

# Poprawianie czasu dostępu do dysku



- ⌘ Organizowanie danych w cylindrach
- ⌘ Dzielenie danych na wiele mniejszych dysków zamiast trzymania na jednym dużym
- ⌘ Powielanie danych na kilku dyskach
- ⌘ Algorytmy szeregowania dostępu do dysku
- ⌘ Wstępne ładowanie bloków do pamięci w przewidywaniu ich późniejszego użycia.

# Organizowanie danych w cylindrach

- ⌘ Połowę średniego czasu dostępu do bloku stanowi czas przeszukiwania, dlatego w wielu wypadkach rozsądnie jest umieszczanie w jednym cylindrze tych danych, które będą potrzebne w tym samym czasie (ewentualnie w cylindrach sąsiadujących, jeśli w jednym się nie mieszczą.)
- ⌘ Podczas czytania kolejnych bloków z jednej ścieżki lub cylindra możemy zaniedbać wszystkie opóźnienia obrotowe i czasy przeszukiwania (poza pierwszymi – ustawieniem się nad właściwą ścieżką i nad odpowiednim blokiem)
- ⌘ W przykładzie sortowania dwufazowego określiliśmy 0.15 s jako dostęp do bloku, przy czym:
  - ☒ 0.0005 - przesyłanie bloku, 0.065 – czas przeszukiwania, 0.078 – opóźnienie obrotowe
  - ☒ posortowanie 10 mln rekordów zajęło nam 250 minut. (2\*odczyt i 2\*zapis)
- ⌘ Przy dysku o parametrach wcześniej ustalonych, tj.:
  - ☒ 8192 cylindrów ( $2^{13}$ ),
  - ☒ 8 powierzchni,
  - ☒ średnio 256 sektorów na ścieżce ( $2^8$ ),
  - ☒ 512 bajtów w sektorze
  - ☒ prędkość obrotowa: 3840 br/min = 1 obr ok. 15,6 ms

mamy ok. 1MB w cylindrze (1 048 576 B). Przy rozmiarze rekordu 100B potrzeba 1000 cylindrów do zapamiętania wszystkich danych z poprzedniego przykładu.

# Organizywanie danych w cylindrach

- ⌘ Jedno załadowanie pamięci (50 MB) to odczyt z 50 cylindrów, czyli dane z jednego cylindra można przeczytać w czasie jednego przeszukiwania. Oprócz tego trzeba 49 razy przesunąć głowicę do sąsiednich cylindrów. Szacunkowy czas przesunięcia głowicy o jedną ścieżkę to koszt ok. 1 milisekundy
- ⌘ Zatem czas całkowitego wypełnienia pamięci to:
  - ☒ 0.065 s (czas wyszukania pierwszego cylindra)
  - ☒ 0,049 s (czas przeszukiwania pozostałych cylindrów)
  - ☒ 6,4 s (przesłanie danych z 12 800 bloków )  $12\ 800 * 0.0005 = 6,4$
- ⌘ Poza ostatnim są to czasy pomijalne.
- ⌘ Pamięć wypełniamy 20 razy, czyli ostatecznie  $20 * 6,4s = 2,15$  minuty
- ⌘ To duży zysk w porównaniu z 125 minutami odczytu przy blokach losowo rozrzuconych.
- ⌘ Analogicznie możemy dokonać zapisu do sąsiadujących bloków na 1000 kolejnych cylindrach. Mamy wówczas kolejne 2,15 minut. Razem 4,30
- ⌘ W przypadku fazy 1 dwufazowego wielowejściowego algorytmu sortowania przez scalania można było sobie na taki zabieg pozwolić, bo i tak dane były sortowane w pamięci, więc **kolejność odczytania bloków nie miała znaczenia**. W drugiej fazie bloki czyta się od początku każdej posortowanej listy, więc tu już powyższych oszczędności nie da się uzyskać.

# Korzystanie z wielu dysków



- ⌘ **Eksperyment:** Gdyby zastąpić jeden dysk o 8 powierzchniach czterema dyskami złożonym tylko z jednej płyty (dwóch powierzchni). Innymi słowy nasz dotychczasowy dysk można zastąpić czterema takimi nowymi jednopłytowymi.
- ⌘ Dane można wówczas podzielić na te 4 dyski w ten sposób, by zająć 1000 kolejnych cylindrów na każdym. Przy zapewnianiu pamięci w pierwszej fazie pobieramy po 1/4 ilości danych z każdego dysku.
- ⌘ Czytając równoległe 4 dyski zyskujemy nie tylko to, co zyskaliśmy poprzednio unikając konieczności pozycjonowania głowic, ale także oszczędzamy na czasie zaczytania bloków. Równoległe czytanie 12 800 bloków zajmuje tyle czasu co odczyt 3 200 bloków. Zatem 50 MB pamięci zaczyta się nie jak dotąd w 6,4 sekundy, ale w 1,6 sekundy. Gdy to samo zastosujemy do operacji zapisu to ostatecznie czas fazy pierwszej zmniejszył się do 1 minuty.
- ⌘ W fazie 2 ponownie sprawa nie jest tak samo prosta - wymaga porównania pierwszych bloków z każdej z list. Algorytm wymaga, aby tego porównania dokonać w pamięci, zatem musimy wszystkie bloki z 20 list tam umieścić. Trudno przewidzieć blok z której listy ma zostać przeczytany jako następny, możliwości zrównoleglenia odczytu są więc ograniczone. Możliwe są jednak oszczędności przy zapisie. Można użyć czterech buforów wyjściowych i zapisywać je kolejno. Każdy bufor po zapewnieniu jest zapisywany w kolejnych cylindrach określonego dysku. Można zatem zapierać kolejny bufor, gdy poprzednie 3 są zapisywane.
- ⌘ Szacunkowo druga faza może być szybsza 2 do 3 razy. Przy 125 minutowym wykonaniu fazy 2 jest to jednak nadal spora oszczędność.

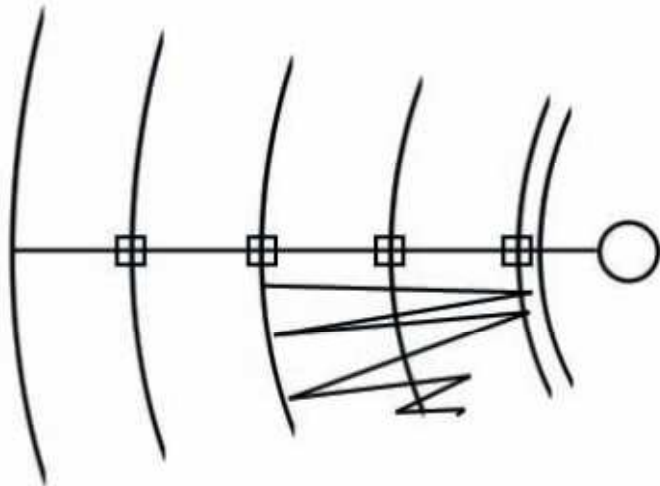
# Lustrzane kopiowanie dysków



- ⌘ W przypadku korzystania z wielu dysków w drugiej fazie, przy odczycie bloków z początku list mogło się zdarzyć tak, że listy, z których miały zostać pobrane bloki znajdowały się na tym samym dysku. Wtedy były zaczytywane kolejno i nie było zysku z podzielenia dysku na cztery. Jeśli użyjemy replikacji to każda lista znajdować się będzie na wszystkich dyskach. Dobrze skonstruowany algorytm poradzi sobie z zadaniem równoległego czytania danych list.
- ⌘ Uwaga: duplikowanie dysków nie przyspiesza zapisu, ale też go nie spowalnia. Zdarzać się mogą różnice w prędkości zapisu na poszczególnych dyskach, ale są one nieistotnie małe (zwłaszcza w przypadku stosowania pierwszej strategii związanej z układaniem danych w cylindrach).

# Algorytmy szeregowania zadań

- ⌘ Jeśli system nie musi czytać danych w określonej kolejności, to warto stosować strategię określania kolejności wykonywania żądań dostępu do danych, jakie otrzymuje sterownik. Ma to znaczenie w przypadku, gdy system obsługuje wiele mniejszych procesów, które sięgają do małej ilości bloków.
- ⌘ **Algorytm windy.** Głowice „omiatają dysk” w dwie strony – od najbardziej wewnętrznego do najbardziej zewnętrznego cylindra i z powrotem. Jeśli pojawia się żądanie dostępu do bloku z cylindra, nad którym akurat głowice się znajdują, to się zatrzymują i wykonywane są żądania dla bloków z tego cylindra (zapis i odczyt). Potem przesuwają się dalej, aż do następnego cylindra z blokami, do których potrzebny jest dostęp. Jeśli w podanym kierunku przesuwu nie ma już żadnych zgłoszeń zapotrzebowania na bloki, to głowica zmienia kierunek.



Zyski mogą być pozorne - W niektórych przypadkach żądania dostępu zostałyby obsłużone wcześniej, gdyby użyto zwykłej metody (FIFO) zamiast algorytmu windy.

Algorytm windy zaczyna dawać wymierne korzyści w momencie, gdy zwiększa się średnia liczba żądań oczekujących na dysk.

# Wstępne ładowanie bloków

- ⌘ W niektórych zastosowaniach możliwa do przewidzenia jest kolejność żądań. Wstępne ładowanie bloków polega na wprowadzaniu ich zawartości do pamięci zanim będzie potrzebna.
- ⌘ Przykład fazy 2 algorytmu dwufazowego wielowejściowego algorytmu przez scalanie był tak skonstruowany by wczytanie początkowych bloków z 20 posortowanych list zostawiało jeszcze dużo wolnego miejsca w pamięci. Można więc poświęcić po dwa bufor do każdej listy i wypełniać jeden w czasie, gdy algorytm wybiera rekordy do scalenia z drugiego bufora. Gdy bufor się wyczerpie, zostaje zastąpiony tym dodatkowym, pełnym, bez opóźnienia i oczekiwania na odczyt z dysku.
- ⌘ **Przykład:** Miejsca w pamięci jest wystarczająco dużo, by utrzymywać 2 bufor dla każdej listy, przy czym każdy bufor będzie miał wielkość jednej ścieżki.
- ⌘ Na każdej ścieżce mieści się 128 KB, więc  $20 \cdot 2 \cdot 128 \text{ KB} = 5120$  (5MB)
- ⌘ Ścieżkę można czytać od dowolnego miejsca, więc nie musimy doliczać opóźnienia obrotowego. Wystarczy czas przeszukiwania (6,5) plus czas potrzebny na jeden obrót dysku (15,6). Razem 22,1 ms.
- ⌘ Do wczytania 20 podlist potrzeba przeczytać wszystkie 1000 cylindrów, czyli 8000 ścieżek.
- ⌘ Całkowity czas: 2,95 s

# Podsumowanie

## ⌘ Typowe sytuacje:

- ☒ (A) bloki odczytywane i zapisywane są w przewidywalnej kolejności, a z dysku korzysta jeden proces
- ☒ (B) Wiele krótkich procesów wykonywanych równolegle i korzystających z tych samych zasobów dyskowych, gdy nie ma możliwości przewidywania do których danych będą sięgały.

## ⌘ Wady i zalety

### ☒ GRUPOWANIE DANYCH W CYLINDRACH

- ☒ (+) doskonale do typu (A)
- ☒ (-) nie poprawia wydajności (B)

### ☒ WIELE DYSKÓW

- ☒ (+) Większa szybkość odczytu i zapisu w obu przypadkach zastosowań
- ☒ (-) Żądania odczytu i zapisu tego samego dysku nie mogą być zapewnione w tym samym czasie.  
Wadą jest także koszt kilku małych dysków (większy niż jednego o pojemności będącej ich sumą)

### ☒ LUSTRZANE KOPIE

- ☒ (+) Zwiększenie zapisu odczytu w obu typach zastosowań. Większa tolerancja błędów.
- ☒ (-) Koszty rosną proporcjonalnie do ilości dysków, a pojemność się nie zwiększa

### ☒ ALGORYTM WINDY

- ☒ (+) Skrócenie średniego czasu odczytu i zapisu bloków w sytuacji nieprzewidywalnych żądań dostępu
- ☒ (-) Algorytm wykazuje swoją efektywność dopiero przy dużej liczbie oczekujących żądań dostępu.

### ☒ WSTĘPNE ŁADOWANIE

- ☒ (+) Większa szybkość dostępu, gdy potrzebne bloki są znane, ale synchronizacja zależy od danych (tak jak w drugiej fazie sortowania)
- ☒ (-) Potrzebne dodatkowe bufony pamięci, nie zawsze dostępne



# Awarie dysku



## ⌘ Typy awarii:

- **Zakłócenia sporadyczne** (jedna lub kilka nieudanych prób zapisu lub odczytu sektora, po których zapis/odczyt zostaje wykonany poprawnie.
- **Uszkodzenie nośnika** Trwałe uszkodzenie jednego lub większej ilości bitów (poprawne odczytanie sektora staje się niemożliwe)
- **Błąd zapisu** – nie udaje się zapisać sektora, ani odczytać sektora zapisanego poprzednio.
- **Uszkodzenie dysku** – cały dysk niemożliwy do odczytu

# Zakłócenia sporadyczne

## ⌘ Odczyt:

- ⌘ Bity fizycznie odczytywane za pośrednictwem sterownika dysku są powiązane ze sobą pewną dodatkową informacją, dzięki której można określić poprawność dokonanego odczytu.
- ⌘ Odczyt z dysku można więc opisać parą ( $w, s$ ), gdzie  $w$  jest to dana odczytana z sektora, a  $s$  jest bitem stanu określającym, czy odczyt się powiódł, tzn. czy  $w$  jest poprawną zawartością sektora.
- ⌘ Przy zakłóceniach odczyt ponawiany jest kilkakrotnie (np. do 100 razy), aż status odczytu będzie poprawny.
- ⌘ **Może się zdarzyć, że status będzie poprawny, a dane jednak poprawne nie są.**

## ⌘ Zapis:

- ⌘ Najprostsza kontrola zapisu polegałaby na zapisaniu sektora, odczytaniu go i sprawdzeniu, czy dane są zgodne. Szybsze jest jednak sprawdzanie statusu odczytu opisanego powyżej. Powstaje analogiczna sytuacja uznania za poprawny zapisu, który się nie powiódł.

# Sumy kontrolne

- ⌘ Każdy sektor zawiera dodatkowe bity (**sumę kontrolną**), których wartość zależy od bitów danych pamiętanych w sektorze. Przy niezgodności status odczytu określa się jako niepoprawny.
- ⌘ Prawdopodobieństwo trafienia w dobrą sumę kontrolną mimo błędnych odczytów jest niewielkie, ale istnieje.
- ⌘ Postać sumy kontrolnej bazuje na **parzystości**:
  - ☒ Jeśli w zbiorze bitów sektora jest nieparzysta liczba jedynek, to „parzystość jest ujemna” (lub „bit parzystości wynosi 1”).
  - ☒ W przeciwnym wypadku bity mają „dodatnią parzystość” (lub „bit parzystości wynosi 0”).
- ⌘ W efekcie liczba jedynek zapisywanych do sektora zawsze jest parzysta i bit parzystości jest zawsze dodatnia.

01101000 -> 01101000 **1**

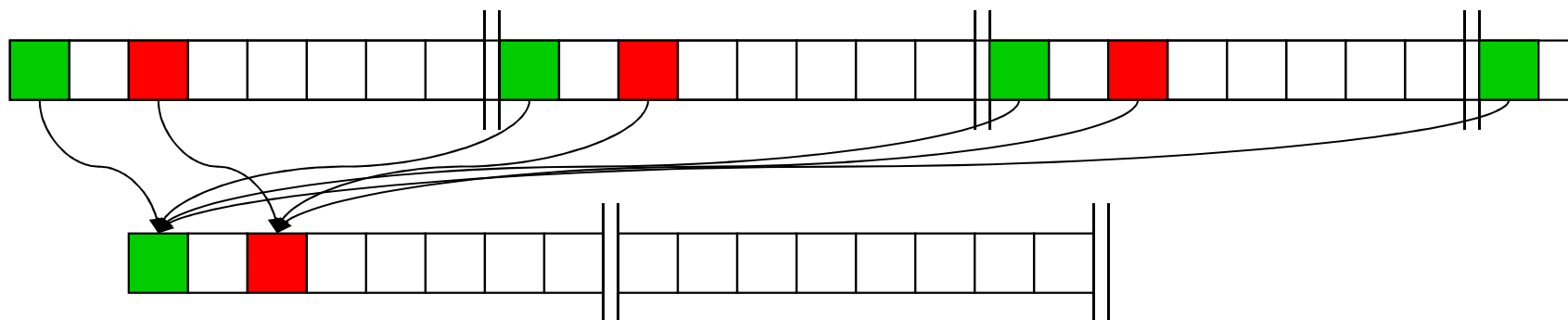
11101110 -> 11101110 **0**

- ⌘ Każdy błąd przy zapisie lub odczycie bitu powoduje powstanie parzystości ujemnej.
- ⌘ Sterownik sprawdza „w locie” liczbę jedynek w sektorze.
- ⌘ Co się dzieje, że jeśli w sektorze bitów uszkodzonych jest więcej?

11101110 -> 1100110 **0**

# Sumy kontrolne

- ⌘ Można zwiększyć szansę wykrycia przez większą liczbę bitów parzystości.
- ⌘ Na przykład utrzymywanie 8 bitów parzystości (po jednym dla każdego pierwszego, drugiego, trzeciego... bitu w bajtach )
- ⌘ Prawdopodobieństwo niewykrycia tego, że wystąpił błąd wynosi tylko 1 przez  $2^8$
- ⌘ Ogólnie - używając  $n$  bitów do kontroli parzystości otrzymujemy  $1 / 2^n$  szans na niezauważenie błędu.
- ⌘ Gdyby stosować na każdy sektor 3 bajty kontrolne są to szanse 1 : 4 miliardy



# Przechowywanie stabilne



- ⌘ Sama świadomość faktu błędu w odczycie/zapisie to za mało - potrzebna jest możliwość przeciwdziałania.
- ⌘ **Przechowywanie stabilne** polega na kompletowaniu par sektorów. Każda para jest przeznaczona do przechowywania zawartości  $X$ , która można zmieścić w jednym sektorze.
- ⌘ Oznaczmy  $X_L$  i  $X_R$  - „lewą” i „prawą” kopię wartości  $X$ .
- ⌘ Zakładamy, że kopie zapisujemy z taką kontrolą parzystości (na wystarczającej ilości bitów), że możemy wyeliminować ryzyko, że zły sektor wygląda jak dobry.

# Przechowywanie stabilne



## ⌘ Strategia zapisu:

- Zapisać  $X$  do  $X_L$ .
- Sprawdzić stan.
- Jeśli jest poprawny, to w kopii  $X_L$  bity parzystości są poprawne.

## ⌘ Jeśli nie - powtórzyć zapis.

- Jeśli po serii prób nie udało się poprawnie zapisać  $X$  do  $X_L$  to przyjmujemy, że w  $X_L$  wystąpiło uszkodzenie nośnika. Wtedy trzeba dokonać naprawy przez zastąpienie uszkodzonego sektora innym wolnym obszarem dysku.

## ⌘ Wykonać wszystkie czynności dla $X_R$ .

## ⌘ Strategia odczytu:

- Odczytać  $X_L$  jako wartość  $X$ .
- Jeśli stan jest niepoprawny, to powtórzyć kilka razy.
- jeśli się uda, to stan poprawny,
- jeśli nie to wykonać to samo dla  $X_R$ .

# Przechowywanie stabilne - korzyści

## ⌘ Uszkodzenie nośnika.

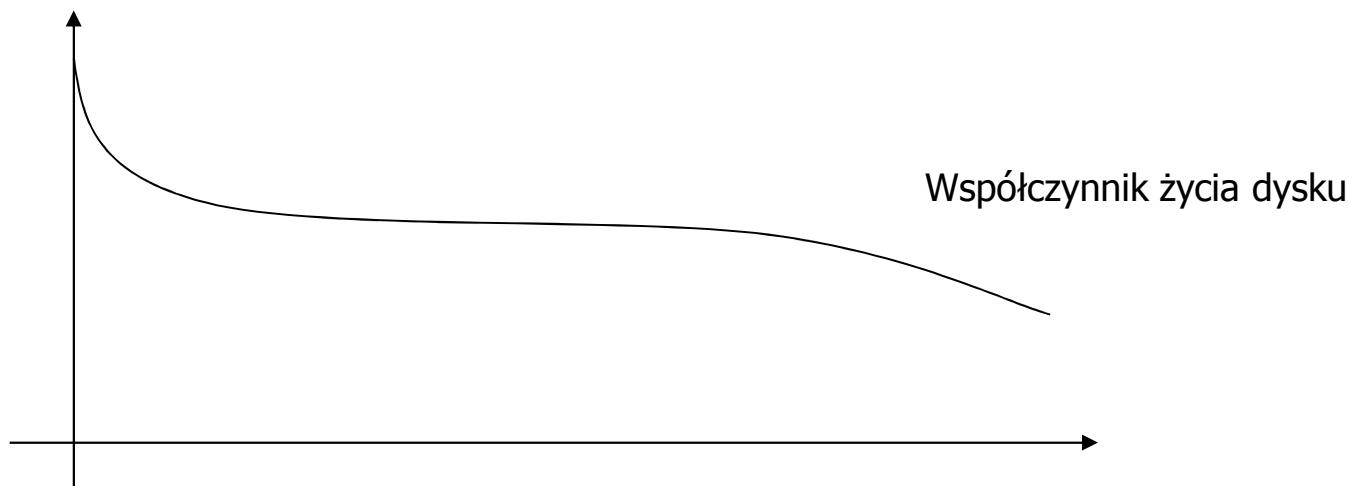
- ☒ Jeśli po zapamiętaniu wartości  $X$  w  $X_L$  i  $X_R$  w jednym z nich wykryte zostanie uszkodzenie nośnika, to można poprawną wartość uzyskać z drugiej kopii.
- ☒ Jeśli obie zostaną zniszczone, to nie ma możliwości odzyskania, ale prawdopodobieństwo jest znikome, że równocześnie utracimy obie kopie równocześnie.

## ⌘ Błąd zapisu (np. przy zaniku zasilania) .

- ☒ Tracimy wtedy na pewno wartość  $X$ , która znajdowała się w pamięci.
- ☒ Jeśli nastąpi to podczas zapisywania w  $X_L$  (można założyć, że zapis został przerwany w połowie zapisu danych sektora) to po ponownym uruchomieniu systemu w  $X_R$  mamy poprawną **starą** kopię wartości. Można więc odtworzyć nie tylko starą wartość, ale również naprawić wartość  $X_L$ .
- ☒ Jeśli zapis do  $X_L$  się zakończył, to ma poprawny **nowy** stan kopii  $X_L$ . Należy wtedy wyrównać zawartość, tak aby również  $X_R$  było poprawne.

# Odzyskiwanie danych po awarii dysku

- ⌘ W przypadku nieodwracalnego zniszczenia dysku z pomocą przychodzą systemy macierzy dysków.
- ⌘ **SYSTEMY RAID (Redundant Arrays of Independent Disks) – czyli Redundantna Matryca Niezależnych Dysków.**
- ⌘ **Średni czas przedawaryjny:** czas w którym 50% dysków danej populacji ulega awariom katastrofalnym (np. uszkodzenia głowic).
- ⌘ Nowoczesne dyski mają czas przedawaryjny około 10 lat.
- ⌘ Awaryjność dysków nie jest liniową funkcją czasu.
- ⌘ Najwięcej awarii wykrywanych jest w początkowym etapie użytkowania dysku (defekty w produkcji), zwykle jeszcze przy testach producenta, zanim zostaną oddane do użytku.





# Odzyskiwanie danych po awarii dysku



- ⌘ **dyski z danymi** - podstawowe dla systemu
- ⌘ **dyski redundancyjne** - służące do odtwarzania po awarii.
- ⌘ system **RAID 1** - najprostsza metoda (opisana wcześniej przy technikach poprawiania czasu dostępu do dysku jako **lustrzane kopiowanie danych**). W tym podejściu nie ma rozróżnienia na to, który dysk jest z danymi, a który redundancyjny. Ilość dysków redundancyjnych jest równa ilości dysków z danymi.
- ⌘ **Średni czas do utraty danych** jest tu dużo większy niż **średni czas przedawaryjny**. Właściwie jedyna możliwość utraty danych to wystąpienie równoczesne awarii dysków redundanтных z dyskami danych.
- ⌘ system **RAID 4** - korzysta tylko z jednego dysku redundancyjnego bez względu na ilość dysków z danymi.
- ⌘ Wszystkie dyski z danymi są identyczne (o tych samych parametrach).
- ⌘ na dysku redundancyjnym i-ty blok składa się z kontroli parzystości i-tych bloków dysków z danymi. Dyski, jako identyczne, mają te same ilości bitów w blokach.

# System RAID 4

⌘ **Przykład:** Pierwsze bajty bloków na dyskach z danymi:

D1: 11110000

D2: 10101010

D3: 00111000

Wówczas na redundancyjnym:

D4: 01100010 (**suma modulo 2**)

⌘ **Odczyt:** Zawartość dowolnego bloku z dysku można ustalić na podstawie pozostałych dysków i dysku redundancyjnego. (Może się to przydać również do poprawiania wydajności, choć taka sposobność rzadko się zdarza - podczas czytania jednego z dysków, gdy jest on zajęty, zawartość kolejnego bloku może zostać obliczona za pomocą odczytów pozostałych dysków, które akurat są wolne.

⌘ **Zapis:** Oprócz zmian na jednym z dysków z danymi konieczne jest dokonanie zmian na dysku redundancyjnym.

⌘ „naiwne” podejście z przeliczaniem poprzez odczytanie bloków z pozostałych dysków byłoby mało wydajne !

⌘ Realizowane jest to w sposób szybszy, dzięki własnościom dodawania modulo 2, przez:

- Obliczenie sumy starej i nowej wartości zmienianego bloku
- Wykonanie zmian na odpowiednich pozycjach dysku redundancyjnego

# System RAID 4

D1: 11110000  
D2: 10101010  
D3: 00111000

⌘ Zmiana w bloku na dysku D2: 10101010 → D2': 11001100

D2 : 10101010  
D2': 11001100  
-----  
01100110

⌘ Odczyt dysk redundancyjnego

D4: 01100010

i zmiana na wartość przeciwną tych pozycji, na których suma D2 i D2' miała jedynkę

→ D4': 00000100

⌘ Sprawdźmy:

D1: 11110000  
D2': 11001100  
D3: 00111000  
-----  
D4': 00000100

# System RAID 4

## ⌘ Naprawa po awarii

- ☑ Jeśli uszkodzony został dysk redundacyjny, to po wymianie informacje są na nim przeliczane dane na podstawie dysków z danymi.
- ☑ W przypadku utraty dysku z danymi sytuacja wygląda dokładnie tak samo.

D1:	11110000
D2:	????????
D3:	00111000
D4:	01100010
-----	
	10101010

## ⌘ Co w sytuacji, gdy ulegną awarii równocześnie 2 dyski ?

RAID 4 nie poradzi sobie z taką sytuacją

# System RAID 5

- ⌘ W systemie RAID 4 dysk redundancyjny może stać się „wąskim gardłem”, ponieważ cokolwiek zmieniamy na jednym z dysków z danymi, to konieczna jest aktualizacja wartości na dysku redundancyjnym. Dla  $n$  dysków z danymi ilość zapisów na dysku redundancyjnym jest więc średnio  $n$  razy większa niż na którymkolwiek dysku z danymi.
- ⌘ Idea systemu **RAID 5** również wykorzystuje sumowanie modulo 2, ale ponieważ nie ma znaczenia, z których dysków pochodzą sumowane dane, więc w systemie tym nie ustalono jednoznacznie dysku redundancyjnego. Sumy parzystości są rozpraszane po całej macierzy.
- ⌘ **Przykład:**
- ⌘  $n=3$  (dyski 0,1,2,3).
  - Dysk 0 jest redundancyjny **dla cylindrów** 4,8,12, itd. (bo *numer cylindra mod 4 = 0*),
  - Dysk 1 jest redundancyjny dla cylindrów 1,5,9,13 ... ,
  - Dysk 2 jest redundancyjny dla cylindrów 2,6,10 ... ,
  - Dysk 3 jest redundancyjny dla cylindrów 3,7,11 ...

Wyrównuje się w ten sposób obciążenie odczytami i zapisami.

# Obsługa awarii wielu dysków

⌘ RAID 4 i 5 nie chronią przed awarią większej ilości dysków. Można skonstruować macierz, która pozwoli na poprawę tej sytuacji

⌘ Siedem dysków:

D1 – D4 - dyski z danymi

D5 – D7 - dyski redundancyjne

których związek jest zdefiniowany za pomocą macierzy o wymiarach 3 x 7:

Nr dysku	1	2	3	4		5	6	7
	1	1	1	0		1	0	0
	1	1	0	1		0	1	0
	1	0	1	1		0	0	1

⌘ Warunki:

- Każda kombinacja zer i jedynek (z wyjątkiem samych zer) występuje w kolumnach
- W kolumnach redundancyjnych jest tylko po jednej jedynce
- Wszystkie kolumny dysków z danymi mają co najmniej dwie jedynki
- Każdy wiersz posiada taką ilość jedynek, że ich suma modulo 2 jest równa zero

Można powiedzieć, że niejako **z każdym wierszem jest związany jeden schemat RAID 4**

# Obsługa awarii wielu dysków

## ⌘ Przykład:

D1: 11110000  
D2: 10101010  
D3: 00111000  
D4: 01000001

-----  
D5: 01100010  
D6: 00011011  
D7: 10001001

D1: 11110000  
D2: 10101010  
D3: 00111000  
D4: 01000001

-----  
D5: 01100010  
D6: 00011011  
D7: 10001001

D1: 11110000  
D2: 10101010  
D3: 00111000  
D4: 01000001

-----  
D5: 01100010  
D6: 00011011  
D7: 10001001

## ⌘ Zapis: na dysku D2 zmieniamy wartość na 00001111.

D2: 10101010  
D2': 00001111  
-----  
W: 10100101

W macierzy w kolumnie **dysku D2 jedynki są w wierszu pierwszym i drugim, a w trzecim nie.**  
Trzeba zatem zaktualizować odpowiednie **dyski redundancyjne: D5 i D6.**

D5: 01100010  
W: 10100101  
-----  
D5': 11000111

D6: 00011011  
W: 10100101  
-----  
D6': 10111110

# Obsługa awarii wielu dysków

Ostatecznie:

D1:	11110000
D2':	00001111
D3:	00111000
D4:	01000001
-----	
D5':	11000111
D6':	10111110
D7:	10001001


⌘ Procedura odtwarzania w przypadku awarii **co najwyżej dwóch dysków** (np. D2 i D5):

Nr dysku	1	2	3	4	5	6	7
1	1	1	1	0	1	0	0
1	1	1	0	1	0	1	0
1	1	0	1	1	0	0	1

⌘ w wierszu drugim wartości macierzy się różnią, więc na podstawie pozostałych dysków z tego wiersza (dla których w macierzy występuje jedynka) można odtworzyć stan uszkodzonych (tzn. wg: D1, D4, D6).



# Obsługa awarii wielu dysków



```
D1:  11110000
D2:  ????????
D3:  00111000
D4:  01000001
-----
D5:  ????????
D6:  10111110
D7:  10001001
```

- ⌘ Najpierw odtwarzamy D2 (suma modulo 2 wartości z dysków D1,D4,D6)

```
      D1:    11110000
      D4:    01000001
      D6:    10111110
      -----
      D2*:   00001111
```

- ⌘ Następnie odtwarzamy D5 (suma modulo 2 wartości z dysków D1,D2,D3)

```
      D1:    11110000
      D2:    00001111
      D3:    00111000
      -----
      D5*:   11000111
```

# Obsługa awarii wielu dysków

## ⌘ UWAGI:

- W opisanym przypadku również może być wykorzystana koncepcja RAID 5 w celu pominięcia „wąskiego gardła” przy zapisywaniu na dyskach redundancyjnych częściej niż na dyskach z danymi.
- Podane rozwiązanie nie jest ograniczone do 4 dysków z danymi.

Ogólny model: łączna liczba dysków  $2^k - 1$ , gdzie  $k$  jest liczbą dysków redundancyjnych, a pozostałe dyski to dyski z danymi

Dane											Redund				
1	2	3	4	5	6	7	8	9	10	11		12	13	14	15
1	1	1	1	0	1	1	0	1	0	0		1	0	0	0
1	1	1	0	1	1	0	1	0	1	0		0	1	0	0
1	1	0	1	1	0	1	1	0	0	1		0	0	1	0
1	0	1	1	1	0	0	0	1	1	1		0	0	0	1