

Microsoft®



SQL Server 2012 自習書シリーズ No.15

インデックスの基礎とメンテナンス

Published: 2008 年 5 月 25 日

SQL Server 2012 更新版: 2012 年 9 月 30 日

有限会社エスキューエル・クオリティ



この文章に含まれる情報は、公表の日付の時点での Microsoft Corporation の考え方を表しています。市場の変化に応える必要があるため、Microsoft は記載されている内容を約束していません。この文書の内容は印刷後も正しいとは保障できません。この文章は情報の提供のみを目的としています。

Microsoft、SQL Server、Visual Studio、Windows、Windows XP、Windows Server、Windows Vista は Microsoft Corporation の米国およびその他の国における登録商標です。

その他、記載されている会社名および製品名は、各社の商標または登録商標です。

© Copyright 2012 Microsoft Corporation. All rights reserved.

目次

STEP 1. インデックスの概要 と自習書を試す環境について	4
1.1 インデックス (Index) の概要	5
1.2 自習書を試す環境について	7
1.3 サンプル データベース (sampleDB) の作成	8
STEP 2. インデックスの基礎	11
2.1 インデックスがない場合の検索時の内部動作	12
2.2 インデックスの作成 : CREATE INDEX	15
2.3 インデックスの内部構造	18
2.4 インデックスの削除と無効化	20
STEP 3. インデックスの構造と内部動作	23
3.1 データベースの内部構造	24
3.2 非クラスター化インデックスの正確な構造	25
3.3 インデックスの階層数の調査 : dm_db_index_physical_stats	27
3.4 インデックスを作成しても効果のない列	30
3.5 インデックスが役立たない例	33
3.6 クラスター化インデックス	36
3.7 自動的に作成されるインデックス (主キー制約と UNIQUE 制約)	42
3.8 クラスター化がある場合の非クラスター化インデックスの内部構造	43
3.9 カバリング インデックス (複合インデックス)	47
3.10 付加列インデックス (Include オプション)	52
STEP 4. インデックスの保守	57
4.1 断片化とは	58
4.2 断片化の調査 : dm_db_index_physical_stats	60
4.3 断片化の解消 (インデックスの再構築と再構成)	64
4.4 断片化の事前防止策 : FILLFACTOR	68

STEP 1. インデックスの概要 と自習書を試す環境について

この STEP では、インデックスの概要と自習書を試す環境について説明します。

この STEP では、次のことを学習します。

- ✓ インデックスの概要
- ✓ 自習書を試す環境について
- ✓ サンプル スクリプトについて

1.1 インデックス (Index) の概要

➡ インデックスとは

インデックス (Index : 索引) は、"検索のパフォーマンスを向上させるための機能"で、本の索引のようなものです。たとえば、本の索引 (巻末にある単語とページ番号の対応表) を使用すれば、本の中で調べたい単語があったときに、その単語が掲載されているページをすばやく開けますが、索引がない場合には、一度目を通したときの記憶を頼りに探したり、勘で探したり、あるいはページの先頭から該当する単語が見つかるまで探し続けたりしなければならず、大変な時間がかかってしまいます。

テーブルとインデックスの関係についても同じようことが言えます。インデックスを利用すれば、テーブル内のデータをすばやく検索することができますが、インデックスがない場合には、すべてのデータを調べなければならず、大変な時間がかかってしまいます。このことは、インデックスがない場合の内部動作を考えると理解しやすいと思います。

➡ インデックスがない場合の検索時の内部動作

インデックスがないテーブルでは、データがどこに格納されているのかが分かりません。基本的には、データを追加 (INSERT) した順に格納されていきますが、データを削除 (DELETE) した場合には、その削除された領域は再利用されるので、追加や削除が繰り返される環境では、どこにどのデータが格納されているのかが分からなくなります。これについて、次の「社員」テーブルを例に考えてみましょう。

社員番号	姓	名	性別	TEL
1	Yamada	Hanako	女性	123-xxxx
2	Suzuki	Ichiro	男性	222-xxxx
3	Abe	Makiko	女性	321-xxxx
201	Baba	Shiho	女性	121-xxxx
5	Aoki	Taro	男性	456-xxxx
202	Yamada	Taro	男性	324-xxxx
7	Etoh	Ryo	男性	500-xxxx
300	Sumino	Naoko	女性	123-xxxx
77	Yokoyama	Tetsuya	男性	222-xxxx
10	Aoki	Kenji	男性	566-xxxx
11	Aoshima	Yukiko	女性	456-xxxx
103	Matsumoto	Takahiro	男性	123-xxxx
305	Aikawa	Nana	女性	321-xxxx
207	Daigo	Tadashi	男性	221-xxxx
81	Fukui	Akira	男性	432-xxxx
:	:	:	:	:

この社員テーブルには、インデックスを作成していません。このとき、次の SELECT ステートメントを実行して、姓が「Aoki」さんの社員を検索したとします。

```
SELECT * FROM 社員 WHERE 姓 = 'Aoki'
```

SQL Server にとっては、「Aoki」さんのデータは、どこにあるのかが分からず、また、何件の「Aoki」さんがあるのかも分かりません。したがって、SQL Server が該当データを検索するには、次のよ

うに先頭から最後まで探し続けなければなりません。

社員番号	姓	姓	性別	TEL
1	Yamada	Hanako	女性	123-xxxx
2	Suzuki	Ichiro	男性	222-xxxx
3	Abe	Makiko	女性	321-xxxx
201	Baba			121-xxxx
5	Aoki			456-xxxx
202	Yamada	Taro	男性	324-xxxx
7	Etoh	Ryo	男性	500-xxxx
300	Sumino	Megumi	女性	123-xxxx
77	Yokoyama		女性	222-xxxx
10	Aoki		女性	566-xxxx
11	Aoshima	Yukiko	女性	456-xxxx
103	Matsumoto	Takahiro	男性	123-xxxx
305	Aikawa	Nana	女性	321-xxxx
207	Daigo	Tadashi	男性	221-xxxx
81	Fukui	Akira	男性	432-xxxx
:	:			:

このように、インデックスが存在しない場合には、どんなデータを検索する場合にも、必ず先頭から最後まで探し続けなければなりません。この動作は「**Table Scan : テーブル スキャン**」、「**全表走査**」、「**全件検索**」などと呼ばれています。テーブル スキャンは、データ量が 1,000 件、2,000 件など少ない場合には、(今のコンピュータの性能では) 瞬間的に結果が返りますが、1,000 万件、1 億件など大量データになった場合には大変な実行時間がかかってしまいます。

Note : テーブル スキャンの速度はどれくらい？

テーブル スキャンの速度は、ディスクの読み取り速度（実測値）に大きく依存します。たとえば、テーブル サイズが **500MB** で、ディスクの読み取り速度が **100MB / 秒** だったとすると、テーブル スキャンには最低でも **5 秒** かかるわけです（メモリにキャッシュされていない場合）。また、テーブル サイズが **5GB** だったとすると、最低でも **50 秒** はかかってしまうことになります。このように、テーブル スキャンの速度は、データ量が増えれば増えるほど、非常に時間がかかってしまうので、これを解決するための検索手法が「**インデックス**」です。

➡ グラフィカル実行プラン

SQL Server には、内部的な実行方法（テーブル スキャンが実行されたか、インデックスが利用されたかなど）を簡単に知ることができる、グラフィカル実行プランという機能があります。



この機能のおかげで、テーブル スキャンが実行される、効率の悪い検索を簡単に調べることができるので、大変便利です。Step 2 以降では、これらの利用方法を具体的に説明します。

1.2 自習書を試す環境について

➡ 必要な環境

この自習書で実習を行うために必要な環境は次のとおりです。

OS

Windows Server 2008 SP2 以降 または
Windows Server 2008 R2 SP1 以降 または
Windows Server 2012 または
Windows Vista SP2 以降 または Windows 7 SP1 以降 または Windows 8

ソフトウェア

SQL Server 2012

この自習書内での画面やテキストは、OS に Windows Server 2008 R2 (x64)、ソフトウェアに SQL Server 2012 Enterprise エディション (x64) を利用して記述しています。

その他

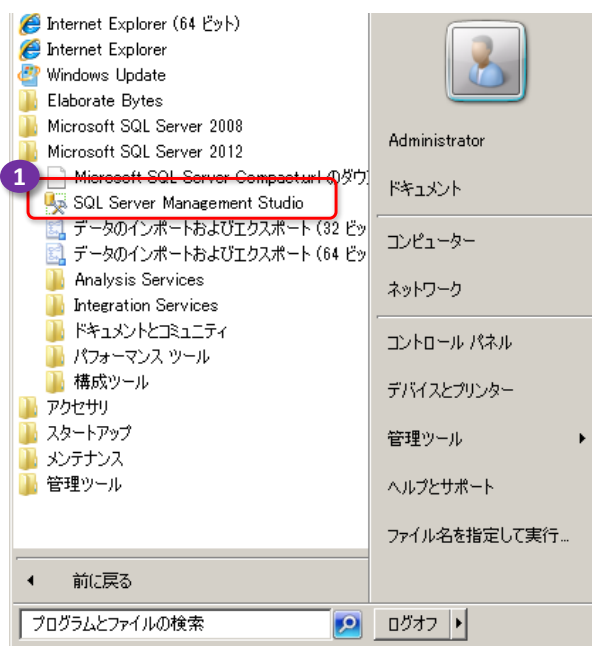
この自習書を試すには、サンプル スクリプトをダウンロードして、次のページの事前作業 (sampleDB データベースの作成) を実行しておく必要があります。

1.3 サンプル データベース (sampleDB) の作成

➡ サンプル データベース (sampleDB) の作成

この自習書では、すべての手順でサンプル スクリプトに含まれる「**sampleDB**」データベースを利用しているので、STEP2 以降を始める前に、このデータベースを SQL Server 2012 上へ作成しておいてください。作成の手順は、次のとおりです。

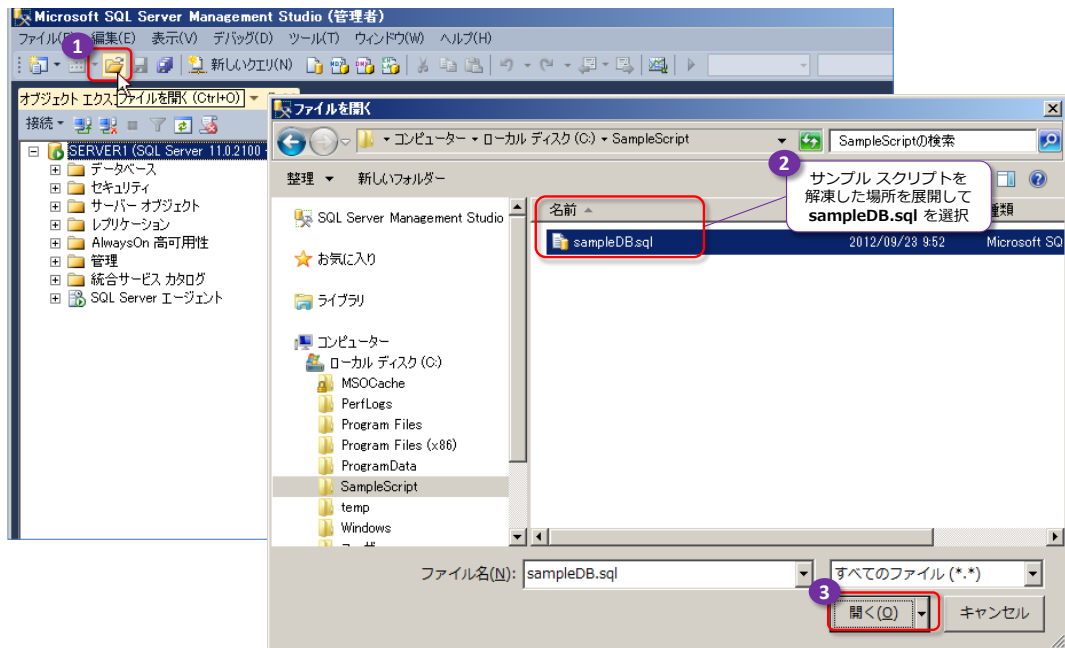
1. まずは、[スタート]メニューの[すべてのプログラム]から[Microsoft SQL Server 2012]の[SQL Server Management Studio]をクリックして、**Management Studio** を起動します。



2. 起動後、次のように[サーバーへの接続]ダイアログが表示されたら、[サーバー名]へ SQL Server の名前を入力し、[接続] ボタンをクリックします。

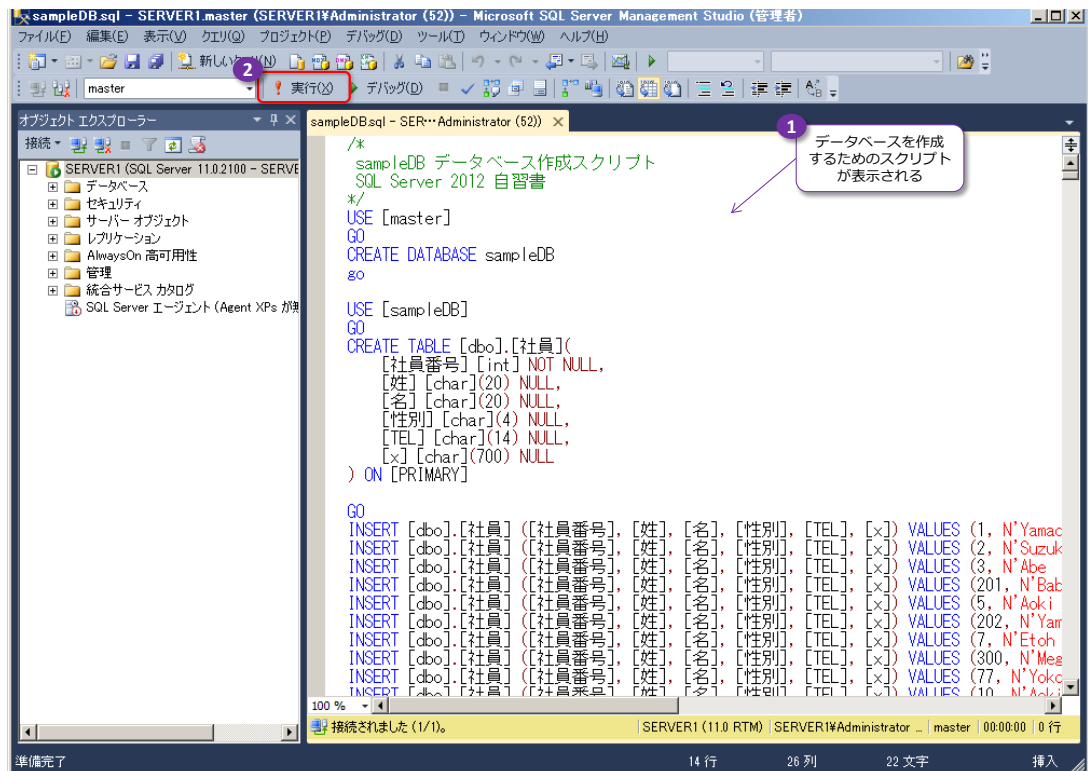


3. 接続完了後、次のようにツールバーの[ファイルを開く] ボタンをクリックします。

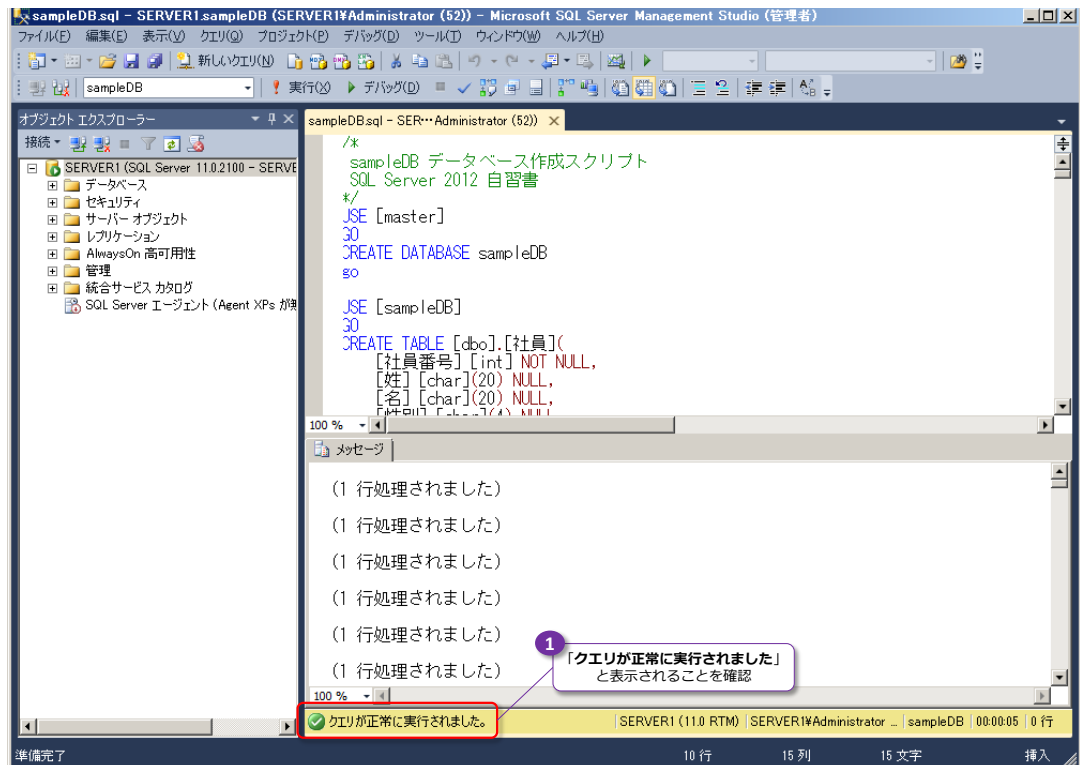


これにより、[ファイルを開く] ダイアログが表示されるので、サンプル スクリプトを解凍したフォルダーを展開して、「sampleDB.sql」ファイルを選択し、[開く] ボタンをクリックします。

4. 次のようにデータベースを作成するためのスクリプトが表示されるので、ツールバーの[実行] ボタンをクリックして、スクリプトを実行します。



5. 数秒後に実行が完了して、次のように画面下に「クエリが正常に実行されました」と表示されることを確認します。



以上でデータベースの作成が完了です。

STEP 2. インデックスの基礎

この STEP では、インデックスがない場合の内部動作（テーブル スキャン）の確認や、グラフィカル実行プランの表示方法、インデックスの作成方法などを説明します。

この STEP では、次のことを学習します。

- ✓ インデックスがない場合の検索時の内部動作の確認
- ✓ グラフィカル実行プランの表示
- ✓ インデックスの作成
- ✓ インデックスの構造
- ✓ インデックスの削除と無効化

2.1 インデックスがない場合の検索時の内部動作

◆ インデックスがない場合の検索時の内部動作

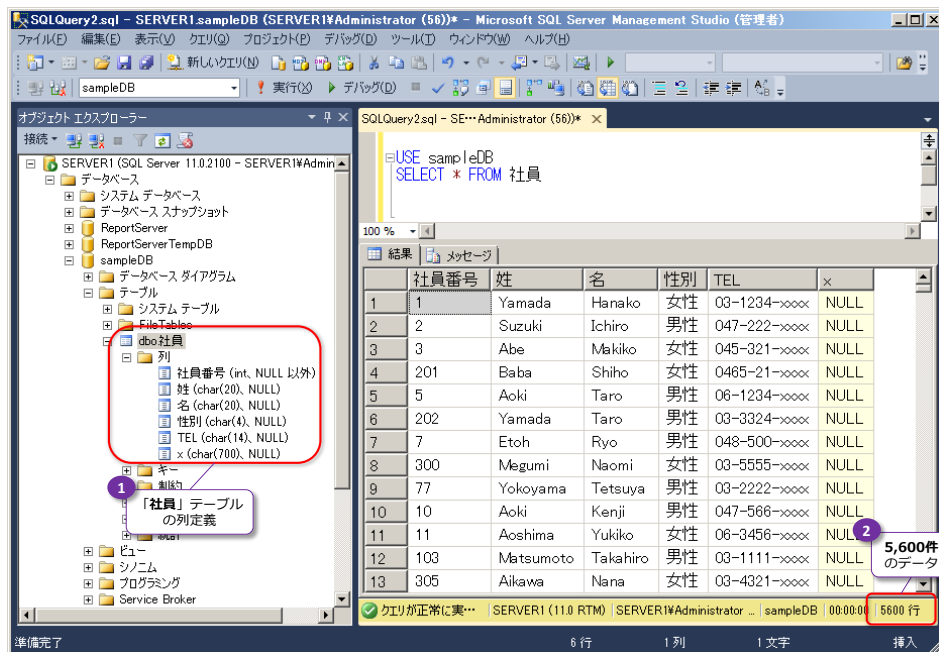
Step1 で説明したように、インデックスが存在しないテーブルでは、どんなデータを検索する場合にも、必ず先頭から最後まで探し続けなければなりません＝**テーブル スキャン**（全件検索）が実行されます。

◆ Let's Try

それでは、これを試してみましょう。

1. まずは、Management Studio を起動して、クエリ エディターを開きます。
2. 次に、Step 1 でアタッチした「sampleDB」データベース内へ作成済みの「社員」テーブルのデータを確認します。

```
USE sampleDB
SELECT * FROM 社員
```



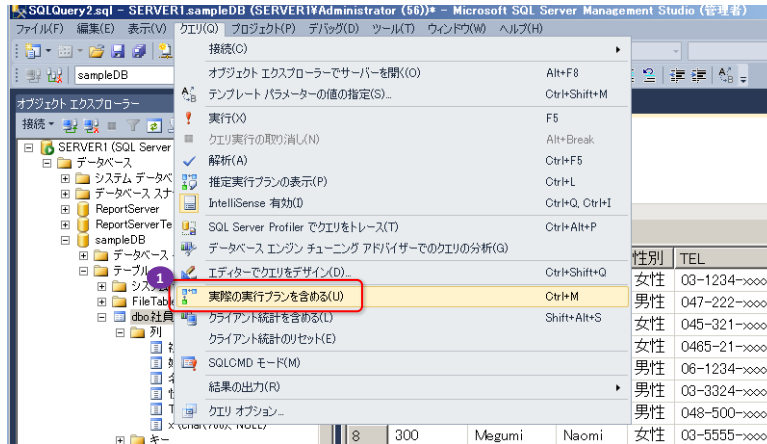
このテーブルには、**5,600 件**の社員データを格納しており、「社員番号」や「姓」、「名」、「性別」、「TEL」などを格納しています。

◆ グラフィカル実行プランで内部動作の確認

SQL Server では、内部的な処理がどのように実行されたのかを簡単に確認できる「**グラフィカル実行プラン**」という機能があります。これにより、データ検索がどのように実行されたのかを確認

することができます。では、これを試してみましょう。

1. グラフィカル実行プランは、Management Studio で、「クエリ」メニューの「**実際の実行プランを含める**」をクリックすることで表示することができます。



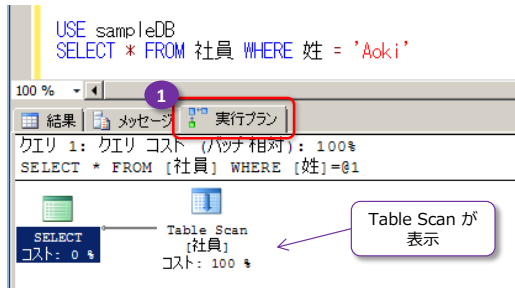
2. 続いて、クエリ エディターで次のように入力して、「社員」テーブルから「姓」列が「Aoki」さんのデータを検索してみましょう。

```
USE sampleDB
SELECT * FROM 社員 WHERE 姓 = 'Aoki'
```

USE sampleDB SELECT * FROM 社員 WHERE 姓 = 'Aoki'						
100 %						
結果 メッセージ 実行プラン						
	社員番号	姓	名	性別	TEL	x
1	5	Aoki	Taro	男性	06-1234-xxxx	NULL
2	10	Aoki	Kenji	男性	047-566-xxxx	NULL
3	386	Aoki	Chiaki	女性	03-0180-xxxx	NULL
4	670	Aoki	Senya	女性	03-1620-xxxx	NULL
5	836	Aoki	Misa	女性	03-2470-xxxx	NULL
6	1030	Aoki	Mikoto	女性	03-3455-xxxx	NULL
7	1879	Aoki	Midori	女性	03-7730-xxxx	NULL
8	2119	Aoki	Kazuo	男性	03-8935-xxxx	NULL
9	3350	Aoki	Hiroyuki	男性	045-510-xxxx	NULL
10	3397	Aoki	Yuuko	女性	045-533-xxxx	NULL
11	5704	Aoki	Taiki	男性	056-695-xxxx	NULL
12	5760	Aoki	Tetsuya	男性	057-723-xxxx	NULL

結果は、**12 件**の「Aoki」さんを取得することができました。

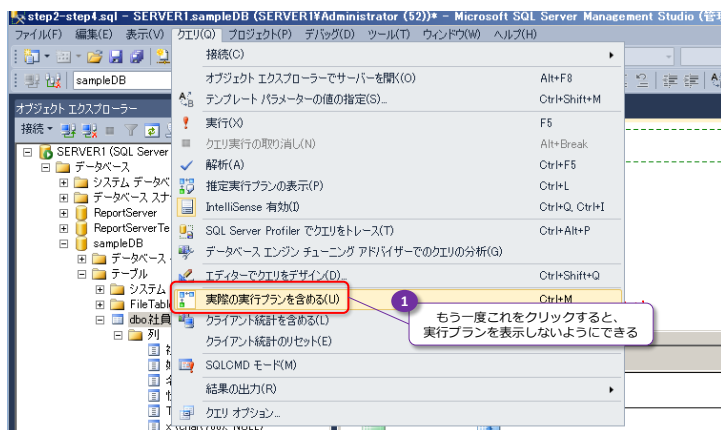
3. 次に、[実行プラン] タブをクリックして、実行プラン（内部的な実行方法）を表示します。



「Table Scan」アイコンが表示されて、テーブル スキャン（全件検索）が実行されたことを確認できます。「社員」テーブルには、インデックスを 1 つも作成していないので、SQL Server が Aoki さんのデータを探すには、テーブル スキャンをするしかありません。

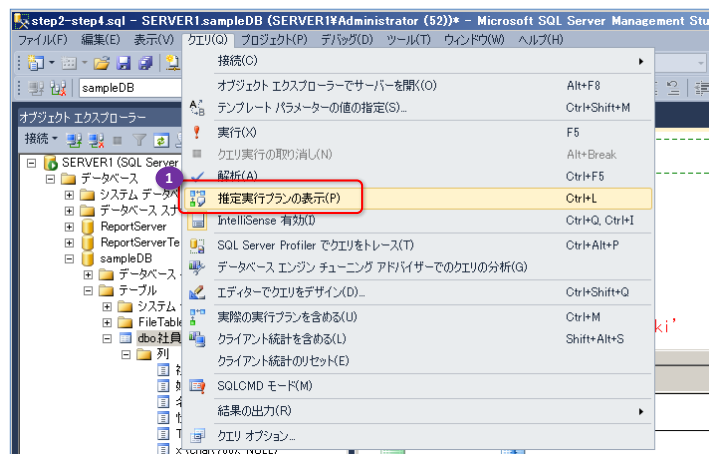
➡ グラフィカル実行プランを表示しないようにする

1. 一度設定した「**実際の実行プランを含める**」は、クエリ エディターを閉じるか、次のように、もう一度[クエリ]メニューから[**実際の実行プランを含める**]をクリックするまで有効です。



Note：推定実行プランの表示（クエリを実行せずに実行プランを確認）

グラフィカル実行プランは、クエリを実行しなくても確認することができます。これを行うには、次のように[クエリ]メニューの[**推定実行プランの表示**]をクリックします。



この機能は、実行時間の長いクエリに対して、実行プランを確認したい場合に大変便利です。

2.2 インデックスの作成： CREATE INDEX

➡ インデックスの作成

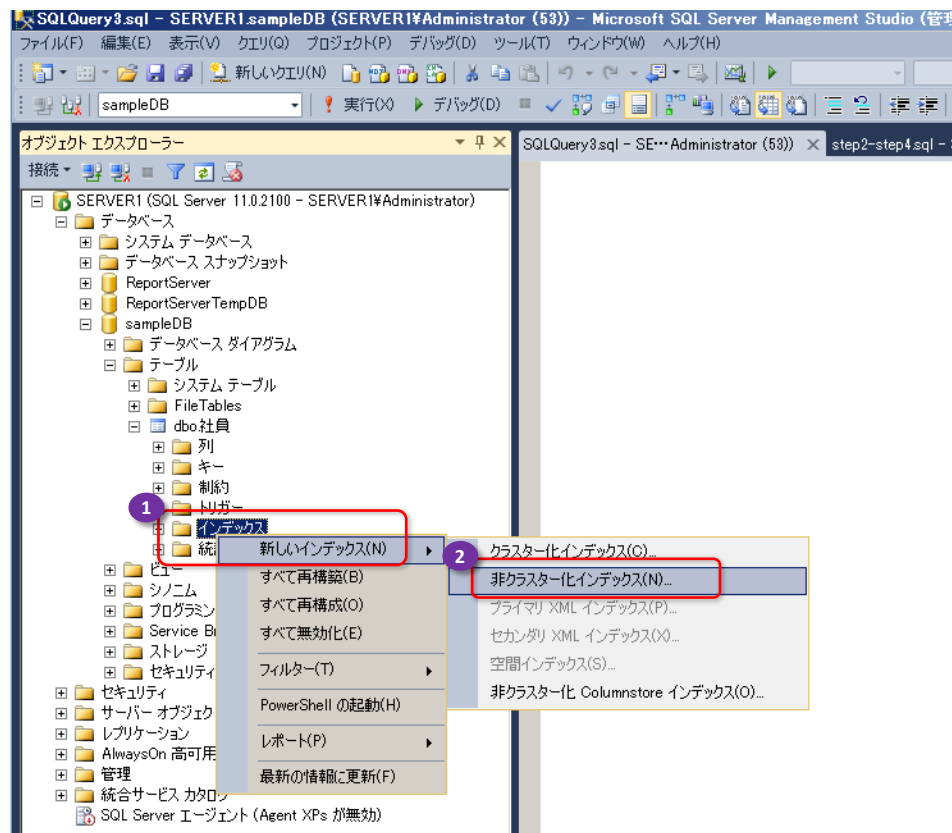
インデックスを作成するには、**CREATE INDEX** ステートメントを利用します。構文は、次のとおりです。

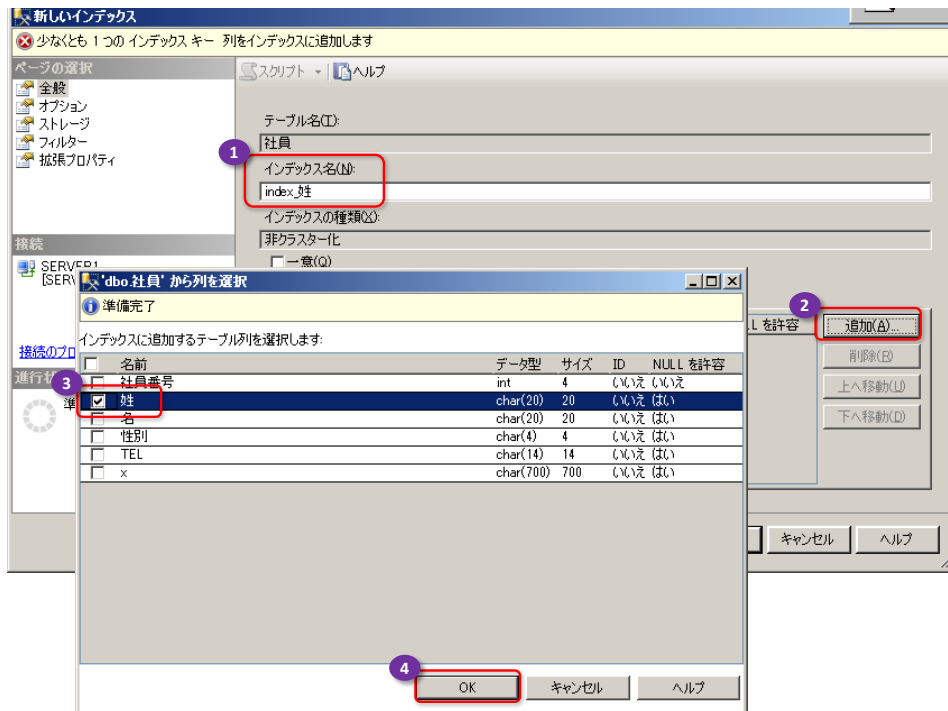
```
CREATE [CLUSTERED または NONCLUSTERED] INDEX インデックス名
ON テーブル名 (列名)
```

CLUSTERED を指定した場合は「**クラスター化インデックス**」、**NONCLUSTERED** を指定した場合は「**非クラスター化インデックス**」が作成されます（両者の違いについては、Step3 で説明します）。省略時は、非クラスター化インデックスが作成されます。

➡ GUI でのインデックスの作成

インデックスは、オブジェクト エクスプローラーで、次のように「**インデックス**」フォルダーを右クリックして、「**新しいインデックス**」をクリックしても作成することができます。



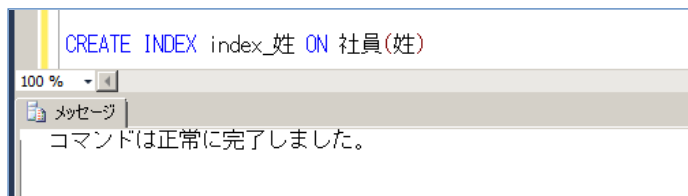


◆ Let's Try

それでは、インデックスを作成してみましょう。

1. クエリ エディターで次のように入力して、社員テーブルの「姓」列に対してインデックスを作成してみましょう。

```
CREATE INDEX index_姓 ON 社員(姓)
```

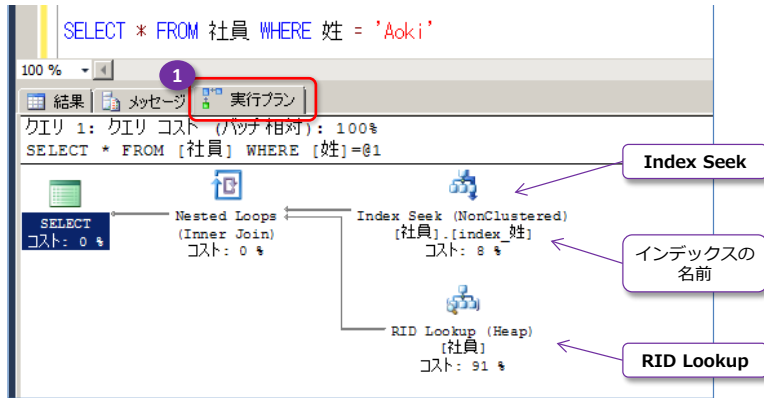


◆ インデックスの効果を確認

次に、作成したインデックスの効果を確認してみましょう。

2. [クエリ] メニューから[実際の実行プランを含める]をクリックして、グラフィカル実行プランの表示を有効にしてから、前の手順でテーブル スキャンになったクエリ（Aoki さんの検索）を実行してみましょう。

```
SELECT * FROM 社員 WHERE 姓 = 'Aoki'
```

実行後、[実行プラン] タブを開くと、今度は、テーブル スキャンではなく、**Index Seek** と **RID Lookup** というアイコンが表示されていることを確認できます。Index Seek は、インデックスを利用した検索であることを表し、利用したインデックスの名前「社員.index_姓」が表示されます。RID Lookup については、後述します。

- 次に、インデックスを作成していない、ほかの列で検索してみましょう。クエリ エディターへ次のように入力して、「名」列に「Taro」が格納されているデータを検索してみましょう。



結果は、テーブル スキャンになります。

インデックスは、列ごとに作成するものなので、作成した列以外の検索では使用されません。「名」列には、インデックスを作成していないので、SQL Server が、「Taro」というデータを探すには、テーブル スキャンをするしかないので。したがって、「名」列での検索を高速化したい場合には、この列にもインデックスを作成する必要があります。

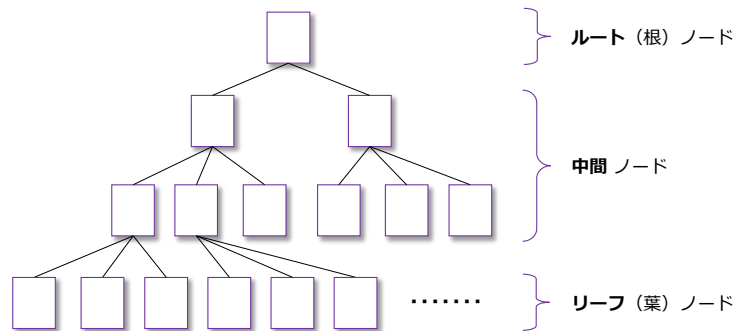
Note : 運用時を想定した効果検証を行うことが重要

開発環境など、1 台のマシンで 1 つの接続で試していたり、データの件数が少ない場合には、テーブル スキャンと Index Seek の体感的な速度差を感じることができません。これは一度読み込んだデータを「データ バッファ キャッシュ」へ格納していることも関係しているのですが、実際の運用環境では、複数テーブルの大量のデータ、そして複数のユーザーによってさまざまな処理が実行されているので速度差が顕著に現れることになります。したがって、開発/テスト時には、実際の運用時を想定したデータ量とユーザー接続数をシミュレートし、インデックスの効果を確認しておくことが重要です。

2.3 インデックスの内部構造

◆ インデックスの内部構造

インデックスは、内部的には次のようなツリー（Tree：木）構造で作成されます。

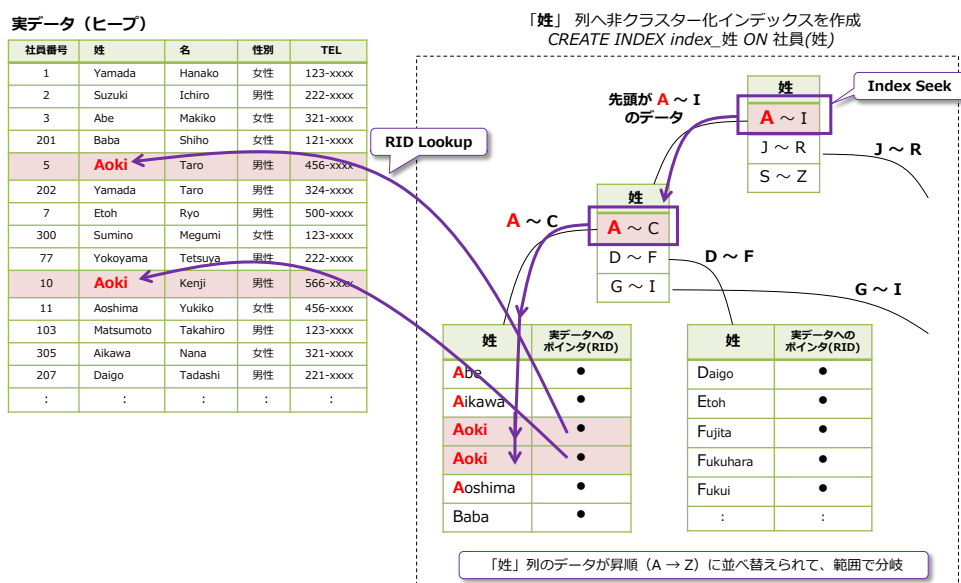


ツリーは、最上部を「**ルート**」ノード、中間部を「**中間**」ノード、最下部を「**リーフ**」ノードと呼びます。データは、昇順（小さい順）に並べ替えられて、データの範囲によって枝分かれます。ツリーの構造（枝分かれの数や何階層になるのか）は、データ量やデータサイズによって変化します。

SQL Server には、インデックスの種類として、「**クラスター化インデックス**」と「**非クラスター化インデックス**」の 2 つがありますが、両者の違いについては、Step3 で詳しく説明します。

◆ 非クラスター化インデックスの内部構造

前の手順では、社員テーブルの「姓」列に対してインデックス（非クラスター化インデックス）を作成しましたが、この場合のインデックスの内部構造は、次のようになります。



インデックス内は、「姓」列のデータが昇順に並べ替えられて、データの範囲によって枝分かれています。リーフ ノードには、実際のデータへのポインター（位置情報）が格納され、ポインタ

ーには「**行識別子**」(**RID : Row ID**)が使用されています。

このように、インデックスを構成した場合の検索は、ツリーを調べるだけで済むので、高速になります。たとえば、前の手順で行った「**WHERE 姓='Aoki'**」という検索条件であれば、「**A**」が格納されている範囲のみを調べるだけで済むのです。

なお、正確なインデックスの内部構造については、Step3 で詳しく説明します。

2.4 インデックスの削除と無効化

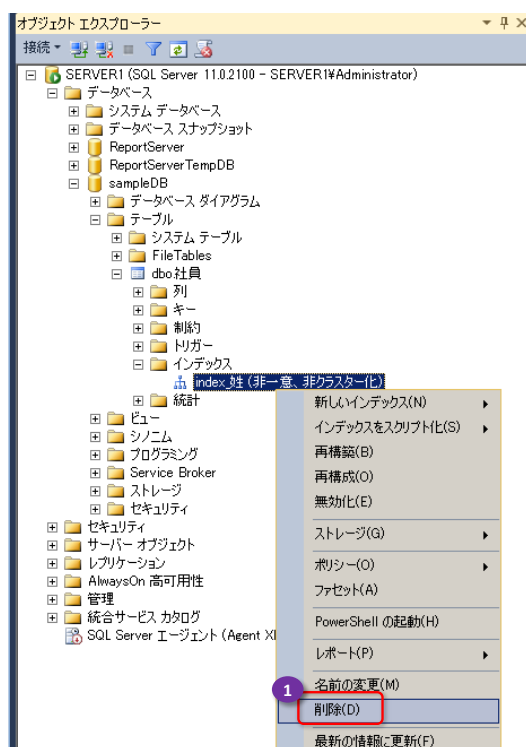
➡ インデックスの削除： DROP INDEX

インデックスを削除するには、**DROP INDEX** ステートメントを利用します。構文は、次のとおりです。

DROP INDEX テーブル名. インデックス名
 または
DROP INDEX インデックス名 **ON** テーブル名

➡ GUI でのインデックスの削除

オブジェクト エクスプローラーから、インデックスを削除したい場合には、次のように「**インデックス**」フォルダーを展開して、該当インデックスを右クリックし、「**削除**」をクリックします。



➡ インデックスの無効化： ALTER INDEX .. DISABLE

インデックスは、無効化することも可能です。無効化は、削除に似ていますが、無効化した場合は、インデックスを再作成することなく、再構築するだけで、再び有効にすることができるので、一時的に外しておきたいインデックスがある場合に便利です（インデックスの再構築については、Step 4 で説明します）。

インデックスを無効化するには、**ALTER INDEX** ステートメントを次のように利用します。

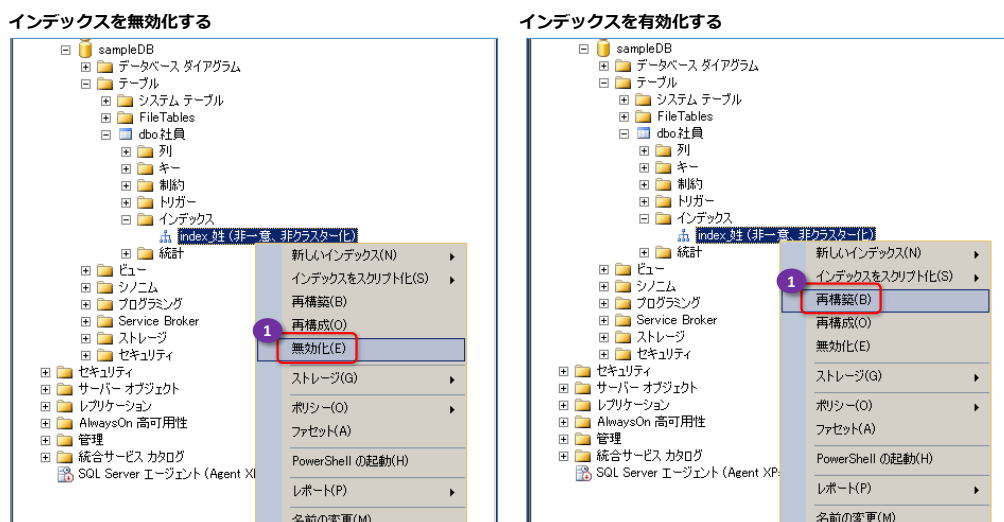
```
ALTER INDEX インデックス名
ON テーブル名 DISABLE
```

無効化したインデックスを、有効化したい場合は、次のように記述して、インデックスを再構築します。

```
ALTER INDEX インデックス名
ON テーブル名 REBUILD
```

◆ GUI でのインデックスの無効化と有効化

オブジェクト エクスプローラーから、インデックスを無効化したい場合は、[インデックス] フォルダーを展開して、該当インデックスを右クリックし、[無効化] をクリックします。また、無効化したインデックスを有効化したい場合は [再構築] をクリックします。

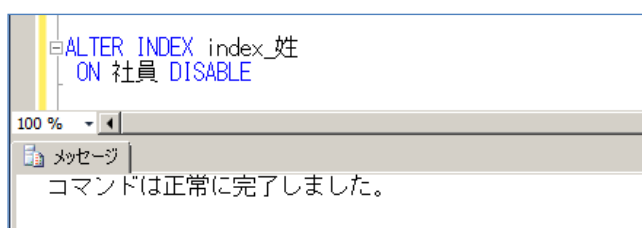


◆ Let's Try

それでは、インデックスの無効化を試してみましょう。

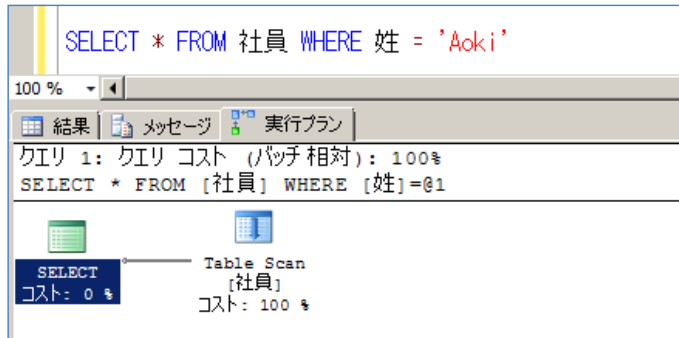
1. 前の手順で作成した「index_姓」インデックスを無効化してみましょう。

```
ALTER INDEX index_姓
ON 社員 DISABLE
```



2. 次に、インデックスが無効化されたことを確認するために、「姓」列が「Aoki」さんのデータを検索してみましょう。

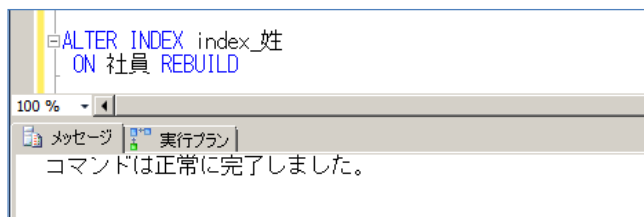
```
SELECT * FROM 社員 WHERE 姓 = 'Aoki'
```



グラフィカル実行プランを表示すると、結果は、テーブル スキャンとなり、インデックスが利用されていないことを確認できます。

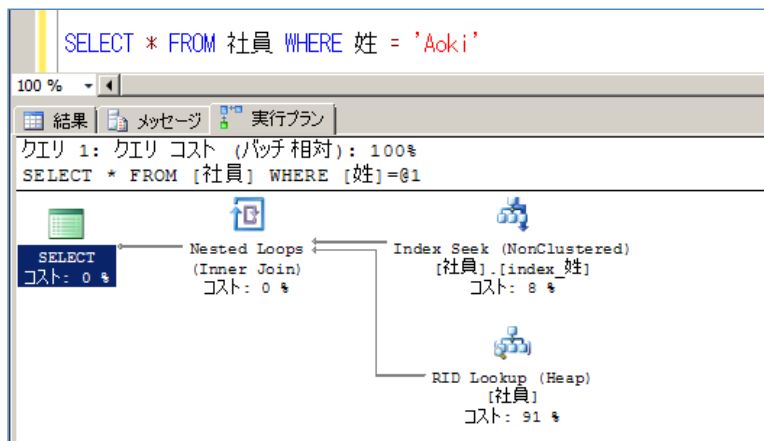
3. 次に、「index_姓」インデックスを有効化してみましょう。

```
ALTER INDEX index_姓
ON 社員 REBUILD
```



4. 有効化（再構築）が完了したら、もう一度同じ検索を実行してみましょう。

```
SELECT * FROM 社員 WHERE 姓 = 'Aoki'
```



今度は、Index Seek が表示されて、インデックスが利用されたことを確認できます。

STEP 3. インデックスの構造と内部動作

この STEP では、「非クラスター化インデックス」と「クラスター化インデックス」の違いや、「カバリング インデックス」、「付加列インデックス」について説明します。これらのインデックスを活用するには、データベースの内部構造を理解しておくことが重要です。

この STEP では、次のことを学習します。

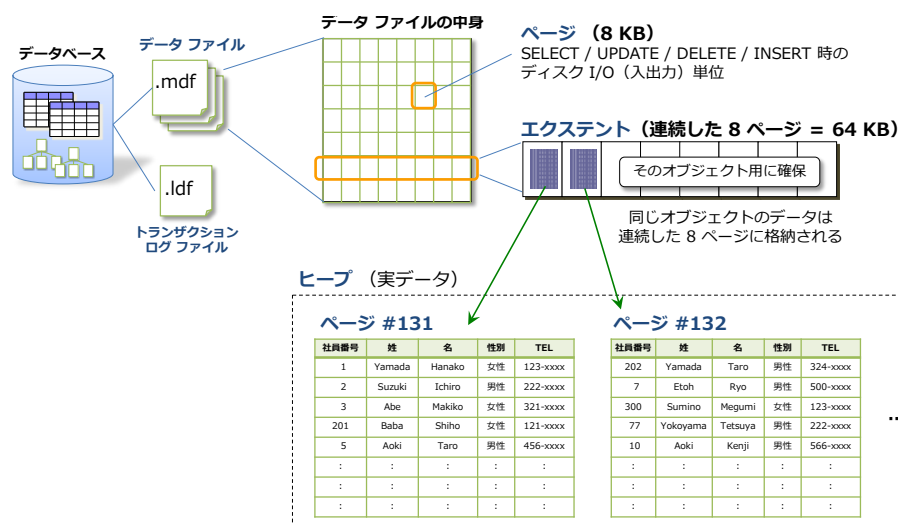
- ✓ データベースの内部構造
- ✓ 非クラスター化インデックスの構造
- ✓ インデックスの階層数の調査： `dm_db_index_physical_stats`
- ✓ インデックスを作成しても効果のない列
- ✓ インデックスが役立たない例
- ✓ クラスター化インデックス
- ✓ 自動的に作成されるインデックス（主キーと UNIQUE 制約）
- ✓ クラスター化インデックスがある場合の非クラスター化インデックスの内部構造
- ✓ カバリング インデックス（複合インデックス）
- ✓ 付加列インデックス（Include オプション）

3.1 データベースの内部構造

➡ データベースの内部構造

インデックスを理解するには、データベースの内部構造を理解しておくことが重要です。まずは、データベースの内部構造から理解していきましょう。

データベースは、次のように「**データ ファイル**」(.mdf) と「**トランザクション ログ ファイル**」(.ldf) の 2 種類で構成されます。データ ファイルには、テーブルやデータ、インデックス、ビュー、ストアド プロシージャなどが格納されます。



➡ ページとヒープ

データ ファイルは、内部的には、「**ページ**」という 8 KBの大きさに区切られています。ページは、ディスク入出力の単位です。たとえば、テーブルの行サイズが 800 バイトであれば、1 ページには 10 行分のデータを格納することができます。実データが格納されているページ全体は「**ヒープ**」と呼ばれています。

インデックス自体もデータ ファイルへ格納されるので、内部的にはページへ格納されています。したがって、ルート ノードは「**ルート ページ**」、中間ノードは「**中間ページ**」、リーフ ノードは「**リーフ ページ**」とも呼ばれます。

➡ エクステント

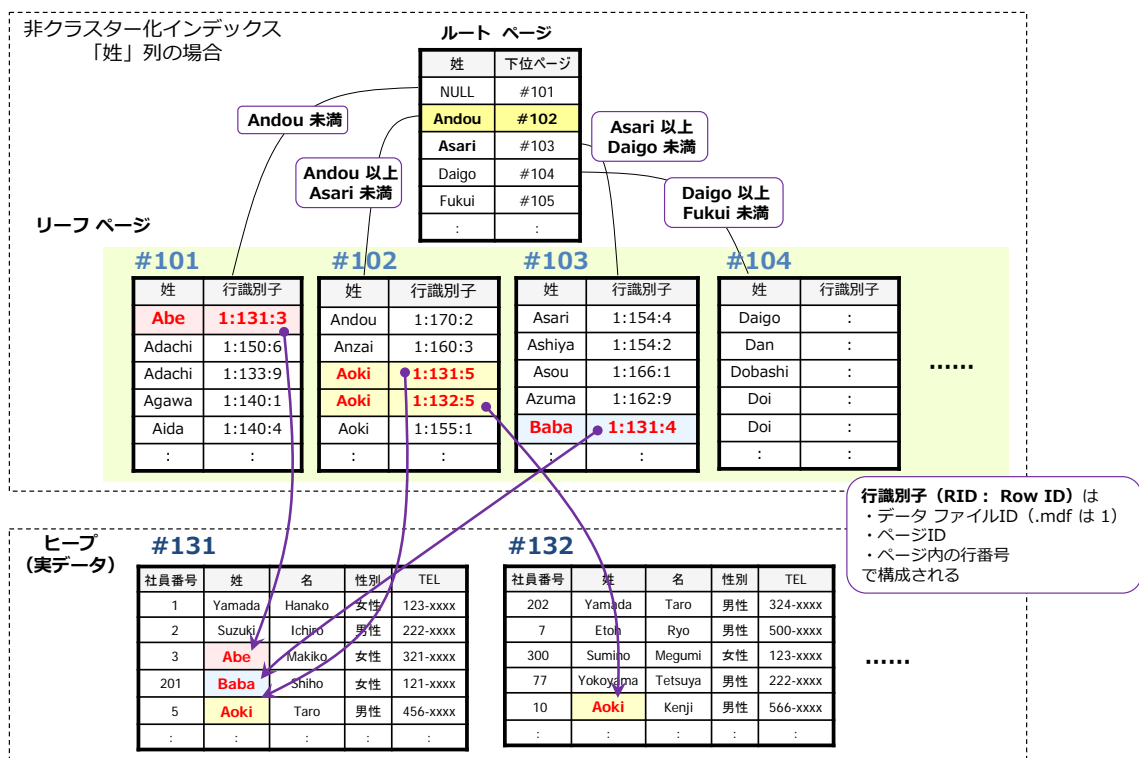
データ ファイル内の連続した 8 ページは、「**エクステント**」と呼ばれ、テーブルやインデックスに割り当てられる領域の単位になります。データが追加されると、エクステントが 1 つ確保され、その後に追加されるデータが連続した 8 ページに格納されます。これによって、同じオブジェクトのデータが連続した 8 ページに格納されることが保証されます。テーブル スキャンや後述の Index Scan などの全件検索または範囲検索は、連続したデータにアクセスするので、ページごとではなく、エクステント単位 (64KBの大きさ) でまとめて読み込むことでパフォーマンスを向上させています。

3.2 非クラスター化インデックスの正確な構造

➡ 非クラスター化インデックスの正確な構造

これまでは、インデックスがイメージしやすいようにデータの範囲で説明してきましたが、正確には、インデックス ページにはデータの範囲ではなくデータの値（インデックスを作成した列の値）が格納されます。検索時には、この値と検索条件に指定された値との大小関係によってツリー構造が走査されます。

たとえば、前の Step で作成した、「社員」テーブルの「姓」列に対する非クラスター化インデックスの構造は、次のようになります（データ量が少ない場合は、2 階層のインデックスが作成されます）。



このインデックスの構造の場合に、「Aoki」さんのデータを検索する場合は、まずルート ページへアクセスして、「Aoki」さんが、「Andou」と「Asari」さんの間のデータであることが分かるので、下位ページ（リーフ ページ）の **102** ページへアクセスします。

102 ページ内で「Aoki」さんのデータを探し、見つかったら、その行識別子 (**1:131:5**) から、データ ファイル ID が **1** (.mdf ファイル)、ページ番号が **131**、そのページ内の **5** 行目に、実際のデータがあることが分かります。

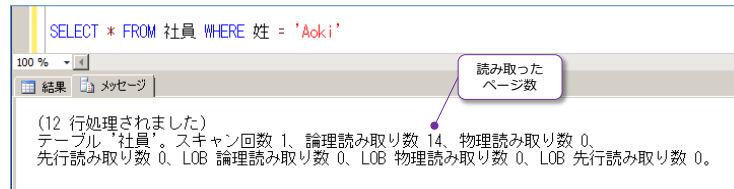
このように、インデックスを利用することで、すべてのデータを検索することなく、わずか数ページを読み取るだけで該当のデータを取得することができます。

Note : SET STATISTICS IO コマンドでクエリが読み取ったページ数を確認

クエリが実際に読み取ったページ数は、SET STATISTICS IO コマンドを利用すると調べることができます。これは、次のように実行します。

SET STATISTICS IO ON

実行後、SELECT ステートメントを実行して、[メッセージ] タブをクリックすると、次のように「論理読み取り数」が表示されて、クエリが読み取ったページ数を確認することができます。



このコマンドは、接続が終了するか、次のように SET STATISTICS IO を OFF に設定するまで有効です。

SET STATISTICS IO OFF

3.3 インデックスの階層数の調査 : dm_db_index_physical_stats

➡ インデックスの階層数の調査

dm_db_index_physical_stats 動的管理関数は、インデックスの階層数や使用しているページ数を調べることができる大変便利な関数です。構文は次のとおりです。

```
SELECT
    index_id, index_level, page_count
FROM
    sys.dm_db_index_physical_stats
    (データベースID, テーブルID, インデックスID, パーティション番号, 'スキャンモード')
ORDER BY index_id, index_level DESC
```

index_level 列で、インデックスの階層番号（リーフが 0、その上位は 1 からの連番）、**page_count** 列で、使用しているページ数を取得することができます。

第 1～第 4 引数へは、調べたいインデックスのデータベースやテーブル、インデックスを ID で指定し、NULL を指定することも可能です。NULL を指定した場合は「すべて」を意味し、たとえば、テーブル ID に NULL を指定した場合は、すべてのテーブルを対象にすることができます。

個別に ID を指定する場合は、「データベース ID」は **DB_ID** 関数、「テーブル ID」は **OBJECT_ID** 関数から取得することができます。

「インデックス ID」は、次のように **sys.indexes** カタログ ビューから取得することができます。

```
SELECT name, index_id FROM sys.indexes
WHERE object_id = OBJECT_ID('テーブル名')
```

name 列でインデックスの名前、index_id でインデックス ID を取得することができます。

➡ スキャンモード

dm_db_index_physical_stats 関数の第 5 引数のスキャン モードには、「**LIMITED**」、「**SAMPLED**」、「**DETAILED**」の 3 つがあります。インデックスの使用ページ数を調べるには、**DETAILED** モードを指定する必要があります（3 つのモードの詳しい違いについては、Step 4 で説明します）。

◆ Let's Try

それでは、これを試してみましょう。

1. まずは、**sys.indexes** カタログ ビューをクエリして、Step 2 で作成した「**index_姓**」非クラスター化インデックスのインデックス ID を調べます。

```
SELECT name, index_id, * FROM sys.indexes
WHERE object_id = OBJECT_ID('社員')
```

	name	index_id	object_id	name	index_id	type	type_desc	is_unique
1	NULL	0	245575913	NULL	0	0	HEAP	0
2	index_姓	3	245575913	index_姓	3	2	NONCLUSTERED	0

index_id 列が「3」となっているので、インデックス ID が 3 であることを確認できます。

2. 次に、**dm_db_index_physical_stats** 関数を利用して、「**index_姓**」インデックスの階層数と使用しているページ数を調べます。

```
SELECT
  index_id, index_level, page_count
FROM
  sys.dm_db_index_physical_stats
  (DB_ID('sampleDB'), OBJECT_ID('社員'), 3, NULL, 'DETAILED')
ORDER BY index_id, index_level DESC
```

	index_id	index_level	page_count
1	3	1	1
2	3	0	24

「1」と「0」で2階層だといことが分かる

階層ごとのページ数 (ルートが1ページ、リーフが24ページ)

index_level 列には、「1」と「0」が表示され、「0」はリーフ ページ、「1」以上は中間ページより上の階層になり、一番大きい値がルート ページになります。したがって、今回は「1」が一番大きい値なので、これがルート ページになります（インデックスが 2 階層で構成されていることが分かります）。

Note : 隠しコマンドの DBCC IND と DBCC PAGE でインデックスの中身を見る

インデックスは、dm_db_index_physical_stats 関数で階層数と使用ページ数を調べることができますが、インデックスの中身までは見ることはできません。インデックスの中身を見るには、ページの中身を見るための隠しコマンド（ヘルプへ記載されていないコマンド）の「**DBCC IND**」と「**DBCC PAGE**」を利用する必要があります。**DBCC IND** コマンドでは、インデックスが使用しているページ番号を取得することができ、たとえば、次のように入力すると、index_姓（インデックス ID が 3）の使用しているページ番号を取得することができます。

DBCC IND (sampleDB, 社員, 3)

	PageFID	PagePID	IAMFID	IAMPID	ObjectID	IndexID	PartitionNumber	PartitionID	iam_chain_type	PageType	IndexLevel
7	1	177	1	169	245575913	3	1	72057594039238656	In-row data	2	0
8	1	178	1	169	245575913	3	1	72057594039238656	In-row data	2	0
9	1	179	1	169	245575913	3	1	72057594039238656	In-row data	2	0
10	1	180	1	169	245575913	3	1	72057594039238656	In-row data	2	1
11	1	181	1	169	245575913	3	1	72057594039238656	In-row data	2	0
12	1	985	1	169	245575913	3	1	72057594039238656	In-row data	2	0
13	1	986	1	169	245575913	3	1	72057594039238656	In-row data	2	0

結果の **PagePID** 列は、インデックスへ割り当てられているページ番号で、**IndexLevel** 列は「0」がリーフ ページで、「1」以上は、リーフより上の階層を表します。

ここで調べたページ番号を **DBCC PAGE** コマンドの第 3 引数へ与えることで、そのページの中身を見ることができます。たとえば、次のように入力すると、ページ番号「179」の中身を参照することができます。

DBCC PAGE (sampleDB, 1, 179, 3)

	FileId	PageId	Row	Level	姓 (key)	HEAP RID (key)	KeyHashValue	Row Size
1	1	179	0	0	Katou	0xEE0000001000200	(6e5d4dd16590)	32
2	1	179	1	0	Katou	0xF40200001000500	(d484d0bccfa3)	32
3	1	179	2	0	Katou	0xF50100001000100	(ba80ddb0513d)	32
4	1	179	3	0	Katou	0xF60100001000100	(51c24583f362)	32
5	1	179	4	0	Katou	0xF60100001000300	(6c77c40b046c)	32
6	1	179	5	0	Katsura	0x260200001000100	(6652e97de8ac)	32
7	1	179	6	0	Katsuragawa	0x870100001000400	(5d14f4fbec89)	32
8	1	179	7	0	Katsuragi	0x770100001000200	(13c9f4ed585f)	32
9	1	179	8	0	Katsuragi	0x920100001000600	(5ae4809422c7)	32
10	1	179	9	0	Katsuragi	0x9A0200001000100	(ce91780e9560)	32

インデックスを
設定した列の値

行識別子 (RID)
ポインタ

3.4 インデックスを作成しても効果のない列

➡ インデックスを作成しても効果のない列

インデックスを作成すれば、基本的にはパフォーマンスが向上しますが、作成してもパフォーマンスの向上が見込めないケースがあります。代表的なのは次の 2 つです。

- WHERE 句の検索条件にほとんど使用されない列

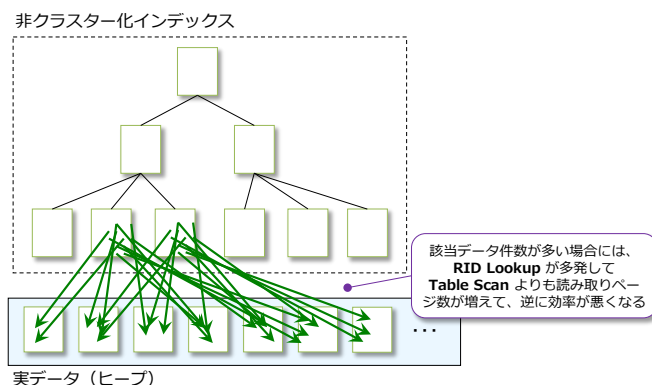
検索条件にほとんど使用されない列に対してインデックスを作成しても、効果はありません。逆に、作成したことによってパフォーマンスの低下を招く恐れがあります。インデックスを作成すると、インデックスを作成した列のデータを更新した際に、インデックスそのもの（ツリー構造）も更新されるからです。したがって、検索条件にほとんど使用されない列で、かつ更新頻度が過剰な列に対してはインデックスを作成しないことをお勧めします。

Note： 使用されていないインデックスを探すには？

動的管理ビューの「`dm_db_index_usage_stats`」を利用すると、SQL Server の起動後に、1 度も使用されていないインデックス（未使用のインデックス）や、使用回数の少ないインデックスを探すことができます。具体的な利用方法については、本自習書シリーズの「**監視ツールの基本操作**」で詳しく説明しています。

- 検索条件に該当するデータが大量にある場合

インデックスは、大量のデータから 1～数百件のデータを取り出すときに最も効果があります。検索条件に該当するデータが大量にある場合には、インデックスの効果が得られません。該当データが多い場合には、**RID Lookup** で（リーフレベルのポインターを使用して）実際のデータを取得すると、テーブル スキャンよりも効率が悪くなることからです。



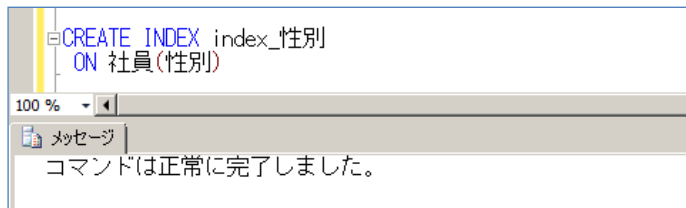
SQL Server は、インデックスを使用するよりもテーブル スキャンの方が効率が良いと判断した場合には、テーブル スキャンを使用してデータ検索を実行します。

➡ Let's Try：「性別」列ヘインデックスを作成した場合の動作

それでは、これを試してみましょう。

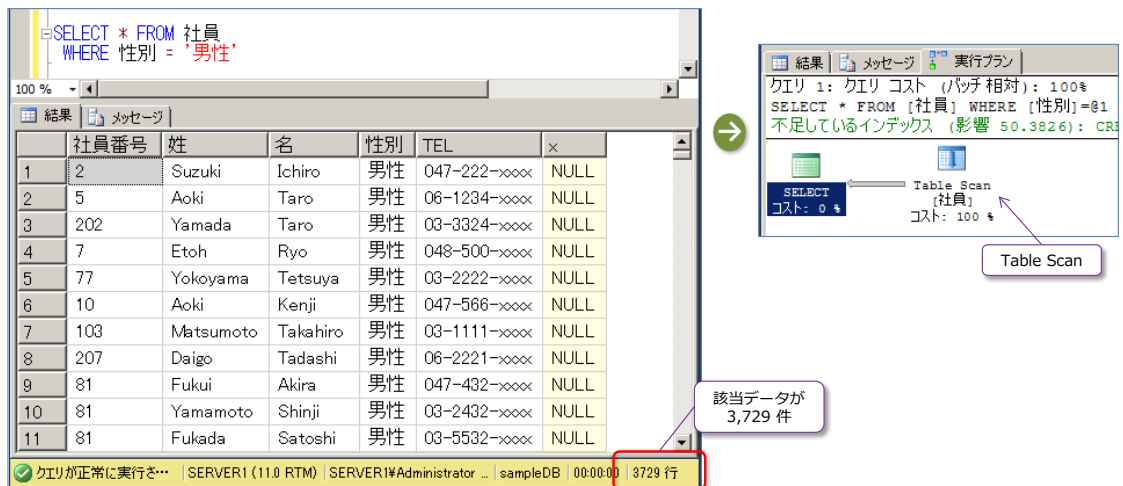
1. まずは、社員テーブルの「**性別**」列ヘインデックスを作成してみましょう。

```
CREATE INDEX index_性別
ON 社員 (性別)
```



2. 次に、グラフィカル実行プランの表示を有効にして、性別が「男性」の社員のみを検索してみましょう。

```
SELECT * FROM 社員
WHERE 性別 = '男性'
```

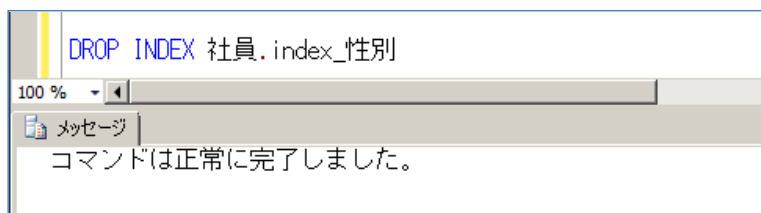


結果は、テーブル スキャンが実行されていることを確認できます。

このようにデータの種類の少ない場合（性別列は「男性」と「女性」の 2 種類）は、検索条件に該当するデータが自ずと多くなるので、インデックスを作成しても利用されません。SQL Server は、インデックスを使用するよりもテーブル スキャンの方が効率が良いと判断した場合は、テーブル スキャンを使用してデータ検索を実行するのです。

3. このインデックス「index_性別」は、不要なので削除しておきます。

```
DROP INDEX 社員.index_性別
```



◆ 検索条件に該当するデータ件数が多い場合

続いて、「姓」列に対して、検索条件に該当するデータ件数が多い場合に、インデックスが利用されるかどうかを確認してみましょう。

1. データ件数を多く取得するために、次のように **LIKE** 演算子を利用して、姓が「A」で始まる社員を検索してみましょう。

```
SELECT * FROM 社員
WHERE 姓 LIKE 'A%'
```

The screenshot shows the SQL Server Enterprise Manager interface. The query editor contains the query: `SELECT * FROM 社員 WHERE 姓 LIKE 'A%'`. The results grid displays 11 rows of employee data. The execution plan window on the right shows a single step: **Table Scan** (社員) with a cost of 100. A callout box points to the results grid, stating "該当データが 293 件" (293 rows of matching data). The status bar at the bottom indicates "293 行" (293 rows).

	社員番号	姓	名	性別	TEL	x
1	3	Abe	Makiko	女性	045-321-xxxx	NULL
2	5	Aoki	Taro	男性	06-1234-xxxx	NULL
3	10	Aoki	Kenji	男性	047-566-xxxx	NULL
4	11	Aoshima	Yukiko	女性	06-3456-xxxx	NULL
5	305	Aikawa	Nana	女性	03-4321-xxxx	NULL
6	336	Asai	Goichi	男性	058-816-xxxx	NULL
7	337	Aritou	Kouichi	男性	058-816-xxxx	NULL
8	338	Aoyama	Yukihito	男性	058-817-xxxx	NULL
9	364	Aihara	Miyuki	女性	03-0070-xxxx	NULL
10	367	Aihara	Rei	女性	03-0085-xxxx	NULL
11	380	Andou	Merijen	女性	03-0150-xxxx	NULL

結果が表示され、293 件のデータが検索条件に該当し、テーブル スキャンが実行されたことを確認できます。このインデックスは、「姓='Aoki'」という検索条件のときには **Index Seek** が実行されていました。

このように、検索条件に該当するデータ件数が多い場合には、インデックスが利用されず、テーブル スキャンが実行されます。

Note : クエリ オプティマイザーによる最適な実行プランの選択

データの検索時に、どのインデックスを利用するかや、インデックスを利用しないでテーブル スキャンを実行するかは、SQL Server の「**クエリ オプティマイザー**」という機能（プログラム）が判断します。クエリ オプティマイザーは、検索条件に該当するデータを取得するために、インデックスよりもテーブル スキャンを利用したほうが「**高速**」だと判断した場合は、テーブル スキャンを実行します。オプティマイザーは、**I/O コスト**（ディスク入出力の負荷がどれくらいか）、と **CPU コスト**（CPU への負荷がどれくらいか）などをもとに判断しています。

3.5 インデックスが役立たない例

➡ インデックスが役立たない例

条件に該当するデータ件数が少ない場合でも、SQL ステートメントの書き方が悪いと、インデックスを利用しない (Index Seek が実行されない) 非効率な検索が行われます。これは、次の 3 つのケースです。

- LIKE 演算子を利用する際に、先頭に % (ワイルドカード文字) を指定している場合

```
SELECT * FROM 社員 WHERE 姓 LIKE '%oki'
```

- 演算子の左辺へ「関数」や「計算式」を記述している場合

```
SELECT * FROM 社員 WHERE UPPER(姓) = 'Aoki'
```

- 列へ設定してある照合順序とは、異なる照合順序を指定している場合

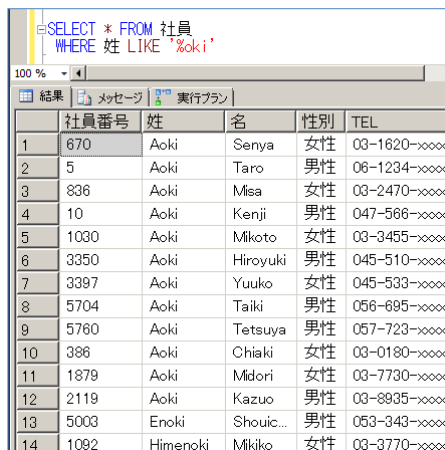
```
SELECT * FROM 社員 WHERE 姓 = 'Aoki' COLLATE Japanese_CS_AS
```

➡ Let's Try : LIKE の先頭に % を指定した場合

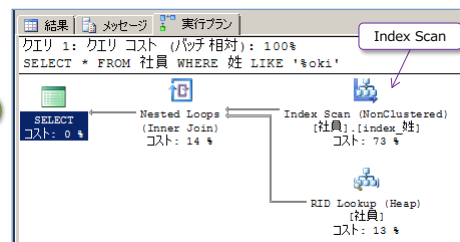
それでは、これを試してみましょう。

1. まずは、LIKE 演算子で先頭をワイルドカード指定した場合の検索を試してみましょう。ここでは、検索条件に「姓 LIKE '%oki'」と指定して、姓が「oki」で終わる社員を検索してみます。

```
SELECT * FROM 社員
WHERE 姓 LIKE '%oki'
```



	社員番号	姓	名	性別	TEL
1	670	Aoki	Senya	女性	03-1620-xxxx
2	5	Aoki	Taro	男性	06-1234-xxxx
3	836	Aoki	Misa	女性	03-2470-xxxx
4	10	Aoki	Kenji	男性	047-566-xxxx
5	1030	Aoki	Mikoto	女性	03-3455-xxxx
6	3350	Aoki	Hiroyuki	男性	045-510-xxxx
7	3397	Aoki	Yuuko	女性	045-533-xxxx
8	5704	Aoki	Taiki	男性	056-695-xxxx
9	5760	Aoki	Tetsuya	男性	057-723-xxxx
10	386	Aoki	Chiaki	女性	03-0180-xxxx
11	1879	Aoki	Midori	女性	03-7730-xxxx
12	2119	Aoki	Kazuo	男性	03-8935-xxxx
13	5003	Enoki	Shouic...	男性	053-343-xxxx
14	1082	Himenoki	Mikiko	女性	03-3770-xxxx



グラフィカル実行プランには、**Index Scan** と **RID Lookup** が表示され、**Index Seek** ではないことを確認できます。

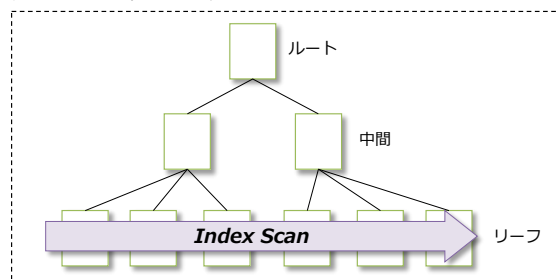
このように、LIKE 演算子の先頭がワイルドカード文字の場合は、Index Seek が実行されません。

➡ Index Scan とは



Index Scan (インデックス スキャン) は、Index Seek のアイコンと非常に似ていますが、内部動作は大きく異なります。Index Scan は、インデックスのツリー構造を利用せずに、リーフ ノードを横方向へ全スキャンして、該当データを探し出す内部動作です。したがって、Index Scan は、Index Seek よりも多くのページにアクセスすることになるので、効率の悪い検索方法です (Index Seek よりも実行時間がかかります)。

非クラスター化インデックス



Index Scan は、リーフページを横方向へ全スキャン

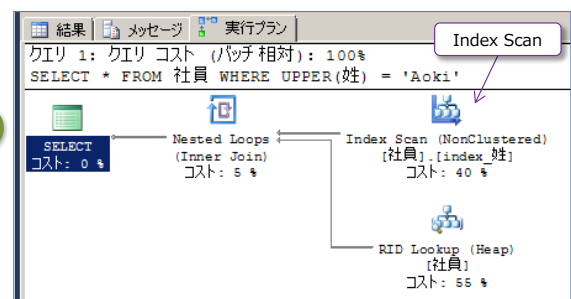
➡ 関数処理をしている場合

次に、WHERE 句の左辺で関数を使用している場合の検索を試してみましょう。

1. ここでは、**UPPER** 関数を利用して、検索を実行してみましょう。

```
SELECT * FROM 社員
WHERE UPPER(姓) = 'Aoki'
```

	社員番号	姓	名	性別	TEL
1	670	Aoki	Senya	女性	03-1620-xxxx
2	5	Aoki	Taro	男性	06-1234-xxxx
3	836	Aoki	Misa	女性	03-2470-xxxx
4	10	Aoki	Kenji	男性	047-566-xxxx
5	1030	Aoki	Mikoto	女性	03-3455-xxxx
6	3350	Aoki	Hiroyuki	男性	045-510-xxxx
7	3397	Aoki	Yuuko	女性	045-533-xxxx
8	5704	Aoki	Taiki	男性	056-695-xxxx
9	5760	Aoki	Tetsuya	男性	057-723-xxxx
10	386	Aoki	Chiaki	女性	03-0180-xxxx
11	1879	Aoki	Midori	女性	03-7730-xxxx
12	2119	Aoki	Kazuo	男性	03-8935-xxxx



結果は、**Index Scan** と RID Lookup になり、Index Seek が実行されていないことを確認


できます (Index Scan によって、リーフページを全スキャンしているので、効率の悪い検索が実行されています)。

➡ 照合順序が異なる場合

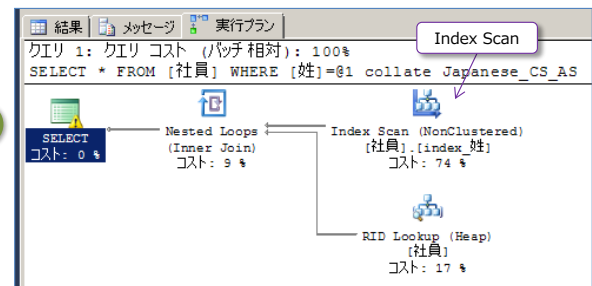
次に、異なる照合順序を指定した場合の検索を試してみましょう。

1. 列に設定してある照合順序は、**Japanese_CI_AS** (大文字と小文字を区別しない) なので、これとは異なる照合順序 **Japanese_CS_AS** (大文字小文字を区別する) を指定して検索を実行してみましょう。

```
SELECT * FROM 社員
WHERE 姓 = 'Aoki' COLLATE Japanese_CS_AS
```



	社員番号	姓	名	性別	TEL
1	670	Aoki	Senya	女性	03-1620-xxxx
2	5	Aoki	Taro	男性	06-1234-xxxx
3	836	Aoki	Misa	女性	03-2470-xxxx
4	10	Aoki	Kenji	男性	047-566-xxxx
5	1030	Aoki	Mikoto	女性	03-3455-xxxx
6	3350	Aoki	Hiroyuki	男性	045-510-xxxx
7	3397	Aoki	Yuuko	女性	045-533-xxxx
8	5704	Aoki	Taiki	男性	056-695-xxxx
9	5760	Aoki	Tetsuya	男性	057-723-xxxx
10	386	Aoki	Chiaki	女性	03-0180-xxxx
11	1879	Aoki	Midori	女性	03-7730-xxxx
12	2119	Aoki	Kazuo	男性	03-8935-xxxx



結果は、**Index Scan** と RID Lookup になり、Index Seek が実行されていないことを確認できます。

なお、照合順序については、本自習書シリーズの「**Transact-SQL 入門**」編で詳しく説明していますので、こちらもぜひご覧いただければと思います。

3.6 クラスター化インデックス

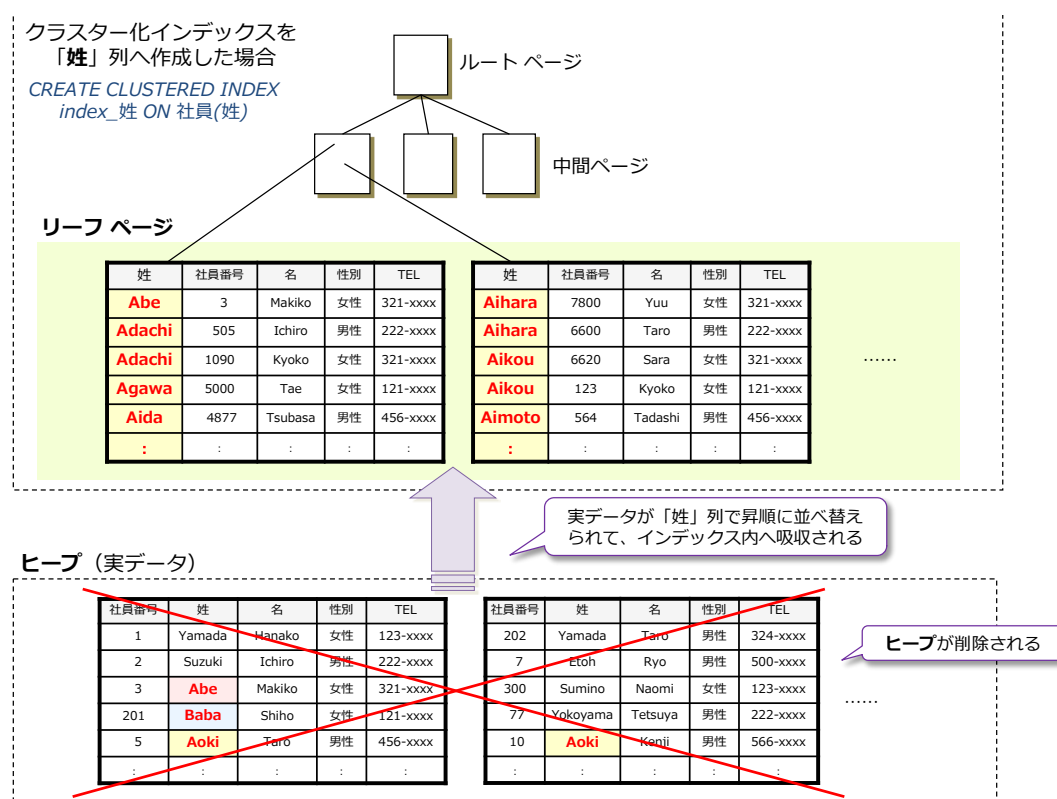
➡ クラスター化インデックス

今までは、「非クラスター化インデックス」について説明してきましたが、ここからは、もう 1 つのインデックス「**クラスター化インデックス**」について説明します。

非クラスター化インデックスとクラスター化インデックスの違いは、非クラスター化インデックスではリーフ レベルへ実際のデータへのポインター（行識別子 : RID）が格納されるのに対して、クラスター化インデックスの場合は、リーフ レベルにはポインターではなく実際のデータそのものを格納するところです。

クラスター化インデックスを作成すると、実際のデータが物理的に並べ替えられて、クラスター化インデックスのリーフ ページとなり、ヒープが削除されます。

たとえば、社員テーブルの「姓」列に対してクラスター化インデックスを作成すると、次のようになります。



このように、クラスター化インデックスでは、実際のデータがインデックス内へ吸収されるので、非クラスター化インデックスよりも検索のパフォーマンスが向上します。

➡ クラスター化インデックスの注意点

クラスター化インデックスは、実際のデータをインデックス内へ格納するので、テーブル内で 1 つしか作成することができません。また、インデックスの作成には、実際のデータの並べ替えも発

生するので、非クラスター化インデックスを作成するよりも、作成時間がかかります（これは Step 4 で説明するインデックスの再構築時間にも当てはまります）。

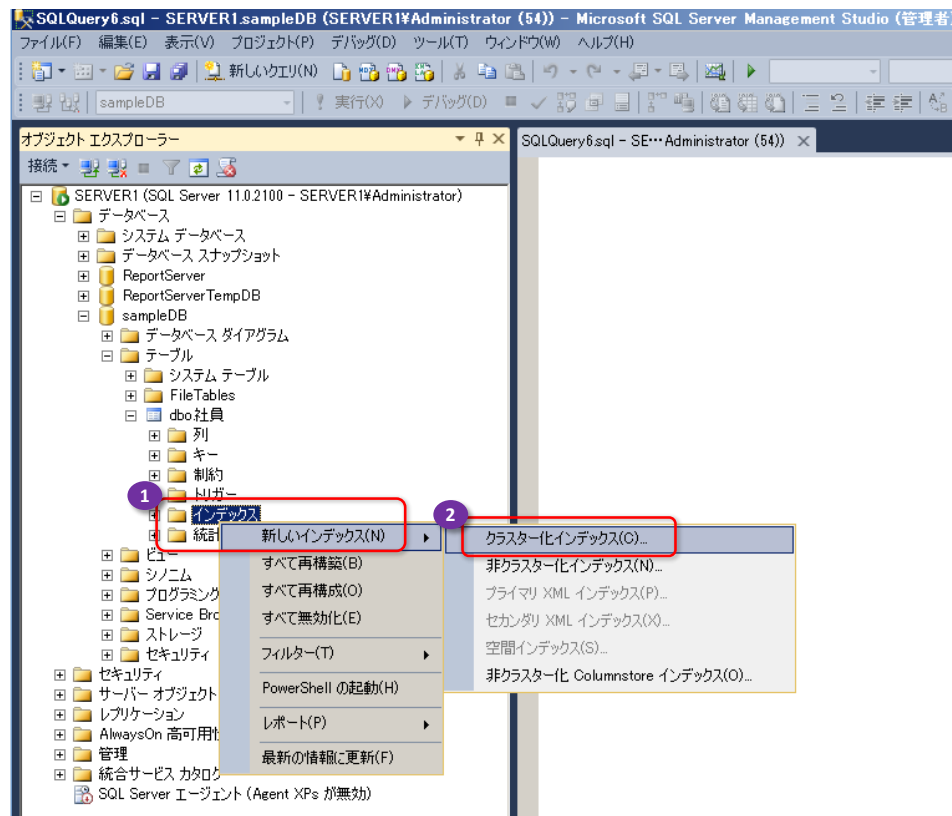
インデックス作成時には、実際のデータ（ヒープ）を並べ替えた結果を格納しておくための作業領域として、テーブル サイズの約 1.5 倍の空き領域が必要になることにも注意してください。空き領域が足りない場合には、インデックスの作成が失敗します。さらに、インデックス作成時には、トランザクション ログへの変更履歴も記録されるので、その分の空き領域も必要になります。

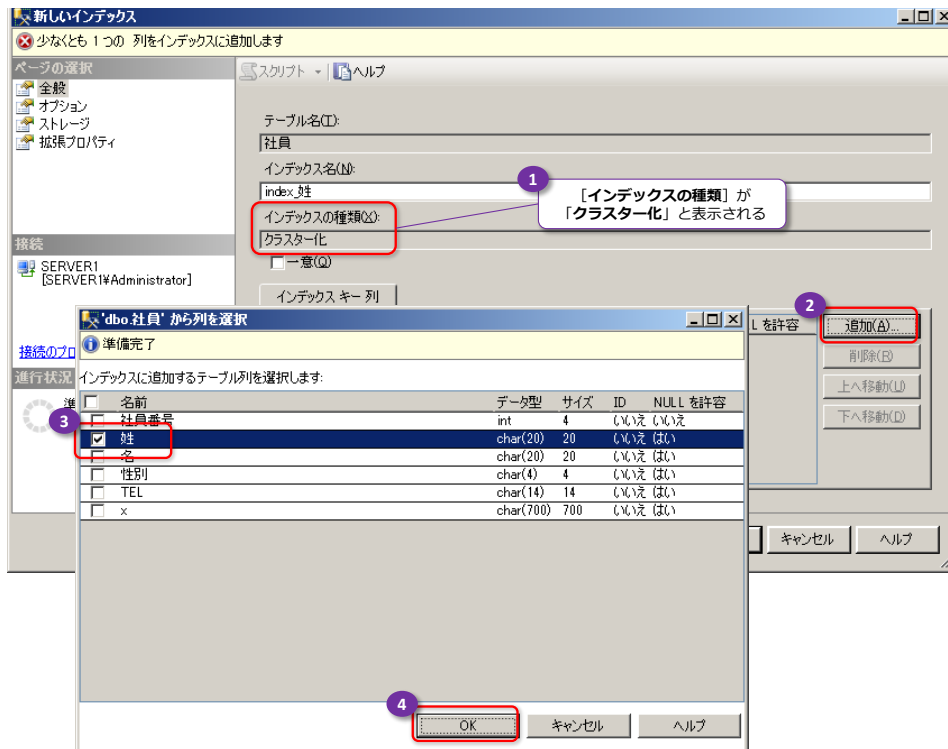
➡ クラスター化インデックスの作成

クラスター化インデックスを作成するには、**CREATE INDEX** ステートメントで、次のように **CLUSTERED** を指定します。

CREATE CLUSTERED INDEX インデックス名 **ON** テーブル名 (列名)

クラスター化インデックスを、オブジェクト エクスプローラーから作成する場合は、次のように操作します。





➡ Let's Try

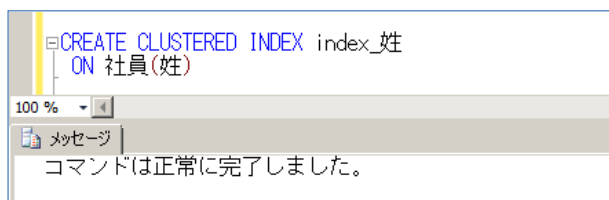
それでは、クラスター化インデックスを作成してみましょう。

1. まずは、「姓」列へ作成していた非クラスター化インデックス「index_姓」を削除します。

```
DROP INDEX 社員.index_姓
```

2. 次に、「姓」列に対してクラスター化インデックスを作成します。

```
CREATE CLUSTERED INDEX index_姓
ON 社員(姓)
```



「コマンドは正常に完了しました」と表示されれば、インデックスの作成が完了です。

3. 次に、**sys.indexes** カタログ ビューをクエリして、クラスター化インデックスのインデックス ID を確認してみましょう。

```
SELECT name, index_id, * FROM sys.indexes
WHERE object_id = OBJECT_ID('社員')
```

```
SELECT name, index_id, * FROM sys.indexes
WHERE object_id = OBJECT_ID('社員')
```

インデックスID

	name	index_id	object_id	name	index_id	type	type_desc
1	index_姓	1	245575913	index_姓	1	1	CLUSTERED

インデックス ID (index_id) は、「1」であることを確認できます。このように、クラスター化インデックスの ID は、常に 1 になります。

4. 続いて、**dm_db_index_physical_stats** 関数を利用して、クラスター化インデックスの階層数と使用しているページ数を確認してみましょう。

```
SELECT
    index_id, index_level, page_count
FROM
    sys.dm_db_index_physical_stats
    (DB_ID('sampleDB'), OBJECT_ID('社員'), 1, NULL, 'DETAILED')
ORDER BY index_id, index_level DESC
```

	index_id	index_level	page_count
1	1	2	1
2	1	1	3
3	1	0	560

実データのページ数

index_level が、「2」、「1」、「0」とあることから、このインデックスが 3 階層であることが分かります。index_level の「0」がリーフ ページ、「1」が中間、「2」がルート ページです。また、クラスター化インデックスの場合は、リーフ ページ「0」が実データのページ数なので、実際のデータが「560」ページであることも確認できます。

➡ クラスター化インデックスの効果の確認

5. 次に、「姓」列が「Aoki」さんの社員を検索して、クラスター化インデックスの効果を確認してみましょう。

```
SELECT * FROM 社員
WHERE 姓 = 'Aoki'
```

クエリ 1: クエリ コスト (バッチ相対): 100%
 SELECT * FROM [社員] WHERE [姓]='Aoki'

Clustered Index Seek (Clustered)
 [社員].[index_姓]
 コスト: 100 %

Clustered Index Seek

	社員番号	姓	名	性別	TEL	x
1	5760	Aoki	Tetsuya	男性	057-723-xxxx	NULL
2	5704	Aoki	Taiki	男性	056-695-xxxx	NULL
3	1030	Aoki	Mikoto	女性	03-3455-xxxx	NULL
4	670	Aoki	Senya	女性	03-1620-xxxx	NULL
5	836	Aoki	Misa	女性	03-2470-xxxx	NULL
6	386	Aoki	Chiaki	女性	03-0180-xxxx	NULL

実行プランを確認すると、「**Clustered Index Seek**」というアイコンが表示され、インデックス名「**社員.index_姓**」を確認することができます。これは、クラスター化インデックスを利用して検索を行ったという内部動作です。また、RID Lookup のアイコンがないところにも注目してください。クラスター化インデックスのリーフページは実際のデータが格納されているので、RID Lookup は必要なく、クラスター化インデックス内の Seek だけでデータを取得することができるのです。

このように、クラスター化インデックスは、RID Lookup をしなくて済む分、非クラスター化インデックスよりも検索のパフォーマンスが向上します。

6. 次に、検索条件を「**姓 LIKE 'A%'**」へ変更して、該当データ件数が多くなるように、データを検索してみましょう（これは、非クラスター化インデックスのときは、テーブル スキャンになった検索です）。

```
SELECT * FROM 社員
WHERE 姓 LIKE 'A%'
```

クエリ 1: クエリ コスト (バッチ相対): 100%
 SELECT * FROM 社員 WHERE 姓 LIKE 'A%'

Clustered Index Seek (Clustered)
 [社員].[index_姓]
 コスト: 100 %

Clustered Index Seek

	社員番号	姓	名	性別	TEL	x
1	3	Abe	Makiko	女性	045-321-xxxx	NULL
2	1282	Abe	Hiroyasu	男性	03-4730-xxxx	NULL
3	1608	Abe	Mayumi	女性	03-6375-xxxx	NULL
4	1758	Abe	Yuka	女性	03-7120-xxxx	NULL
5	2106	Abe	Jun	男性	03-8870-xxxx	NULL
6	2229	Abe	Isao	男性	03-9485-xxxx	NULL
7	2974	Abe	Mitsutoshi	男性	043-321-xxxx	NULL
8	2981	Abe	Shigenori	男性	043-325-xxxx	NULL
9	3103	Abe	Youichi	男性	043-386-xxxx	NULL
10	4084	Abe	Seiji	男性	048-880-xxxx	NULL
11	4358	Abe	Masato	男性	050-019-xxxx	NULL
12	4698	Abe	Katsuhiko	男性	051-190-xxxx	NULL
13	5193	Abe	Takayuki	男性	054-438-xxxx	NULL

該当データが 293 件

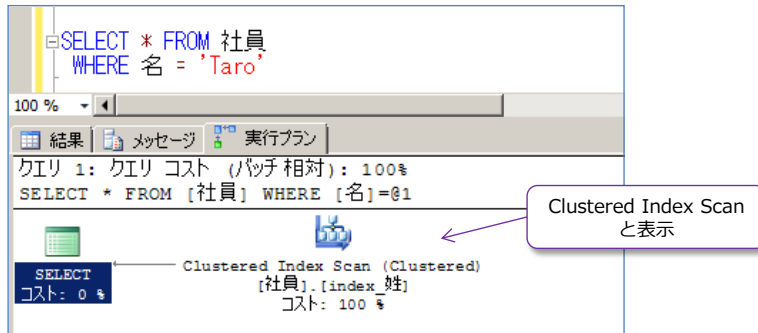
結果は、293 件の結果が返り、実行プランを確認すると、Clustered Index Seek が表示されて、クラスター化インデックスの Seek のみで検索が完了したことを確認できます。

クラスター化インデックスでは、リーフ レベルに実際のデータが昇順に並び替えられているので、このような検索（先頭文字を指定した範囲検索）であっても、インデックスを利用して高速にデータを取得することができます。

◆ Clustered Index Scan とテーブル スキャン

1. 次に、クラスター化インデックスを作成した「姓」列以外の列を利用して検索してみましょう。

```
SELECT * FROM 社員
WHERE 名 = 'Taro'
```



実行プランを確認すると、「**Clustered Index Scan**」というアイコンが表示されます。このアイコンは、Clustered Index Seek と非常に似ていますが、意味は大きく異なります。

「名」列には、インデックスを作成していないので、テーブル スキャンが実行されるはずですが、Clustered Index Scan と表示されています。このアイコンは、クラスター化インデックスのリーフレベルを横方向へ全件スキャンしたという内部動作です。つまり、実行されていることはテーブル スキャンとほとんど同じです。これは、クラスター化インデックスを作成すると、実際のデータがインデックスに吸収されるので、「**テーブル = クラスター化インデックス**」となり、テーブル スキャンという概念がなくなるためです。

3.7 自動的に作成されるインデックス（主キー制約と UNIQUE 制約）

➡ 自動的に作成されるインデックス

PRIMARY KEY（主キー）制約または **UNIQUE** 制約を設定している場合は、自動的にインデックスが作成されます。デフォルトでは、PRIMARY KEY 制約の場合には「**クラスター化インデックス**」、UNIQUE 制約の場合には「**非クラスター化インデックス**」が作成されて、インデックスの名前は制約の名前と同じになります。

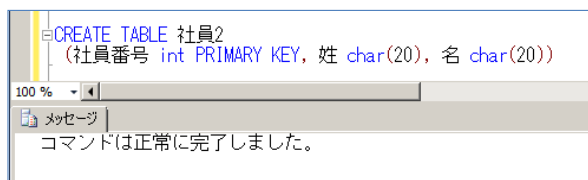
どちらの制約もデータを一意に保つためのものなので、この列を検索条件に指定した場合は、必ず 1 件の検索結果が返ります。インデックスは、大量のデータの中から数件のデータを取り出すときに最も効果を発揮するので、一意な値を持つ列に対して作成されたインデックスは、最も効果があります。また、主キーは検索条件としても頻繁に使用されるものなので、インデックスが自動的に作成されることはうれしいことです。

➡ Let's Try

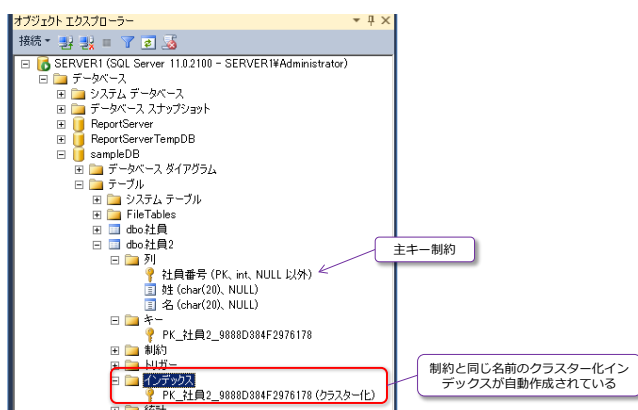
それでは、これを試してみましょう。

1. クエリ エディターへ次のように入力して、「社員番号」列を主キーとする「社員 2」という名前のテーブルを作成します。

```
CREATE TABLE 社員2
(社員番号 int PRIMARY KEY, 姓 char(20), 名 char(20))
```



2. 作成後、オブジェクト エクスプローラーで、「社員 2」テーブルの「インデックス」フォルダーを展開して、「PK_社員 2_~ (クラスター化)」というインデックスが作成されていることを確認してみましょう。

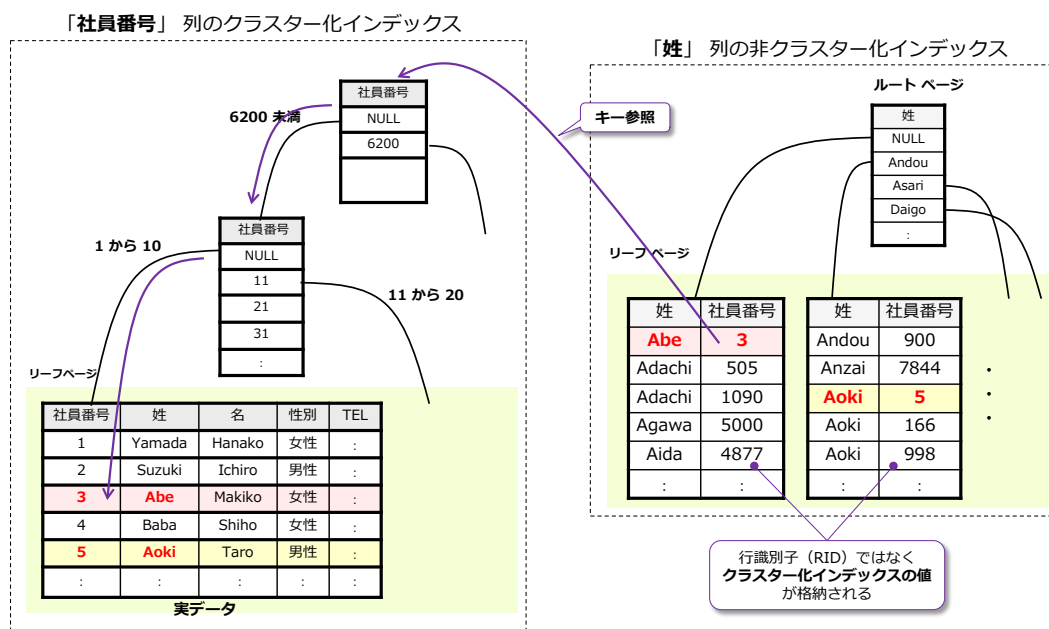


3.8 クラスター化がある場合の非クラスター化インデックスの内部構造

➡ クラスター化インデックスがある場合の非クラスター化インデックスの内部構造

ここでは、クラスター化インデックスと非クラスター化インデックスの両方が存在する場合について説明します。

クラスター化インデックスを作成すると、非クラスター化インデックスの構造が変更され、リーフレベルへ格納されるポインターが「行識別子」から「クラスター化インデックスの値」へ変更されます。たとえば、社員テーブルの「社員番号」列へクラスター化インデックスを作成し、「姓」列に、非クラスター化インデックスを作成している場合は、次の図のようなインデックス構造になります。



このような構造になるメリットは、「姓」と「社員番号」列のみを取得する検索を非常に高速に取得できるようになる点です。

➡ Let's Try

それでは、これを試してみましょう。

1. まずは、「姓」列へ作成していたクラスター化インデックスを削除します。

```
DROP INDEX 社員.index_姓
```

2. 次に、「社員番号」列に対してクラスター化インデックスを作成します。

```
CREATE CLUSTERED INDEX index_社員番号  
ON 社員 (社員番号)
```

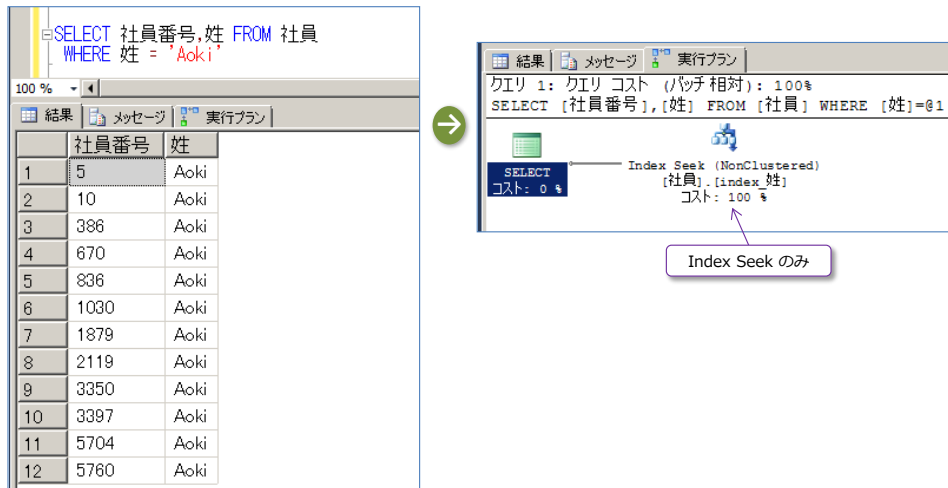
3. 続いて、「姓」列に対して非クラスター化インデックスを作成します。

```
CREATE INDEX index_姓
ON 社員 (姓)
```

➡ クラスター化インデックス + 非クラスター化インデックスの効果

4. 次に、「Aoki」さんの「社員番号」と「姓」列のみを取得するようにデータを検索してみましょう。

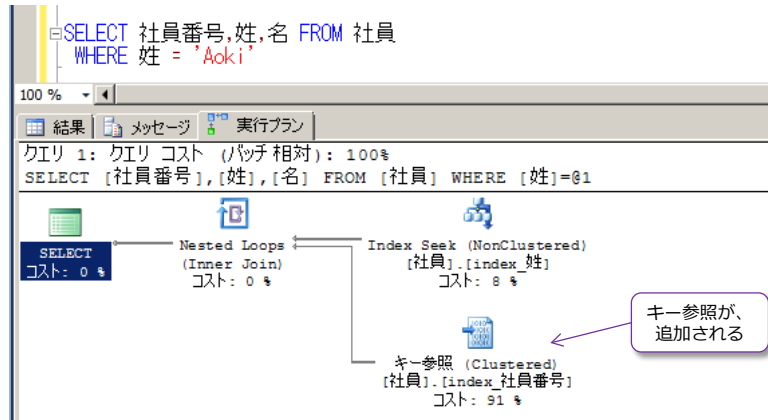
```
SELECT 社員番号, 姓 FROM 社員
WHERE 姓 = 'Aoki'
```



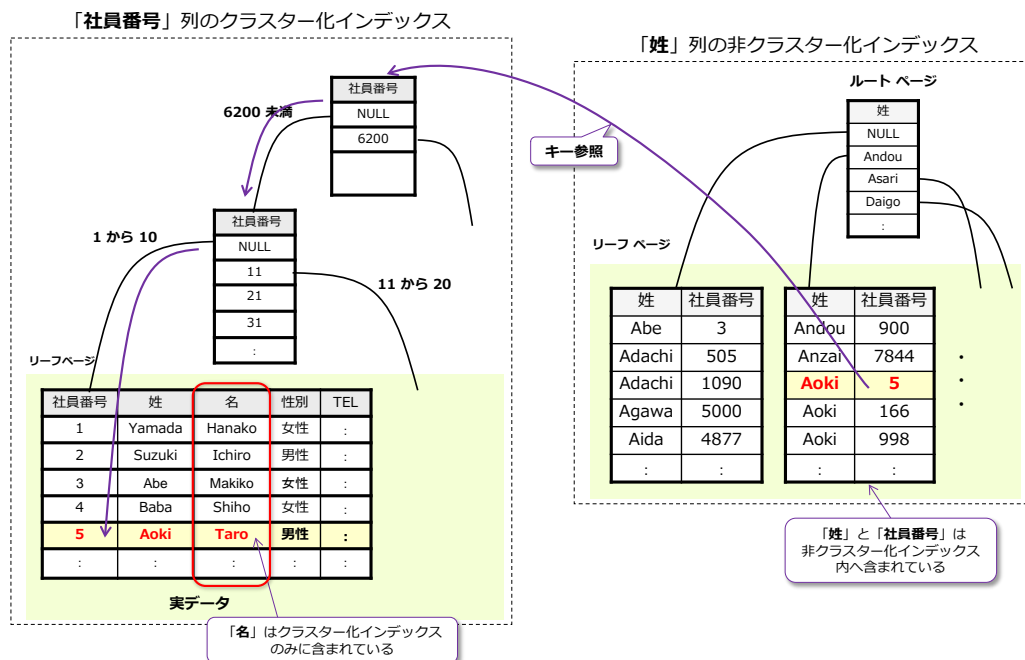
グラフィカル実行プランを表示すると、「姓」列へ作成した非クラスター化インデックス「社員.index_姓」の **Index Seek** のみが実行されたことを確認できます。RID Lookupアイコンが表示されないことがポイントです。「社員番号」と「姓」列のデータは、非クラスター化インデックスのリーフ レベルへ格納されているので、実際のデータを探すことなく（RID Lookup をすることなく）、インデックスの Seek のみで検索が完了しているのです（検索が高速に実行できています）。

5. 続いて、「社員番号」と「姓」列以外の列として「名」列も取得してみましょう。

```
SELECT 社員番号, 姓, 名 FROM 社員
WHERE 姓 = 'Aoki'
```



今度は、グラフィカル実行プランに「キー参照」アイコンが表示されます。これは、RID Lookupと同様に、実際のデータを参照したという内部動作です。RID Lookup からキー参照へ変わったのは、実際のデータへのポインターがクラスター化インデックスの値（キー）に変更されたためです。「名」列の値は、非クラスター化インデックスのリーフ レベルに格納されていないので、取得するには、実データへアクセスしなければなりません。



Note : リーフ ページの中身を DBCC PAGE コマンドで参照

前の Step の Note で紹介した、隠しコマンドの DBCC IND と DBCC PAGE を利用すれば、非クラスター化インデックスのリーフ ページへ、(行識別子 : RID ではなく) クラスター化インデックスのキー値が格納されていることを確認することができます。

DBCC IND(sampleDB, '社員', 3) ← 非クラスター化インデックスの ID を指定

IndexLevel = 0 がリーフページ

	PageFID	PagePID	IAMFID	IAMPID	ObjectID	IndexID	PartitionNumber	PartitionID	iam_chain_type	PageType	IndexLevel
1	1	171	NULL	NULL	245575913	3	1	72057594039631872	In-row data	10	NULL
2	1	169	1	171	245575913	3	1	72057594039631872	In-row data	2	0
3	1	172	1	171	245575913	3	1	72057594039631872	In-row data	2	0
4	1	174	1	171	245575913	3	1	72057594039631872	In-row data	2	0
5	1	175	1	171	245575913	3	1	72057594039631872	In-row data	2	0
6	1	177	1	171	245575913	3	1	72057594039631872	In-row data	2	0
7	1	178	1	171	245575913	3	1	72057594039631872	In-row data	2	0
8	1	179	1	171	245575913	3	1	72057594039631872	In-row data	2	0
9	1	180	1	171	245575913	3	1	72057594039631872	In-row data	2	0
10	1	856	1	171	245575913	3	1	72057594039631872	In-row data	2	1
11	1	880	1	171	245575913	3	1	72057594039631872	In-row data	2	0

DBCC PAGE(sampleDB, 1, 180, 3)

	Field	PageId	Row	Level	姓 (key)	社員番号 (key)	UNIQUEIFIER (key)	KeyHashValue	Row Size
1	1	180	0	0	Kinoshita	4890	0	(00f7da93b11f)	28
2	1	180	1	0	Kinoshita	5128	0	(9b8447348fd6)	28
3	1	180	2	0	Kinoshita	5562	0	(13d97626a635)	28
4	1	180	3	0	Kinseki	347	0	(d01892101e59)	28
5	1	180	4	0	Kinugasa	4423	0	(c2274e...5140)	28
6	1	180	5	0	Kirara	868	0	(86227436273f)	28
7	1	180	6	0	Kirimoto	2146	0	(a8dd755eb240)	28
8	1	180	7	0	Kirishima	481	0	(38467248d5df)	28
9	1	180	8	0	Kirishima	559	0	(1d3956841f6c)	28
10	1	180	9	0	Kirishima	712	0		28
11	1	180	10	0	Kirishima	781	0		28

RID (行識別子) がなくなり、クラスター化インデックスのキー値が格納されている

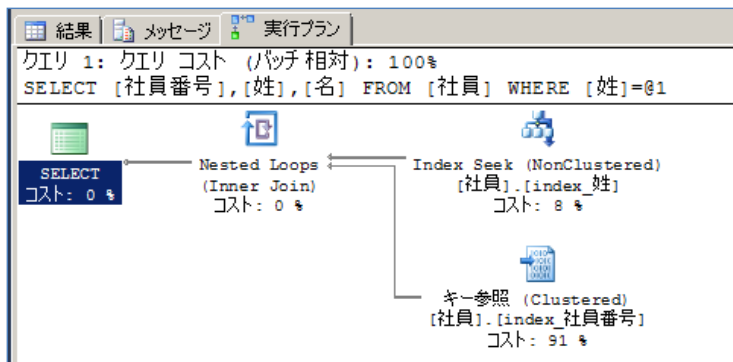
3.9 カバリング インデックス（複合インデックス）

➡ カバリング インデックス（複合インデックス）

カバリング インデックス（Covering Index）は、非クラスター化インデックスのリーフレベルへ検索で取得したいデータを格納して、キー参照（または RID Lookup）を行うことなく、Index Seek のみで完了する、つまり検索をインデックスのみでカバーするテクニックです。

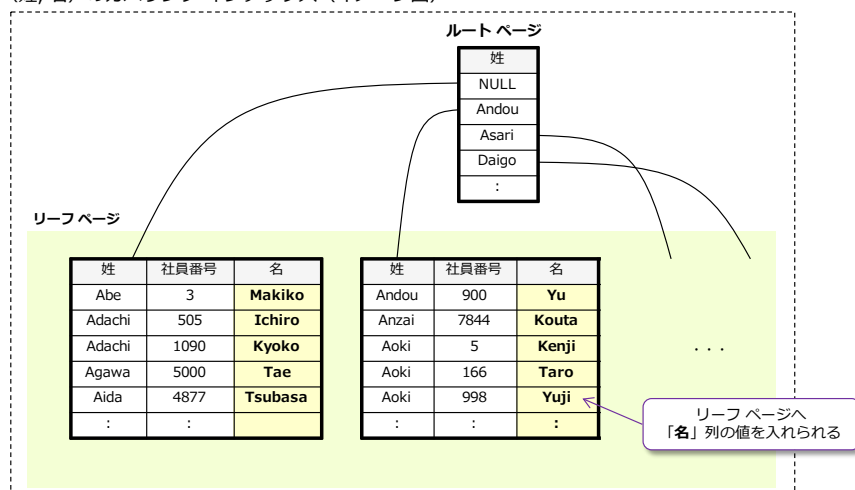
たとえば、前の Step で試した次の検索は、「名」列のデータを取得するために、「**キー参照**」が行われていました。

```
SELECT 社員番号, 姓, 名 FROM 社員 WHERE 姓 = 'Aoki'
```



このような場合に、「姓」と「名」列のカバリング インデックスを作成しておけば、リーフ レベルへ「名」列の値を格納できるようになるので、実際のデータを探す（キー参照をする）ことなく、インデックスのみで検索を完了させることができます。これにより、パフォーマンスを向上させることができます。

（姓, 名）のカバリング インデックス（イメージ図）



➡ カバリング インデックスの作成

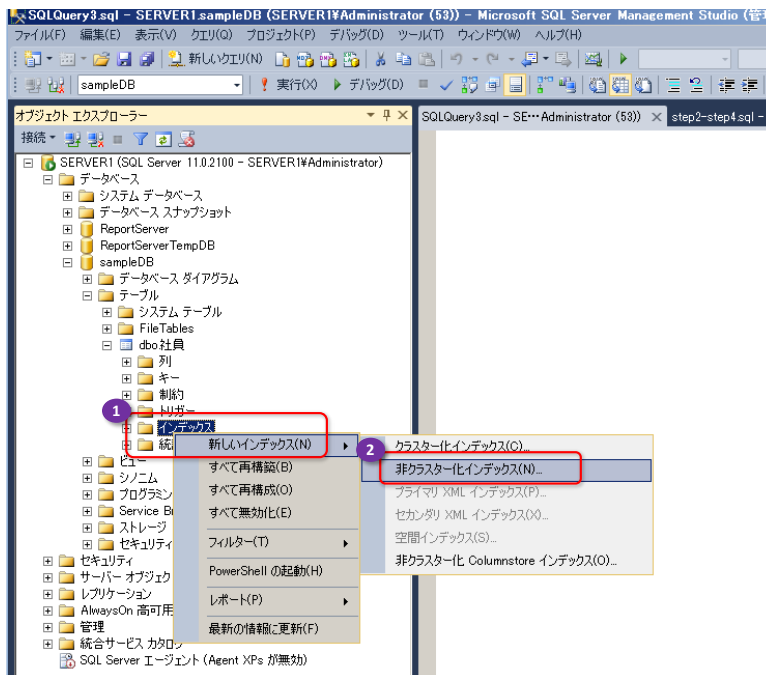
カバリング インデックスを作成するには、CREATE INDEX ステートメントを次のように利用します。

CREATE INDEX インデックス名
ON テーブル名 (列 1, 列 2, ...)

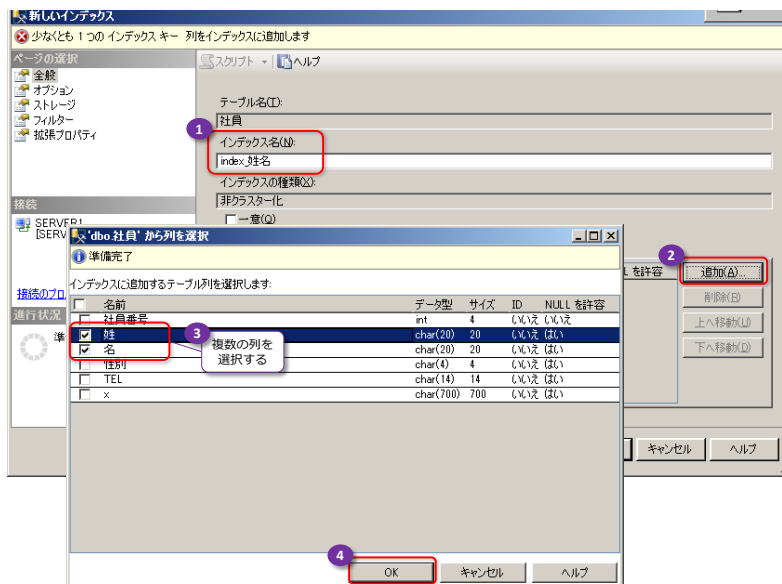
列名をカンマで区切って指定することで、それらの列をインデックスのリーフ ページへ含めることができるようになります。

➡ GUI での作成

カバリング インデックスを、オブジェクト エクスプローラーから作成する場合は、次のように操作します。



インデックスを作成する列の選択時に、複数の列を選択することで、カバリング インデックスを作成することができます。



◆ Let's Try

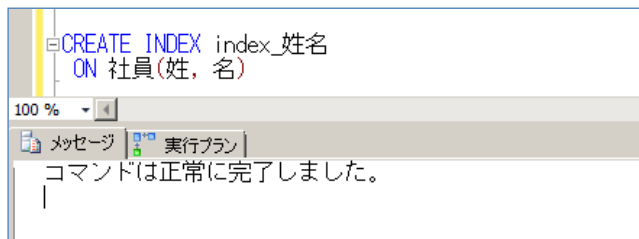
それでは、カバリング インデックスを作成してみましょう。

1. まずは、「姓」列へ作成していた非クラスター化インデックスを削除します。

```
DROP INDEX 社員.index_姓
```

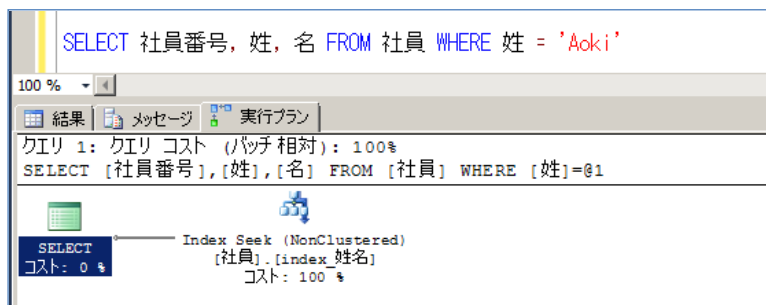
2. 次に、「姓」と「名」列を含めたカバリング インデックスを作成してみましょう。

```
CREATE INDEX index_姓名
ON 社員(姓, 名)
```



3. 作成が完了したら、姓が「Aoki」さんの「社員番号」と「姓」、「名」列を取得する検索を実行してみましょう。

```
SELECT 社員番号, 姓, 名 FROM 社員 WHERE 姓 = 'Aoki'
```



今度は、Index Seek のみで検索が完了し、「**キー参照**」は実行されていないことを確認できます。「社員番号」と「名」列はリーフ レベルへ格納されているので、実際のデータを探すことなく、インデックスを Seek するだけで検索が完了しているのです。

このようにカバリング インデックスを利用すると、「**キー参照**」をしなくて済む分、大きくパフォーマンスを向上させることができます。

◆ カバリング インデックスの注意

カバリング インデックスを作成するときは、列の順番に注意する必要があります。**(姓, 名)** と指定する場合と **(名, 姓)** と指定する場合では、構造が異なります。カバリング インデックスは、

一番左へ指定した列でツリー構造が作成されるので、WHERE 句の検索条件で指定される列を、一番左へ指定しておくようにします。

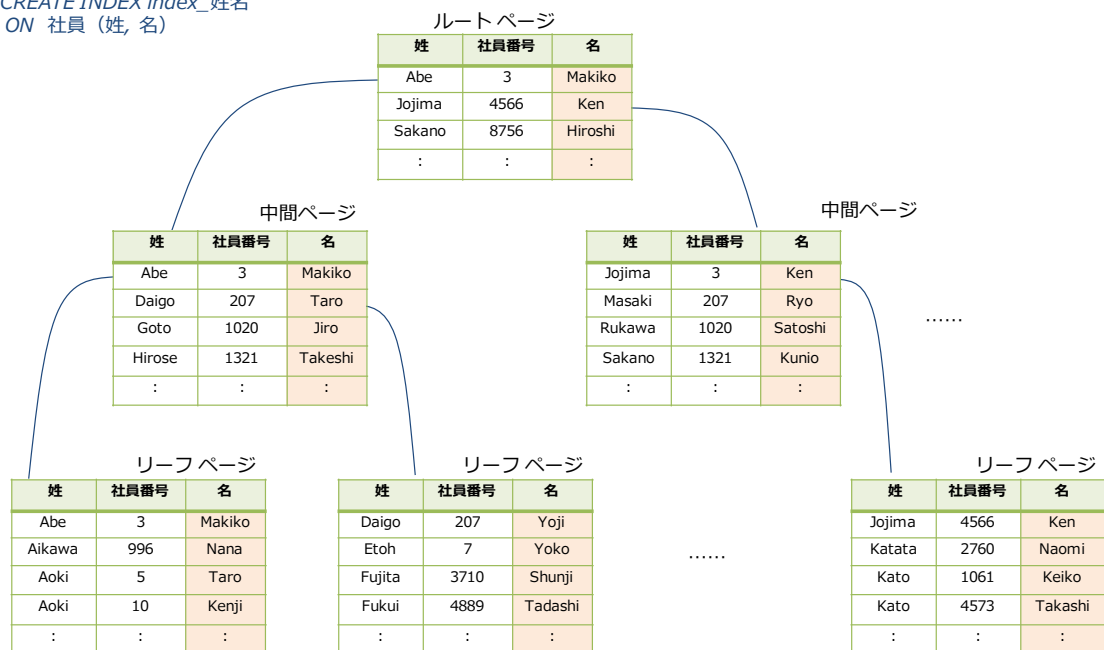
2 つ目以降の列データは、リーフ レベルへ格納されますが、その分余計にディスク領域を消費することにも注意する必要があります。特に追加した列のデータ サイズが大きい場合には要注意です。また、実際のデータの更新時のインデックス自身を更新するオーバーヘッドもデータ サイズが増える分だけ大きくなります。

➡ カバリング インデックスの正確な内部構造

カバリング インデックスで 2 つ目以降へ指定した列データは、正確には、次の図のように、中間ページとルート ページにも格納されます。また、中間ページとルート ページには、行識別子 (RID) またはクラスター化インデックスのキー値も格納されます。

カバリング インデックスの正確な内部構造

```
CREATE INDEX index_姓名
ON 社員 (姓, 名)
```

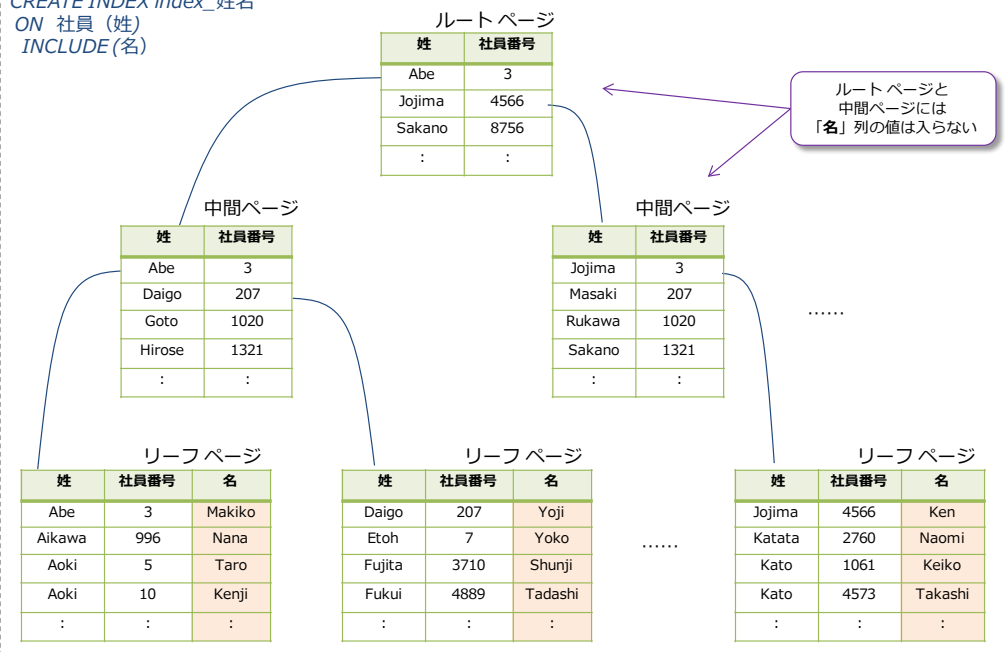


このように、カバリング インデックスでは、中間ページとルート ページにも 2 つ目以降へ指定した列データ格納するので、その列データのサイズが大きい場合には、非常にインデックス サイズが大きくなってしまい、パフォーマンスの低下に繋がるケースがあります。

これを解消してくれる機能が、次に説明する「付加列インデックス」(Include オプション) です。付加列インデックスの場合は、次のように中間とルートへ値を格納することなく、リーフのみへ値を格納できるようになります。

付加列インデックスの内部構造

```
CREATE INDEX index_姓名
ON 社員 (姓)
INCLUDE (名)
```

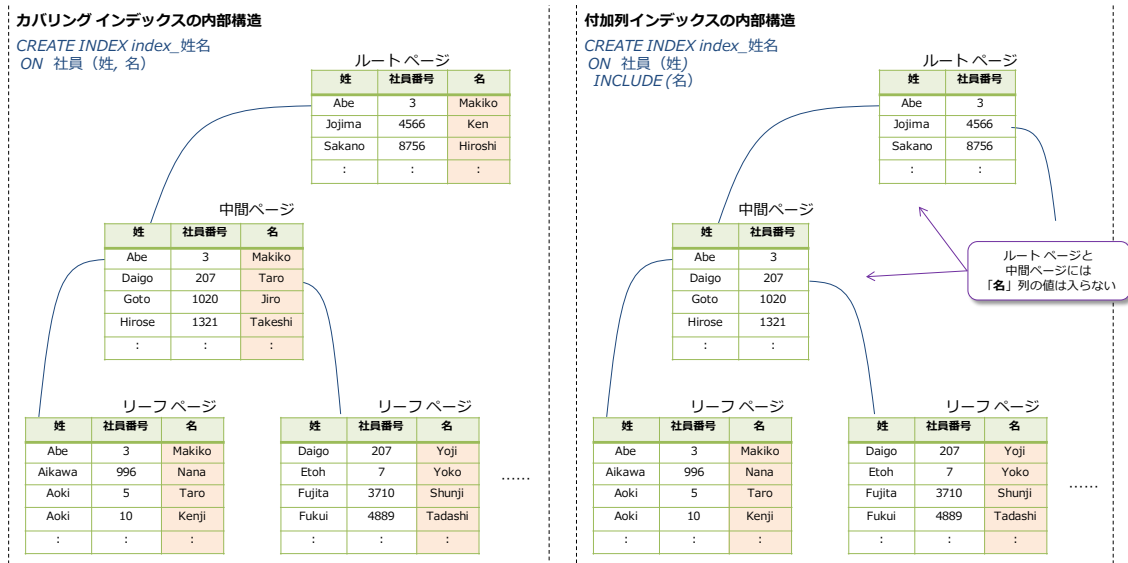


付加列インデックスは、カバリング インデックスの欠点を補うために搭載された機能です。

3.10 付加列インデックス (Include オプション)

➡ 付加列インデックス

付加列インデックスは、「**Include オプション**」とも呼ばれ、インデックスのリーフ レベルへ指定した列を含める (Include する) ことができる機能で、SQL Server 2005 から提供されました。カバリング インデックス (複合インデックス) との違いは、前のページにも掲載しましたが、次の図のとおりです。



カバリング インデックスは、インデックスのリーフ レベルだけでなく、ルートと中間ページにもカバリングへ含めた列 (「名」列) の値を格納しますが、付加列インデックスの場合はリーフ レベルにしか値を格納しません。これにより、インデックス サイズを小さくすることができます。インデックス サイズを小さくできれば、インデックスの Seek および Scan で読み取るページ数 (I/O 数) を少なくできるので、パフォーマンスが向上します。

付加列インデックスは、カバリング インデックス (複合インデックス) の欠点を補った機能なので、どちらを利用するかを迷ったら、迷わず付加列インデックスを利用することをお勧めします。

➡ 付加列インデックスの作成

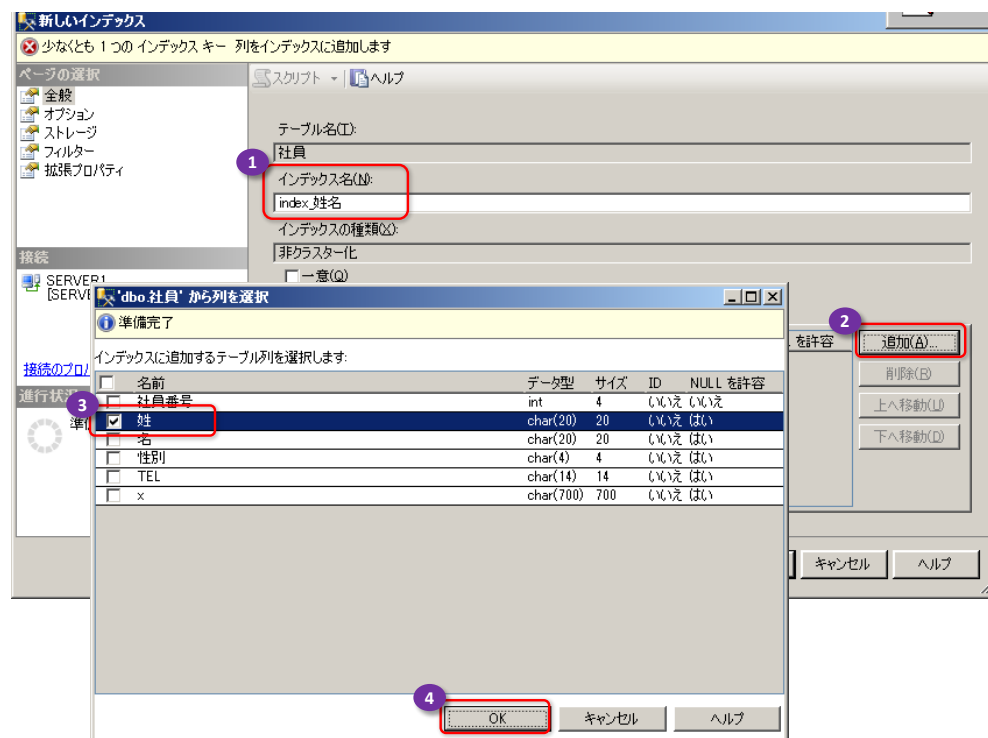
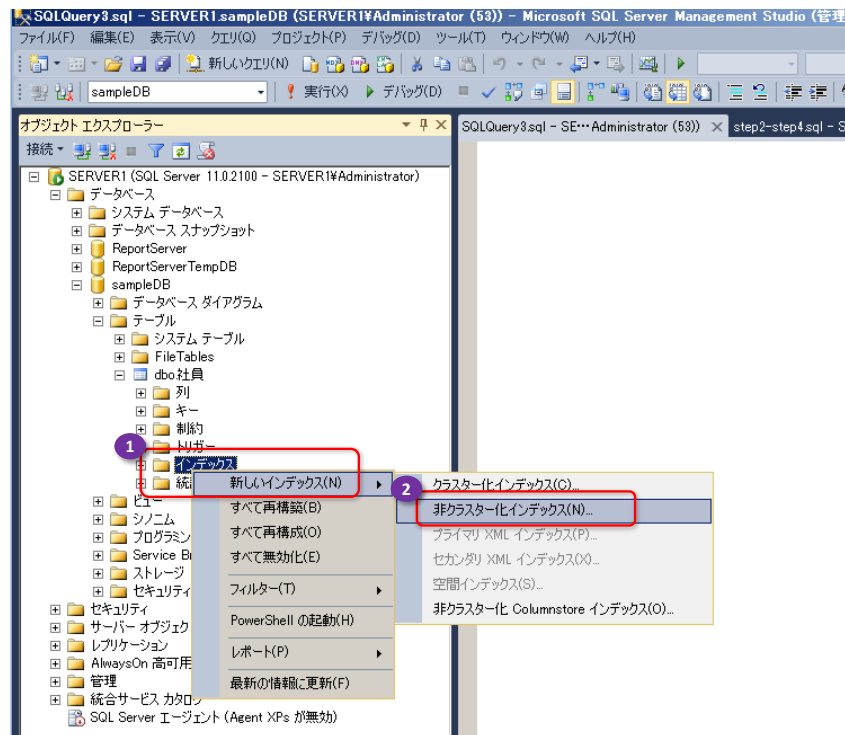
付加列インデックスを作成するには、CREATE INDEX ステートメントを次のように利用します。

```
CREATE INDEX index_姓名
ON 社員 (列名)
INCLUDE (リーフへ含めたい列名 1, 列名 2, ...)
```

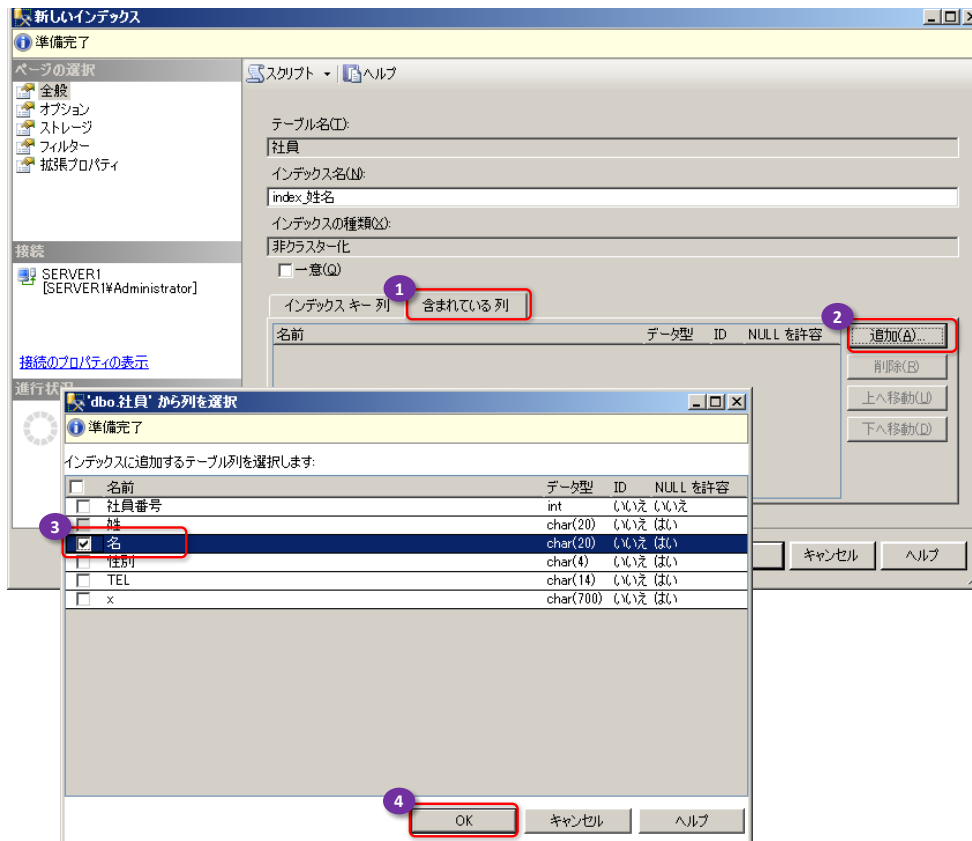
INCLUDE キーワードを付けて、含めたい列をカッコ内へ記述します。

◆ GUI でのインデックスの作成

付加列インデックスを、オブジェクト エクスプローラーから作成する場合は、次のように操作します。



[新しいインデックス] ダイアログで、インデックスを作成する列を追加したあとに、次のように [含まれている列] タブをクリックして開きます。



[追加] ボタンをクリックして、リーフページに含めたい列をチェックし、[OK] ボタンをクリックします。

➡ Let's Try

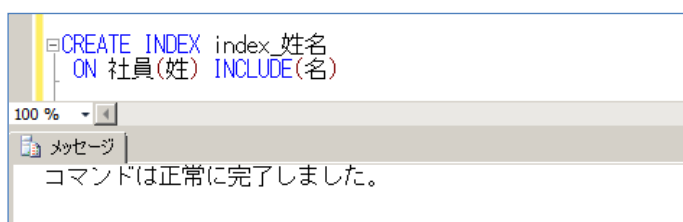
それでは、付加列インデックスを作成してみましょう。

1. まずは、前の Step で作成したカバリング インデックス「index_姓名」を削除します。

```
DROP INDEX 社員.index_姓名
```

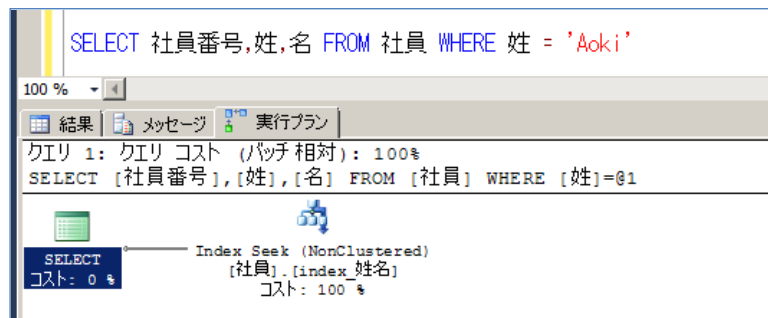
2. 次に、「姓」列に対して、「名」列を含む付加列インデックスを作成します。

```
CREATE INDEX index_姓名
ON 社員(姓) INCLUDE(名)
```



3. 作成が完了したら、カバリング インデックスのときと同じ検索を実行してみましょう。

```
SELECT 社員番号, 姓, 名 FROM 社員 WHERE 姓 = 'Aoki'
```



結果は、カバリング インデックスのときと同様、Index Seek のみが実行されたことを確認できます。

Note : カバリングと付加列インデックスの中身の参照

隠しコマンドの DBCC IND と DBCC PAGE を利用して、実際のカバリング インデックスの中身（中間またはルート ページ）を参照すると、次のようになります。

カバリング インデックスの中間ページの中身

```
-- カバリング インデックスの作成
CREATE INDEX index_姓名
ON 社員2(姓, 名)

DBCC IND(sampleDB, '社員2', 2)
/*
Index Level が 1 以上だと中間またはルート ページ
0 はリーフページ
*/

DBCC PAGE(sampleDB, 1, 1152, 3)
```

カバリングだと
中間ページとルート
ページに「名」列
が含まれる

	FileId	PageId	Row	Level	ChildFileId	ChildPageId	姓 (key)	名 (key)	社員番号 (key)	KeyHashValue
1	1	1152	0	1	1	1128	NULL	NULL	NULL	NULL
2	1	1152	1	1	1	1129	Aoyagi	Mitsuo	5362	NULL
3	1	1152	2	1	1	1130	Chino	Takashiosamu	5781	NULL
4	1	1152	3	1	1	1131	Fujishima	Kennichirou	5757	NULL
5	1	1152	4	1	1	1132	Hagino	Nobuyuki	2472	NULL
6	1	1152	5	1	1	1133	Hayanioi	Mia	866	NULL
7	1	1152	6	1	1	1134	Honda	Mika	1002	NULL
8	1	1152	7	1	1	1135	Ikeda	Kyou	3907	NULL
9	1	1152	8	1	1	1136	Ishikawa	Miyuki	780	NULL

カバリング インデックスは、中間またはリーフ ページにも、「名」列が含まれていることを確認できます。
これに対して、付加列インデックスの中身を参照すると、次のようになります。

```
DBCC PAGE(sampleDB, 1, 1056, 3)
```

中間ページとルート
ページには「名」列
は含まれていない

	FileId	PageId	Row	Level	ChildFileId	ChildPageId	姓 (key)	社員番号 (key)	UNIQUEIFIER (key)
1	1	1056	0	1	1	1024	NULL	NULL	NULL
2	1	1056	1	1	1	1025	Aoyagi	1504	0
3	1	1056	2	1	1	1026	Chino	5781	0
4	1	1056	3	1	1	1027	Fujishima	5757	0
5	1	1056	4	1	1	1028	Hagawa	3256	0
6	1	1056	5	1	1	1029	Hayama	826	0
7	1	1056	6	1	1	1030	Honda	1272	0
8	1	1056	7	1	1	1031	Ikeda	631	0
9	1	1056	8	1	1	1032	Ishikawa	2162	0

付加列インデックスの場合は、「名」列が含まれていないことを確認できます。

これにより、付加列インデックスの方がカバリング インデックスよりも、インデックスのサイズを小さくすることが出来ます。この差はデータのサイズが大きければ大きいほど差が顕著になります。

前述したように、インデックス サイズが小さくなれば、インデックスの Seek および Scan で読み取るページ数 (I/O 数) を少なくできるので、付加列インデックスの方がパフォーマンスを向上させることができます。

STEP 4. インデックスの保守

インデックスを作成したテーブルに対してデータの追加や更新を行っていくと、いずれデータの断片化が発生します。この STEP では、パフォーマンスの低下を引き起こす断片化と、断片化の解消方法、事前防止策などを説明します。

この STEP では、次のことを学習します。

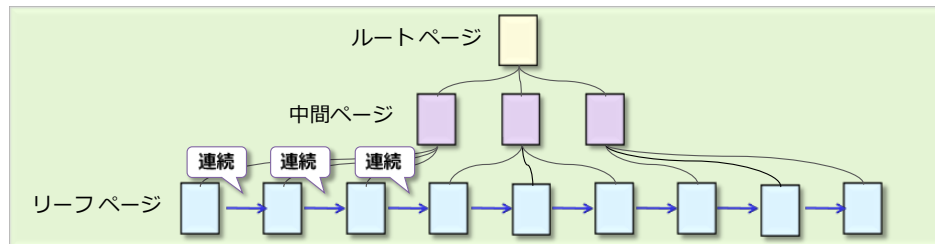
- ✓ 断片化とは
- ✓ 断片化の調査： `dm_db_index_physical_stats`
- ✓ 断片化の解消
- ✓ 再構築と再編成の違い
- ✓ 断片化の事前防止策： `FILLFACTOR`

4.1 断片化とは

➡ 断片化とは

インデックスのリーフ ページは、Index Scan を高速化するために、次のようにそれぞれのページ同士がリンクしています。

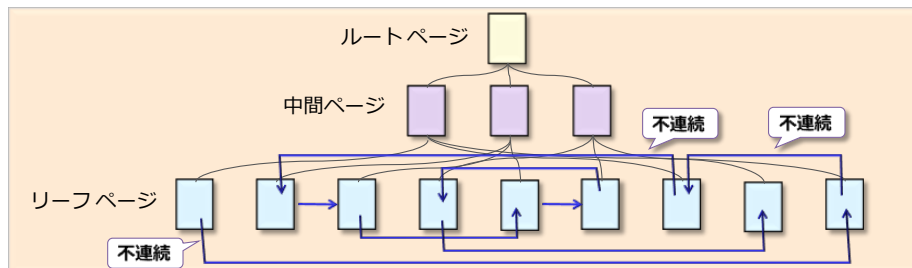
断片化発生前のインデックス = Index Scan が高速



リーフ ページは、物理的に連続して格納されている場合は、先読み（先行読み取り）機能が効果的に働き、パフォーマンス良くデータを取得することができます。

断片化（fragmentation: フラグメンテーション）は、次のようにリーフ ページが連続的ではなく断片的に格納された状態です。

断片化した状態のインデックス = Index Scan に影響が出る

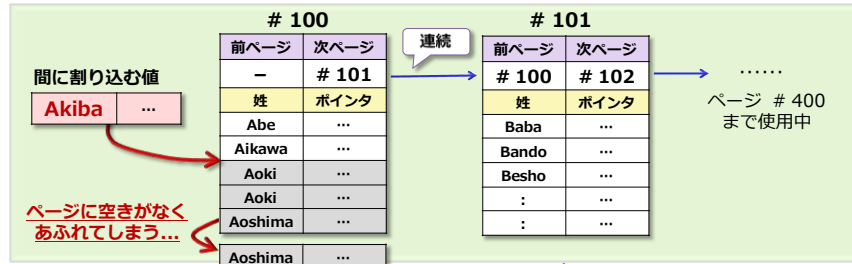


このように、リンクしているページが物理的に離れた場所にある状態が断片化が発生した状態です。このままでは、先読み機能が効果的に働かなくなり、Index Scan のパフォーマンスが大きく低下します。したがって、インデックスを作成した後は、断片化を事前防止する、あるいは断片化を解消するための保守を行うことが非常に重要になります。

➡ 断片化の原因（ページ分割）

デフォルトでは、ページ内は、追加または更新されたデータによって満杯に埋まっています。この状態で、間に割り込む値が追加されると、「**ページ分割**」が発生します。たとえば、次の図は、「姓」列へ作成したインデックスのリーフ ページが満杯に埋まっている状態を表していますが、このとき「Akiba」さんという間に割り込む値が INSERT されたとすると、ページには入りきらないので、物理的に離れた場所へ新しいページが作成されます。

断片化発生前=連続



断片化の発生



この動作（ページ分割）は、インデックスが常に昇順に並べ替えられた状態に保たれている必要があるために行われています。間に割り込む値が追加されると、その値によって溢れ出たデータを格納するための新しいページが必要になるのです。また、このページには、溢れ出た1行のみが格納されるのではなく、元のページと新しいページが半分半分になるように調整されます。これは、もう一度間に割り込む値が追加されたとしても、簡単にページ分割を発生させないようにするための処置です（ページ分割を連続して発生させないための処置です）。このようなページ分割は、元のページと新しいページに半分半分にデータを割り当てることから、「**50-50 ページ分割**」とも呼ばれています。

50-50 ページ分割によって作成された新しいページは、物理的には、最後の空きページが利用される場合がほとんどです。これによって、リーフ ページが断片化した状態が生まれます。

4.2 断片化の調査： dm_db_index_physical_stats

➡ 断片化の調査： dm_db_index_physical_stats

断片化が発生しているかどうかを調べるには、Step3 でインデックスの階層数と使用ページ数を調べるために利用した **dm_db_index_physical_stats** 動的管理関数を利用します。構文は次のとおりです。

```
SELECT * FROM sys.dm_db_index_physical_stats
(データベースID, テーブルID, インデックスID, パーティション番号, 'スキャンモード')
```

この関数の出力結果のうち、主なものは次のとおりです。

主な結果列	説明	LIMITED モードの場合
index_id	インデックスID	○
index_depth	インデックスの階層数	○
index_level	インデックスのレベル	○
avg_fragmentation_in_percent	断片化の割合 (%)	○
fragment_count	断片化しているページ数	○
page_count	ページ数	○
avg_page_space_used_in_percent	ページ内をデータで占める割合の平均	NULL
record_count	レコード数	NULL
avg_record_size_in_bytes	平均レコードサイズ	NULL

avg_fragmentation_in_percent という列で、断片化の割合を確認することができます。

➡ スキャン モード

dm_db_index_physical_stats 関数の第 5 引数で指定するスキャン モードは、次の 3 種類があります。

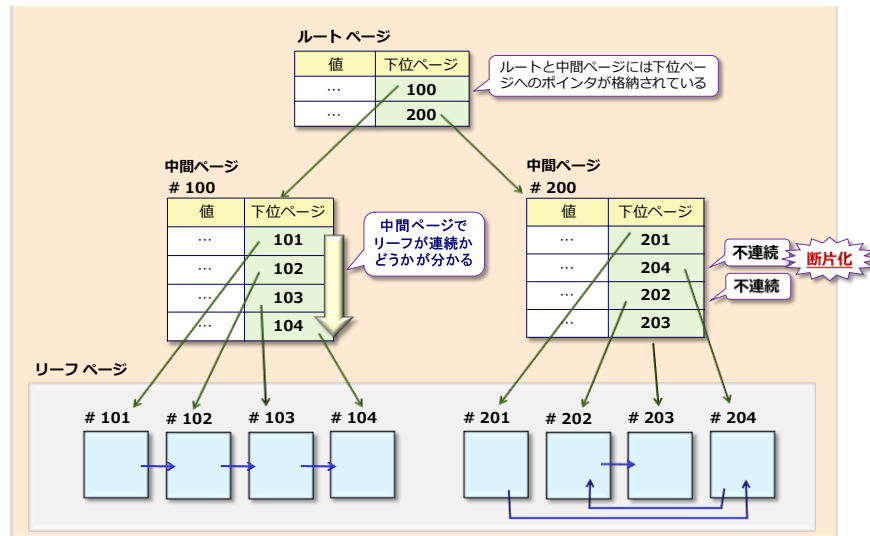
モード	スピード	説明
LIMITED	最速	断片化の割合のみを判別。SQL Server 2000 での DBCC SHOWCONTIG コマンドの WITH FAST に相当するモード
SAMPLED	高速	LIMITED モードのスキャンに加えて、一部のリーフ ページの読み取りを行う。10,000 ページ以上のインデックスでは、1% と 2% のサンプリング（サンプル抽出）した結果を比較し、その結果に差がある場合は、10% のサンプリングを行い、その結果を返す。差がない場合は 2% の結果を返す。
DETAILED	低速	100% のデータ（すべてのリーフ）を対象に、すべての情報を取得

LIMITED モードを使用すると、最も高速に断片化を調査することができます。

➡ LIMITED モードが高速な理由

dm_db_index_physical_stats 関数の LIMITED モードが高速な理由は、インデックスのリーフ レベルをスキャンせずに、ルート ページと中間ページのみを調査して、断片化の割合を算出するためです。次の図のように、中間ページにはリーフ ページのページ番号が格納されているので、

ページ番号が連続か不連続であるかをチェックするだけで、断片化の割合を調べることができます。



ただし、LIMITED モードの場合には、**avg_page_space_used_in_percent**（ページの平均使用密度）や、**record_count**（ページ内の行数）など、結果を取得できない列もあります。詳しくは、オンライン ブックの次の場所を参考にしてください。

SQL Server（データベース エンジン）

- > Transact-SQL リファレンス
- > システム ビュー
 - > 動的管理ビューおよび関数
 - > インデックス関連の動的管理ビューおよび関数
 - > sys.dm_db_index_physical_stats

➡ Let's Try

それでは、断片化について試してみましょう。

- まずは、前の Step で作成した付加列インデックス「**index_姓名**」の、インデックス ID を調べましょう。

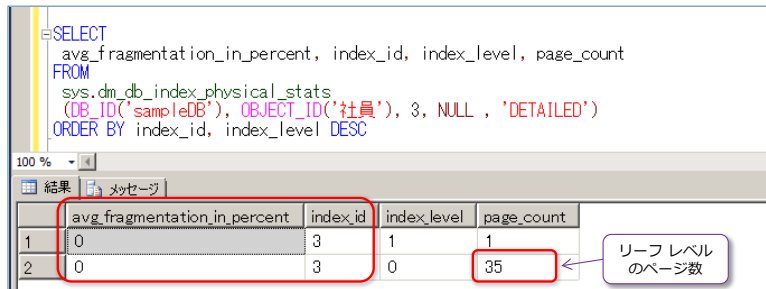
```
SELECT name, index_id, * FROM sys.indexes
WHERE object_id = OBJECT_ID('社員')
```

<pre>SELECT name, index_id, * FROM sys.indexes WHERE object_id = OBJECT_ID('社員')</pre>							
100 %							
結果 メッセージ							
	name	index_id	object_id	name	index_id	type	type_desc
1	index_社員番号	1	245575913	index_社員番号	1	1	CLUSTERED
2	index_姓名	3	245575913	index_姓名	3	2	NONCLUSTERED

インデックス ID (index_id) が「**3**」であることを確認できます。

2. 続いて、dm_db_index_physical_stats 関数を利用して、「index_姓名」の断片化の状態をチェックしてみましょう。

```
SELECT
    avg_fragmentation_in_percent, index_id, index_level, page_count
FROM
    sys.dm_db_index_physical_stats
        (DB_ID('sampleDB'), OBJECT_ID('社員'), 3, NULL, 'DETAILED')
ORDER BY index_id, index_level DESC
```



	avg_fragmentation_in_percent	index_id	index_level	page_count
1	0	3	1	1
2	0	3	0	35

「avg_fragmentation_in_percent」列は、すべての階層（index_level）で、「0」となっていることから、断片化が全く発生していないことを確認できます。また、このときの、リーフレベル（index_level が 0）のページ数（page_count 列）が、「35」ページであることも確認できます。

3. 次に、社員テーブルに対して、データを 1 件 INSERT してみましょう。

```
INSERT INTO 社員
VALUES(10001, 'Aiba', 'Jiro', '男性', '03-1234-xxxx', '')
```

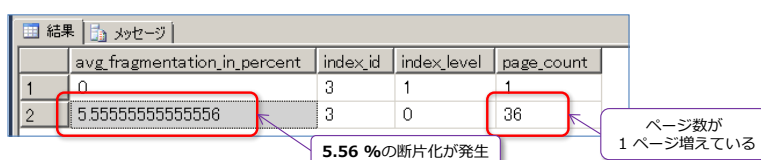


```
INSERT INTO 社員
VALUES(10001, 'Aiba', 'Jiro', '男性', '03-1234-xxxx', '')
```

(1 行処理されました)

4. データの追加後、もう一度同じクエリを実行して、index_姓名の断片化の状態をチェックしましょう。

```
SELECT
    avg_fragmentation_in_percent, index_id, index_level, page_count
FROM
    sys.dm_db_index_physical_stats
        (DB_ID('sampleDB'), OBJECT_ID('社員'), 3, NULL, 'DETAILED')
ORDER BY index_id, index_level DESC
```



	avg_fragmentation_in_percent	index_id	index_level	page_count
1	0	3	1	1
2	5.55555555555556	3	0	36

今度は、リーフ レベルで、「5.555…」パーセントの断片化が発生し、ページ数がデータを追加する前より 1 ページ増えて、「36」ページになっていることを確認できます。

5. 続いて、さらに、社員テーブルに対して、9 件のデータを追加してみましょう。

```
INSERT INTO 社員
VALUES(10002, 'Eto', 'Goro', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10003, 'Fujiwara', 'Goro', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10004, 'Goto', 'Goro', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10005, 'Inoue', 'Goro', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10006, 'Kato', 'Goro', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10007, 'Matsumoto', 'Goro', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10008, 'Oshima', 'Goro', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10009, 'Saito', 'Goro', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10010, 'Yamamoto', 'Goro', '男性', '03-1234-xxxx', '')
```

6. データの追加後、もう一度断片化の状態をチェックします。

```
SELECT
  avg_fragmentation_in_percent, index_id, index_level, page_count
FROM
  sys.dm_db_index_physical_stats
  (DB_ID('sampleDB'), OBJECT_ID('社員'), 3, NULL, 'DETAILED')
ORDER BY index_id, index_level DESC
```

	avg_fragmentation_in_percent	index_id	index_level	page_count
1	0	3	1	1
2	37.2093023255814	3	0	43

34.9 %の断片化が発生

ページ数が 43ページに増えている

今度は、リーフ レベルで、「34.88…」パーセントの断片化が発生していることを確認できます。また、ページ数も「43」ページへ増えています。

4.3 断片化の解消（インデックスの再構築と再構成）

➡ 断片化の解消

断片化を解消するには、次の 3 つの方法があります。

- インデックスのオフライン再構築
- インデックスのオンライン再構築（Enterprise エディション のみで利用可能）
- インデックスの再編成

これらは、いずれも **ALTER INDEX** ステートメントを利用して実行することができます。

➡ インデックスのオフライン再構築（REBUILD）

インデックスの再構築（オフライン）は、内部的には新しい領域へインデックスを再作成し、古いインデックスを削除することで、断片化を解消します。したがって、インデックスの再構築中は、インデックス全体がロックされるので、再構築中は、別のトランザクションからはインデックスに対して一切アクセスすることができません。再構築が完了するまでは、ユーザー操作は待ち状態になります。これが「**オフライン**」と呼ばれる理由です。

クラスター化インデックスの場合は、実際のデータそのものをインデックス内へ格納しているので、再構築中にはテーブル データすべてがアクセスできないことに注意する必要があります。

Note：再構築にかかる時間

再構築にかかる時間は、インデックスが使用するページ数（ディスク容量）が大きければそれだけ時間がかかります。使用するページ数が大きいのは、行サイズが大きいインデックスで、クラスター化インデックス、カバリングインデックスなどです。特にクラスター化インデックスは、実際のデータそのものを格納しているため、再構築には非常に時間がかかります。

インデックスをオフライン再構築（REBUILD）するには、**ALTER INDEX** ステートメントを次のように利用します。

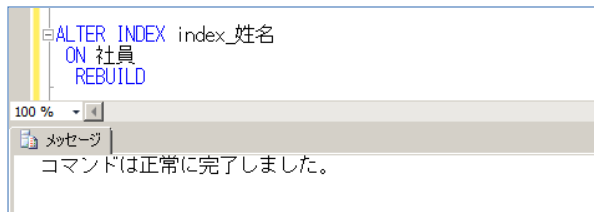
```
ALTER INDEX インデックス名
ON テーブル
REBUILD
```

➡ Let's Try

それでは、これを試してみましょう。

1. 社員テーブルへ作成した付加列インデックス「**index_姓名**」を再構築してみましょう。

```
ALTER INDEX index_姓名
ON 社員
REBUILD
```

「コマンドは正常に完了しました」と表示されれば、再構築が完了しています。

- 次に、**dm_db_index_physical_stats** 関数をクエリして、断片化の状態をチェックしてみましょう。

```
SELECT
    avg_fragmentation_in_percent, index_id, index_level, page_count
FROM
    sys.dm_db_index_physical_stats
    (DB_ID('sampleDB'), OBJECT_ID('社員'), 3, NULL, 'DETAILED')
ORDER BY index_id, index_level DESC
```

	avg_fragmentation_in_percent	index_id	index_level	page_count
1	0	3	1	1
2	0	3	0	35

リーフ レベルの **avg_fragmentation_in_percent** が「0」% になり、断片化が完全に解消されたことを確認できます。また、**page_count** も「35」ページ（断片化する前と同じ）へ戻っていることを確認できます。

◆ インデックスのオンライン再構築

インデックスの再構築は、オンラインで行うことも可能です。これは、SQL Server 2005 から提供された機能で、Enterprise エディションでのみ利用することができます。インデックスの再構築をオンラインで実行した場合は、再構築中にユーザーがデータへアクセスすることができます。

インデックスの再構築をオンラインで行うには、ALTER INDEX ステートメントを次のように実行します。

```
ALTER INDEX インデックス名
ON テーブル
REBUILD
WITH ONLINE = ON
```

◆ インデックスの再編成（REORGANIZE）

インデックスの再編成は、SQL Server 2000 までは、**DBCC INDEXDEFRAG** コマンドとして提供されていた機能です。インデックスの再編成は、リーフ ページの断片化のみを解消し、再編

成の実行中もユーザーがアクセスすることができます。ただし、ロック中のページはスキップされて、また、断片化の度合いがひどい場合には、インデックスの再構築（REBUILD）よりも実行時間のかかってしまうので注意してください。

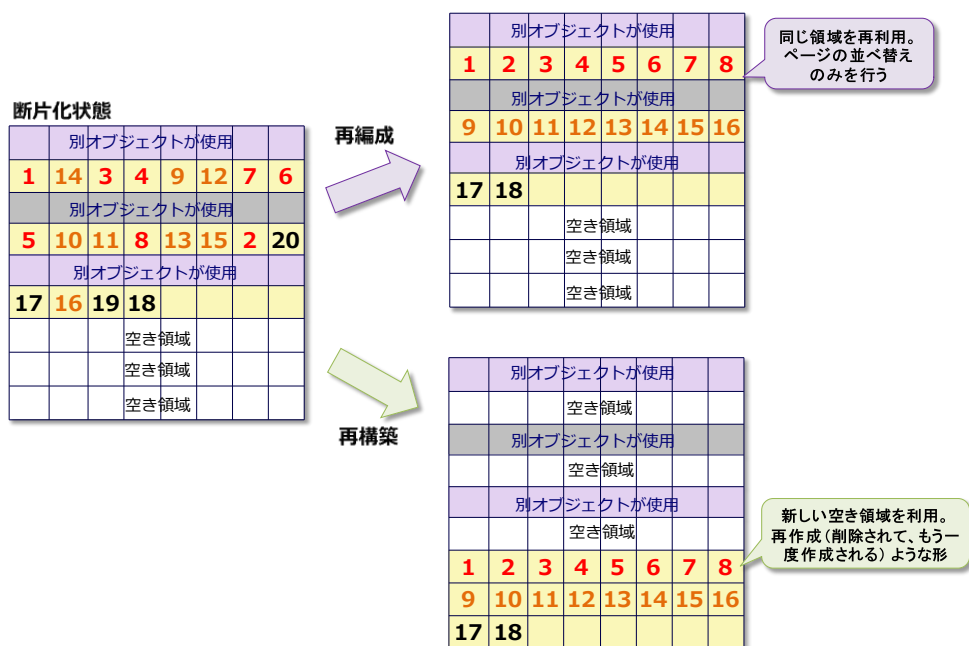
インデックスの再編成を実行するには、**ALTER INDEX** ステートメントを次のように実行します。

```
ALTER INDEX インデックス名
ON テーブル
REORGANIZE
```

➡ 再構築と再編成の違い

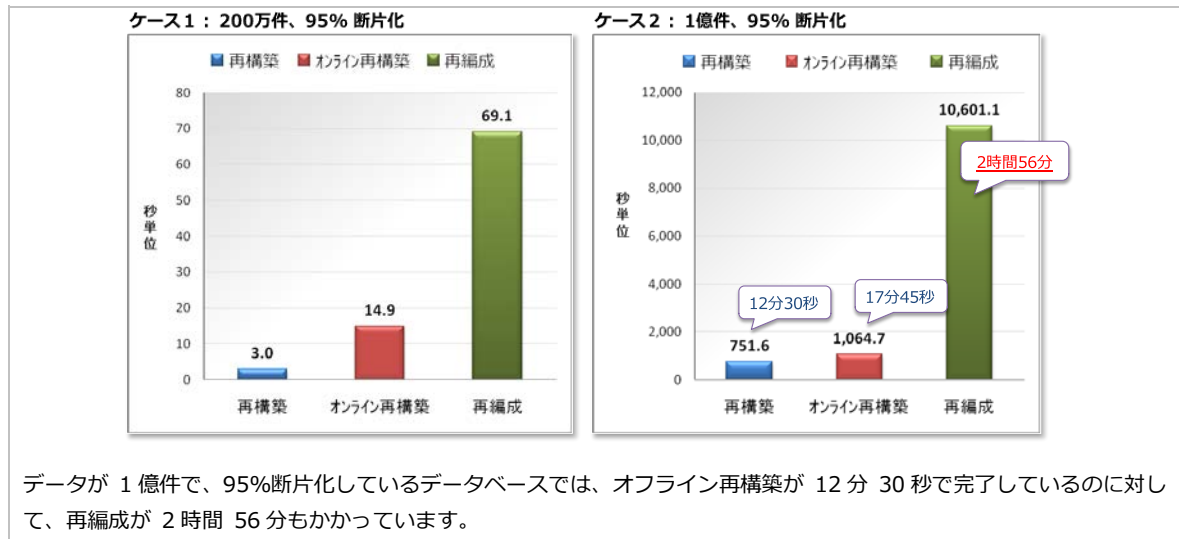
インデックスの再編成は、同じ領域を再利用して、それぞれのページを比較して並び替えを行うことで、断片化を解消しています。このような内部動作の違いがあるので、再編成は断片化の割合が大きい場合には、非常に時間がかかってしまいます。

これに対して、インデックスの再構築は、新しい領域にインデックスを再作成します。



Note : 断片化の割合が 30% 未満なら再編成、それ以上なら再構築

再構築を行うか、再編成を行うのかのおおまかな指針は、断片化の割合が 30% 未満なら再編成、それ以上なら再構築です。再編成（REORGANIZE）は、断片化の割合がひどい場合には、非常に時間がかかります。たとえば、次の 2 つのグラフは、弊社のお客様のデータで、再編成と再構築の実行時間を比較したものです。



Note：再構築と再編成の詳細比較

再構築と再編成の違いについては、次の表を参考にしてください。

	オフライン再構築	オンライン再構築	再編成 (ReOrganize)
SQL Server 2000 での機能	DBCC DBREINDEX	—	DBCC INDEXDEFRAG
Online / Offline	オフライン	オンライン	オンライン
オンライン中の動作	—	行バージョン管理機構を使用	ロック中のページをスキップ
追加のディスク領域	既存のインデックスサイズ + a	既存のインデックスサイズ + a + 一時マッピングインデックス (≒キーの長さ×行数)	不要
一括ログモデル	最小ログ記録		完全記録
再コンパイルと統計の自動更新	行う		行わない
FILLFACTOR	設定可能		設定不可 既存の設定に基づく
並列処理	MAXDOP で制御可 (並列処理は Enterprise Edition の機能)		不可 (常に 1)

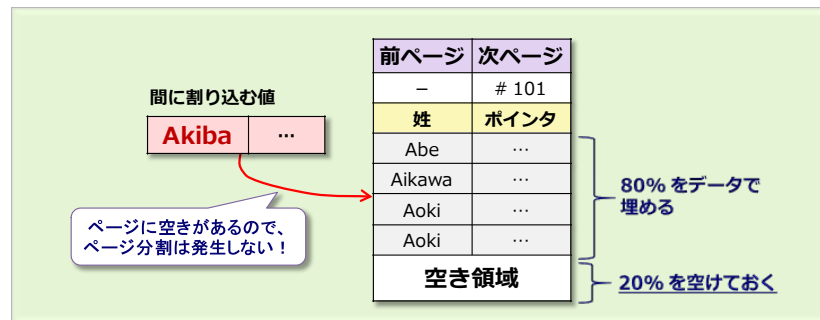
4.4 断片化の事前防止策： FILLFACTOR

➡ 断片化の事前防止策： FILLFACTOR

インデックスの再構築や再編成を実行して断片化を解消しても、データの追加や更新、削除が行われていくと、また断片化が発生します。断片化の度合いがひどくなっていくと、パフォーマンスへの悪影響も起こってきます。したがって、断片化がすぐに発生しないように事前防止策を講じておくことが重要です。これを行えるのが「**FILLFACTOR**」です。

FILLFACTORは、「**充填率**」とも呼ばれ、インデックスの再構築時にリーフ ページ内の領域をデータで占める割合を制御するためのオプションです。これは 0～100%の間に設定することができます。たとえば、FILLFACTOR を 80%に設定した場合は、次のように 80%をデータで埋め、20%を空き領域として残すことができます。

例： FILLFACTOR を 80% に設定した場合のリーフ ページ



このように、事前に空き領域を作成しておけば、データが追加されてもこの領域が利用されるので、（空き領域がなくなるまでは）50-50 ページ分割の発生（断片化）を防ぐことができます。

Note： FILLFACTOR のデフォルト値

FILLFACTOR は、デフォルトは 0% に設定されますが、これは 100% と同じ意味で、ページ内の領域をデータですべて埋めます。したがって、デフォルトでは、リーフページは満杯に埋められ、この状態で間に割り込むデータが追加されると 50-50 ページ分割（断片化）が発生します。

➡ FILLFACTOR を設定したインデックスの再構築

FILLFACTOR は、インデックスの再構築時に設定することができます。これは、ALTER INDEX ステートメントを次のように記述します。

```
ALTER INDEX インデックス名
ON テーブル
REBUILD
WITH ( FILLFACTOR = 値 )
```

◆ Let's Try

それでは、これを試してみましょう。

1. まずは、社員テーブルに対してデータを 10 件 INSERT して、断片化が発生することを確認してみましょう。(前の Step では、インデックスの再構築を実行したので、断片化が完全に解消されている状態です)。

```
INSERT INTO 社員
VALUES(10011, 'Aiba', 'Saburo', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10012, 'Matuda', 'Saburo', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10013, 'Kato', 'Goro', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10014, 'Matsumoto', 'Goro', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10015, 'Okada', 'Ryu', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10016, 'Yamamoto', 'Hiroko', '女性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10017, 'Oda', 'Hiroshi', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10018, 'Sakamoto', 'Kenji', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10019, 'Uchia', 'Yumi', '女性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10020, 'Nakamura', 'Hiromi', '女性', '03-1234-xxxx', '')
```

2. データの追加後、dm_db_index_physical_stats 関数を利用して、断片化の状態をチェックしてみましょう。

```
SELECT
    avg_fragmentation_in_percent, index_id, index_level, page_count
FROM
    sys.dm_db_index_physical_stats
    (DB_ID('sampleDB'), OBJECT_ID('社員'), 3, NULL, 'DETAILED')
ORDER BY index_id, index_level DESC
```

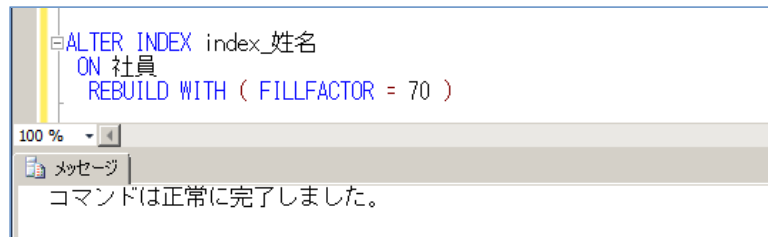
	avg_fragmentation_in_percent	index_id	index_level	page_count
1	0	3	1	1
2	37.2093023255814	3	0	43

リーフ ページが「37.2…」パーセントの断片化が発生していることを確認できます。このように、インデックスを再構築した後に、データが追加された場合は、また、断片化が発生してしまうのです。

3. 次に、**FILLFACTOR** を利用して、断片化を事前防止してみましょう。FILLFACTOR を設定

するには、次のように ALTER INDEX ステートメントを実行して、インデックスを再構築します。

```
ALTER INDEX index_姓名
ON 社員
REBUILD WITH ( FILLFACTOR = 70 )
```



FILLFACTOR には、**70** を指定することで、30% の空き領域を作っています。

4. 次に、dm_db_index_physical_stats 関数を利用して、断片化の状態をチェックします。

```
SELECT
    avg_fragmentation_in_percent, index_id, index_level, page_count
FROM
    sys.dm_db_index_physical_stats
    (DB_ID('sampleDB'), OBJECT_ID('社員'), 3, NULL, 'DETAILED')
ORDER BY index_id, index_level DESC
```

	avg_fragmentation_in_percent	index_id	index_level	page_count
1	0	3	1	1
2	0	3	0	50

断片化が「0」% になって、断片化が起きていない状態であることを確認できます。

また、リーフ レベルの page_count が、「50」ページへ増えていることにも注目します。これは、30% の空き領域を作ったことで、使用するページ数が増えているためです。

5. dm_db_index_physical_stats 関数では、avg_page_space_used_in_percent 列を取得すると、ページの平均使用密度を調べることができます。

```
SELECT
    avg_page_space_used_in_percent
    , avg_fragmentation_in_percent, index_id, index_level, page_count
FROM
    sys.dm_db_index_physical_stats
    (DB_ID('sampleDB'), OBJECT_ID('社員'), 3, NULL, 'DETAILED')
ORDER BY index_id, index_level DESC
```

```

SELECT
    avg_page_space_used_in_percent
    ,avg_fragmentation_in_percent, index_id, index_level, page_count
FROM
    sys.dm_db_index_physical_stats
    (DB_ID('sampleDB'), OBJECT_ID('社員'), 3, NULL, 'DETAILED')
ORDER BY index_id, index_level DESC

```

	avg_page_space_used_in_percent	avg_fragmentation_in_percent	index_id	index_level	page_count
1	22.2139856683963	0	3	1	1
2	69.4212997281937	0	3	0	50

FILLFACTOR を 70%へ設定しているなので、**avg_page_space_used_in_percent**（ページの平均使用密度）が「69.4…」(約 70%)であることを確認できます。

6. 次に、もう一度、社員テーブルに対して、データを 10 件 INSERT してみましょう。

```

INSERT INTO 社員
VALUES(10011, 'Aiba', 'Saburo', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10012, 'Matuda', 'Saburo', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10013, 'Kato', 'Goro', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10014, 'Matsumoto', 'Goro', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10015, 'Okada', 'Ryu', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10016, 'Yamamoto', 'Hiroko', '女性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10017, 'Oda', 'Hiroshi', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10018, 'Sakamoto', 'Kenji', '男性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10019, 'Uchia', 'Yumi', '女性', '03-1234-xxxx', '')
INSERT INTO 社員
VALUES(10020, 'Nakamura', 'Hiromi', '女性', '03-1234-xxxx', '')

```

7. データの追加後、dm_db_index_physical_stats 関数を利用して、断片化の状態をチェックしてみましょう。

```

SELECT
    avg_fragmentation_in_percent, index_id, index_level, page_count
FROM
    sys.dm_db_index_physical_stats
    (DB_ID('sampleDB'), OBJECT_ID('社員'), 3, NULL, 'DETAILED')
ORDER BY index_id, index_level DESC

```

	avg_fragmentation_in_percent	index_id	index_level	page_count
1	0	3	1	1
2	0	3	0	50

結果は、断片化が発生していないことを確認できます。また、page_count も増えていないことを確認できます。

このように、FILLFACTOR を設定しておくことで、今後追加されるデータを考慮して、あらかじめページに余裕をもたせて、インデックスを再構築することができるので、再構築後すぐに断片化が発生することを防ぐことができます。

➡ おわりに

最後までこの自習書の内容を試された皆さま、いかがでしたでしょうか？

インデックスは、SQL Server のパフォーマンスを大きく左右する非常に重要な機能です。1つのテーブルに対して複数のインデックスを作成することができますが、むやみに作成するのは良くありません。更新のオーバーヘッド（データ更新時に、実際のデータだけでなく、インデックス自体を更新する負荷）があるからです。

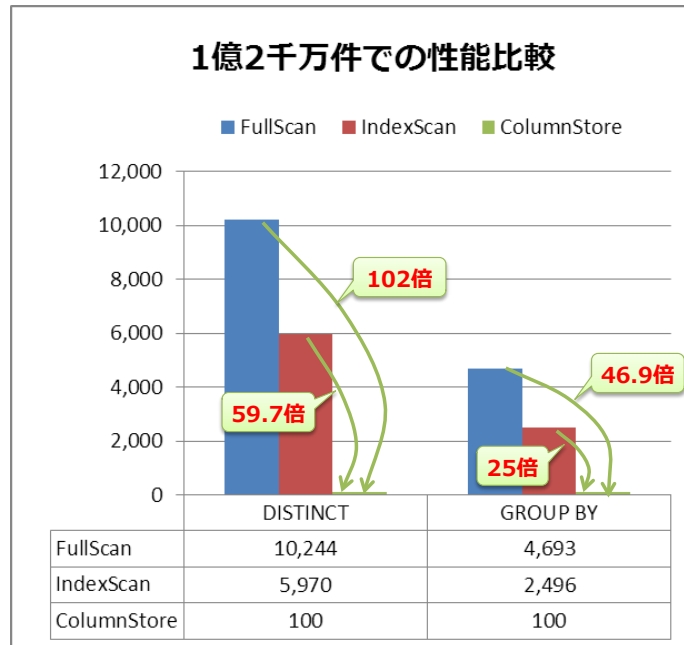
また、せっかくインデックスを作成しても、インデックスが活用されなかったり、検索のパフォーマンスが向上しなかったりすることもあります。インデックスは、特性をきちんと理解した上で、付加列インデックスなどを効果的に活用して、パフォーマンスの向上に役立てていただければと思います。

また、インデックスの作成後は、定期的な断片化のチェックも重要です。これを怠ると、「ある日突然遅くなった」という事態になりかねません。FILLFACTOR を設定して断片化を事前防止し、定期的にインデックスを再構築または再編成を実行して、健全な状態へ保つようにしましょう。

➡ 列ストア インデックスによる飛躍的な性能向上（SQL Server 2012 の新機能）

SQL Server 2012 からは、パフォーマンスを飛躍的に向上させることができる「**列ストア インデックス**」という新しいタイプのインデックス機能も提供されました。このインデックスは、SQL Server 2008 R2 の PowerPivot for Excel で実装されたインメモリのカラムベース エンジン（xVelocity エンジン）を RDB へ応用したものです。このエンジンでは、列単位でインデックスを格納し、それらは高度に圧縮されています。

列ストア インデックスは、大量のデータに対する集計処理時に大きな性能向上を期待できる機能で、特に、**夜間バッチ処理時**（夜間バッチでの日次集計や月次集計処理など）や、**DWH**（データ ウェアハウス）環境での集計処理時に大変役立つ機能です。弊社のお客様データ（**1 億 2 千万件の DWH**）を利用して、**列ストア インデックス**の性能効果を検証したところ、以下のような結果が得られました。



検証で利用したのは売上データが 1 億 2 千万件格納されている「**売上**」テーブルで、このテーブルには「**売上年月**」、「**売上金額**」、「**売上数量**」列などがあり、以下の **SELECT** ステートメントを実行したときの結果を比較しました。

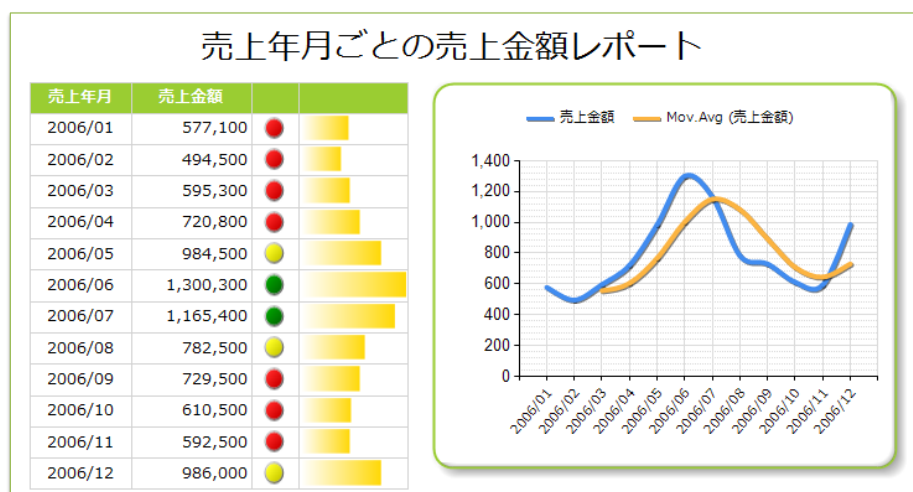
```
-- DISTINCT
SELECT DISTINCT 売上年月 FROM 売上

-- GROUP BY
SELECT 売上年月, SUM(売上金額), SUM(売上数量) FROM 売上 GROUP BY 売上年月
```

グラフの **FullScan** は、インデックスが使用されないフル テーブル スキャンが実行されたときの速度、**IndexScan** は、非クラスター化インデックス（カバリング インデックス）を（**売上年月**、**売上金額**、**売上数量**）で作成／利用したとき、**ColumnStore** が列ストア インデックスを作成／利用したときの実行速度です。ベンチマーク結果の公開は、使用許諾契約書で禁じられているので、グラフ内の結果は、それぞれ列ストア インデックスでの実行時間を **100** とした場合の相対値で表しています。

列ストア インデックスを利用することで、**DISTINCT** 処理では、フル スキャンに比べて **102 倍**、インデックス スキャンに比べて **59.7 倍**の性能向上、**GROUP BY** 処理では、フル スキャンに比べて **46.9 倍**、インデックス スキャンに比べて **25 倍**もの性能向上を確認することができました。なお、各テストで使用したデータは、すべてメモリ上のデータ バッファ キャッシュ内へ格納されている状態でテストしました。したがって、初回アクセス時の（メモリ内にデータが存在しない場合の）ディスクからの読み取りが伴う場合には、さらに性能差が顕著に現れることになります。

今回の結果の **25 倍**という差は非常に大きいもので、たとえば次のような「売上年月ごとの売上金額を表示するレポート」を思い浮かべてみてください。



このレポートを表示するのに今まで **10 秒**かかっていたとします。ここに**列ストア インデックス**を利用できるとすれば、**25 倍**もの性能向上が得られるので、**わずか 0.4 秒（10 秒の 25 分の 1 のスピード）**でレポートが表示されるわけです。コンピューターの世界では、**3 秒以上**の待ち時間があつたとすると、ユーザーが **"遅い"** と感じると言われています。しかし、従来のインデックスの作成方法では、データ件数の規模が**数億件**レベルになる場合には、数秒間もの待ち時間が発生してしまうのです。これを**列ストア インデックス**に変更することができれば、1 秒以内でのレスポンスを期待できるようになります。

また、**列ストア インデックス**は、このように **GROUP BY** 演算などによる**集計処理**で大きな効果を発揮するので、夜間バッチ処理時（夜間バッチでの日次集計や月次集計処理など）にも大きな性能向上を期待できます。

➡ 列ストア インデックスの作成方法

列ストア インデックスの作成方法は、非常に簡単で、次のように **COLUMNSTORE** キーワードを追加するだけです。

```
CREATE NONCLUSTERED COLUMNSTORE INDEX インデックス名
ON テーブル名 (列名 1, 列名 2, 列名 3, ~)
```

列ストア インデックスは、性能を向上させるために大変役立つ機能なので、ぜひ活用してみてください。本自習書シリーズの新機能編「**No.3 DWH (データ ウェアハウス) 関連の新機能**」では、列ストア インデックスの効果を簡単に確認できるようにしたスクリプト（ステップ バイ ステップ形式での検証スクリプト）を紹介していたり、列ストア インデックスの利用時の注意点などを説明したりしているので、こちらもぜひご覧いただければと思います。

執筆者プロフィール

有限会社エスキューエル・クオリティ (<http://www.sqlquality.com/>)

SQLQuality (エスキューエル・クオリティ) は、**日本で唯一の SQL Server 専門の独立系コンサルティング会社**です。過去のバージョンから最新バージョンまでの SQL Server を知りつくし、多数の実績と豊富な経験を持つ、OS や .NET にも詳しい **SQL Server の専門家 (キャリア 17 年以上) がすべての案件に対応します**。人気メニューの「**パフォーマンス チューニング サービス**」は、100%の成果を上げ、過去すべてのお客様環境で驚異的な性能向上を実現。チューニング スキルは**世界トップレベル**を自負、検索エンジンでは (英語情報を含めて) ヒットしないノウハウを多数保持。ここ数年は **BI/DWH システム構築支援**のご依頼が多い。

主なコンサルティング実績

- ▶ 大手映像制作会社の **BI システム構築支援** (会計/業務システムにおける予算管理/原価管理など)
- ▶ 大手流通系の **DWH/BI システム構築支援** (POS データ/在庫データ分析)
大規模テラバイト級データ ウェアハウスの物理・論理設計支援および運用管理設計支援
- ▶ 大手アミューズメント企業の **BI システム構築支援** (人事システムにおける人材パフォーマンス管理)
- ▶ 外資系医療メーカーの Analysis Services による「**販売分析**」システムの構築支援 (売上/顧客データ分析)
- ▶ **9 TB** データベースの物理・論理設計支援 (パーティショニング対応など)
- ▶ ハードウェア リプレース時の**ハードウェア選定** (最適なサーバー、ストレージの選定)、**高可用性環境**の構築
- ▶ **SQL Server 2000** (32 ビット) から **SQL Server 2008** (x64) への移行/アップグレード支援
- ▶ 複数台の SQL Server の **Hyper-V 仮想環境**への移行支援 (サーバー統合支援)
- ▶ **2 時間**かかっていた日中バッチ実行時間を、わずか **5 分**へ短縮 (**95.8%** の性能向上)
- ▶ ピーク時の CPU 利用率 **100%** のシステムを、わずか **10%** にまで軽減し、大幅性能向上
- ▶ 平均 **185.3ms** かかっていた処理を、わずか **39.2ms** へ短縮 (**78.8%** の性能向上)
- ▶ **Java 環境** (Tomcat, Seasar2, S2Dao) の SQL Server パフォーマンス チューニング etc

コンサルティング時の作業例 (パフォーマンス チューニングの場合)

- ▶ アプリケーション コード (VB, C#, Java, ASP, VBScript, VBA) の解析/改修支援
- ▶ ストアド プロシージャ/ユーザー定義関数/トリガー (Transact-SQL) の解析/改修支援
- ▶ インデックス チューニング/SQL チューニング/ロック処理の見直し
- ▶ 現状のハードウェアで将来のアクセス増にどこまで耐えられるかを測定する高負荷テストの実施
- ▶ IIS ログの解析/アプリケーション ログ (log4net/log4j) の解析
- ▶ ボトルネック ハードウェアの発見/ボトルネック SQL の発見/ボトルネック アプリケーションの発見
- ▶ SQL Server の構成オプション/データベース設定の分析/使用状況 (CPU, メモリ, ディスク, Wait) 解析
- ▶ 定期メンテナンス支援 (インデックスの再構築/断片化解消のタイミングや断片化の事前防止策など) etc

松本美穂 (まつもと・みほ)

有限会社エスキューエル・クオリティ 代表取締役

Microsoft MVP for SQL Server (2004 年 4 月~)

経産省認定データベース スペシャリスト/MCDBA/MCSD for .NET/MCITP Database Administrator

SQL Server の日本における最初のバージョンである「SQL Server 4.21a」から SQL Server に携わり、現在、SQL Server を中心とするコンサルティングを行っている。得意分野はパフォーマンス チューニングと Reporting Services。コンサルティング業務の傍ら、講演や執筆も行い、マイクロソフト主催の最大イベント Tech・Ed などでスピーカーとしても活躍中。SE や ITPro としての経験はもちろん、記名/無記名含めて多くの執筆実績も持ち、様々な角度から SQL Server に携わってきている。著書の『SQL Server 2000 でいってみよう』と『ASP.NET でいってみよう』(いずれも翔泳社刊)は、トップ セラー (前者は 28,500 部、後者は 16,500 部発行)。近刊に『SQL Server 2012 の教科書』(ソシム刊)がある。

松本崇博 (まつもと・たかひろ)

有限会社エスキューエル・クオリティ 取締役

Microsoft MVP for SQL Server (2004 年 4 月~)

経産省認定データベース スペシャリスト/MCDBA/MCSD for .NET/MCITP Database Administrator

SQL Server の BI システムとパフォーマンス チューニングを得意とするコンサルタント。過去には、約 3,000 本のストアド プロシージャのチューニングや、テラバイト級データベースの論理・物理設計、運用管理設計、高可用性設計、BI・DWH システム設計支援などを行う。アプリケーション開発 (ASP/ASP.NET, C#, VB 6.0, Java, Access VBA など) やシステム管理者 (IT Pro) 経験もあり、SQL Server だけでなく、アプリケーションや OS、Web サーバーを絡めた、総合的なコンサルティングが行えるのが強み。Analysis Services と Excel による BI システムも得意とする。マイクロソフト認定トレーナー時代の 1998 年度には、Microsoft CPLS トレーナー アワード (Trainer of the Year) を受賞。