Columnstore indexes – Indeksy kolumnowe

Poruszymy następujące zagadnienia:

- Czym są Columnstore Indexes
- Architektura Columnstore
- Rowgroup
- Column segments
- Delta rowgroup
- Delta store
- Tuple-mover
- Batch mode
- Algorytmy kompresji
- Przykłady w SQL Server
- Kiedy używać Columnstore a kiedy Rowstore

Czym są Columnstore Indexes?

Indeksy kolumnowe zostały wprowadzone w 2012 roku. To nowy sposób przechowywania danych z tabeli, który zdecydowanie poprawia wydajność niektórych zapytań(np. INSERT i SELECT). Zdaniem Microsoftu w niektórych przypadkach występuje nawet 10-krotne poprawienie wydajności.

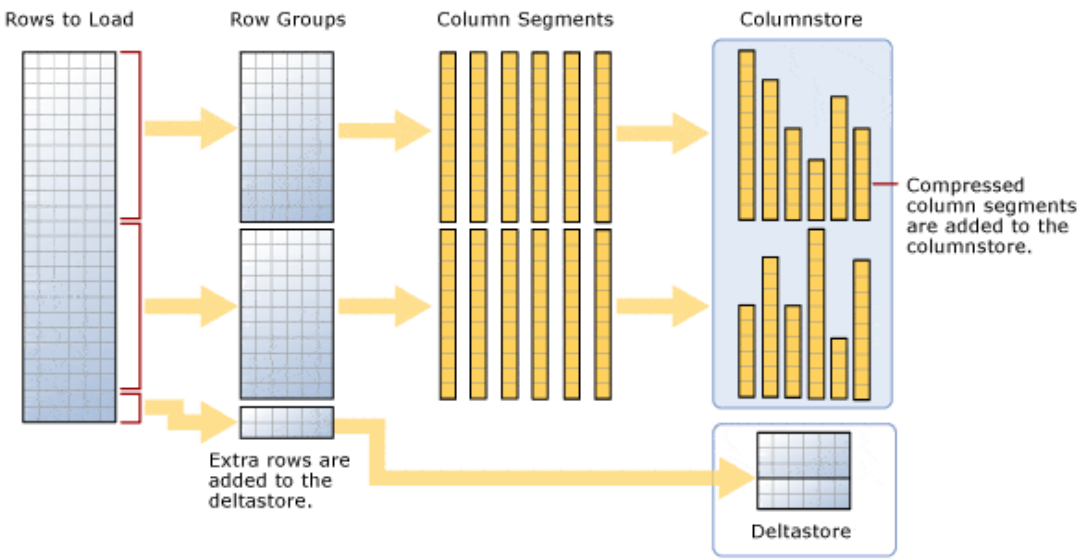
Są szczególnie użyteczne w hurtowniach danych, gdzie dokonuje się operacji na ogromnych zbiorach danych.

Terminologia:

- **Columnstore –** sposób trzymania danych w indeksie/tabeli. Zamiast domyślnego Heapu bądź B-drzewa gdzie dane są logicznie i fizycznie organizowane w postaci tabeli z wierszami i kolumnami, dane w Columnstore są fizycznie układane w kolumny. Zamiast przetrzymywać całe rekordy, na stronach dane są teraz trzymane w postaci kolumn. Dzięki tej różnicy w architekturze, dane mogą być bardzo efektywnie skompresowane, jako że w kolumnach zazwyczaj występuje wysoki poziom redundancji. Implikuje to wysokie wzrosty wydajności.
- **Rowgroup –** przed stworzeniem segmentów danych dane są „cięte" na grupy o wielkości od 102,400 do 1,048,576 rekordów.
- **Column Segemnt –** domyślna jednostka przetrzymywania danych w columnstore indexes. W rowstore jest tyle segmentów, ile kolumn ma tabela. Jeden segment przetrzymuje co najwyżej 1048576 danych. W jednym segmencie mogą występować dane tylko i wyłącznie z jednej kolumny. Segment dzieli się na strony.
- **Delta rowgroup –** sktruktura B-drzewa wykorzystywana w Columnstore index do przetrzymywania rekordów, dopóki ich liczba nie osiągnie 1,048,576. Wtedy rekordy są dodawane do columnstore. Kiedy liczba rekordów osiągnie maksimum, stan delta rowgroup zmienia się z OPEN na CLOSED. Proces zwany **tuple-mover** sprawdza, czy w delta store nie ma CLOSED rowgroups. Jeżeli proces znajdzie CLOSED rowgroup, kompresuje ją oraz wrzuca do columnstore jako COMPRESSED rowgroup. Następnie COPRESSED delta rowgroup w delta store zmienia stan na TOMBSTONE i czeka na proces tuple-mover na usunięcie, jeżeli nie ma do niego żadnych referencji.
- **Deltastore –** Columnstore Index może mieć więcen niż jedną delta rowgroup. Zbiór tych grup nazywamy Deltastore. Podczas tworzenia Columnstore index większość danych nie przechodzi przez deltastore. Jednak na końcu tego procesu może się zdarzyć, iż pozostałych rekordów jest zbyt mało, aby utworzyć rowgroup. Te rekordy są umieszczane w deltastore w postaci B-drzewa.

- **Tuple-mover –** proces działający w tle, służący przechodzenia po deltastore b-tree i kompresji grup, które osiągnęły wystarczającą ilość rekordów.
- **Batch mode –** to sposób przetwarzania kwerend, służący do przetwarzania wiele wierszy w tym samym czasie. Ten proces jest zintegrowany z indeksami kolumnowymi i znacznie optymalizuje ich działanie. Każda kolumna w batch jest przetrzymywana w postaci wektora w różnych miejscach w pamięci. Batch-mode processing używa też algorytmów, które są zoptymalizowane pod kątem wielordzeniowych procesorów.

Architektura



Załóżmy, że mamy tabele z 2.1 milionami rekordów i 6 kolumnami. Oznacza to, że Columnstore index stworzy 2x rowgroup po 1,048,576 rekordów oraz 1x delta rowgroup zawierająca 2848 rekordów. Z racji, iż każda Rowgroupa może przetrzymywać conajmniej 102,400 rekordów, delta rowgroup będzie służyła do trzymania reszty rekordów, aż nie będzie ich na tyle, aby stworzyć rowgroup. Wszystkie deltagroups są trzymane w deltastore, która ma zazwyczaj strukturę B-drzewa(taka sama jak w rowstore index). Końcowo, nasz columnstore index będzie posiadał rowgroupy składające się z jak największej liczby rekordów, aby zmiejszyć ilość operacji skanowania.

Żeby skomplikować sprawy bardziej, istnieje proces, który służy do przenoszenia delta rowgroups z delta store do columnstore index. Mówimy tutaj o wspomniamym wcześniej tuple mover. Proces ten szuka zamkniętych delta rowgroups, które są niczym innym jak grupami w deltastore które osiągnęły górny limit ich pojemności i je kompresuje. Jak widać

na pokazanym powyżej obrazku, columnstore index posiada dwie rowgroups, które następnie dzieli na segmenty(tyle segmentów ile kolumn ma tabela). To tworzy sześć segmentów po jeden milion rekordów w jednym rowgroup. Segmenty te są przetrzymywane i skompresowane oddzielnie na dysku. Silnik bazy danych dobiera się do tych segmentów oddzielnie i je przetwarza równolegle. Można również zmusić tuple-mover do przejścia po deltastore poprzez wywołanie komendy REORGANIZE na naszym columnstore index.

```
alter index ccitest_temp_cci on ccitest_temp reorganize
```

By przyspieszyć szybszy dostęp do danych, w nagłówku rowgroup są przetrzymywane metadane w postaci MIN oraz MAX value. W dodatku, jak już było wspomniane wcześniej, silnik bazy danych podczas procesowania columnstore index używa batch mode, który pozwala na przetwarzanie wiele kolumn jednocześnie. W niektórych przypadkach daje to zaskakujące rezultaty w kontekście szybkości(nawet 4-krotnie lepsze rezultaty).

Dla przykładu, jeżeli robimy agregacje, te wykonują się niebywale szybko, z racji tego iż wczytujemy do pamięci tylko kolumnę, którą agregujemy. W dodatku silnik procesuje rowgroupy równolegle(batch mode).

Kolejną ciekawą różnicą między Columnstore indexes i b-tree indexes to fakt, iż columnstore index nie posiada klucza. Możesz wrzucić do indeksu wszystkie kolumny, które zawiera tabela.

Przykład indeksu kolumnowego

Przykład pokażę na przykładowej hurtowni danych *AdventureWorksDW2016_EXT*.

W poniższym skrypcie stworzę nową, czystą bazę bez żadnych Indexes oraz Constraints.

```
SELECT * into tmpFactResellerSalesXL From [dbo].[FactResellerSalesXL_CCI]
```

Następnie tworze na niej klucz główny oraz CLUSTERED INDEX(Rowstore) na nim.

```
ADD CONSTRAINT [PK_tmp_SalesOrderNumber_SalesOrderLineNumber]
PRIMARY KEY CLUSTERED
(
    [SalesOrderNumber] ASC,
    [SalesOrderLineNumber] ASC
)
```

Włączmy przy okazji opcje STATISTICS IO oraz STATISTICS TIME na ON.

- **SET STATISTICS IO** wyświetla statystyki dotyczące ilości przetwarzanych stron. Daje to nam wartościowe informacje typu logical reads, physical reads, scans, lob reads itd.
- **SET STATISTICS TIME** wyświetla ilość czasu potrzebną na przetworzenie, kompilacje i egzekucję każdego polecenia w kwerendzie. Wynik pokazuje czas w milisekundach. To pozwala nam na zobaczenie prawdziwej różnicy czasowej między columnstore i rowstore.

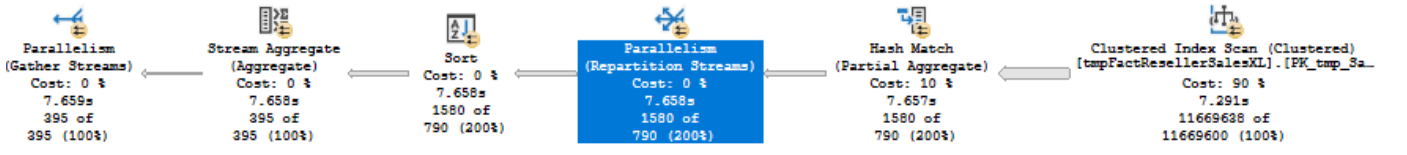
```
SET STATISTICS IO ON
SET STATISTICS TIME ON;
```

Na początku dla kontrastu wywołajmy kosztowną kowerendę z agregacjami na tabeli z domyślnym clustered index (rowstore).

```
SELECT ProductKey, sum(SalesAmount) SalesAmount, sum(OrderQuantity) ct
FROM [dbo].[tmpFactResellerSalesXL]
GROUP BY ProductKey
```

```
ALTER TABLE [dbo].[tmpFactResellerSalesXL]
ADD CONSTRAINT [PK_tmp_SalesOrderNumber_SalesOrderLineNumber]
PRIMARY KEY CLUSTERED
(
    [SalesOrderNumber] ASC,
    [SalesOrderLineNumber] ASC
)
```

Plan wykonania tej komendy składa się z 8 różnych operacji, które kończą się skanowaniem CLUSTERED INDEX.




```
Table 'tmpFactResellerSalesXL'. Scan count 5, logical reads 317704, physical reads 18894,
```

```
 SQL Server Execution Times:
   CPU time = 4812 ms,  elapsed time = 7739 ms.
```

Patrząc na wyniki powyżej, baza wykonała 5 skanów, 317704 logical reads oraz 18894 physical reads. Również CPU time jest bardzo duży. Wynosi on prawie 5 sekund. Baza przeskanowała miliony rekordów, aby zwrócić nam 395 przez nas pożądanych.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	11669638
Actual Number of Rows for All Executions	11669638
Actual Number of Batches	0
Estimated I/O Cost	233,63
Estimated Operator Cost	240,048 (90%)
Estimated CPU Cost	6,41838
Estimated Subtree Cost	240,048
Number of Executions	4
Estimated Number of Executions	1
Estimated Number of Rows for All Executions	11669600
Estimated Number of Rows Per Execution	11669600
Estimated Number of Rows to be Read	11669600
Estimated Row Size	21 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	7

Powyższą informację dostajemy po najechaniu myszką na plan egzekucji Clustered Index scan. Możemy tam dostrzec, iż Clustered index stan był przeprowadzony na Storage type of Rowstore.

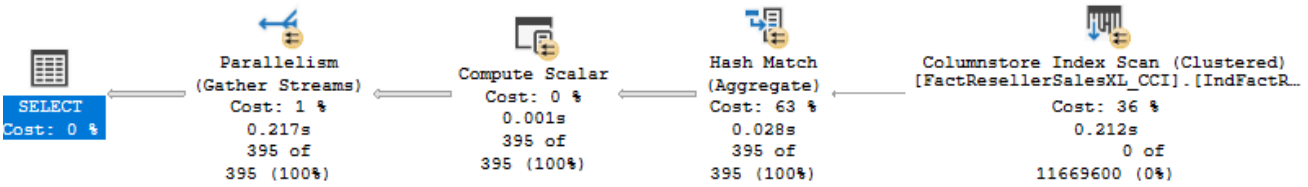
Stwórzmy teraz Clustered Columnstore Index na tej samej tabeli. Tak się miło składa, że baza AdventureWorksDW2016 posiada tabelę [dbo].[FactResellerSalesXL_CCI], która jest tą samą tabelą na której pracowaliśmy, ale posiada CCI(Clustered Columnstore Index).

Możemy stworzyć CCI poprzez naciśnięcie prawym przycieskiem myszy na folder Indexes w pożądanej tabeli, oraz wybranie opcji New Index -> Clustered Columnstore Index.

Innym sposobem jest użycie skrytpu:

```
USE AdventureWorks2012;
GO
-- Create a new table with three columns.
CREATE TABLE dbo.TestTable
    (TestCol1 int NOT NULL,
     TestCol2 nchar(10) NULL,
     TestCol3 nvarchar(50) NULL);
GO
-- Create a clustered index called IX_TestTable_TestCol1
-- on the dbo.TestTable table using the TestCol1 column.
CREATE CLUSTERED INDEX IX_TestTable_TestCol1
    ON dbo.TestTable (TestCol1);
GO
```

Następnie wywołajmy to samo zapytanie, ale na tabeli z Columnstore Index. Dodajmy, iż tabela może mieć tylko jeden columnstore index.



Jak widzimy, execution plan zmniejszyl się z 8 operacji na 5.

```
Table 'FactResellerSalesXL_CCI'. Scan count 4, logical reads 0, physical reads 0, page server reads 0, re
Table 'FactResellerSalesXL_CCI'. Segment reads 12, segment skipped 0.
```

Jak widzimy rowstore index pokazuje nam zdecydowanie więcej czytań z dysku i ramu. Mamy tutaj informacje dot. Segment reads, które zostały omówione wcześniej.

Physical Operation	Columnstore Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Batch
Estimated Execution Mode	Batch
Storage	ColumnStore
Actual Number of Rows for All Executions	0
Actual Number of Batches	0
Estimated Operator Cost	1,58719 (36%)
Estimated I/O Cost	0,945347
Estimated CPU Cost	0,641838
Estimated Subtree Cost	1,58719
Estimated Number of Executions	1
Number of Executions	4
Estimated Number of Rows for All Executions	11669600
Estimated Number of Rows to be Read	11669600
Estimated Number of Rows Per Execution	11669600
Estimated Row Size	21 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Actual Number of Locally Aggregated Rows	11669638
Node ID	3

Również mamy informacje na temat execution mode. Widzimy, że optymalizator przewidział tutaj użycie batch mode.

Kiedy używać Columnstore Indexes?

Columnstore indexes zostały zaprojektowane z myślą u dużych hurtowniach danych. Nie powinniśmy korzystać z tych indeksów bez posiadania konkretnych powodów. Stąd też najpierw powinniśmy zrobić research i przetestować czy Indeksy Kolumnowe spełnią nasze potrzeby.

Jak z każdymi innymi indeksami, istotną rzeczą jest poznanie naszych danych i to, do czego będą one używane. Ważne jest aby znać typy zapytań, które będą na danej bazie wykonywane. Wtedy możemy dobrze dopasować odpowiednie typy indeksów.

Przygotowałem kilka pytań, które musimy sobie zadań zanim zdecydujemy się na indeksy kolumnowe:

Czy tabela jest wystarczająco duża, aby indeks kolumnowy mógł przynieść efekty?

Mówiąc duża, mamy tutaj na myśli miliony rekordów. Indeks kolumnowy dzieli naszą tabele na rowgroups, które mają ponad milion rekordów każdy. Każdy rowgroup jest dzielony na segmenty kolumnowe. Stąd posiadanie indeksu kolumnowego z danymi nie przekraczającymi milion rekordów nie ma najmniejszego sensu, ponieważ tabela jest zbyt mała aby zyskała benefity z kopresji danych, które zachodzą w segmentach. Ogólna rekomendacja do używania indeksów kolumnowych to tabele hurtowni danych, tabele wielowymiarowe zawierające więcej, niż 5 milionów rekordów.

Czy twoje dane się często zmieniają?

Według rekomendacji, chcemy tabele, których dane są rzadko modyfikowane. Bardziej szczegółowo, mniej niż 10% danych w tabeli ulega modyfikacji. Duża ilość usuwanych rekordów może powodować fragmentację, która negatywnie wpływa na wydajność indeksu. Modyfikacje są w ogólności bardzo drogie z racji iż są procesowane jako usunięcie rekordu i ponowne dodanie rekordu po zmianach.

Jakie typy danych są w twojej tabeli?

Jest kilka typów danych, które nie są wspierane przez indeksy kolumnowe. Typy danych takie jak: varchar(max), nvarchar(max), varbinary(max) nie były wspierdane do 2017 przez SQL Server oraz nie są polecane do takiego typu indeksów, z racji że ich kompresja nie będzie wystarczająco efektywna. Co więcej, jeżeli korzystasz z unique identifiers(GUIDs), nie dasz radę stworzyć Indeksu Kolumnowego ponieważ te wciąż nie są wspierane.

Czy twoje kwerendy składają się w głównej mierze z agregacji czy szukania szczególnych danych?

Standardowe indeksy Rowstore oparte o B-drzewo są najlepszym rozwiązaniem do szukania pojedynczych wartości. Jeżeli używasz indexu do przyspieszenia kwerendy zawierającej klauzule WHERE, która nie specyfikuje wartości z pewnego zakresu tylko pojedyncze predykaty, to indeks kolumnowy nie przynosi tutaj żadnych korzyści.

Indeksy kolumnowe zostały stworzone z myślą o agregacjach(na przykład average price z jakiegoś okresu jakiegoś produktu). W takich przypadkach, użycie Rowstore i Columnstore Indexes może przynieść duże wzrosty w wydajności. Indeks kolumnowy byłby odpowiedzialby za grupowanie oraz agregacje, natomiast Indeks wierszowy za znalezienie produktu(index seek).

Algorytmy kompresji używanie w Indeksie Kolumnowym

- **Value scale**
  Value scale to bardzo prosta zasada. Jest on bazowany na operacjach arytmetycznych takich jak dodawanie, mnożenie czy dzielenie, które są stosowane na wszystkich wartościach w kompresowanej kolumnie. W tym przykładzie widzimy, że kolumna posiada następujące wartości.. 1023, 1002, itd. Ich częścią wspólną jest fakt, że wszystkie są większe od 1000, więc możemy w tym przypadku użyć tej liczby jako *value scale* i trzymać tylko resztę tej liczby.
  Jaką to robi różnice? Otóż do trzymania liczby typu 1023, SQL Server potrzebuje 2 bajtów pamięci. Wartości od 0 do 255 potrzebują jedynie 1 bajta. W naszym przypadku możemy zaobserwować oszczędność pamięciową rzędu 50%.

Amount
1023
1002
1007
1128
1096
1055
1200
1056

➡

Amount
23
2
7
128
96
55
200
56

Base/Scale: 1000

- **Bit Array**
  Bit Array to struktura danych, która pozwala nam na trzymanie niektórych danych zwięźle, kompresując je. Ten typ kompresji nadaje się do kolumn, których zbiór wartości jest mały, ponieważ algorytm ten tworzy kolumny bitów dla każdej możliwej wartości.

Name
Mark
Andre
John
Mark
John
Andre
John
Mark

Mark	Andre	John
1	0	0
0	1	0
0	0	1
1	0	0
0	0	1
0	1	0
0	0	1
1	0	0

W tym przykładzie mamy jedną kolumnę składającą się z ośmiu rekordów z trzema różnymi wartościami. Po prawej stronie tworzymy trzy kolumny z wartościami 1 gdy rekord zawiera tą wartość, oraz 0 w przeciwnym przypadku.

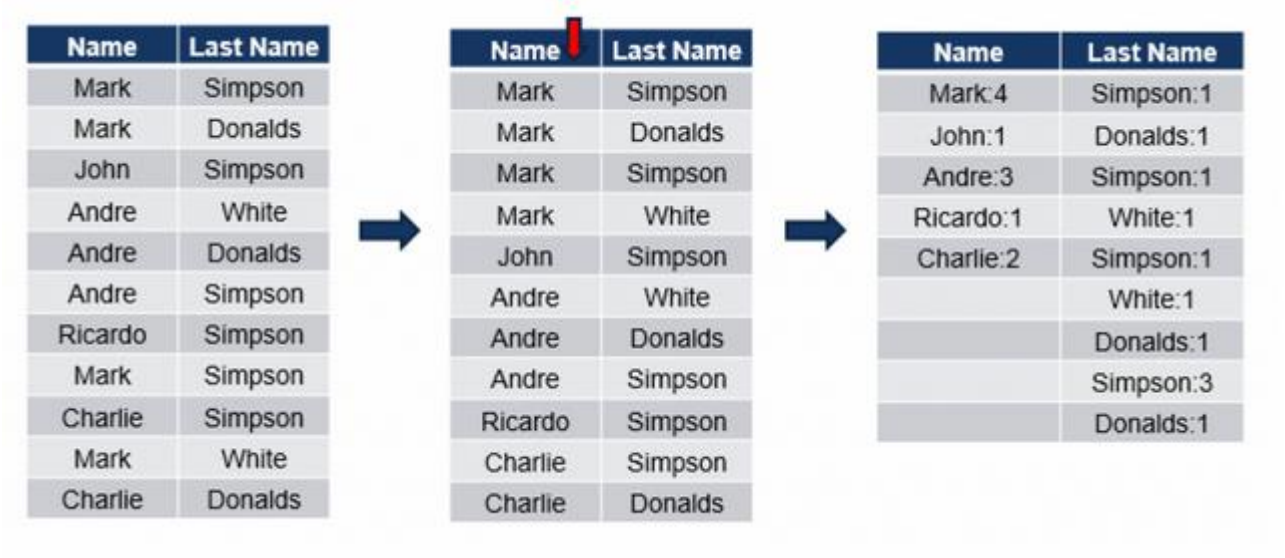
```
select name from dbo.T where name = 'Niko';
```

W powyższym zapytaniu nie musimy nawet skanować tabeli, by dostrzec, że „Niko” nie jest jest tam obecny. Dzieje się tak, gdyż SQL Server przetrzymuje Metadane na ten temat.

```
select name from dbo.T where name = 'Andre';
```

W podanym przykładzie możemy łatwo połączyć odpowiadającą Bit-Kolumnę z naszą główną kolumną by znaleźć rekordy, które nas interesują.

- **Run-Length Encoding**
  Run-length encoding to bardzo prosty typ kompresji, którego efektywność jest zależna od zbioru wartości danej kolumny(podobnie jak w Bit Array).
  Na początku algorytm sortuje dane w różnych kolumnach w celu dokonania decyzji, w której kolumnie kompresja przyniesie najlepsze efekty. Dla każdej wartości którą znajdziemy, dodajemy ją do Hash Dictionary. Dla każdego następnego powtórzenia danej wartości zwiększamy wartość tego wiersza w Hash Dictionary o 1.
  W poniższym przykładnie mamy dwie kolumny, więc dokonujemy dwóch prób sortowania i kompresji.



Name	Last Name
Mark	Simpson
Mark	Donalds
John	Simpson
Andre	White
Andre	Donalds
Andre	Simpson
Ricardo	Simpson
Mark	Simpson
Charlie	Simpson
Mark	White
Charlie	Donalds

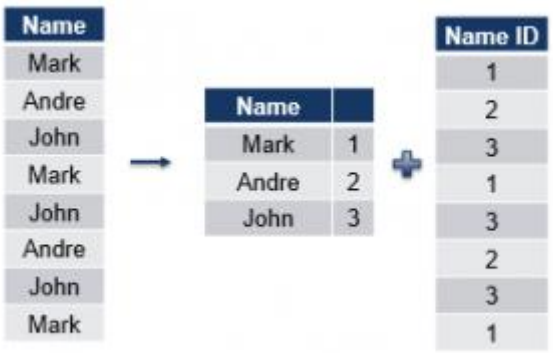
Name	Last Name
Mark	Simpson
Mark	Donalds
Mark	Simpson
Mark	White
John	Simpson
Andre	White
Andre	Donalds
Andre	Simpson
Ricardo	Simpson
Charlie	Simpson
Charlie	Donalds

Name	Last Name
Mark:4	Simpson:1
John:1	Donalds:1
Andre:3	Simpson:1
Ricardo:1	White:1
Charlie:2	Simpson:1
	White:1
	Donalds:1
	Simpson:3
	Donalds:1

Jak widzimy w kolumnie name mamy 5 różnych wartości, natomiast w Last name 9 różnych wartości.

Jeżeli chcemy znaleźć wartości rekordu nr 8, widzimy że Mark występuje 4 razy, John 1 raz, Andre 3 razy. Stąd Imię w ósmym rekordzie to Andre. Po takich samych obliczeniach w [Last Name] dowiemy się, że Imię i nazwisko to kolejno Andre Simpson.

- **Dictionary Encoding**
  Dictionary encoding bazuje na zasadzie szukania powtarzających się wartości i kodowania ich w postaci mniejszych wartości. Tak jak dla przykładu widzimy na obrazku poniżej.



Name
Mark
Andre
John
Mark
John
Andre
John
Mark

Name	
Mark	1
Andre	2
John	3

Name ID
1
2
3
1
3
2
3
1

- **Huffman Encoding**
- **Lempzel-Ziv-Welch algorithm**
- **Binary compression**