

ADZD

Materiały

Opracowanie Morgula

[opracowanie_2021_2022.pdf](#)

▼ Egzamin 2020/2021

Odpowiedzi miały się zmieścić na stronie A4

- 1. Omów zalety i ograniczenia przetwarzania równoległego dla analizy dużych zbiorów danych. Podaj przykłady metryk wydajności.**
- 2. Scharakteryzuj i porównaj 2 wybrane modele przetwarzania dużych zbiorów danych, oceń ich przydatność i ograniczenia.**
- 3. Podaj i omów 4 przykłady mechanizmów odporności na awarie oraz optymalizacji w poznanych środowiskach analizy dużych zbiorów danych.**

▼ Egzamin 2021/2022

[egzamin_termin_1_2021_2022.pdf](#)

▼ Egzamin 2022/2023

1. Spark:

- a) Różnice między transformacją a akcją w Sparku
- b) jak odtworzyć RDD
- c) porównanie DataFrame i Spark SQL
wady i zalety
- d) jak Java i Python współpracują w Sparku

2. MapReduce

- a) podaj pseudokod word count w MP
- b) wyjaśnij dlaczego sortowanie jest banalne do implementacji w MP
- c) jak radzić sobie z stragglers
- d) opisz architekturę/strukturę typowego MR

3. Pregel,

- a) Co to jest bulk synchronous parallel
- b) Jakie kroki wykonuje każdy wierzchołek
- c) Jaki jest warunek stopy w Pregelu
- d) Podaj przykład algorytmów grafowych w pregelu

4. Porównaj GFS z DynamoDB (ale dynamoDB nie było na tym przedmiocie, więc finalnie 'opisz GFS')

cos takiego bylo

forma pisemna

po jednej stronie A4 na pytanie



Nagrania 2023

- 09.11 ([Streaming pt 1](#))
- 21.12 ([ML - Intro](#))
- 11.01 ([ML - Rekomendacje](#))
- 18.01 ([ML - Regresja](#))
- Reszta na MsTeams

Opracowanie

▼ 1. Podaj i omów 4 przykłady mechanizmów odporności na awarie oraz optymalizacji w poznanych środowiskach analizy dużych zbiorów danych.

Replikacja

Odporność na awarie

- Prosta redundancja przez stworzenie jednej lub kilku kopii danych i trzymanie ich na różnych node'ach, aby w razie awarii jednego nie utracić danych i móc je odtworzyć po naprawie workera.

Optymalizacje

- Replikacja może być użyta do eliminacji straggling tasków, czyli dużych tasków odpalonych na słabych węzłach

Erasure Coding

Odporność na awarie

- Przez podział danych na fragmenty i inteligentne generowanie dodatkowych fragmentów, mamy matematyczny wzór na odtwarzanie zniszczonych fragmentów.
Istnieją różne mechanizmy na różny poziom odporności na zniszczenia,

Optymalizacje

- Jest to bardziej wydajny mechanizm replikacji niż redundancja, ponieważ generuje mniejszy narzut pamięciowy i komunikacyjny.

popularny i prosty to *parity check*, gdzie generujemy jeden dodatkowy fragment jako sumę pozostałych

Rodowód (lineage)

Odporność na awarie

- Przez przechowywanie historii deterministycznych transformacji danych, możemy w razie awarii węzła odtworzyć utracony i przetworzony fragment danych.

Optymalizacje

- Przechowywujemy jedynie rodowód a nie pełną kopię danych, co znacznie redukuje narzut pamięciowy. Dodatkowo, rodowód jest leniwym mechanizmem, wartościujemy go jedynie w ostateczności, dzięki czemu mając informację o sekwencji transformacji możemy ją zoptymalizować przez wykonanie w odpowiedniej kolejności.

Punkty kontrolne (Checkpointing)

- Okresowe zapisywane stanu rozproszonych obliczeń pozwala na wznowienie obliczeń po awarii od ostatniego zapisu, a nie od samego początku.
- Ogranicza ilość wykonywanych obliczeń po awarii. Nie musimy liczyć od samego początku, a od ostatniego checkpointu.

▼ 2. Scharakteryzuj i porównaj 2 wybrane modele przetwarzania dużych zbiorów danych, oceń ich przydatność i ograniczenia. (Porównaj modele obliczeń MapReduce i Apache Spark)

Map Reduce

Historyczny wcześniejszy model przetwarzania danych, zaproponowany przez Google. Jest zbudowany w oparciu o pamięć dyskową i dzieli przetwarzanie danych

na dwie główne fazy *Map* oraz *Reduce*. *Map* przetwarza dane jako seria par klucz-wartość i wykonuje transformacje na danych. Następnie *Reduce* zbiera wyniki z *Map* i agreguje w ostateczny wynik.

Zalety

- **Opłacalny** - działa w oparciu o *Commodity Hardware* i nie wymaga specjalistycznego sprzętu
- Dobrze się **skaluje** i jest idealny do przetwarzania batchowego
- **Odporny na awarie** - przy awarii node'u, powtarzane są tylko te obliczenia które były na nim przetwarzane

Ograniczenia

- Przez korzystanie z pamięci dyskowej, intensywnie korzysta z **operaci I/O** co powoduje bottelnecks
- Nie nadaje się do **iteracyjnych** obliczeń, często występujących w algorytmach uczenia maszynowego
- Nie nadaje się do **interaktywnych** obliczeń i szybkiej analizy, ponieważ wszystko dzieje się na dysku co powoduje duże *latency* (opóźnienia).
- Mniej wysoko poziomowych abstrakcji, programista musi sprowadzić cały kod do prostych operacji *Map* oraz *Reduce*, co może być **skaplikowane**

Apache Spark

Powstał aby usprawnić i pokonać ograniczenia Map Reduce jako oprogramowanie Open Source. Wykonuje obliczenia w pamięci wykorzystując do tego abstrakcję Resilient Distributed Dataframes (RDD). Generuje acykliczny graf obliczeń (Lineage), dzięki czemu może odtworzyć utracone obliczone dane, a także pozwala optymalizować sekwencje zapytań. Leniwe obliczanie *transformacji* i materializowanie Data Frames przez *akcje* dopiero w ostateczności co redukuje ilość obliczeń.

Zalety

- **Szybsze obliczenia** dzięki modelowi obliczeń w pamięci RAM
- Bardziej **wszechstronny** model, oprócz obliczeń batchowych nadaje się do obliczeń iterowanych, przetwarzania danych strumieniowych oraz

interaktywnych zapytań.

- **Wysokopoziomowe API** np do zapytań SQL, napisane w wielu językach (Scala, Java, Python)

Ograniczenia

- Wyższe **wymagania sprzętowo pamięciowe** związane z obliczeniami *in-memory*
- Ograniczenia związane z przetwarzaniem danych w **reżimie real-time**. Chociaż spark Streaming pozwala na przetwarzanie strumieniowe, to architektura sparka nie jest optymalna dla tego typu danych
- Bardziej **skąplikowane API** i stroma krzywa uczenia. Różnorodność Sparka wydłuża czas efektywnego opanowania tej technologii, co dla korporacji może oznaczać większy koszt na szkolenia pracowników.

▼ 3. Omów zalety i ograniczenia przetwarzania równoległego dla analizy dużych zbiorów danych. Podaj przykłady metryk wydajności.

Zalety

- **Szybkość i Skalowalność** - zrównoleglenie obliczeń często pomaga je przyspieszyć. Architektura równoległa jest naturalnie przystosowana do skalowania problemu - wystarczy dołożyć więcej rdzeni
 - **Metryka Speedup** mierzy zysk z wykorzystania równoległości. Jest to stosunek czasu przetwarzania na 1 węźle do czasu przetwarzania na n węzłach. (idealnie liniowy)
 - **Metryka Parallel Efficiency** - jest to znormalizowany speedup przez podzielenie przez liczbę węzłów n , pozwala wyrazić przyspieszenie jako procent. Idealnie 100%
- **Wykorzystanie Zasobów** - w przetwarzaniu dużych zbiorów danych wąskim gardłem stają się operacje na pamięci (czytania i zapisywania). Przez

zrównoleglenie zarówno obliczeń jak i pamięci, możemy pozbyć się tego ograniczenia odpowiednio dobierając stosunek dysków do workerów.

- **Metryka procent wykorzystania zasobów** (np czas CPU, pamięć) w czasie przetwarzania równoległego. Im bliżej 100% tym lepiej.
- **Odporność na Awarie** - Równoległe przetwarzanie danych na wielu węzłach ma tę cechę, że w przypadku awarii jednego, jego zadania mogą przejąć pozostałe, przez co cały system może dalej działać. Tak samo w przypadku danych, replikacja pozwala na ich oddtworzenie po utracie części.
 - **Metryka Mean Time Between Failures (MTBF)** mierzy średni czas bezawaryjnego działania systemu

Ograniczenia

- **Narzut Komunikacyjny** - przetwarzanie równoległe wymaga komunikacji między częściami systemu (Worker-Worker albo Master-Worker). Algorytmy dobrze zaprojektowane redukują komunikację do minimum, jednak pewien narzut zawsze jest obecny.
 - **Metryka funkcji narzutu** mierzy różnicę całkowitego łącznego czasu wykonania na wszystkich procesorach i czasu wykonania programu optymalnym algorytmem sekwencyjnym.
- **Złożoność Systemu** - projektowanie systemów równolegle przetwarzających dane jest zadaniem złożonym i wymagającym specjalistycznej wiedzy. Nie wszystkie algorytmy dają się w prosty sposób zrównoleglić, a w przypadku ewentualnej awarii, debugowanie systemu równoległego jest zadaniem dużo bardziej skomplikowanym i przez to czasochłonnym.
 - **Czas wytwarzania oprogramowania** może być dobrą metryką pośrednią odzwierciedlającą złożoność systemu równoległego
- **Zrównoważenie Obciążenia** - przy projektowaniu systemów przetwarzania równoległego trzeba zadbać o równomierne obciążanie workerów, aby uniknąć sytuacji że niektóre workery są bezczynne i marnotrawią czas. Równoległy podział pracy między workery skutkuje krótszym czasem wykonania obliczeń.

- **Metryka Jain's Fairness Index** mówi o tym jak nierówny jest podział zasobów. Wartości bliższe 1 świadczą o tym że obciążenie jest zrównoważone

Scalability! But at what COST

Przy ewaluacji wydajności rozwiązania równoległego bardz dobrą miarą jest metryka **COST** - Configuration that Outperforms a Single Thread. Jak sama nazwa wskazuje, jest to liczba rdzeni która pozwala na uzyskanie rozwiązanie w krótszym czasie niż rozwiązanie sekwencyjne. Jest to o tyle wartościowa miara, że uwzględnia wszystkie narzuty zarówno implementacyjne jak i komunikacyjne roziwiązań równoległych i jednocześnie jest bardzo prosta.

▼ 4. Wyjaśnij działanie każdej linii poniższego zapytania strumieniowego (spośród oznaczonych 1-6), odnosząc się do podstawowych pojęć strumieniowego przetwarzania danych

```
PCollection<KV<Team, Integer>> totals = input
1.     .apply(Window.into(FixedWindows.of(TWO_MINUTES)))
2.         .triggering(
3.             AfterWatermark()
4.             .withEarlyFirings(AlignedDelay(ONE_MINUTE))
5.             .withLateFirings(AfterCount(1))
6.             .withAllowedLateness(ONE_MINUTE))
    .apply(Sum.integersPerKey());
```

1. Stosujemy stałe okna czasowe długości dwóch minut każde (rozłączne, pokrywające całą dziedzinę)
2. Stosujemy mechanizm wyzwalaczy (triggers), mówiących kiedy dane z poszczególnych okien się materializują i są generowane wyniki
3. Stosujemy mechanizm znaków wodnych (watermarks), funkcji określających dla danego czasu przetwarzania P , zakładany najwcześniejszy czas zdarzenia E ,

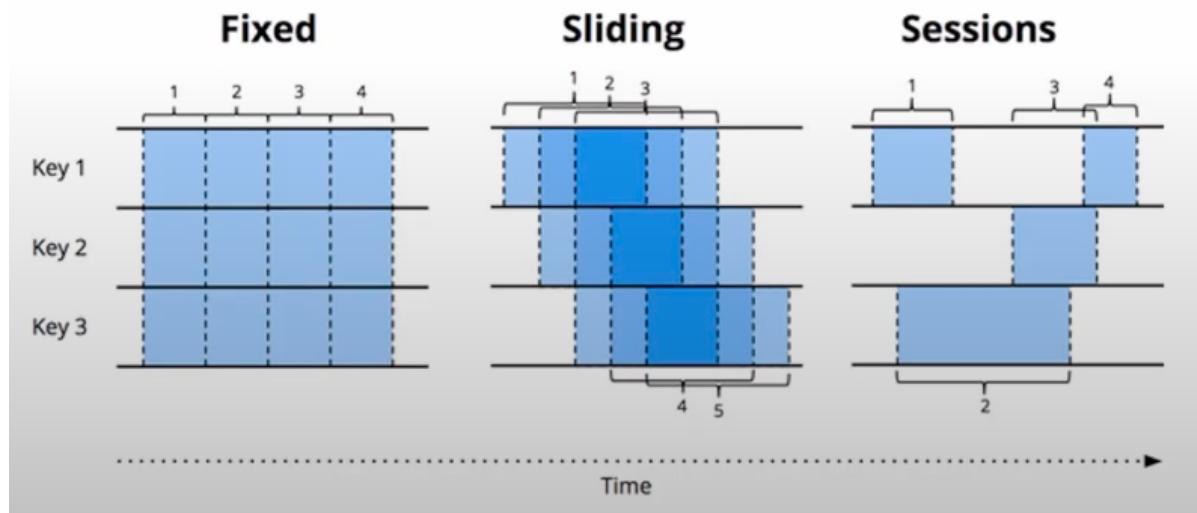
który jeszcze będziemy uwzględniać w analizie. Ustalamy, że w momencie zamknięcia się okna przez watermark zostanie wywołany trigger rapportujący wynik z okna. [on-time]

4. Stosujemy mechanizm emisji wczesnych wyników periodycznie co 1 minutę, synchronicznie między oknami czasowymi (aligned) [early]
5. Stosujemy mechanizm emisji opóźnionych wyników, co każde wystąpienie [late]
6. Określa rozpatrywany horyzont opóźnienia 1 minutę, powyżej którego, wyniki opóźnione nie są rozpatrywane

Punkty 3-6 określają typ *Early/On-time/Late* triggerów, który jest podejściem holistycznym dla mechanizmów watermark oraz triggerów i pozwala na uwzględnienie danych opóźnionych, dając jednocześnie wcześniejszą aproksymację wyników.

Diagramy pomagające w zrozumieniu mechanizmu okien czasowych, triggerów i watermarków:

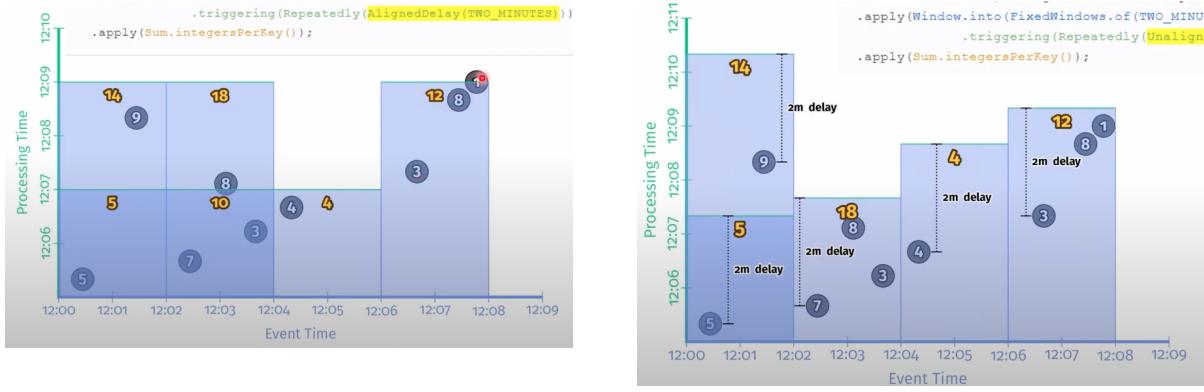
Typy okien czasowych



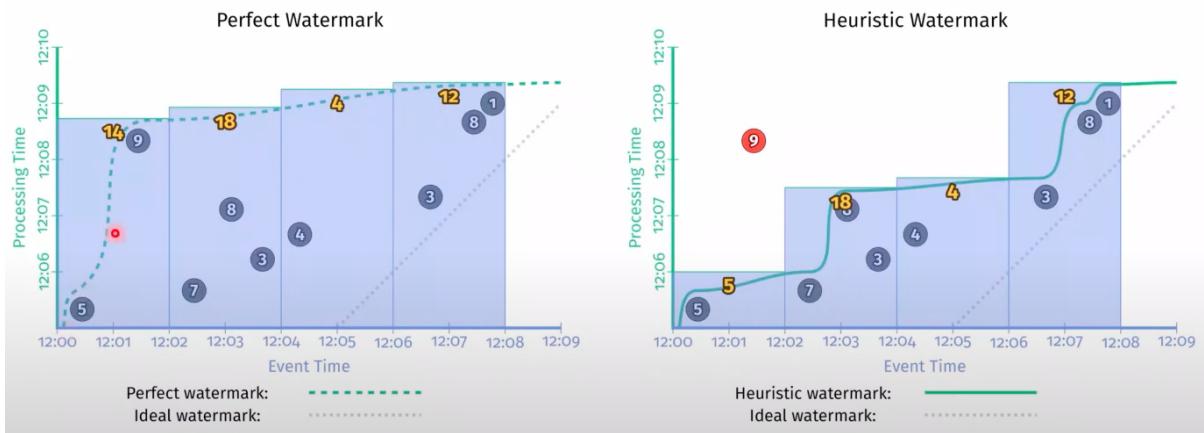
Trigery *Aligned* vs *Unaligned*

Aligned

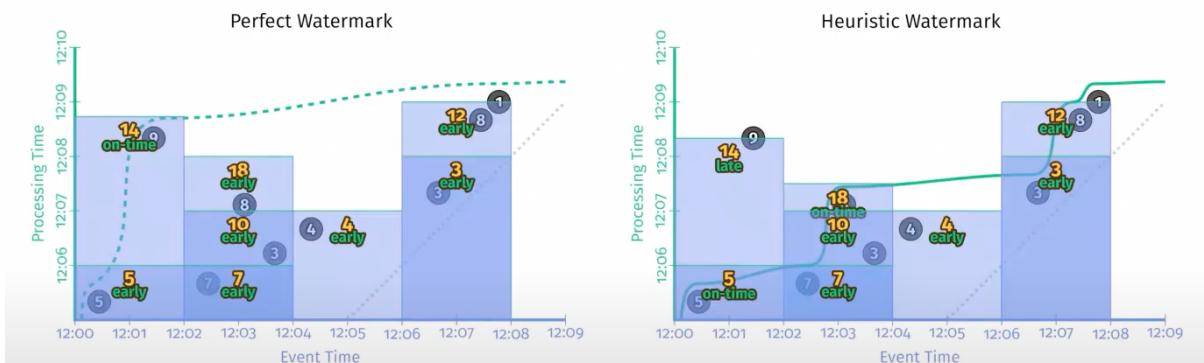
Unaligned



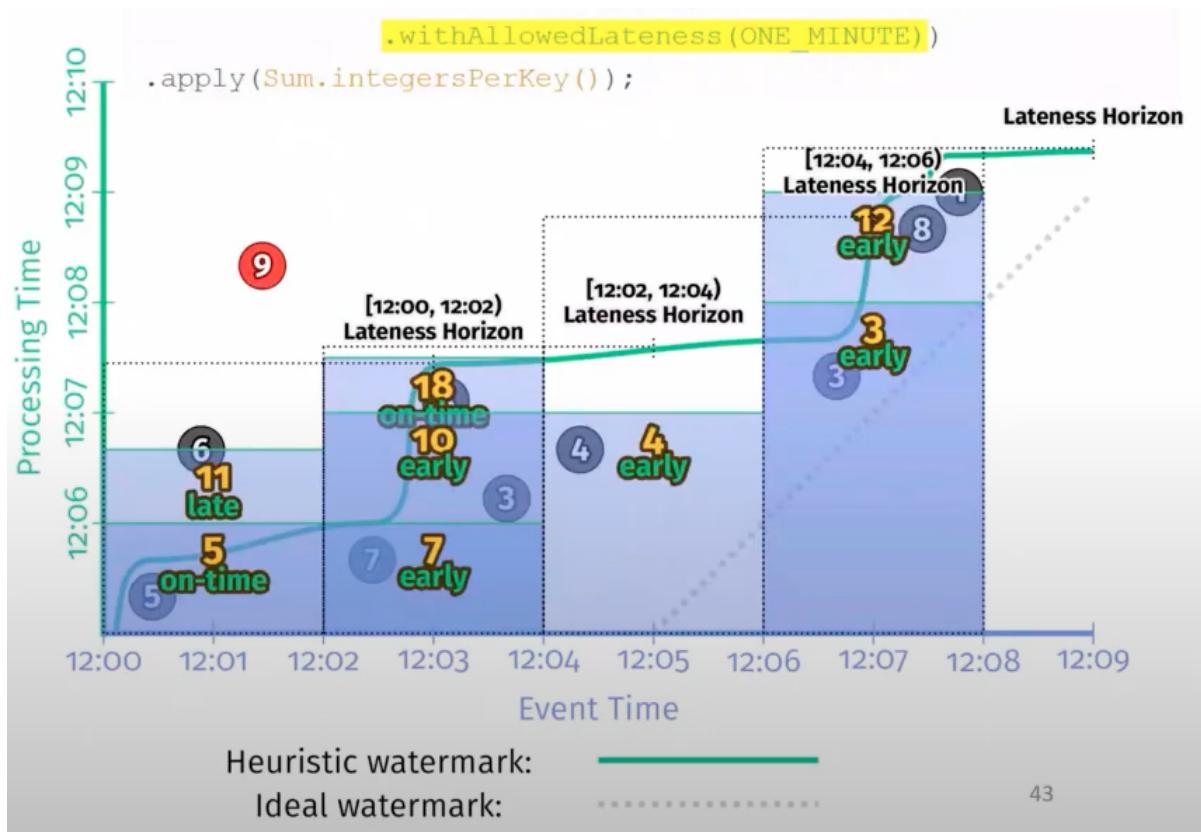
Watermark Perfect vs Heuristic



Trigger Early/On-time/Late



Horyzont Opóźnienia (Lateness horizon)



▼ 5. Na poniższym grafie wykonaj algorytm wyszukiwania maksymalnej wartości wierzchołka według schematu Pregel. W tym celu:

(1) Wypełnij wierzchołki grafu Superkroku 1 różnymi, losowymi wartościami z przedziału 1-99.

(2)
dodaj do grafu jedną nową krawędź (jednokierunkową) pomiędzy dowolnymi wierzchołkami.

(3) Wypełnij stan grafu w kolejnych superkrokach:

(a) wartości w wierzchołkach;

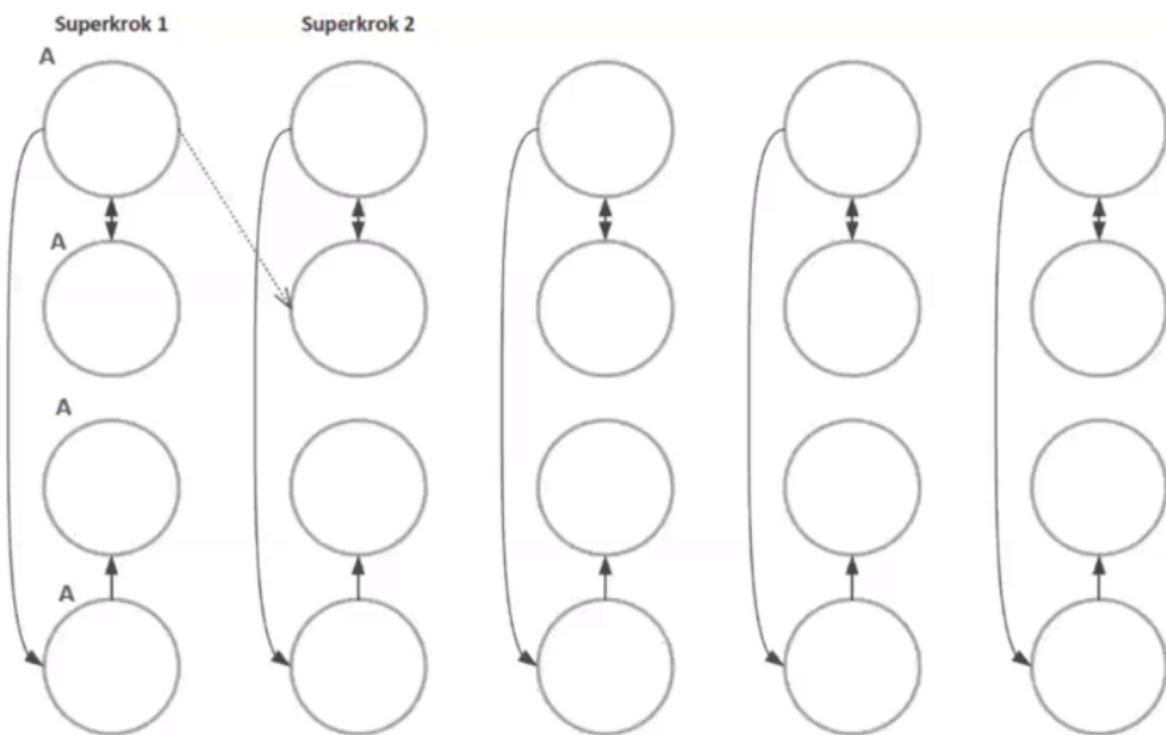
(b) literą

A oznacz aktywne wierzchołki (tak jak oznaczono w Superkroku 1, w którym wszystkie są aktywne);

(c) strzałką przerywaną oznacz przepływ wiadomości pomiędzy Superkrokami (przykładowa strzałka została narysowana).

Uwaga: na rysunku zaznaczono 5 Superkroków, niekoniecznie tyle będzie potrzebne.

Rysunek z treści



Przykładowe rozwiązanie

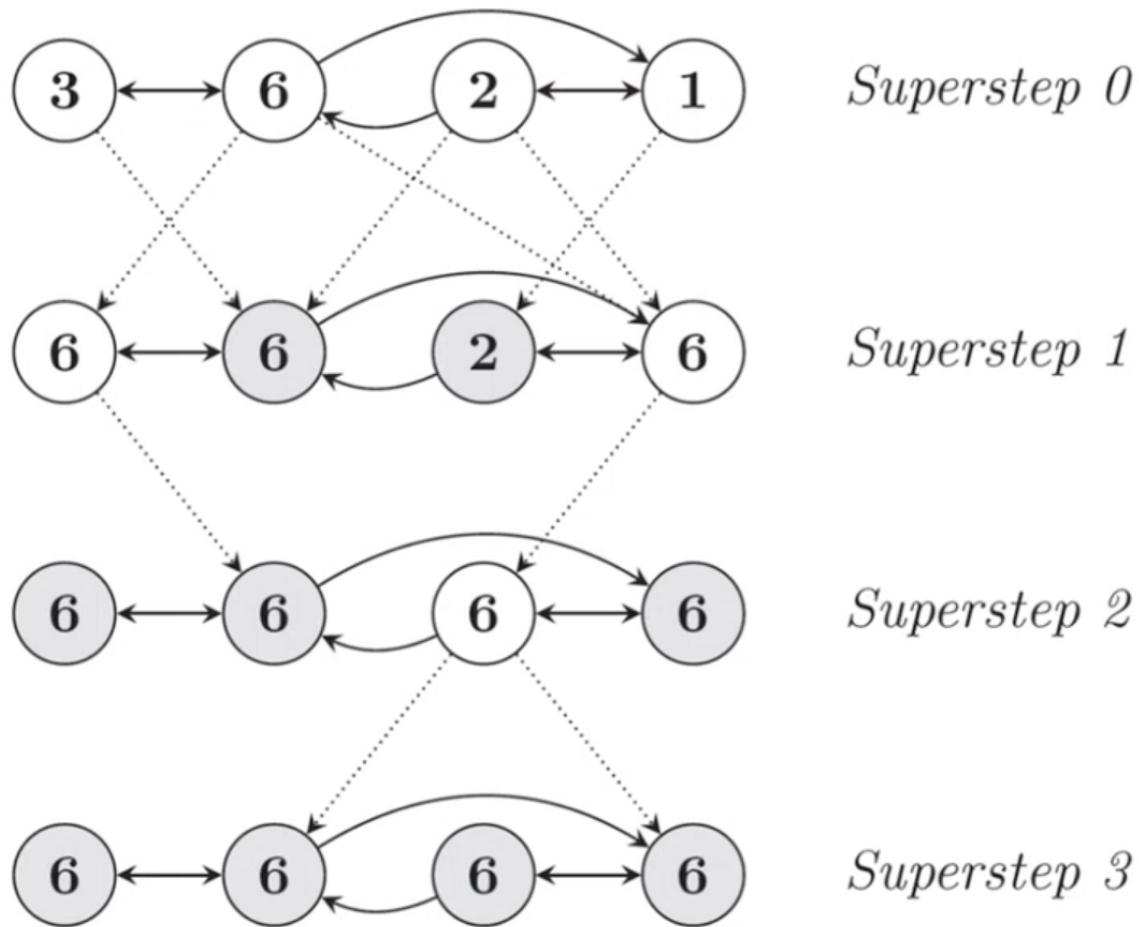
Sytuacja jest troszkę inną (dołączono 2 krawędzie zamiast 1), ale schemat jest taki sam. Na szaro zaznaczono wierzchołki nieaktywne. Każdy wierzchołek wykonuje równolegle ten sam algorytm:

```

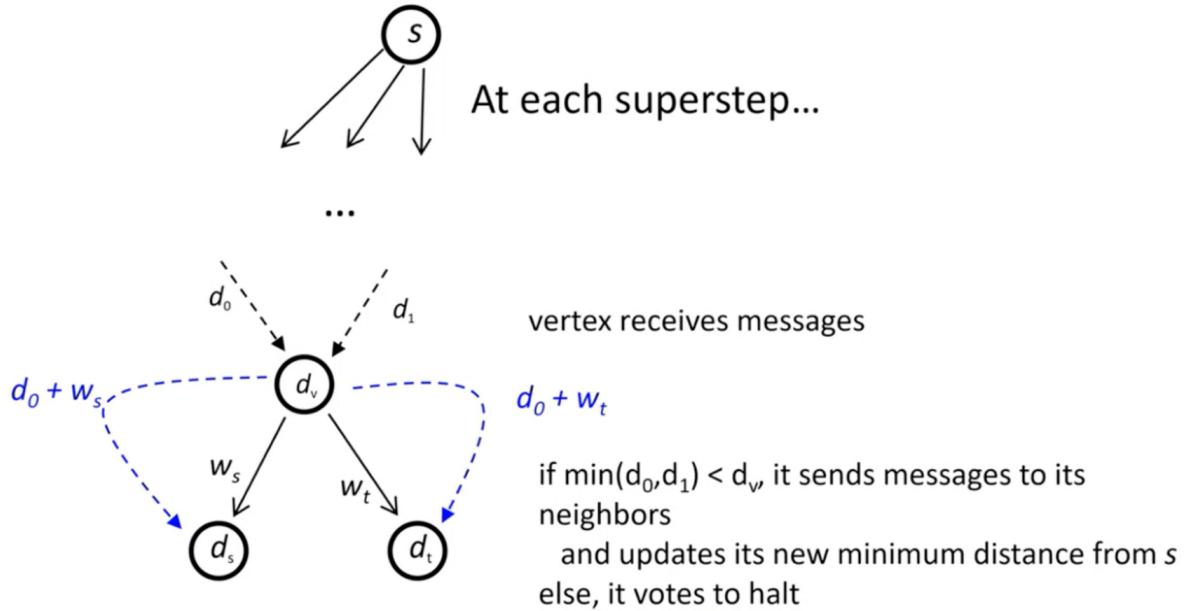
i_val := val
for each message m
    if m > val then val := m
    if i_val == val then
        vote_to_halt
    else
        for each neighbor v
            send_message(v, val)

```

Ewolucja systemu z zaznaczonym *message passing* oraz aktywnością stanów:



Example: Parallel BFS in Pregel



Example: Page Rank in Pregel

The Page Rank Algorithm

The PageRank algorithm outputs a probability distribution representing the likelihood that a person randomly clicking on links will arrive at any particular page

In practice, the algorithm adds a *damping factor* – the probability that, at any step, a person will continue clicking

$PR(u)$ – PageRank of vertex u
 B_u – set of all vertices linking to u
 $L(v)$ – number of outbound links
 N – total number of vertices

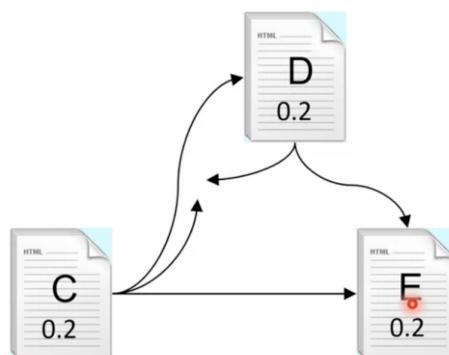
$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)} \quad \text{Without damping factor}$$

$$PR(A) = \frac{1-d}{N} + d \left(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + \dots \right) \quad \begin{array}{l} \text{With damping factor } d \\ \text{Usually } d \text{ is set to } \mathbf{0.85} \end{array}$$

1. Inicjalizowany zbiorem N stron i powiązaniami (linkami) między nimi

2. Każdej stronie przypisujemy startowy rank $\frac{1}{N}$
3. Dla każdej ze stron sukcesywnie aktualizujemy ranki przez zsumowanie ranków każdej z przychodzących stron podzielonych przez liczbę wychodzących z nich krawędzi

$$PR(E) = \frac{PR(C)}{3} + \frac{PR(D)}{2} = \frac{0.2}{3} + \frac{0.2}{2} = \frac{1}{6}$$



4. Jeżeli jakaś strona nie ma wychodzących krawędzi (przypadek strony E), to rozsyłamy i dodajemy do pozostałych wierzchołków jej znormalizowany rank $\frac{PR(E)}{N}$
5. Powtarzamy kroki 3-4 jako jeden superkrok w algorytmie Pregel. Jeśli ranki stron przestaną się zmieniać, to osiągneliśmy zbierzność.

▼ 6. Spark

a) Różnice między transformacją a akcją

Transformacje

Transformacje to operacje które przetwarzają jedne RDD w inne, są leniwie wartościowane, tzn materializowane są dopiero w momencie kiedy wykonamy na nich akcje. Przykładowe transformacje to np. *map*, *filter*, *join*, *union*.

Akcje

Akcje to operacje które zwracają pewną wartość lub exportują dane do systemu pamięci. W momencie wykonania akcji, cały rodowód (lineage) jest materializowany, tzn wykonywane są wszystkie transformacje na RDD poprzedzające wykonanie akcji, i dopiero na koniec wykonywana jest akcja. Np. *collect*, *reduce*, *count*.

b) Jak odtworzyć RDD

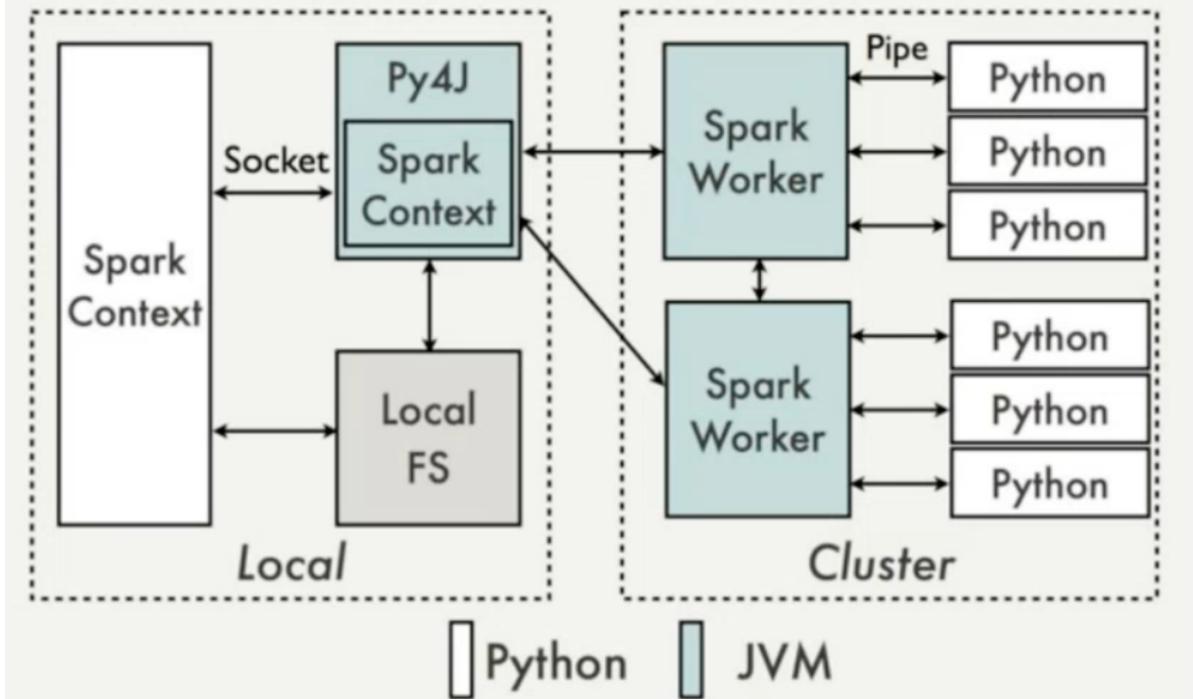
W kontekście awarii workera, aby odwzorzyć utracone RDD, wystarczy wykonać po kolej wszystkie operacje z rodowodu (lineage) danego RDD. Operacje są z założenia deterministyczne, więc otrzymamy wynik identyczny z pierwotnym, utraconym. Dodatkowo, spark zapewnia możliwość checkpointów, co pozwala na odtwarzanie RDD od tego punktu, zamiast od samego początku.

c) DataFrame vs SQL APIs

Aspect	Spark SQL	Spark DataFrames API
Ease of Use	- Familiar SQL syntax for SQL users, but may have a learning curve for non-SQL users.	- Generally easier to learn for beginners, higher-level abstractions, can be easily debuged
Expressiveness	- As expressive as SQL. Complex logic may need to be defined with UDFs	- More expressive for complex data manipulations by supporting a wide range of transformations.
Optimization	- Both uses Catalyst optimizer	- Both uses Catalyst optimizer
Supported Languages	- Limited to SQL operations.	- Accessible from multiple programming languages: Scala, Java, Python, R
Control of Execution	- Limited control over execution plan, as these are abstracted by SQL engine.	- More control over execution plan if needed, however fine-grained control may require working with RDDs.
Use Case	- Well-suited for traditional SQL queries, very efficient with structured data	- More flexible for diverse data processing needs.

d) Jak Java i Python współpracują w Spark?

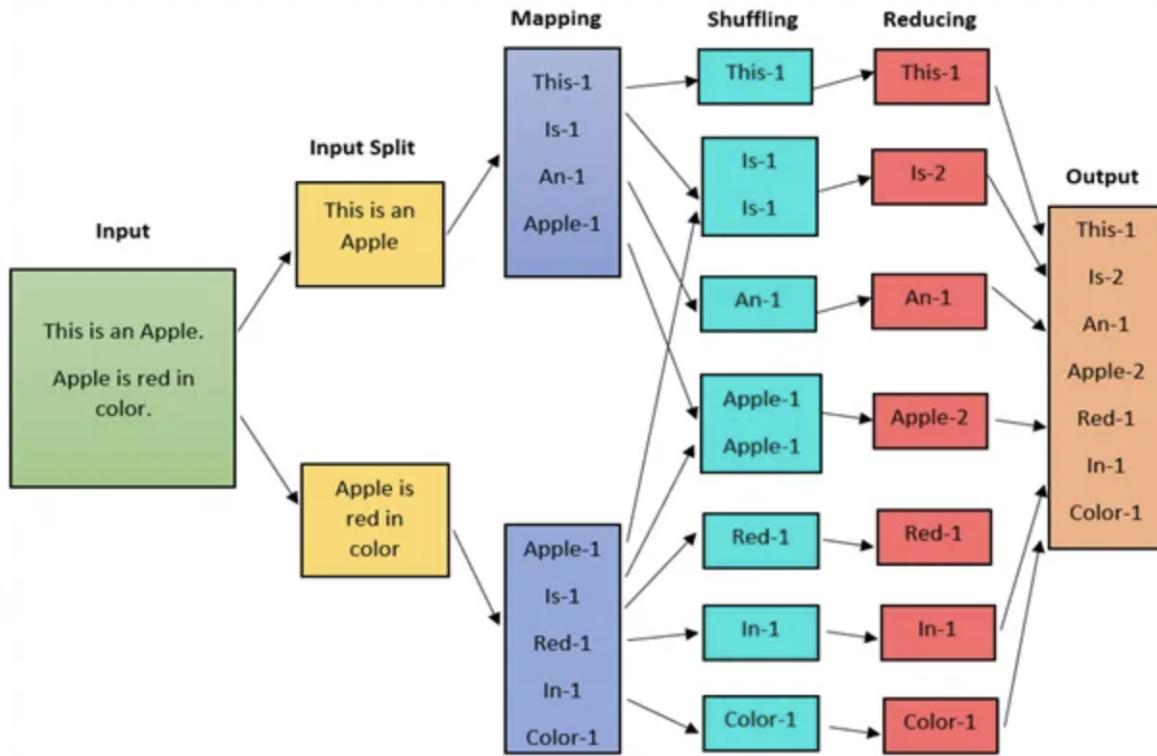
Data Flow



▼ 7. MapReduce

a) Pseudokod word count w MapReduce

1. Rozdziel passage tekstu na wyrazy
2. Mapuj wyraz na parę klucz-wartość: (wyraz, 1)
3. Zgrupuj te same wyrazy
4. Zredukuj grupy wyrazów, przepisując klucz, a dodając wartość
5. Zwróć nową listę kluczy i wartości opisujących liczbę wyst



```

map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));

```

From the original Google article: *MapReduce: Simplified Data Processing on Large Clusters*

b) Dlaczego Sortowanie jest banalne w MapReduce?

Sortowanie jest postrzegane jako trywialne w modelu MapReduce, ponieważ jego wewnętrzne mechanizmy domyślnie sortują wartości (Partition oraz Shuffle-and-

Sort). To znaczy, wykorzystujemy gwarancję kolejności funkcji partycjonującej, czyli

$$k1 < k2 \Rightarrow p(k1) < p(k2)$$

c) Jak radzić sobie ze *Stragglers*?

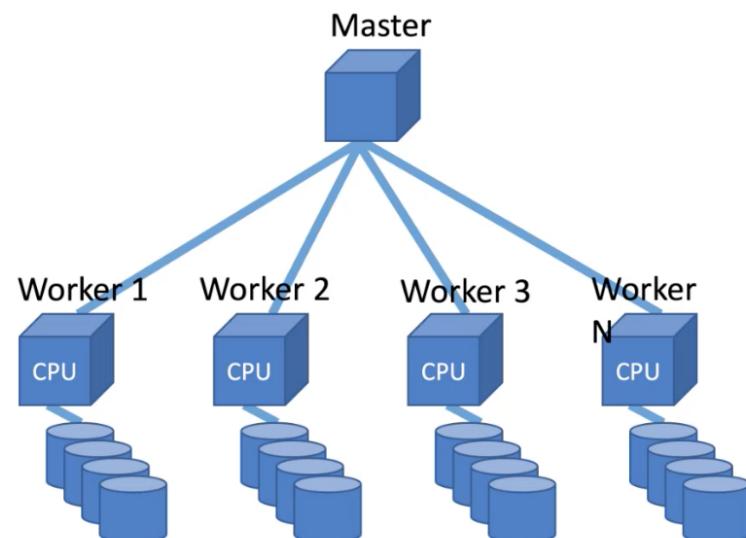
Straggler (gapowicz, włóczęga), to duży task który odpalił się na słabym węźle i idzie trzeba na niego czekać. Aby pozbyć się tego problemu, odpalamy taska na kilku węzłach, bierzemy wynik z tego który policzy się najszybciej, a resztę ubijamy.

d) Opisz architekturę/strukturę typowego MapReduce

Architektura

Architecture of MapReduce cluster

- Just a Bunch of Disks (JBOD)
- Commodity hardware
- Distributed storage
- Storage local to CPU
- Move computing to data



Struktura/Execution-flow

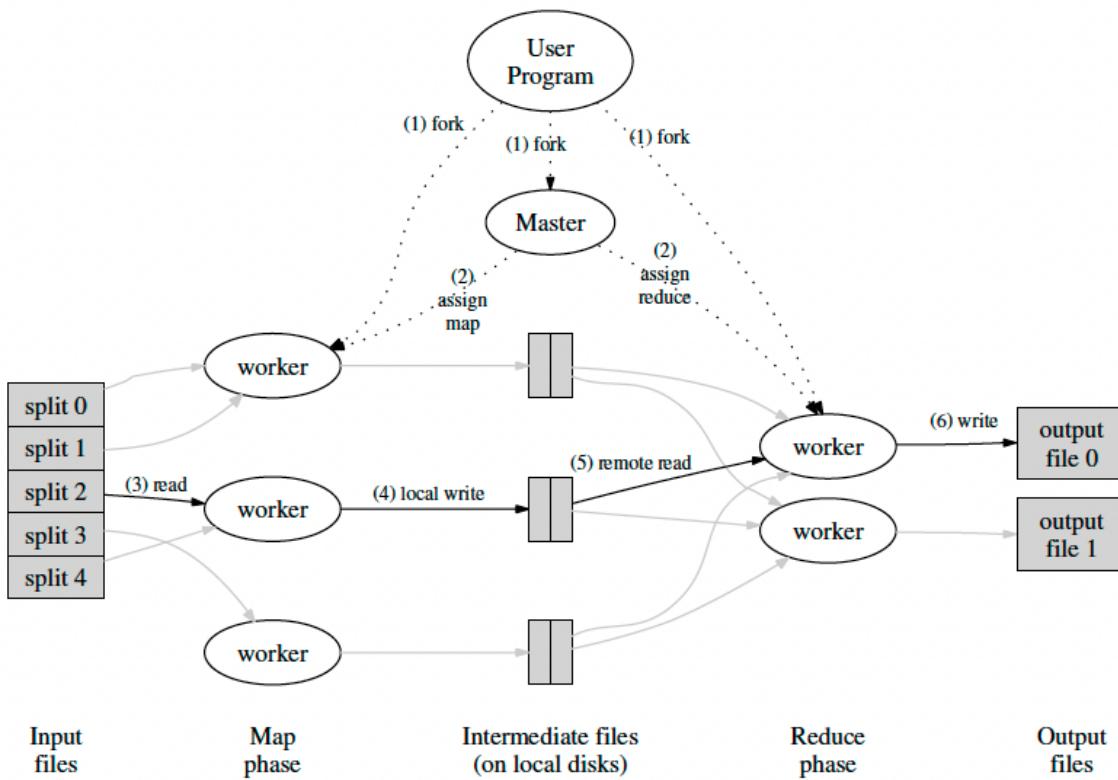


Figure 1: Execution overview

Figure 1: Execution overview (from the original Google article: *MapReduce: Simplified Data Processing on Large Clusters*)

0. Użytkownik definiuje funkcję *map*, która przetwarza wejściową parę klucz-wartość na pośrednią, nową parę klucz-wartość. Definiuje również funkcję *reduce*, która mając klucz i zbiór wartości, zwraca klucz i nową zagregowaną wartość.
1. Biblioteka MapReduce najpierw dzieli pliki wejściowe na M fragmentów
2. Następnie, budzeni są workery oraz mastera. Master przyporządkowuje każdemu wolnemu workerowi jedno z M zadań *Map*, lub jedno z R zadań *Reduce*.
3. Worker któremu przypisano zadanie *Map*, wczytuje odpowiadający fragment z dysku, wykonuje zdefiniowaną przez użytkownika funkcję *map* i buforuje w pamięci wynik.

4. Okresowo, te zbuforowane wyniki są zapisywane przez funkcję partycjonującą na dysk w R obszarów pamięci, a master jest powiadamiany o lokacji zapisu.
5. Kiedy worker *Reduce* jest powiadamiany przez mastera o lokacji danych, czyta je z lokalnego dysku *Map*. Następnie grupuje dane o tym samym kluczu i je sortuje.
6. Worker *Reduce* przebiega po zgrupowanych i posortowanych danych podając parę klucz i grupę odpowiadających wartości do funkcji *reduce*. Zagregowany wynik *reduce* jest dopisywany do finalnego pliku wynikowego tej partycji *Reduce*.
7. Gdy wszystkie zadania *Map* oraz *Reduce* się zakończą, master budzi program użytkownika.

▼ 8. Pregel

a) Co to jest *bulk synchronous parallel*?

Bulk Synchronous Parallel (BSP) to model który zapewnia ustrukturyzowane podejście do projektowania i analizy równoległych algorytmów przetwarzania danych. W oparciu o BSP powstał Pregel. Algorytmy w modelu BSP składają się z sekwencji *Superkroków*, a każdy superkrok składa się z 3 kolejnych po sobie faz:

Faza obliczeń

W tej fazie każdy procesor wykonuje lokalne, niezależne i nie wymagające komunikacji między procesami obliczenia. W modelu Pregel, są to operacje wykonywane na wierzchołkach grafu.

Faza komunikacji

W kolejnej fazie komunikacji, generowane są wiadomości przesyłane między procesami. Dla grafów w modelu pregel, wiadomości wysyłamy po krawędziach.

Faza synchronizacji (Barier synchronization)

W fazie synchronizacji, procesy czekają na siebie nawzajem, aż każdy z nich wykona swoje obliczenia, aby razem wejść do kolejnego superkroku.

b) Jakie kroki wykonuje każdy wierzchołek?

```

i_val := val
for each message m
    if m > val then val := m
    if i_val == val then
        vote_to_halt
    else
        for each neighbor v
            send_message(v, val)

```

0. Wierzchołek w pierwszym superkroku jest inicjalizowany jako
1. Wierzchołek odbiera otrzymane wiadomości
2. Na podstawie wiadomości wykonuje operacje na swoich danych
3. Opcjonalnie przesyła wiadomości do swoich sąsiadów
4. Jeżeli w tym superkroku nie wykonał żadnych akcji, to staje się nieaktywny
5. Gdy wszystkie wierzchołki będą nieaktywne, algorytm się zatrzyma

c) Jaki jest warunek stopu w Preglu?

Aby wszystkie wierzchołki weszły w stan nieaktywny. W stan nieaktywny wierzchołek wchodzi jeżeli nie zmodyfikował swojego kontekstu ani nie rozesłał żadnych wiadomości.

d) Podaj przykłady algorytmów grafowych w Preglu

PageRank

Single Source Shortest Path (SSSP)

Min, Max, Sum itd

Konwolucje (z GNN'ów)?

▼ 9. Porównaj GFS z X

Architektura GFS

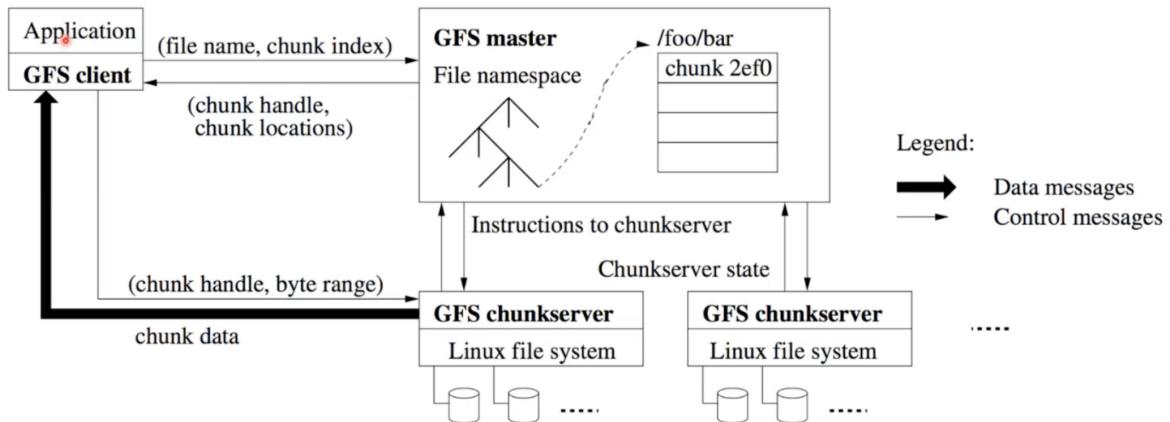


Figure 1: GFS Architecture

Write i data flow w GFS

- Mutation = write or record append
 - Must be done for all replicas
- Goal: minimize master involvement
- Lease mechanism
 - Master picks one replica as primary and gives it a “lease” for mutations (60 sec.)
 - Primary defines a serial order of mutations
 - All replicas follows this order
- Data flow decoupled from control flow

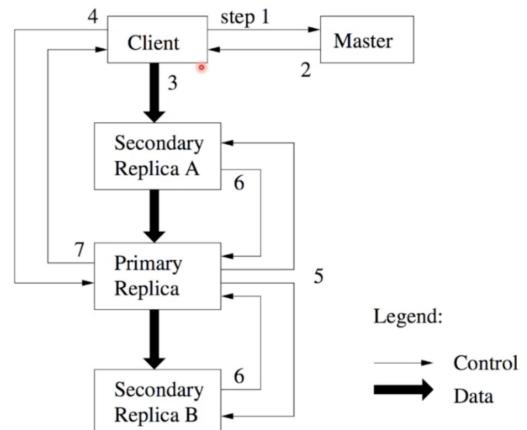


Figure 2: Write Control and Data Flow

Założenia i decyzje projektowe

Assumptions

- High component failure rates
 - Inexpensive commodity components often fail
- Modest number of huge files
 - A few million 100 MB or larger files
- Files are write-once, mostly appended to
- Large streaming reads
- High sustained bandwidth is favored over low latency

Design Decisions

- Files stored as chunks
 - Fixed size (64 MB) *
- Reliability through replication
 - Each chunk is replicated across 3+ chunkservers
- Single master to coordinate access and keep metadata
 - Simple centralized management
- No data caching
 - Little benefit due to large datasets, streaming reads
- Familiar interface but customize the API
 - Snapshot and record append