo podawany w domenie event time
- trzeba utrzymywać stan okna, dopóki watermark nie przekroczy lateness horizon dla końca okna

Akumulacja wyników dla okna:

- kiedy produkujemy wiele wyników częściowych dla okna (panes), trzeba zdecydować, co z poprzednimi
- discarding:
    - usuwamy poprzedni pane
    - trzeba zagregować wszystkie pane'y dla okna, żeby dostać pełny wynik
    - przydatne, gdy klienci (konsumenci outputu strumienia) chcą dokonywać własnych agregacji na częściowych wynikach
- accumulating:
    - nowe wyniki nadpisują poprzednie
    - znajdujemy deltę względem poprzedniego panelu i zwracamy zakumulowany wynik
    - ostatni pane to pełny poprawny wynik dla okna

- accumulating & retracting:
    - nowe wyniki nadpisują poprzednie, ale dodatkowo zwracamy korektę poprzednich wyników, czyli deltę, jaką teraz uwzględniliśmy
    - gwarantuje, że i ostatni pane, i agregacja wszystkich pane'ów dla okna daje poprawny pełny wynik

Streaming architectures and systems

Architektura Lambda:

- uwaga: nie ma nic wspólnego z AWS Lambda! (chociaż można jej użyć do implementacji live processingu)
- architektura systemu łączącego stream i batch processing
- idea:
    - 2 równoległe flow danych: speed layer (streaming), batch layer (batch processing), po których jest serving layer
    - streaming daje szybkie wyniki live, batch processing daje wyniki rzeczy cięższych do obliczenia, ale mniej aktualne
    - serving layer przygotowuje widoki z wyników warstw speed i batch, które już można szybko query'ować
    - utrzymujemy master dataset, czyli zbiór wszystkich danych historycznych (jego przetwarza batch processing)
- streaming daje niekoniecznie zawsze dokładne wyniki, ale działa szybko, do tego wystarczy, żeby obliczał tylko deltę od ostatniego wyniku batch processingu
- batch processing daje w pełni dokładne wyniki, pozwala na głębsze analizy, ale wymaga dużo czasu i jest uruchamiany np. raz dziennie
- użycie batch processingu pozwala na zmiany w architekturze aplikacji, np. dodanie nowych pól w bazie danych
- serving layer:
    - bierze dane przetworzone przez streaming (delta od ostatniego batcha) i batch processing (pełne przetworzone dane), łączy je i udostępnia widoki, które mogą być łatwo i efektywnie odpytywane przez aplikacje
    - aktualizowany typowo tak często, jak często batch processing zapewnia nowe dane (co kilka godzin / raz dziennie)
    - wymaga tylko batch write'ów (bo zawsze wrzucamy duże aktualizacje, wynik batch processingu) i random read; jako że to random write jest problematyczny, to w tym wypadku write można napisać bardzo efektywnie
    - przykłady: Voldemort, HBase, Druid
- zalety:
    - robustness, fault tolerance - w razie jakiegokolwiek grubszego problemu mamy master dataset i można wszystko odtworzyć
    - niskie opóźnienia (latency) - dzięki warstwie streamingowej
    - elastyczność i skalowalność - wszystkie warstwy można skalować niezależnie
    - ogólność - można użyć do wielu aplikacji, dostosowując szczegóły techniczne poszczególnych elementów
- wady:
    - nieprecyzyjne dane - streaming przez zgubione / zduplikowane wiadomości lub błędy samego streamingu może mieć błędne dane
    - niekonsystencja pipeline'ów - ciężko idealnie zsynchronizować obie ścieżki i dostać z nich identyczne wyniki
    - złożoność i koszty - trzeba utrzymywać (i płacić za) wiele różnych technologii
    - nieprzewidywalność - ciężko stwierdzić, jak duże błędy zawierają dane streamingowe i jak dużo zmieni się po wykonaniu batch processingu

    - opóźnienie - precyzyjne wyniki batch processingu (i w ogóle rzeczy polegające na batch processingu) mają duże opóźnienie (rzędu godzin / całego dnia)

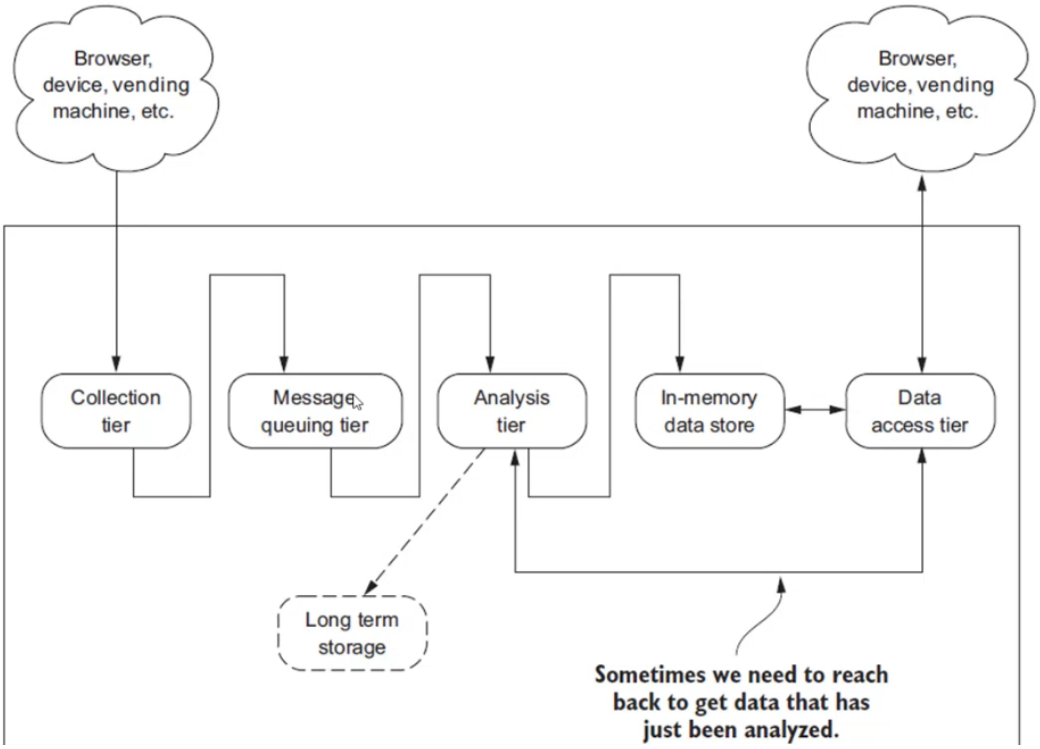
Architektura Kappa:

- alternatywa do Lambdy; mamy pojedynczy streaming pipeline, oparty o klaster Kafki
- założenie: architektura Lambda myli się w tym, że przetwarzanie streamingowe z natury dostarcza niedokładnych wyników i ma wady względem batch processingu; jeśli zrobimy dość dobry stream processing to będzie to działać dobrze
- założenie 2: Kafka to tak dobry i niezawodny system streamingowy, że dobry klaster Kafki da sobie radę sam
- rozwiązanie problemu zmiany struktury danych (np. dodania kolumny do bazy):
    - trzeba systemu z persystencją danych, historią i wieloma subskrybentami (Kafka spełnia te wymagania)
    - tworzymy drugą, identyczną instancję streaming jobu i przetwarzamy dane od początku
    - kiedy nowy job przetworzył dane do aktualnych włącznie, przełączamy się na niego i wyłączamy stary
    - wymaga zwiększenia równoległości dla nowego joba, żeby był w stanie nadgonić (przetworzyć dane historyczne szybciej, niż przychodzą nowe)
- zalety:
    - niskie opóźnienie, niezawodne i szybkie aktualizacje danych
    - pojedynczy pipeline, mniej utrzymania niż w Lambdzie
- wady:
    - brak batch processingu i narzędzi do niego, nie można zrobić łatwo przetwarzania dużych danych (np. do analityki)
    - brak szczególnych przewag wydajnościowych - głównie po prostu prostota

Lambda vs Kappa:

- Lambda:
    - dobrze działa w praktyce
    - daje niskie opóźnienie i wysoką przepustowość
    - możliwy batch processing
    - wysokie zużycie zasobów, zarówno ludzkich, jak i sprzętowych
- Kappa:
    - lżejsza i prostsza od Lambdy
    - usuwa większość wad Lambdy
    - mniej możliwości od Lambdy, brak batch processingu

Przykładowa architektura streamingowa:



- collection tier - samo zbieranie danych, tworzy eventy i wysyła na kolejkę
- message queuing tier - kolejka wiadomości do przetworzenia
- analysis tier - warstwa analityczna, zapytania ciągłe przetwarzające strumień wejściowy w strumień wyjściowy
- in-memory data store - dane przechowywane w pamięci dla szybkiego dostępu
- data access tier - dostęp do danych

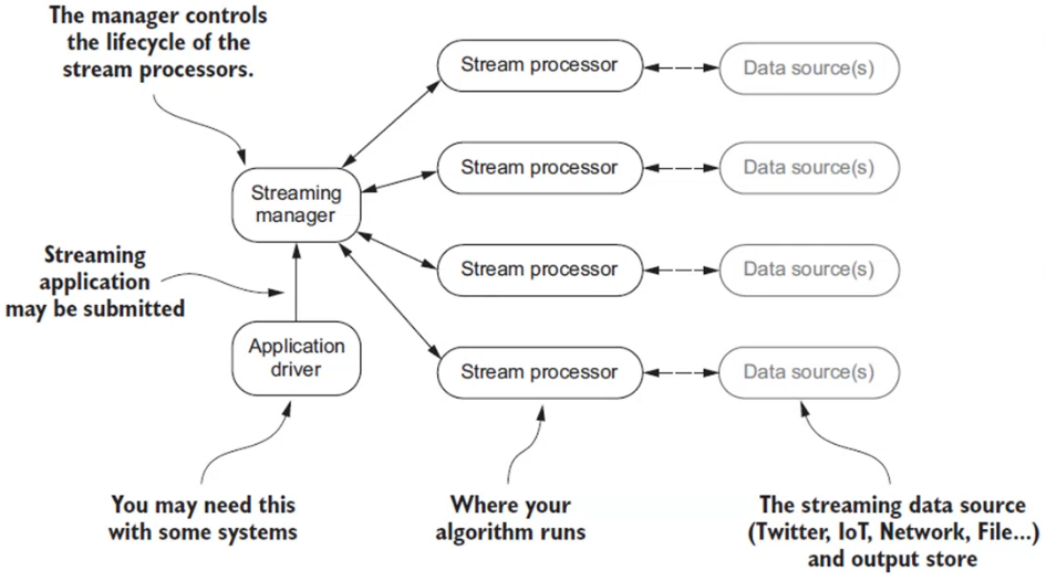
Warstwa kolejkowania danych:

- message brokering z producentami i konsumentami
- producenci - elementy collection tier, urządzenia wysyłające eventy do przetworzenia
- konsumenci - strumień wejściowy warstwy analitycznej
- implementuje różne rozwiązania systemów kolejkowych (message queuing system), na przykład:
    - backpressure - spowalnianie producentów, jeżeli konsumenci nie wyrabiają z przetwarzaniem danych
    - durable messaging, reliable message delivery - żeby przetworzyć wszystkie eventy

Kafka:

- system message brokeringu i strumieniowania danych, model producer/consumer
- utrzymuje uporządkowane kolekcje eventów zorganizowane w tematy (topics), czyli nazwane strumienie rekordów
- konsumenci utrzymują offset - numer ostatniego przetworzonego eventu z topicu
- możliwość reprocessingu - rozpoczęcia zadania z innym offsetem, np. przetworzenia topicu od początku
- różne bajery: fault tolerance, skalowalność, partycjonowanie etc.

Ogólna architektura streamingowa:



- application driver - to z nim się komunikujemy, definiuje program do wykonania, wysyła go do streaming managera i zbiera wyniki
- streaming manager - zarządza wykonaniem, rozsyła joby do workerów
- stream processors - workerzy, wykonują zadania
- data source(s) - źródła danych, traktowane jak strumienie (nawet gdy są statyczne, np. plik)
- data sink(s) - miejsce na output, może to być też driver (np. printowanie)

Spark Structured Streaming:

- część Sparka do stream processingu używająca Spark SQL API (i DataFrame'ów)
- wyższy poziom abstrakcji od starszego Spark Streaming używającego RDD API, które jest już niezalecane
- wspiera 2 modele przetwarzania danych:
    - micro-batch - opóźnienia rzędu 100 ms i silne gwarancje exactly-once fault tolerance
    - continous processing - opóźnienia rzędu 1 ms, ale słabsze gwarancje at least once fault tolerance
- typowy model przetwarzania danych jako tabeli - mamy nieskończoną tabelę (unbounded table), do której są dopisywane wiersze ze strumienia, a zapytania są identyczne z batchowymi
- co określony czas (trigger, np. co sekundę) dane są dopisywane do tabeli wejściowej, a wynik przeliczany
- to, co zapisujemy na wyjściu zależy od wybranego trybu, przy czym nie każdy tryb jest wspierany przez każde zapytanie, zależy od użytych elementów SQLa
- tryby:
    - append (domyślny) - tylko nowe wiersze, które pojawiły się od ostatniego triggera
    - complete - za każdym razem cała tabela wyjściowa, można wybrać tylko gdy mamy agregacje w zapytaniu
    - update - tylko wiersze zaktualizowane od ostatniego triggera, ale gdy nie ma agregacji w zapytaniu, to działa jak append

- przetwarzanie danych w oknach:
    - windowing on event time
    - okno przesuwne, może być różna długość length i interval (np. dane z ostatnich 10 minut co 5 minut)
    - event time jest dostępny jako kolumna w tabeli wejściowej
    - działa praktycznie jak zrobienie group by dla event time
- dane opóźnione:
    - prosty system znaków wodnych
    - dla zapytania narzuca się stały margines czasowy, liczony w event time
    - odrzucamy zbyt stare zdarzenia
    - tylko dla trybów update i append, bo dla trybu complete przechowujemy i dajemy na wejście całość danych
- joiny:
    - można robić join statyczny zbiór - strumień oraz strumień-strumień
    - wynik joina ze streamingiem jest generowany inkrementalnie, natomiast działa zasadniczo tak samo jak zwykły join na danych statycznych
    - są wspierane inner joiny i niektóre outer joiny
    - problemem z joinami stream-stream jest to, że w żadnym momencie nie znamy w pełni żadnej ze stron joina, więc dowolny wiersz z każdej strony może mieć złączenie dopiero z przyszłym wierszej z drugiej strony
    - wymaga:
        - buforowania przeszłych danych ze strumieni
        - watermarkingu do matchowania danych opóźnionych
        - definiowania limitów czasowych w warunku joina
    - w przypadku outer joinów wiersze bez matcha (z NULLami) będą wygenerowane wszystkie na koniec, przy limicie czasowym joina, bo dopiero wtedy mamy gwarancję, że nie znajdziemy pary dla wierszy

Apache Flink:

- framework do przetwarzania strumieniowego
- podejście "stream first" - wszystko jest strumieniem, tryb przetwarzania batchowego też robi streamy
- używa systemu aktorowego (Akka) do komunikacji rozproszonej
- dane opóźnione:
    - zaawansowany system znaków wodnych
    - watermarki są dodawane do strumienia eventów jako markery
    - w(x) oznacza, że wszystkie eventy do czasu x włącznie powinny już przybyć (liczone w event time)
    - watermark triggeruje obliczenie dla okien, które są za nim

Gwarancje konsystencji (semantics) w fault tolerance:

- at most once - wiadomość może zaginąć, ale nigdy nie zostanie przeczytana więcej niż raz
- at least once - wiadomość nigdy nie zaginie, ale może dotrzeć wielokrotnie
- exactly once - wiadomości są niezawodne, zawsze docierają dokładnie raz

Fault tolerance w Spark Streaming:

- w ogóle w Sparku są 2 możliwe rodzaje awarii:
    - drivera
    - workera - mało problematyczne, bo wystarczy uruchomić task znowu na innych workerach
- mocna gwarancja exactly once jest osiągana przez:
    - replayable sources
    - state fault tolerance
    - idempotent sinks
    - checkpointing
- fault tolerance drivera:
    - **offset** - pozycja danego konsumenta w ramach strumienia, można wyróżnić committed offset (pozycja, do której dane już zostały przetworzone i wysłane) i available offset (pozycja, do której można przetworzyć dane)
    - **write-ahead logs (WAL)** - tryb działania, w którym najpierw operacja jest zapisywana do persystentnego logu, a dopiero potem wykonywana
    - driver persystuje offsety poszczególnych workerów w logu z WAL, w przypadku błędu można odtworzyć operacje z loga
- replayable sources:
    - wielokrotnie odczytywalne źródła
    - wymaganie stawiane źródłom danych, żeby zapewnić fault tolerance samym źródłom
    - dla danych offsetów źródło ma generować te same dane
    - pozwala odtworzyć dane z lineage'u RDD, bo jego początkiem są właśnie dane
    - nie muszą być replayable w nieskończoność, ale co najmniej ostatnie minuty, żeby w przypadku awarii workera można było szybko odtworzyć RDD
- state fault tolerance:
    - niektóre operacje w kwerendach na strumieniach wymagają utrzymywania stanu, np. agregacje, joiny, usuwanie duplikatów
    - stan jest zapisywany (checkpointing) do persystentnego storage'u
    - schema stanu nie może się zmieniać między restartami
- idempotent sinks:
    - może się zdarzyć, że zapis nastąpi wielokrotnie
    - miejsce outputu (data sink) musi być idempotentny, czyli daje takim sam wynik przy stosowaniu wielokrotnie identycznej operacji
    - duplikaty danych mogą być też po prostu wykrywane i usuwane

Large graph processing

MapReduce nie działa dla grafów:

- równoległe przetwarzanie grafów to często algorytmy iteracyjne, o dużej liczbie iteracji, a MapReduce dla takich algorytmów działa słabo
- każda iteracja MapReduce wymaga zmaterializowania (obliczenia) pełnych wyników pośrednich - słaba wydajność

Single Source Shortest Path (SSSP):

- typowy problem grafowy, chcemy znaleźć najkrótsze ścieżki z jednego wierzchołka do wszystkich innych
- zastosowania: badania operacyjne, głównie w logistyce
- warianty:
    - pojedynczy procesor - algorytm Dijkstry
    - MapReduce, Pregel - współbieżny BFS, wykonywany iteracyjnie wielokrotnie

Model Pregel:

- model do obliczeń grafowych, oparty o wykonywanie w pętli "superkroków"
- idea "think like a vertex", programista zapewnia kod wykonywany przez każdy wierzchołek
- pojedynczy superkrok (superstep) jest wykonywany współbieżnie dla wszystkich wierzchołków:
    - otrzymuje wiadomości od sąsiadów, wyemitowane w poprzednim superkroku; jeżeli był nieaktywny, to zostaje aktywowany
    - wykonuje zdefiniowaną funkcję, aktualizując swoją wartość albo wychodzących krawędzi
    - tworzy wiadomość i emituje ją do sąsiadów
    - jeżeli nie ma więcej pracy do wykonania, to głosuje za zakończeniem algorytmu i przechodzi w stan nieaktywności
- warunek końca:
    - brak wysłanych wiadomości
    - wszystkie wierzchołki są jednocześnie nieaktywne
- porównanie do MapReduce:
    - w Pregelu wierzchołki i krawędzie są trzymane cały czas (w kolejnych iteracjach) na maszynie, która wykonuje obliczenia, natomiast w MapReduce trzeba by złączyć stan po każdym kroku
    - Pregel przesyła przez sieć tylko wiadomości między wierzchołkami grafu, natomiast MapReduce cały stan grafu (trzeba go zapisać między iteracjami)
- combiners:
    - optymalizacja, którą można dodać do funkcji
    - łączy (combine) wiadomości, żeby zmniejszyć liczbę wiadomości przesyłanych przez sieć, np. agreguje wartości (min/max/sum/avg)
    - łączenie następuje i lokalnie, i zdalnie, w sumie jest przesyłane mniej wiadomości

- aggregators:
    - mechanizm do obliczania statystyk agregowanych po wszystkich wierzchołkach
    - podczas superkroku każdy worker agreguje informacje z wierzchołków które przechowuje, tworząc częściowy agregat
    - częściowe agregaty są układane w strukturę drzewa, co pozwala na uwspółbieżnienie procesu
    - finalny agregat jest przesyłany do mastera
- architektura systemu:
    - graf jest dzielony na partycje (zbiory wierzchołków), rozsyłane na workerów
    - węzły rozsyłają wiadomości asynchronicznie, ale synchronizują się przed końcem superkroku
    - master może kazać na koniec workerom zapisać część albo całość grafu

Spark GraphX:

- dość niskopoziomowa abstrakcja nad RDD do obliczeń grafowych
- operuje na strukturze Property Graph:
    - skierowany multigraf
    - RDD dla krawędzi i wierzchołków, z określonymi przez użytkownika typami (można przypisać arbitralne atrybuty)
    - niemutowalny, rozproszony, fault tolerant
    - rozproszony między workerów wykorzystując heurystyki partycjonowania wierzchołków
    - każdy wierzchołek ma unikalne ID
- oferuje Pregel API do definiowania programów w stylu Pregela
- partycjonowanie:
    - odbywa się przez cięcie grafu według wierzchołków (vertex cut)
    - skutkuje podzieleniem wierzchołka i przechowywaniem go na wielu węzłach
    - wydajniejsze niż cięcie krawędzi (edge cut) dla grafów z wierzchołkami o dużym stopniu (hubs)
- przechowywanie wierzchołków:
    - atrybuty wierzchołków są dołączone do krawędzi
    - typowo krawędzi jest znacznie więcej niż wierzchołków, więc taka reprezentacja jest wydajniejsza
    - routing table pozwala szybko sprawdzić, gdzie przesyłać informacje przy joinie dla takiej reprezentacji

Big data ecosystem

Distributed file systems

Google File System:

- pierwszy rozproszony system plików wielkiej skali, stworzony w Google'u na ich potrzeby wewnętrzne
- założenia:
    - wysoka częstotliwość awarii komponentów - bo używa się typowego, taniego hardware'u
    - przeciętna liczba, ale ogromne pliki - kilka milionów plików rozmiaru 100 MB albo większych
    - pliki są zwykle write-once, najwyżej się do nich dopisuje
    - duże ciągłe odczyty, w tym streamowe
    - ważniejsza jest wysoka, ciągła przepustowość niż niskie opóźnienia
- cechy:
    - pliki przechowywane w dużych chunkach o stałej wielkości 64 MB
    - niezawodność przez replikację - przynajmniej 3 kopie dla każdego chunka
    - proste, scentralizowane zarządzanie z pojedynczym masterem przechowującym metadane
    - brak cache'owania danych - daje niewielkie zyski przy dużych zbiorach i ciągłym odczycie
    - proste, ale customowe API - wspiera dodatkowo np. snapshoty, dopisywanie rekordów
- klaster:
    - pojedynczy master
    - wiele chunkserverów (przechowują chunki z danymi)
    - współbieżny dostęp wielu klientów
- stworzony nie jako ogólne narzędzie, ale z założeniem, że aplikacje będą projektowane pod korzystanie z GFSa
- interfejs:
    - podobny do typowego Unixowego, ale nie spełnia standardu POSIX
    - pliki zorganizowane w katalogi, typowe operacje plikowe
    - dodatkowa operacja snapshot, tworząca kopie plików i katalogów
    - dodatkowa możliwość dopisywania do plików, co więcej wielu klientów naraz może współbieżnie dopisywać do tego samego pliku
    - pozwala na multi-way merge i kolejki producer-consumer na plikach

Pliki w GFS:

- podzielone na chunki po 64 MB
- każdy chunk ma swoją etykietę, 64-bitową, przydzielaną centralnie przez mastera podczas tworzenia
- w ramach metadanych GFS przechowuje logiczne mapowanie pliku na chunki, które go tworzą
- chunki są replikowane przynajmniej 3 razy, ale więcej razy dla plików szczególnie ważnych lub często używanych
- chunki są duże, bo minimalizuje to:
    - liczbę interakcji z masterem
    - ilość metadanych
    - liczbę requestów TCP
- duże chunki zwiększają prawdopodobieństwa hot spotów, czyli bardzo intensywnie używanych chunków, ale nie jest to duży problem, bo mamy replikację
- można albo tworzyć, albo dopisywać nowe części, nie ma modyfikacji samego pliku

Master w GFS:

- pojedynczy, co jest bardzo dużym uproszczeniem
- powyższe ogranicza niezawodność (single point of failure) i skalowalność, stąd 2 rozwiązania: shadow masters i minimalizacja mastera
- master spełnia minimalną rolę, dzięki czemu minimalnie wpływa na skalowalność:
    - nigdy nie przesyła danych, klienci komunikują się bezpośrednio z chunkserverami
    - master przesyła tylko metadane
    - duże chunki, minimalizacja metadanych
- shadow masters - repliki read-only
- master przechowuje log z najważniejszymi aktualizacjami metadanych i checkpointami, persystowany na dysku i replikowany, co przyspiesza w razie potrzeby recovery
- kiedy klient pyta o pliki, to dla optymalizacji zwraca nie tylko chunki, o które prosi, ale też o kolejne, bo odczyt jest zwykle strumieniowy (wiele kolejnych chunków)
- monitoruje status chunkserverów regularnymi requestami heartbeat
- zadania:
    - przechowywanie metadanych
    - zarządzanie namespace'ami
    - heartbeat z chunkserverami
    - zarządzenie chunkami:
        - tworzenie
        - balansowanie replikacją: zużycie pamięci vs prędkość dostępu
        - re-replikowanie danych (jeżeli spadnie liczba replik)
        - rebalansowanie danych między węzłami, żeby równoważyć obciążenie pamięciowe i obliczeniowe
        - garbage collection - powstawanie starych śmieci jest bardzo częste w systemie rozproszonym, gdzie często mamy awarie
        - wykrywanie i usuwanie stale replicas

Metadane w GFS:

- przechowywane na masterze (i shadow masterach)
- namespace'y plików i chunków
- mapowanie plik -> chunki tego pliku
- lokacje replik z danym chunkiem
- przechowywane w pamięci, bo są małe, a zapewnia to szybkość i łatwość dostępu

Chunkservers w GFS:

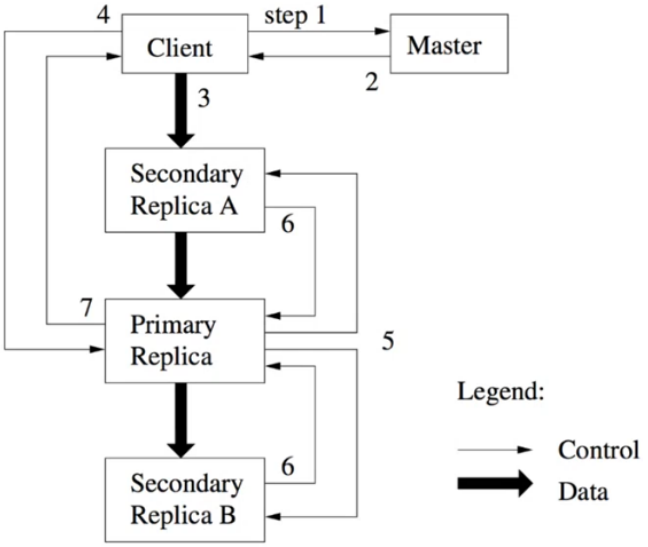
- Linuxowy system plików
- bezpośrednie przesyłanie danych z klientem
- ani one, ani klienci nie cache'ują plików
- Linuxowe bufory i tak I/O cache'ują w RAMie często używane chunki

Model spójności GFS:

- wszystkie operacje na metadanych (namespace mutations) przechodzą przez mastera i są atomowe
- region pliku (jego chunki) może być w stanie:
    - consistent - wszyscy klienci widzą spójne dane dla pliku
    - defined - wszyscy klienci widzą spójne dane oraz uwzględniają one najnowszą aktualizację (lepsze od consistent)
    - undefined but consistent - wszyscy klienci widzą spójne dane, ale niekoniecznie z najnowszymi zmianami
    - inconsistent - różni klienci widzą różne dane dla pliku
- mutacje danych (data mutations):
    - write - zapisuje dane na określonym offsecie, tworzy nowy plik
    - record append - dopisanie danych co najmniej raz (semantyka at least once) na offsecie obliczanym przez GFS
- spójność zapewniają:
    - wykonywanie mutacji dla wszystkich chunków pliku w tej samej kolejności
    - wersjonowanie chunków (oznaczanie numerami wersji)

Proces mutacji w GFS:

- mutacja - write (nowy plik) lub append (dopisanie do istniejącego), wykonywany dla wszystkich replik
- cel: minimalizacja udziału mastera
- schemat:



- mechanizm dzierżawy (lease) - master wybiera replikę, oznacza ją jako primary i daje jej lease na mutacje, typowo na 60 sekund (z wydłużaniem)
- primary definiuje kolejność mutacji, wszystkie pozostałe repliki mają podążyć za tą kolejnością
- rozdzielenie przepływu danych i przepływu kontroli - pozwala zminimalizować udział mastera
- mutacja krok po kroku:
    1. Klient pyta mastera, który chunkserver ma aktualnie dzierżawę na dany chunk (primary) i o lokalizacje pozostałych (secondary). Jeżeli nikt nie ma dzierżawy, to master wybiera primary i go o tym informuje.
    2. Master zwraca tożsamość primary i secondary. Klient cache'uje te dane dla kolejnych mutacji dla chunka - z masterem będzie musiał gadać dopiero, gdy primary stanie się niedostępny lub straci dzierżawę.
    3. Klient wysyła dane do wszystkich replik, w dowolnej kolejności, a one je cache'ują.
    4. Kiedy wszystkie repliki potwierdzą otrzymanie danych, klient wysyła write request do primary. Primary wyznacza kolejność mutacji i wykonuje je po kolei.
    5. Primary forwarduje write request do pozostałych replik. Każda secondary replika wykonuje mutacje w tej samej kolejności co primary.
    6. Repliki odpowiadają primary potwierdzeniem.
    7. Primary odpowiada klientowi. Jeżeli na którejkolwiek replice wystąpił błąd (nie wystąpił na primary, bo wtedy w ogóle nie doszlibyśmy do tego kroku), to jego treść jest dołączona i write jest uznawany za porażkę, a repliki są w stanie inconsistent. Typowo w takim wypadku klient powinien powtórzyć próbę mutacji.

Snapshoty:

- kopiowanie plików i katalogów równolegle z normalnymi operacjami
- używa podejścia copy-on-write
- dane kopiowane są tymczasowo read-only, żeby zapewnić spójność kopii
- tworzenie snapshotu nie kopiuje całych chunków, tylko zwiększa reference counter dla każdego chunka, co jest tanie
- faktyczne kopiowanie:
    - kiedy klient pisze chunk, to master sprawdza wartość reference counter
    - jeżeli jest większa niż 1, to najpierw tworzona jest faktyczna kopia chunków
    - klient może zaktualizować kopię, zamiast chunka będącego częścią snapshotu
- kopiowanie jest opóźnione w nadziei że nie wszystkie chunki będą modyfikowane, więc będzie można zrobić mniej kopii

HDFS:

- bardzo silnie wzorowany na GFSie, ale open source
- różnice względem GFSa są raczej kosmetyczne:
    - NameNode zamiast mastera
    - DataNodes zamiast chunkserverów
    - bloki (blocks) zamiast chunków
- dalej zakłada dostęp sekwencyjny, opóźnienie (latency) dalej jest duże
- używany przez Hadoopa i HBase

Big data DBs

Google BigTable:

- rozproszony storage system do zarządzania danymi strukturalnymi i semi-strukturalnymi
- cele projektowe:
    - skalowanie do petabajtów danych
    - możliwość ciągłej zmiany różnych elementów danych przez asynchroniczne procesy, a przy tym dostęp do najbardziej aktualnego elementu
    - potężna przepustowość read/write (miliony operacji na sekundę)
    - efektywne skanowanie całości lub podzbiorów danych
    - efektywne joiny dużych zbiorów jeden-do-jeden i jeden-do-wiele
    - możliwość analizy zmian danych w czasie
- cechy:
    - rozproszony, wielopoziomowy słownik - wide-column store
    - persystentny, fault tolerant
    - skalowalny
    - self-managing: dynamiczne dodawanie/usuwanie serwerów, load balancing

Model danych Big Table:

- oparty o Sorted String Table (SSTable), w których wiersze mogą mieć różne zbiory kolumn;
- baza NoSQL typu **wide-column store**
- rzadka, rozproszona, wielowymiarowa, posortowana mapa (słownik)
- odwzorowanie:
  `(row, column, timestamp) -> cell content`
- wiersze:
    - nazwa - dowolny string
    - dostęp atomowy (transakcyjny) do wierszy, pomiędzy wieloma wierszami nie ma spójności
    - wiersz jest implicite tworzony przy zapisywaniu danych
    - wiersze są posortowane leksykograficznie
- tablet:
    - grupa kolejnych wierszy
    - jednostka rozproszenia i load balancingu
    - odczyt krótkich zakresów wierszy jest efektywny, bo odpowiada kilku tabletom, więc jest też typowo na niewielkiej liczbie maszyn
- kolumny:
    - dwupoziomowa struktura z grupowaniem kolumn w rodziny:
      `family:optional_qualifier`
    - rodzina to typowo dane o tym samym typie, chociaż w ramach rodziny można użyć dowolnych kolumn
    - rodzina to jednostka kontroli dostępu
    - liczba kolumn jest nieograniczona, bo to baza rzadka - jeżeli dany wiersz nie ma kolumny, to nie zajmuje ona miejsca
    - liczba rodzin powinna być dość niewielka, rzędu setek

- timestampy:
    - używane, żeby przechowywać wiele różnych wersji danych w jednej komórce bazy
    - typowo zapis używa aktualnego czasu, ale można explicite podać inny timestamp
    - możliwości:
        - zwróć najnowsze K wartości
        - zwróć wartości z timestampem z zakresu od X do Y
        - zwróć wszystkie wartości
    - pozwalają implementować garbage collection:
        - przechowuj tylko ostatnie K wartości
        - trzymaj wartości dopóki nie będą starsze niż K sekund
    - garbage collection ustawia się na poziomie rodzin kolumn

Architektura BigTable:

- klient kieruje requesty do frontendu, który przekierowuje requesty do klastra BigTable
- do klastra można swobodnie dodawać węzły, zwiększając przepustowość
- można replikować klaster (w całości), żeby włączyć failover
- węzły BigTable zawierają tylko metadane, faktyczne dane są przechowywane w systemie Colossus (następca GFS)
- load balancing:
    - ze względu na obciążenie obliczeniowe (workload) i pamięciowe (data volume)
    - za duże / zbyt obciążone tablety są dzielone na pół, za małe / mało używane są łączone
    - w razie potrzeby następuje redystrybucja między węzłami

Filtry Blooma:

- probabilistyczna struktura danych do stwierdzania, czy element jest częścią zbioru
- pozwala na false positive'y, ale nie na false negative'y - zwraca "możliwe, że jest w zbiorze" albo "na pewno nie jest w zbiorze"
- wykorzystywana do optymalizacji w BigTable
- operacja odczytu musi wczytać SSTable z dysku, jeżeli nie ma go w pamięci
- filtr Blooma pozwala stwierdzić, czy dany SSTable może zawierać dane dla danej pary wiersz-kolumna i wczytać znacznie mniej tabel do pamięci
- szczególnie ogranicza zużycie dla nieistniejących wpisów, bo dzięki temu mogą w ogóle nie używać dysku

Spanner:

- relacyjna baza danych: SQL, ACID etc.
- highly available + consistent (global read consistency without locking)
- CP, ale "CAP":
    - consistent - zawsze daje spójne dane
    - tolerant to partitions - niezbędne w dużych systemach
    - highly available - "five 9s" dostępności, nie jest to 100% (więc nie ma własności A), ale w praktyce właściwie zawsze wystarczające
- wykorzystuje mechanizm splitów dla skalowalności i niezawodności

Spanner - splity:

- problem: tabele mogą być albo za duże jak na pojedyncza maszynę, albo małe ale bardzo intensywnie używane (zbyt obciążona maszyna)
- rozwiązanie: partycjonowanie danych
- tabele są podzielone na ciągłe zakresy wierszy - splity, gdzie pojedyncza maszyna może obsługiwać 1 lub więcej splitów
- centralna szybka usługa do sprawdzania która maszyna ma który zakres kluczy
- splity są replikowane dla wysokiej dostępności, a spójność zapewnia algorytm Paxos

Spanner - spójność:

- strong read:
    - domyślny tryb odczytu
    - aktualny timestamp, gwarantuje spójność
    - może mieć wyższe opóźnienie (latency), bo trzeba czekać, aż repliki uspójnią stan
- stale read:
    - odczyt w określonym timestampie w przeszłości
    - żadnych blokad i uspójniania replik
    - mniejsze opóźnienie, bo nie trzeba czekać na uspójnienie replik
- write, read-write:
    - wymaga blokady
    - główny węzeł (lider) zakłada blokadę, buforuje blokadę
    - repliki wykonują u siebie te zmiany co lider, głosują w kworum na udanie się zapisu
- ogólna gwarancja spójności:
    - tradycyjna mocna spójność to serializability, czyli transakcje są widoczne tak, jakby wykonywały się po kolei
    - Spanner oferuje jeszcze silniejszą spójność - **external consistency**
    - system zachowuje się tak, jakby wszystkie transakcje były sekwencyjne na wszystkich serwerach jednocześnie, każda transakcja jest widoczna jak pojedynczy punkt w czasie
    - jeżeli T2 zacznie się commitować po tym jak T1 skończy się commitować, to timestamp dla T2 jest większy niż dla T1
    - może powodować konieczność czekania przez litera na globalne uspójnienie systemu, ale umożliwia brak blokad

Spanner - synchronizacja czasu:

- wykorzystuje system TrueTime, zegar rozproszony o wysokiej dostępności
- pozwala dość ściśle synchronizować zegary w globalnie rozproszonym systemie
- `TrueTime.now()` zwraca zakres `[t - ε, t + ε]`:
    - ε to zwykle ~2 ms
    - dolna granica nastąpiła już wszędzie na świecie
    - górna granica jeszcze nigdzie nie nastąpiła
- synchronizowane przez osobną sieć serwerów, synchronizującą zegary atomowe z użyciem systemu GPS

Architectures of data centers

Klastry big data:

- osobne duże dyski dla każdego workera (w przeciwieństwie do scentralizowanego systemu plików z HPC)
- połączenie typu fat tree (cienkie node-switch, grube switch-switch)
- zwykle nie mają RAID

Superkomputery i ich ciekawe cechy:

- Fugaku:
  - hierarchiczna budowa
  - procesory ARM
  - zoptymalizowane połączenia, 6-wymiarowy torus (Tofu-D)
- Summit:
  - użycie GPU Nvidii
- Sunway TaihuLight:
  - centralna sieć i dyski
  - gęsto połączone wewnętrznie supernode'y z komunikacją do zewnętrznej sieci i dysków, połączone centralną switch network między sobą
- SuperMUC-NG:
  - architektura Intela
  - zakupione jako całość, tylko z przedstawionymi wymaganymi osiągami

Tensor Processing Unit (TPU) - akcelerator ASIC do uczenia maszynowego, zasadniczo wyspecjalizowane GPU z odpowiednimi układami do mnożenia macierzy.

Resource management

Uruchamianie zadań w Sparku:

- definiujemy zadanie w kodzie, potem następuje optymalizacja, wybór planu itp. aż dostaniemy plan fizyczny
- według planu fizycznego uruchamia się jeden job per action (akcje - zwracają wartości w Sparku)
- job jest rozbijany na etapy (stages):
    - liczba etapów zależy od liczby operacji shuffle
    - stage to grupa tasków, które mogą być razem wykonywane
    - Spark optymalizuje to tak, żeby jak najwięcej pracy było wykonywane per stage, żeby zminimalizować ilość przesyłanych danych (etap shuffle)
- shuffle:
    - faktyczne fizyczne repartycjonowanie danych
    - wymaga skoordynowania executorów tak, żeby poprzesyłać dane
    - kosztowna operacja, im mniej, tym lepiej
- tasks:
    - task - połączenie bloków danych i serii transformacji, które wykonają się na pojedynczym executorze
    - dużo mniejszych tasków = mogą się wykonać równolegle = jest szybciej
    - task to pojedyncza jednostka obliczeń wykonywana na pojedynczej jednostce danych - partycji (partition)
- pipelining:
    - sekwencja operacji, z których każda przekazuje swoje wyjście następnej jako wejście (tworzą pipeline), bez potrzeby przenoszenia tych danych między węzłami, jest łączona w pojedynczy etap
    - transparentny podczas pisania aplikacji
    - optymalizacja, bo minimalizuje przesyłanie danych

Wykonywanie aplikacji w Sparku:

- każde wykonanie aplikacji wymaga drivera, executorów i managera klastra
- driver:
    - kontroluje wykonanie aplikacji i stan klastra
    - komunikuje się z managerem klastra, żeby dostawać dostęp do fizycznych zasobów i uruchamiać executory
    - w dowolnym języku Sparka, np. Python, Scala
- executory:
    - procesy wykonujące taski
    - bierze task, wykonuje, zwraca stan (sukces lub porażka) i wyniki
    - wykonują kod w JVMie
- cluster manager:
    - utrzymuje sam klaster: maszyny, zasoby i komunikację
    - ma własnego drivera (mastera) i workery
    - związany z fizycznymi maszynami, nie procesami

Zarządzanie zasobami w Sparku:

- każda aplikacja ma swoje osobne procesy executorów - daje izolację
- executory istnieją przez cały czas istnienia aplikacji
- każdy executor może mieć wiele wątków i uruchamiać w nich wiele tasków
- zasoby są osobne dla aplikacji - w szczególności różne aplikacje (z innymi SparkContext) nie mogą współdzielić danych bez użycia zewnętrznego storage'u
- Spark jest niezależny od managera klastra, można to dość swobodnie podmieniać

Rodzaje przydzielania zasobów:

- statyczne (static resource allocation):
    - każdy executor zaczyna z pewną narzuconą z góry ilością zasobów
    - niewydajne, bo poszczególnym workerom często albo będzie brakować zasobów, albo będą ich zużywali za dużo, a zasoby uwalniane są dopiero na koniec joba
- dynamiczne (dynamic resource allocation):
    - regularne sprawdzanie obciążenia i zmiana liczby executorów w razie potrzeby
    - Spark używa heurystyk, żeby szacować, czy mamy za dużo, czy za mało executorów
    - dodaje się 1, 2, 4, 8… executorów - dla małych aplikacji takich wzrostów będzie mało, a przy dużym obciążeniu daje to dynamiczne skalowanie
    - executory są usuwane, jeżeli są nieużywane (idle) przez określony czas
- w Sparku każdy cluster manager udostępnia oba tryby, przy czym domyślnie używany jest przydział statyczny

Typowe managery klastra w Sparku:

- Spark Standalone - prosty i łatwy w instalacji, dostarczany razem ze Sparkiem
- Apache Mesos - ogólnego przeznaczenia, do jobów ad-hoc i długo działających serwisów, może też uruchamiać Hadoopa i inne serwisy
- Hadoop YARN - stworzony na potrzeby zarządzania Hadoopem
- Kubernetes - do zarządzania dowolnymi aplikacjami wdrażanymi jako kontenery Dockera

YARN (Yet Another Resource Negotiator):

- stworzony na potrzeby jobów MapReduce w Hadoopie
- dość ogólny cluster manager, pozwala na współdzielenie zasobów fizycznych klastra przez różne aplikacje ekosystemu Hadoopa, np. Spark, Hive, HBase
- dynamicznie balansuje zasoby między wiele aplikacji
- stworzony do długo działających, dużych aplikacji, głównie wymagających skalowania w górę
- działanie:
    - ResourceManager - nadzoruje globalną pulę zasobów klastra, pozwala na globalną konfigurację ograniczeń w klastrze
    - NodeManager - slave ResourceManagera, zarzadza konkretnym węzłem
    - ApplicationMaster - po jednym dla każdej aplikacji (np. Spark i MapReduce), żąda zasobów od ResourceManagera i używa NodeManagerów do wykonywania tasków

    - container - zbiór zasobów, reprezentowany jako dyskretny zbiór (żeby dało się przydzielać po kawałku) np. pamięć, CPU dysk
    - Scheduler - używany przez ResourceManagera, przydziela containery do procesów, zarządza tym, ile zasobów przyznać i w jakiej chwili
- Spark na YARNie:
    - każdy executor działa jako osobny container
    - można skonfigurować z poziomu Sparka, ile każdy node dostanie rdzeni i pamięci
    - containery dostają 1 lub więcej tasków Sparkowych do wykonania

Apache Mesos:

- stworzony jako cluster manager ogólnego przeznaczenia
- dobrze nadaje się i do szybkiego przetwarzania interaktywnego, i do długo działających serwisów wymagających skalowania w górę i w dół
- bardziej dynamiczny od YARNa, nadaje się do szerszego grona zadań działających na jednym klastrze, np. równolegle Sparka i aplikacji webowej z interfejsem do niego
- działanie:
    - frameworks - poszczególne aplikacje
    - każdy framework ma swój scheduler, replikowanego mastera i negocjuje z Mesos masterem o zasoby
    - Mesos tworzy listę ofert z zasobami i przesyła je frameworkom żądającym zasobów, przy czym zarządza on też przydziałem zasobów zgodnie z narzuconymi zasadami i konfiguracją
    - framework może akceptować albo odrzucać oferty zasobów
    - decyzja o tym, jak użyć przydzielonych zasobów zależy i od schedulera, i od frameworku, więc jest to skalowalne i elastyczne
- Spark na Mesosie:
    - Spark to po prostu kolejny framework z perspektywy Mesosa, dostaje przydział zasobów, a potem sam już sobie nimi zarządza
    - obecnie używany jest tryb coarse-grained, w którym każdy Spark executor to pojedynczy task w Mesosie

Kubernetes (k8s):

- system do zarządzania klastrem opartym o kontenery Dockera
- Spark na Kubernetesie jest eksperymentalny, ale intensywnie rozwijany
- pod - podstawowa jednostka wdrożenia, grupa wspólnie działających kontenerów
- Spark driver i executory działają jako pody, przy czym driver sam może spawnować pody executorów

Szeregowanie (scheduling):

- zadanie ułożenia zadań do wykonania po kolei i przydzielenia im zasobów
- wiele różnych heurystycznych polityk
- można zrobić wiele kolejek, gdzie np. każdy klient dostaje osobną kolejkę, typowo mają jednak jeden sposób szeregowania
- FIFO:
    - First In, First Out
    - zadania ułożone w kolejce, wykonujemy po kolei
    - proste, nie wymaga konfiguracji
    - nie nadaje się do klastrów współdzielonych przez wielu użytkowników
    - duże aplikacje głodzą krótkie, które są za nimi w kolejce
- capacity scheduler:
    - 2 osobne kolejki: jedna na małe zadania, druga na duże zadania
    - lepsze dla małych zadań
    - może powodować mniejszą utylizację klastra, bo możemy mieć okresy bez małych zadań
    - duże zadania wykonują się wolniej
- fair scheduler:
    - dynamicznie przydziela zasoby do zadań
    - każde z N zadań dostaje około 1/N zasobów (fair share)
    - szybko kończy małe zadania, a opóźnienia w dużych zadaniach są lepsze niż w capacity schedulerze
    - kiedy klaster jest obciążony, to trzeba by potencjalnie długo czekać, żeby zdobyć zasoby dla nowego joba, dlatego można dodać **wywłaszczanie (preemption)**, czyli ubicie kontenerów zajmujących więcej niż fair share i realokację zasobów
    - wywłaszczanie zmniejsza ogólną wydajność klastra, bo zadania przerwane trzeba wykonać od początku
- przy systemie z wieloma kolejkami jeżeli jedna jest wolna, to jej zasoby można pożyczyć innej kolejce - queue elasticity
- przy wielu kolejkach i fair schedulerze można dodać fair scheduling pomiędzy wieloma kolejkami, też np. dodać ważenie kolejek

Delay scheduling:

- chcemy, żeby aplikacje mogły odpalać się na wskazanym węźle, np. ze względu na dostępność danych
- wszystkie schedulery w YARNie starają się honorować lokalność danych, ale typowo węzeł będzie już używany
- delay scheduling polega na poczekaniu kilka sekund na zwolnienie węzła - bardzo zwiększa to szansę na to, że węzeł się zwolni i zadanie dostanie pożądany węzeł, co daje lepszą wydajność
- każdy node manager regularnie wysyła do resource managera heartbeat z sugestią scheduling opportunity, czyli ofertą uruchomienia na danym węźle kontenera
- scheduler nie bierze pierwszego z brzegu scheduling opportunity, tylko czeka zbierając pewne N ofert i jeżeli dostanie lokalną, to przydziela zadanie (szanując żądanie lokalności), a jak nie, to dopiero w takim pesymistycznym wypadku przydzieli task do innego węzła

Big data ML

Machine learning in Spark

Typy zadań w uczeniu maszynowym:

- klasyfikacja
- regresja
- klasteryzacja
- rekomendacja
- wyszukiwanie częstych wzorców
- uczenie ze wzmocnieniem

MLlib:

- biblioteka do uczenia maszynowego w Sparku
- czasem używa się określenia "Spark ML", kiedy używa się API dla DataFrame'ów
- rodzaje elementów:
    - DataFrame - po prostu ramka danych Sparka z danymi, może przechowywać np. teksty, wektory (element z MLliba), prawdziwe klasy, predykcje (dodawane jako kolumna)
    - transformer - algorytmy przekształcające jeden DataFrame w drugi, np. przekształcenia cech, model ML
    - estimator - algorytm ML, który może zostać wytrenowany (fit) na danych, żeby stworzyć model (typu transformer)
    - parameter - argument transformera / estimatora, hiperparametr

MLlib - operacje na cechach:

- ekstrakcja cech:
    - tworzenie cech na podstawie danych
    - np. TF-IDF, Word2Vec
- transformacja cech:
    - przekształcenie istniejących cech
    - np. StringIndexer, Normalizer, QuantileDiscretizer
- selekcja cech:
    - wybranie podzbioru cech z istniejących
    - np. VectorSlicer, ChiSqSelector
- locality sensitive hashing (LSH):
    - technika haszowania danych w wektory (punkty) tak, żeby podobne dane leżały blisko siebie w docelowej przestrzeni
    - można traktować jako zbiór różnych algorytmów redukcji wymiarowości, które starają się zachować względną odległość między punktami po redukcji
    - np. Bucketed Random Projection

Przykładowe operacje na cechach:

- StringIndexer - zamienia stringi (z typowo niewielkiego zbioru) na kolejne inty
- QuantileDiscretizer - dyskretyzuje dane, dzieląc je na kubełki według kwantyli
- ChiSqSelector - selekcja cech w oparciu o test chi kwadrat, testuje zależność klasy od każdej cechy po kolei i wybiera cechy, od których najbardziej zależy klasa
- StandardScaler - standaryzacja cech, czyli odjęcie średniej i podzielenie przez odchylenie standardowe
- MinMaxScaler - zmiana zakresu wartości na [a,b], typowo [0,1]
- OneHotEncoder - one hot encoding dla danych kategorycznych

Klasyfikacja:

- regresja logistyczna
- drzewa decyzyjne:
    - algorytm zachłanny top-down
    - maksymalizuje metrykę information gain, czyli ilość informacji, którą przynosi nam podział, czyli zmniejszenie entropii zawartej w danych dzięki podziałowi
    - idea: czystsze węzły, czyli zawierające mniej pomieszane klasy, są prostsze, więc wystarczy mniej bitów, żeby je opisać, zatem mają mniejszą entropię
    - chcemy maksymalizować; maksymalizacja information gain jest równoważna minimalizacji entropii
    - jest to różnica między sumą entropii dzieci przy danym podziale, ważona liczbą punktów, które trafiły do poszczególnych dzieci, a entropią rodzica:

      $$H(X) = \sum_{i}^{N} p(x_i) \log p(x_i)$$

      $$IG(X) = \frac{|X_{left}|}{|X|} * H(X_{left}) + \frac{|X_{right}|}{|X|} * H(X_{right}) - H(X)$$

      `H(X)` - entropia Shannona
    - ważenie bierze się stąd, że mało punktów o dużej entropii boli nas dużo mniej, niż dużo punktów o dużej entropii
- lasy losowe
- gradient boosting
- liniowy SVM
- naive Bayes

Regresja:

- regresja liniowa
- uogólnione modele liniowe (generalized linear model / regression)
- drzewa decyzyjne - przy podziale minimalizują sumę kwadratów różnic, czyli wariancję
- lasy losowe

Rekomendacja:

- zadanie wyznaczenia top K przedmiotów dla użytkownika
- typowo rozwiązuje się jako zadanie regresji, gdzie chcemy policzyć wyniki (scores) dla przedmiotów dla użytkownika, a potem wybrać K z najwyższymi wynikami
- regresja liniowa:
    - najłatwiejsze podejście
    - minimalizujemy (regularyzowany) błąd średniokwadratowy
    - używa typowo spadku wzdłuż gradientu (da się efektywnie zaimplementować dla dużych danych)
    - wymaga wyznaczenia wektorów cech dla przedmiotów
- collaborative filtering:
    - podejście oparte o rekonstrukcję macierzy - nie przewidujemy explicite wartości przedmiotów, tylko traktujemy wejściową macierz jak wybrakowaną i staramy się ją uzupełnić
    - dalej optymalizuje się błąd średniokwadratowy, ma bardzo naturalny zapis dla całych macierzy
    - wykorzystuje przybliżoną dekompozycję macierzy wejściowej, typowo rzadkiej, na iloczyn dwóch macierzy gęstych:
      `R ~ U^T * V`
      `R` - kształt (m * n)
      `U^T` - kształt (m * k)
      `V` - kształt (k * n)
    - `k` jest typowo małe (rzędu 10-100), dzięki czemu predykcja jest bardzo efektywna, bo
    - macierze gęste reprezentują nauczone cechy użytkowników i przedmiotów (latent vectors), czyli ich reprezentacje w pewnej k-wymiarowej przestrzeni (latent space)
    - wykorzystuje algorytm Alternating Least Squares (ALS), w którym na przemian optymalizujemy metodą najmniejszych kwadratów macierze `U` i `V`
    - ma problem zimnego startu, czyli napotkanie nowych użytkowników lub przedmiotów, dla których jeszcze nie ma ocen; w takim wypadku MLlib zwraca NaN i trzeba to obsłużyć ręcznie, np. zwrócić wartość średnią

Selekcja modelu:

- prosty holdout (train-validation) oraz walidacja skrośna
- używają grid search'a
- ma dostępne metryki dla klasyfikacji, regresji i rankingu (rekomendacja)

Federated learning

Federated learning:

- trening modeli ML używając nielokalnych danych, do których nie mamy bezpośredniego dostępu, brak możliwości przesłania do centralnego serwera
- zastosowania: urządzenia mobilne, medycyna
- zdalne węzły trenują modele na podstawie lokalnych danych i z centralnym serwerem synchronizują tylko model
- celem praktycznym jest minimalizacja funkcji kosztu ważonej ilością danych przypadającą na klienta, czyli wartości oczekiwanej kosztu:

$$f(w) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(w)$$

$$\mathbb{E}_{\mathcal{P}_k}[F_k(w)] = f(w)$$

- w przeciwieństwie do tradycyjnego scentralizowanego treningu w data center:
  - bardzo duże koszty komunikacyjne
  - rzadki udział klientów
  - niewielka liczba aktualizacji w każdym udziale klienta
  - niewielkie koszty obliczeniowe per klient (bo zbiory danych są małe, a procesory mobilne szybkie)
- typowo opłaca się zrobić więcej rund lokalnego treningu i zmniejszyć ilość komunikacji

Model averaging:

- najprostszy algorytm federated learning
- serwer wysyła węzłom niewytrenowany model (żeby wiedziały, co trenować), a węzły używając swoich lokalnych danych trenują modele i wysyłają je serwerowi
- finalny model jest tworzony przez proste uśrednienie parametrów z modeli
- uśrednienie modeli jest równoważne z uśrednianiem gradientów, ale wersja z modelem pozwala na lokalny trening przez wiele iteracji

$$w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$$

Algorytm federated averaging:

1. Inicjalizuj losowo wagi początkowe
2. Wykonaj T rund treningowych, w każdej rundzie wykonuj poniższe kroki
3. Wybierz losowy podzbiór klientów, dla każdego z tych klientów wykonaj aktualizację klienta (trening sieci neuronowej przez E lokalnych epok) i zbierz ich wagi
4. Uśrednij wagi od klientów, ważąc je odpowiednio

Wyzwania w federated learningu:

- masywnie rozproszony trening - mnóstwo urządzeń z danymi
- ograniczona komunikacja z klientami
- niezbalansowane dane - różni klienci mają bardzo różną ilość danych
- brak własności i.i.d.
- zawodne węzły obliczeniowe - klienci mogą w dowolnym momencie się rozłączyć
- dynamiczna dostępność danych - zmienia się z czasem, różnie u różnych klientów

Secure aggregation protocol:

- protokół agregujący modele od klientów tak, żeby też na podstawie samego modelu nie dało się ustalić, z którego węzła został on otrzymany
- do modeli zwracanych przez klientów dodaje się maski, losowe patrząc na pojedynczą maskę, co uniemożliwia identyfikację
- maski są tworzone tak, żeby się znosić przy zsumowaniu, dzięki czemu dalej można korzystać z uśredniania modeli
- maski:
    - pairwise distributed, czyli klienci mają parami uzgodnione maski, dzięki czemu po zsumowaniu się znoszą
    - używa algorytmu Diffie-Hellman secret key agreement
    - współdzielenie secretów k-out-of-n, czyli dzielimy secret (np. całość maski) na n kawałków i jeżeli mamy mniej niż k kawałków to nie da się nic ustalić, a mając co najmniej k da się idealnie zrekonstruować cały szyfrowany element
    - wykorzystuje interpolację wielomianową - secret traktujemy jak wielomian stopnia k-1
- uśrednienie masek wymaga informacji od co najmniej k klientów, ale że $k < n$, to możemy uzyskać uśredniony model bez informacji od wszystkich węzłów

Differential privacy:

- indywidualne modele mogą się przeuczyć, a w przypadku ekstremalnego overfittingu zwrócony model byłby praktycznie równoważny danym
- differential privacy polega na dodaniu losowego szumu (typowo rozkład Gaussa lub Laplace'a) do modeli
- szum znosi się przy uśrednianiu modeli (bo rozkład jest wycentrowany i jednakowy dla wszystkich klientów), ale utrudnia identyfikację pojedynczych aktualizacji

Biblioteki do uczenia federacyjnego:

- TensorFlow Federated (TFF) - framework do TensorFlowa umożliwiający symulowany federated learning, nie umożliwia faktycznego uczenia federacyjnego
- PySyft - ogólny framework do federated learningu, umożliwia nie tylko prywatność danych ale też treningu (niezaufany klient nie wie, co trenuje)
- Flower.dev - framework ogólnego przeznaczenia do obliczeń federacyjnych, oparty o gRPC, niezależny od biblioteki i języka