**SQL Server Internals**

**Dlaczego (prawie) każda tabela powinna posiadać indeks klastrowy?**

Jednym z najczęstszych błędów popełnianych przez developerów T-SQL’a jest tworzenie tabel bez indeksów klastrowych. Wynika to z różnych powodów: pośpiechu, braku dobrych nawyków lub po prostu z wewnętrznego przeświadczenia, że taki indeks nie jest konieczny.

W tym artykule będę chciał udowodnić, że tworzenie takich indeksów powinno wejść nam w krew już podczas zakładania struktury nowej tabeli. Gdy tego nie zrobimy, w przyszłości możemy się spotkać z przykrymi konsekwencjami. Z drugiej strony brak takiego indeksu przez wiele lat może być nieodczuwalny. Wszystko zależy od ilości, rodzaju danych składowanych w tabeli oraz operacji na nich wykonywanych. Generalnie jednak, lepiej mieć ‘niepotrzebny’, ale dobry indeks klastrowy niż go nie mieć wcale i być może po czasie odczuć jego brak.

**Do not have a clustered index**

Jednym z przejawów, że możemy mieć z tym problem, jest wynik raportu RAP as a Service for SQL Server. Raport ten jest przygotowywany przez Microsoft np. w ramach wykupionej usługi Premier Support i wskazuje na słabe punkty na naszej instancji bazy danych. Sprawdza ją między innymi pod kątem występowania tabel bez indeksów klastrowych. Gdy takie znajdzie, w raporcie Medium Issues & Risks pojawi się taki oto wpis:



Co prawda może się okazać, że zgłoszony problem nie ma poziomu High czy Critical, ale mimo wszystko nawet Medium powinno nas zobligować do tego by się temu przyjrzeć. W opisie problemu czytamy, że znaleziono na instancji tzw. sterty (ang. heap), czyli tabele bez indeksów klastrowych.

**Sterta**

Czym zatem jest sterta? Najprościej rzecz ujmując to właśnie tabela bez indeksu klastrowego. Dane przechowywane w stercie nie są w żaden sposób uporządkowane i stanowią po prostu stertę stron i fragmentów. Spróbujmy taką sytuację zasymulować i zobaczyć jak to wygląda od strony silnika baz danych. Stwórzmy testową bazę danych a następnie załóżmy w niej tabelę np. z tytułami książek:

CREATE DATABASE TEST;

GO

USE TEST;

GO

CREATE TABLE Books

(

ID INT IDENTITY(1, 1) NOT NULL,

Title VARCHAR(1000) NULL

);

GO

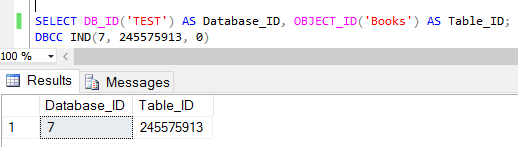
Tabela Books ma tylko dwa pola – narastający identyfikator ID oraz tytuł książki, jako znakowy typ danych o maksymalnej długości 1000 znaków. Jeszcze nie mamy w niej żadnych danych, ale możemy sprawdzić czy tabela zajmuje miejsce w pamięci oraz w pliku danych. Jak to sprawdzić?

Aby znaleźć strony w pamięci tej tabeli użyjemy nieudokumentowanego polecenia DBCC IND. Jako parametry podajemy nazwę bazy, tabelę oraz identyfikator indeksu. Ostatnim parametrem jest identyfikator partycji. Jeżeli tabela nie jest podzielona ma partycję, możemy go pominąć:

DBCC IND('TEST', 'Books', 0, 1)

DBCC IND('TEST', 'Books', 0)

Zamiast nazw obiektów możemy użyć identyfikatorów.



Jeżeli w tabeli nie mamy jeszcze żadnych danych DBCC IND zwróci tylko komunikat:

DBCC execution completed. If DBCC printed error messages, contact your system administrator.

*UWAGA: Dostępny jest również widok dynamiczny sys.dm\_db\_database\_page\_allocations, który wyciąga więcej danych niż DBCC IND, ale działa dopiero od wersji SQL Server 2012.*

SELECT \*

FROM sys.dm\_db\_database\_page\_allocations(DB\_ID(), OBJECT\_ID('Books'), 0, NULL, 'DETAILED')

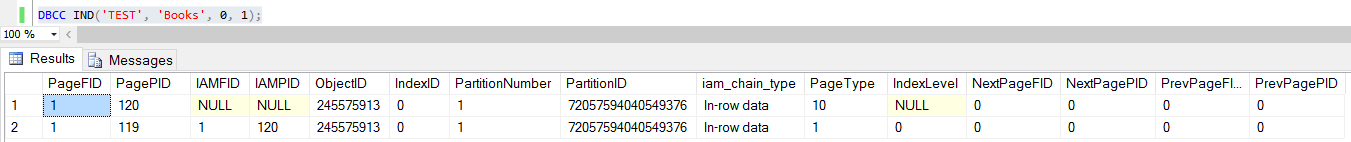
WHERE is\_allocated = 1;

GO

Spróbujmy dodać jeden wiersz:

INSERT INTO Books VALUES ('Romeo and Juliet');

Teraz polecenie IND zwraca nam dwa wiersze:



Widzimy teraz, że na potrzeby tego jednego wiersza silnik SQL Server’a zaalokował nam dwie strony pamięci o numerach 120 i 119 (PagePID) w pliku nr 1 (PageFID). Rodzaj stron to In-row-data, czyli są to jednostki przechowujące dane, których rozmiar wiersza nie przekracza 8060 bajtów. To jest podstawowy rodzaj stron. Natomiast obie strony różnią się typem: pierwsza to typ 10 czyli jest to tzw. mapa alokacji indeksu (Index Allocation Map - IAM). Ten specjalny typ strony służy tutaj do mapowania stron danych należących do sterty. Zajrzyjmy do środka tej strony.

**Index Allocation Map (IAM\_PAGE)**

Do tego celu użyjemy kolejnego polecenia konsoli - PAGE. Najpierw jednak musimy skierować wynik polecenia do okienka naszej sesji przez ustawienie flagi śledzenia 3604 DBCC TRACEON(3604):

DBCC TRACEON(3604);

GO

DBCC PAGE ('TEST', 1, 120, 1);

Parametry polecenia PAGE są podobne do IND, czyli najpierw podajemy bazę danych, potem nr pliku i numer strony. Ostatni parametr steruje szczegółowością raportu ze strony:

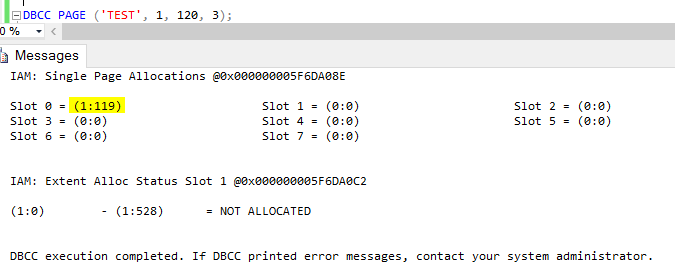
0 – pokazuje tylko nagłówek strony

1 – pokazuje nagłówek oraz zrzut pamięci poszczególnych wierszy tabeli (o ile strona zawiera wiersze, np. IAM ich nie zawiera, ponieważ jest tylko mapą bitową na właściwe strony z danymi)

2 – pokazuje nagłówek oraz zrzut pamięci całej strony

3 – pokazuje nagłówek oraz detale poszczególnych wierszy zapisanych na stronie

Dla stron typu IAM szczególnie przydatny jest parametr 3, ponieważ dodatkowo prezentuje nam wskaźniki do stron, które SQL Server alokował na potrzeby już konkretnych danych:



Pierwszy wpis IAM czyli Single Page Allocations zawiera wskaźniki do pierwszych ośmiu stron danych (8 stron = fragment, ang. extent). Widać tam naszą stronę z pierwszym wstawionym tytułem książki (PID = 119).

Kolejny wpis (Extent Alloc Status Slot) to wskaźnik na początkowy zakres 4 GB (528 stron), który strona IAM mapuje w pliku danych. Generalnie jest tu dość pusto, więc zapełnijmy tabelkę tytułami, tak aby wypełnić pierwsze 8 stron i wrzucić co nieco do Extent Alloc Status Slot. 10 tys. tytułów wystarczy, dodajmy tylko numerację każdego tytułu z wiodącymi zerami:

DECLARE @NumberOfRecords INT = 10000;

DECLARE @Title VARCHAR(1000);

WHILE (@NumberOfRecords > 0)

BEGIN

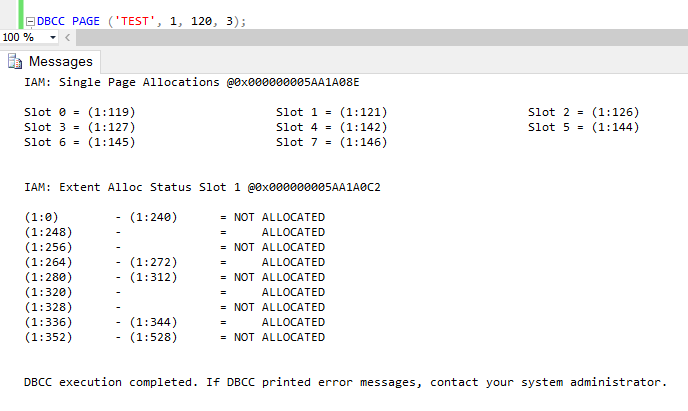
SET @Title = 'Romeo and Juliet ' + FORMAT(@NumberOfRecords, '00000');

INSERT INTO Books VALUES (@Title);

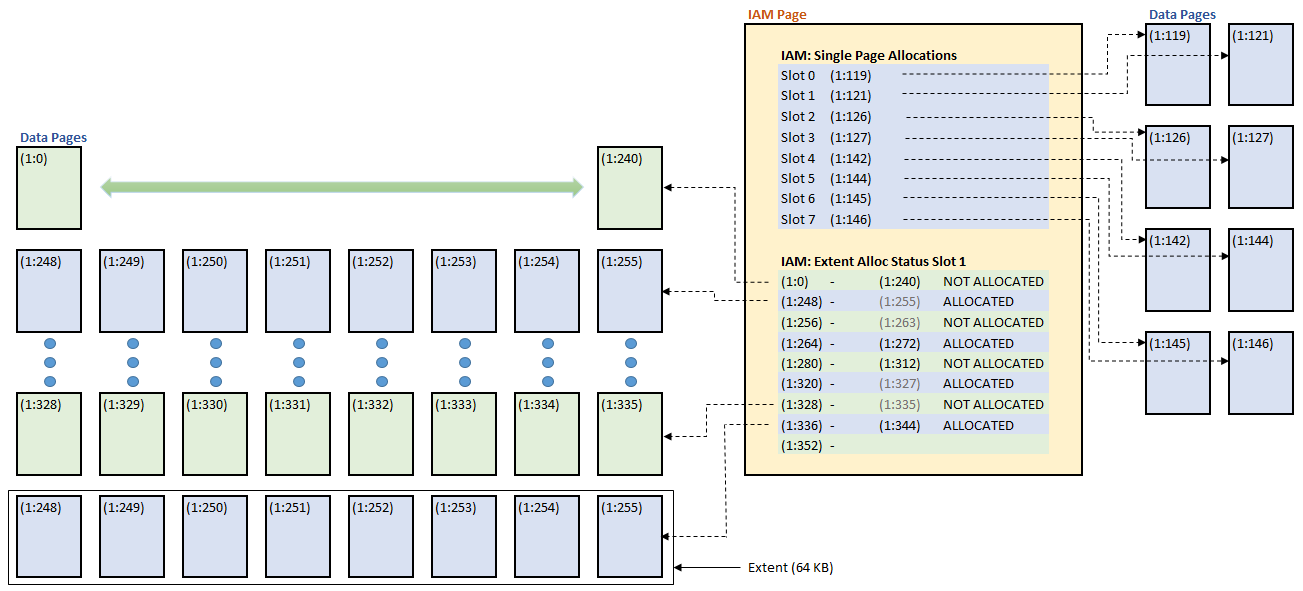
SET @NumberOfRecords = @NumberOfRecords - 1;

END

Jak podejrzymy teraz stronę IAM to jest znacznie ciekawiej:

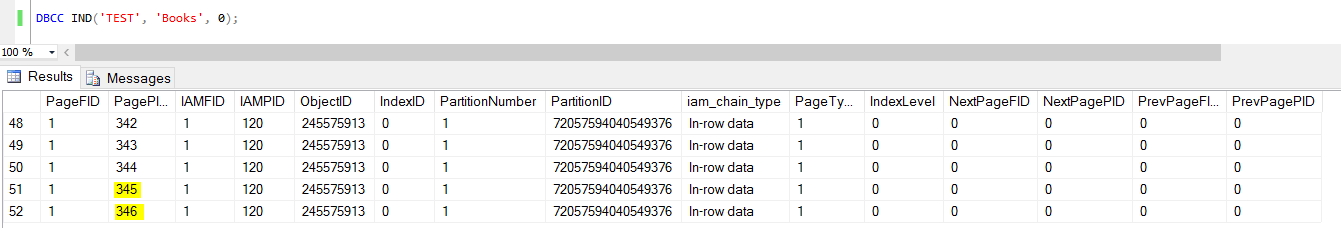


Już teraz widać, że dane wyglądają jakby były umieszczane w przypadkowych miejscach. W końcu to sterta. Jeszcze lepiej możemy to sobie uzmysłowić patrząc na poniższy diagram:



Strony IAM mają tutaj kluczowe znaczenie, ponieważ tylko dzięki nim silnik wie gdzie szukać danych konkretnych tabel (strony zaalokowane są na diagramie przedstawione w kolorze fioletowym, strony wolne w zielonym). To właśnie wskaźniki wskazują na pojedyncze strony lub całe fragmenty. Oczywiście stron IAM może być więcej, aby zaalokować więcej stron danych. Wtedy jest tworzony łańcuch takich stron.

Zauważmy, że w sekcji Extent Alloc Status Slot dane są prezentowane całymi fragmentami (8 kolejnych stron), dlatego powyższy diagram jest trochę nieprecyzyjny. Otóż nie widać na nim pojedynczych stron zaalokowanych po ostatnim fragmencie (po 1:344). Tutaj bardziej miarodajne jest jednak polecenie IND:



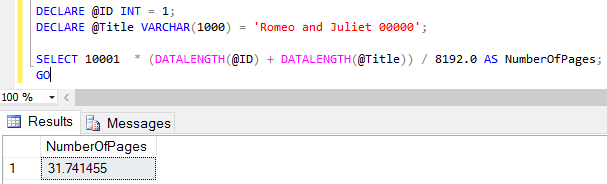
Mamy zatem jedną stronę IAM ze wskaźnikami na 51 stron typu Data Pages. Przechowujemy w nich 10 001 wierszy typu INT oraz VARCHAR. Póki co nie widać jakiegoś zagrożenia braku indeksu klastrowego. Jeżeli chcielibyśmy odczytać całą tabelę silnik wykona skanowanie zgodnie z porządkiem pliku posługując się wskaźnikami ze strony IAM. Jeżeli będziemy szukać danych po jakiś warunkach to przecież wystarczy założyć zwykły indeks nieklastrowy i po kłopocie.

Problem tkwi jednak w czymś innym niż wyszukiwanie danych w stercie, a przynajmniej nie na etapie gdy dopiero co zrobiliśmy import danych. Właściwy problem znajdziemy w detalach układu danych na stronach typu Data Pages.

**Data Pages**

Patrząc na nasz przykład spróbujmy najpierw oszacować ile wierszy taka strona powinna przechowywać.

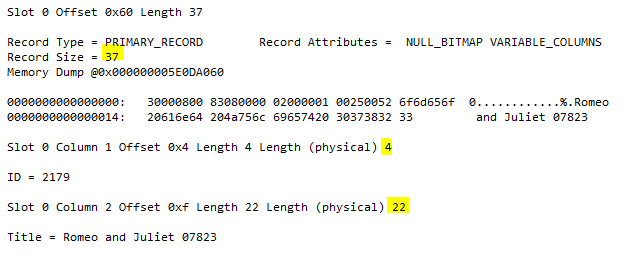
Typ danych INT zajmuje 4 bajty, każdy znak typu VARCHAR to 1 bajt. Wstawialiśmy ciągi o stałej wielkości ('Romeo and Juliet 00000') – czyli 22 bajty. Strona to 8 KB, więc powinniśmy się zmieścić na 32 stronach danych i na każdej stronie powinno być około 312 wierszy:



My jednak mamy zaalokowanych 51 stron, z czego wychodzi, że na stronie jest około 197 wierszy. Skąd ta różnica? Wynika właśnie ze struktury strony danych. Weźmy pod lupę jedną z nich np. stronę 1:251. Na początku najczytelniej będzie ją prześwietlić również z parametrem 3:

DBCC PAGE ('TEST', 1, 251, 3);

Parametr 3 dla strony danych udostępnia nam nagłówek strony, ale także poszczególne rekordy z zawartością i dokładnym opisem, co, ile zajmuje, wiersz po wierszu (tutaj są to kolejne sloty numerowane od zera):



No i mamy nieścisłość – przestrzeń na nasze właściwe dane faktycznie wynosi 26 bajtów, ale silnik rezerwuje nieco więcej miejsca na rekord (Record Size = 37). Pokolorujmy trochę ten zrzut i zobaczmy jakie informacje dla danego wiersza są tam trzymane:

0000000000000000: **30000800** **83080000** **02000001 00250052 6f6d656f** 0............%.**Romeo**

0000000000000014: **20616e64 204a756c 69657420 30373832 33** **and Juliet 07823**

**Czerwony kolor** – nagłówek wiersza danych – zawsze 4 bajty. Dwa pierwsze bajty to typ rekordu – tutaj to PRIMARY\_RECORD, czyli zwykły rekord danych, a nie np. indeks czy blob. Natomiast dwa kolejne bajty 0x0008 to wskaźnik na mapę bitową NULL tego wiersza (patrz dalej => **mapa bitowa NULL**)

**Kolor zielony** – dane kolumny INT – 4 bajty (czyt. 0x0883 = 2179 dec)

Teraz dochodzimy do atrybutów tego rekordu. Już wcześniej, na tym zrzucie, zostało nam zasygnalizowane, że możemy w tym wierszu spotkać się z dwoma atrybutami: Record Attributes = NULL\_BITMAP VARIABLE\_COLUMNS

**Kolor fioletowy** – to wspomniana wcześniej mapa bitowa NULL, wskazująca nam, czy dana kolumna w wierszu jest akurat NULL’em – 3 bajty. Dwa pierwsze bajty opisują ilość kolumn w wierszu (tutaj: 0x0002 czyli 2 kolumny). Kolejny bajt to właściwa mapa bitowa kolumn zawierających NULL. My takowej kolumny nie mamy więc wartość wynosi 0x00. Gdyby w naszej kolumnie Title była wartość NULL to ten bajt miałby wartość 0x02 = 0010 bin, czyli druga kolumna jest NULL. Cała mapa bitowa miałaby postać 0x020002. Taka konstrukcja może wydawać się nadmiarowa i nieco skomplikowana ale w połączeniu z nagłówkiem wiersza pozwala silnikowi błyskawicznie zlokalizować wartości NULL w wierszu. Taki zapis jest szczególnie efektywny gdy kolumn jest kilkanaście lub kilkadziesiąt.

**Kolor szary** – offset do końca rozmiaru kolumny o zmiennej szerokości – 4 bajty. Dwa pierwsze bajty to informacja o ilości takich kolumn w wierszu (0x0001 = 1 kolumna). Kolejne dwa bajty to właściwy offset do ostatniego bajta w tej kolumnie, licząc od pierwszego bajta całego wiersza (0x0025 = 37 dec czyli w zasadzie to wskaźnik na koniec naszego wiersza ponieważ kolumna Title jest ostatnią w wierszu).

Po zaszytych w środku atrybutach kolumny Title pora na właściwe dane:

**Niebieski kolor** – Dane kolumny VARCHAR – 22 bajty. Zauważmy, że doklejona do łańcucha znaków liczba jest oczywiście reprezentowana w bajtach – 1 cyfra = 1 bajt.

Teraz już wiemy skąd się wziął 37 bajtowy rozmiar wiersza. Do jego obliczenia możemy zastosować ogólny wzór:

**Row\_Size** = **Row\_header** + **Fixed\_data\_size** + **Variable\_data\_size** + **NULL\_bitmap**

Gdzie:

**Row\_header** = 4

**Fixed\_data\_size** = suma bajtów kolumn o stałej szerokości

**Variable\_data\_size** = 2 + (Ilość kolumn o zmiennej szerokości x 2) + aktualna szerokość kolumn o zmiennej szerokości

**NULL\_bitmap** = 2 + (Ilość wszystkich kolumn + 7) / 8) – zaokrąglamy do pełnych bajtów (w dół)

W naszym przypadku:

**Row\_size** = **4** + **4** + **(2 + (1 x 2) + 22)** + **(2 + (2 + 7) / 8)** = **4** + **4** + **26** + **3** = **37**

Gdy chcemy obliczyć maksymalny rozmiar danego wiersza, w zmiennej **Variable\_data\_size** zamiast aktualnej dodajemy maksymalną szerokość kolumny.

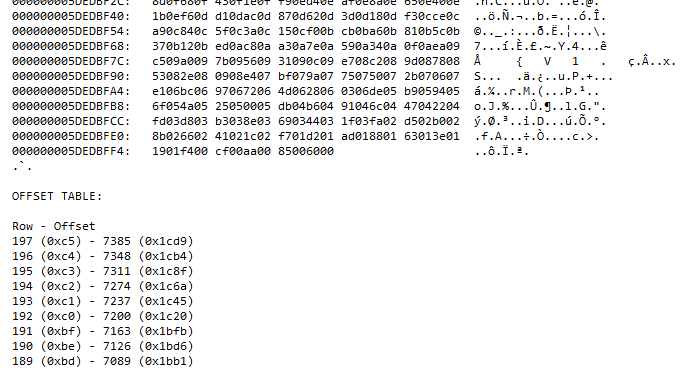
Strukturę wiersza mamy już prześwietloną. Spójrzmy teraz całościową na naszą przykładową stronę danych. Jak przewiniemy okienko wyniku DBCC PAGE do końca, zauważymy, że faktycznie strona zawiera około 200 wierszy (dokładnie 198 dla strony PID 1:251). Tę informację znajdziemy również w nagłówku strony:

m\_slotCnt = 198

Nagłówek strony zawiera wiele przydatnych informacji i jako taki również zajmuje przestrzeń – 96 bajtów i ten rozmiar jest stały dla wszystkich rodzajów stron. Gdy zaczniemy to analizować okaże się, że do pełnych 8 KB dalej nam brakuje trochę bajtów: 198 wierszy po 37 bajtów, daje nam 7326, plus 96 z nagłówka strony, to razem mamy 7422. Poszukajmy brakującego 0,5 KB uruchamiając DBCC Page z parametrem 2:

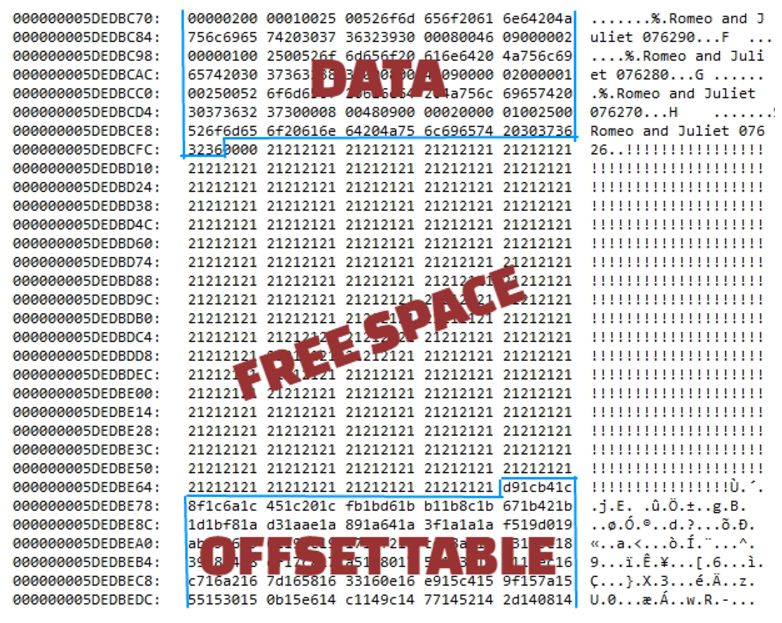
DBCC PAGE ('TEST', 1, 251, 2);

Teraz zamiast szczegółów poszczególnych wierszy widzimy zrzut pamięci oraz zamieszczoną na końcu tablicę offsetów:



Te offsety to dwubajtowe wskaźniki do konkretnych wierszy na stronie. Są zapisywane od końca strony i dlatego te na samym końcu wskazują na pierwsze wiersze danych. Dzięki nim bardzo szybko możemy znaleźć właściwy wiersz na stronie.

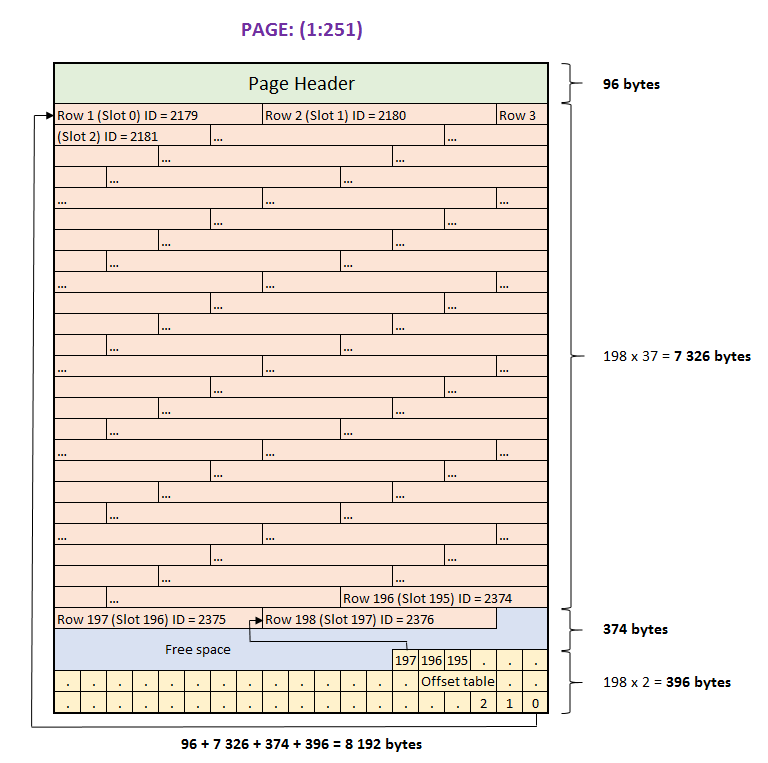
Gdy spojrzymy na ten zrzut pamięci nieco powyżej tablicy offsetów, od razu domyślimy się co zajmuje resztę miejsca na stronie:



Tak, to wolna przestrzeń. I nie musimy jej obliczać ponieważ tę informację także znajdziemy w nagłówku strony:

m\_freeCnt = 374

Zobaczmy jak wygląda diagram całej naszej strony 1:251



Teraz wiemy już wystarczająco dużo na temat składowania danych na stercie. Przynajmniej, jeśli chodzi o tak małą tabelkę, jaką jest nasza przykładowa Books. Ale patrząc na ten diagram spróbujmy wydedukować, jakie operacje wykona silnik, gdy rozszerzymy jeden z wierszy, na początek – dwukrotnie. Na przykład pierwszy wiersz na tej stronie, czyli ID = 2179. Czy przesunie offsety wszystkich pozostałych wierszy o kolejne 22 bajty w przód? Albo od razu zagospodaruje wolną przestrzeń przesuwając offset pierwszego wiersza na dane przesunięte pod koniec strony, do tego wolnego miejsca? Albo stworzy wskaźnik z końca pierwszego wiersza do reszty danych w końcu strony? To ostatnie chyba odpada – przecież na końcu wiersza nie ma już miejsca…

No to koniec teorii i zaczynamy właściwą zabawę ☺

**UPDATE**

Przed następnymi operacjami warto zrobić sobie backup naszej bazy (nazwijmy go np. TEST.bak). Oryginalny układ danych pomoże nam później porównać, jakie zmiany zaszły w strukturach stron oraz jak to będzie się przekładało na wydajność.

Wykonajmy poniższe polecenie:

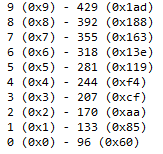
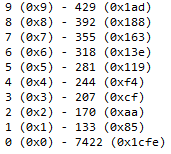
UPDATE Books SET Title = Title + Title WHERE ID = 2179;

GO

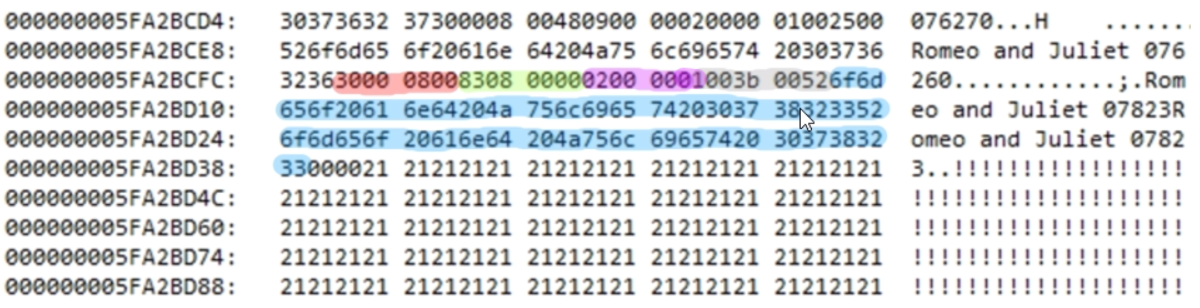
Gdy spojrzymy teraz w nagłówek strony 1:251 to zobaczymy:

m\_slotCnt = 198 m\_freeCnt = 352

Ilość wierszy na stronie się zgadza, liczba wolnych bajtów zmniejszyła się o 22. Czyli wszystko w najlepszym porządku. A jak wygląda tabela offsetów dla wierszy na stronie? Porównajmy oryginalną z tą po update’cie dla pierwszych 10 wierszy:

Jak widać silnik nie ruszył pozostałych offsetów, natomiast cały rozszerzony wiersz przesunął na koniec fragmentu z danymi:



Super! Wydaje się to najbardziej optymalnym rozwiązaniem: zmiana jednego offsetu i przepisanie wiersza w ramach strony do nowej lokalizacji. Jakbyśmy spojrzeli na początek naszej strony, zaraz po nagłówku, zauważymy, że poprzednia wartość (pojedynczy Title) pozostała na swoim miejscu. Silnik nawet nie trudził się nadpisywaniem zwolnionego obszaru.

Odtwórzmy teraz bazę z backupu TEST.bak.

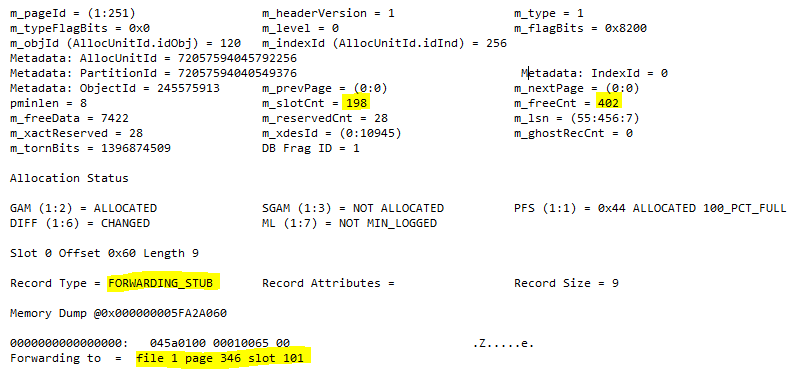
Spróbujmy całkowicie zapełnić wolną przestrzeń na stronie, tzn. rozszerzmy tytuł o minimum 375 bajtów, czyli powinniśmy go np. zreplikować co najmniej 19 razy.

Wykonajmy poniższy skrypt:

UPDATE Books SET Title = REPLICATE(Title, 19) WHERE ID = 2179;

GO

PAGE z parametrem 3 pokazuje nam to, co poniżej:



Ilość wierszy bez zmian, więcej wolnego miejsca na stronie, natomiast w tablicy offsetów pierwszy wskaźnik to dalej ten nasz zmodyfikowany wiersz z rozszerzoną kolumną Title. Tyle, że tego wiersza nie ma fizycznie na tej stronie. Mówi o tym typ rekordu: FORWARDING\_STUB. To jest wskaźnik przenoszący do właściwej lokalizacji - 102 wpis na ostatniej stronie tabeli: PID 1:346.

Czy widzimy już potencjalne niebezpieczeństwo związane ze stertami? Strona po zapełnieniu nie podzieliła się, jak to ma miejsce w strukturze b-drzewa. Powstał za to wskaźnik do innej lokalizacji. Operacja jest szybka i nie wymaga tylu operacji jak przy podziale stron. Ale wyobraźmy sobie, że dotyczy ona większej ilości rekordów. Odtwórzmy TEST.bak i wykonajmy poniższy skrypt:

UPDATE Books SET Title = Title + Title;

GO

Ile takich rekordów zostało przeniesionych?

SELECT OBJECT\_NAME(object\_id) AS table\_name

,forwarded\_record\_count

,page\_count

FROM sys.dm\_db\_index\_physical\_stats(DB\_ID(), OBJECT\_ID('Books'), DEFAULT, DEFAULT, 'DETAILED');



Ponad 40%. Dla tylu wierszy trzeba będzie odczytać dodatkową stronę by znaleźć właściwe dane. Dla sterty bez indeksu nieklastrowego nie ma to znaczenia, bo i tak zawsze będziemy mieli skanowanie tabeli w kolejności alokacji. Ale przeanalizujmy przypadek, gdy taki indeks (nieklastrowy!) mamy.

**Problem 1 – Czytanie**

Odtwórzmy jeszcze raz bazę z backupu TEST.bak i załóżmy wspomniany indeks nieklastrowy:

CREATE NONCLUSTERED INDEX IX\_ID ON Books (ID);

Będziemy chcieli wyszukiwać tytuły po identyfikatorach ID. Zanim to zrobimy ustawmy zbieranie statystyk dla operacji IO oraz włączmy podgląd aktualnego planu zapytań:

SET STATISTICS IO ON;

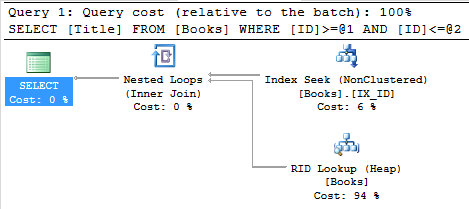
GO

Wykonajmy zapytanie:

SELECT Title FROM Books WHERE ID BETWEEN 2179 AND 2199

GO

Co nam dał ten indeks? Nie musimy skanować całej tabeli by wybrać interesujące nas tytuły:



Według planu szukamy po indeksie ID z zadanego zakresu a następnie dociągamy (RID Lookup) konkretne tytuły wcześniej wybierając ze strony IAM właściwe PID’y stron danych. Mamy tylko 23 logiczne odczyty. Gdyby nie było tego indeksu musielibyśmy zeskanować całość danych i tych odczytów byłoby 51.

Teraz wykonujemy aktualizację powodującą pojawienie się wskaźników przenoszących dla tych wierszy (UPDATE Books SET Title = REPLICATE(Title, 19) WHERE ID BETWEEN 2179 AND 2199;) i ponawiamy wyszukiwanie: plan ten sam, ale musieliśmy wykonać 19 odczytów więcej: logical reads 42.

Wniosek: przy tak małej ilości danych musimy zrobić ponad 80% więcej odczytów. Wiadomo, że przykład jest nieco przejaskrawiony i może się wydawać, że część z tych dodatkowych odczytów wynika z większej ilości danych. Ale jak dokładnie to sprawdzimy to zobaczymy, że dodatkowych stron wynikających z rozszerzonych kolumn jest… tylko jedna. Reszta odczytów wynikała z tego, że silnik musiał odwiedzić strony, na których nie było danych, a odczytane były tylko po to by pobrać wskaźnik przenoszący.

Zobaczmy teraz, co się stanie, gdy zaczniemy kasować dane przenoszone tymi wskaźnikami.

**Problem 2 – Usuwanie**

Odtwórzmy ponownie bazę z backupu TEST.bak. Wykonajmy naszego update’a na całości danych:

UPDATE Books SET Title = REPLICATE(Title, 19);

GO

Uff.. Zrobiło się tego całkiem sporo. Mamy 570 stron i ponad 90% rekordów przeniesionych z oryginalnych lokalizacji:



Teraz je wszystkie skasujmy. Po prostu rozmyśliliśmy się i będziemy zapełniać tabelkę na nowo – chcemy przy tym zachować ciągłość identyfikatora ID, więc zrobimy po prostu:

DELETE FROM Books;

GO

Zobaczmy jak nasz silnik musiał się napracować usuwając te wszystkie dane:



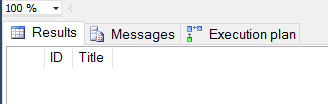
Prawie 10 tys. odczytów tabelki, która ma tylko 570 stron…

Ale to, że podróże po wskaźnikach są kosztowne to już wiemy. W tym przykładzie ważne jest coś innego. Upewnijmy się, że tabelka jest już pusta:

SELECT \* FROM Books;

GO

Jest pusto:



A teraz kliknijmy w zakładkę Messages, aby zobaczyć statystyki:

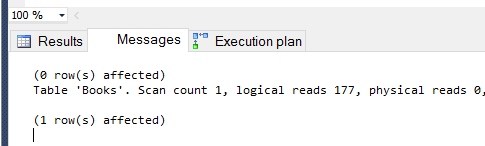
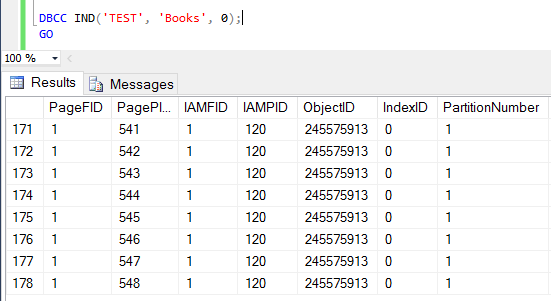


Tabela jest pusta a silnik w pamięci przeszukuje 177 stron! Sprawdzamy dalej:



Faktycznie, mamy 177 ‘pustaków’ zamapowanych na stronie IAM. Ile to może ‘ważyć’? Sprawdźmy to wykorzystując poniższe zapytanie:

SELECT

t.NAME AS TableName,

s.Name AS SchemaName,

p.rows AS RowCounts,

SUM(a.total\_pages) \* 8 AS TotalSpaceKB,

SUM(a.used\_pages) \* 8 AS UsedSpaceKB,

(SUM(a.total\_pages) - SUM(a.used\_pages)) \* 8 AS UnusedSpaceKB

FROM

sys.tables t

INNER JOIN

sys.indexes i ON t.OBJECT\_ID = i.object\_id

INNER JOIN

sys.partitions p ON i.object\_id = p.OBJECT\_ID AND i.index\_id = p.index\_id

INNER JOIN

sys.allocation\_units a ON p.partition\_id = a.container\_id

LEFT OUTER JOIN

sys.schemas s ON t.schema\_id = s.schema\_id

WHERE

t.NAME NOT LIKE 'dt%'

AND t.is\_ms\_shipped = 0

AND i.OBJECT\_ID > 255

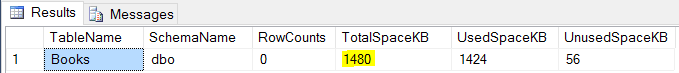
GROUP BY

t.Name, s.Name, p.Rows

ORDER BY

TotalSpaceKB DESC

Wynik:



Pusta tabela zajmuje 1,5 MB przestrzeni! To jest chyba jedna z najpoważniejszych wad sterty. Jedynym pocieszeniem jest to, że ponownie zapełniając tabelę prawdopodobnie wykorzystamy te puste strony. Prawdopodobnie – bo pamiętajmy, że sterty są zapełniane losowo.

Na wstępie pisałem, że to właśnie RAP as a Service sygnalizuje nam, że na instancji bazy danych są struktury bez indeksu klastrowego. Przeważnie jest tak, że na tym samym poziomie ryzyka pojawiają się jeszcze inne wpisy z tym związane. W podsumowaniu będzie jeszcze o tym mowa, że sterty można tworzyć o ile robimy to świadomie i mam ku temu jakiś cel. Jeżeli jednak w raporcie pojawi się wpis jak poniżej, to celowość takiego działania jest mocno wątpliwa:



**High number of Forwarded Records**

Otóż to. Duża liczba wskaźników przenoszących. Ale co to właściwie oznacza ‘high number’? Ten problem jest aktywowany, jeżeli liczba wyszukiwanych rekordów po wskaźnikach na sekundę jest większa od 10 na 100 uruchamianych wsadów T-SQL.

Jak to samemu sprawdzić? Nic prostszego. Wystarczy porównać dwa liczniki:

1. SQLServer:Access Methods – Forwarded Records/sec
2. SQLServer:SQL Statistics – Batch Requests/sec

Możemy to zrobić bezpośrednio z poziomu T-SQL’a, jednak musimy pamiętać, że tak uzyskane liczniki będą zawierały wartości kumulatywne, czyli w zasadzie zebrane od ostatniego restartu instancji:

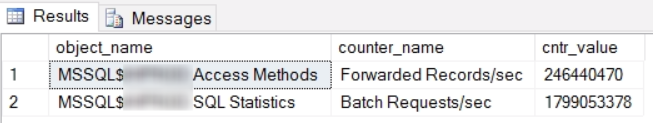
SELECT object\_name, counter\_name, cntr\_value

FROM sys.dm\_os\_performance\_counters

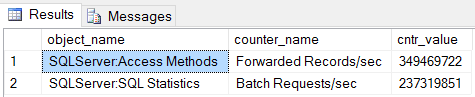
WHERE object\_name LIKE '%Access Methods%' and counter\_name = 'Forwarded Records/sec'

OR object\_name LIKE '%SQL Statistics%' and counter\_name = 'Batch Requests/sec';

Niemniej nawet takie wskazania pomogą nam określić czy już mamy problem z Forwarded Records czy jeszcze mamy czas i możemy odłożyć to na później. Zobaczmy przykładowe wskazania dla dwóch instancji, o których można powiedzieć, że są typowymi instancjami OLTP:

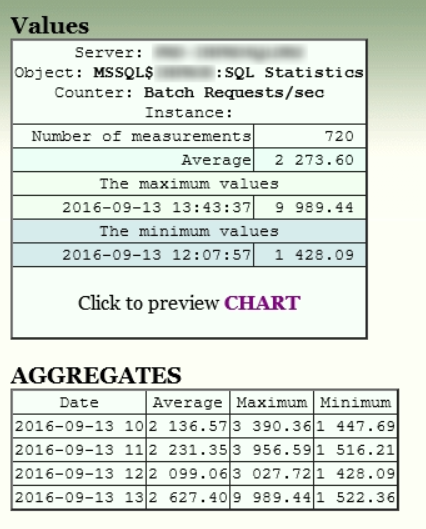
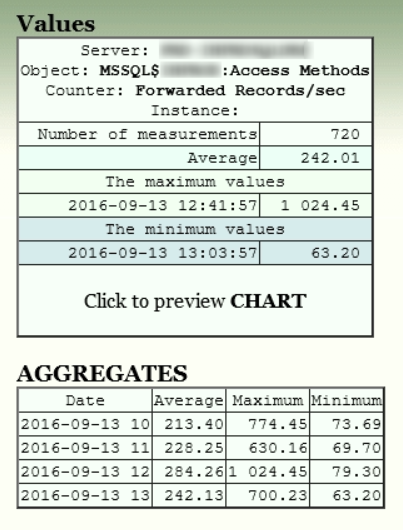


Spójrzmy: na 1,8 mld wsadów mamy 246 mln wierszy wyszukanych po wskaźnikach przenoszących. Inaczej mówiąc mamy prawie 14 takich rekordów na 100 batch’y T-SQL (246/1800 \* 100). Nie jest dobrze ale zawsze może być gorzej, tak jak na kolejnej instancji:



Tu już mamy ewidentny problem wymagający natychmiastowego działania: na 100 wsadów po wskaźnikach szukamy prawie 150 wierszy.

Dobrze jest mieć monitoring zrzucający chwilowe wartości tych liczników, ponieważ wiele nam to może powiedzieć o skali problemu w godzinach, gdy biznes działa najbardziej intensywnie. Dla naszej pierwszej instancji będą to godziny od 10:00 do 14:00. Pomiary są zrzucane co 20 sekund:



Tutaj wyraźnie widać, że średnie z godzin od 12 do 13 oscylują właśnie w granicach obliczonych z liczników kumulatywnych (284/2100 \* 100 = 13,5). Problem zatem jest istotny. Wynika to też tylko z samego licznika Forwarded Records/sec. Wyobraźmy sobie, że w każdej sekundzie pracy naszego systemu robimy dodatkowo średnio ponad 200 odczytów. A czasami zdarza się, że tych odczytów jest ponad tysiąc…

Ale to nie koniec problemów powiązanych ze stertami. Kiedy w raporcie zauważymy, że wykryto sterty i są Forwarded Records to najpewniej będą również…



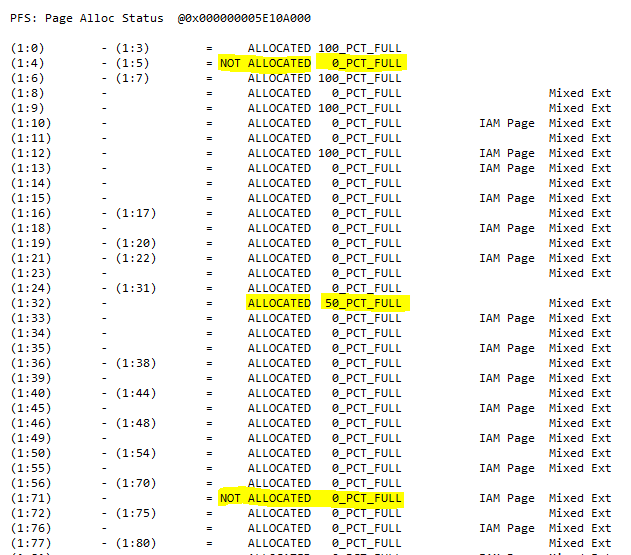
**High number of Free Space Scans**

Cóż to takiego? Komunikat informuje nas o nadmiarowym skanowaniu stron PFS. To kolejny typ strony, który najprościej rzecz biorąc jest mapą bajtową (nie bitową!) wolnego miejsca w pliku danych. Dotychczas wspominaliśmy o stronach danych i IAM. Strony danych zawierają konkretne dane, czyli wiersze tabel, natomiast strony IAM zawierały wskaźniki do stron danych dla konkretnego obiektu w bazie danych. Co jednak w przypadku, gdy chcemy dodać nowe wiersze lub rozszerzyć istniejące kolumny? System musi sprawdzić gdzie szukać tego wolnego miejsca. I dla lokalizacji wolnej przestrzeni w pliku, w przypadku stert wykorzystywane są właśnie strony typu PFS (Page Free Space).

Aby lepiej zrozumieć ten mechanizm wyszukiwania, obejrzyjmy sobie tę stronę. W każdym pliku danych jest to strona o numerze 1 (druga strona pliku, ponieważ strona o numerze 0 to nagłówek pliku). Można sobie odtworzyć TEST.bak by zobaczyć jak wyglądał PFS przed rozszerzaniem kolumn:

DBCC PAGE ('TEST', 1, 1, 3);

Czyli mamy nagłówek strony oraz naszą mapę bajtową do konkretnych stron w pliku:



Przykładowo: strony 4 i 5 oraz 71 są niezalokowane a strona 32 jest w połowie pusta. Każdy bajt na stronie odpowiada jednej stronie w pliku. Dlatego jest to właśnie mapa bajtowa, w której pierwsze 3 bity określają zapełnienie wskazywanej strony:

000 – strona jest pusta

001 – 50% zapełnienia

010 – 80% zapełnienia

011 – 95% zapełnienia

100 – strona zapełniona w 100%

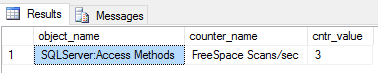
Mechanizm wyszukiwania stron gotowych na przyjęcie dodatkowych rekordów wydaje się przemyślany, ale sprawdźmy, co się będzie działo, gdy będziemy chcieli wstawić 100 rekordów. Wcześniej zanotujmy sobie kumulatywne wskazania licznika FreeSpace Scans/sec:

SELECT object\_name, counter\_name, cntr\_value

FROM sys.dm\_os\_performance\_counters

WHERE object\_name LIKE '%Access Methods%' and counter\_name = 'FreeSpace Scans/sec'

Na mojej instancji (całkiem niedawno zrestartowanej) to było:

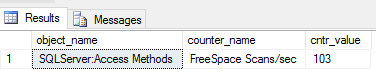


Wykonajmy teraz wstawienie 100 rekordów:

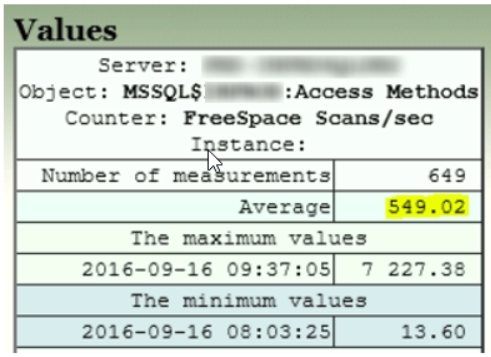
INSERT INTO Books VALUES ('Romeo and Juliet 10001');

GO 100

Sprawdźmy jeszcze raz licznik FreeSpace Scans/sec:



Wniosek? Każdy Insert poprzedzony został skanowaniem (czyli dodatkowym odczytem) strony PFS. Każdy jeden! Wg Microsoftu, podobnie jak w przypadku Forwarded Records, jeżeli średnia nie przekracza 10% w stosunku do ilości batch’y, to póki co nie ma się czym martwić. Ale znowu – dobrze mieć chwilowe wartości z godzin intensywnej pracy systemu, przykładowo z godziny 8 do 16:



Jeżeli ilość wsadów w tym czasie oscyluje w granicach 2200 na sekundę to mamy wynik 25%. To już jest problem, który wymaga naszej interwencji, bo oznacza, że strony PFS są intensywnie skanowane w naszym systemie podczas składowania lub aktualizacji danych. Mamy dodatkowe odczyty, ale również zwiększoną utylizację CPU.

A gdy połączymy te wnioski z poprzednimi będziemy mieli pełen obraz: sterty w systemie zostały stworzone nieświadomie i używane są w niekontrolowany sposób.

Jak temu zaradzić? Są dwie drogi życiowe…

**Jak żyć?**

Odpowiedź jest prosta: nie żyć, rozwieść się. Pozbyć się wszystkich stert z systemu, szczególnie tych, które są namiętnie aktualizowane, uzupełniane i wyszukiwane. Bo i po co nam sterta? Przydaje się jedynie w bardzo wąskich zastosowaniach np. gdy chcemy zaimportować dużą ilość danych do istniejącej tabeli, która już zawiera jakieś rekordy. Wtedy warto ściągnąć indeks klastrowy (i pozostałe indeksy również) i włączyć importowanie danych z minimalnym zakresem logowania. Po imporcie i tak należy przywrócić indeksy.

Ktoś może jednak powiedzieć, że w jego systemie sterty są wydajniejsze niż struktura b-drzewa, bo bardzo często na raz pobiera całość danych lub większą jej część, np. podczas procesów ETL. Albo używa tabeli do przechowywania logów aplikacji i przeszukuje je tylko w wyjątkowych sytuacjach. Albo korzysta z polecenia SELECT INTO do robienia kopii tabel.

No i super – to się właśnie nazywa świadome tworzenie stert i gdy nie idzie w parze z wysokimi wartościami liczników Forwarded records/sec – to wszystko jest w jak najlepszym porządku.

Ale i tak trzeba pilnować takich delikwentów i regularnie sprawdzać poziomy liczników, na bieżąco kasować kopie tabel i ogólnie dbać o porządek na bazie. A co zrobić by uporządkować stertę? Wykonać polecenie przebudowy tabeli:

ALTER TABLE Books REBUILD

Polecenie tak naprawdę usunie całą strukturę tabeli i stworzy ją najprawdopodobniej w innym miejscu, z inną stroną IAM i innymi stronami danych. Najprawdopodobniej, bo znowu będzie to przypadkowe i nikt nam nie da gwarancji, że strony świeżo przebudowanej sterty będą ułożone w jednym ciągu. Ale przynajmniej pozbędziemy się Forwarded Records.

Kiedy jednak developer nie ma żadnych argumentów na obronę swoich stert – trzeba ten bałagan po nim posprzątać.

**Jak się rozwieść?**

Najlepiej zacząć od wyszukania wszystkich stert w danej bazie danych:

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

GO

SELECT OBJECT\_NAME(object\_id) AS table\_name

,forwarded\_record\_count

,page\_count

,record\_count

,FORMAT(page\_count \* 8 / 1024.0, '### ### ##0.00') AS size\_MB

,index\_type\_desc

,avg\_fragmentation\_in\_percent

,fragment\_count

,FORMAT(avg\_page\_space\_used\_in\_percent, '0.00') AS avg\_page\_space\_used\_in\_percent

FROM sys.dm\_db\_index\_physical\_stats(DB\_ID(), NULL, DEFAULT, DEFAULT, 'DETAILED')

WHERE index\_type\_desc = 'HEAP'

ORDER BY forwarded\_record\_count DESC, size\_MB DESC, record\_count DESC;

UWAGA: Na dużych bazach danych powyższe zapytanie będzie generowało sporo odczytów fizycznych, więc należy je wykonywać poza godzinami wzmożonej pracy systemu.

Zapytanie wyciągnie nam z bazy danych wszystkie sterty z uwzględnieniem tych, które mają największą ilość wskaźników Forwarded Records, w drugiej kolejności są największe i mają najwięcej rekordów ogółem.

Najlepszą metodologią jest wyeliminowanie najpierw stert z dużą ilością Forwarded Records. Następnie obserwowanie liczników Forwarded Records/sec oraz FreeSpace Scans/sec. Gdy oba liczniki się nie poprawiają to przystępujemy do dalszej redukcji stert, tym razem biorąc pod uwagę sam rozmiar sterty i ilość rekordów. I znowu monitorujemy wskazane liczniki. Gdy dalej się nie poprawiają to bierzemy pod uwagę nawet te najmniejsze tabelki, z małą ilością rekordów oraz te, w których zmienia się ilość rekordów. Spotkałem się też z przypadkiem, że jeden z systemów używał sterty w bazie użytkownika, jako tabelki tymczasowej, którą regularnie aktualizował, ale suma tych rekordów nigdy nie przekraczała kilkuset…

W każdym bądź razie jedynym sposobem na pozbycie się sterty jest założenie indeksu klastrowego, czyli zamiana struktury w b-tree (balansed tree – zbalansowane drzewo), która nie posiada wad sterty i jednocześnie posiada mnóstwo zalet ułatwiających np. złączenia w zapytaniach. Nie ma trzeciej możliwości – stąd też powszechnie używana, skrótowa nazwa struktury tabeli: HOBT (ang. Heap or B-Tree).

Zachodzi jednak pytanie – jaką kolumnę lub kolumny wybrać na indeks klastrowy?

**Wybierz idealnego kandydata**

Idealny kandydat na indeks klastrowy:

1. Zawiera niezbyt szerokie kolumny (np. INT, CHAR(5)).
2. Posiada unikalne wartości – jeżeli nie będą unikalne SQL Server sam o to zadba dodając odpowiednią kolumnę, a tego wolelibyśmy uniknąć…
3. Zaindeksowana kolumna (lub kolumny) nigdy nie będą aktualizowane.
4. Jest monotonicznie rosnący (np.: IDENTITY(1,1))

Mamy też limity, które musimy brać pod uwagę:

- indeks zgrupowany nie może zawierać więcej niż 16 kolumn,

- szerokość indeksu nie może być większa niż 900 bajtów.

Oczywiście dobry indeks klastrowy nawet w połowie nie zbliży się do tych wartości…

Niestety – tutaj jak w życiu – trudno jest znaleźć idealnego kandydata i w większości przypadków trzeba iść na kompromis. Generalnie jednak najlepiej nadają się na indeksy klastrowe następujące typy danych (mój osobisty ranking, od najlepszych do trochę gorszych):

1. INT – ideał zwłaszcza, jeżeli jest monotonicznie narastający (IDENTITY(1,1))
2. BIGINT – prawie ideał zwłaszcza, jeżeli jest monotonicznie narastający (IDENTITY(1,1))
3. DATE, SMALLDATETIME – dobre, kiedy większość zapytań jest o zakres
4. CHAR – dobry, jeżeli jest stosunkowo krótki i jeżeli się nie zmienia (PESEL, NIP, REGON?)
5. VARCHAR – dopuszczalny, jeżeli jest stosunkowo krótki i jeżeli się nie zmienia
6. UNIQUEIDENTIFIER – dopuszczalny dla identyfikatorów generowanych sekwencyjnie (newsequentialid), tu jednak musimy wziąć pod uwagę, że narastanie może zostać zaburzone…

Dla indeksów klastrowych zgrupowanych idealnymi kandydatami mogłyby być kolumny INT + DATE, INT + CHAR(5) lub podobne, najlepiej z unikalnymi grupami wartości.

Aby nie mieć takich rozterek przy zakładaniu indeksów klastrowych na istniejących tabelach, warto wyrobić sobie odpowiednie nawyki, które zadziałają już wtedy, gdy będziemy tworzyć struktury bazodanowe.

Od wersji SQL Server 2014 możemy już tworzyć indeksy klastrowe in-line w definicji tabeli:

CREATE TABLE Books

(

ID INT NOT NULL INDEX IX\_ID CLUSTERED,

Title VARCHAR(1000) NULL

);

GO

Jeszcze lepszym sposobem jest stworzenie klucza głównego, który w SQL Server domyślnie jest wymuszany właśnie za pomocą indeksu klastrowego:

CREATE TABLE Books

(

ID INT IDENTITY(1, 1) CONSTRAINT PK\_ID\_Books PRIMARY KEY,

Title VARCHAR(1000) NULL

);

GO

To też dowodzi, że Microsoft przewidział opisane w tym artykule problemy i niejako zabezpieczył się przed masowym tworzeniem stert, ponieważ większość programistów i tak będzie tworzyła ograniczenie PRIMARY KEY w tabeli. Należy jednak pamiętać, że nieco inne ograniczenie (UNIQUE) w domyśle będzie tworzyło na kolumnie indeks nieklastrowy:

CREATE TABLE Books

(

ID INT IDENTITY(1, 1) CONSTRAINT UQ\_ID\_Books UNIQUE,

Title VARCHAR(1000) NULL

);

GO

Dlatego najlepszym wyjściem będzie wyrobienie sobie nawyku jawnego definiowania ograniczenia jako indeksu klastrowego, ponieważ wtedy nie będziemy mieli żadnej wątpliwości co tworzymy i w jakiej postaci:

CREATE TABLE Books

(

ID INT IDENTITY(1, 1) CONSTRAINT PK\_ID\_Books PRIMARY KEY **CLUSTERED**,

Title VARCHAR(1000) NULL

);

GO

CREATE TABLE Books

(

ID INT IDENTITY(1, 1) CONSTRAINT UQC\_ID\_Books UNIQUE **CLUSTERED**,

Title VARCHAR(1000) NULL

);

GO

**Po słowie**

Niniejszy artykuł miał za zadanie udowodnić pewną tezę: sterty w MS SQL są strukturami, które mogą pogorszyć wydajność naszej bazy. Oczywiście, czym innym jest **świadome** tworzenie takich struktur, w jakimś konkretnym celu. Z doświadczenia jednak wiem, że 99% stert powstaje całkowicie przez przypadek. A w większości sytuacji indeks klastrowy jest rozwiązaniem znacznie lepszym. Dlaczego? Choćby dlatego, że nie powiela wad sterty. Omówienie innych zalet struktury b-drzewa wykracza poza zakres tego artykułu. A mi tylko chodziło o wyraźne zdyskredytowanie kandydata na tabelę, jakim może być sterta ☺. Dlatego zachęcam wszystkich programistów T-SQL do wyrabiania sobie dobrych nawyków tworzenia, niejako z automatu, indeksów klastrowych, już podczas pisania kodu tabelki.

A jeżeli nie jesteście pewni jak działa pewien mechanizm, to eksperymentujcie i zaglądajcie bezpośrednio do stron danych – i to jest drugi cel tego artykułu – zachęta do eksperymentowania. Można się wiele nauczyć podczas takich analiz, ja na przykład, przy okazji pisania tego dokumentu, dowiedziałem się ile tak naprawdę waży NULL ☺.

Użyty w artykule kod T-SQL można w całości ściągnąć z:

<https://github.com/miros72/HeapMustDie.git>

Przydatne linki, w tym artykuły pioniera w dziedzinie SQL Server Internals, Paula Randala:

<http://www.sqlskills.com/blogs/paul/inside-the-storage-engine-gam-sgam-pfs-and-other-allocation-maps/>

<http://www.sqlskills.com/blogs/paul/inside-the-storage-engine-anatomy-of-a-record/>

<http://www.sqlskills.com/blogs/paul/inside-the-storage-engine-anatomy-of-a-page/>

<https://sqlsunday.com/2016/09/01/compelling-case-for-heaps/>

<http://string-functions.com/string-hex.aspx>