

SQL Server 2016

SQL Server 2016 自習書シリーズ No.2

Operational Analytics ～OLTP とデータ分析の両立～

Published: 2016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るわけではありません。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。

この文章内での引用（図版やロゴ、文章など）は、日本マイクロソフト株式会社からの許諾を受けています。

© Copyright 2016 Microsoft Corporation. All rights reserved.

目次

STEP 1.	SQL Server 2016 Operational Analytics の概要.....	4
1.1	SQL Server 2016 評価版のダウンロード	5
1.2	SQL Server 2016 で提供された主な新機能	11
1.3	Operational Analytics の概要 ～OLTP とデータ分析の両立～	13
STEP 2.	Operational Analytics の検証の詳細	22
2.1	Operational Analytics の検証の詳細	23
2.2	インメモリ OLTP の検証（もちろん OLTP に強い）	27
2.3	Operational Analytics の検証環境	35
2.4	Operational Analytics を試す方法	36
2.5	Operational Analytics を試すスクリプト	49
STEP 3.	列ストア インデックスを 利用した Operational Analytics	56
3.1	列ストア インデックスを利用した Operational Analytics.....	57
3.2	列ストア インデックスの基本的な利用方法	59
3.3	列ストア インデックスの性能比較.....	62
3.4	列ストア インデックスでのデータ更新とインデックスの再構築	78
3.5	BULK INSERT で 10 万件分のデータを一括インポート.....	82
3.6	インメモリ OLTP の利用	87
STEP 4.	インメモリ OLTP の基本操作	91
4.1	インメモリ OLTP の主な特徴	92
4.2	インメモリ OLTP の基本操作	94
4.3	メモリ最適化テーブルの作成／性能比較	97
4.4	テーブル サイズ（メモリ使用量）の確認	102
4.5	ネイティブ コンパイル ストアド プロシージャによる性能向上	104
4.6	データの永続化（SCHEMA_AND_DATA）	111
4.7	メモリ最適化テーブルでの同時更新（Write-Write 競合）	118

STEP 1. SQL Server 2016 Operational Analytics の概要

この STEP では、SQL Server の最新バージョンである「**SQL Server 2016**」で提供される「**Operational Analytics**」機能の概要を説明します。

この STEP では、次のことを学習します。

- ✓ SQL Server 2016 評価版のダウンロード
- ✓ SQL Server 2016 の主な新機能
- ✓ Operational Analytics の概要

1.1 SQL Server 2016 評価版のダウンロード

SQL Server の最新バージョンである「SQL Server 2016」は、2016年6月に発売されました。

SQL Server 2016 を評価するための評価版 (Evaluation Edition) は、次の URL からダウンロードすることができます。

評価版のダウンロード : Microsoft SQL Server 2016

<http://www.microsoft.com/ja-jp/evalcenter/evaluate-sql-server-2016>

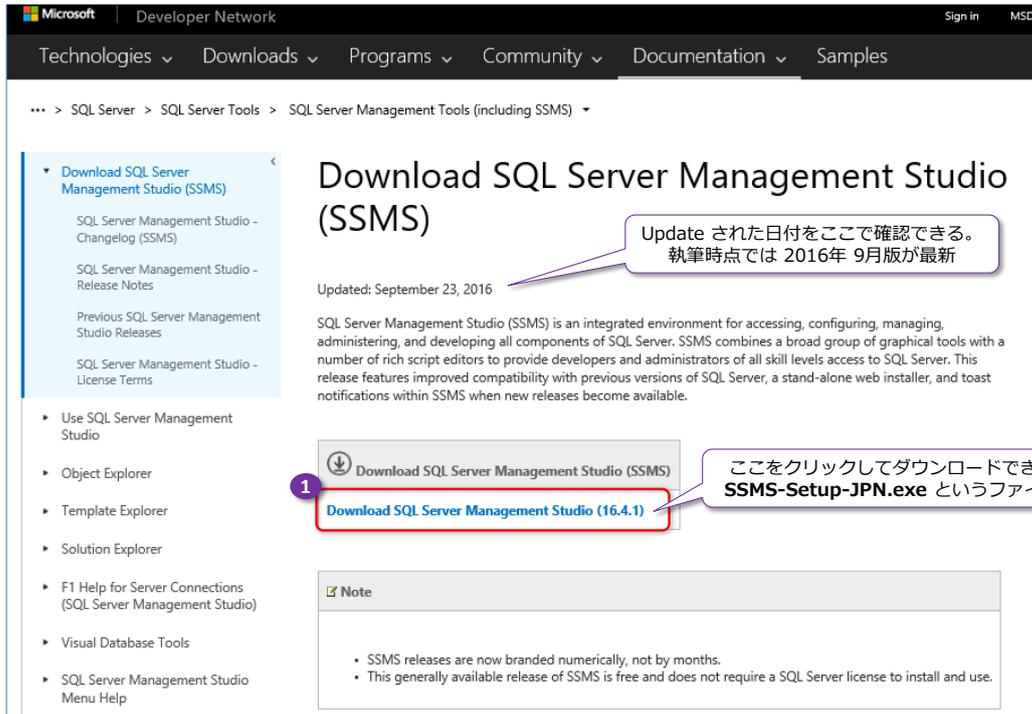
➡ Management Studio はダウンロード提供に変更

SQL Server 2016 からは、Management Studio (SQL Server の管理ツール) がダウンロード版の提供のみに変更されました。これは、クラウド (Microsoft Azure の提供する各種サービス) のアップデートにいち早く対応したり、製品へのフィードバックをいち早く反映させるためです。執筆時点 (2016年9月) では、2016年6月に提供された RTM (製品) 版を皮切りに、7月、8月、9月と、1ヶ月ごとに Update 版の Management Studio が提供されています (今後も定期的な間隔での最新版の提供が予定されています)。

最新版の Management Studio は、次の URL からダウンロードすることができます。

Management Studio の最新版のダウンロード

<http://msdn.microsoft.com/en-us/library/mt238290.aspx>

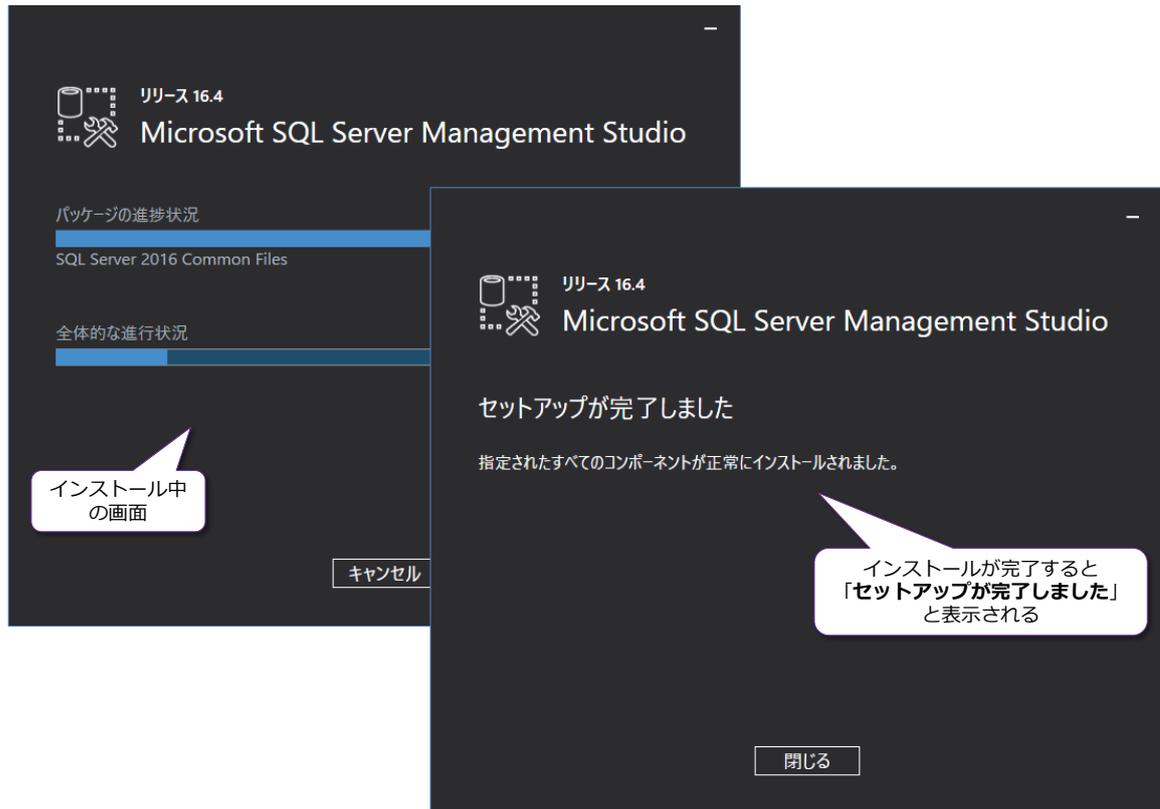


この Web サイトは、英語のページですが、日本語環境であれば、日本語版のインストーラーである「**SSMS-Setup-JPN.exe**」ファイルのダウンロードが始まるので、「**-JPN**」が付くことを確認しておいてください。

ダウンロードした **SSMS-Setup-JPN.exe** ファイルを実行すると、次のようにインストーラーが起動するので、**[インストール]** ボタンをクリックすれば、インストールを開始できます。なお、**Management Studio** と **SSDT** (SQL Server Data Tools) または **Visual Studio 2015** を同じマシンにインストールする場合には、Management Studio を最後にインストールした方が良いので、SSDT/Visual Studio 2015 と共存させる場合には、後述の手順を参考にしてください。



Management Studio のインストール中は、次のように進行状況が表示されます。



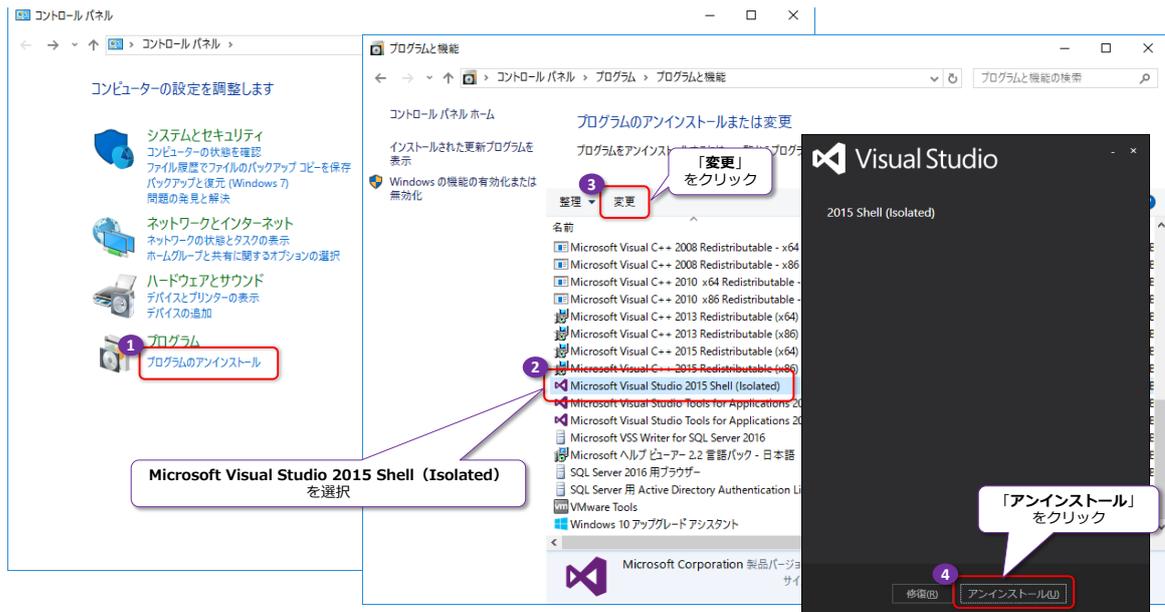
「**セットアップが完了しました**」と表示されれば、Management Studio のインストールが完了です（環境によっては、完了後に再起動が促される場合があります）。

➡ **SSDT/Visual Studio 2015 と共存する場合は、インストール順に注意**

SSDT (SQL Server Data Tools) は、SQL Server 2008 R2 までは、**BIDS** (Business Intelligence Development Studio) と呼ばれていた BI 向け (Reporting Services や Analysis Services、Integration Services) のプロジェクトを作成するためのツールです。SQL Server 2012 と 2014 では、データベース プロジェクト (.dacpac) を作成するための **SSDT** と、BIDS の後継ツールとなる **SSDT-BI** の **2 種類**の **SSDT** がありましたが、SQL Server 2016 からは 1 つの SSDT に統合されました (SSDT のダウンロードおよびインストール方法については後述します)。

執筆時点 (2016 年 9 月) での **Management Studio** の最新版では、**SSDT** または **Visual Studio 2015** を 同じマシンにインストール する場合には、Management Studio をインストールする前に、SSDT または Visual Studio 2015 を先にインストール しておく必要があります。もし、Management Studio を先にインストールした後に、後から SSDT や Visual Studio 2015 をインストールする場合には、インストールに失敗してしまうからです。

なお、SSDT のインストールに失敗してしまった場合には、次のように [コントロール パネル] の [プログラムのアンインストール] から [Microsoft Visual Studio 2015 Shell (Isolated)] を選択して、[変更] ボタンをクリックし、Visual Studio 2015 Shell (Isolated) をアンインストールするようにします。



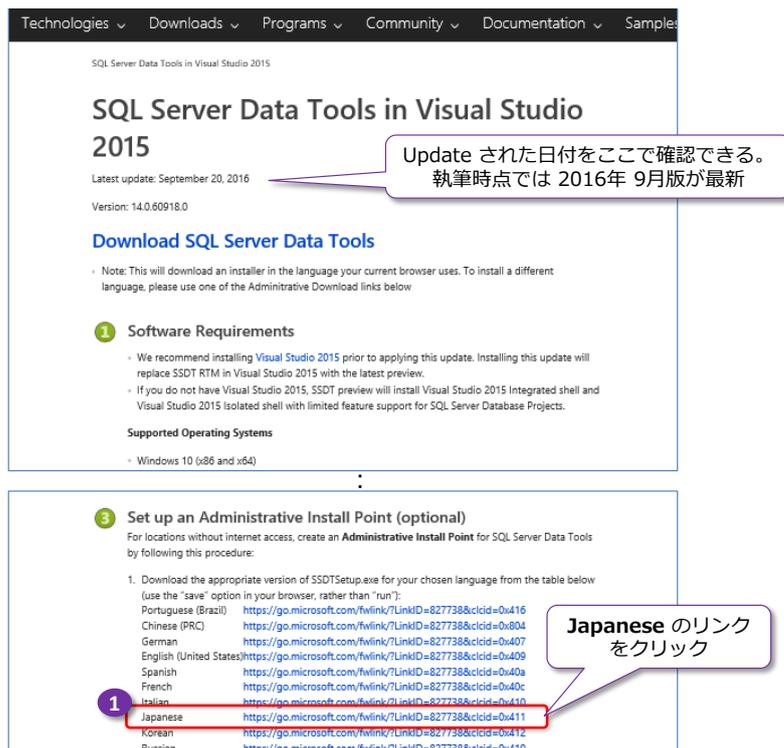
このように Visual Studio 2015 Shell (Isolated) をアンインストールしておけば、もう一度 SSDT をセットアップすることで、SSDT のインストールを成功させることができます。

もちろん、こういった手順を踏まないためにも、SSDT を先にインストールして、その後に Management Studio をインストールするようにすることをお勧めします。

➤ SSDT の最新版のダウンロードとインストール

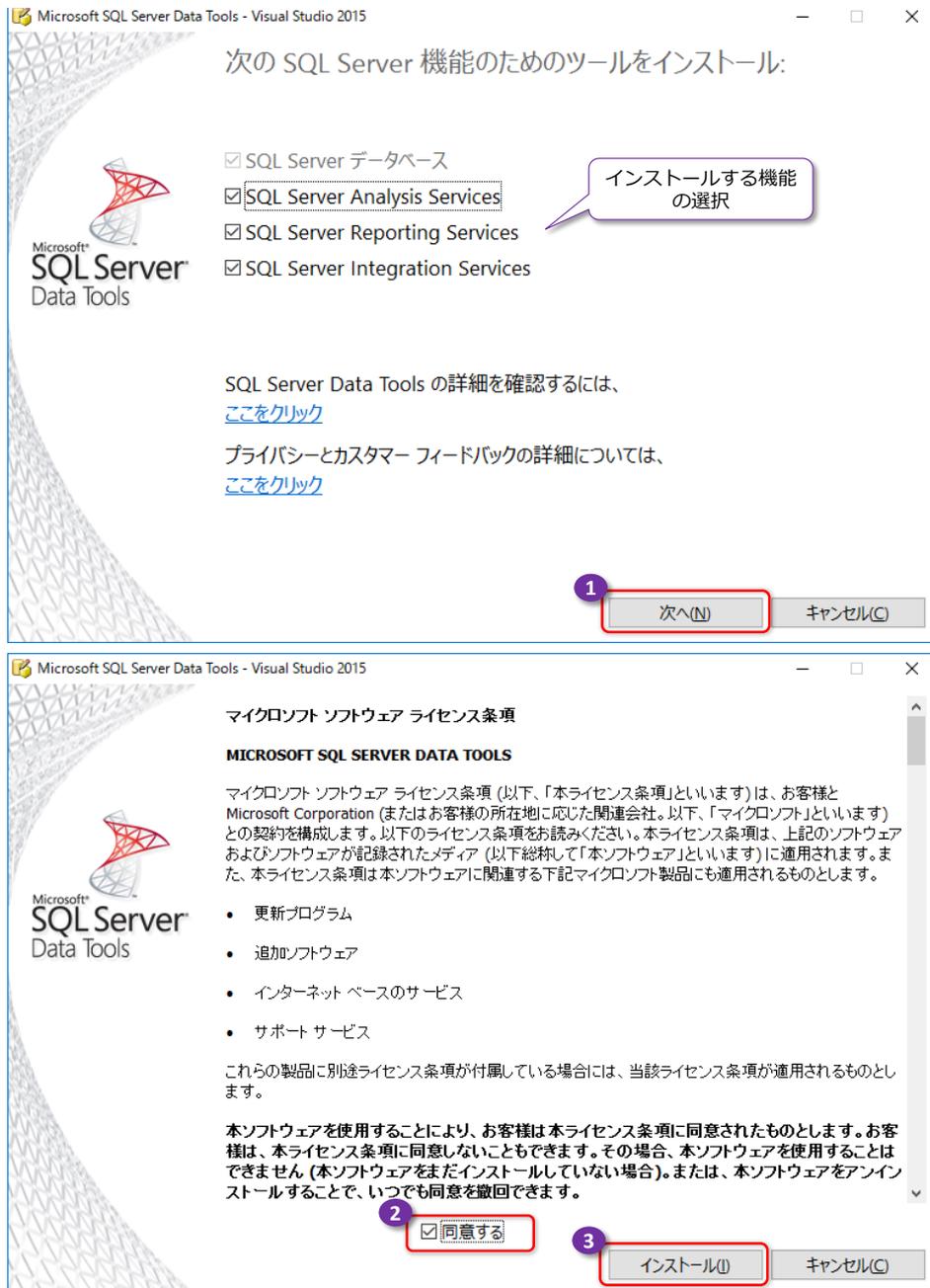
SSDT の最新版は、次の URL からダウンロードすることができます。

<http://msdn.microsoft.com/en-us/mt186501>

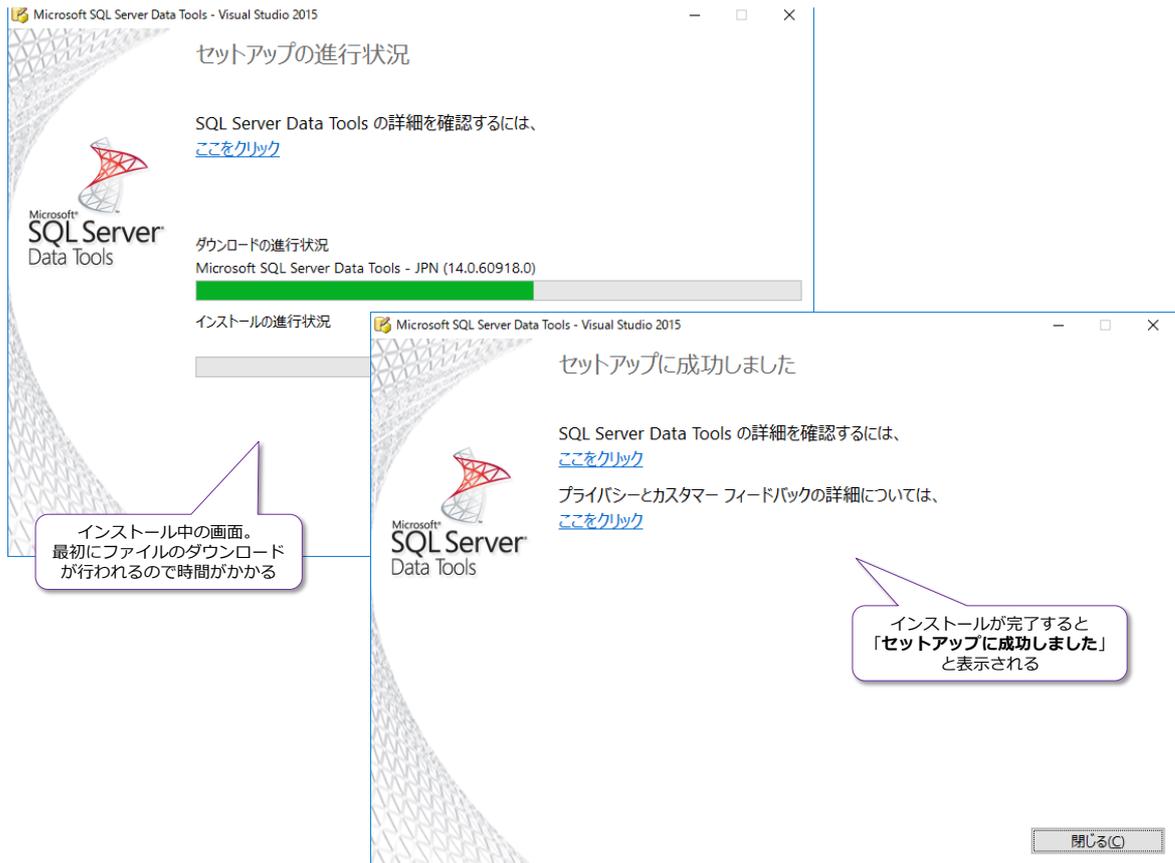


この Web サイトも英語のページですが、**Japanese** のリンクをクリックすれば、日本語版のインストーラー (**SSDTSetup.exe**) をダウンロードすることができます。

ダウンロードした **SSDTSetup.exe** ファイルをダブル クリックすると、次のようにインストールを開始することができます。



インストール中は、次のように進行状況が表示されます（最初にファイルのダウンロードが行われるので、数十分ぐらい時間がかかります）。



「**セットアップに成功しました**」と表示されれば、SSDT のインストールが完了です（環境によっては、完了後に再起動が促される場合があります）。

➡ Management Studio と SSDT、Visual Studio 2015 の 3 つを共存させる場合

Management Studio と SSDT、Visual Studio 2015 の 3 つを共存させる場合には、**Visual Studio 2015** を一番最初にインストールしておくことが重要です。これを行っておけば、Management Studio の後に SSDT をインストールしても、インストールに失敗しません。

もし、SSDT や Management Studio をインストール済みの環境に、後から Visual Studio 2015 をインストールしなければいけない場合には、Visual Studio 2015 の Update 2 以降が組み込まれたインストーラー（**with Update 2** または **with Update 3** など）を利用するようにしてください。Visual Studio 2015 の **with Update 1** のインストーラーを利用する場合は、インストールに失敗してしまうので注意してください。

なお、Visual Studio 2015 のインストールに失敗してしまった場合は、SSDT のときと同様、Visual Studio 2015 Shell (Isolated) のアンインストールを行います。Visual Studio 2015 の場合は、これに加えて、Visual Studio 2015 自体（中途半端にインストールされてしまった Visual Studio 2015）もアンインストールを行います。これを行った後に、Visual Studio 2015 のインストールを再度実行すれば、今度はインストールを成功させることができます。こうした事態にならないためにも、**with Update 2** 以降のインストーラーを利用する、あるいは Visual Studio 2015 を先にインストールしておくようにすることをお勧めします。

1.2 SQL Server 2016 で提供された主な新機能

SQL Server 2016 には、非常にたくさんの新機能が提供されています。これらをまとめると、次のようになります。

	SQL Server 2016 からの主な新機能
性能向上	<ul style="list-style-type: none"> • Operational Analytics (OLTPとデータ分析の両立) インメモリ OLTP と列ストア インデックスの融合 • インメモリ OLTP の飛躍的な進化 クラスター化列ストア インデックスのサポート、最大サイズ 2TB、64コア以上での性能向上、並列プランへの対応、TDE のサポート、ALTER のサポート、BIN2 以外の照合順序のサポート、統計の自動更新への対応、LOB のサポートなど • 列ストア インデックスの大幅強化 バッチ モードの性能向上、集計関数のプッシュダウンによる性能向上、クラスター化列ストア インデックスでの追加の B-tree インデックスのサポート、非クラスター化列ストア インデックスでの更新サポート、フィルター列インデックスのサポート、主キー/外部キー制約のサポート、AlwaysOn 可用性グループの読み取り可能セカンダリのサポート、COMPRESS_DELAY のサポートなど
セキュリティ強化	<ul style="list-style-type: none"> • 動的データ マスクによるデータのマスク (情報漏洩対策) • 行レベル セキュリティによる行レベルのアクセス制御 • Always Encrypted による暗号化 • TDE (透過的なデータ暗号化) の性能向上、インメモリ OLTP 対応 • テンポラル テーブルによる Audit
注目の新機能	<ul style="list-style-type: none"> • R 統合 (SQL Server R Services) • JSON 対応 • Stretch Database (ストレッチ データベース) • Azure バックアップ URL の性能向上 • PolyBase で Hadoop アクセス (HDFS)
可用性	<ul style="list-style-type: none"> • AlwaysOn 可用性グループの強化 自動フェールオーバーの台数が 2から 3 に増加、ログ転送性能の向上 (マルチ スレッドで処理)、ラウンドロビン レプリカ、TDE のサポート、DTC (分散トランザクション) のサポート、ワークグループ環境でも利用可能 (Windows Server 2016 を利用している場合)、Standard エディションでも利用可能 など
既存機能の強化	<ul style="list-style-type: none"> • DROP .. IF EXISTS (オブジェクトが存在しているなら DROP を実行) • tempdb の複数ファイル/初期サイズをセットアップ時に選択可能に • トレースフラグ 1117、1118、4199 のオン/オフを DB 単位で設定可能に • ライブ クエリ統計、クエリストアによる性能監視、プラン固定 • T-SQL の強化 (STRING_SPLIT、COMPRESS、FORMATMESSAGE など) • DB 単位で MAXDOP や CE (基数推定) 設定を変更可能に • bcp と Bulk Insert で Unicode 対応
BI 関連の強化	<ul style="list-style-type: none"> • Reporting Services の大幅強化 パラメーター ボックスのカスタマイズ、新しいグラフのサポート (サンバースト、ツリーマップ)、PowerPoint レンダリング、印刷コントロールの変更、iframe での埋込対応、Datazen 統合 (モバイル レポート、新しいレポート マネージャー) など • Analysis Services の強化 Tabular Model でのパーティションの並列処理のサポート、DAX エンジンの性能向上、DirectQuery の新しい実装と大幅な性能向上、計算テーブルのサポート、新しい DAX 関数 (50以上)、双方向フィルターのサポート、スクリプト モデルの変更 など • Integration Services の強化 Execute SQL タスクで R スクリプトのサポート、Azure Feature Pack、Hadoop (HDFS) のサポート、データ フローでのエラー時の列名のサポート、制御フローのテンプレート作成、AlwaysOn 可用性グループでの SSIS カタログ DB のサポート、AutoAdjustBufferSize プロパティ など • MDS の強化 性能の向上、セキュリティの強化、各種の機能強化 など

SQL Server 2016 では、**性能向上**および**セキュリティ強化**を実現できる機能が多く提供されているのが大きなポイントです。そして、この自習書のタイトルにもなっている「**Operational**

Analytics」(OLTP とデータ分析の両立) が一番の目玉機能で、Operational Analytics は今後のデータベースのあり方を大きく変えるのではないかと思えるほど、非常に大きな可能性を感じさせるものです(次の項以降で詳しく説明します)。

セキュリティに関しては、マイナンバーや各種の情報漏洩事件など、業界全体のセキュリティ意識の高まりもあって、ここ数年弊社へのお問い合わせが非常に増えている分野です。SQL Server 2016 には、セキュリティを向上させることができる機能が数多く提供されているので、そういったニーズに答えることができます。

その他、**R 統合**(SQL Server R Services) や、**JSON** 対応、**PolyBase**、**Hadoop** 対応など、最近のマイクロソフト社の方向性と同様、SQL Server でもオープン化が非常に進んでいます。

セキュリティや R 統合、JSON 対応、PolyBase、Hadoop 対応、T-SQL の強化などの既存の機能の強化については、本自習書シリーズの No.1「**SQL Server 2016 新機能の概要**」編で詳しく説明しているので、こちらもぜひご覧いただければと思います。

1.3 Operational Analytics の概要 ～OLTP とデータ分析の両立～

SQL Server 2016 の一番の目玉の新機能は「Operational Analytics」です。Operational は「OLTP」(オンライン トランザクション処理)、Analytics は「データ分析」と捕らえると分かりやすい言葉で、機能面で言うと、「インメモリ OLTP」と「列ストア インデックス」の進化/融合です(インメモリ OLTP と列ストア インデックスの良いとこどりをして、OLTP とデータ分析を両立できるようにしたものです)。

Operational Analytics (OLTPとデータ分析の両立)

Operational ワークロード

= OLTP (オンライン トランザクション処理) など



SQL Server 2014 までは
インメモリ OLTP や
通常リレーショナル テーブルを利用

Analytics ワークロード

= データ集計/分析処理、DWH、夜間バッチなど



SQL Server 2014 までは
列ストア インデックスや
Analysis Services などを利用



SQL Server 2016 では両立可能 次の 3パターンで実装できる

1. インメモリ OLTP + クラスター化列ストア インデックス (完全なインメモリ実装)
2. 通常リレーショナル テーブル + 非クラスター化列ストア インデックス
3. 通常リレーショナル テーブル + クラスター化列ストア インデックス

これまでの SQL Server では、「Operational ワークロード」と「Analytics ワークロード」に対しては、別々のテクノロジー/機能を利用して、使い分ける必要がありましたが(同時に利用するのが難しいところがありましたが)、SQL Server 2016 からは「インメモリ OLTP」と「列ストア インデックス」が飛躍的な進化を遂げたことで両立できるようになりました。具体的には、次の3つのパターンで実現することができます。

1. インメモリ OLTP + クラスター化列ストア インデックス (完全なインメモリ実装)
2. 通常リレーショナル テーブル + 非クラスター化列ストア インデックス
3. 通常リレーショナル テーブル + クラスター化列ストア インデックス

SQL Server 2014 までのインメモリ OLTP は、OLTP 向けのインメモリ リレーショナル データベース エンジン、列ストア インデックスは、データ分析/集計に強いカラム指向データベースの SQL Server 実装として別々に提供されてきましたが、SQL Server 2016 からは、インメモリ OLTP と列ストア インデックスが融合して、それぞれの良いとこどりをして (OLTP にもデータ分析にも強くなって)、完全なインメモリで Operational Analytics を実現 (OLTP とデータ分析を両立) できるようになりました。

また、これまでは読み取り専用として提供されていた「非クラスター化列ストア インデックス」が、SQL Server 2016 からは更新可能になったことで、通常のリレーショナル テーブルに、容易に列ストア インデックスを追加できるようになりました。これによって、従来ながらの b-tree インデックスを追加するのと同じような感覚で、列ストア インデックスを追加できるようになって、これを追加すれば、**Analytics ワークロード** (データ集計/分析クエリ) の性能を大幅に向上させることができます (**Operational Analytics** の実現)。

また、SQL Server 2016 では、列ストア インデックスそのものの**性能向上** (バッチ モードの強化、プッシュダウン、更新性能の向上、パラレル Insert など) も実現しているので、現在の SQL

Server で性能に問題を抱えているという方には、ぜひ試してみてください。ぜひ試してほしい機能です。

➡ インメモリ OLTP の飛躍的な進化 ~Analytics ワークロードにも対応~

インメモリ OLTP は、SQL Server 2014 から提供されたものですが、SQL Server 2014 のときには、(最初のバージョンであったこともあり) 制限事項が非常に多くありました。SQL Server 2016 からは、そういった制限事項が大きく取り払われて、インメモリ OLTP が飛躍的に進化しました。一番の進化は、前述したように**クラスター化列ストア インデックス**を追加できるようになったことで(完全なインメモリでの **Operational Analytics** の実現)、その他の具体的な強化内容は、次のとおりです。

SQL Server 2016 からのインメモリ OLTP の強化ポイント

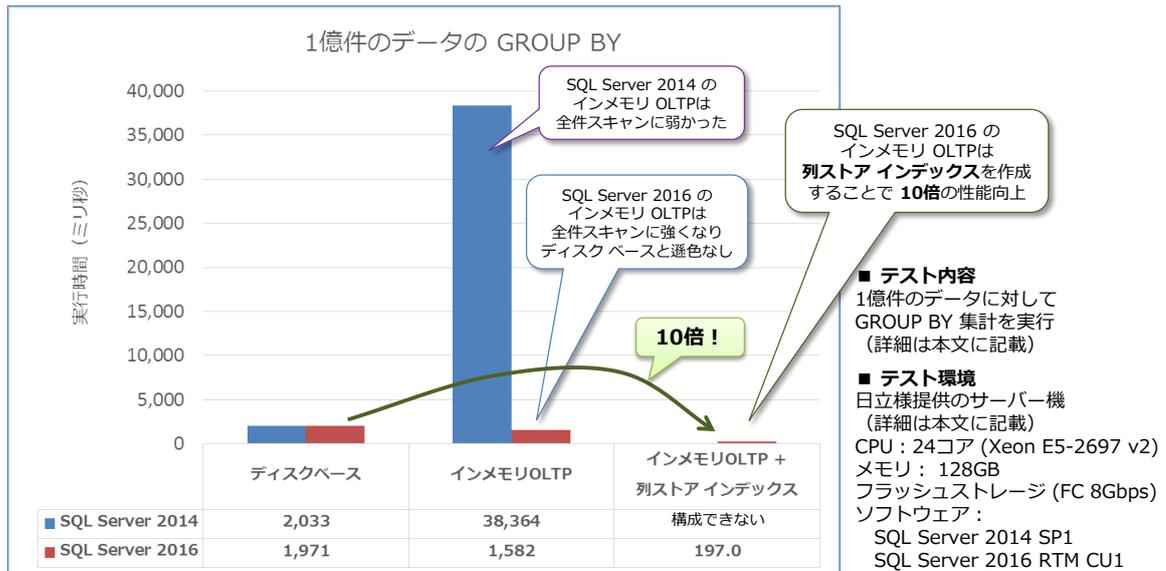
- **インメモリ OLTP で列ストア インデックスのサポート** (インメモリ OLTP のメモリ最適化テーブルにクラスター化列ストア インデックスを追加可能に)
- **最大サイズが 2TB に拡張** (大量データへの対応)
- **64 コア以上での性能向上** に対応 (スケールアップが可能)
- **並列プラン**への対応 (メニー コアの活用が可能に)
- **TDE** (透過的なデータ暗号化) のサポート (セキュリティの強化)
- **ALTER/BIN2 以外の照合順序**のサポート (既存環境からの移行しやすさが大幅に向上)
- **統計の自動更新**のサポートや、**LOB** のサポート (varchar(max) や varbinary(max) などのラージ オブジェクトのサポート)

SQL Server 2014 のときには、制限事項の多さや、大規模環境でのスケール面 (64 コア以上だとスケールしない)、集計クエリでの性能面 (並列プランに未対応) など、既存環境 (ディスクベースのテーブル) をインメモリに移行するのが難しかったところがありましたが、SQL Server 2016からは、そういった制限事項が大きく取り払われました。特に **ALTER のサポート**や **BIN2 以外の照合順序のサポート**は、移行という観点で非常に大きく、SQL Server 2014 のときには、これが原因で移行を断念しているというお客さんを見てきたので、SQL Server 2016 からは格段と移行しやすくなりました。

また、SQL Server 2016 からはインメモリ OLTP の最大サイズが **2TB** に増えたことで (SQL Server 2014 のときは **512GB** が最大サイズ)、データ量が多くてもインメモリ OLTP に移行できるようになりました。

➡ 検証結果 : インメモリ OLTP + クラスター化列ストア インデックス

実際に、SQL Server 2016 のインメモリ OLTP とクラスター化列ストア インデックス (完全なインメモリでの Operational Analytics の実装) を利用して、どれぐらいの性能が出るのかを検証してみたのが、次のグラフです。



* ベンチマークの結果の公表は、使用許諾契約書で禁止されていますが、その効果をわかりやすく表現するため、日本マイクロソフト株式会社の監修のもと、数値を掲載しております。

このグラフは、**SQL Server 2016** の **RTM** (製品版) に **CU1** (累積的な修正プログラム 1) を適用したものと、**SQL Server 2014** に **SP1** を適用したものを利用して、**1億件のデータ** に対する **GROUP BY** 演算 (データ集計処理) を行った結果です (テーブル構成や実行した SQL の詳細は後述します)。

ディスク ベースの通常テーブルでは **1.97 秒** かかった集計処理が、**インメモリ OLTP** にすることで **1.58 秒** (19.8%の性能向上)、さらに**列ストア インデックス** (クラスター化列ストア インデックス) を作成することで、わずか **197 ミリ秒** で処理できるようになり、**10 倍もの性能向上** を確認することができました。これに対して、SQL Server 2014 では、**2.03 秒** かかっていた処理が、インメモリ OLTP を利用することで **38.36 秒** もかかってしまい、**18.9 倍** も遅い結果になってしまっています (SQL Server 2014 で遅くなる理由については後述します)。

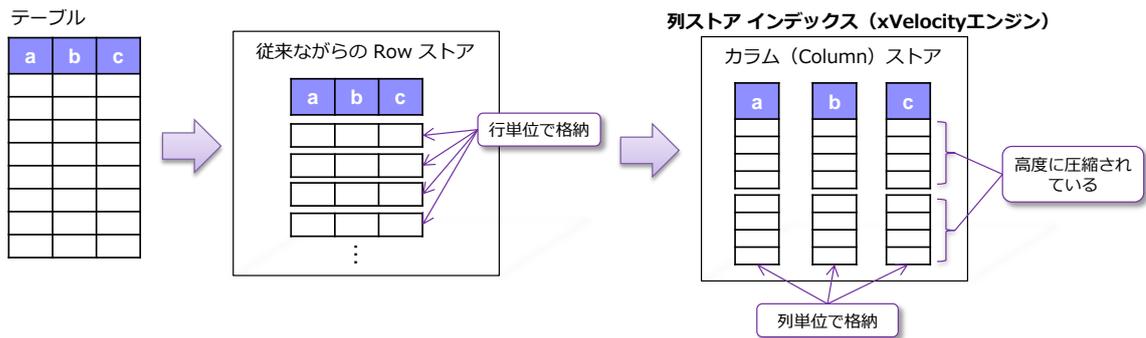
このように、SQL Server 2016 のインメモリ OLTP は、クラスター化列ストア インデックスを追加できるようになったことで、データ分析/集計 (Analytics ワークロード) にも強くなりました。

➡ 列ストア インデックスの進化 ~カラム指向とリレーショナル DB の融合~

列ストア インデックス (Column-store Index) は、大量のデータを分析/集計するときに強さを発揮する**カラム指向データベースの SQL Server 実装**として、SQL Server 2012 のときに提供されたものです。列ストア インデックスは、「**xVelocity 列ストア インデックス**」と呼ばれることもあり、PowerPivot および Analysis Services の Tabular Mode (テーブル モード) で採用されている**インメモリの BI エンジン**である「**xVelocity エンジン**」を RDB (リレーショナル データベース) に応用したものです。なお、このエンジンは、SQL Server 2008 R2 のときに開発されて (PowerPivot for Excel として実装)、その当時は **VertiPaq エンジン**と呼ばれていました。

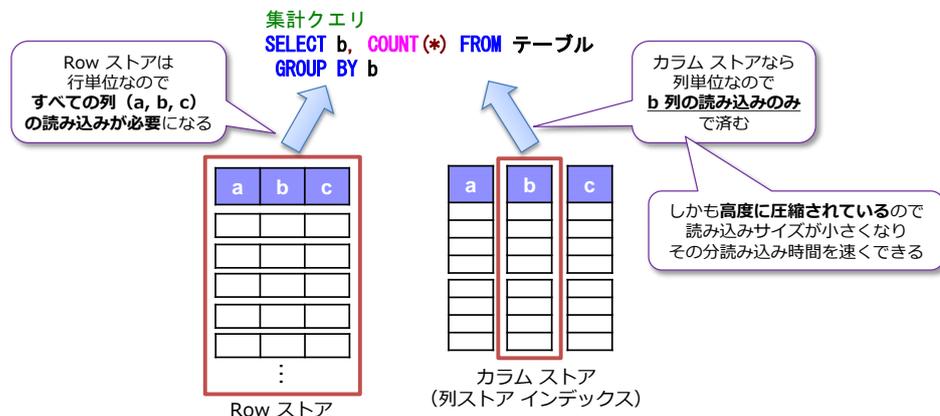
列ストア インデックスの xVelocity エンジンでは、次のように**列単位**でインデックスを格納し、

それらは高度に圧縮されています。



このインデックスによって、大量のデータに対する集計処理 (GROUP BY 演算など) の性能を大きく向上させることができます。性能が向上する理由は、次のような集計クエリを考えると分かりやすいと思います。

集計クエリの動作の違い



列ストア インデックス (コラム ストア) であれば、集計対象となる列データを読み込むだけで済み、かつそのデータは高度に圧縮されているので、読み込み時間を大幅に短縮することができます。例えば、弊社のお客様 (**100 億件の DWH**) では、列ストア インデックスを作成することで、**531GB** のテーブル (Row ストア) が **90GB** (コラム ストア) へと、**約 1/6** のサイズにまで圧縮することができます。このことから、読み込み量を小さくできることが分かります。

列ストア インデックスは、SQL Server 2012 では、読み取り専用モードの「**非クラスター化列ストア インデックス**」のみがサポートされていましたが、SQL Server 2014 からは更新可能な列ストア インデックスとして「**クラスター化列ストア インデックス**」が提供されました。そして、今回の SQL Server 2016 からは、これらが大幅に強化されて、その主なものは次のとおりです。

- **インメモリ OLTP にクラスター化列ストア インデックス**を作成可能に
(=完全なインメモリでの Operational Analytics の実装)
- **列ストア インデックスの性能向上** (バッチ モードの性能向上、集計のプッシュダウン、更新性能の向上、ウィンドウ関数のバッチ モード対応など)
- 非クラスター化列ストア インデックスが**更新可能**に
- 非クラスター化列ストア インデックスで**フィルター列**が利用可能に

- クラスター化列ストア インデックスでの追加の **B-tree インデックス** のサポート
- 主キー／外部キー制約のサポート
- AlwaysOn 可用性グループの読み取り可能セカンダリのサポート
- COMPRESS_DELAY (遅延圧縮) のサポート

これまでは **Analytics ワークロード** (データ集計／分析) に特化していた列ストア インデックスが、SQL Server 2016 からは **Operational ワークロード (OLTP)** にも対応して、**Operational Analytics** (OLTP とデータ分析の両立) を実現できるようになりました。特に、OLTP ワークロードで強さを発揮する「**インメモリ OLTP**」に**クラスター化列ストア インデックス**を作成できるようになったことが大きな進化で、その効果は前掲のグラフのとおりです。

また、従来は、読み取り専用で提供されていた「**非クラスター化列ストア インデックス**」が更新可能になったことで、今まで利用していたシステムに、容易に列ストア インデックスを追加できるようになりました。現在、「**データ分析／集計で時間がかかっている**」や「**夜間バッチの実行に時間がかかっている**」などの悩みを抱えている場合には、まず非クラスター化列ストア インデックスを作成してみることをお勧めします。夜間バッチの前に (バッチ実行時の最初の処理として) 非クラスター化列ストア インデックスを追加して、夜間バッチが完了したら削除する、といった使い方もできるので、性能に問題を抱えている場合は、ぜひ試してみてください。

既存環境からの移行という観点では、**PRIMARY KEY 制約** がサポートされるようになった点は大きく、これまでの構成を大きく変更することなく移行できるようになりました。

今回の SQL Server 2016 の列ストア インデックスは、バッチ モードの性能向上や、更新性能の向上も実現しているので、SQL Server 2014 のときに試したことがあるという方も、ぜひもう一度試してみてください。

➔ 列ストア インデックスはハイブリッドな動作が可能

完全なインメモリでの Operational Analytics (インメモリ OLTP と列ストア インデックスの組み合わせ) を利用する場合には、インメモリ OLTP が完全にインメモリで動作するので、メモリ上の制限 (最大サイズが 2TB までという上限) を受けます。したがって、数 TB (テラバイト) 規模になるなど、メモリに載りきれないデータ量になる場合には、インメモリ OLTP を利用することができません。

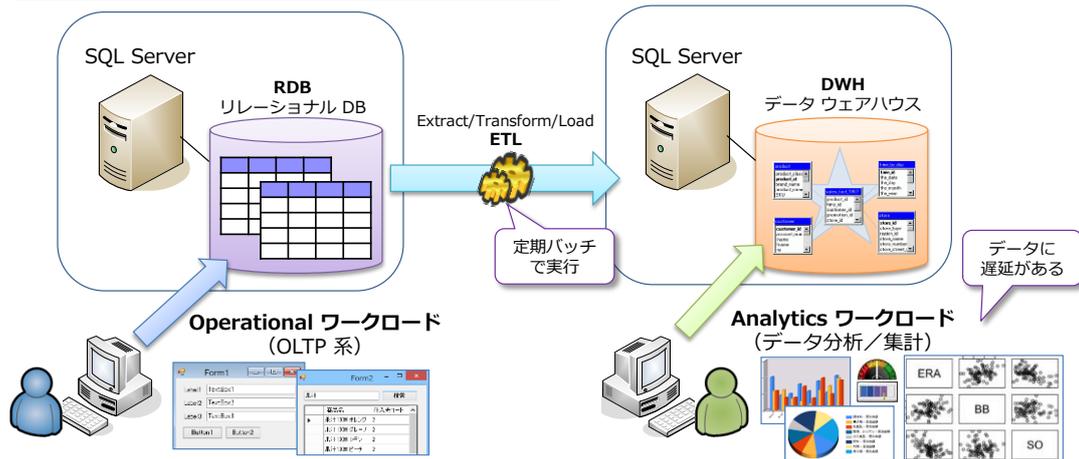
このような場合には、通常のリレーショナル テーブル (従来ながらのディスク ベースのテーブル) に、**列ストア インデックス**を追加して、Operational Analytics を実現することができます。列ストア インデックスは、基本はインメモリで動作しますが (データ量が物理メモリに載りきる場合はインメモリで動作)、メモリに載りきれないデータがあった場合には、ディスクを利用して動作させることができるからです (ハイブリッドな動作ができます)。

このように性能向上に関するオプションが増えたことは、本当にワクワクします (今回の SQL Server 2016 の進化は、今後のデータベースのあり方を左右するのではないかと思えるほど、非常に大きな可能性を感じています)。

➔ DWH (データ ウェアハウス) は不要? ~リアルタイムなデータ分析へ~

最近では、データ分析/集計をできる限りリアルタイムで実現したいというニーズの高まりから、DWH (データ ウェアハウス) を構築しないというケースが増えてきました。これまでは、次の図のように、ETL (Extract/Transform/Load) を定期的なバッチ処理で実行して、DWH を構築しているという形が主流でした。

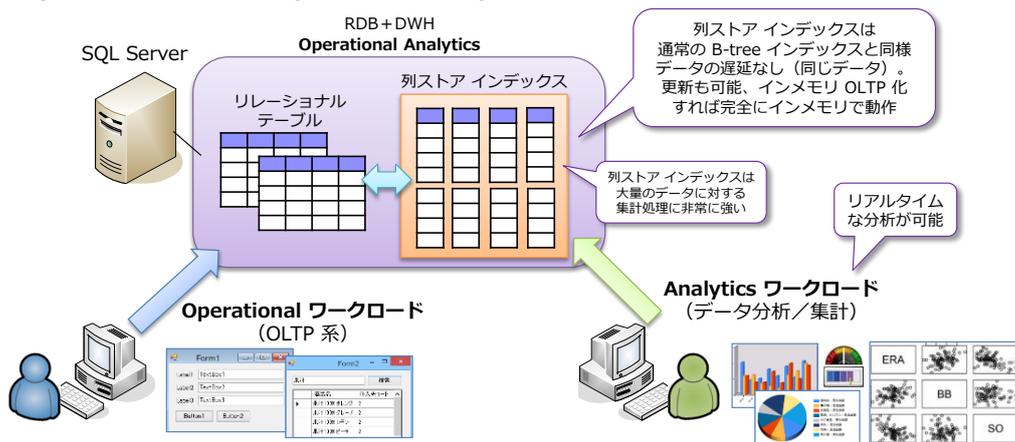
従来ながらの DWH では、データ分析に遅延がある



このようなシステムの場合、ETL ツールを介することによって (定期バッチでの実行になるため)、データ分析には遅延が発生していました (データの鮮度が落ちていました)。

そこで、最近では、ハードウェアが進化したことも影響していますが (特に大容量メモリを安価に搭載できるようになったことも大きいのですが)、データ ウェアハウスを構築せずに、1 台のマシンで OLTP もデータ分析も実現してしまおうという動きが広まっています。

SQL Server 2016 での Operational Analytics



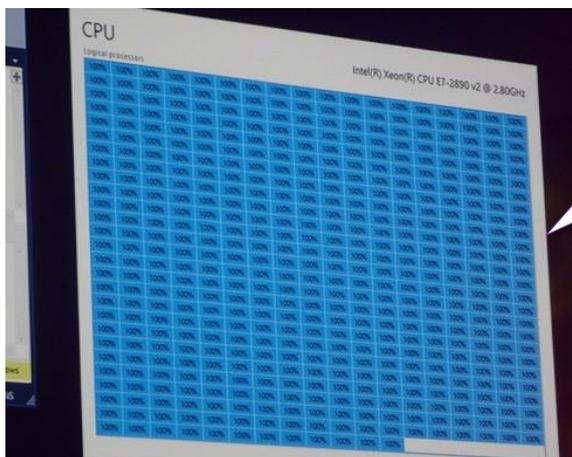
このように OLTP に DWH の役割も任せてしまえば、データ分析/集計をリアルタイムでできるようになり、このような実装に最適なのが SQL Server 2016 の Operational Analytics です。SQL Server 2016 では、インメモリ OLTP と列ストア インデックスが飛躍的に進化したことで、こうした動きがさらに加速していくように思っています。

なお、このような RDB と DWH を融合した構成を「リレーショナル データ ウェアハウス」や

「Real-Time Operational Analytics」と呼んでいる人もいます。

➔ SQL Server 2016 では 480 スレッドでもスケール

最近では、1 個の CPU あたりのコア数がどんどん増える “メニー コア” 時代になりましたが、コア数が増えてもスケールしないデータベースが存在する中（特に 64 コアを超える K グループ環境に対応していないデータベースがある中）、SQL Server はメニー コア環境でも、K グループをまたがったとしてもスケールします。昨年（2015 年）に開催された SQL Server の世界最大規模のイベントである「PASS Summit 2015」の基調講演では、**480 スレッド**分もの CPU を 100% フル活用して、スケールしているデモが行われました（以下の写真）。



Hewlett-Packard 社の Superdome で Xeon E7-2890v2 15コアの CPU を **16基**（合計**240コア**）搭載したマシンで Hyper-Threadingオンで **480スレッド分** 全てのスレッドが **100%** になってる様子

* PASS Summit 2015 の基調講演中のデモを筆者が撮影した写真

これは**データ分析／集計クエリ**を実行しているときの画面で、CPU をフル活用できていることが分かります。1 個の CPU で 15 コア（30 スレッド）というのも時代の流れを感じますが、こうしたメニー コアに対応できるのも SQL Server の大きな特徴です。

➔ SQL Server 2016 は TPC-H ベンチマークの 10TB でワールド レコード

TPC-H は、データ ウェアハウス／意思決定支援システム向けの**ベンチマーク テスト**として有名なものですが、SQL Server 2016 は、**10TB**（テラ バイト）規模のベンチマーク テストで**世界記録**（ワールド レコード）を更新しました。執筆時点（2016 年 9 月）では、Non-Clustered の 10TB 部門では、上位 1、2 位を SQL Server 2016、3 位～7 位を SQL Server 2014 が獲得して、SQL Server だけで**上位 1 位～7 位を独占**しています（以下の URL で最新のベンチマーク結果を参照できます）。

TPC-H - Top Ten Performance Results - Non-Clustered

http://www.tpc.org/tpch/results/tpch_perf_results.asp?resulttype=noncluster

この世界記録に関しては、SQL Server チームの Blog にも投稿されているので、以下の記事が参考になると思います。

SQL Server Team Blog :

SQL Server 2016 posts world record TPC-H 10 TB benchmark

<http://blogs.technet.microsoft.com/dataplatforminsider/2016/07/18/sql-server-2016-posts-world-record-tpc-h-10-tb-benchmark/>

Server & Tools Blogs > Data Platform Blogs > SQL Server Blog Sig

Data Development
Data Quality Services
OLTP
Integration Services
Data Security & Storage
Data in the Cloud
Cortana Intelligence and Machine Learning

SQL Server Blog

Official News from Microsoft's Information Platform

SQL Server 2016 posts world record TPC-H 10 TB benchmark

July 18, 2016 by [SQL Server Team](#) // 4 Comments ★★★★★

f 0

t 330

in 321

f

t

in

e

r

SQL Server 2016 delivers unparalleled performance and security built-in for your most mission critical transactional systems and data warehouses, along with an integrated business intelligence and advanced analytics solution for building intelligent applications. Blazing-fast performance is key to ensuring you can deliver a flawless transactional experience while at the same time support demanding real-time operational analytics over the data as fast as the data is coming in.

Recently, Lenovo announced the [number one TPC-H 10TB benchmark world record](#)¹ using SQL Server 2016 and Windows Server 2016 on Lenovo System x3850 X6 using the latest Intel Xeon E7 processor technology. In May 2016, Lenovo also published a [new number one TPC-H 30TB world record](#)² using SQL Server 2016 and Windows Server 2016 on Lenovo System x3950 X6. These results, in addition to recent benchmarks by software and hardware partners, as well as key applications, show that SQL Server 2016 is the fastest in-memory database on the planet for your applications.³

SQL Server 2016 owns the top TPC-E performance benchmarks⁴ for transaction processing, the top TPC-H performance benchmarks for data warehousing, and the top performance benchmarks with leading business applications. PROS Holdings uses SQL Server 2016's superior performance and built-in R Service to deliver advanced analytics more than 100x faster than before, resulting in higher profits for their customers. KPMG, a leader in audit, tax, and advisory solution, posted 2.5x faster execution time with ten times the table compression with their solution using SQL Server 2016.

Customers can also gain tremendous performance improvement by simply upgrading to SQL Server 2016 without application changes (e.g. queries will run up to 34x faster)⁵. In addition to leading performance benchmarks, SQL Server 2016 also delivers top price/performance for both workloads providing customers with significantly reduced total cost of ownership.

Easily experience SQL Server 2016 by creating a test environment using an [Azure SQL VM](#). You can also experience the full features through the [free developer edition](#) (you will be prompted to sign in to Visual Studio Dev Essentials before you can download SQL Server 2016 Developer Edition). Visit [SQL Server 2016](#) to learn more about new features and download the [SQL Server 2016 e-book](#).

SQL Server 2016

Delivers unparalleled performance

#1 OLTP Performance	#1 DW Performance	#1 Price/Performance	Unparalleled App Performance
Leader in TPC-E	#1 in 30TB, 10TB, 1TB TPC-H Windows Server 2016	#1 in TPC-E & 1TB, 10TB, 30TB TPC-H Windows Server 2016	100x faster scoring

Performance gains just by upgrading

7x faster throughput
34x faster queries
3.6x faster reporting

Search

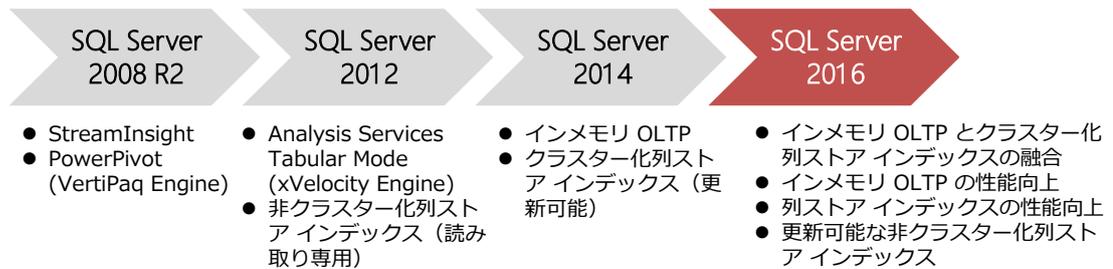
Search this blog Search all blogs

Archives

- August 2016 (3)
- July 2016 (7)
- June 2016 (10)
- May 2016 (3)
- April 2016 (10)
- March 2016 (18)
- February 2016 (8)
- January 2016 (11)
- December 2015 (13)
- All of 2016 (70)
- All of 2015 (96)
- All of 2014 (124)
- All of 2013 (115)
- All of 2012 (109)
- All of 2011 (80)
- All of 2010 (71)
- All of 2009 (54)
- All of 2008 (66)
- All of 2007 (12)

Note : SQL Server のインメモリ技術の歴史は長い (CEP、BI、ビッグデータもカバー)

SQL Server では、既に多くのインメモリ技術が提供されています。3つ前のバージョンである SQL Server 2008 R2 から提供された StreamInsight および PowerPivot に始まり、以下のインメモリ技術が提供されています。

SQL Server のインメモリ技術

StreamInsight は、**CEP** (Complex Event Processing : 複合イベント処理) を実現することができる機能で、SQL Server 2008 R2 から提供されています。CEP は、各種のセンサーデータや株取引情報、Web サーバーのログ (クリック ストリーム) などのように、絶え間なく大量に流れてくるようなストリーム データ (イベント) を処理することができるアプリケーションのことを指し、StreamInsight は、インメモリで動作することで高速なイベント処理が可能です。現在、StreamInsight は、Microsoft Azure 上のクラウド サービスである「**Stream Analytics**」として利用することもできます (**IoT** : Internet of Things データの処理に利用することができます)。

PowerPivot は、**インメモリの BI** 機能で、こちらも SQL Server 2008 R2 から提供されました。SQL Server 2008 R2 では、クライアント版の PowerPivot for Excel、サーバー版の PowerPivot for SharePoint が提供され、数億件レベルのデータでも、非常に高速な集計処理ができるインメモリの BI エンジン (インメモリで動作するカラムベースのエンジン) です。PowerPivot のエンジンは、SQL Server 2012 からは、**xVelocity エンジン**へと名称変更・改良されて、Analysis Services (分析サーバー) でも利用できるようになりました (テーブルモデル : Tabular Model と呼ばれる)。これにより、ビッグデータ/より大量のデータにも対応可能な**サーバー版のインメモリ BI** 機能が利用できるようになりました。そして、この**xVelocity エンジン**を RDB に応用したのが「**列ストア インデックス**」です。

SQL Server 2014 からは、OLTP 向けの「**インメモリ OLTP**」が提供され、そして今回の SQL Server 2016 では「**インメモリ OLTP と列ストア インデックスの進化/融合**」によって、OLTP もデータ分析も両立できるようになりました。

このように、SQL Server のインメモリ技術の歴史は古く、CEP から BI、ビッグデータ、OLTP まで、幅広くカバーしています。

STEP 2. Operational Analytics の検証の詳細

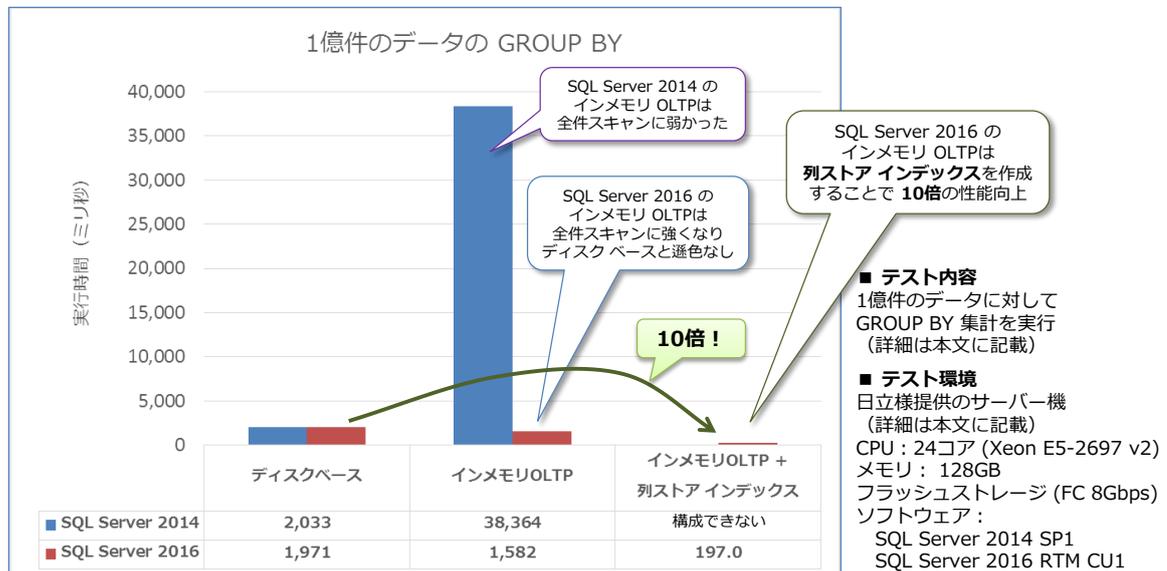
この STEP では、SQL Server 2016 を利用して Operational Analytics の検証を行った、その検証の詳細を説明します。また、インメモリ OLTP の検証として、多重度 150 で 5,000 万件の INSERT を行った場合の性能も測定しているので、これについても説明します。

この STEP では、次のことを学習します。

- ✓ Operational Analytics の検証内容
- ✓ 検証で利用したスクリプト
- ✓ インメモリ OLTP の検証（多重度 150 で 5,000 万件の INSERT）
- ✓ 検証環境（株式会社 日立製作所様のハーモニアス・コンピテンス・センター）

2.1 Operational Analytics の検証の詳細

1章で紹介した Operational Analytics の検証結果を再掲します。



* ベンチマークの結果の公表は、使用許諾契約書で禁止されていますが、その効果をわかりやすく表現するため、日本マイクロソフト株式会社の監修のもと、数値を掲載しております。

このグラフは、**SQL Server 2016** の **RTM** (製品版) に **CU1** (累積的な修正プログラム 1) を適用したものと、**SQL Server 2014** に **SP1** を適用したものを利用して、**1億件のデータ** に対する **GROUP BY** 演算 (データ集計処理) を行った結果です (テーブル構成や実行した SQL の詳細は後述します)。

ディスク ベースの通常テーブルでは **1.97 秒** かかった集計処理が、インメモリ OLTP にすることで **1.58 秒** (19.8%の性能向上)、さらに列ストア インデックスを作成することで、わずか **197 ミリ秒** で処理できるようになり、**10 倍もの性能向上** を確認することができました。

SQL Server 2014 では、**2.03 秒** かかっていた処理が、インメモリ OLTP を利用することで **38.36 秒** もかかってしまい、**18.9 倍** も遅い結果になってしまっている理由は、スキャン性能の差によるものですが、詳しくは後述します。

➡ 検証の詳細 ~テーブル構成、1億件のデータ~

検証で使用したテーブルは、次のように作成しています。

検証で使用したテーブル構成

ディスク ベースの通常テーブル

```
CREATE TABLE OnDisk_CL
(
  col1 int IDENTITY(1,1) NOT NULL
    CONSTRAINT idx1 PRIMARY KEY CLUSTERED
  ,col2 int NOT NULL
  ,col3 int NOT NULL
  ,col4 int NOT NULL
  ,col5 datetime NOT NULL
)
```

インメモリ OLTP のメモリ最適化テーブル

```
CREATE TABLE InMem_HASH
(
  col1 int IDENTITY(1,1) NOT NULL
    PRIMARY KEY NONCLUSTERED
    HASH WITH (BUCKET_COUNT = 100000000)
  ,col2 int NOT NULL
  ,col3 int NOT NULL
  ,col4 int NOT NULL
  ,col5 datetime NOT NULL
) WITH ( MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA )
```

PRIMARY KEY は
HASH インデックス

インメモリ OLTP のメモリ最適化テーブルに列ストア インデックスを追加

```
CREATE TABLE InMem HASH ccsi
( col1 int IDENTITY(1,1) NOT NULL
  PRIMARY KEY NONCLUSTERED
  HASH WITH (BUCKET_COUNT = 100000000)
, col2 int NOT NULL
, col3 int NOT NULL
, col4 int NOT NULL
, col5 datetime NOT NULL
, INDEX CCI_InMem_HASH_ccsi CLUSTERED COLUMNSTORE
) WITH ( MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA )
```

PRIMARY KEY は
HASH インデックス

クラスター化列ストア
インデックスを追加

3種類のテーブル (ディスク ベースの通常テーブル、インメモリ OLTP のメモリ最適化テーブル、メモリ最適化テーブルに列ストア インデックスを追加したもの) を、**同じ列構成** (col1~col5 の5列) で作成しています。

1億件のデータは、次のように作成しています。

1億件のデータの追加 (乱数を利用)

```
DECLARE @i int = 1
WHILE @i <= 100000000
BEGIN
  DECLARE @col2 int = CONVERT(int, RAND() * 20000000)
  , @col3 int = CONVERT(int, RAND() * 1000000)
  , @col4 int = CONVERT(int, RAND() * 10000)
  , @col5rnd int = CONVERT(int, RAND() * 2628000) + 1
  DECLARE @col5 datetime = DATEADD(minute, @col5rnd, '2009/01/01')
  INSERT INTO data100M VALUES (@col2, @col3, @col4, @col5)
  SET @i += 1
END
```

col2, col3, col4, col5 には
乱数が格納されるように
RAND 関数を利用

col1 (PRIMARY KEY) には IDENTITY で生成した連番、**col2, col3, col4, col5** には**乱数** (RAND 関数で生成した値) を格納して、実際のデータは次のようになっています。

SELECT * FROM data100M

col1	col2	col3	col4	col5
1	5023503	324864	8013	2010-05-28 01:12:00.000
2	1159690	859684	6817	2013-09-02 02:08:00.000
3	12554582	562761	7336	2013-02-03 15:16:00.000
4	11344397	873227	9046	2010-01-01 01:53:00.000
5	6066207	697032	9903	2010-01-01 12:58:00.000
6	16121127	770155	233	2009-01-01 22:53:00.000
7	4623271	172953	301	2009-01-01 00:00:00.000
8	6364128	971945	5264	2009-03-27 21:23:00.000
9	124477	35879	7549	2013-06-25 04:50:00.000
10	1	825	11	2011-07-24 00:31:00.000
11	1	496	12	2012-07-31 13:04:00.000
12	124477	35879	7549	2013-06-25 04:50:00.000
13	12975051	1000000	10000	2009-01-01 00:00:00.000
14	6474652	472140	923	2012-04-09 14:07:00.000
15	1967	1999999	757	2011-12-09 05:35:00.000
16	1020	1999999	65	2009-10-14 19:40:00.000
17	17	20000000	10	2010-05-28 01:12:00.000
18	18	20000000	7	2010-05-28 01:12:00.000

col4 int
0~9,999
の乱数

col1 int
IDENTITY で
1 からの連番

col3 int
0~999,999
の乱数

col5 datetime
2009/1/1 以降
の日付の乱数

col2 int
0~19,999,999
の乱数

CONVERT(int, RAND() * 2628000) + 1
DATEADD(minute, @col5rnd, '2009/01/01')

3種類のテーブルには、同じデータが格納されるようにするために (公平な検証にするために)、**data100M** という名前の中間テーブルに 1 億件のデータを格納していて、これを **INSERT .. SELECT** でそれぞれのテーブルにデータ複製するようにしています。

➔ 検証で使った集計クエリ (col4 で GROUP BY)

検証では、次のように col4 列で GROUP BY (データ集計) をしたクエリを使用しています。

```
-- 検証で使ったクエリ (1億件のデータを集計、col4 列で GROUP BY)
SELECT col4, COUNT(*) AS cnt FROM <テーブル名>
GROUP BY col4
ORDER BY col4
```

col4 で GROUP BY をすることで 1万件の結果が返る

各値には約 1万件のデータがあり 1万件 * 1万件 = 1億件のデータ

col4	cnt
0	9881
1	9936
2	10016
3	9988
4	10004
5	10007

col4 列には、0~9,999 の範囲の乱数が格納されているので、1万件の結果が返ります。

➔ 検証結果の詳細 ~実行プランなど~

ディスク ベースの通常テーブルで、集計クエリを実行したときの結果は、次のとおりです (SET STATISTICS TIME/IO ON で計測した実行時間と I/O 数)。

ディスク ベースの通常テーブルに対して実行

1万件の結果が返っていることが分かる

SQL Server 実行時間: CPU 時間 = 34388 ミリ秒、経過時間 = 1936 ミリ秒。 実行時間は 約2秒 (SQL Server 2016 の場合)

また、このときの実行プランは、次のように Clustered Index Scan になっています。

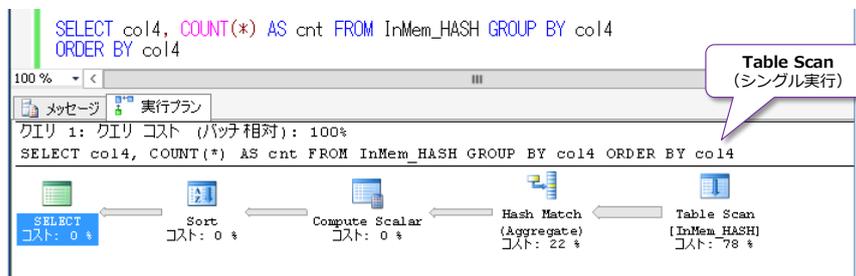
ディスク ベースでは Clustered Index Scan

これに対して、インメモリ OLTP のメモリ最適化テーブルで同じクエリを実行したときの実行プランは、次のようになります。

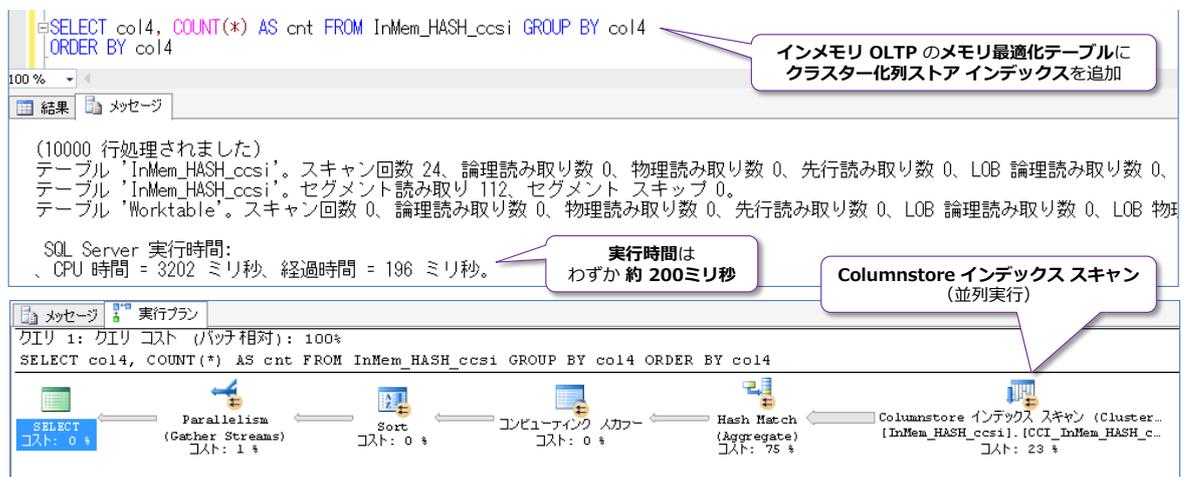


Table Scan で実行されて、**Parallelism** (並列実行) になっているのがポイントです (実行時間は、前述のグラフのとおり、約 **1.6 秒**です)。SQL Server 2016 からは並列プランに対応するようになったので、**Parallelism** で処理されています。

これと同じクエリを **SQL Server 2014** で実行した場合は、次のように **Parallelism** にはなりません (SQL Server 2014 は、並列プランに対応していないので、シングル スレッド実行になってしまいます)。



同じクエリを、SQL Server 2016 のインメモリ OLTP (メモリ最適化テーブル) に**クラスター化列ストア インデックス**を追加したテーブルで実行すると次のようになります。



実行プランには、**Columnstore インデックス スキャン**と表示されて、列ストア インデックスが利用されていることが分かります。また、実行時間もわずか約 **200 ミリ秒**であることが分かり、1 億件のデータ集計を 200 ミリ秒以内で実行できるという、**圧倒的な性能** (ディスク ベースよりも 10 倍も速い) が出ていることを確認できます。

このように、SQL Server 2016 のインメモリ OLTP は、列ストア インデックスと融合することによって、データ分析/集計にも強くなっています。

2.2 インメモリ OLTP の検証（もちろん OLTP に強い）

ここまでは、**インメモリ OLTP + 列ストア インデックス**（OLTP とデータ分析の両立）の検証結果を説明しましたが、インメモリ OLTP は、その名のとおりに **OLTP に強い**のが最大の特徴です。例えば、SQL Server 2014 の時点で、次のような導入効果がありました。

社名/システム概要	システムの特徴	導入効果
bwin 社 オンライン ゲームなどを提供	大量のユーザーによる同時更新。 トランザクションとしては小さい。 毎日 15万人以上のアクティブ ユーザー。毎年 100万人以上の新規ユーザーが増加。 ASP.NET のセッション状態データベースで利用。	約 16.7倍の性能向上 15,000 バッチ要求/sec が 250,000 バッチ要求/sec へ向上。 450,000 バッチ要求/sec も確認 (約 30倍の性能向上)
SBIリクイディティ・マーケット株式会社 FX 取引（オンライントレード）	大量のユーザーによる同時更新。 トランザクションとしては小さい。 顧客のトレーディング データ（取引履歴）をリアルタイムに集計	約 2.5倍の性能向上 52,080 件/sec が 131,921 件/sec へ向上。 ピーク時の Latency（遅延）は、 約 4秒だったのを、1秒以下に短縮
Edgenet 社 商品データを提供するデータプロバイダー	DWH 環境でのステー징 テーブルで利用。ETL 処理など。 バッチ処理での大量のデータ更新。 更新中の参照アクセスあり	約 8～11倍の性能向上 2時間 20分かかっていたバッチ処理が、わずか 20分に短縮。 更新中の参照アクセスが、更新によってブロック（ロック待ちなど）されることがなくなり、読み取り性能も向上
TPP 社 臨床ソフトウェアの提供	大量のユーザーによる同時更新。 トランザクションとしては小さい。 1秒あたり 72,000 ユーザーが同時アクセス（ピーク時）	約 7倍の性能向上 34,700 Transaction/sec が 数十万 Transaction/sec へ向上
弊社のお客様 A社 ポイントカード システム	大量のユーザーによる同時更新。 トランザクションとしては小さい。 ポイントカードにおけるポイントの入金や出金、残高照会処理	約 2.8倍の性能向上

* 弊社執筆の SQL Server 2014 実践シリーズ No.1「インメモリ OLTP の実践的な利用方法」の 1.3 から引用

インメモリ OLTP は、ロック フリー/ラッチ フリーのアーキテクチャなので（ロックとラッチを利用しないアーキテクチャで、ロック待ちやラッチ待ちを回避することができるので）、**大量のユーザーによる同時更新が発生するシステム**（いわゆる OLTP システム）で大きな効果を発揮します。

bwin 社 (bwin.party) では、ASP.NET のセッション状態データベースにインメモリ OLTP を採用して、SQL Server 2014 のときにはラボ環境で **45 万 Batch Requests/sec**（1 秒あたり 45 万件ものバッチ要求を処理）を達成していました。さらに、昨年開催された SQL Server の世界最大規模のイベントである「**PASS Summit 2015**」では、同システムで **66 万 Batch Requests/sec** を達成したというアナウンスもありました（以下のスライドの最後の行に記載）。

Session State - Results

bwin.party digital entertainment

SQL Server 2014 In-Memory OLTP

250,000 batch requests/Sec on a single server - approx 16x gains
Only one server needed for all webfarms (from 18)
 Reduction in cost: hardware/software, power consumption, datacenter space
 Easier to manage
 Less workload for DBA's (only single server to change)
 Less points of failure to troubleshoot

SQL Server 2014 では
45万 Batch Requests/sec

Lab testing called out at PASS 2013

450,000 batch requests/sec 4-socket, 15-core = 60-core total (no HT)

SQL Server 2016 In-Memory OLTP

Testing done in Lab on 4-socket, 18-core (144-core total) system

Results = 660,000 batch requests/second!!!

SQL Server 2016 で
66万 Batch Requests/sec を達成

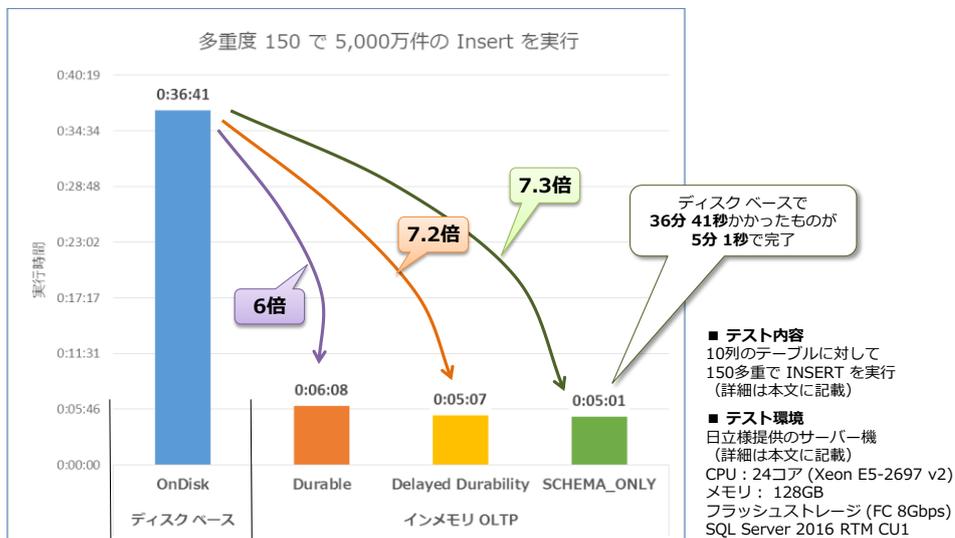
PASS

* PASS Summit 2015 での「SQLCAT:SQL Server 2016 Early Adopter Experiences」セッションのスライドを引用。吹き出しは筆者が追加

1秒あたりに **66万**ということは、1分（60秒）に換算すると 66万*60秒 = **3,960万**ものバッチ要求を処理できるよという数値です。このような驚異的な数値が出せるのは、インメモリ OLTP ならではです。

➡ 検証結果：インメモリ OLTP は、単純 INSERT にも強い

単純 INSERT の性能検証は、SQL Server 2014 の実践シリーズでも行いましたが、今回、日立様のハードウェア（詳細は後述）をお借りすることができたので、SQL Server 2016 でも検証を行ってみました。想定しているシステムは、大規模イベントでの登録や予約、投票のみを受け付けるようなシステムや、IoT などセンサー系データを常に INSERT し続けるようなシステムで、多数のユーザーが同時に INSERT を行う状況を検証しました。結果は、次のとおりです。



* ベンチマークの結果の公表は、使用許諾契約書で禁止されていますが、その効果をわかりやすく表現するため、日本マイクロソフト株式会社の監修のもと、数値を掲載しております。

ディスク ベースの通常テーブルでは **36分41秒**かかった実行時間が、インメモリ OLTP のメモリ最適化テーブルの **Durable** (永続化: SCHEMA_AND_DATA) にした場合は **6分8秒** (6倍の性能向上)、**Delayed Durability** (遅延永続化) にした場合は **5分7秒** (7.2倍の性能向上)、**SCHEMA_ONLY** (永続化なし) にした場合はわずか **5分1秒** で完了し、**7.3倍**もの性能向上を確認することができました。また、このときの **1秒あたりのトランザクション数**を計算すると、次のようになります (実行時間を 5,000 万件で割り算したもの)。

		Transaction/sec
ディスク ベース	OnDisk	22,717
	Durable	135,870
インメモリ OLTP	Delayed Durability	162,870
	SCHEMA_ONLY	166,109

1秒あたり 16.6万件も INSERT できる

SCHEMA_ONLY では、**1秒あたり 16.6万件**も INSERT できることを確認できました。

この検証は、**10個の列**を持ったテーブルを作成して、そのテーブルヘデータを **1件ずつ INSERT** 行って、**5,000万件分**を INSERT したときの実行時間を測定したものです。データには、実際のアプリケーションを想定して、乱数を利用し、同じ値が格納されないようにしています。

テストに使用したテーブル、ネイティブ コンパイル ストアド プロシージャ

```

CREATE TABLE OnDisk_10col_CL
( col1 int IDENTITY(1,1) NOT NULL
  PRIMARY KEY CLUSTERED
, col2 int
, col3 int
, col4 nvarchar(20)
, col5 nvarchar(50)
, col6 nvarchar(100)
, col7 datetime
, col8 datetime
, col9 int
, col10 nchar(1) )
        
```

テーブルには10個の列

従来ながらのディスクベースのテーブル

```

CREATE TABLE InMem_10col_Dura_HASH
( col1 int IDENTITY(1,1) NOT NULL
  PRIMARY KEY NONCLUSTERED
  HASH WITH (BUCKET_COUNT = 100000000)
, col2 int
, col3 int
, col4 nvarchar(20)
, col5 nvarchar(50)
, col6 nvarchar(100)
, col7 datetime
, col8 datetime
, col9 int
, col10 nchar(1) )
WITH ( MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA )

-- Native Compile SP
CREATE PROC spl_Insert_InMem_10col_Dura_HASH
@c2 int, @c3 int, @c4 nvarchar(20), @c5 nvarchar(50), @c6 nvarchar(200)
, @c7 datetime, @c8 datetime, @c9 int, @c10 nchar(1)
WITH NATIVE_COMPILATION, EXECUTE AS OWNER, SCHEMABINDING
AS
BEGIN ATOMIC
WITH ( TRANSACTION ISOLATION LEVEL = SNAPSHOT,
LANGUAGE = N'japanese' )
INSERT INTO dbo.InMem_10col_Dura_HASH
VALUES (@c2, @c3, @c4, @c5, @c6, @c7, @c8, @c9, @c10)
END
GO
        
```

インメモリ OLTP のテーブル

HASH インデックス

データの永続化の設定

1件のデータを INSERT するネイティブ コンパイル ストアド プロシージャ

テスト実行後に格納されるデータ (実際のアプリケーションを想定して、乱数を利用し、同じ値にならないように考慮)

col1	col2	col3	col4	col5	col6	col7	col8	col9	col10
1	4343	226	AAAAA	BBBBBBBBBB	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
2	798	83	AAAAA	BBBBBBBBBB	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
3	8180	267	AA	BBBB	CCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
4	4079	599	AAAAAAAAAAAAAAAAAAAA	BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
5	657	723	AAAAAAAAAAAA	BBBBBBBBBBBBBBBBBBBB	CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
6	3776	690	AAAAAAAAAAAAAAAAAAAA	BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
7	4012	416	AAAAAAAAAAAA	BBBBBBBBBBBBBBBBBBBB	CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
8	4907	38	AAA	BBBBBB	CCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
9	6998	190	AAAAAAAA	BBBBBBBBBBBBBBBBBB	CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
10	973	198	AAAAA	BBBBBBBBBBBB	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
11	6528	142	AAAAAAAAAAAA	BBBBBBBBBBBBBBBBBBBBBB	CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
12	3972	600	AAAA	BBBBBBBB	CCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
13	8509	96	AAAAAAAAAAAA	BBBBBBBBBBBBBBBBBBBBBB	CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
14	4354	732	AAAAAAAAAAAA	BBBBBBBBBBBBBBBBBB	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
15	2427	102	AAAAAAAAAAAA	BBBBBBBBBBBBBBBBBB	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
16	8432	654	AAAAA	BBBBBBBB	CCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
17	9372	8	AAAAAAAAAAAA	BBBBBBBBBBBBBB	CCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
18	3117	114	AAAAAAAAAAAA	BBBBBBBBBBBBBBBBBB	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
19	9282	212	AAAAAAAAAAAA	BBBBBBBBBBBBBBBBBB	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
20	778	241	AAAAAAAAAAAA	BBBBBBBBBBBBBBBBBB	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
21						2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
22						2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
23						2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
24						2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
25						2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0
26						2014-07-08 22:55:04.393	2015-07-08 22:55:04.393	1	0

このテストは、Bulk Insert や SELECT INTO などの一括操作を利用したものではなく、またネイティブ コンパイル ストアド プロシージャ (インメモリ OLTP の性能を向上させることができる ストアド プロシージャ機能) の中でループ処理を記述して、複数件をまとめて INSERT するようなものでもなく、実際のアプリケーションを想定して、1件ずつの INSERT 処理を行った場合の実行時間を測定しています。ネイティブ コンパイル ストアド プロシージャ内の処理も、前述の図のように 1 件の INSERT を行うものしか記述していません。接続に関しても、1 件の INSERT ごとに Open と Close を行っています。

テストで利用したアプリケーション (VB+ADO.NET)

このテストで利用したアプリケーションは、実際のアプリケーションを想定して、次のように .NET (VB+ADO.NET) で作成しています (C#で作成してもほぼ同じコードになります)。

```

89
90 Dim rnd As New System.Random()
91 Dim col2 As Integer = rnd.Next(10000) ' col2 は 0~100000 の乱数
92 Dim col3 As Integer = rnd.Next(1000) ' col3 は 0~10000 の乱数
93
94 Dim col4rnd As Integer = rnd.Next(19) ' col4~col6 用の乱数。0~19
95 Dim col4 As String = New String("A", col4rnd) ' col4 nvarchar(20)
96 Dim col5 As String = New String("B", col4rnd * 2) ' col5 nvarchar(50)
97 Dim col6 As String = New String("C", col4rnd * 5) ' col6 nvarchar(100)
98 Dim col7 As DateTime = DateTime.Now ' col7 は現在の日付
99 Dim col8 As DateTime = DateTime.Now.AddYears(1) ' col8 は1年後
100
101 Using cn As New SqlConnection(cnstr) ' ADO.NET の SqlConnection
102 Using cmd As New SqlCommand() ' ADO.NET の SqlCommand
103     cn.Open()
104     cmd.Connection = cn
105     cmd.CommandText = "INSERT INTO dbo.OnDisk_10col_CL "
106         & "VALUES (@c2, @c3, @c4, @c5, @c6, @c7, @c8, @c9, @c10)" ' SqlParameter を利用
107
108     Dim p1 As SqlParameter = cmd.Parameters.Add("@c2", SqlDbType.Int)
109     p1.Value = col2
110     Dim p2 As SqlParameter = cmd.Parameters.Add("@c3", SqlDbType.Int)
111     p2.Value = col3
112     Dim p3 As SqlParameter = cmd.Parameters.Add("@c4", SqlDbType.NVarChar, 20)
113     p3.Value = col4
114     Dim p4 As SqlParameter = cmd.Parameters.Add("@c5", SqlDbType.NVarChar, 50)
115     p4.Value = col5
116     Dim p5 As SqlParameter = cmd.Parameters.Add("@c6", SqlDbType.NVarChar, 100)
117     p5.Value = col6
118     Dim p6 As SqlParameter = cmd.Parameters.Add("@c7", SqlDbType.DateTime)
119     p6.Value = col7
120     Dim p7 As SqlParameter = cmd.Parameters.Add("@c8", SqlDbType.DateTime)
121     p7.Value = col8
122     Dim p8 As SqlParameter = cmd.Parameters.Add("@c9", SqlDbType.Int)
123     p8.Value = 1
124     Dim p9 As SqlParameter = cmd.Parameters.Add("@c10", SqlDbType.NChar, 1)
125     p9.Value = "0"
126
127     cmd.ExecuteNonQuery()
128     cn.Close()
129 End Using
130 End Using

```

このコードは、ディスク ベースのテーブルへ INSERT を行う場合のものですが、インメモリ OLTP のネイティブ コンパイル ストアド プロシージャを利用する場合には、次のように 2 ヶ所を修正しています。

```

Using cn As New SqlConnection(cnstr)
Using cmd As New SqlCommand()
    cn.Open()
    cmd.Connection = cn
    cmd.CommandText = "INSERT INTO dbo.0nDisk_10col_CL "
    & " VALUES (@c2, @c3, @c4, @c5, @c6, @c7, @c8, @c9, @c10)"

    ' ネイティブ コンパイル ストアド プロシージャの場合
    cmd.CommandType = CommandType.StoredProcedure
    cmd.CommandText = "sp_InMem_10col_Dura_HASH"

Dim p1 As SqlParameter = cmd.Parameters.Add("@c2", SqlDbType.Int)
p1.Value = col2
Dim p2 As SqlParameter = cmd.Parameters.Add("@c3", SqlDbType.Int)
p2.Value = col3
Dim p3 As SqlParameter = cmd.Parameters.Add("@c4", SqlDbType.NVarChar, 20)
p3.Value = col4
Dim p4 As SqlParameter = cmd.Parameters.Add("@c5", SqlDbType.NVarChar, 50)
p4.Value = col5
    
```

SqlCommand の CommandType を変更して、CommandText をネイティブ コンパイル ストアド プロシージャの名前に変更するだけで、ネイティブ コンパイル ストアド プロシージャを実行することができます。

以上のコードを、**多重実行（並行実行）**して、実行時間を計測したのが前掲のグラフです（**150 多重**で実行したときを測定しました）。

このように、インメモリ OLTP は、常に INSERT をし続けるようなシステムでも大きな効果を発揮します。

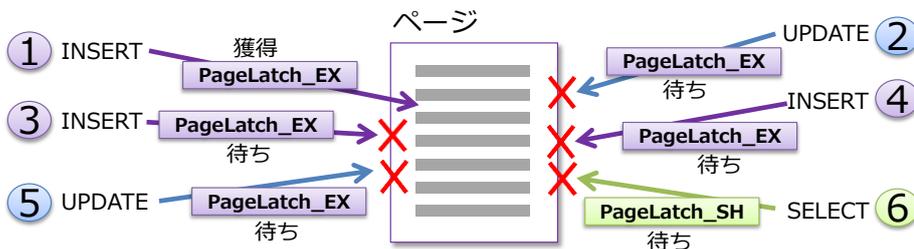
ディスク ベースの通常テーブルで性能が向上しない理由 ～ラッチ待ち～

ディスク ベースの通常テーブルは、多重度が上がれば上がるほど、性能が頭打ちになります（多重度が上がるほど遅くなります）。性能が上がらない一番の理由は、**ラッチ待ち**（Latch Wait）や**ロック待ち**（Lock Wait）などの**ブロッキング**によるものです。これに対して、インメモリ OLTP ではラッチ/ロックを利用しないアーキテクチャなので、ラッチ待ち/ロック待ちに悩まされることはありません（インメモリ OLTP であれば、多重度が上がってもスケールします）。

ディスク ベースで利用される**ラッチ**（Latch）には、主に**ページ ラッチ**（PAGELATCH_SH、PAGELATCH_EX）と**IO ラッチ**（PAGEIOLATCH_SH、PAGEIOLATCH_EX）がありますが、前者は、ページへの同時アクセスを制御するための、SQL Server が内部的にページに対してかけるロックのようなもの、後者は、データ ファイル (.mdf) からメモリ内のデータ バッファへページを取り出すとき/書き出すときにかかるロックのようなものです。

ページ ラッチ（PAGELATCH_SH や PAGELATCH_EX）は、次の図のように、同じページに対して、多数のユーザーからの同時アクセスが発生した場合を制御するためのもので、同時に同じページを操作させないように、後からきたラッチを**"待ち"**にします（ページ ラッチ待ちの発生）。

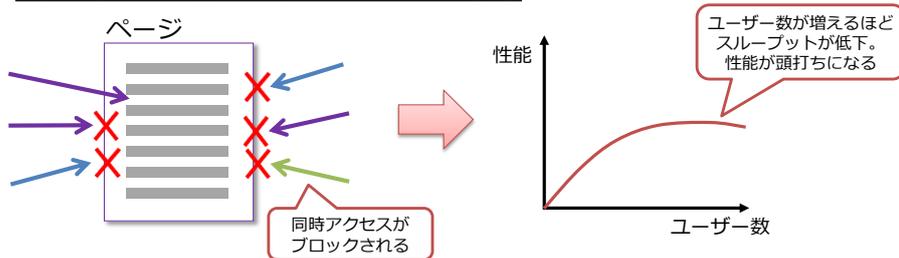
同じページへの同時アクセスは、ページ ラッチ待ちが発生する



更新系のステートメント (INSERT/UPDATE/DELETE) では、**PAGELATCH_EX** (排他ページラッチ。EX は Exclusive : 排他の略) をかけにいき、SELECT ステートメントによる参照時には **PAGELATCH_SH** (共有ページラッチ。SH は Shared : 共有の略) をかけにいります。このとき、先にアクセスしている処理がある場合は (ラッチが既にかかっている場合には)、それが解放されるまで、"待ち" が発生します。このような、同時アクセスによって待ちが発生する状態は、**ラッチ競合 (Latch Contention)** とも呼ばれています。

このように、ラッチ待ちが発生すると、ユーザー数が増えれば増えるほど、ラッチ待ちも増えることになるので、スループットが低下していきます。

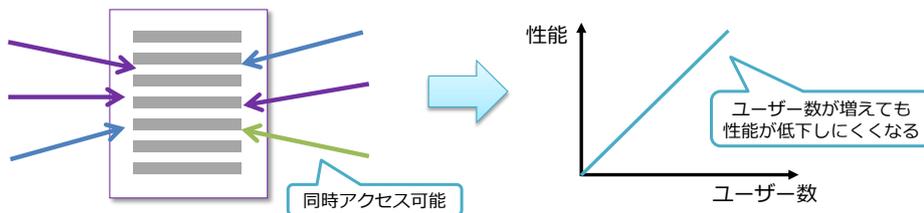
ラッチ待ちが発生するとスループットが低下



これでは、同時実行数が増えれば増えるほど、システムの処理能力は頭打ちになり、スケールしなくなってしまう。

これに対して、インメモリ OLTP が採用している、ラッチを利用しない「**ラッチ フリー**」のアーキテクチャであれば、ユーザー数が増えてたとしても性能低下は発生しにくくなります。

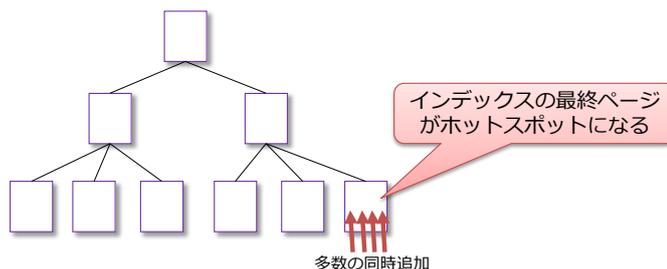
ラッチ フリーならユーザー数が増えても性能低下を抑えられる



ラッチ待ちの様子 ~インデックスの最終ページがホットスポット~

今回の検証で利用したテーブルは、**col1** 列を **IDENTITY(1, 1)** の **PRIMARY KEY** に設定しているので、ディスク ベースのテーブルでは、多数のユーザーが同時にデータを追加することによって、次のように インデックスの最終ページ にアクセスが集中します。

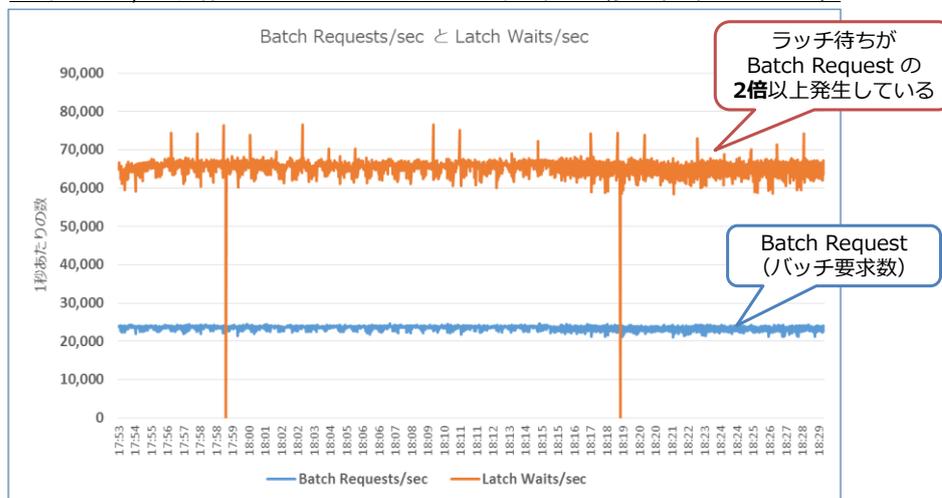
インデックスの最終ページでページ ラッチ待ちが多発する



ディスク ベースのテーブルの場合は、PRIMARY KEY 制約を作成することで、自動的に**クラスタ化インデックス** (b-tree) が作成されるので、データが追加されると、最終ページに値が集中することになります。**連番系**の列 (IDENTITY を設定した列や、シーケンスを設定した列) など、データを追加するたびに連続した値 (1、2、3、…) が格納されていくような場合には、このようなインデックスの最終ページでページ ラッチ待ちが多発する (最終ページがホット スポットになる) ことがよくあります。これは、シングル実行 (1 人のユーザーによる単体実行) では、発生しないものですが、多数のユーザーが同時にデータを追加する場合には発生してしまいます。

ラッチ待ちが発生しているかどうかは、パフォーマンス カウンターの **Latch Waits/sec** (**SQLServer: Latches** オブジェクト) を参照することで簡単に確認することができます。実際に、検証を行ったときのパフォーマンス カウンターは、次のようになりました。

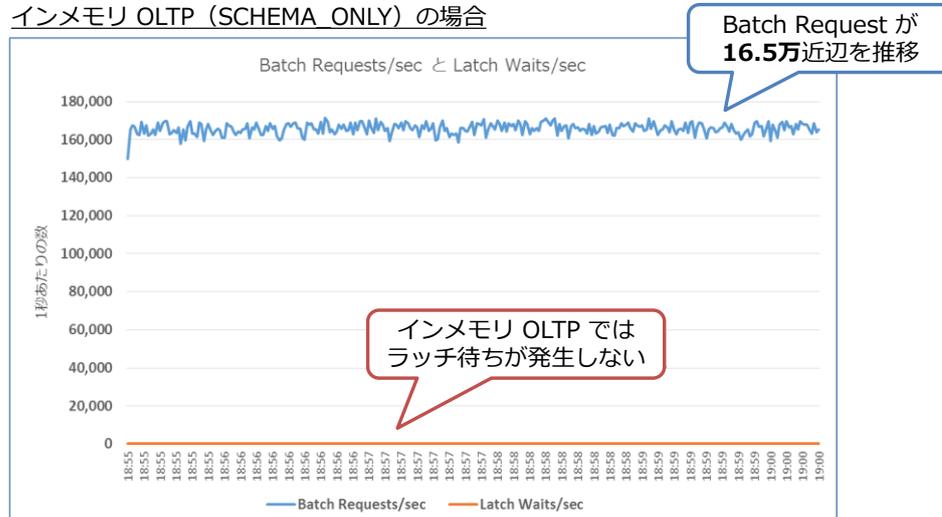
150多重で 5,000万件を INSERT しているときのラッチ待ちの様子 (ディスクベース)



Batch Request (バッチ要求数) が **2.3 万**ぐらいを推移しているのに対して、**ラッチ待ち (Latch Waits)** が **6.5 万** (2 倍以上) も発生してしまっています。このようなラッチ待ちは、多重度が 200、300 と上がっていくとさらに顕著に表れて、性能がどんどん低下していくこととなります。

一方、インメモリ OLTP ではラッチ待ちは発生しないので、次のような性能が出ます。

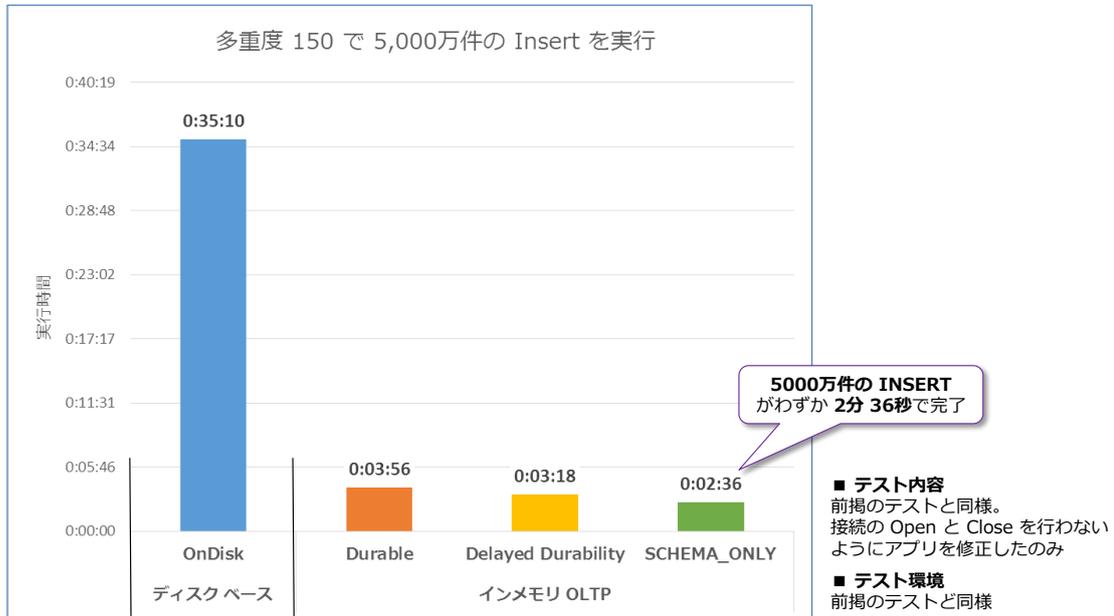
インメモリ OLTP (SCHEMA_ONLY) の場合



このように、インメモリ OLTP は、大量ユーザーからの INSERT/常に INSERT をし続けるよう

なシステムでも大きな効果を発揮します。

なお、今回の検証では、実際のアプリケーションを想定していたので、1 件の INSERT ごとに接続の Open と Close 処理を行っていましたが、もし接続をキープできるような状況（接続の Open と Close を行わずに同じ接続を再利用する形）であれば、次のような性能を出すことができます。



		Transaction/sec
ディスク ベース	OnDisk	23,697
	Durable	211,862
インメモリ OLTP	Delayed Durability	252,525
	SCHEMA_ONLY	320,521

1秒あたり 32万件も INSERT できる

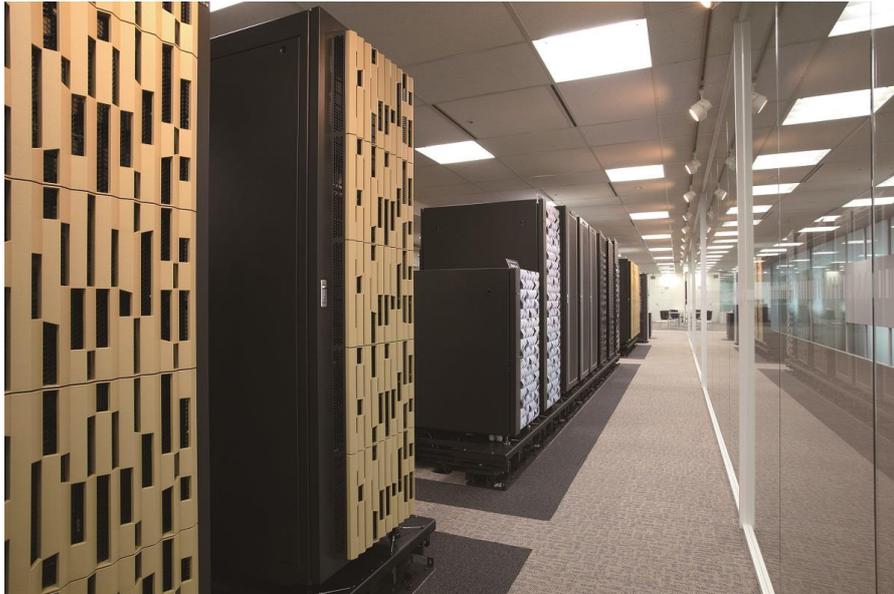
ディスク ベースでは、接続をキープしても、ラッチ待ちがボトルネックになっているので、性能はほとんど変わりません。これに対して、インメモリ OLTP の SCHEMA_ONLY では、**1 秒あたり**に **32 万件ものトランザクション**を処理できるようになっています。

2.3 Operational Analytics の検証環境

前掲の Operational Analytics の検証では、株式会社 日立製作所様のご厚意でハードウェアをお借りすることができましたので、ここで環境をご紹介します。

今回利用させていただいた日立様の施設は、品川にある「ハーモニアス・コンピテンス・センター」です。

ハーモニアス・コンピテンス・センターのマシンの様子



検証で利用させていただいたハードウェアは、「**統合サービスプラットフォーム BladeSymphony BS500**」の 24 コア マシン (Xeon E5-2697v2 を 2 個搭載したサーバー) とエンタープライズ ディスク アレイ システムである「**Hitachi Virtual Storage Platform G1000**」です。

検証で利用させていただいたハードウェア



サーバー

BladeSymphony BS500
 ブレード： BS520H サーバブレード
 CPU: Xeon E5-2697v2 × 2 (24core)
 メモリ： 128G
 内蔵HDD： 600GB×2
 ストレージ接続： 8Gbps FC



ストレージ

Hitachi Virtual Storage Platform G1000
 1.6TBフラッシュメモリーユニット×9
 (RAID5:3D+1P (4915.19GB) ,
 イミレーション:OPEN-V×2、SASディスク×1

2.4 Operational Analytics を試す方法

ここでは、インメモリ OLTP とクラスター化列ストア インデックスを利用した Operational Analytics を試す方法を説明します。

➡ インメモリ OLTP の基本的な利用方法

まずは、インメモリ OLTP の基本的な利用方法を説明します。インメモリ OLTP では、インメモリ化したテーブルのことを「メモリ最適化テーブル」と呼び、次のように作成することができます。

```

-- データベースの作成
CREATE DATABASE testDB
ON PRIMARY (
    NAME = testDB_data,
    FILENAME = 'C:\%testDB%testDB_data.mdf',
    SIZE = 10GB ),
FILEGROUP fg1 CONTAINS MEMORY_OPTIMIZED_DATA
( NAME = testDB_InMem,
  FILENAME = 'C:\%testDB%testDB_InMem' )
LOG ON
( NAME = testDB_log,
  FILENAME = 'C:\%testDB%testDB_log.ldf',
  SIZE = 10GB )

-- メモリ最適化テーブルの作成
USE testDB
CREATE TABLE t1_InMem
( col1 int NOT NULL
  PRIMARY KEY NONCLUSTERED
  HASH WITH (BUCKET_COUNT = 2000000)
, col2 nvarchar(100) )
WITH ( MEMORY_OPTIMIZED = ON,
      DURABILITY = SCHEMA_AND_DATA )

```

メモリ最適化テーブルを作成するには **CONTAINS MEMORY_OPTIMIZED_DATA** を指定したファイル グループを追加しておく

メモリ最適化テーブルでは **HASH インデックス**が基本になる

MEMORY_OPTIMIZED=ON を指定するとメモリ最適化テーブルになる

DURABILITY=SCHEMA_AND_DATA でデータを永続化できる

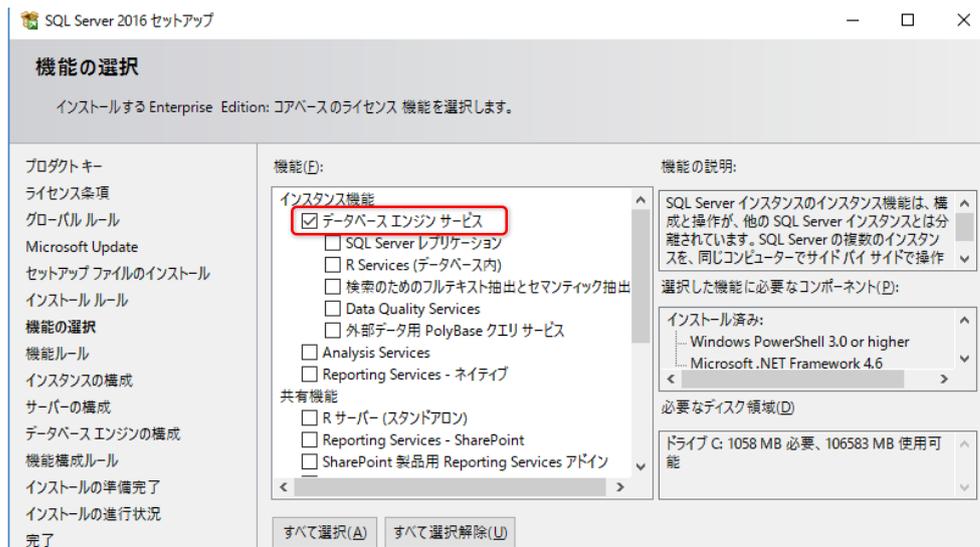
100 %
メッセージ
コマンドは正常に完了しました。

このようにメモリ最適化テーブルを作成するには、次の作業が必要になります。

- 通常の SQL Server と同様、SQL Server のインストール時に [**データベース エンジン サービス**] をインストールしておく
- **データベースの互換性レベル**を **130** にする (これは必須ではありませんが、インメモリ OLTP で**並列プラン**を利用するために必要になるので、**130** にしておくことを強く推奨)
- メモリ最適化テーブルを格納するための**ファイル グループ**の作成 (CONTAINS MEMORY_OPTIMIZED_DATA を指定したファイル グループをデータベースに追加する)
- **メモリ最適化テーブル**の作成 (CREATE TABLE ステートメントで **MEMORY_OPTIMIZED = ON** を指定)

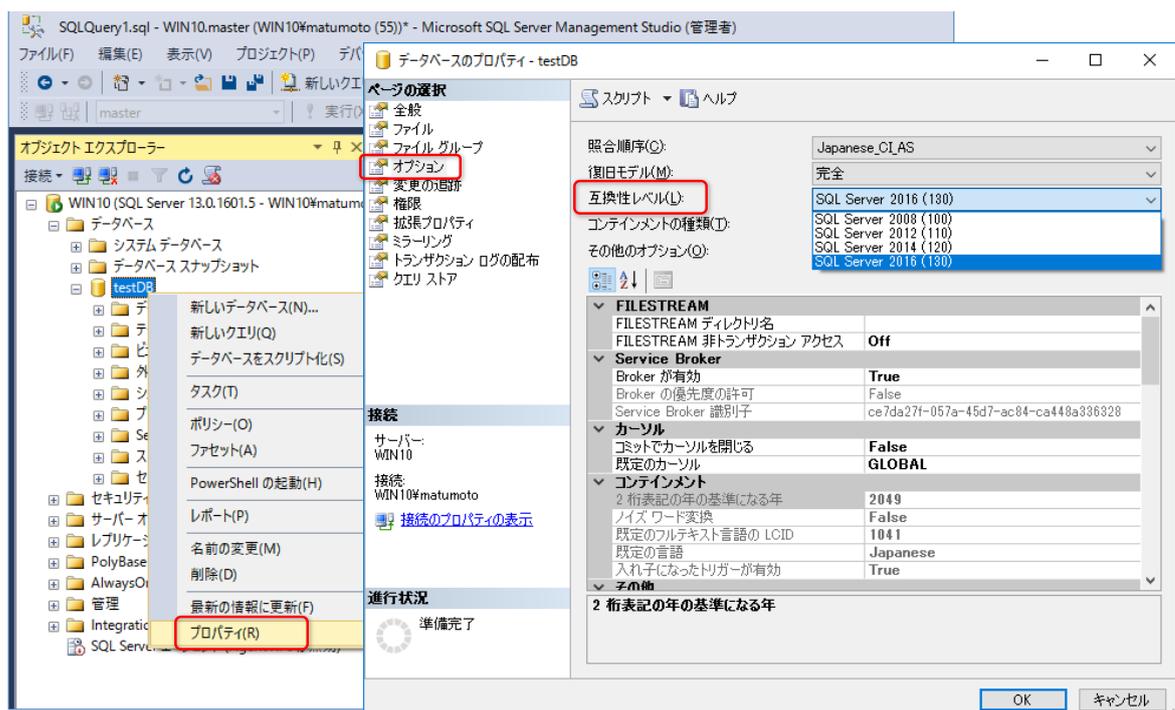
➡ 通常と同様「データベース エンジン サービス」のインストール

インメモリ OLTP は、SQL Server のデータベース エンジンに完全に統合されているので、通常どおりに SQL Server をインストールするだけで利用できます。インストール時には、次のように [機能の選択] ページで、通常どおり「データベース エンジン サービス」を選択しておけば、インメモリ OLTP を利用することができます。



➡ データベースの互換性レベルを「130」へ設定

データベースの互換性レベルを確認/設定するには、次のようにデータベースのプロパティの [オプション] ページを利用します。



SQL Server 2016 では、データベースを新しく作成したときの既定の互換性レベルは「**130**」になります（**130** は、SQL Server 2016 の内部バージョン番号である **13.0** という意味です）。

130 レベルでは、**インメモリ OLTP** での**並列プラン**や、**INSERT..SELECT** での**マルチ スレッド、列ストア インデックスでのバッチ モードの動作の違い**（MAXDOP 1 でもバッチ モードで動作する）など、性能に関する大きな違いが出るので、130 レベルを利用することを強くお勧めします。SQL Server 2014 や 2012 など、古いバージョンの SQL Server で取得したバックアップを SQL Server 2016 上にリストアしたり、古いバージョンのデータベース ファイル(.mdf/.ldf)を SQL Server 2016 上にアタッチした場合には、そのバージョンの互換性レベルが保たれるので、130 レベルに上げて問題ないか、検証してみることをお勧めします。

➔ ファイル グループの作成

メモリ最適化テーブルを格納するためのファイル グループを作成するには、**CREATE DATABASE** ステートメントで、次のように **FILEGROUP** 句で **CONTAINS MEMORY_OPTIMIZED_DATA** を指定します。

```
-- インメモリ OLTP を利用するためのファイル グループを作成する例
CREATE DATABASE testDB
ON PRIMARY (
    NAME = testDB_data,
    FILENAME = 'C:%testDB%testDB_data.mdf',
    SIZE = 10GB ),
FILEGROUP fg1 CONTAINS MEMORY_OPTIMIZED_DATA
( NAME = testDB_InMem,
  FILENAME = 'C:%testDB%testDB_InMem' )
LOG ON
( NAME = testDB_log,
  FILENAME = 'C:%testDB%testDB_log.ldf',
  SIZE = 10GB )
```

```
-- データベースの作成
CREATE DATABASE testDB
ON PRIMARY (
    NAME = testDB_data,
    FILENAME = 'C:%testDB%testDB_data.mdf',
    SIZE = 10GB ),
FILEGROUP fg1 CONTAINS MEMORY_OPTIMIZED_DATA
( NAME = testDB_InMem,
  FILENAME = 'C:%testDB%testDB_InMem' )
LOG ON
( NAME = testDB_log,
  FILENAME = 'C:%testDB%testDB_log.ldf',
  SIZE = 10GB )
```

メモリ最適化テーブルを作成するには
CONTAINS MEMORY_OPTIMIZED_DATA
を指定したファイル グループを追加しておく

FILENAME は
ファイルを作成する場所

メッセージ
コマンドは正常に完了しました。

FILEGROUP に続けてファイル グループ名を指定して（ここでは **fg1** という名前を指定）、**「CONTAINS MEMORY_OPTIMIZED_DATA」**と記述することで、メモリ最適化テーブルを格

納できるファイル グループを作成することができます。**FILENAME** には作成先となるファイルパスを指定しますが、ここで指定するファイル名は、**NAME** で指定する論理ファイル名（上記では **oaTestDB_InMem**）と同じ名前にするようにします。

なお、**CREATE DATABASE** ステートメントでの新規データベースの作成時ではなく、既存のデータベースに対して、後からファイル グループを追加したい場合には、次のように **ALTER DATABASE** ステートメントを 2 つ実行します。

```
-- 既存のデータベースにファイル グループを追加する例
ALTER DATABASE testDB
  ADD FILEGROUP fg1
  CONTAINS MEMORY_OPTIMIZED_DATA

ALTER DATABASE testDB
  ADD FILE ( NAME = testDB_InMem,
            FILENAME = 'C:¥testDB¥testDB_InMem' )
  TO FILEGROUP fg1
```

1 つ目の **ALTER DATABASE** ステートメントでは、**ADD FILEGROUP** に続けてファイル グループ名を指定して、「**CONTAINS MEMORY_OPTIMIZED_DATA**」を記述します。2 つ目では、**ADD FILE** で、論理ファイル名とファイルの作成場所を指定し、**TO FILEGROUP** で 1 つ目で作成したファイル グループを指定します。

➔ メモリ最適化テーブルの作成

ファイル グループを作成した後は、次のようにメモリ最適化テーブルを作成することができます。

```
-- メモリ最適化テーブルを作成する例
CREATE TABLE t1_InMem
( col1 int NOT NULL
  PRIMARY KEY NONCLUSTERED
  HASH WITH (BUCKET_COUNT = 2000000)
, col2 nvarchar(100) )
WITH ( MEMORY_OPTIMIZED = ON,
      DURABILITY = SCHEMA_AND_DATA )
```

The screenshot shows a SQL query window with the following code:

```
-- メモリ最適化テーブルの作成
USE testDB
CREATE TABLE t1_InMem
( col1 int NOT NULL
  PRIMARY KEY NONCLUSTERED
  HASH WITH (BUCKET_COUNT = 2000000)
, col2 nvarchar(100) )
WITH ( MEMORY_OPTIMIZED = ON,
      DURABILITY = SCHEMA_AND_DATA )
```

Callouts from the screenshot explain the options:

- MEMORY_OPTIMIZED=ON** を指定するとメモリ最適化テーブルになる
- DURABILITY=SCHEMA_AND_DATA** でデータを永続化できる
- SCHEMA_AND_DATA** では **PRIMARY KEY** が必須になる
- メモリ最適化テーブルでは **NONCLUSTERED** なインデックスが必須
- メモリ最適化テーブルでは **HASH** インデックスが基本になる

At the bottom, a message box says: "メッセージ: コマンドは正常に完了しました。"

メモリ最適化テーブルは、**CREATE TABLE** ステートメントの末尾の **WITH** 句で「**MEMORY_**

OPTIMIZED = ON」を追加することで作成することができます。「**DURABILITY = SCHEMA_AND_DATA**」の部分は、データを保存／永続化するかどうか（Durable かどうか）を指定するところになりますが、**SCHEMA_AND_DATA** を指定すると永続化、**SCHEMA_ONLY** を指定すると永続化なしに設定することができます。永続化なし（**SCHEMA_ONLY**）の場合は、SQL Server を再起動した場合には「**データが空**」になり、永続化有り（**SCHEMA_AND_DATA**）の場合は、ディスク ベースの通常テーブルと同様、SQL Server を再起動してもデータが消えることはありません（∵トランザクション ログに更新履歴を記録しているため）。

col1 列には、**NONCLUSTERED**（非クラスター化）な **HASH インデックス**を作成していますが、メモリ最適化テーブルでは、**NONCLUSTERED** のインデックスを作成するのが必須になるので、このように指定しています。また、**SCHEMA_AND_DATA**（永続化有り）の場合には、**PRIMARY KEY** 制約も必須になるので、これを設定しています（こうしたメモリ最適化テーブルを作成するための条件については、後述します）。

なお、HASH インデックスを作成しなくても、次のようにメモリ最適化テーブルを作成することができますが、この場合は **bw-tree インデックス**という種類のインデックスが作成されます。**bw-tree インデックス**は、通常のディスク ベースのインデックスである「**b-tree インデックス**」のインメモリ版です。

```
CREATE TABLE t1_InMem
( col1 int NOT NULL
  PRIMARY KEY NONCLUSTERED
, col2 nvarchar(100) )
WITH ( MEMORY_OPTIMIZED = ON,
      DURABILITY = SCHEMA_AND_DATA )
```

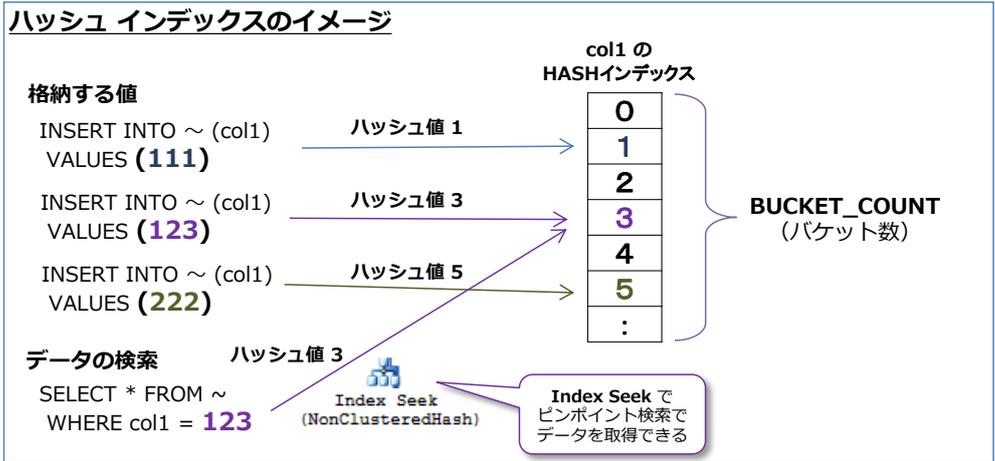
HASH インデックスを指定しない場合は
bw-tree インデックスが作成される
(b-tree インデックスのインメモリ版)

100 %
メッセージ
コマンドは正常に完了しました。

この bw-tree インデックスは、HASH インデックスよりも範囲スキャン（Range Scan）に強いというメリットがありますが、更新性能（INSERT／UPDATE／DELETE の性能）は HASH インデックスよりも劣るというデメリットがあるので、まずは HASH インデックスを利用することをお勧めします（もちろん、更新操作よりも、範囲スキャンのようがより重要なクエリになるという場合には、bw-tree インデックスを作成することを検討してみてください）。

➡ メモリ最適化テーブルでは HASH インデックスが基本、BUCKET_COUNT

HASH インデックスは、メモリ最適化テーブルでは基本となるインデックスで、次のようなイメージのものです。



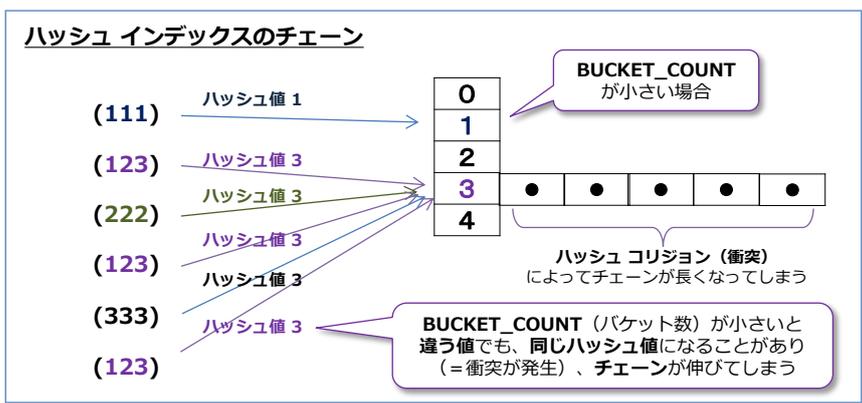
格納する値に対して、内部的にハッシュ値が計算されてハッシュ テーブルに格納され、これをもとに **Index Seek** でピンポイント検索ができるようになっています。ハッシュ テーブルの大きさ（何個のハッシュ値を格納できるようにするのか）は、**BUCKET_COUNT** (バケット数) で指定します。前の例では「**HASH WITH (BUCKET_COUNT=2000000)**」と指定して、200 万個のバケットを作成しました。

BUCKET_COUNT は、SQL Server 2014 ではテーブルを作成した後に変更することができませんでしたが（∵ **ALTER TABLE** ステートメントがサポートされなかったため）、SQL Server 2016 からは **ALTER TABLE** ステートメントがサポートされるようになったので、後から変更できるようになりました。これは、次のように実行することができます。

```
-- BUCKET_COUNT を後から変更する場合 (SQL Server 2016)
ALTER TABLE t1_InMem
ALTER INDEX インデックス名
REBUILD WITH ( BUCKET_COUNT = 50000000 )
```

PRIMARY KEY に対して設定した **HASH** インデックスの **BUCKET_COUNT** は、実際のデータ件数～データ件数の 2 倍ぐらいの大きさに設定するのがお勧めです。例えば、データ件数が 5,000 万件になる予定なら、**BUCKET_COUNT** は 5,000 万～1 億ぐらいに設定するようにします。

もし、**BUCKET_COUNT** を実際のデータ件数よりも小さい値に設定している場合には、**ハッシュ コリジョン**（ハッシュ値の衝突）が発生して、次のようにチェーンが長くなってしまいます。



チェーンが長くなってしまうと性能に影響が出るので (Index Seek が遅くなったり、更新性能が低下するので)、BUCKET_COUNT は十分な大きさに設定しておくことがお勧めになります。

➔ BUCKET_COUNT の違いによるメモリ使用量の差

BUCKET_COUNT で指定したバケット数は、内部的には指定した値に最も近い **2 のべき乗** (切り上げ) に設定されます。例えば、BUCKET_COUNT を 1 億に指定した場合は、2 の 27 乗 (134,217,728) に設定されます。したがって、代表的な **2 のべき乗** を知っておいた方が設定がしやすくなるので、次の表が参考になると思います。

代表的な 2 のべき乗と内部的なメモリ使用量

2のべき乗	バケット数	備考	使用メモリ (KB) 1バケット 8バイト
17	131,072	約 13万	1,024
18	262,144	約 26万	2,048
19	524,288	約 50万	4,096
20	1,048,576	約 100万	8,192
21	2,097,152	約 210万	16,384
22	4,194,304	約 420万	32,768
23	8,388,608	約 840万	65,536
24	16,777,216	約 1,680万	131,072
25	33,554,432	約 3,360万	262,144
26	67,108,864	約 6,700万	524,288
27	134,217,728	約 1.3億	1,048,576
28	268,435,456	約 2.7億	2,097,152
29	536,870,912	約 5.4億	4,194,304
30	1,073,741,824	約 10.7億	8,388,608
31	2,147,483,648	約 21.5億	16,777,216
32	4,294,967,296	約 43億	33,554,432

BUCKET_COUNT を 100万に設定すると 8MB を消費

400万に設定すると 32MB を消費

1,000万に設定すると 130MB を消費

1億に設定すると 1GB を消費

また、1つのバケットあたりのメモリ使用量は、**8バイト**になるので、BUCKET_COUNT を 100 万に設定した場合は 8MB、1 億に設定した場合は 1GB のメモリを消費する形になります。

実際にインメモリ OLTP がどれぐらいのメモリを消費しているかは、次のように **dm_db_xtp_table_memory_stats** 動的管理ビューを利用して確認することもできます。

```
SELECT OBJECT_NAME(object_id) AS テーブル名, *
FROM sys.dm_db_xtp_table_memory_stats
```

テーブル名	object_id	memory_allocated_for_table_kb	memory_used_by_table_kb	memory_allocated_for_indexes_kb	memory_used_by_indexes_kb
InMem_BC_10000	693577509	626240	625000	1048704	1048704
InMem_BC_100000	725577623	626240	625000	1049600	1049600
InMem_BC_1000000	757577737	626240	625000	1056768	1056768
InMem_BC_4000000	789577851	626240	625000	1081344	1081344

memory_allocated_for_indexes_kb が、インデックスに割り当てられたメモリ量で、この値がバケット数をもとに決定されています。

➔ メモリ最適化テーブルにクラスター化列ストア インデックスを追加

メモリ最適化テーブルに**クラスター化列ストア インデックス** (Clustered Column-store Index) を追加して、**Operational Analytics** を実現するには、次のようにテーブルを作成します。

```
-- メモリ最適化テーブルにクラスター化列ストア インデックスを追加
CREATE TABLE InMem_ccsi
( col1 int IDENTITY(1,1) NOT NULL
  PRIMARY KEY NONCLUSTERED
  HASH WITH (BUCKET_COUNT = 100000000)
, col2 int NOT NULL
, col3 int NOT NULL
, col4 int NOT NULL
, col5 datetime NOT NULL
, INDEX idx_InMem_ccsi CLUSTERED COLUMNSTORE
) WITH ( MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA )
```

```
-- メモリ最適化テーブルにクラスター化列ストア インデックスを追加
CREATE TABLE InMem_ccsi
( col1 int IDENTITY(1,1) NOT NULL
  PRIMARY KEY NONCLUSTERED
  HASH WITH (BUCKET_COUNT = 100000000)
, col2 int NOT NULL
, col3 int NOT NULL
, col4 int NOT NULL
, col5 datetime NOT NULL
, INDEX idx_InMem_ccsi CLUSTERED COLUMNSTORE
) WITH ( MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA )
```

100 %

メッセージ
コマンドは正常に完了しました。

INDEX に続けてインデックス名（上の例では **idx_InMem_ccsi**）を指定して、**CLUSTERED COLUMNSTORE** と指定すれば、クラスター化列ストア インデックスを作成することができます。クラスター化列ストア インデックスでは、列を指定する必要がないので（テーブルに 1 つしか作成することができないので）、これだけの記述で作成が完了です。

なお、メモリ最適化テーブルにクラスター化列ストア インデックスを作成するには、**DURABILITY**（永続化するかどうか）で **SCHEMA_AND_DATA**（永続化有り）を指定しておく必要があります。

また、SQL Server 2016 では、後からクラスター化列ストア インデックスを追加することはできないので（ALTER TABLE/INDEX でのクラスター化列ストア インデックスの追加がサポートされていないので）、**CREATE TABLE** ステートメントでの作成時に指定する必要があります。

➡ メモリ最適化テーブルを利用する条件

メモリ最適化テーブルを利用するための条件や利用にあたってのポイントは、次のとおりです。

- データベースの互換性レベルを **130** にしておくことでパラレル処理が可能
- メモリ最適化テーブルに**クラスター化列ストア インデックス**を追加する場合は、**SCHEMA_AND_DATA** (永続化) が必須になる。
SCHEMA_AND_DATA (永続化) を利用する場合は、**PRIMARY KEY 制約**も必須
- メモリ最適化テーブルには**非クラスター化列ストア インデックス**を追加できない
- メモリ最適化テーブルでは、1つ以上の**非クラスター化 (NONCLUSTERED)** の **HASH** または **bw-tree インデックス**を作成する必要がある。
SCHEMA_AND_DATA (永続化) の場合は、**PRIMARY KEY 制約**も必須になる。
SCHEMA_ONLY (永続化なし) の場合は、**PRIMARY KEY 制約**は必須ではないが、**インデックス**が必須になる。
- **HASH** または **bw-tree** インデックスは、1つのテーブルに**最大8個**まで作成可能。
なお、SQL Server 2014 のときには、インデックスを作成する列に **NOT NULL** が必須だったが、SQL Server 2016 からは必須ではなくなった。
- **HASH** インデックスを利用する場合は、適切な **BUCKET_COUNT** (バケット数) を設定しておくことが重要 (SQL Server 2016 からは後から **BUCKET_COUNT** を変更することも可能/**ALTER TABLE** で **REBUILD** が可能になった)
- **IDENTITY** は (1, 1) のみがサポートされる (**SEQUENCE** を利用することも可能)
- 利用できないデータ型には、**xml**、**sql_variant**、**datetimeoffset**、**hierarchyid**、**geography**、**geometry**、**rowversion**、**UDT** (ユーザー定義データ型) がある

その他、メモリ最適化テーブルを利用するにあたっての細かい制限事項としては、次のようなものがあります。

- **TRUNCATE TABLE** ステートメントがサポートされない (**DELETE** ステートメントでデータを削除する必要がある)
- **SELECT INTO** ステートメントでのターゲット テーブルには指定することができない
- **MERGE** ステートメントでのターゲット テーブルには指定することができない
- **CLR アクセス**がサポートされない
- データベースをまたがったクエリ/トランザクションがサポートされない
- **READ COMMITTED 分離レベル** (SQL Server の既定の分離レベル) は、**自動コミットトランザクション**のみでサポートされる
- トランザクション内でサポートされる分離レベルは、**SNAPSHOT** または **REPEATABLE READ**、**SERIALIZABLE** のみで、**WITH** 句でのテーブル ヒントとして指定、または **ALTER DATABASE** で **ELEVATE_TO_SNAPSHOT** を有効化しておく

ようにする。ネイティブ コンパイル ストアド プロシージャを利用している場合には **BEGIN ATOMIC** 句の **ISOLATION LEVEL** で分離レベルを指定する

➡ SQL Server 2014 との違い (多くの制限が取り払われた)

SQL Server 2014 のときには、インメモリ OLTP の最初のバージョンであったこともあり、メモリ最適化テーブルを利用するにあたって多くの制限がありました。SQL Server 2016 からはそういった制限の多くが取り払われました。その主なものは、次のとおりです。

- **ALTER TABLE のサポート**

後から列を追加／削除したり、データ型を変更したり、テーブル変更が可能。SQL Server 2014 では、後からインデックスを作成することができなかった。

- **後からインデックスを追加可能に**

SQL Server 2014 では、後からインデックスを作成することができなかった。

- **インデックスでの NULL のサポート**

SQL Server 2014 では、インデックスを作成するには NOT NULL が必須だった。

- **BIN2 以外の照合順序のサポート**

SQL Server 2014 では、インデックスを作成する列が char 系の場合に、BIN2 照合順序が必須だった。

- **varchar/char でのコードページ 1252 以外のサポート**

SQL Server 2014 では、varchar/char では、コードページ 1252 が必須だった。その他のコードページを利用するには nvarchar/nchar など n 付きのデータ型を利用するのが必須だった。

- **LOB (ラージ オブジェクト) のサポート**

SQL Server 2014 では、行サイズの上限が 8060 バイトだった。

- **CHECK 制約や FOREIGN KEY 制約のサポート**

SQL Server 2014 では、CHECK 制約と FOREIGN KEY 制約を利用できなかった。

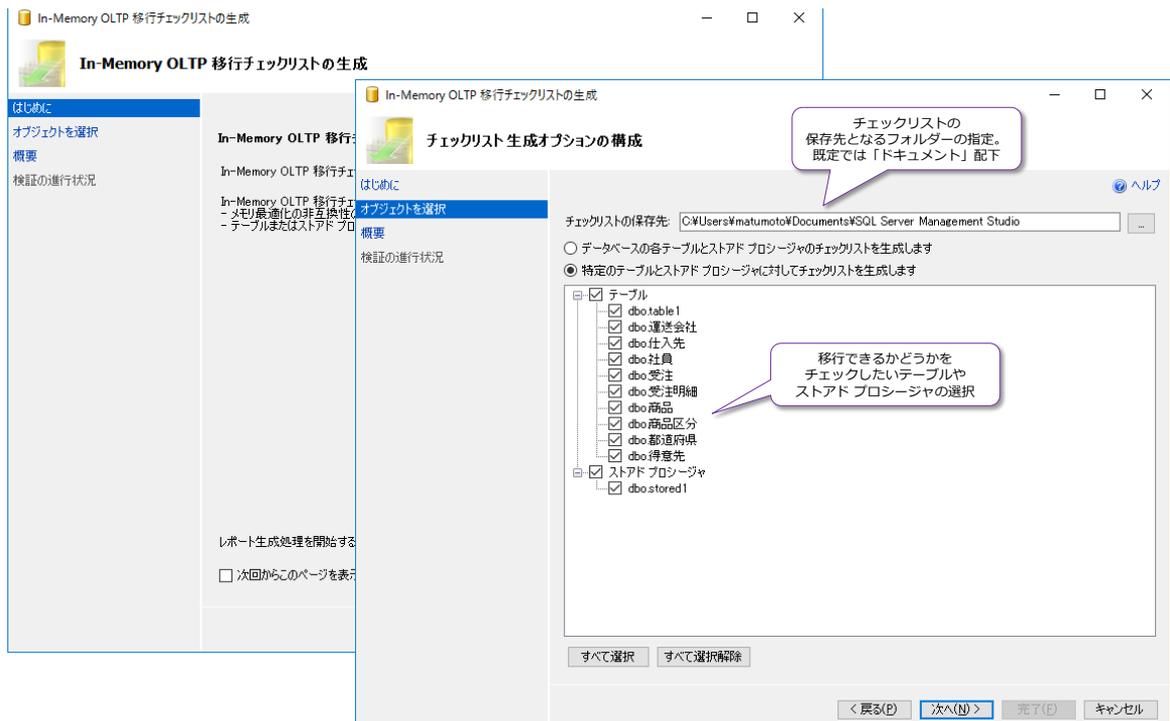
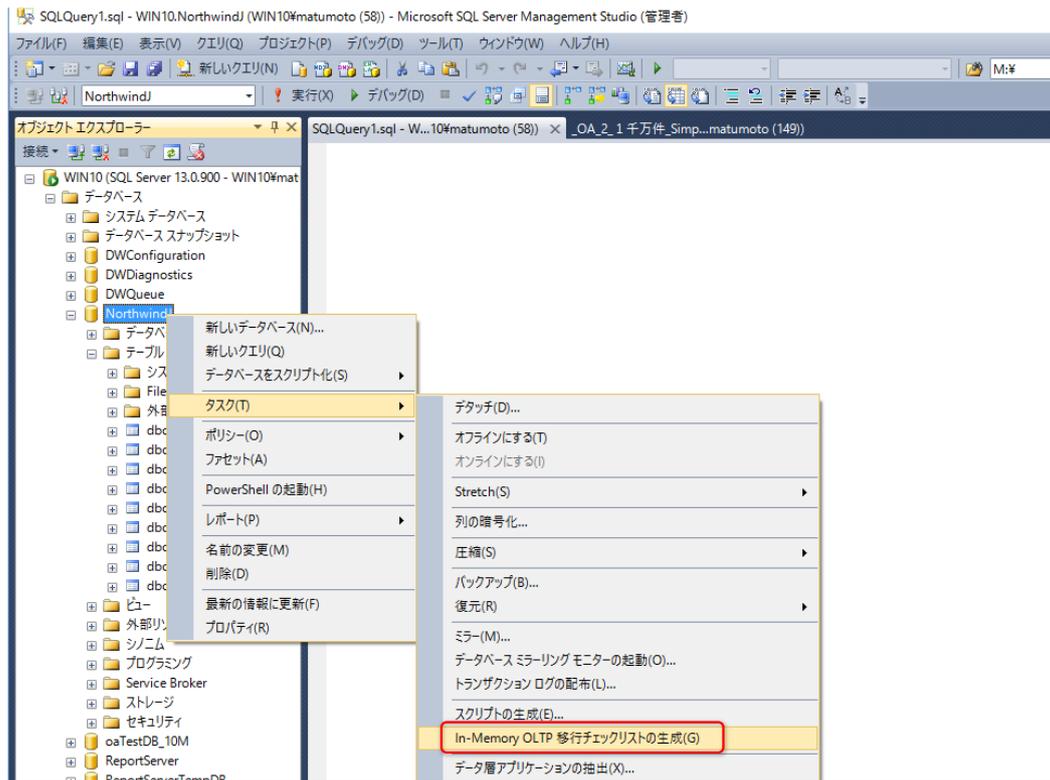
その他の SQL Server 2014 との違いについては、オンライン ブックの以下のトピックがお勧めです。

データベース エンジンの新機能

<http://msdn.microsoft.com/ja-jp/library/bb510411.aspx>

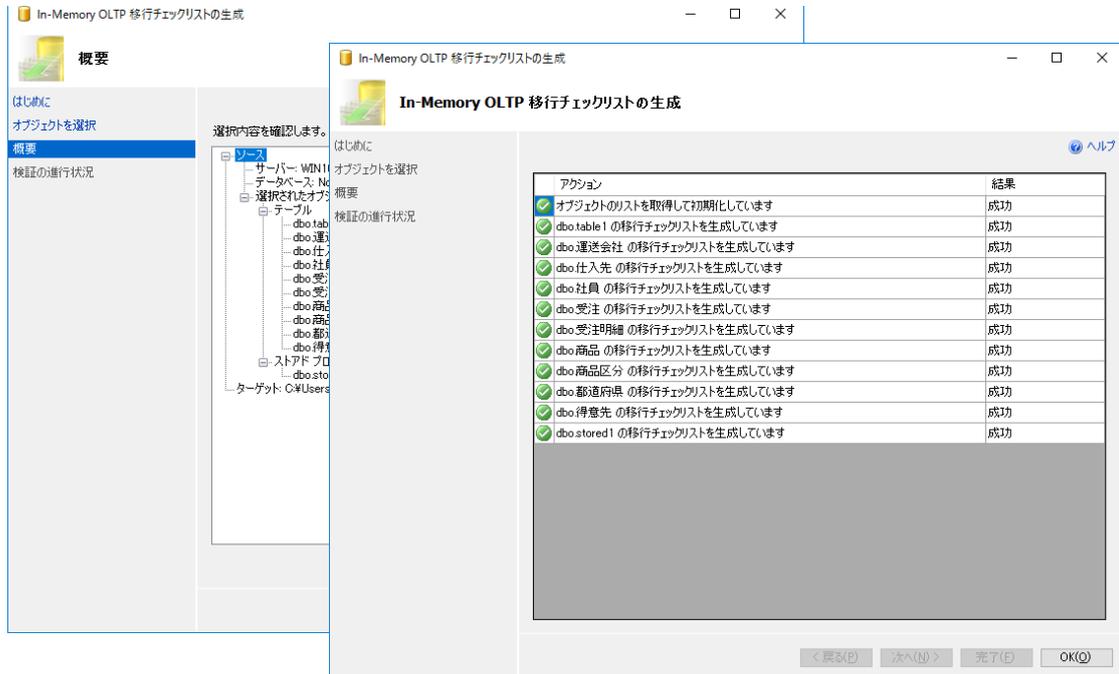
➔ メモリ最適化テーブルへの移行

SQL Server 2016 からは、ディスク ベースのテーブルを、メモリ最適化テーブルに移行するにあたって、移行可能かどうかをチェックしてくれる「**In-Memory OLTP 移行チェックリスト**」機能が提供されました。これは、次のようにデータベースを右クリックして、[タスク] メニューの [In-Memory OLTP 移行チェックリストの生成] をクリックすることで起動することができます。

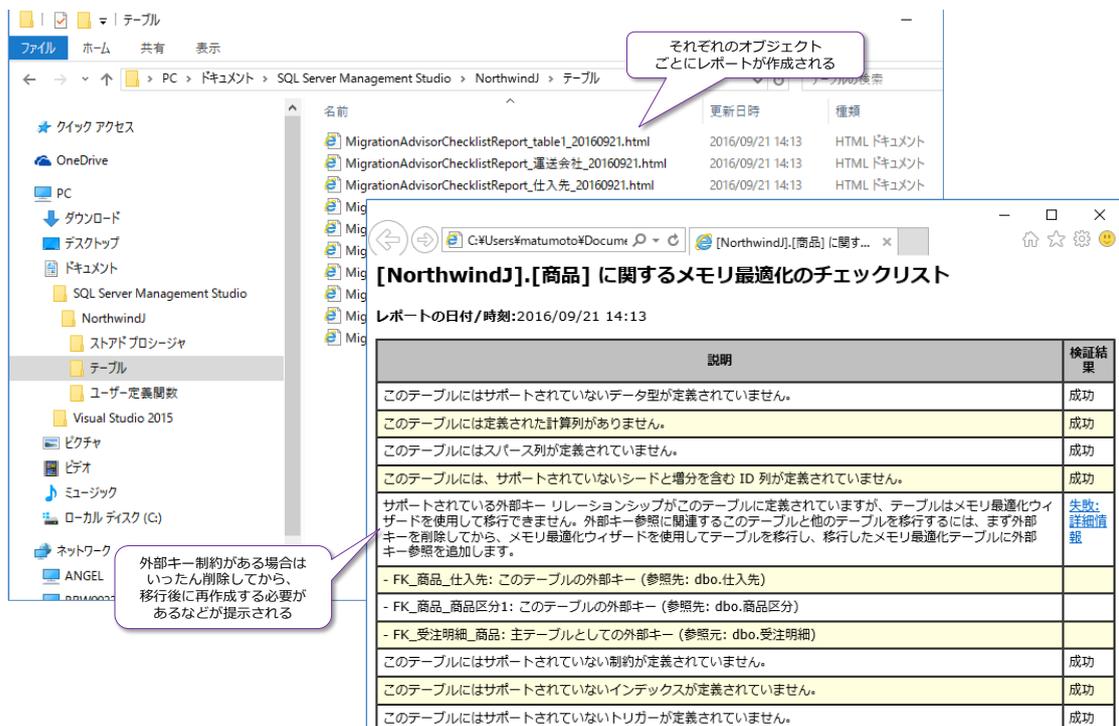


なお、執筆時点の最新版である 2016 年 9 月版の Management Studio (日本語版) だと、この次の画面から先に進むことができなので、2016 年 6 月版 (RTM) の Management Studio を利用する必要があります (今後の Management Studio のアップデート版では修正される予定です)。

チェックが完了すると、次のように表示されます。



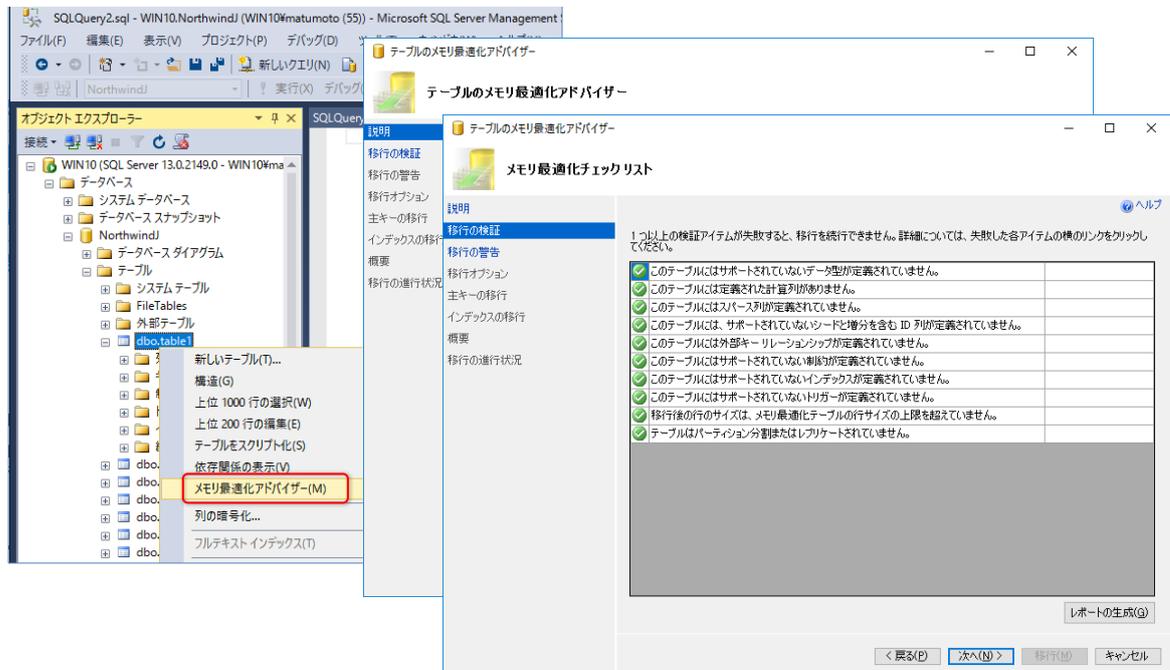
ウィザード完了後は、保存先として指定したフォルダーに、次のようにレポートが作成されていて、移行できるかどうかを確認することができます。



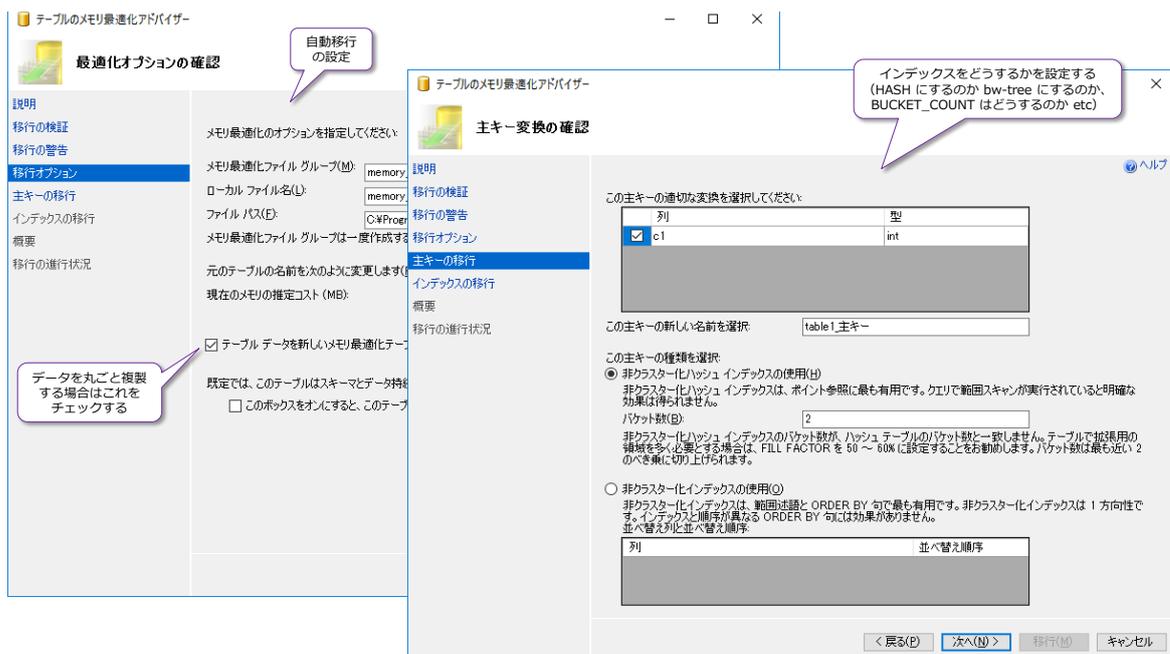
このように、SQL Server 2016 からは、データベース内のオブジェクトが移行可能かどうかをまとめてチェックできるようになったので、大変便利です。

➔ メモリ最適化アドバイザーによる自動移行

メモリ最適化テーブルに移行可能なオブジェクトは、[メモリ最適化アドバイザー] ウィザードを利用することで、**自動移行**（テーブルを自動変換して、データも丸ごと複製）することもできます。このウィザードは、SQL Server 2014 から提供されているツールで、次のように利用できます。



該当テーブルを右クリックして、[メモリ最適化アドバイザー] をクリックすれば、ウィザードが起動して、（移行可能な場合は）次のように自動移行を実行することができます。



このように、既存のテーブルをメモリ最適化テーブルに移行することも簡単に行えます。

2.5 Operational Analytics を試すスクリプト

ここでは、前掲の検証（1 億件のデータに対する GROUP BY 集計）と同じものを試すためのスクリプトを紹介します。なお、この手順どおりに 1 億件のデータを試すには、物理メモリを **64GB** 以上搭載しているマシンが必要になります（：中間テーブルや通常テーブルのデータ格納用のメモリも必要になるため）。物理メモリが **32GB** 以下の場合には、データ件数を 1 千万件にするなどして試してみてください。

1. まずは、データベースを作成します。

```
CREATE DATABASE oaTestDB_1oku
ON PRIMARY (
    NAME = oaTestDB_1oku_data,
    FILENAME = 'M:¥testDB¥oaTestDB_1oku_data.mdf',
    SIZE = 100GB ) ,
FILEGROUP fg1 CONTAINS MEMORY_OPTIMIZED_DATA
( NAME = oaTestDB_1oku_InMem,
  FILENAME = 'M:¥testDB¥oaTestDB_1oku_InMem' )
LOG ON
( NAME = oaTestDB_1oku_log,
  FILENAME='M:¥testDB¥oaTestDB_1oku_log.ldf',
  SIZE = 100GB )
```

ファイルの作成先となるフォルダーには「M:¥testDB」を指定していますが、皆さんの環境に合わせて適宜変更してください。また、データ ファイルおよびトランザクション ログ ファイルのサイズは 100GB で作成しています。

2. 次に、データベースのバックアップとログの切り捨てを行っておきます（：公平な検証にするため、NUL デバイスにダミー バックアップを取得しておきます）。

```
-- DB バックアップ
BACKUP DATABASE oaTestDB_1oku
TO DISK = 'NUL' WITH COMPRESSION, STATS

-- ログの切り捨て
BACKUP LOG oaTestDB_1oku
TO DISK = 'NUL' WITH COMPRESSION, STATS
```

3. 次に、1 億件のデータを格納するためのテーブル（中間テーブル）を作成します。

```
-- 1億件のデータ格納用のテーブル作成
USE oaTestDB_1oku
CREATE TABLE data100M
( col1 int IDENTITY(1,1) NOT NULL
, col2 int NOT NULL
, col3 int NOT NULL
, col4 int NOT NULL
, col5 datetime NOT NULL )
```

このテーブルは、検証用の 3 種類のテーブルにデータをコピーするための、コピー元となるテーブルです。3 種類のテーブルには、同じデータが格納されるように（公平な検証にするために）、この中間テーブルを作成しています。

4. 次に、**WHILE** ループ（1 億回のループ）を利用して、**1 億件のデータ**を追加します。この実行には、2~6 時間ぐらいかかります（実行時間は、CPU のクロック数などで大きく変わります）。

```
-- 1億件のデータ追加.
SET NOCOUNT ON
DECLARE @i int = 1
WHILE @i <= 100000000
BEGIN
    DECLARE @col2 int = CONVERT(int, RAND() * 20000000)
           , @col3 int = CONVERT(int, RAND() * 1000000)
           , @col4 int = CONVERT(int, RAND() * 10000)
           , @col5rnd int = CONVERT(int, RAND() * 2628000) + 1
    DECLARE @col5 datetime = DATEADD(minute, @col5rnd, '2009/01/01')
    INSERT INTO data100M VALUES (@col2, @col3, @col4, @col5)
    SET @i += 1
END
SET NOCOUNT OFF
```

col1 (PRIMARY KEY) には **IDENTITY** で生成した連番、**col2**、**col3**、**col4**、**col5** には乱数 (RAND 関数で生成した値) を格納するようにしています。

1 億件のデータの追加が完了すると、次のように乱数が格納されていることを確認できます。

SELECT * FROM data100M

	col1	col2	col3	col4	col5
1	1	5023503	324864	8013	2010-05-28 01:12:00.000
2	2	1159690	859684	6817	2013-09-02 02:08:00.000
3	3	12554582	562761	7336	2013-02-03 15:16:00.000
4	4	11344397	873227	9046	2013-01-01 01:53:00.000
5	5	6066207	697032	9903	2013-01-01 12:58:00.000
6	6	16121127	770155	233	2013-01-01 22:53:00.000
7	7	4623271	172953	300	CONVERT(int, RAND() * 10000)
8	8	6364128	971945	5264	2009-03-27 21:23:00.000
9	9	35879	7549	2013-06-25 04:50:00.000	
10	10	IDENTITY で 1 からの連番	825	col3 int 0~999,999 の乱数	2011-07-24 00:31:00.000
11	11	1	496	col4 int 0~9,999 の乱数	2012-07-31 13:04:00.000
12	12	124477	CONVERT(int, RAND() * 1000000)	CONVERT(int, RAND() * 2628000) + 1	DATEADD(minute, @col5rnd, '2009/01/01')
13	13	12975051	CONVERT(int, RAND() * 1000000)	CONVERT(int, RAND() * 2628000) + 1	DATEADD(minute, @col5rnd, '2009/01/01')
14	14	6474654	477140	923	2012-04-09 14:07:00.000
15	15	1967	col2 int 0~19,999,999 の乱数	757	2011-12-09 05:35:00.000
16	16	1020	CONVERT(int, RAND() * 20000000)	65	2009-10-14 19:40:00.000
17	17	CONVERT(int, RAND() * 20000000)	10	10	10:54:00.000
18	18	CONVERT(int, RAND() * 20000000)	17	17	05:17:00.000

➔ 3 種類のテーブルの作成 (通常、インメモリ OLTP、Operational Analytics)

次に、検証で利用する 3 種類のテーブルを作成します。

1. まずは、ディスク ベースの通常テーブルを作成します。

```
-- ディスク ベースの通常テーブル
CREATE TABLE OnDisk_GL
( col1 int IDENTITY(1,1) NOT NULL
    PRIMARY KEY CLUSTERED
, col2 int NOT NULL
, col3 int NOT NULL
, col4 int NOT NULL
, col5 datetime NOT NULL )
```

2. 次に、Operational Analytics を試すための、インメモリ OLTP のメモリ最適化テーブルに、クラスター化列ストア インデックスを追加したテーブルを作成します。

```
-- Operational Analytics (InMemory OLTP + CGSI)
CREATE TABLE InMem_HASH_ccsi
( col1 int IDENTITY(1,1) NOT NULL
    PRIMARY KEY NONCLUSTERED
    HASH WITH (BUCKET_COUNT = 100000000)
, col2 int NOT NULL
, col3 int NOT NULL
, col4 int NOT NULL
, col5 datetime NOT NULL
, INDEX idx_InMem_HASH_ccsi CLUSTERED COLUMNSTORE
) WITH ( MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA )
```

HASH インデックスの BUCKET_COUNT (バケット数) は 1 億に設定して、WITH 句で「MEMORY_OPTIMIZED = ON」をしてメモリ最適化テーブルを作成しています。また、「INDEX ~ CLUSTERED COLUMNSTORE」を指定することで、クラスター化列ストア インデックスを追加しています。

3. 次に、インメモリ OLTP のみのテーブル(クラスター化列ストア インデックスを追加しない、単純なメモリ最適化テーブル) を作成します。

```
-- InMemory OLTP のみ (CGSI なし)
CREATE TABLE InMem_HASH
( col1 int IDENTITY(1,1) NOT NULL
    PRIMARY KEY NONCLUSTERED
    HASH WITH (BUCKET_COUNT = 100000000)
, col2 int NOT NULL
, col3 int NOT NULL
, col4 int NOT NULL
, col5 datetime NOT NULL
) WITH ( MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA )
```

➡ 1 億件のデータのコピー

1. 3 種類のテーブルの作成が完了したら、中間テーブル「**data100M**」から、1 億件のデータをコピーします。これには **INSERT .. SELECT** ステートメントを利用します。

```
-- 1億件のデータをコピー
INSERT INTO OnDisk_CL
  SELECT col2, col3, col4, col5 FROM data100M

INSERT INTO InMem_HASH_ccsi
  SELECT col2, col3, col4, col5 FROM data100M

INSERT INTO InMem_HASH
  SELECT col2, col3, col4, col5 FROM data100M
```

それぞれの実行時間は、環境によって大きく異なりますが、3分～数十分程度なので、合計で9分～1時間程度かかります。

2. データのコピーが完了した後、インメモリ OLTP のメモリ使用量を確認するには、次のように **dm_db_xtp_table_memory_stats** システム ビューを利用します。

```
SELECT OBJECT_NAME(object_id), *
FROM sys.dm_db_xtp_table_memory_stats
```

➡ 検証の実行 ～集計クエリの実行～

データのコピーが完了したら、**GROUP BY** を利用した**集計クエリ**を実行して、性能をチェックしてみましょう。

1. **SET STATISTICS TIME ON** を付けて、実行時間を計測するようにして、次の 3 つのクエリを実行してみます。

```
SET STATISTICS TIME ON

-- On Disk
SELECT col4, COUNT(*) AS cnt FROM OnDisk_CL
GROUP BY col4
ORDER BY col4

-- Operational Analytics
SELECT col4, COUNT(*) AS cnt FROM InMem_HASH_ccsi
GROUP BY col4
ORDER BY col4

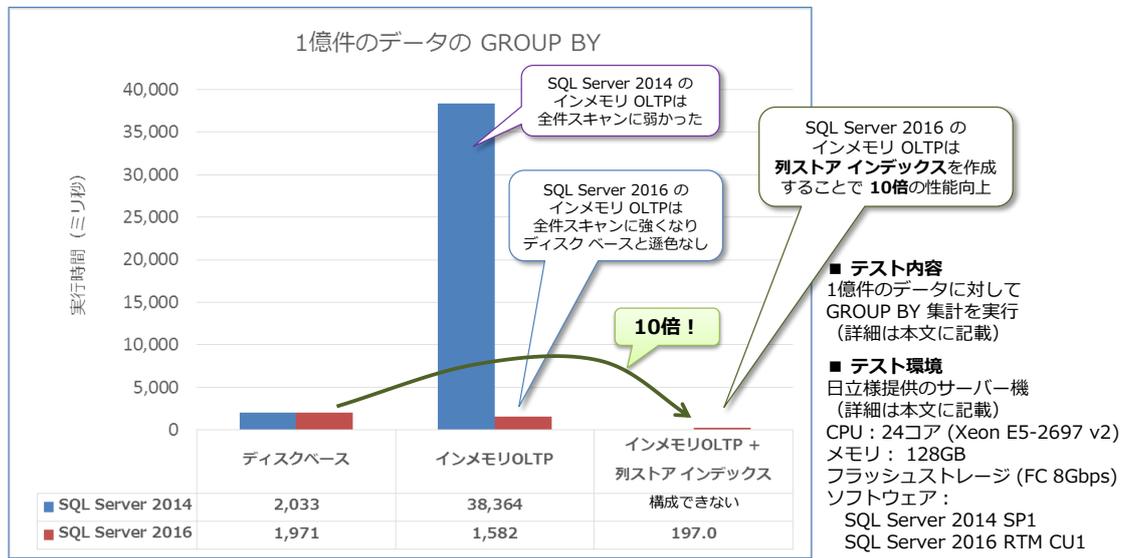
-- InMemory OLTP のみ
SELECT col4, COUNT(*) AS cnt FROM InMem_HASH
GROUP BY col4
```

```
ORDER BY col4
```

```
SET STATISTICS TIME OFF
```

実行時の注意点は、1回目の実行時には、コンパイルと統計の作成時間が含まれるので、これを除外するようにします。SQL Server では、GROUP BY で指定した列に統計が作成されていない場合には、初回実行時に統計を自動的に作成するので、その分の実行時間が余分にかかってしまいます。2回目以降は、統計の作成は行われないので、2回、3回、4回、5回と実行して、そのときの実行時間を比較（3～5 回目の実行時の平均値を計算するなど）してみてください。

このクエリを筆者が日立様のハードウェア（24 コア機）で検証したときの結果が、次のグラフ（再掲）です。



* ベンチマークの結果の公表は、使用許諾契約書で禁止されていますが、その効果をわかりやすく表現するため、日本マイクロソフト株式会社の監修のもと、数値を掲載しております。

➡ 性能が確認できない場合の対処方法

もし、クラスター化列ストア インデックスを追加した効果（性能向上）を確認できない場合は、次の手順を試してみてください。

1. まずは、`dm_db_column_store_row_group_physical_stats` 動的管理ビュー（DMV）を利用して、データの状態を確認します。

```
-- データの格納状態を確認
SELECT OBJECT_NAME(object_id), *
FROM sys.dm_db_column_store_row_group_physical_stats
WHERE OBJECT_NAME(object_id) = 'InMem_HASH_ccsi'
```

```
-- データの格納状態を確認
SELECT OBJECT_NAME(object_id), *
FROM sys.dm_db_column_store_row_group_physical_stats
WHERE OBJECT_NAME(object_id) = 'InMem_HASH_ccsi'
```

(列名なし)	object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_desc	total_row
InMem_HASH_ccsi	613577224	1	1	-1	NULL	1	OPEN	1000000

この DMV は、列ストア インデックスの **Row Group** (列ストアを内部的に分割する単位で、通常は 100 万件ごとに 1 つの Row Group が作成される) を確認できるものです。インメモリ OLTP では、定期的に Row Group の再作成(圧縮/COMPRESS)が行われるのですが、これがまだ実行されていない場合には、[state_desc] が「OPEN」と表示されます。ここで、**OPEN** と表示される場合は、次のように「sp_memory_optimized_cs_migration」ストアード プロシージャを実行して、強制的に圧縮を実行するようにします。

- 「sp_memory_optimized_cs_migration」ストアード プロシージャの実行は、次のように OBJECT_ID を指定して行えます。

```
-- 強制的に圧縮を実行
DECLARE @objid int = OBJECT_ID('InMem_HASH_ccsi')
EXEC sp_memory_optimized_cs_migration @objid
```

- 実行後、もう一度 dm_db_column_store_row_group_physical_stats 動的管理ビューを参照して、データの状態を確認します。

```
SELECT OBJECT_NAME(object_id), *
FROM sys.dm_db_column_store_row_group_physical_stats
WHERE OBJECT_NAME(object_id) = 'InMem_HASH_ccsi'
```

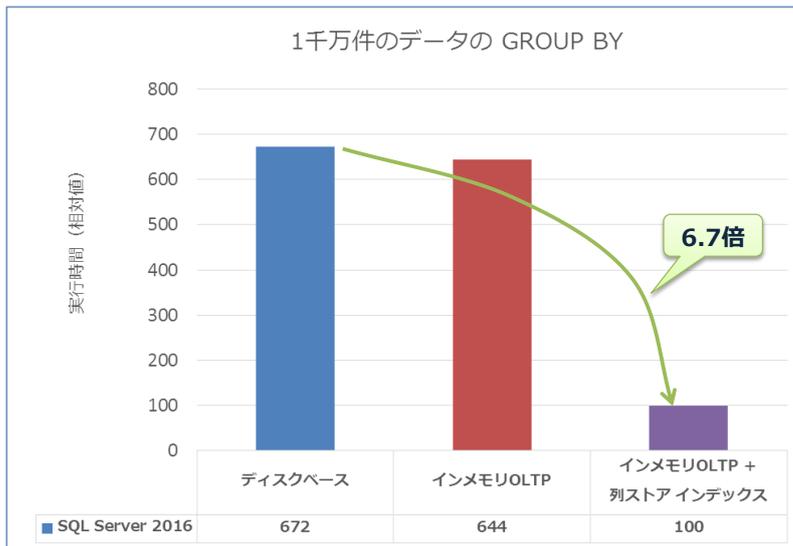
```
-- データの格納状態を確認
SELECT OBJECT_NAME(object_id), *
FROM sys.dm_db_column_store_row_group_physical_stats
WHERE OBJECT_NAME(object_id) = 'InMem_HASH_ccsi'
```

(列名なし)	object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_desc	total_row
InMem_HASH_ccsi	613577224	1	1	-1	NULL	1	OPEN	0
InMem_HASH_ccsi	613577224	1	1	3	NULL	3	COMPRESSED	1048576
InMem_HASH_ccsi	613577224	1	1	8	NULL	3	COMPRESSED	1048576
InMem_HASH_ccsi	613577224	1	1	11	NULL	3	COMPRESSED	216997
InMem_HASH_ccsi	613577224	1	1	2	NULL	3	COMPRESSED	1048576
InMem_HASH_ccsi	613577224	1	1	7	NULL	3	COMPRESSED	1048576
InMem_HASH_ccsi	613577224	1	1	10	NULL	3	COMPRESSED	445253
InMem_HASH_ccsi	613577224	1	1	1	NULL	3	COMPRESSED	1048576
InMem_HASH_ccsi	613577224	1	1	5	NULL	3	COMPRESSED	1048576
InMem_HASH_ccsi	613577224	1	1	9	NULL	3	COMPRESSED	447291
InMem_HASH_ccsi	613577224	1	1	4	NULL	3	COMPRESSED	1048576

COMPRESS と表示された行が追加されれば成功です (だいたい 100 万件ごとに 1 つの Row Group が作成されるので、1 億件なら 100 個の行が返ります)。この状態になったら、再度 **GROUP BY** クエリ (1 億件のデータに対する集計クエリ) を実行して、性能を確認してみてください。

➔ デスクトップ機での 1 千万件の場合の性能結果

同じスクリプトを **1 千万件のデータ量**に変更して (**WHILE** ループで 1 億回している部分を 1 千万回に変更して)、弊社のデスクトップ機 (Core i7 6700K、64GB メモリ) のマシンで性能を検証した結果は、次のとおりです。



インメモリ OLTP+クラスター化列ストア インデックスでは、**約 6.7 倍**の性能向上を確認することができました。

このように、SQL Server 2016 のインメモリ OLTP は、クラスター化列ストア インデックスを追加することで、データ分析/集計 (Analytics ワークロード) に強くすることができるので、ぜひ試してみてください。

STEP 3. 列ストア インデックスを 利用した Operational Analytics

この STEP では、インメモリ OLTP を利用しない形での Operational Analytics（通常のリレーショナル テーブルに列ストア インデックスを追加するパターン）を実現する方法として、列ストア インデックスの基本的な利用方法を説明します。

この STEP では、次のことを学習します。

- ✓ 列ストア インデックスの基本的な利用方法
- ✓ 1,000 万件のデータを格納した通常テーブルでの性能チェック
- ✓ 非クラスター化列ストア インデックスでの性能チェック
- ✓ クラスター化列ストア インデックスでの性能チェック
- ✓ 列ストア インデックスの圧縮率の比較
- ✓ データ更新とインデックスの再構築
- ✓ 10 万件の BULK INSERT を実行したときの性能比較

3.1 列ストア インデックスを利用した Operational Analytics

ここまでで説明してきたインメモリ OLTP とクラスター化列ストア インデックスの組み合わせは、**完全なインメモリ**での **Operational Analytics** の実装になりますが、インメモリ OLTP を利用しなくても、次のように Operational Analytics を実装することができます（以下は、冒頭で紹介した図を再掲）。

Operational Analytics (OLTPとデータ分析の両立)

Operational ワークロード

= OLTP (オンライン トランザクション処理) など



SQL Server 2014 までは
インメモリ OLTP や
通常リレーショナル テーブルを利用

Analytics ワークロード

= データ集計/分析処理、DWH、夜間バッチなど



SQL Server 2014 までは
列ストア インデックスや
Analysis Services などを利用



SQL Server 2016 では両立可能

次の 3パターンで実装できる

1. インメモリ OLTP + クラスター化列ストア インデックス (完全なインメモリ実装)
2. 通常リレーショナル テーブル + 非クラスター化列ストア インデックス
3. 通常リレーショナル テーブル + クラスター化列ストア インデックス

インメモリ OLTP と列ストア インデックスを組み合わせる場合には、インメモリ OLTP が完全にインメモリで動作するので、メモリ上の制限（最大サイズが **2TB** までという上限）を受けます。このため、数 TB (テラバイト) 規模になるなど、メモリに載りきれないデータ量になる場合には、インメモリ OLTP を利用することができません。

このような場合には、通常のリレーショナル テーブル（従来ながらのディスク ベースのテーブル）に、**列ストア インデックス**を追加して、Operational Analytics を実現することができます。列ストア インデックスは、基本はインメモリで動作しますが（データ量が物理メモリに載りきる場合はインメモリで動作）、メモリに載りきれないデータがあった場合には、ディスクを利用して動作させることができるからです（ハイブリッドな動作ができます）。

➡ 更新可能になった非クラスター化列ストア インデックス

SQL Server 2016 の列ストア インデックスに関して、私たちが一番喜んでいるのは、これまでは読み取り専用だった「**非クラスター化列ストア インデックス**」が更新可能になったことです。これを利用すれば、通常のリレーショナル テーブルに、容易に列ストア インデックスを追加することができます（従来ながらの B-tree インデックスを追加するのと同じような感覚で、列ストア インデックスを簡単に追加できるようになりました）。

現在、「**データ分析/集計で時間がかかってしまっている**」や「**夜間バッチの実行に時間がかかってしまっている**」などの悩みを抱えている場合には、まず非クラスター化列ストア インデックスを作成してみることをお勧めします。夜間バッチの前に（バッチ実行時の最初の処理として）非クラスター化列ストア インデックスを追加して、夜間バッチが完了したら削除する、といった使い方もできるので、性能に問題を抱えている場合は、ぜひ試してみてください。

また、SQL Server 2016 では、列ストア インデックスそのものの**性能向上**（バッチ モードの強

化、プッシュダウン、更新性能の向上、パラレル Insert など) も実現しているので、現在の SQL Server で性能に問題を抱えているという方には、ぜひ試してみてほしい機能です。

➔ SQL Server 2016 からの列ストア インデックスの飛躍的な進化

SQL Server 2016 からの列ストア インデックスの強化ポイントをまとめると、次のようになります。

- **インメモリ OLTP にクラスター化列ストア インデックス**を作成可能に
(=完全なインメモリでの Operational Analytics の実装)
- **列ストア インデックスの性能向上** (バッチ モードの性能向上、集計のプッシュダウン、更新性能の向上、ウィンドウ関数のバッチ モード対応など)
- 非クラスター化列ストア インデックスが**更新可能**に
- 非クラスター化列ストア インデックスで**フィルター列**が利用可能に
- クラスター化列ストア インデックスでの**追加の B-tree インデックス**のサポート
- 主キー/外部キー制約のサポート
- AlwaysOn 可用性グループの読み取り可能セカンダリのサポート
- COMPRESS_DELAY (遅延圧縮) のサポート

何度も繰り返しになりますが、本書のタイトルにもなっている **Operational Analytics** に対応したことが一番のポイントです。これまでは **Analytics ワークロード** (データ集計/分析) に特化していた列ストア インデックスが、SQL Server 2016 からは **Operational ワークロード (OLTP)** にも対応して OLTP とデータ分析を両立できるようになりました。

既存環境からの移行という観点では、**PRIMARY KEY 制約**がサポートされるようになった点は大きく、これまでの構成を大きく変更することなく移行できるようになりました。

3.2 列ストア インデックスの基本的な利用方法

ここでは、列ストア インデックスの基本的な利用方法を説明します。

➔ 非クラスター化列ストア インデックスの作成方法

既存のディスク ベースのテーブルに対して、非クラスター化列ストア インデックスを追加する場合には、次のように **CREATE NONCLUSTERED COLUMNSTORE INDEX** ステートメントを利用します。

```
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, ...)
[WHERE フィルター条件]
```

WHERE を付けて、フィルター条件を追加できるようになったのは、SQL Server 2016 からの新機能です。フィルターを利用すれば、インデックス サイズを小さくすることができるので、集計クエリで扱うデータのフィルター条件と合致する場合には、ぜひ試してみてください。

なお、非クラスター化列ストア インデックスでは、列名を指定して、作成することができるので、次のように特定のクエリにのみ強いインデックスを作成することも可能です。

```
-- 速くしたい集計クエリ
SELECT b, MAX(c) FROM t1_nccsi
GROUP BY b

-- 非クラスター化列ストア インデックスの作成
CREATE NONCLUSTERED COLUMNSTORE INDEX idx1
ON t1_nccsi(b, c)
```

b 列で集計して、c 列のデータの MAX を取得しているので、非クラスター化列ストア インデックスには b, c のみを含めれば良い

ただし、テーブルに作成できる非クラスター化列ストア インデックスは **1つのみ**なので、列ごとに非クラスター化列ストア インデックスを作成しようとする（b 列と c 列に対するインデックスをそれぞれ作成しようとする）、次のようにエラーが返されます。

```
-- b列に 非クラスター化列ストア インデックスを作成
CREATE NONCLUSTERED COLUMNSTORE INDEX idx_b
ON t1_nccsi(b)

-- c列に 非クラスター化列ストア インデックスを作成
CREATE NONCLUSTERED COLUMNSTORE INDEX idx_c
ON t1_nccsi(c)
```

テーブルに 2つ目の非クラスター化列ストア インデックスを作成することはできない

メッセージ 35339、レベル 16、状態 1、行 53
複数の columnstore インデックスはサポートされていません。

このように、複数の列ストア インデックスはサポートされていないので、前掲の例のように集計クエリで b と c 列を利用するのであれば、b, c を含めた非クラスター化列ストア インデックスを作成するようにします。

➔ クラスター化列ストア インデックスの作成方法

既存のテーブルに対して、クラスター化列ストア インデックスを追加する場合には、次のように **CREATE CLUSTERED COLUMNSTORE INDEX** ステートメントを利用します。

```
CREATE CLUSTERED COLUMNSTORE INDEX インデックス名
ON テーブル名
```

クラスター化列ストア インデックスは、すべての列に対してカラム ストアを作成するので、列名を指定する必要はありません。

クラスター化列ストア インデックスを作成するための条件は、次のようになります。

- テーブルに対して作成できるのは **1つのみ**
- b-tree の**クラスター化インデックス**とは共存できない (**非クラスター化インデックス**とは共存できる)

PRIMARY KEY 制約を作成している場合には、既定で b-tree のクラスター化インデックスが作成されるので、これは非クラスター化インデックスに変更しておく必要があります (SQL Server 2014 では、非クラスター化インデックスとも共存することができませんでしたが、SQL Server 2016 からは共存できるようになりました)

- クラスター化列ストア インデックスを作成したテーブルに、非クラスター化列ストア インデックスを追加することはできない (複数の列ストア インデックスは作成できない)

テーブルの作成時にクラスター化列ストア インデックスを作成する場合

テーブルの作成時にクラスター化列ストア インデックスを作成する場合には、次のように **CREATE TABLE** ステートメントを記述します。

```
CREATE TABLE テーブル名
( 列名1 データ型
, 列名2 データ型
:
, INDEX インデックス名 CLUSTERED COLUMNSTORE )
```

```
-- テーブルの作成時にクラスター化列ストア インデックスを追加する例
CREATE TABLE ccsiTest
( col1 int IDENTITY(1,1) NOT NULL
  PRIMARY KEY NONCLUSTERED
, col2 int NOT NULL
, col3 int NOT NULL
, INDEX idx_ccsiTest CLUSTERED COLUMNSTORE )
```

100 % <

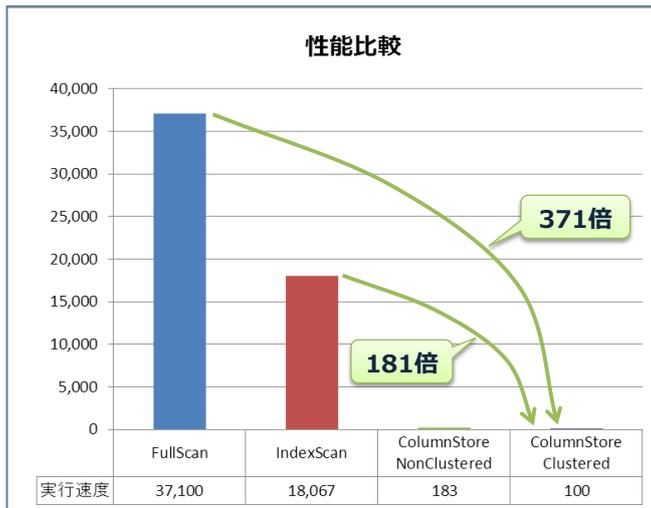
メッセージ
コマンドは正常に完了しました。

INDEX キーワードに続けてインデックス名を指定し、**CLUSTERED COLUMNSTORE** を付ける

ことでクラスター化列ストア インデックスを作成することができます。

➡ 列ストア インデックスによる性能向上

この後の手順では、**1,000 万件のデータ**に対する **GROUP BY 演算**（集計クエリ）を実行するスクリプトを紹介していますが、このスクリプトを実行したときの実行時間を比較すると、次のグラフのようになります。



テストに利用したハードウェア環境

- ・ Core i7 6700K 4GHz
- ・ メモリ 64GB
- ・ SSD Crucial MX200 1TB

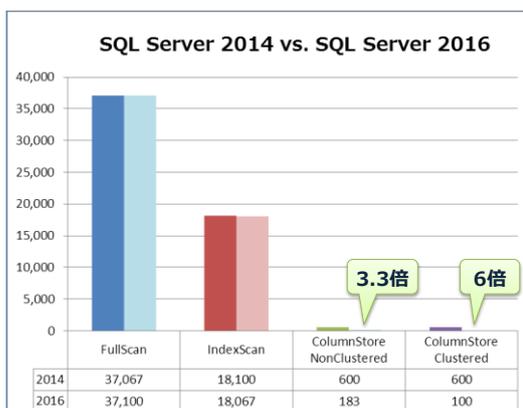
性能結果は相対値

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、クラスター化列ストア インデックスの値を 100 とした場合の相対値で表しています。

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、列ストア インデックスではない通常の b-tree の非クラスター化インデックスを作成/利用したとき、**ColumnStore NonClustered** が非クラスター化列ストア インデックスを作成/利用したとき、**ColumnStore Clustered** がクラスター化列ストア インデックス (CCSI) を作成/利用したときの実行速度です（相対値）。

列ストア インデックスを利用することで、フル スキャンに比べて **371 倍**、インデックス スキャンに比べて **181 倍**もの性能向上を実現することができます。このように、列ストア インデックスを利用すれば、集計クエリ（Analytics ワークロード）に強くすることができます。

なお、同じクエリを SQL Server 2014 で実行したときと比較すると、次のグラフのようになります。SQL Server 2014 に比べても性能がアップしていることを確認できると思います。



テストに利用したハードウェア環境

- ・ Core i7 6700K 4GHz
- ・ メモリ 64GB
- ・ SSD Crucial MX200 1TB

性能結果は相対値

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、クラスター化列ストア インデックスの値を 100 とした場合の相対値で表しています。

3.3 列ストア インデックスの性能比較

それでは、**列ストア インデックス**を試してみましょう。インデックスを何も作成していない場合や、b-tree のインデックスを作成した場合、非クラスター化列ストア インデックスを作成した場合、クラスター化列ストア インデックスを作成した場合の性能比較などを行います。

列ストア インデックスは、複数コア環境でより威力を発揮するので、検証する際には、複数コアを搭載した物理マシン、または複数コアを割り当てた仮想マシンを利用するのがお勧めです。なお、SQL Server 2014 のときには、1 コア環境だとバッチ モード（内部的に列ストア インデックスを処理する際の動作モード）で動作することができなかったので、1 コア環境だと列ストア インデックスの性能を確認することができなかったのですが、SQL Server 2016 からは、1 コアでもバッチ モードがサポートされるようになったので、1 コアでも性能差を確認できるようになりました。

➡ データベースの作成

1. まずは、データベースを作成します。データベースの名前は「**csTestDB**」として、次のように **CREATE DATABASE** ステートメントを実行します。

```
CREATE DATABASE csTestDB
ON PRIMARY ( NAME = 'csTestDB'
, FILENAME = 'C:\%testDB%\csTestDB_data.mdf'
, SIZE = 15GB )
LOG ON ( NAME = 'csTestDB_log'
, FILENAME = 'C:\%testDB%\csTestDB_log.ldf'
, SIZE = 10GB )
```

FILENAME で指定しているファイル パス（データ ファイルとトランザクション ログ ファイルの作成先となるフォルダー）には、「**C:\%testDB**」フォルダーを指定していますが、皆さんの環境に合わせて適宜変更してください。

➡ テーブル「t1」の作成（ディスク ベースの通常テーブル）

1. 次に、**CREATE TABLE** ステートメントを利用して、通常どおりに「**t1**」という名前のテーブルを作成します。

```
USE csTestDB
CREATE TABLE t1
( a int IDENTITY(1, 1) PRIMARY KEY
, b int
, c char(200) DEFAULT 'dummy1'
, d char(200) DEFAULT 'dummy2'
)
```

```
-- 通常テーブル「t1」の作成
USE csTestDB
CREATE TABLE t1
( a int IDENTITY(1, 1) PRIMARY KEY
, b int
, c char(200) DEFAULT 'dummy1'
, d char(200) DEFAULT 'dummy2'
)

100 % <
メッセージ
コマンドは正常に完了しました。
```

a、b、c、d の 4 つの列を作成して、a 列には **IDENTITY** (1 からの連番) と **PRIMARY KEY** 制約を設定しておきます (これによって、内部的には、**b-tree** の**クラスター化インデックス**が自動作成されています)。

➡ 1,000 万件のデータの INSERT

- 次に、**WHILE** ループを利用して、**1,000 万件のデータ**を **INSERT** します。

```
SET NOCOUNT ON
DECLARE @i int = 1, @b int = 1
WHILE @i <= 10000000
BEGIN
    IF @i % 10000 = 0 SET @b = @i
    INSERT INTO t1(b) VALUES(@b)
    SET @i += 1
END
SET NOCOUNT OFF
```

```
-- 10,000,000 (1000万件) のデータを追加
SET NOCOUNT ON
DECLARE @i int = 1, @b int = 1
WHILE @i <= 10000000
BEGIN
    IF @i % 10000 = 0 SET @b = @i
    INSERT INTO t1(b) VALUES(@b)
    SET @i += 1
END
SET NOCOUNT OFF

100 % <
メッセージ
コマンドは正常に完了しました。
```

1,000 万件のデータを追加しているので、環境にもよりますが、実行には 10 分～数時間くらいの時間がかかります (ディスクが低速な場合には、より実行時間が長くなります)。

- データの追加が完了したら、次のように **SELECT** ステートメントを実行して、追加されたデータを確認しておきましょう。

```
-- 上位 10万件を確認
SELECT TOP 100000 * FROM t1
```

```
-- 上位 10万件を確認
SELECT TOP 100000 * FROM t1
```

a	b	c	d
9997	9997	1	dummy1
9998	9998	1	dummy1
9999	9999	1	dummy1
10000	10000	10000	dummy1
10001	10001	10000	dummy1
10002	10002	10000	dummy1
	03	10000	dummy1
	04	10000	dummy1

結果

メッセージ

a 列には 1 からの連番

b 列は 10000 ごとに同じ値

a 列には、IDENTITY による連番、b 列には、10,000 件ごとに、1、10,000、20,000、30,000 という値が入るようにしています。

3. 次に、COUNT 関数を利用して、データ件数が 1,000 万件であることを確認しておきます。

```
SELECT COUNT(*) FROM t1
```

```
-- データ件数を確認
SELECT COUNT(*) FROM t1
```

(列名なし)
1

結果

メッセージ

➡ 通常テーブルでの速度チェック

次に、通常テーブルでの実行速度をチェックします（この後に作成する非クラスター化列ストア インデックスや、クラスター化列ストア インデックスとの比較を行うために、通常テーブルでの速度をチェックしておきます）。

1. まずは、実行速度をチェックするために、SET STATISTICS TIME を ON へ設定して、GROUP BY 演算を利用した SELECT ステートメントを実行します。

```
SET STATISTICS TIME ON

SELECT b, COUNT(*) FROM t1
GROUP BY b
```

```
-- 実行時間をチェックする
SET STATISTICS TIME ON

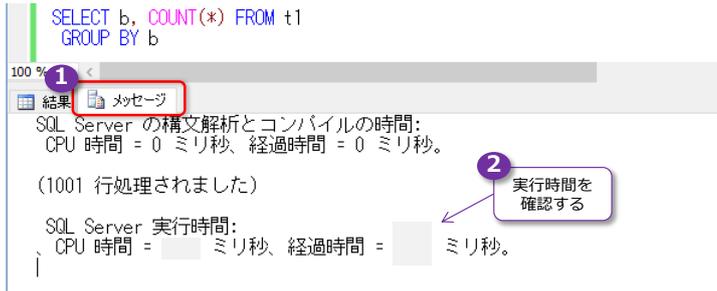
SELECT b, COUNT(*) FROM t1
GROUP BY b
```

	b	(列名なし)
1	1	9999
2	20000	10000
3	30000	10000
4	60000	10000

結果

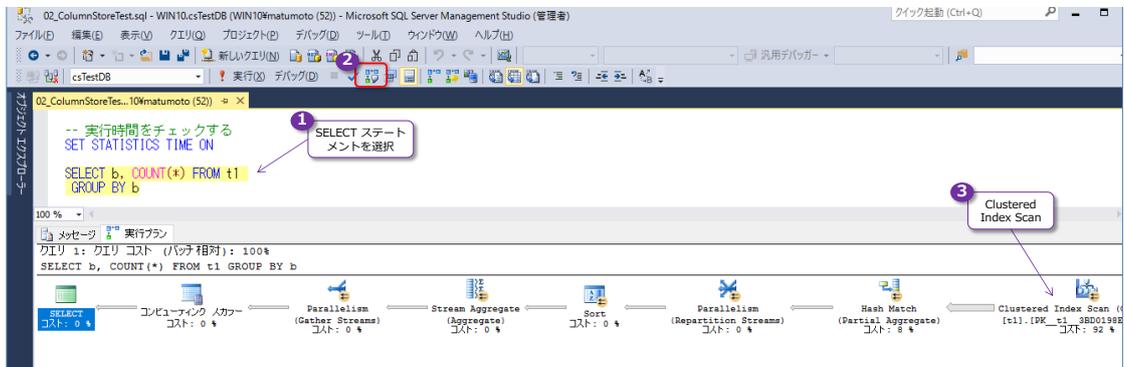
メッセージ

実行が完了したら、[メッセージ] タブを開きます。



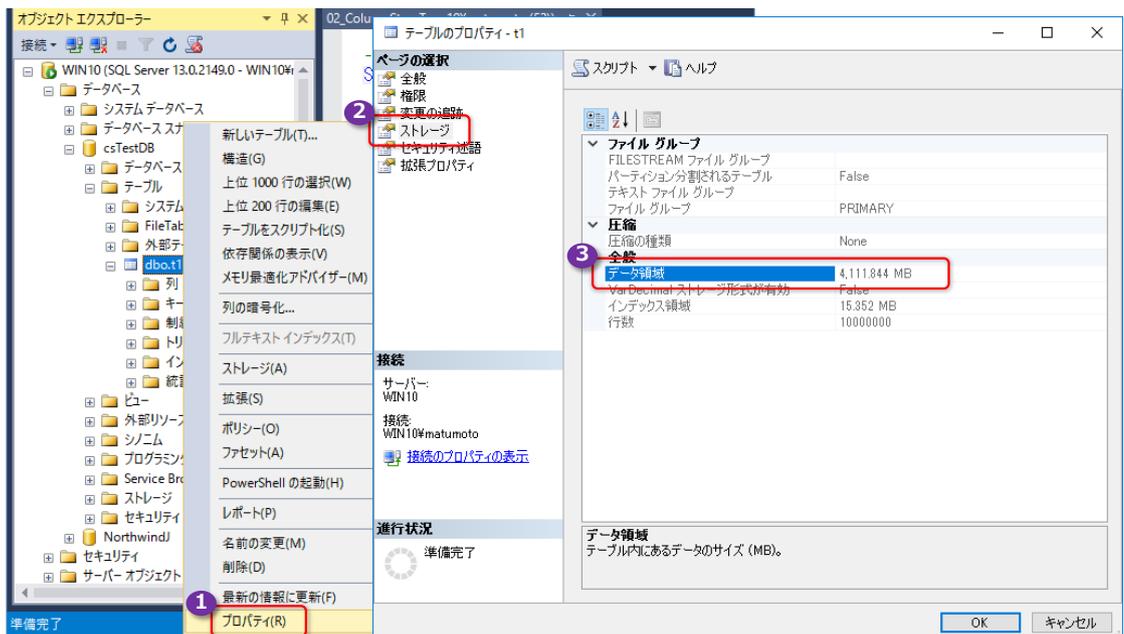
経過時間のところで実行時間を確認することができるので、これをメモしておいてください。

- 次に、**SELECT** ステートメントを選択して、ツールバーの [推定実行プラン] ボタンをクリックし、実行プランを確認します。



Clustered Index Scan (全データのスキャン) が選択されていることを確認できます。t1 テーブルの a 列には、**PRIMARY KEY** 制約を設定しているため、クラスター化インデックスが自動作成されていて、このインデックス (実データ) が全スキャンされています。

- 次に、**テーブルのプロパティ**を開いて、**テーブル サイズ**を確認します。



オブジェクト エクスプローラーで、t1 テーブルを右クリックして、[プロパティ] をクリッ

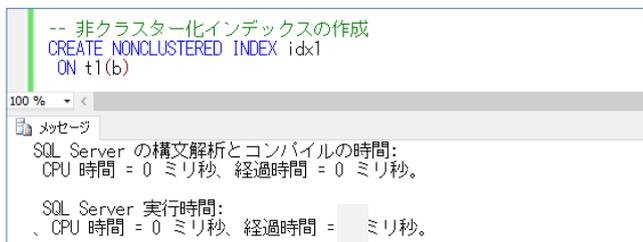
くし、[ストレージ] ページを開くと、[データ領域] でテーブル サイズを確認することができます（約 4GB のサイズであることを確認できます）。

➔ b-tree 非クラスター化インデックスの作成（通常のインデックスの場合の性能）

次に、**非クラスター化インデックス**（列ストア インデックスではなく、通常の b-tree インデックス）を作成した場合の性能を調べてみましょう。

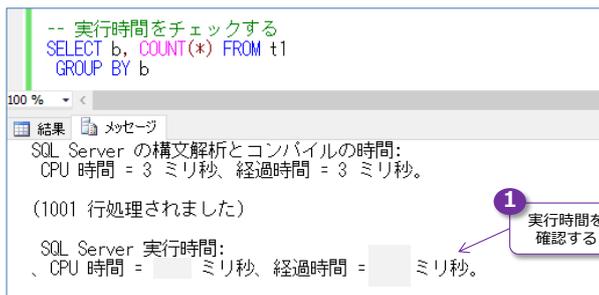
1. 今回作成する非クラスター化インデックスは、前掲の **GROUP BY** 演算（b 列の COUNT）を速く実行するためのものになるので、次のように **b** 列に対して作成します。

```
CREATE NONCLUSTERED INDEX idx1
ON t1 (b)
```



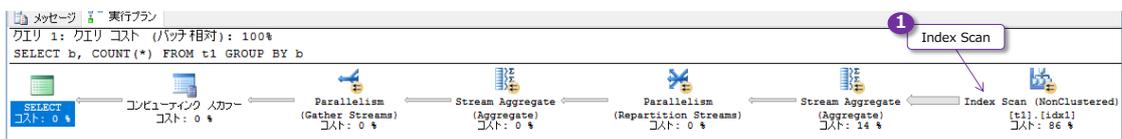
2. インデックスの作成が完了したら、**SELECT** ステートメントを実行して、**実行時間**を確認します。

```
SELECT b, COUNT(*) FROM t1
GROUP BY b
```



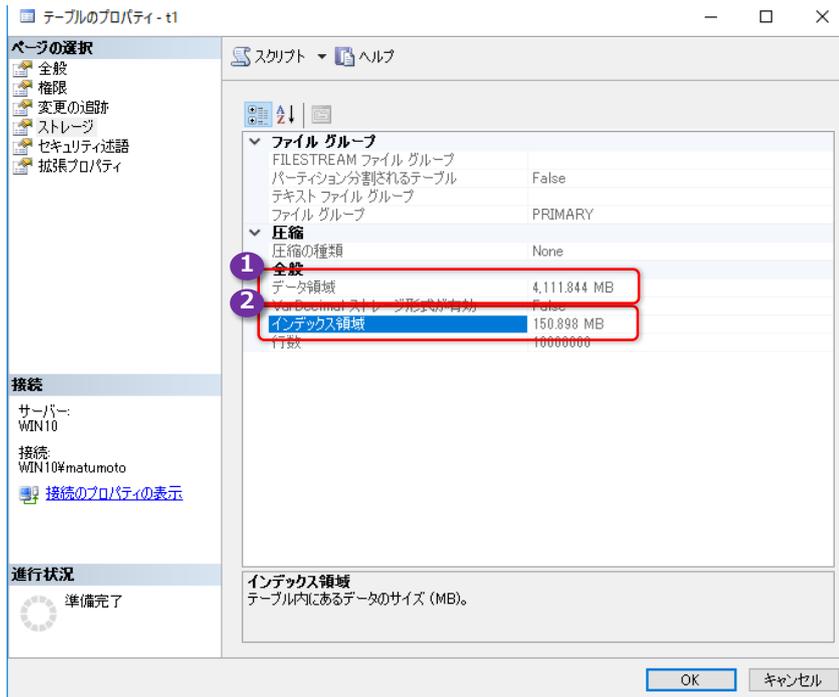
経過時間が小さくなっていることを確認できると思います（弊社環境では、約 **61%** 速く実行できることを確認しています）。

3. 次に、SELECT ステートメントを選択して、ツールバーの [**推定実行プラン**] ボタンをクリックし、実行プランも確認しておきます。



Index Scan と表示されていることを確認できると思います。

4. 次に、テーブル サイズとインデックス サイズも確認します。



データ領域（テーブル サイズ）が約 4GB、インデックス領域（インデックスのサイズ）が約 151MB であることを確認できます。

➡ 非クラスター化列ストア インデックスの作成、性能チェック

次に、非クラスター化列ストア インデックスを作成した場合の性能を調べてみましょう。

1. まずは、非クラスター化列ストア インデックスを試すためのテーブルを「t1_nccsi」という名前で作成します（t1 テーブルと全く同じ構造にします）。

```
CREATE TABLE t1_nccsi
( a int IDENTITY(1, 1) PRIMARY KEY
, b int
, c char(200) DEFAULT 'dummy1'
, d char(200) DEFAULT 'dummy2'
)
```

2. 次に、1 千万件分のデータを INSERT しますが、t1 テーブルと同じデータにするために、次のように INSERT .. SELECT ステートメントを利用して、データを丸ごとコピーします。

```
INSERT INTO t1_nccsi (b, c, d)
SELECT b, c, d FROM t1
```

```
-- 1千万件のコピー
INSERT INTO t1_nccsi (b, c, d)
SELECT b, c, d FROM t1
```

100 % < >

メッセージ

SQL Server の構文解析とコンパイルの時間:
CPU 時間 = 0 ミリ秒、経過時間 = 3 ミリ秒。

SQL Server 実行時間:
、CPU 時間 = ミリ秒、経過時間 = ミリ秒。

(10000000 行処理されました)

3. 次に、**CREATE NONCLUSTERED COLUMNSTORE INDEX** ステートメントを利用して、**非クラスター化列ストア インデックス**を作成します。

```
CREATE NONCLUSTERED COLUMNSTORE INDEX idx1
ON t1_nccsi (a, b, c, d)
```

```
-- 非クラスター化列ストア インデックスの作成
CREATE NONCLUSTERED COLUMNSTORE INDEX idx1
ON t1_nccsi(a, b, c, d)
```

100 % < >

メッセージ

SQL Server の構文解析とコンパイルの時間:
CPU 時間 = 0 ミリ秒、経過時間 = 0 ミリ秒。

SQL Server 実行時間:
、CPU 時間 = ミリ秒、経過時間 = ミリ秒。

4. 作成が完了したら、同じ **SELECT** ステートメントを実行して、**実行時間**を確認します。

```
SELECT b, COUNT(*) FROM t1_nccsi
GROUP BY b
```

```
-- 実行時間をチェックする
SELECT b, COUNT(*) FROM t1_nccsi
GROUP BY b
```

100 % < >

結果 | メッセージ

SQL Server の構文解析とコンパイルの時間:
CPU 時間 = 0 ミリ秒、経過時間 = 0 ミリ秒。

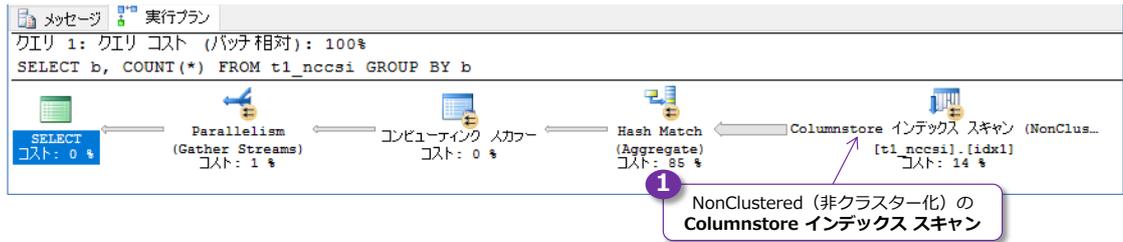
(1001 行処理されました)

SQL Server 実行時間:
、CPU 時間 = ミリ秒、経過時間 = ミリ秒。

1 実行時間を確認する

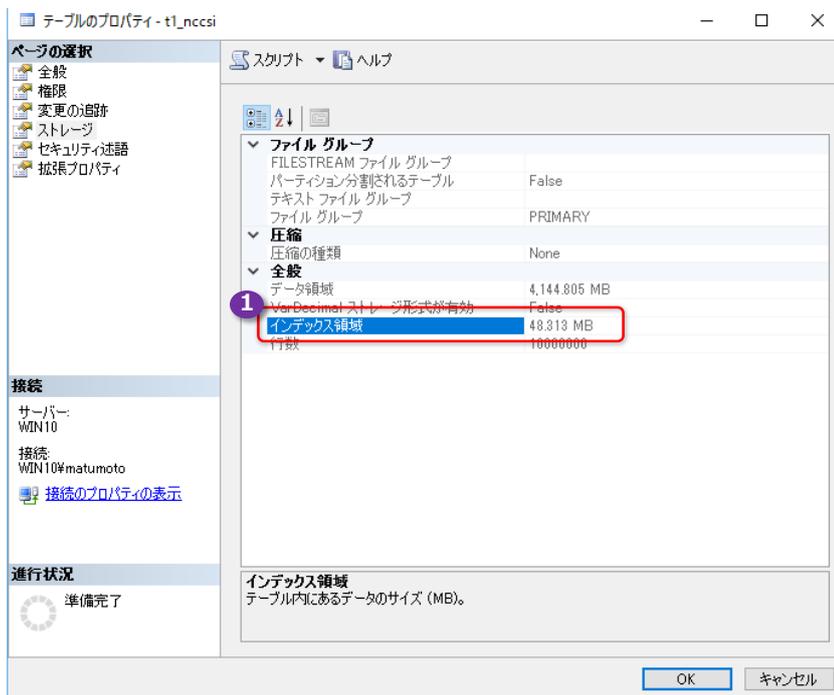
経過時間が桁違いに小さくなっていることを確認できると思います（弊社環境では、約 202 倍速く実行できることを確認しています。1 回目の実行は、コンパイル時間もあるので、それは除外して、2 回、3 回、4 回と実行したときの、それらの平均実行時間で比較しています）。

5. 次に、**SELECT** ステートメントを選択して、ツールバーの [**推定実行プラン**] ボタンをクリックし、実行プランも確認しておきます。



Columnstore インデックス スキャン (NonClustered) と表示されて、非クラスター化列ストア インデックスのスキャンで実行されていることを確認できると思います。

6. 次に、インデックス サイズも確認します。



インデックス領域（インデックスのサイズ）が約 **49MB** であることを確認できます（b-tree の非クラスター化インデックスの場合は 約 **151MB** でした）。

➡ SQL Server 2016 からは更新可能に

SQL Server 2016 からは、非クラスター化列ストア インデックスでも、更新ができるようになったので、これも試してみましょう（SQL Server 2012/2014 では読み取り専用でした）。

1. まずは、**UPDATE** ステートメントを実行して、データを更新してみます。

```
UPDATE t1_nccsi
SET c = 'zzzz'
WHERE a = 1
```

```
-- SQL Server 2016 からはデータ更新が可能に
UPDATE t1_nccsi
SET c = 'zzzz'
WHERE a = 1
```

100 % <

メッセージ

(1 行処理されました)

問題なくデータが更新できたことを確認できると思います。なお、SQL Server 2012/SQL Server 2014 で同じステートメントを実行した場合には、次のようにエラーが返されています。

```
-- SQL Server 2012/2014 では更新できない (読み取り専用)
UPDATE t1_nccsi
SET c = 'zzzz'
WHERE a = 1
```

100 % <

メッセージ

メッセージ 35330、レベル 15、状態 1、行 41
 UPDATE ステートメントが失敗しました。
 非クラスター化 columnstore インデックスを含むテーブルではデータを更新できません。
 UPDATE ステートメントを実行する前にいったん columnstore インデックスを無効にし、
 UPDATE の完了後に再構築することを確認してください。

2. 次に、更新したデータを確認してみます。

```
SELECT * FROM t1_nccsi
WHERE a = 1
```

```
-- 更新したデータの確認
SELECT * FROM t1_nccsi
WHERE a = 1
```

100 % <

結果

	a	b	c	d
1	1	1	zzzz	dummy2

3. 次に、**INSERT** ステートメントを実行して、データを追加してみます。

```
INSERT INTO t1_nccsi (b, c, d)
VALUES (222, 'aaa', 'aaa')
```

```
-- データの追加
INSERT INTO t1_nccsi (b, c, d)
VALUES (222, 'aaa', 'aaa')
```

100 % <

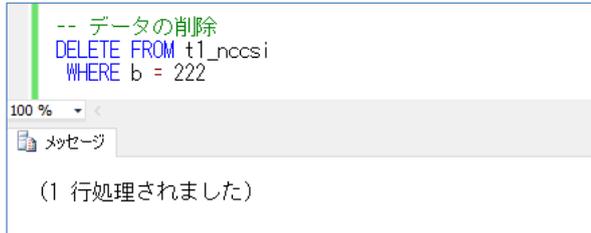
メッセージ

(1 行処理されました)

データの追加も問題なくできることを確認できます。

4. 次に、**DELETE** ステートメントを実行して、データを削除してみます。

```
DELETE FROM t1_nccsi
WHERE b = 222
```



```
-- データの削除
DELETE FROM t1_nccsi
WHERE b = 222
```

100 %

メッセージ

(1 行処理されました)

データの削除も問題なくできることを確認できます。

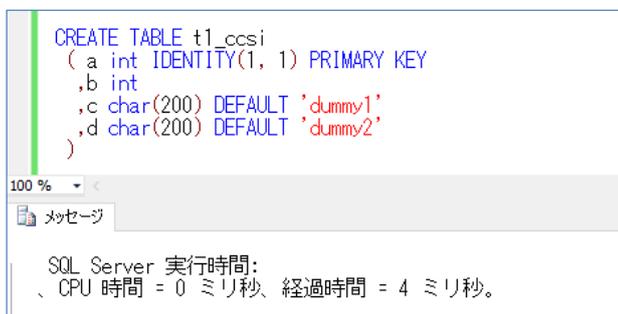
このように、SQL Server 2016 からは、非クラスター化列ストア インデックスを作成しても、データを更新/追加/削除できるようになったので、**Operational Analytics** を実現 (OLTP とデータ分析の両立) できるようになりました。

➡ クラスター化列ストア インデックスの作成、性能チェック

次に、**クラスター化列ストア インデックス**を作成した場合の性能を調べてみましょう。

1. まずは、クラスター化列ストア インデックスを試すためのテーブルを「**t1_ccsi**」という名前で作成します (**t1**、**t1_nccsi** テーブルと全く同じ構造にします)。

```
CREATE TABLE t1_ccsi
( a int IDENTITY(1, 1) PRIMARY KEY
, b int
, c char(200) DEFAULT 'dummy1'
, d char(200) DEFAULT 'dummy2'
)
```



```
CREATE TABLE t1_ccsi
( a int IDENTITY(1, 1) PRIMARY KEY
, b int
, c char(200) DEFAULT 'dummy1'
, d char(200) DEFAULT 'dummy2'
)
```

100 %

メッセージ

SQL Server 実行時間:
、CPU 時間 = 0 ミリ秒、経過時間 = 4 ミリ秒。

2. 次に、**1 千万件**分のデータを **INSERT** しますが、**t1** テーブルと同じデータにするために、次のように **INSERT .. SELECT** ステートメントを実行します。

```
INSERT INTO t1_ccsi (b, c, d)
SELECT b, c, d FROM t1
```

```
-- 1千万件のコピー
INSERT INTO t1_ccsi (b, c, d)
SELECT b, c, d FROM t1
```

100 %

メッセージ

SQL Server の構文解析とコンパイルの時間:
CPU 時間 = 0 ミリ秒、経過時間 = 1 ミリ秒。

SQL Server 実行時間:
CPU 時間 = [] ミリ秒、経過時間 = [] ミリ秒。

(10000000 行処理されました)

3. 次に、**CREATE CLUSTERED COLUMNSTORE INDEX** ステートメントを利用して、**クラスター化列ストア インデックス**を作成します(非クラスター化列ストア インデックスでは、**ON 句**で列名を指定しましたが、クラスター化列ストア インデックスの場合は、すべての列を含める必要があるため、**ON 句**ではテーブル名を指定するだけになります)。

```
CREATE CLUSTERED COLUMNSTORE INDEX idx1
ON t1_ccsi
```

```
-- クラスター化列ストア インデックスの作成
CREATE CLUSTERED COLUMNSTORE INDEX idx1
ON t1_ccsi
```

100 %

メッセージ

SQL Server の構文解析とコンパイルの時間:
CPU 時間 = 0 ミリ秒、経過時間 = 0 ミリ秒。
メッセージ 35372、レベル 16、状態 3、行 164
テーブル 't1_ccsi' には複数のクラスター化インデックスを作成できません。
'with (drop_existing = on)' オプションを使用して新しいクラスター化インデックスを作成することを検討してください。

しかし、結果はエラーになります。クラスター化列ストア インデックスを作成するには、テーブル内にクラスター化インデックスがないことが条件になるためです。**t1_ccsi** テーブルには、**PRIMARY KEY** 制約を設定しているため、自動的に b-tree のクラスター化インデックスが作成されているので、このエラーが返っています。これを回避するには、**クラスター化インデックスを非クラスター化インデックスに変更**するか、**PRIMARY KEY 制約を削除**するようにします。

なお、SQL Server 2014 では、クラスター化列ストア インデックスで PRIMARY KEY 制約がサポートされていなかったため、PRIMARY KEY 制約を削除する方法しかとれませんでした。SQL Server 2016 からは、PRIMARY KEY 制約がサポートされるようになったため、制約は残しつつ、インデックスのみを変更するという方法がとれるようになりました。

4. 次に、PRIMARY KEY 制約で作成されたクラスター化インデックスを**非クラスター化インデックス**に変更するために、次のように **sp_help** システム ストアド プロシージャを実行して、PRIMARY KEY 制約の名前(インデックス名)を確認します。

```
EXEC sp_help 't1_ccsi'
```

```
-- PRIMARY KEY 制約の名前 (インデックス名) の確認
EXEC sp_help 't1_ccsi'
```

Name	Owner	Type	Created_datetime
t1_ccsi	dbo	user table	2016-01-29 01:18:13.680

Column_name	Type	Computed	Length	Prec	Scale	Nullable	TrimTrailingBlanks	FixedLenNullInS
a	int	no	4	10	0	no	(n/a)	(n/a)
b	int	no	4	10	0	yes	(n/a)	(n/a)
c	char	no	200			yes	no	yes
d	char	no	200			yes	no	yes

Identity	Seed	Increment	Not For Replication
a	1	1	0

RowGuidCol
No rowguidcol column defined.

Data_located_on_filegroup
PRIMARY

index_name	index_description	index_keys
PK__t1_ccsi__3BD0198EBD156D09	clustered, unique, primary key located on PRIMARY	a

index_name に表示される「PK__t1_ccsi__～」が PRIMARY KEY 制約の名前（およびインデックスの名前）になるので、これをコピーします。

5. コピーした名前を、次のように **ALTER TABLE** ステートメントの **DROP CONSTRAINT** の部分に貼り付けて、**PRIMARY KEY 制約**をいったん削除します。

```
ALTER TABLE t1_ccsi
DROP CONSTRAINT PK__t1_ccsi__~
```

```
-- PRIMARY KEY 制約をいったん削除
ALTER TABLE t1_ccsi
DROP CONSTRAINT PK__t1_ccsi__3BD0198EBD156D09
```

SQL Server の構文解析とコンパイルの時間:
CPU 時間 = 0 ミリ秒、経過時間 = 1 ミリ秒。

SQL Server 実行時間:
、CPU 時間 = ミリ秒、経過時間 = ミリ秒。

sp_help で調べたインデックス名に置換する

6. PRIMARY KEY 制約の削除が完了したら、今度は **ALTER TABLE** ステートメントの **ADD CONSTRAINT** で **PRIMARY KEY NONCLUSTERED** を指定して、非クラスター化インデックスとして PRIMARY KEY 制約を作成します（制約の名前は **pk_t1_ccsi** にします）。

```
ALTER TABLE t1_ccsi
ADD CONSTRAINT pk_t1_ccsi PRIMARY KEY NONCLUSTERED(a)
```

```
-- PRIMARY KEY 制約を非クラスター化インデックスで作成
ALTER TABLE t1_ccsi
ADD CONSTRAINT pk_t1_ccsi PRIMARY KEY NONCLUSTERED(a)
```

SQL Server の構文解析とコンパイルの時間:
CPU 時間 = 0 ミリ秒、経過時間 = 0 ミリ秒。

SQL Server 実行時間:
、CPU 時間 = ミリ秒、経過時間 = ミリ秒。

7. PRIMARY KEY 制約の再作成後は、もう一度、**CREATE CLUSTERED COLUMNSTORE INDEX** ステートメントを実行して、クラスター化列ストア インデックスを作成します。

```
CREATE CLUSTERED COLUMNSTORE INDEX idx1
ON t1_ccsi
```

```
-- クラスター化列ストア インデックスの作成
CREATE CLUSTERED COLUMNSTORE INDEX idx1
ON t1_ccsi
```

100 %

メッセージ

SQL Server の構文解析とコンパイルの時間:
CPU 時間 = 0 ミリ秒、経過時間 = 0 ミリ秒。

SQL Server 実行時間:
、CPU 時間 = [] ミリ秒、経過時間 = [] ミリ秒。

今度は、作成が完了することを確認できます。

なお、SQL Server 2014 のときには、上と同じように PRIMARY KEY 制約が存在している場合には、次のようにクラスター化列ストア インデックスを作成することができませんでした (SQL Server 2014 では、PRIMARY KEY 制約を削除しなければなりません)。

```
-- 2014 クラスター化列ストア インデックスの作成
CREATE CLUSTERED COLUMNSTORE INDEX idx1
ON t1_ccsi
```

100 %

メッセージ

SQL Server の構文解析とコンパイルの時間:
CPU 時間 = 0 ミリ秒、経過時間 = 0 ミリ秒。
メッセージ 35304、レベル 16、状態 1、行 290
CREATE INDEX ステートメントが失敗しました。
非クラスター化 columnstore インデックスのあるテーブルではクラスター化インデックスを作成できません。
非クラスター化インデックスをすべて削除して、やり直してください。

8. クラスター化列ストア インデックスの作成が完了したら、同じ **SELECT** ステートメントを実行して、**実行時間**を確認します。

```
SELECT b, COUNT(*) FROM t1_ccsi
GROUP BY b
```

```
-- 実行時間をチェックする
SELECT b, COUNT(*) FROM t1_ccsi
GROUP BY b
```

100 %

結果

SQL Server の構文解析とコンパイルの時間:
CPU 時間 = 0 ミリ秒、経過時間 = 0 ミリ秒。

(1001 行処理されました)

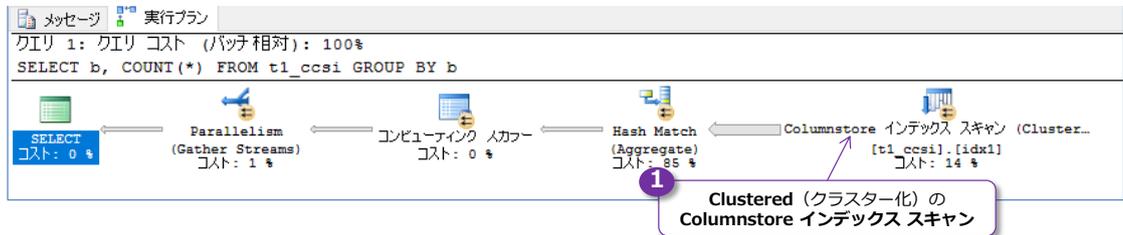
SQL Server 実行時間:
、CPU 時間 = 0 ミリ秒、経過時間 = [] ミリ秒。

1 実行時間を確認する

1 回目の実行はコンパイル時間が加算されるので、2 回、3 回、4 回と実行して、そのときの実行時間を確認してみてください。経過時間が桁違いに小さくなっていることを確認できると思います (弊社環境では、非クラスター化列ストア インデックスのときよりも速く実行でき

ることを確認しています。b-tree の非クラスター化インデックスのときと比べると、約 **181 倍** 速く実行)。

9. 次に、SELECT ステートメントを選択して、ツールバーの **[推奨実行プラン]** ボタンをクリックし、実行プランも確認しておきます。



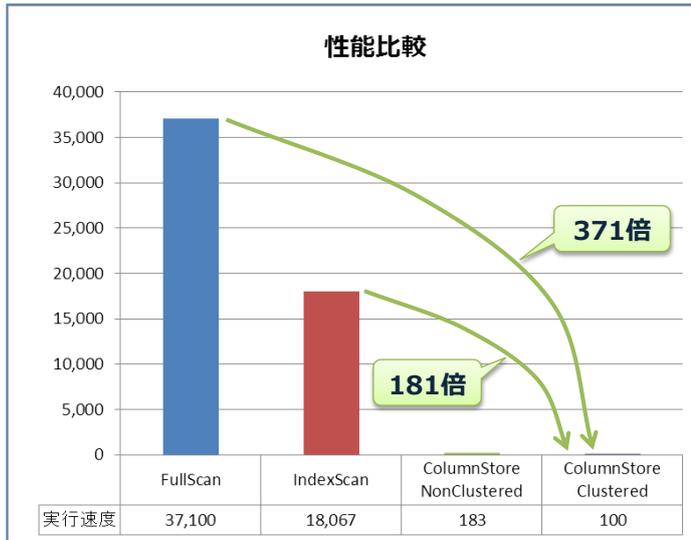
Columnstore インデックス スキャン (Clustered) と表示されて、クラスター化列ストアインデックスのスキャンで実行されていることを確認できると思います。

10. 次に、テーブル サイズとインデックス サイズも確認します。

項目	値
データ領域	31.602 MB
インデックス領域	212.672 MB

データ領域 (テーブル サイズ) が約 **32MB** となっていて、これが**クラスター化列ストア インデックス**のサイズです。**インデックス領域** (インデックス サイズ) が約 **213MB** になっているのは、PRIMARY KEY 制約に作成した b-tree の非クラスター化インデックスの分になります。なお、非クラスター化列ストア インデックスのときは、データ領域が約 **4GB**、インデックス領域が約 **49MB** でした。

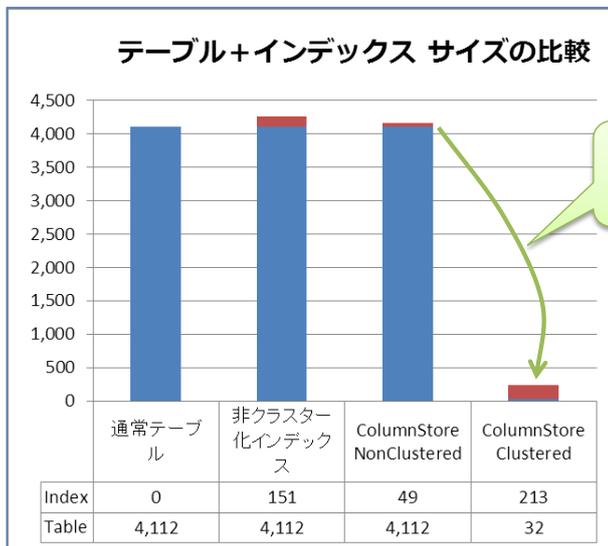
ここまでの結果をまとめると、次のようになります。



FullScan は Clustered Index Scan が実行されたとき、**IndexScan** は非クラスター化インデックスを作成/利用したとき、**ColumnStore NonClustered** は非クラスター化列ストア インデックスを作成/利用したとき、**ColumnStore Clustered** はクラスター化列ストア インデックスを作成/利用したときの実行速度です（相対値）。

クラスター化列ストア インデックスを作成することで、フル スキャンに比べて **371 倍**、インデックス スキャンに比べて **181 倍**もの性能向上を確認できます。このように、集計処理に強いのが列ストア インデックスです。

テーブルおよびインデックス サイズをまとめると、次のようになります。

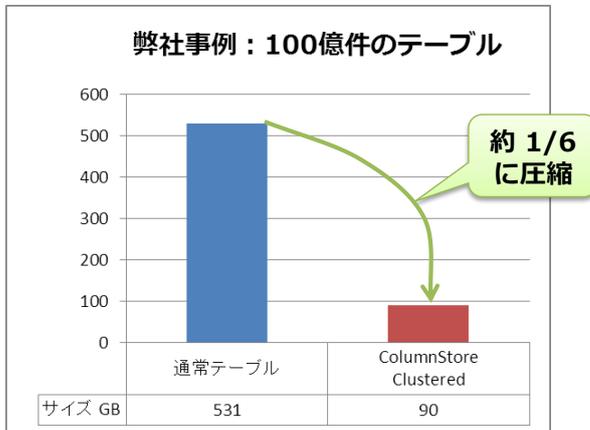


通常テーブルでは、約 **4GB** 分のテーブル領域を利用しているのに対して、クラスター化列ストア インデックスでは、テーブル データそのものが列ストア インデックスになるので、わずか **32MB** のサイズに小さくすることができます（インデックス領域の **213MB** は、PRIMARY KEY 制約を設定した b-tree の非クラスター化インデックスの分です）。

ただし、今回は、**c** および **d** 列には、まったく同じデータ (**dummy1**) を格納しているので、圧縮率が極端に高くなっていることを差し引く必要があることにも注意してください。

➡ 弊社のお客様データでの圧縮例（100 億件の DWH で 1/6 に）

弊社のお客様のデータ（100 億件の DWH）で、クラスター化列ストア インデックスを作成した場合の圧縮効果は、次のようになりました。



通常テーブルが **531GB** であるところを、**90GB** へと、約 **1/6** のサイズにすることができました。このように、クラスター化列ストア インデックスは、**ディスク領域を削減できる**メリットもあります。また、ディスクから読み取るサイズが小さくなる分、性能向上にも貢献します（圧縮および解凍に伴う CPU パワーとのトレードオフも存在します）。

3.4 列ストア インデックスでのデータ更新とインデックスの再構築

次に、クラスター化列ストア インデックスに対して、データの更新を行った場合の性能を確認してみましょう。

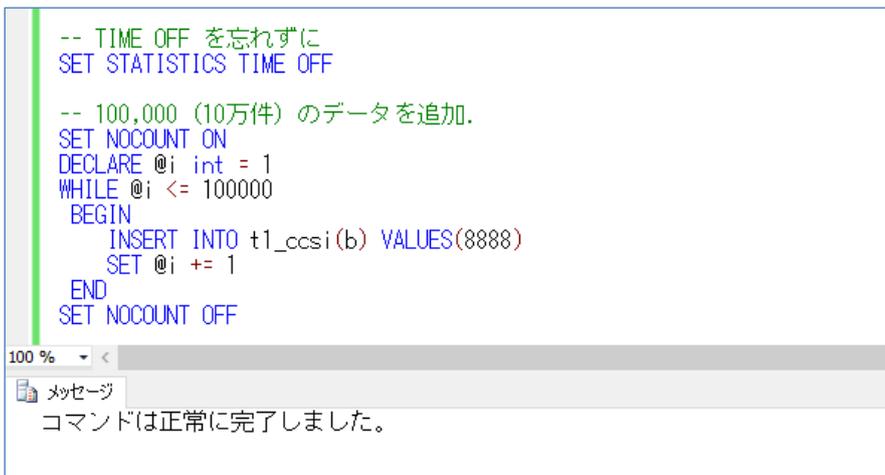
➡ 10 万件のデータの INSERT

1. まずは、**SET STATISTICS TIME** コマンドを **OFF** へ設定して、実行時間の記録を停止しておきます（これは、次の手順の **WHILE** ループを実行前に、必ず実行しておいてください）。

```
SET STATISTICS TIME OFF
```

2. 次に、**10 万件のデータ**を **INSERT** します。

```
SET NOCOUNT ON
DECLARE @i int = 1
WHILE @i <= 100000
BEGIN
    INSERT INTO t1_ccsi (b) VALUES (8888)
    SET @i += 1
END
SET NOCOUNT OFF
```



```
-- TIME OFF を忘れずに
SET STATISTICS TIME OFF

-- 100,000 (10万件) のデータを追加.
SET NOCOUNT ON
DECLARE @i int = 1
WHILE @i <= 100000
BEGIN
    INSERT INTO t1_ccsi (b) VALUES (8888)
    SET @i += 1
END
SET NOCOUNT OFF
```

100 % < 100000
メッセージ
コマンドは正常に完了しました。

何の問題もなく、データを INSERT できることを確認できたと思います。

なお、もしこのスクリプトを実行する前に、**SET STATISTICS TIME** を **OFF** にし忘れてしまった場合は、10 万件分の結果メッセージが出力されることになり、実行時間が非常に長くなってしまいますので注意してください（出力メッセージ数が多いと、Management Studio が終了してしまう場合もあります）。

3. 次に、**SET STATISTICS TIME ON** を実行して、前の Step と同じ **SELECT** ステートメント（**GROUP BY** 演算）の実行時間を計測してみます。

```
SET STATISTICS TIME ON
```

```
SELECT b, COUNT(*) FROM t1_ccsi
GROUP BY b
```

```
SET STATISTICS TIME ON
-- 実行時間をチェックする
SELECT b, COUNT(*) FROM t1_ccsi
GROUP BY b
```

100 %

結果 メッセージ

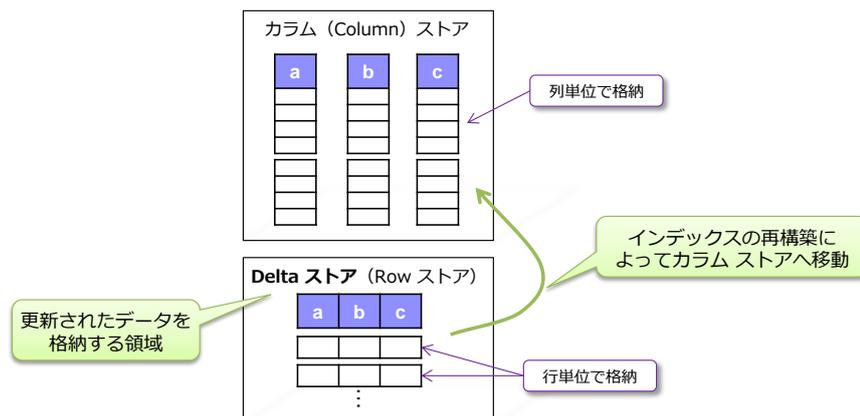
(1002 行処理されました)

SQL Server 実行時間:
、CPU 時間 = 31 ミリ秒、経過時間 = ミリ秒。

1 実行時間を確認する

10 万件追加しただけにも関わらず、実行時間が長くなっていることを確認できると思います。

これは、クラスター化列ストア インデックスでは、更新されたデータは、「**Delta ストア**」と呼ばれる、更新データの格納用の領域に格納されているためです。



クラスター化列ストア インデックスでは、カラム（列）単位でデータを格納しているカラムストア領域と、更新データのための Delta ストア（差分領域）があり、Delta ストアにデータがある場合は、読み取り時のオーバーヘッドが発生します。これを回避するには、**インデックスを再構築**するようにします。再構築を行うことで、Delta ストア内の差分データをカラムストアへ移動させることができるからです。

➡ クラスター化列ストア インデックスの再構築（REBUILD）

それでは、インデックスを再構築してみましょう。

1. クラスター化列ストア インデックスの再構築は、通常のインデックスを再構築するのと同様、**ALTER INDEX** ステートメントを利用して、次のように実行します。

```
ALTER INDEX idx1
ON t1_ccsi REBUILD
```

```
-- インデックスの再構築
ALTER INDEX idx1
ON t1_ccsi REBUILD
```

100 %

メッセージ

SQL Server の構文解析とコンパイルの時間:
CPU 時間 = 0 ミリ秒、経過時間 = 0 ミリ秒。

SQL Server 実行時間:
、CPU 時間 = [] ミリ秒、経過時間 = [] ミリ秒。

2. 再構築が完了したら、もう一度 SELECT ステートメントを実行してみましょう。

```
SELECT b, COUNT(*) FROM t1_ccsi
GROUP BY b
```

```
-- 実行時間をチェックする
SELECT b, COUNT(*) FROM t1_ccsi
GROUP BY b
```

100 %

結果 メッセージ

SQL Server の構文解析とコンパイルの時間:
CPU 時間 = 0 ミリ秒、経過時間 = 0 ミリ秒。

(1002 行処理されました)

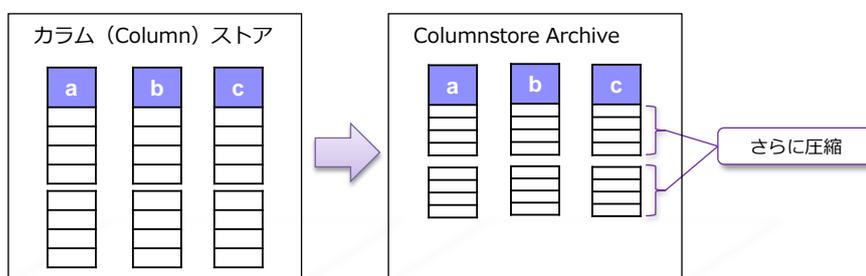
SQL Server 実行時間:
、CPU 時間 = 0 ミリ秒、経過時間 = [] ミリ秒。

1 実行時間を確認する

今度は、1,000 万件のときとほとんど同じスピードで検索できたことを確認できると思います。このように、列ストア インデックスでは、データの更新を行うことができますが、より良いパフォーマンスを保つには、定期的にインデックスを再構築することが重要になります。

Note : COLUMNSTORE_ARCHIVE (列ストア アーカイブ)

クラスター化列ストア インデックスでは、さらに圧縮率を高めた、「COLUMNSTORE_ARCHIVE」(列ストア アーカイブ) というモードもあります。



さらに圧縮をすることで、ディスク使用領域をより小さくできることがメリットです (圧縮率を高くする分、さらに CPU パワーを余分に使うことになるので、それとのトレードオフになります)。

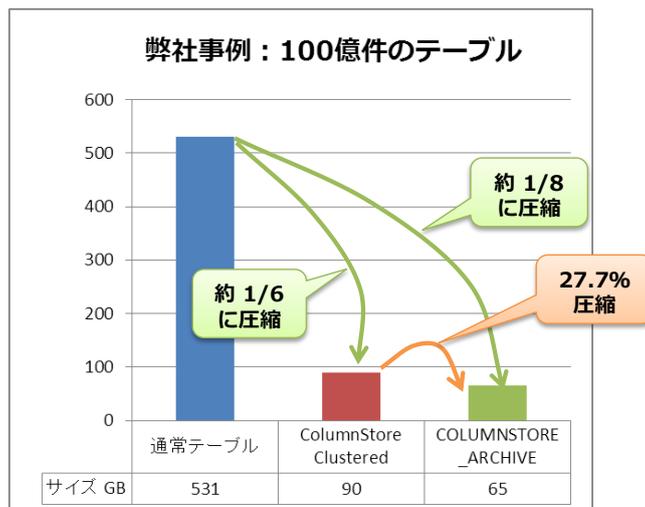
データパーティションを利用して、複数のパーティションを作成している場合は、特定のパーティションのみを **COLUMNSTORE_ARCHIVE** へ設定することもできるので、アクセスされ

る頻度の低いパーティションのみを **COLUMNSTORE_ARCHIVE** へ設定するという使い方もできます。

COLUMNSTORE_ARCHIVE を設定するには、次のように **ALTER INDEX** ステートメントでの **REBUILD** 時に **DATA_COMPRESSION** を指定します。

```
ALTER INDEX idx1
ON t1_ccsi REBUILD
WITH (DATA_COMPRESSION = COLUMNSTORE_ARCHIVE)
```

前掲の弊社のお客様（100 億件の DWH）で、これを設定した場合には、次のような効果がありました。



COLUMNSTORE_ARCHIVE を設定することで、通常テーブルが **531GB** であるところを、**65GB** へと、約 **1/8** のサイズにすることができました。クラスター化列ストア インデックスの **90GB** と比べても、**27.7%** も圧縮することができています。

このように、さらに圧縮率を高めることができるのが、**COLUMNSTORE_ARCHIVE** のメリットです（圧縮率を高くする分、CPU パワーを余計に利用することになるので、それとのトレードオフになります）。

なお、**COLUMNSTORE_ARCHIVE** へ設定したテーブル（やパーティション）をもとに戻したい場合（通常の列ストアに戻したい場合）には、次のように **ALTER INDEX** ステートメントを実行します（**DATA_COMPRESSION** で **COLUMNSTORE** を指定します）。

```
ALTER INDEX idx1
ON t1_ccsi REBUILD
WITH (DATA_COMPRESSION = COLUMNSTORE)
```

3.5 BULK INSERT で 10 万件分のデータを一括インポート

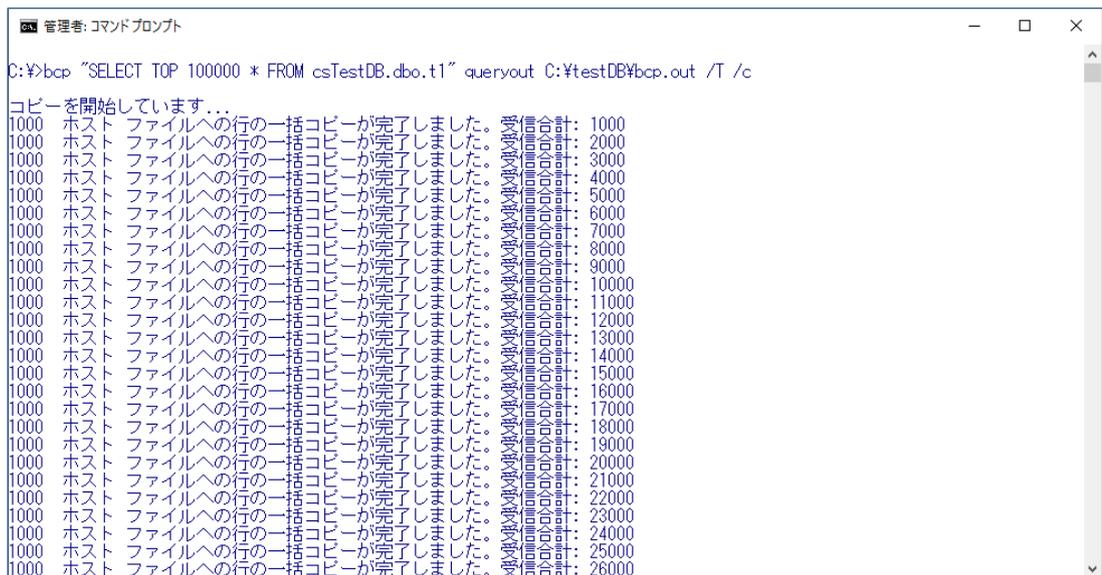
次に、**BULK INSERT** ステートメントを利用して、10 万件分のデータを一括インポートするときの性能をチェックしてみましょう。

➡ bcp コマンドで 10 万件分のテキスト ファイルの作成

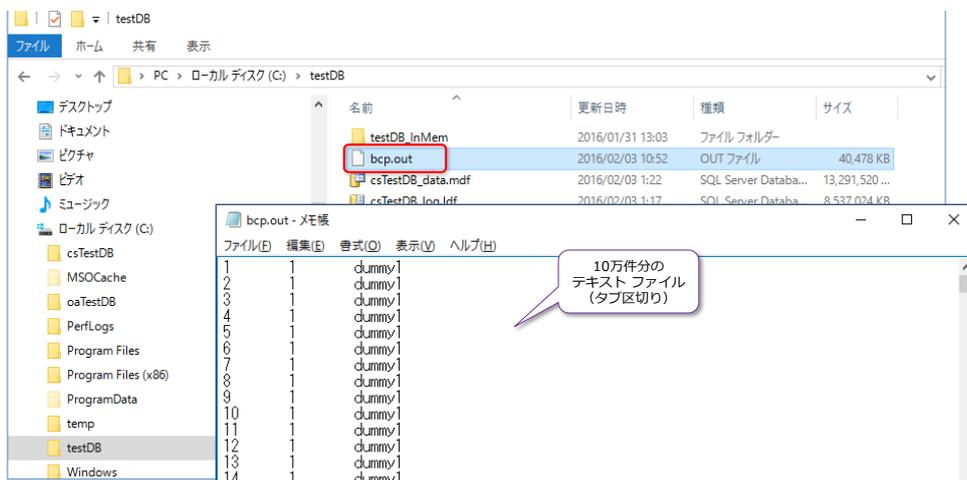
まずは、**BULK INSERT** ステートメントでインポートするためのテキスト ファイルを、**bcp** コマンドで作成しておきます。

1. **bcp** コマンドを実行するには、コマンド プロンプトを起動して、次のように記述します。

```
bcp "SELECT TOP 100000 * FROM csTestDB.dbo.t1" queryout C:¥testDB¥bcp.out /T /c
```



t1 テーブルから **10 万件分**を取得する **SELECT** ステートメントを記述して、**queryout** で **C:¥testDB** フォルダに **bcp.out** という名前のテキスト ファイルとして取得しています。実行が完了すると、次のようなテキスト ファイルが作成されます (タブ区切りのファイル)。



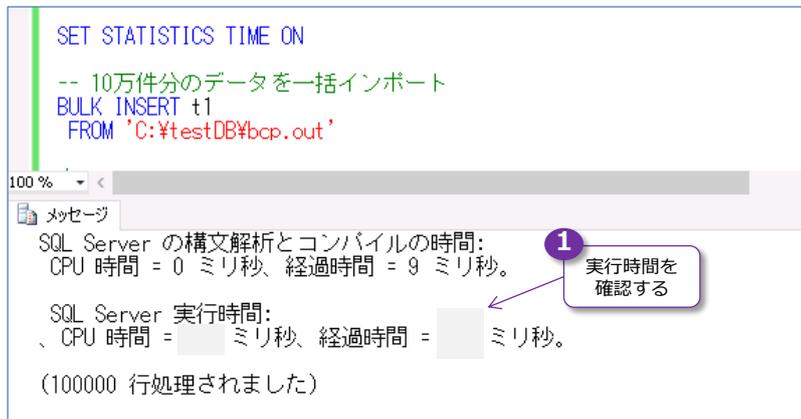
➔ BULK INSERT ステートメントで 10 万件を一括インポート

次に、**BULK INSERT** ステートメントを利用して、10 万件分のデータを一括インポートしてみましょう。

1. まずは、**t1** テーブル (b-tree の非クラスター化インデックスを付与したテーブル) にインポートしてみましょう (実行時間は **SET STATISTICS TIME ON** で確認します)。

```
SET STATISTICS TIME ON

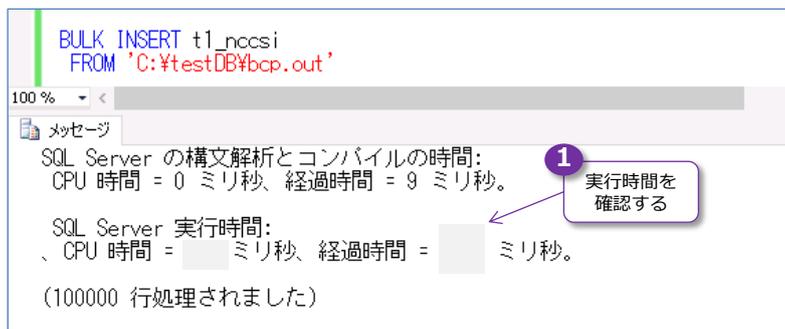
-- 10万件分のデータを一括インポート
BULK INSERT t1
FROM 'C:\testDB\bcp.out'
```



実行が完了したら、さらに 2 回、3 回と同じステートメントを実行して、そのときの実行時間も確認してみてください。

2. 次に、**t1_nccsi** テーブル (非クラスター化列ストア インデックスを作成しているテーブル) にインポートしてみましょう。

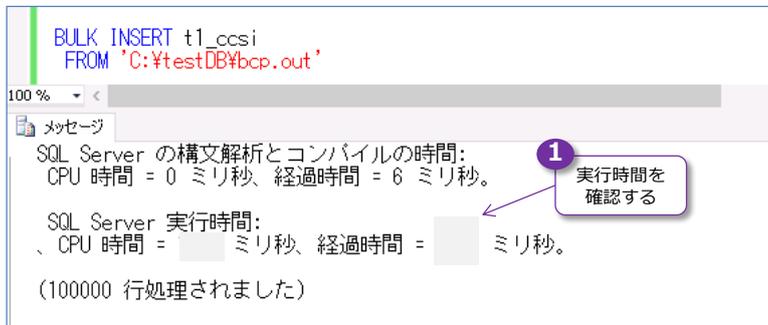
```
-- 非クラスター化列ストア インデックスのテーブルにインポート
BULK INSERT t1_nccsi
FROM 'C:\testDB\bcp.out'
```



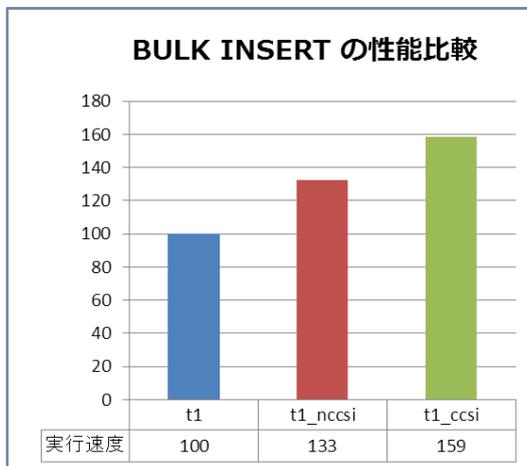
3. 次に、**t1_ccsi** テーブル (クラスター化列ストア インデックスを作成しているテーブル) にインポートしてみましょう。

-- クラスター化列ストア インデックスのテーブルにインポート

```
BULK INSERT t1_ccsi
FROM 'C:\testDB\bcp.out'
```



ここまでの結果をまとめると、次のようになります。



テストに利用したハードウェア環境

- ・ Core i7 6700K 4GHz
- ・ メモリ 64GB
- ・ SSD Crucial MX200 1TB

性能結果は相対値

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、通常テーブル (t1) の値を 100 とした場合の相対値で表しています。

非クラスター化列ストア インデックスを作成したテーブル (**t1_nccsi**) は **33%**、クラスター化列ストア インデックスを作成したテーブル (**t1_ccsi**) は **59%** のオーバーヘッドになりました。クラスター化列ストア インデックスを作成したテーブルには、**PRIMARY KEY 制約**があるの、b-tree の非クラスター化インデックスも作成されているので、もし、次のように **PRIMARY KEY 制約**を外している場合は、オーバーヘッドを軽減できます。

-- PK を外したクラスター化列ストア インデックスのテーブル

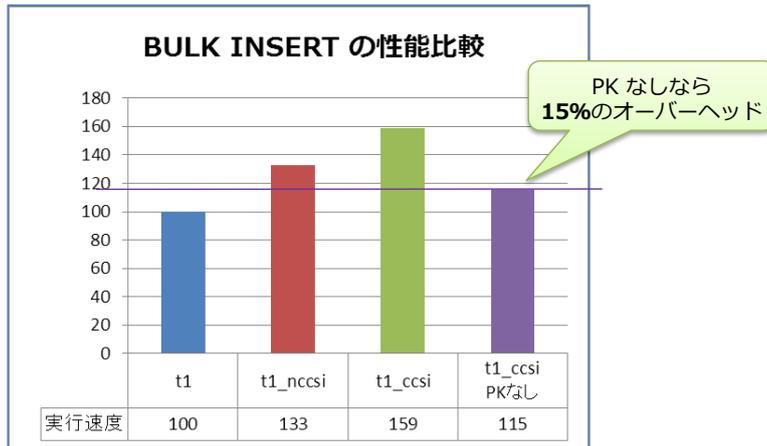
```
CREATE TABLE t1_ccsi_notPK
( a int IDENTITY(1, 1)
, b int
, c char(200) DEFAULT 'dummy1'
, d char(200) DEFAULT 'dummy2'
, INDEX ccsi_t1_ccsi_notPK CLUSTERED COLUMNSTORE )
```

-- 1千万件のコピー (.*他のテーブルと条件を合わせるため)

```
INSERT INTO t1_ccsi_notPK (b, c, d) SELECT b, c, d FROM t1
```

-- 10万件の一括インポート

```
BULK INSERT t1_ccsi_notPK
FROM 'C:\testDB\bcp.out'
```



なお、t1 テーブルでも、PRIMARY KEY 制約を外したり、b 列に対する b-tree の非クラスター化インデックスを削除することで（ヒープにすることで）、BULK INSERT の性能を向上させることができます。このあたりのインデックスと一括インポートの考え方（インデックスのオーバーヘッドへの対処方法）は、従来どおりです。

➡ インポート後はインデックスの再構築

一括インポートを行った後は、集計クエリの性能が低下することになるので、前の項で紹介したインデックスの再構築（REBUILD）を行っておくのがお勧めになります。

1. これを確認するには、これまでと同じ **SELECT** ステートメント（**GROUP BY** 演算）を実行します。

```
SET STATISTICS TIME ON

SELECT b, COUNT(*) FROM t1_ccsi
GROUP BY b
```

```
-- 実行時間をチェックする
SELECT b, COUNT(*) FROM t1_ccsi
GROUP BY b
```

100 %

結果 メッセージ

(1002 行処理されました)

SQL Server 実行時間:
、CPU 時間 = 15 ミリ秒、経過時間 = ミリ秒。

1 実行時間を確認する

実行時間が長くなっていることを確認できると思います。

2. 次に、**ALTER INDEX** ステートメントを利用して、インデックスを再構築します。

```
ALTER INDEX idx1
ON t1_ccsi REBUILD
```

```

-- インデックスの再構築
ALTER INDEX idx1
ON t1_ccsi REBUILD

```

100% <

メッセージ

SQL Server の構文解析とコンパイルの時間:
CPU 時間 = 0 ミリ秒、経過時間 = 0 ミリ秒。

SQL Server 実行時間:
、CPU 時間 = [] ミリ秒、経過時間 = [] ミリ秒。

3. 再構築が完了したら、もう一度 SELECT ステートメントを実行します。

```

SELECT b, COUNT(*) FROM t1_ccsi
GROUP BY b

```

```

-- 実行時間をチェックする
SELECT b, COUNT(*) FROM t1_ccsi
GROUP BY b

```

100% <

結果 メッセージ

SQL Server の構文解析とコンパイルの時間:
CPU 時間 = 0 ミリ秒、経過時間 = 0 ミリ秒。

(1002 行処理されました)

SQL Server 実行時間:
、CPU 時間 = 0 ミリ秒、経過時間 = [] ミリ秒。

1 実行時間を確認する

今度は、最初のスピードに戻ったことを確認できると思います。

3.6 インメモリ OLTP の利用

次に、インメモリ OLTP を利用して、完全なインメモリでの Operational Analytics を確認してみましょう。

➡ Let's Try

1. まずは、インメモリ OLTP を利用するために、データベースに**ファイル グループ**を追加します。

```
-- ファイルグループの追加
ALTER DATABASE csTestDB
ADD FILEGROUP fg1
CONTAINS MEMORY_OPTIMIZED_DATA

ALTER DATABASE csTestDB
ADD FILE ( NAME = csTestDB_InMem,
          FILENAME = 'C:¥testDB¥csTestDB_InMem' )
TO FILEGROUP fg1
```

2. 次に、**CREATE TABLE** ステートメントを利用して**メモリ最適化テーブル**を作成しますが、このときに**クラスター化列ストア インデックス**を作成するようにします。

```
-- メモリ最適化テーブルの作成
USE csTestDB
CREATE TABLE t1_InMem_ccsi
( a int IDENTITY(1, 1)
  PRIMARY KEY NONCLUSTERED
  HASH WITH (BUCKET_COUNT = 10000000)
, b int
, c char(200) DEFAULT 'dummy1'
, d char(200) DEFAULT 'dummy2'
, INDEX idx_InMem_ccsi CLUSTERED COLUMNSTORE
) WITH ( MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA )
```

t1 テーブルと同じ構成で、「**t1_InMem_ccsi**」という名前で作成します。**WITH** 句では **MEMORY_OPTIMIZED=ON** を指定することでメモリ最適化テーブルにして、**a** 列には非クラスター化の **HASH** インデックス (**BUCKET_COUNT** は 1,000 万に設定) を作成しています。また、**INDEX** キーワードに続けてインデックス名 (**idx_InMem_ccsi**) を指定して、**CLUSTERED COLUMNSTORE** を指定することで、クラスター化列ストア インデックスを作成しています。インメモリ OLTP のクラスター化列ストア インデックスでは、**DURABILITY** で **SCHEMA_AND_DATA** (データの永続化有り) が必須になるので、これも指定しています。

3. 次に、**INSERT..SELECT** を利用して、**t1** テーブルから **1 千万件**分のデータをコピーします。

```
-- 1千万件のデータをコピー
INSERT INTO t1_InMem_ccsi (b, c, d)
SELECT b, c, d FROM t1
```

```
-- 1千万件のコピー
INSERT INTO t1_InMem_ccsi (b, c, d)
SELECT b, c, d FROM t1
```

00 %

メッセージ

(10000000 行処理されました)

4. データのコピーが完了したら、これまでと同じ **SELECT** ステートメント (**GROUP BY** 演算) を実行します。

```
SET STATISTICS TIME ON

SELECT b, COUNT(*) FROM t1_InMem_ccsi
GROUP BY b
```

```
SET STATISTICS TIME ON

-- 実行時間をチェックする
SELECT b, COUNT(*) FROM t1_InMem_ccsi
GROUP BY b
```

100 %

結果 メッセージ

SQL Server の構文解析とコンパイルの時間:
CPU 時間 = 0 ミリ秒、経過時間 = 0 ミリ秒。

(1001 行処理されました)

SQL Server 実行時間:
、CPU 時間 = 0 ミリ秒、経過時間 = ミリ秒。

1 実行時間を確認する

弊社環境では、非クラスター化列ストア インデックス (NCCSI) を作成したときとほとんど同じスピードで結果が返ることを確認できました。

➡ 性能が確認できない場合の対処方法

もし、性能効果を確認できない場合は、次の手順を試してみてください。

1. まずは、**dm_db_column_store_row_group_physical_stats** 動的管理ビュー (DMV) を利用して、データの状態を確認します。

```
-- データの格納状態を確認
SELECT OBJECT_NAME(object_id), *
FROM sys.dm_db_column_store_row_group_physical_stats
WHERE OBJECT_NAME(object_id) = 't1_InMem_ccsi'
```

```
-- データの格納状態を確認
SELECT OBJECT_NAME(object_id), *
FROM sys.dm_db_column_store_row_group_physical_stats
WHERE OBJECT_NAME(object_id) = 't1_InMem_ccsi'
```

(列名なし)	object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_desc	total_rows
1	t1_InMem_ccsi	629577281	1	1	-1	NULL	OPEN	10000000

この DMV は、列ストア インデックスの **Row Group** (列ストアを内部的に分割する単位で、通常は 100 万件ごとに 1 つの Row Group が作成される) を確認できるものです。インメモリ OLTP では、定期的に Row Group の再作成(圧縮/COMPRESS)が行われるのですが、これがまだ実行されていない場合には、[state_desc] が「OPEN」と表示されます。ここで、**OPEN** と表示される場合は、次のように「sp_memory_optimized_cs_migration」ストアード プロシージャを実行して、強制的に圧縮を実行するようにします。

- 「sp_memory_optimized_cs_migration」ストアード プロシージャの実行は、次のように OBJECT_ID を指定して行えます。

```
-- 強制的に圧縮を実行
DECLARE @objid int = OBJECT_ID('t1_InMem_ccsi')
EXEC sp_memory_optimized_cs_migration @objid
```

- 実行後、もう一度 dm_db_column_store_row_group_physical_stats 動的管理ビューを参照して、データの状態を確認します。

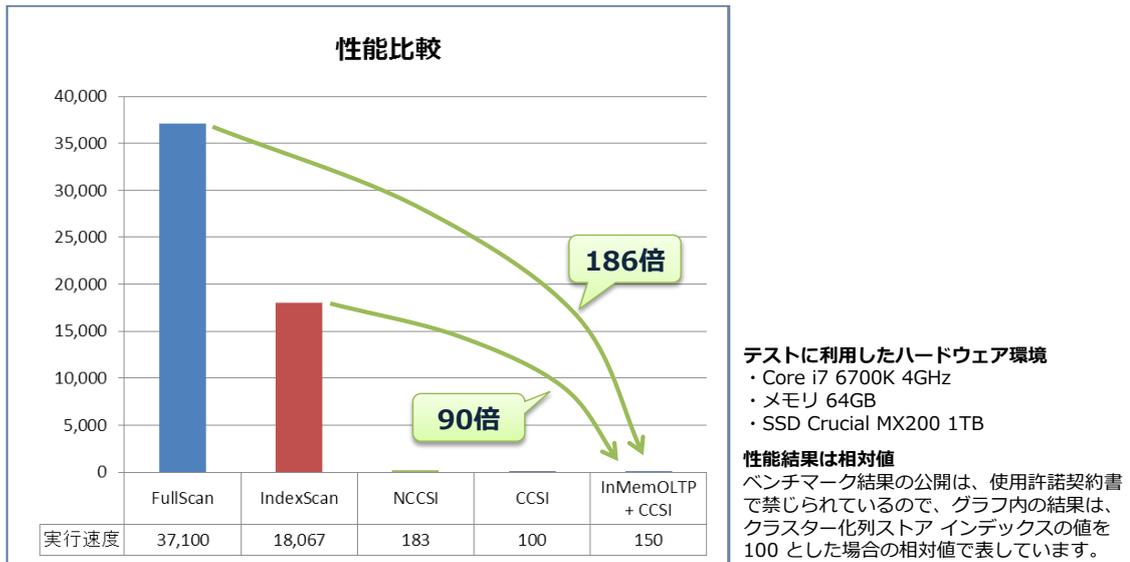
```
SELECT OBJECT_NAME(object_id), *
FROM sys.dm_db_column_store_row_group_physical_stats
WHERE OBJECT_NAME(object_id) = 't1_InMem_ccsi'
```

```
-- データの格納状態を確認
SELECT OBJECT_NAME(object_id), *
FROM sys.dm_db_column_store_row_group_physical_stats
WHERE OBJECT_NAME(object_id) = 't1_InMem_ccsi'
```

(列名なし)	object_id	index_id	partition_number	row_group_id	delta_store_hobt_id	state	state_desc	total_rows
1	t1_InMem_ccsi	629577281	1	1	-1	NULL	OPEN	0
2	t1_InMem_ccsi	629577281	1	3	NULL	3	COMPRESSED	1048576
3	t1_InMem_ccsi	629577281	1	6	NULL	3	COMPRESSED	1048576
4	t1_InMem_ccsi	629577281	1	10	NULL	3	COMPRESSED	432788
5	t1_InMem_ccsi	629577281	1	2	NULL	3	COMPRESSED	1048576
6	t1_InMem_ccsi	629577281	1	8	NULL	3	COMPRESSED	1048576
7	t1_InMem_ccsi	629577281	1	12	NULL	3	COMPRESSED	466248
8	t1_InMem_ccsi	629577281	1	1	NULL	3	COMPRESSED	1048576
9	t1_InMem_ccsi	629577281	1	5	NULL	3	COMPRESSED	1048576
10	t1_InMem_ccsi	629577281	1	9	NULL	3	COMPRESSED	234968
11	t1_InMem_ccsi	629577281	1	4	NULL	3	COMPRESSED	1048576
12	t1_InMem_ccsi	629577281	1	7	NULL	3	COMPRESSED	1048576
13	t1_InMem_ccsi	629577281	1	11	NULL	3	COMPRESSED	477388

COMPRESS と表示された行が追加されれば成功です (だいたい 100 万件ごとに 1 つの Row Group が作成されるので、1 千万件だと 10 個ぐらいの行が返ります)。この状態になったら、再度 **GROUP BY** クエリを実行して、性能を確認してみてください。

弊社環境での性能結果は、次のようになりました。



インメモリ OLTP とクラスター化列ストア インデックス (CCSI) の組み合わせでは、非クラスター化列ストア インデックス (NCCSI) を利用した場合とほとんど同じ性能が出ることを確認できました。

➡ 列ストア インデックスのまとめ

以上のように、列ストア インデックスは、大きな性能向上およびディスク領域の削減を期待できる、大変便利な機能です。圧縮/解凍に伴う CPU パワーとのトレードオフや、更新時のデータを格納するための Delta ストアを利用することによるオーバーヘッドなどがありますが、インデックスを定期的に再構築するなど、うまく活用することで、大きな性能向上を実現することができるので、ぜひ試してみてください。

また、インメモリ OLTP と組み合わせ利用することで、OLTP にも強くすることができる（特に多重度が上がった場合の同時実行性能に強くできる）ので、ぜひ試してみてください。

STEP 4. インメモリ OLTP の基本操作

この STEP では、まだインメモリ OLTP を試したことがない方のために、インメモリ OLTP の基本的な利用方法を説明します。

なお、この Step で説明する内容は、SQL Server 2014 自習書シリーズの「インメモリ OLTP 入門」編とほとんど同じ内容になるので、これを試したことがある方は、読み飛ばしていただいて大丈夫です。

この STEP では、次のことを学習します。

- ✓ インメモリ OLTP の概要
- ✓ インメモリ OLTP の基本操作（メモリ最適化テーブルの作成）
- ✓ ネイティブ コンパイル ストアド プロシージャの作成
- ✓ 同時更新の場合の動作の確認（Write-Write 競合）

4.1 インメモリ OLTP の主な特徴

インメモリ OLTP は、**SQL Server 2014** から提供された、インメモリのデータベース エンジンで、当時の開発コード名は「**Hekaton**」(ヘカトン)と呼ばれていました。このエンジンの一番の特徴は、なんといっても「**インメモリ**」で動作することで、**非常に高速に処理できる**点です。テーブル内のデータを、全てメモリ内に載せることができるので、従来のディスク ベースのデータベース エンジンよりも非常に高速に動作させることができます。

➡ インメモリ OLTP 機能の主な特徴

インメモリ OLTP 機能の主な特徴は、次のとおりです。

- **インメモリで動作**するので、従来のデータベース エンジンよりも**非常に高速**
- **SQL Server のデータベース エンジンと完全に統合**されている。
SQL Server のデータベース エンジンに完全に統合されているので、従来どおりの Transact-SQL ステートメントおよびツールを利用してインメモリの操作が可能。
シームレスに利用できるので、"移行"が容易に行える。新しいハードウェアを購入する必要もない。
AlwaysOn 可用性グループによるデータの保護/可用性向上もサポートされる
- **メモリ内**にのみテーブル データを配置することが可能 (ディスク上にはデータを配置しない)。メモリ内に配置したテーブルは、「**メモリ最適化テーブル**」(Memory-optimized Table) と呼ばれる
- 「**ネイティブ コンパイル ストアド プロシージャ**」(Natively Compiled Stored Procedure) を作成することで、さらに高速な処理が可能
- SQL Server の再起動後も、**データ復旧が可能**な **Durability (永続化)** オプションも用意されている (ディスク上に更新ログを記録して、再起動後にデータの復旧が可能)
- **ブロッキングの最小化** (ロックやラッチ待ちがほとんど発生しない同時実行性を提供)
- SQL Server 2016 からは、**クラスター化列ストア インデックス**を作成できるようになり、Analytics ワークロードにも対応

➡ ネイティブ コンパイル ストアド プロシージャの作成例

インメモリ OLTP では、ネイティブ コンパイル ストアド プロシージャを作成することで、さらなる性能向上が可能です。これは、通常のストアド プロシージャと同じように **CREATE PROCEDURE** ステートメントを利用して、次のように作成することができます。

```
-- メモリ最適化テーブルの作成例
CREATE TABLE t1_InMem
( col1 int NOT NULL
```

```

        PRIMARY KEY NONCLUSTERED
        HASH WITH (BUCKET_COUNT = 10000000)
    , col2 nvarchar(100) )
WITH ( MEMORY_OPTIMIZED = ON
      , DURABILITY = SCHEMA_AND_DATA )

-- ネイティブ コンパイル ストアド プロシージャの作成例
CREATE PROC nativeTest
@i int
WITH
    NATIVE_COMPILATION,
    EXECUTE AS OWNER,
    SCHEMABINDING
AS
BEGIN ATOMIC
    WITH (
        TRANSACTION ISOLATION LEVEL = SNAPSHOT,
        LANGUAGE = N'japanese')
    INSERT INTO dbo.t1_InMem VALUES (@i, N'AAAAA')
END

```

CREATE PROCEDURE の WITH 句で **NATIVE_COMPILATION** を指定すると、ネイティブ コンパイル ストアド プロシージャを作成することができます。

➡ ブロッキングの最小化（ロック待ちやラッチ待ちを最小化した同時実行性を提供）

従来のデータベース エンジンでは、データの更新時（INSERT/UPDATE/DELETE 時）に、**ロック待ち**（Lock Wait）や**ラッチ待ち**（Latch Wait）が原因で、同時実行性が低下することがありましたが、インメモリ OLTP 機能では、このようなブロッキングはほとんど発生しません。インメモリ OLTP 機能では、**ロックを利用しないマルチ バージョンの楽観的同時実行制御**（Optimistic Concurrency Control）が採用されているからです。これは、SQL Server 2005 からサポートされているスナップショット分離機構をインメモリ OLTP 向けに改良したものです。この新しいスナップショット分離機構は、tempdb を利用しない、完全なインメモリ アーキテクチャです。

このように、インメモリ OLTP 機能では、ロック待ちやラッチ待ちがほとんど発生しない、同時実行性を実現することができるので、**大量のユーザーからの多数の更新要求**が発生するシステム（オンライン ゲームやオンライン トレード、チケット予約、ポイントカード システム）など、**トランザクションとしては小さく**（実行されるステートメントが単純で、短いトランザクション）、**多数の同時実行によるブロッキング**が発生しやすいシステムで最も効果を発揮します。

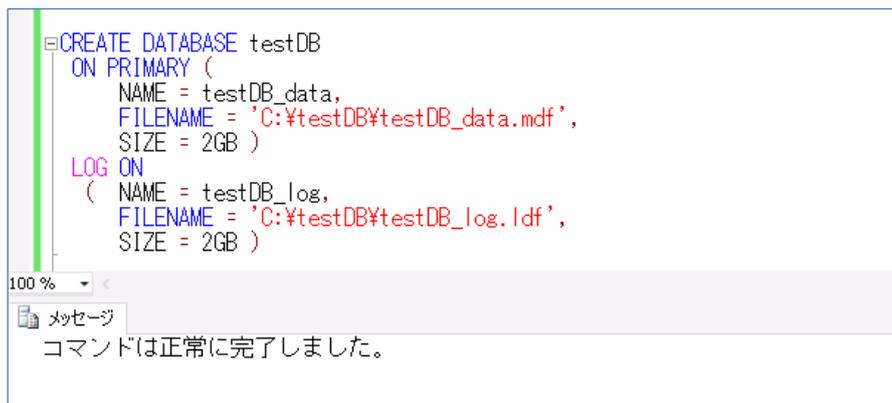
4.2 インメモリ OLTP の基本操作

それでは、インメモリ OLTP の基本的な利用方法を試してみましょう。

➡ データベースの作成 (testDB)

1. まずは、データベースを作成します。データベースの名前は「testDB」として、次のように **CREATE DATABASE** ステートメントを実行します。

```
CREATE DATABASE testDB
ON PRIMARY (
    NAME = testDB_data,
    FILENAME = 'C:¥testDB¥testDB_data.mdf',
    SIZE = 2GB )
LOG ON
( NAME = testDB_log,
  FILENAME = 'C:¥testDB¥testDB_log.ldf',
  SIZE = 2GB )
```



FILENAME で指定しているファイルパス（データファイルとトランザクションログファイルの作成先となるフォルダー）には、「C:¥testDB」フォルダーを指定していますが、皆さんの環境に合わせて適宜変更してください。

各ファイルのサイズ（**SIZE**）を **2GB** に設定しているのは、この後の性能検証で、**100万件のデータを INSERT** したりするので、ファイルのサイズを大きくしておかないと、ファイルの自動拡張が発生して、その分実行速度が遅くなってしまふのを避けるためです。もし、ファイルの自動拡張が発生してしまうと、メモリ最適化テーブルにとって有利な検証になってしまうので、そうならないようにしています（サイズを大きくして自動拡張が発生しないようにし、公平な検証になるようにしています）。

➡ 通常テーブルの作成 (t1_disk)

1. 次に、**CREATE TABLE** ステートメントを利用して、通常のテーブル（従来どおりのディスク

ベースのテーブル) を、「**t1_disk**」という名前で作成します。

```
USE testDB
CREATE TABLE t1_disk
( col1 int PRIMARY KEY CLUSTERED
, col2 nvarchar(100) )
```

```
USE testDB
CREATE TABLE t1_disk
( col1 int PRIMARY KEY CLUSTERED
, col2 nvarchar(100) )
```

100 %

メッセージ
コマンドは正常に完了しました。

col1 列を **CLUSTERED** (クラスター化インデックス) の **PRIMARY KEY** (主キー制約)、**col2** 列を **nvarchar(100)** で作成しています。

➔ 通常テーブルへの 100 万件のデータ INSERT

- 次に、**WHILE** ループを利用して、**100 万件のデータ**を **INSERT** し、そのときの実行時間を調べておきます。

```
SET NOCOUNT ON
BEGIN TRAN
DECLARE @i int = 1
WHILE @i <= 1000000
BEGIN
    INSERT INTO t1_disk VALUES (@i, N'AAAAA')
    SET @i += 1
END
COMMIT TRAN
SET NOCOUNT OFF
```

```
-- 100万件の INSERT
SET NOCOUNT ON
BEGIN TRAN
DECLARE @i int = 1
WHILE @i <= 1000000
BEGIN
    INSERT INTO t1_disk VALUES (@i, N'AAAAA')
    SET @i += 1
END
COMMIT TRAN
SET NOCOUNT OFF
```

100 %

メッセージ
コマンドは正常に完了しました。

実行が完了したら、実行にかかった時間をメモしておいてください。

2. 次に、**SELECT** ステートメントを実行して、きちんとデータが **INSERT** されたかどうかを確認します。

```
-- データの確認
SELECT TOP 1000 * FROM t1_disk
```

100 %

結果 メッセージ

	col1	col2
1	1	AAAAA
2	2	AAAAA
3	3	AAAAA
4	4	AAAAA
5	5	AAAAA
6	6	AAAAA
7	7	AAAAA

3. また、**COUNT** 関数を利用して、データ件数が **100 万件**であることも確認しておきます。

```
-- データ件数の確認
SELECT COUNT(*) FROM t1_disk
```

100 %

結果 メッセージ

	(列名なし)
1	1000000

4.3 メモリ最適化テーブルの作成／性能比較

次に、**メモリ最適化テーブル**を作成して、性能をチェックしてみましょう。メモリ最適化テーブルを作成するには、次の作業が必要になります。

- メモリ最適化テーブルを格納するための**ファイル グループ**の作成 (**CONTAINS MEMORY_OPTIMIZED_DATA** を指定)
- **メモリ最適化テーブル**の作成 (**MEMORY_OPTIMIZED = ON** を指定)

➡ ファイル グループの作成

まずは、メモリ最適化テーブルを格納するための**ファイル グループ**を作成します。

1. ファイル グループを作成するには（既存のデータベースにファイル グループを追加するには）、次のように **ALTER DATABASE** ステートメントを実行します。

```
ALTER DATABASE testDB
ADD FILEGROUP fg1
CONTAINS MEMORY_OPTIMIZED_DATA

ALTER DATABASE testDB
ADD FILE ( NAME = testDB_InMem,
          FILENAME = 'C:¥testDB¥testDB_InMem' )
TO FILEGROUP fg1
```

```
-- ファイル グループの追加
ALTER DATABASE testDB
ADD FILEGROUP fg1
CONTAINS MEMORY_OPTIMIZED_DATA

ALTER DATABASE testDB
ADD FILE ( NAME = testDB_InMem,
          FILENAME = 'C:¥testDB¥testDB_InMem' )
TO FILEGROUP fg1
```

100 %

メッセージ
コマンドは正常に完了しました。

ADD FILEGROUP に続けてファイル グループ名を指定して（ここでは **fg1** という名前を指定）、「**CONTAINS MEMORY_OPTIMIZED_DATA**」と記述することで、メモリ最適化テーブルを格納できるファイル グループを作成することができます。

次の **ALTER DATABASE** ステートメントの **ADD FILE** では、ファイル グループ内 (**fg1** 内) にファイルを作成しますが、**FILENAME** に指定するファイル名は、**NAME** で指定する論理ファイル名（上記では **testDB_InMem**）と同じ名前に設定します。

このように、通常のデータベースに、メモリ最適化テーブル用のファイル グループを追加することで、通常のテーブル（従来どおりのディスク ベースのテーブル）と、メモリ最適化テーブルを**同じデータベース内に共存**させることができます。

➔ メモリ最適化テーブルの作成

次に、メモリ最適化テーブルを作成します。

1. メモリ最適化テーブルを作成するには、次のように **CREATE TABLE** ステートメントを実行します。

```
USE testDB
CREATE TABLE t1_InMem
( col1 int NOT NULL
  PRIMARY KEY NONCLUSTERED
  HASH WITH (BUCKET_COUNT = 2000000)
, col2 nvarchar(100) )
WITH ( MEMORY_OPTIMIZED = ON,
      DURABILITY = SCHEMA_ONLY )
```

```
-- メモリ最適化テーブルの作成
USE testDB
CREATE TABLE t1_InMem
( col1 int NOT NULL
  PRIMARY KEY NONCLUSTERED
  HASH WITH (BUCKET_COUNT = 2000000)
, col2 nvarchar(100) )
WITH ( MEMORY_OPTIMIZED = ON,
      DURABILITY = SCHEMA_ONLY )
```

100 %

メッセージ
コマンドは正常に完了しました。

「**t1_InMem**」という名前で作成し、**col1** 列を **PRIMARY KEY** (主キー) へ設定し、**NONCLUSTERED** (非クラスター化) の **HASH インデックス**を作成します (∵メモリ最適化テーブルを作成するには、**NONCLUSTERED** のインデックスを作成するのが必須になるため。なお、**NOT NULL** は SQL Server 2014 のときには必須でしたが、SQL Server 2016 からは必須ではなくなったので省略しても大丈夫です)。**BUCKET_COUNT** は、ハッシュ インデックスの Bucket サイズ (バケット数) になりますが、この後 100 万件のデータを挿入するので、200 万に設定しています (良好なパフォーマンスを得るためには、**BUCKET_COUNT** をデータサイズと同程度~2 倍ぐらいの大きさに設定しておくようにします)。

WITH 句では、「**MEMORY_OPTIMIZED = ON**」を記述することで、メモリ最適化テーブルとして設定することができます。また、「**DURABILITY = SCHEMA_ONLY**」とすることで、メモリ内のみ存在するテーブル (ディスクにはデータを保存しないテーブル) を作成することができます。データを保存 / 永続化したい場合には、**DURABILITY** で **SCHEMA_AND_DATA** を指定します。

➔ 100 万件のデータ INSERT

1. 次に、メモリ最適化テーブル「**t1_InMem**」へ **100 万件のデータ**を **INSERT** してみましょ

う。

```
SET NOCOUNT ON
BEGIN TRAN
DECLARE @i int = 1
WHILE @i <= 1000000
BEGIN
  INSERT INTO t1_InMem VALUES (@i, N'AAAAA')
  SET @i += 1
END
COMMIT TRAN
SET NOCOUNT OFF
```

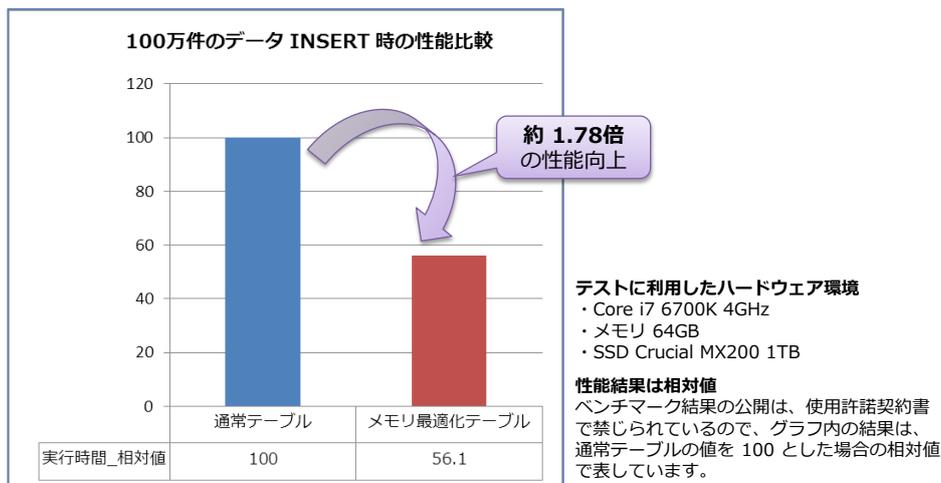
```
-- メモリ最適化テーブルの場合の性能確認
SET NOCOUNT ON
BEGIN TRAN
DECLARE @i int = 1
WHILE @i <= 1000000
BEGIN
  INSERT INTO t1_InMem VALUES (@i, N'AAAAA')
  SET @i += 1
END
COMMIT TRAN
SET NOCOUNT OFF
```

100 %

メッセージ
コマンドは正常に完了しました。

実行が完了したら、実行時間をメモして、通常のテーブルで 100 万件のデータを INSERT した場合と比較してみてください。

弊社環境では、以下のグラフのように**約 1.78 倍**の性能向上になることを確認できました（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、通常テーブルでの実行時間を **100** とした場合の相対値で表しています）。



2. 次に、データが正しく追加されているかどうかを確認しておきましょう。

```
-- データの確認
SELECT TOP 1000 * FROM t1_InMem
```

```
-- データの確認
SELECT TOP 1000 * FROM t1_InMem
```

	col1	col2
1	999018	AAAAA
2	999019	AAAAA
3	999020	AAAAA
4	999021	AAAAA
5	999022	AAAAA
6	999023	AAAAA
7	999024	AAAAA

メモリ最適化テーブルでは、col1 列に **HASH インデックス**を作成しているのですが、データがバラバラに表示されていますが、実際にデータが格納されていることを確認できます。

3. COUNT 関数を利用して、データ件数が 100 万件であることも確認しておきましょう。

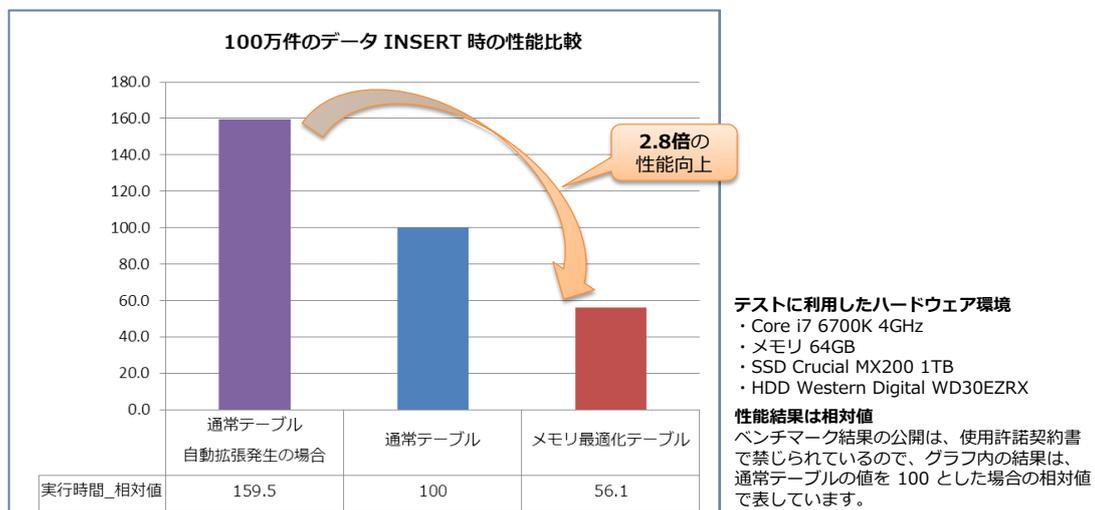
```
-- データ件数の確認
SELECT COUNT(*) FROM t1_InMem
```

```
-- データ件数の確認
SELECT COUNT(*) FROM t1_InMem
```

	(列名なし)
1	1000000

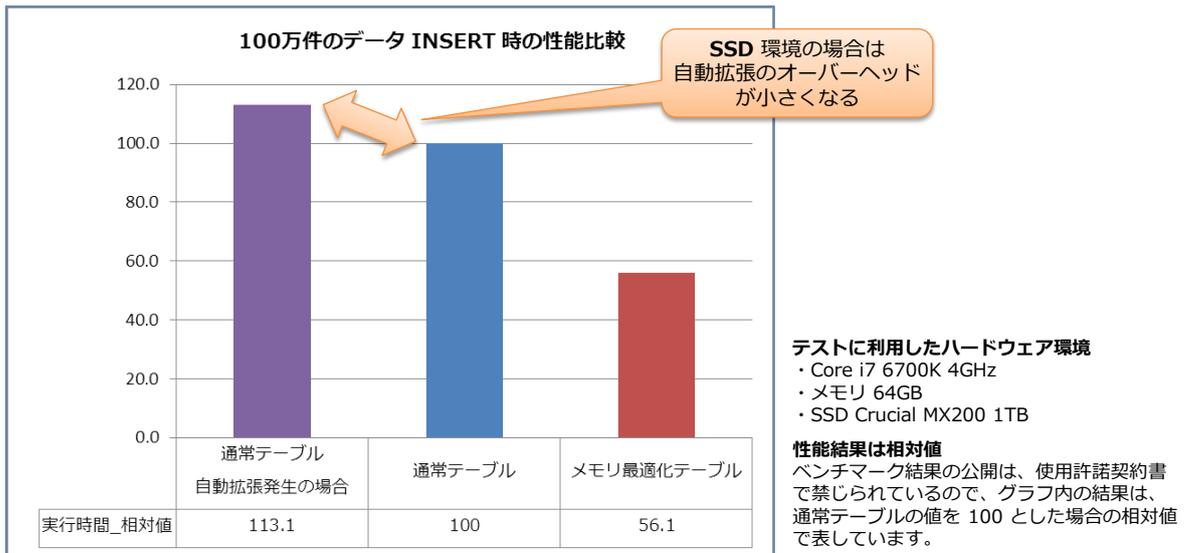
➡ ディスク ベースで自動拡張が発生する場合の性能差

今回の検証では、データベース (testDB) を作成するときに、ファイル サイズ (SIZE) を大きく設定して、データ ファイル (.mdf) およびトランザクション ログ ファイル (.ldf) の自動拡張が発生しないようにして、公平な検証を行っていましたが、もし自動拡張が発生する場合 (SIZE を指定しないでデータベースを作成した場合) は、次のような結果になります (弊社の HDD 環境の場合)。



自動拡張が発生した場合と比較すると、メモリ最適化テーブルは **2.8 倍の性能向上**になります。このように、従来のディスク ベースのテーブルでは、ファイルの自動拡張によるパフォーマンス低下が発生するのに対して、メモリ最適化テーブルは自動拡張の影響は受けません（∵メモリ内にのみテーブルが存在して、SCHEMA_ONLY オプションでは、ログへの書き込みを行わないため）。

なお、ファイルの自動拡張は、ストレージの速度にも大きな影響を受けます。したがって、**SSD** など、より速度の速いストレージを利用すれば、自動拡張のオーバーヘッドを小さくすることができ、次のような結果になります（弊社の SSD 環境の場合）。



このように、従来のディスク ベースのエンジンでは、自動拡張の影響や、ストレージの性能に大きな影響を受けます。インメモリ OLTP 機能では、メモリ内にテーブルを配置するので、自動拡張は発生しませんし、ストレージの性能に影響を受けることもありません（ただし、後述の Durability オプションでデータの永続化を設定した場合には、ストレージへの書き込みが発生します）。

4.4 テーブル サイズ (メモリ使用量) の確認

次に、通常テーブルおよびメモリ最適化テーブルのテーブル サイズ (メモリ使用量) を確認してみましょう。

1. 通常のテーブルは、次のように **sp_spaceused** システム ストアド プロシージャを利用して、テーブル サイズを確認することができます。

```
EXEC sp_spaceused 't1_disk'
```

```
-- テーブル サイズを確認
EXEC sp_spaceused 't1_disk'
```

	name	rows	reserved	data	index_size	unused
1	t1_disk	1000000	26888 KB	26760 KB	112 KB	16 KB

2. これに対して、メモリ最適化テーブルでは、次のように **sp_spaceused** システム ストアド プロシージャを利用しても、テーブル サイズを確認することができません。

```
EXEC sp_spaceused 't1_InMem'
```

```
EXEC sp_spaceused 't1_InMem'
```

	name	rows	reserved	data	index_size	unused
1	t1_InMem	0	0 KB	0 KB	0 KB	0 KB

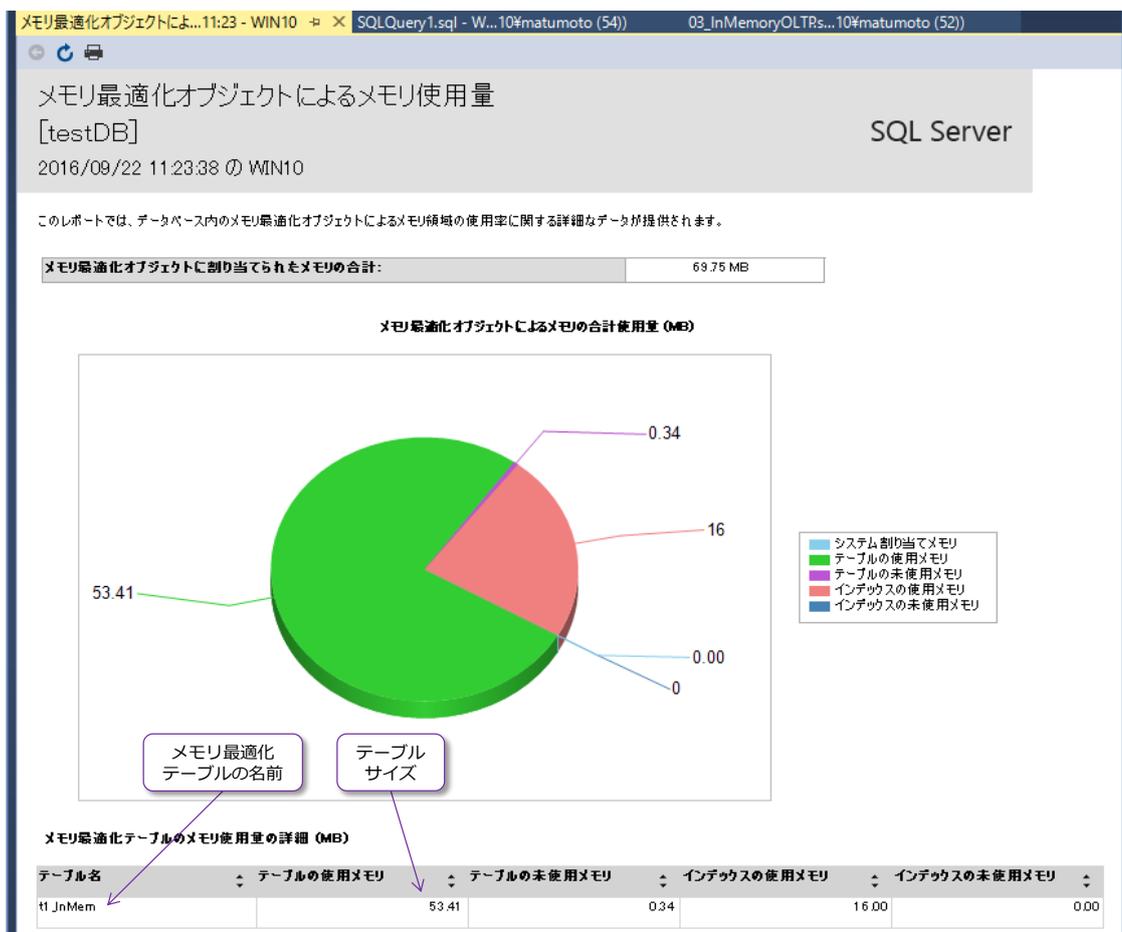
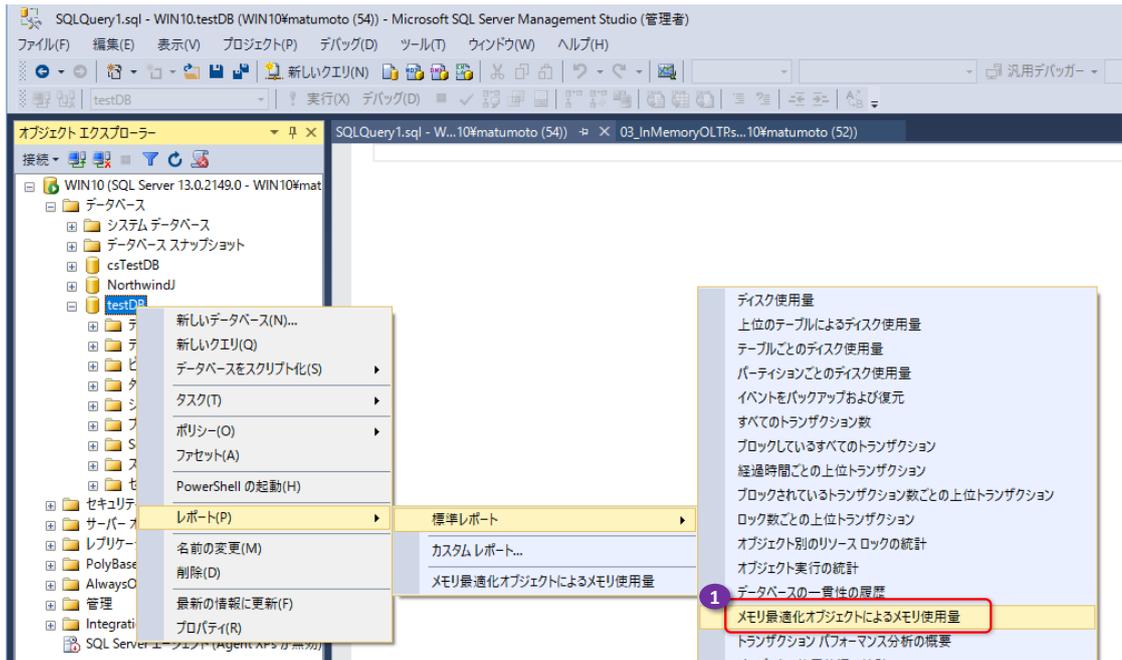
3. メモリ最適化テーブルでは、テーブル サイズを確認するには、**動的管理ビュー**または **Management Studio のレポート機能**を利用します。動的管理ビューを利用する場合は、次のように「**dm_db_xtp_table_memory_stats**」ビューを参照します。

```
SELECT OBJECT_NAME(object_id), *
FROM sys.dm_db_xtp_table_memory_stats
```

```
SELECT OBJECT_NAME(object_id), *
FROM sys.dm_db_xtp_table_memory_stats
```

	(列名なし)	object_id	memory_allocated_for_table_kb	memory_used_by_table_kb	memory_allocated_for_index_kb
1	t1_InMem	597577167	55040	54687	16384

4. Management Studio のレポート機能を利用する場合には、次のようにオブジェクト エクスプローラーで該当データベース (**testDB**) を右クリックして、[レポート] の [標準レポート] から [メモリ最適化オブジェクトによるメモリ使用量] をクリックします。



メモリ最適化テーブルの一覧が表示されて、[テーブルの使用メモリ] でテーブル サイズ (メモリ使用量) を確認することができます。

このように、メモリ最適化テーブルのテーブル サイズ (メモリ使用量) を確認するには、動的管理ビューまたはレポート機能を利用するようにします。

4.5 ネイティブ コンパイル ストアド プロシージャによる性能向上

メモリ最適化テーブルでは、**ネイティブ コンパイル ストアド プロシージャ** (Natively Compiled Stored Procedure) を作成することで、さらなる性能向上を実現することができます。

➡ ネイティブ コンパイル ストアド プロシージャの作成

ネイティブ コンパイル ストアド プロシージャは、次のように作成することができます。

```
CREATE PROC ストアドプロシージャ名
WITH
    NATIVE_COMPILATION,
    EXECUTE AS OWNER,
    SCHEMABINDING
AS
BEGIN ATOMIC
    WITH (
        TRANSACTION ISOLATION LEVEL = SNAPSHOT,
        LANGUAGE = N'japanese')
    -- 実行したいステートメント
END
```

CREATE PROCEDURE ステートメントで、「**WITH NATIVE_COMPILATION**」を指定することで、ネイティブ コンパイル ストアド プロシージャとして、ストアド プロシージャを作成することができます。また、ネイティブ コンパイル ストアド プロシージャでは、**BEGIN ATOMIC** と **END** で囲んで、1 つのトランザクションとして、ステートメントを実行するようにし、**EXECUTE AS OWNER** と **SCHEMABINDING** も指定する必要があります。

➡ 100 万件のデータ INSERT

次に、**100 万件のデータ**を **INSERT** するネイティブ コンパイル ストアド プロシージャを作成してみましょう。

1. 次のように、**nativeTest** という名前で作成します。

```
CREATE PROC nativeTest
@LoopCount int
WITH
    NATIVE_COMPILATION,
    EXECUTE AS OWNER,
    SCHEMABINDING
AS
BEGIN ATOMIC
    WITH (
        TRANSACTION ISOLATION LEVEL = SNAPSHOT,
        LANGUAGE = N'japanese')
```

```

DECLARE @i int = 1
WHILE @i <= @LoopCount
BEGIN
    INSERT INTO dbo.t1_InMem VALUES (@i, N'AAAAA')
    SET @i += 1
END
END
go

```

```

-- ネイティブ コンパイル ストアド プロシージャの作成
CREATE PROC nativeTest
@LoopCount int
WITH
    NATIVE_COMPILATION,
    EXECUTE AS OWNER,
    SCHEMABINDING
AS
BEGIN ATOMIC
    WITH (
        TRANSACTION ISOLATION LEVEL = SNAPSHOT,
        LANGUAGE = N'japanese')

    DECLARE @i int = 1
    WHILE @i <= @LoopCount
    BEGIN
        INSERT INTO dbo.t1_InMem VALUES (@i, N'AAAAA')
        SET @i += 1
    END
END
go

```

100 %

メッセージ
コマンドは正常に完了しました。

ネイティブ コンパイル ストアド プロシージャ内では、テーブル名をスキーマ名付きで指定する必要があるため、「**dbo.t1_InMem**」と指定しています。

- 作成が完了したら、次のように **DELETE** ステートメントを実行して、**t1_InMem** テーブルのデータをすべて削除します。

```
DELETE FROM t1_InMem
```

```

-- いったんデータを削除
DELETE FROM t1_InMem

```

100 %

メッセージ
(1000000 行処理されました)

- 削除後、**@LoopCount** 入力パラメーターに「**1000000**」を与えて、**100 万件のデータ**を **INSERT** します。

```
EXEC nativeTest @LoopCount = 1000000
```

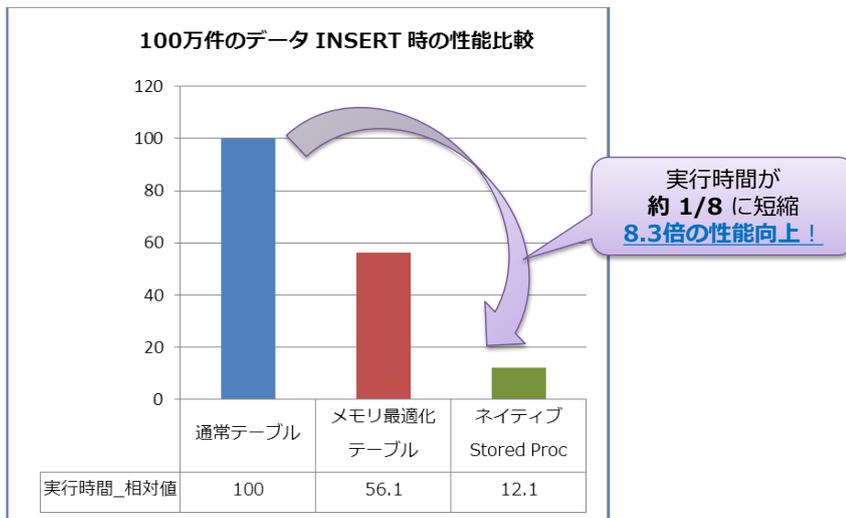
```
-- 100万件の INSERT
EXEC nativeTest @LoopCount = 1000000
```

100 %

メッセージ
コマンドは正常に完了しました。

実行が完了したら、実行時間をメモして、**通常のテーブルで 100 万件のデータを INSERT**した場合と比較してみてください。

弊社環境では、以下のグラフのように**約 1/8 の実行時間**（8.3 倍の性能向上）になることを確認できました（通常テーブルでの実行時間を **100** とした場合の相対値）。



4. 次に、データが正しく追加されているかどうかを確認しておきましょう。

```
SELECT TOP 1000 * FROM t1_InMem
SELECT COUNT(*) FROM t1_InMem
```

```
-- データの確認
SELECT TOP 1000 * FROM t1_InMem
SELECT COUNT(*) FROM t1_InMem
```

100 %

結果

	col1	col2
1	999198	AAAAA
2	999199	AAAAA
3	999200	AAAAA
4	999201	AAAAA
5	999202	AAAAA

(列名なし)

1	1000000
---	---------

100 万件のデータが INSERT されていることを確認できると思います。

➔ 1,000 万件のデータの INSERT

次に、1,000 万件のデータを INSERT してみましよう。

1. まずは、DELETE ステートメントを実行して、全データを削除します。

```
DELETE FROM t1_InMem
```

2. 削除が完了したら、@LoopCount 入力パラメーターに「10000000」を与えて、1,000 万件のデータを INSERT してみます。

```
EXEC nativeTest @LoopCount = 10000000
```

```
-- 1千万件の INSERT
DELETE FROM t1_InMem
EXEC nativeTest @LoopCount = 10000000
```

100 %

メッセージ
コマンドは正常に完了しました。

1,000 万件のデータの INSERT でも、数秒程度で実行が完了することを確認できたのではないのでしょうか？ 弊社環境では、100 万件のデータを INSERT したときの約 10 倍の実行速度で実行が完了することを確認しています。

3. 実行後、データが正しく INSERT されたことを確認しておきましょう。

```
SELECT COUNT(*) FROM t1_InMem
```

```
SELECT COUNT(*) FROM t1_InMem
```

100 %

結果 メッセージ

(列名なし)
10000000

1,000 万件のデータが INSERT されていることを確認できると思います。

このように、ネイティブ コンパイル ストアド プロシージャを利用すれば、1,000 万件のデータでも数秒で処理することができるようになります。

4. 次に、「dm_db_xtp_table_memory_stats」動的管理ビューを利用して、テーブル サイズ（メモリ使用量）も確認しておきましょう。

```
SELECT OBJECT_NAME(object_id), *
FROM sys.dm_db_xtp_table_memory_stats
```

```
SELECT OBJECT_NAME(object_id), *
FROM sys.dm_db_xtp_table_memory_stats
```

(列名なし)	object_id	memory_allocated_for_table_kb	memory_used_by_table_kb	memory_allocated_for_t
1	t1_InMem	597577167	660416	656250

Note : メモリが足りない場合の動作

もし、メモリが足りない場合には（物理メモリのサイズが小さく、メモリ最適化テーブルを格納しきれない場合には）、次のようにエラーが発生します。

```
EXEC netiveTest @LoopCount = 10000000
```

メッセージ 701、レベル 17、状態 103、プロシージャ netiveTest、行 252
このクエリを実行するには、リソース プール 'default' のシステム メモリが不足しています。

■ メモリ サイズの見積もり

メモリ サイズの見積もり方法については、SQL Server 2014 実践シリーズの「No.1 インメモリ OLTP の実践的な利用方法」で説明しているので、こちらもぜひご覧いただければと思います。

インメモリ OLTP の実践的な利用方法

https://www.microsoft.com/ja-jp/sqlserver/2014/technology/self-learning.aspx#practical_contents

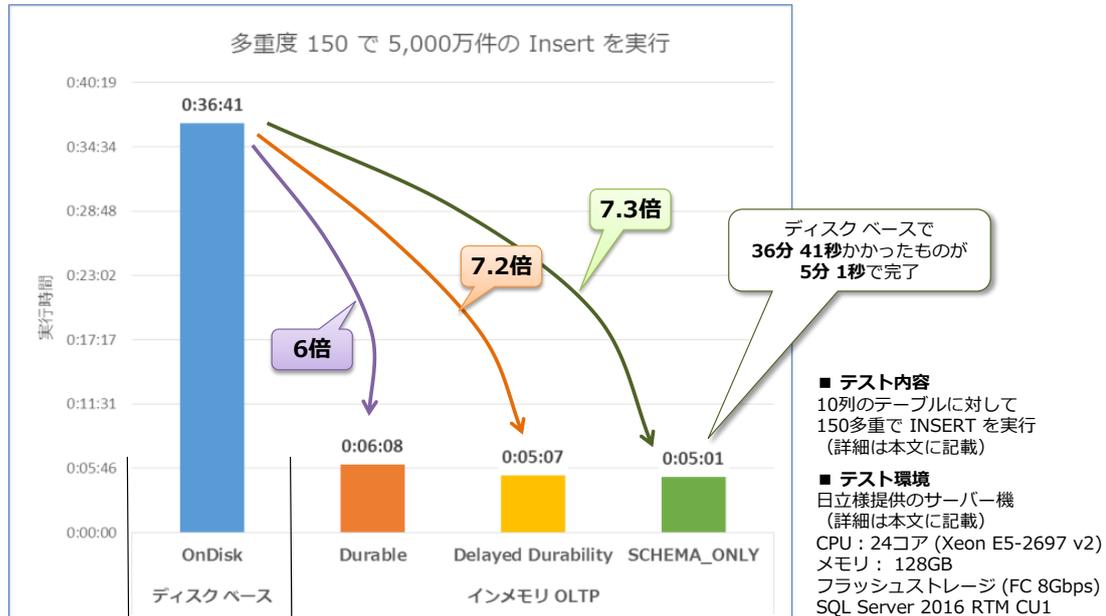
➡ 1 件の INSERT をネイティブ コンパイル ストアド プロシージャ化

ここまで作成したネイティブ コンパイル ストアド プロシージャは、100 万件のデータを INSERT するものでしたが、次のように **1 件のデータだけを INSERT** するものも、もちろん作成できます。

```
CREATE PROC nativeInsert1
  @p1 int,
  @p2 nvarchar(100)
WITH
  NATIVE_COMPILATION,
  EXECUTE AS OWNER,
  SCHEMABINDING
AS
BEGIN ATOMIC
  WITH (
    TRANSACTION ISOLATION LEVEL = SNAPSHOT,
    LANGUAGE = N'japanese')

  INSERT INTO dbo.t1_InMem VALUES (@p1, @p2)
END
go
```

このように 1 件のデータのみを INSERT するネイティブ コンパイル ストアド プロシージャを利用して、多重度 150 で 5,000 万件のデータを INSERT したときの結果が、Step 2.2 で紹介した以下のグラフです。



* ベンチマークの結果の公表は、使用許諾契約書で禁止されていますが、その効果をわかりやすく表現するため、日本マイクロソフト株式会社の監修のもと、数値を掲載しております。

このように、ネイティブ コンパイル ストアド プロシージャを利用すれば、1 件だけの INSERT を行うネイティブ コンパイル ストアド プロシージャを作成しても、性能向上を実現することができます。なお、ネイティブ コンパイル ストアド プロシージャの詳細については、**SQL Server 2014 実践シリーズ**の「**No.1 インメモリ OLTP の実践的な利用方法**」でも説明しているので、こちらもぜひご覧いただければと思います。

インメモリ OLTP の実践的な利用方法

https://www.microsoft.com/ja-jp/sqlserver/2014/technology/self-learning.aspx#practical_contents

Note : コンテキスト スイッチや性能の測定方法に注意

1 件だけのネイティブ コンパイル ストアド プロシージャを作成した場合は、コンテキスト スイッチのオーバーヘッドと、性能の測定方法に注意する必要があります。コンテキスト スイッチに関しては、次のような状況です。

```
BEGIN TRAN
DECLARE @i int = 1
WHILE @i <= 1000000
BEGIN
    EXEC nativeInsert1 @i, 'BBB'
    SET @i += 1
END
COMMIT TRAN
```

このように、Transact-SQL ステートメントの中に、ネイティブ コンパイル ストアド プロシージャを混ぜて実行すると、コンテキスト スイッチが発生してしまうオーバーヘッドがあります。これでは、逆に性能が低下してしまうことになるので、このような使い方はしないように注意する必要があります。

また、1件だけのステートメントの実行時間を計測する場合は、性能の測定方法に注意する必要があります。たとえば、次のように **SET STATISTICS TIME** コマンドを利用した場合は、**0 ミリ秒**と結果が表示されます。

```

SET STATISTICS TIME ON
EXEC nativeInsert1 1000005, 'BBB'

```

SQL Server の構文解析とコンパイルの時間:
CPU 時間 = 0 ミリ秒、経過時間 = 0 ミリ秒。

SQL Server 実行時間:
CPU 時間 = 0 ミリ秒、経過時間 = 0 ミリ秒。

実行速度がマイクロ秒だと
0ミリ秒と表示される

SQL Server は、1件のみを扱うステートメントは、インメモリ OLTP を利用しているかどうかに関わらず、通常のテーブルに対してもマイクロ秒単位で処理ができるので、**SET STATISTICS TIME** コマンドのようにマイクロ秒単位での性能測定ができないコマンドでは、0 ミリ秒と表示されて、正しい計測をすることができません。

また、.NET Framework での **Stopwatch** クラスを利用する場合も、マイクロ秒単位での性能測定ができないことに注意する必要があります。これは、次のような状況です。

```

Module1.vb
15
16
17 Try
18     Using cn As New SqlConnection(cnstr)
19         Using cmd As New SqlCommand()
20             cn.Open()
21             cmd.Connection = cn
22             cmd.CommandType = CommandType.StoredProcedure
23             cmd.CommandText = "nativeInsert1"
24             Dim p1 As SqlParameter, p2 As SqlParameter
25             p1 = cmd.Parameters.Add("@p1", SqlDbType.Int)
26             p1.Value = 1000007
27             p2 = cmd.Parameters.Add("@p2", SqlDbType.NVarChar, 100)
28             p2.Value = "BBB"
29
30             ' ストップウォッチ開始
31             Dim sw As New System.Diagnostics.Stopwatch()
32             sw.Start()
33
34             cmd.ExecuteScalar()
35
36             ' ストップウォッチを停止
37             sw.Stop()
38             Console.WriteLine("処理時間: " & sw.ElapsedMilliseconds & " ミリ秒")
39         End Using
40     End Using
41

```

C:\Windows\system32\cmd.exe
処理時間: 0 ミリ秒

実行速度がマイクロ秒だと
0ミリ秒と表示される

マイクロ秒で処理された場合には、0 ミリ秒と表示されて、正しく計測をすることができません。

また、.NET Framework を利用する場合は、デバッグ実行を利用した場合には、デバッグのオーバーヘッドがあることにも注意する必要があります。このほかにも注意点がありますが、それらについても、**SQL Server 2014 実践シリーズ**の「**No.1 インメモリ OLTP の実践的な利用方法**」で説明しているので、こちらませひご覧いただければと思います。

ネイティブ コンパイル ストアド プロシージャを利用すれば、性能向上を実現することができるので、ぜひ検討してみてください。

4.6 データの永続化 (SCHEMA_AND_DATA)

メモリ最適化テーブルでは、SQL Server の再起動後にも、**データを復旧**できる、**Durability (永続化)** オプションが用意されています。これは、次のように利用することができます。

```
CREATE TABLE テーブル名
( 列名1 データ型 NOT NULL
    PRIMARY KEY NONCLUSTERED
    HASH WITH (BUCKET_COUNT=バケット数)
, 列名2 データ型, ... )
WITH ( MEMORY_OPTIMIZED = ON,
    DURABILITY = SCHEMA_AND_DATA )
```

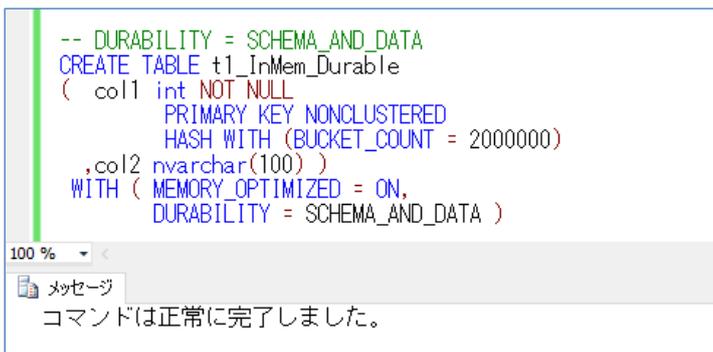
今まで試してきたメモリ最適化テーブルは、**DURABILITY=SCHEMA_ONLY** と指定して作成していましたが、この場合はスキーマ (テーブルやインデックスの定義) のみを永続化 (ディスクへ保存) して、データは永続化されません (SQL Server が再起動されるとデータが失われます)。これに対して、**DURABILITY=SCHEMA_AND_DATA** と指定した場合は、スキーマだけでなく、**データも永続化**することができ、SQL Server が再起動されたとしても、データが失われることはありません。このオプション (**SCHEMA_AND_DATA**) では、データに対する更新情報をディスク (ログ) へ記録しておくことで、永続化を可能にしています。

➔ Let's Try

それでは、これを試してみましょう。

1. 次のように **t1_InMem_Durable** という名前で、**DURABILITY=SCHEMA_AND_DATA** を指定してメモリ最適化テーブルを作成します。

```
CREATE TABLE t1_InMem_Durable
( col1 int NOT NULL
    PRIMARY KEY NONCLUSTERED
    HASH WITH (BUCKET_COUNT = 2000000)
, col2 nvarchar(100) )
WITH ( MEMORY_OPTIMIZED = ON,
    DURABILITY = SCHEMA_AND_DATA )
```



```
-- DURABILITY = SCHEMA_AND_DATA
CREATE TABLE t1_InMem_Durable
( col1 int NOT NULL
    PRIMARY KEY NONCLUSTERED
    HASH WITH (BUCKET_COUNT = 2000000)
, col2 nvarchar(100) )
WITH ( MEMORY_OPTIMIZED = ON,
    DURABILITY = SCHEMA_AND_DATA )
```

100 %

メッセージ
コマンドは正常に完了しました。

2. 次に、**100万件のデータを INSERT** して、そのときの実行時間を調べます。

```

SET NOCOUNT ON
BEGIN TRAN
DECLARE @i int = 1
WHILE @i <= 1000000
BEGIN
    INSERT INTO t1_InMem_Durable
    VALUES (@i, N'AAAAA')
    SET @i += 1
END
COMMIT TRAN
SET NOCOUNT OFF

-- データ件数の確認
SELECT COUNT(*) FROM t1_InMem_Durable

```

```

-- 100万件の INSERT
SET NOCOUNT ON
BEGIN TRAN
DECLARE @i int = 1
WHILE @i <= 1000000
BEGIN
    INSERT INTO t1_InMem_Durable
    VALUES (@i, N'AAAAA')
    SET @i += 1
END
COMMIT TRAN
SET NOCOUNT OFF

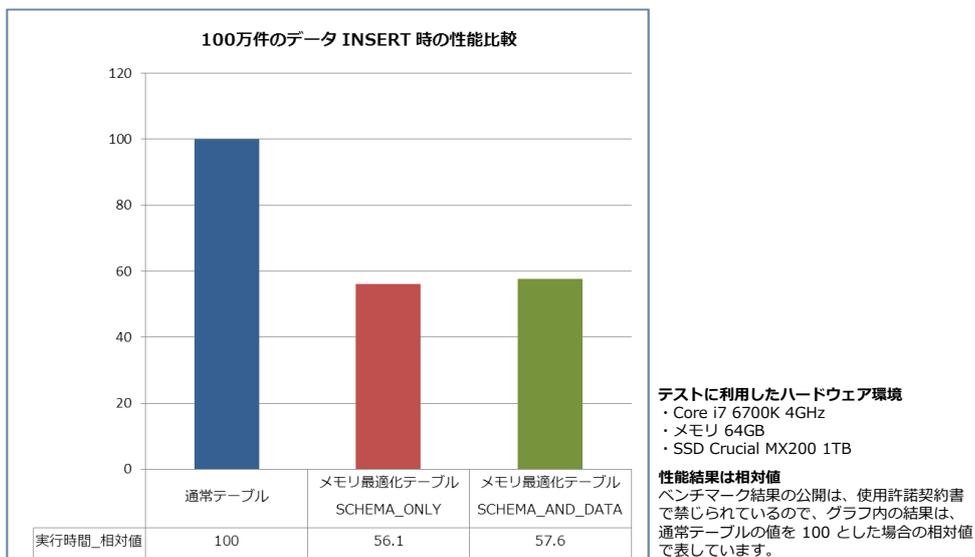
-- データ件数の確認
SELECT COUNT(*) FROM t1_InMem_Durable

```



(列名なし)	
1	1000000

弊社環境では、次のような実行時間になりました（通常テーブルの場合の実行時間を 100 とした場合の相対値）。



SCHEMA_ONLY に比べると、実行時間がかかっていますが、通常のテーブルと比べれば、性

能向上していることを確認できると思います。なお、**SCHEMA_AND_DATA** では、更新情報をログ（ディスク）に書き込むので、HDD 環境の場合は、オーバーヘッドがもう少し大きくなります（上のグラフは SSD を利用した場合の結果です）。

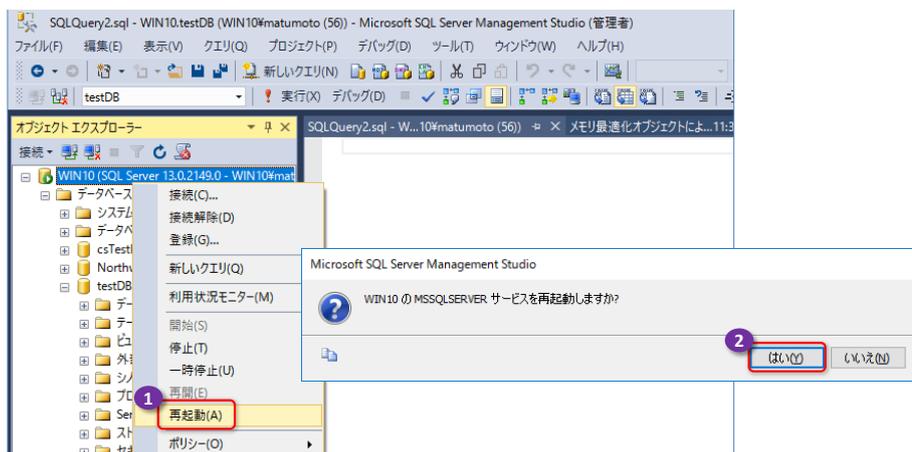
- 次に、[メモリ最適化オブジェクトによるメモリ使用量] レポートを表示して、メモリ サイズも確認しておきましょう。



➡ SQL Server 再起動後のデータ復旧の確認

次に、SQL Server を再起動して、データが復旧できるかどうかを確認してみましょう。

- まずは、オブジェクト エクスプローラーで、SQL Server の名前を右クリックして、[再起動] をクリックし、SQL Server を再起動します。



2. 再起動が完了したら、**SCHEMA_ONLY** のメモリ最適化テーブル「**t1_InMem**」に対して **SELECT** ステートメントを実行してみましょう。

```
SELECT COUNT(*) FROM t1_InMem
```

The screenshot shows a SQL query window with the following text:

```
USE testDB
SELECT COUNT(*) FROM t1_InMem
```

Below the query, there are tabs for '結果' (Results) and 'メッセージ' (Messages). The '結果' tab is active, showing a table with one column labeled '(列名なし)' and one row with the value '0'.

(列名なし)
0

結果は **0 件** と表示されて、SQL Server の再起動によって、データが失われていることを確認できます。

3. 次に、**SCHEMA_AND_DATA** のメモリ最適化テーブル「**t1_InMem_Durable**」に対して **SELECT** ステートメントを実行してみましょう。

```
SELECT COUNT(*) FROM t1_InMem_Durable
```

The screenshot shows a SQL query window with the following text:

```
SELECT COUNT(*) FROM t1_InMem_Durable
```

Below the query, there are tabs for '結果' (Results) and 'メッセージ' (Messages). The '結果' tab is active, showing a table with one column labeled '(列名なし)' and one row with the value '1000000'.

(列名なし)
1000000

今度は、**100 万件** と表示されて、データが復旧していることを確認できます。

4. 次に、**SELECT *** で実際のデータも確認してみましょう。

```
SELECT * FROM t1_InMem
```

The screenshot shows a SQL query window with the following text:

```
SELECT * FROM t1_InMem
```

Below the query, there are tabs for '結果' (Results) and 'メッセージ' (Messages). The '結果' tab is active, showing a table with two columns labeled 'col1' and 'col2'.

col1	col2
------	------

SCHEMA_ONLY の「**t1_InMem**」テーブルは、**0 件**であることを確認できます。

5. **SCHEMA_AND_DATA** の「**t1_InMem_Durable**」テーブルには、**100 万件**のデータが有ることを確認できます。

```
SELECT * FROM t1_InMem_Durable
```

```
SELECT * FROM t1_InMem_Durable
```

	col1	col2
1	999018	AAAAA
2	999019	AAAAA
3	999020	AAAAA
4	999021	AAAAA
5	999022	AAAAA

➔ Delayed Durability (遅延永続化)

SCHEMA_AND_DATA でデータを永続化する場合には、**更新のオーバーヘッドが発生**しますが、**このオーバーヘッドを軽減**するためのオプションとして「**Delayed Durability**」(遅延永続化)が用意されています。これは、データの永続化を遅延させる(非同期に書き込む)ことで、更新性能を向上させて、その分データロス(データの損失)が発生する可能性がある、というオプションです。

Delayed Durability を利用するには、次のように **ALTER DATABASE** ステートメントの **SET** オプションで、データベース レベルで許可をしておく必要があります。

```
ALTER DATABASE データベース名
SET DELAYED_DURABILITY = ALLOWED
```

DELAYED_DURABILITY に **ALLOWED** を指定することで、**Delayed Durability** を利用できるようになります。ここで **DISABLED** を指定した場合は、無効化して、元に戻すことができます。また、**FORCED** を指定した場合は、どんなトランザクションでも強制的に **Delayed Durability** で実行するという設定にすることもできます。

ALLOWED を指定した場合は、次のようにトランザクションの **COMMIT TRAN** 時に、**WITH** オプションで **DELAYED_DURABILITY** を **ON** に指定することで、**Delayed Durability** を有効化することができます。

```
BEGIN TRAN
:
COMMIT TRAN
WITH ( DELAYED_DURABILITY = ON )
```

ネイティブ コンパイル ストアド プロシージャの場合は、次のように **BEGIN ATOMIC** の **WITH** オプションで **DELAYED_DURABILITY** を **ON** に指定することで、**Delayed Durability** を有効化することができます。

```
CREATE PROC ストアド プロシージャ名
WITH NATIVE_COMPILATION,
EXECUTE AS OWNER,
SCHEMABINDING
AS
```

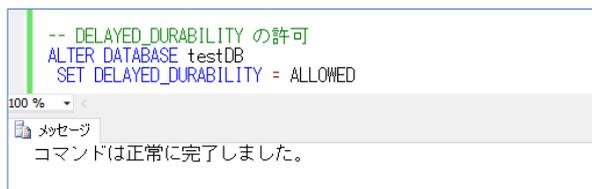
```
BEGIN ATOMIC
WITH (
    DELAYED_DURABILITY = ON,
    TRANSACTION ISOLATION LEVEL = SNAPSHOT,
    LANGUAGE = N'japanese' )
:
END
```

➔ Let's Try

それでは、これを試してみましょう。

1. まずは、データベースに対して、**Delayed Durability** を許可しておきます。

```
ALTER DATABASE testDB
SET DELAYED_DURABILITY = ALLOWED
```



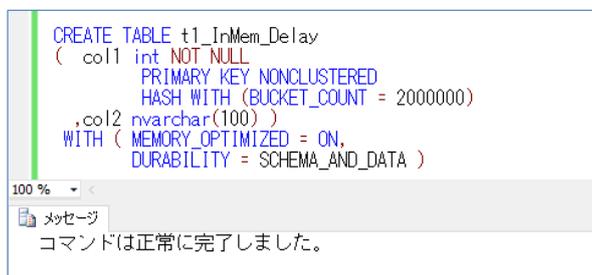
```
-- DELAYED_DURABILITY の許可
ALTER DATABASE testDB
SET DELAYED_DURABILITY = ALLOWED
```

100 %

メッセージ
コマンドは正常に完了しました。

2. 次に、**t1_InMem_Delay** という名前でメモリ最適化テーブルを作成します。

```
CREATE TABLE t1_InMem_Delay
( col1 int NOT NULL
    PRIMARY KEY NONCLUSTERED
    HASH WITH (BUCKET_COUNT = 2000000)
, col2 nvarchar(100) )
WITH ( MEMORY_OPTIMIZED = ON,
    DURABILITY = SCHEMA_AND_DATA )
```



```
CREATE TABLE t1_InMem_Delay
( col1 int NOT NULL
    PRIMARY KEY NONCLUSTERED
    HASH WITH (BUCKET_COUNT = 2000000)
, col2 nvarchar(100) )
WITH ( MEMORY_OPTIMIZED = ON,
    DURABILITY = SCHEMA_AND_DATA )
```

100 %

メッセージ
コマンドは正常に完了しました。

3. 次に、**100万件のデータを INSERT** して、そのときの実行時間を調べます。

```
SET NOCOUNT ON
BEGIN TRAN
DECLARE @i int = 1
WHILE @i <= 1000000
BEGIN
```

```

INSERT INTO t1_InMem_Delay
VALUES (@i, N'AAAAA')
SET @i += 1
END
COMMIT TRAN
WITH ( DELAYED_DURABILITY = ON )
SET NOCOUNT OFF

-- データ件数の確認
SELECT COUNT(*) FROM t1_InMem_Delay
    
```

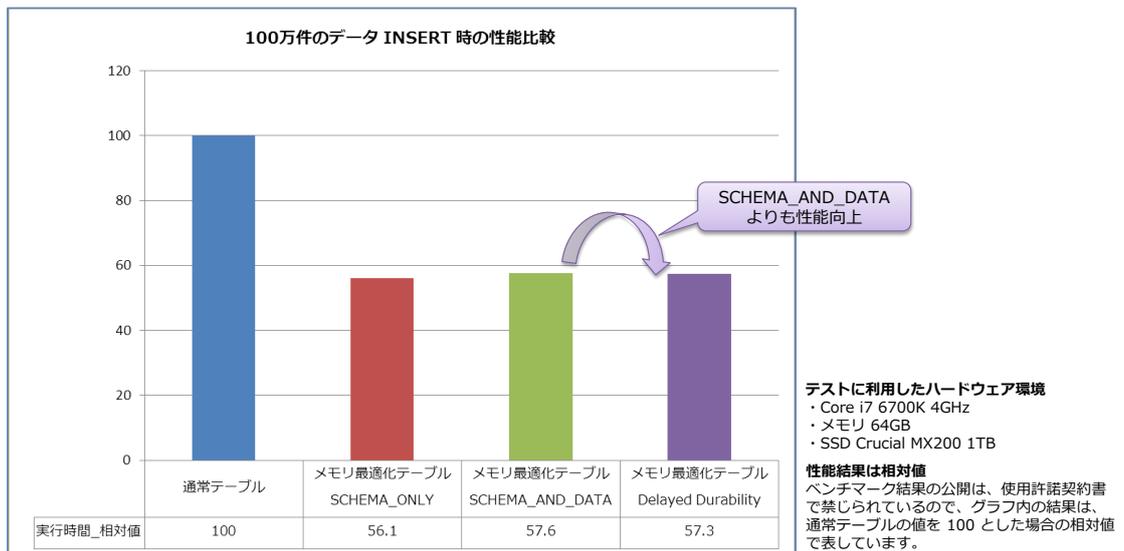
```

-- 100万件のデータ INSERT
SET NOCOUNT ON
BEGIN TRAN
DECLARE @i int = 1
WHILE @i <= 1000000
BEGIN
    INSERT INTO t1_InMem_Delay
    VALUES (@i, N'AAAAA')
    SET @i += 1
END
COMMIT TRAN
WITH ( DELAYED_DURABILITY = ON )
SET NOCOUNT OFF

-- データ件数の確認
SELECT COUNT(*) FROM t1_InMem_Delay
    
```

結果	メッセージ
(列名なし)	
1	1000000

弊社環境では、次のような実行時間になりました（通常テーブルの場合の実行時間を 100 とした場合の相対値）。



SCHEMA_AND_DATA よりも性能が向上していることを確認できると思います。Delayed Durability は、性能向上を実現する上で、大変有用なオプションの 1 つになるので、ぜひ試してみてください (Step 2.2 で紹介した多重度 150 での 5,000 万件のデータ INSERT でも、Delayed Durability での性能向上を確認しています)。ただし、Delayed Durability では、遅延永続化をする分だけ、データ ロス (データの損失) が発生する可能性があるがあるので、それとのトレードオフになります。

4.7 メモリ最適化テーブルでの同時更新（Write-Write 競合）

ここでは、メモリ最適化テーブルに対して、同時更新（書き込みの競合）が発生した場合の動作を確認してみましょう。結論から言うと、先に更新した方が勝つ方式（First Writer Win）です。

➡ Let's Try

それでは、これを試してみましょう。

1. まずは、次のように **UPDATE** ステートメントを実行して、「**t1_InMem_Durable**」テーブルの「**col1 = 1**」のデータを更新します。

```
BEGIN TRAN
UPDATE t1_InMem_Durable WITH(SNAPSHOT)
SET col2 = '1111'
WHERE col1 = 1
```

```
-- 1つ目の接続
BEGIN TRAN
UPDATE t1_InMem_Durable WITH(SNAPSHOT)
SET col2 = '1111'
WHERE col1 = 1
```

メッセージ
(1 行処理されました)

BEGIN TRAN でトランザクションを開始し、わざと **COMMIT TRAN** を省略して、トランザクション中のままにしておきます。

また、**WITH(SNAPSHOT)** を付けて、スナップショット分離レベルを指定していますが、メモリ最適化テーブルでは、トランザクション内でのステートメントは READ COMMITTED 分離レベル（既定の分離レベル）が許可されないため、スナップショット分離レベルなどで実行しなければなりません。

Note : **WITH(SNAPSHOT)** を付けない場合

もし、トランザクション内で **WITH(SNAPSHOT)** を付けていない場合は、次のようにエラーが返されます。

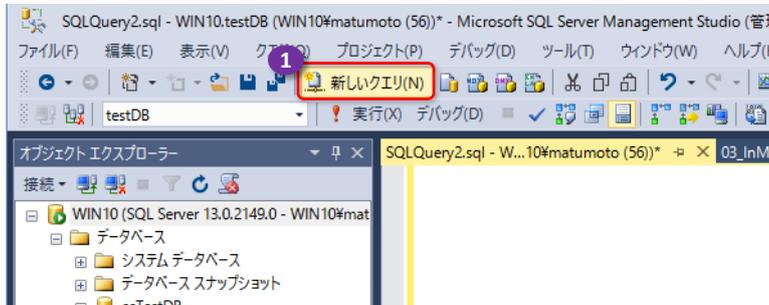
```
-- WITH(SNAPSHOT) を付けない場合のエラー
BEGIN TRAN
UPDATE t1_InMem_Durable
SET col2 = '1111'
WHERE col1 = 1
```

メッセージ 41368、レベル 16、状態 0、行 327
READ COMMITTED 分離レベルを使用したメモリ最適化テーブルへのアクセスは、自動コミット トランザクションの場合にのみサポートされています。明示的および暗黙的トランザクションの場合はいずれもサポートされません。WITH (SNAPSHOT) などのテーブル ヒントを使用して、メモリ最適化テーブルのサポートされている分離レベルを指定してください。

➔ 別の接続からデータの参照

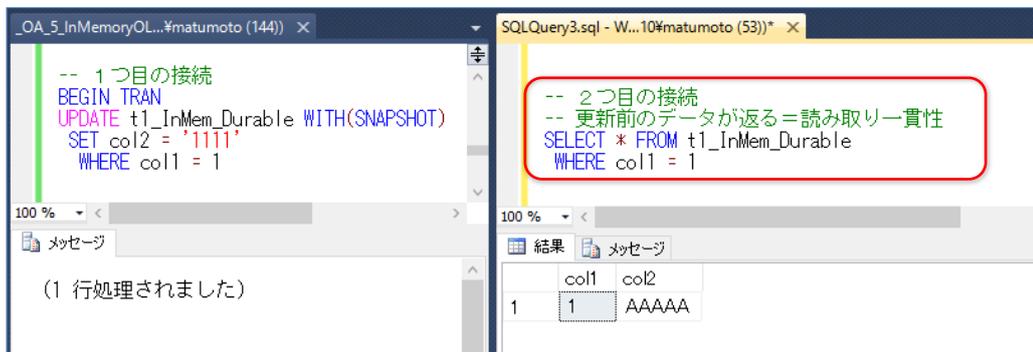
次に、別の接続（ユーザー）から、同じデータを参照してみましょう、

- 別の接続を作成するには、ツールバーの「新しいクエリ」ボタンをクリックします。

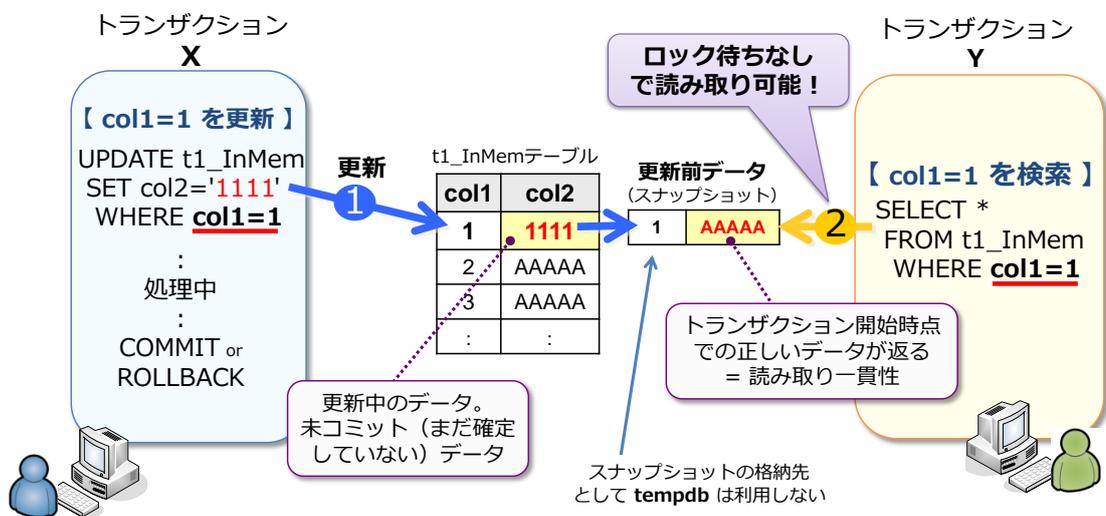


- 新しく表示されたクエリ エディター（別の接続）から、更新中のデータ「col1 = 1」を参照してみます。

```
SELECT * FROM t1_InMem_Durable
WHERE col1 = 1
```



結果は、更新前のデータ（AAAAA）が表示されることを確認できます。

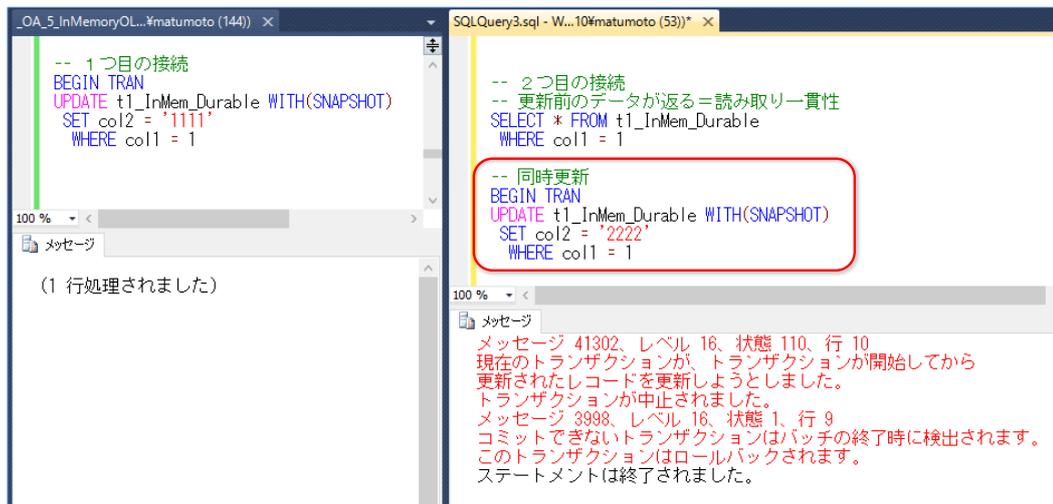


メモリ最適化テーブルは、スナップショット分離レベルで動作するので、トランザクション中

のデータは更新前のデータを返すことで、読み取り一貫性を実現しています。また、ロックを利用しない、マルチバージョンの楽観的同時実行制御も実装されています（従来型のスナップショット分離機構のように、更新前のデータが tempdb へ保存されることもありません）。

4. 続いて、更新中のデータ「col1 = 1」に対して、同時更新をしてみます。

```
BEGIN TRAN
UPDATE t1_InMem_Durable WITH(SNAPSHOT)
SET col2 = '2222'
WHERE col1 = 1
```

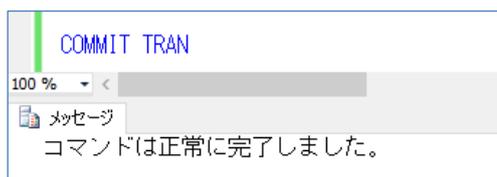


結果は、エラー「41302」が発生して同時更新が検出され、トランザクションが中止されていることを確認できます。また、トランザクションがロールバック（取り消し）されていることも確認できます。

このように、メモリ最適化テーブルでは、同時更新が発生した場合は、先に更新した方が勝つ方式（First Writer Win）が採用されています。

5. 最後に、最初の接続側のクエリ エディターへ戻って、**COMMIT TRAN** を実行して、トランザクションを完了しておきます。

```
COMMIT TRAN
```



Note : ネイティブ コンパイル ストアド プロシージャでの同時更新発生時の再処理

ネイティブ コンパイル ストアド プロシージャでは、BEGIN TRY や THROW を利用することもできるので、同時更新が発生した場合（エラー 41302 が発生した場合）に再試行を行うような作り込みも可能です。これに

については、SQL Server 2014 実践シリーズの「**No.1 インメモリ OLTP の実践的な利用方法**」で、具体的な実装方法を例に説明しているので、こちらもぜひご覧いただければと思います。

インメモリ OLTP の実践的な利用方法

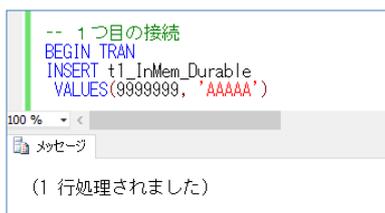
https://www.microsoft.com/ja-jp/sqlserver/2014/technology/self-learning.aspx#practical_contents

➡ INSERT 競合 (同じ値を INSERT)

次に、同じ値を INSERT した場合の競合についても見ていきましょう。

1. まずは、1 つ目の接続から、次のように **INSERT** ステートメントを実行して、「col1」が「9999999」のデータを追加します。

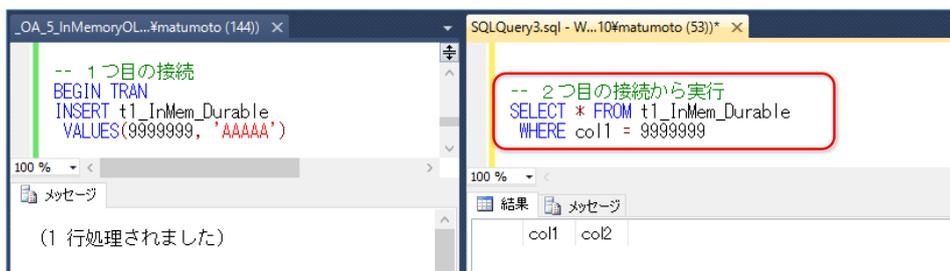
```
BEGIN TRAN
INSERT t1_InMem_Durable
VALUES (9999999, 'AAAAA')
```



BEGIN TRAN でトランザクションを開始し、**COMMIT TRAN** を省略して、トランザクション中のままにしておきます。

2. 次に、ツールバーの【新しいクエリ】ボタンをクリックして、2 つ目の接続を作成します。新しく表示されたクエリ エディター (別の接続) から、追加中のデータ「col1 = 9999999」を参照します。

```
SELECT * FROM t1_InMem_Durable
WHERE col1 = 9999999
```

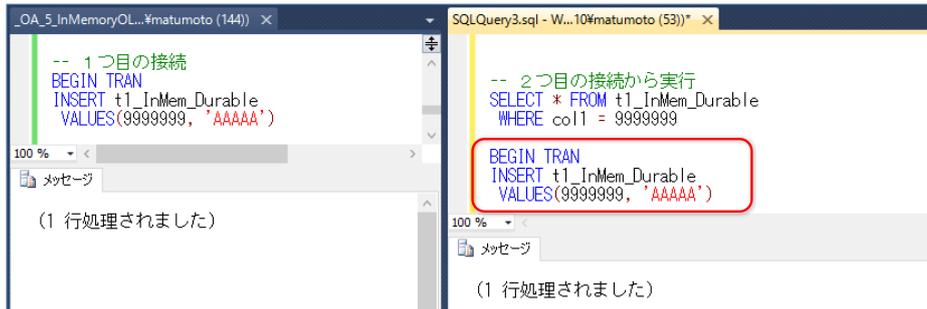


結果は、データが表示されず、追加中のデータは参照できないことを確認できます。

3. 続いて、同じデータを追加してみます。

```
BEGIN TRAN
INSERT t1_InMem_Durable
```

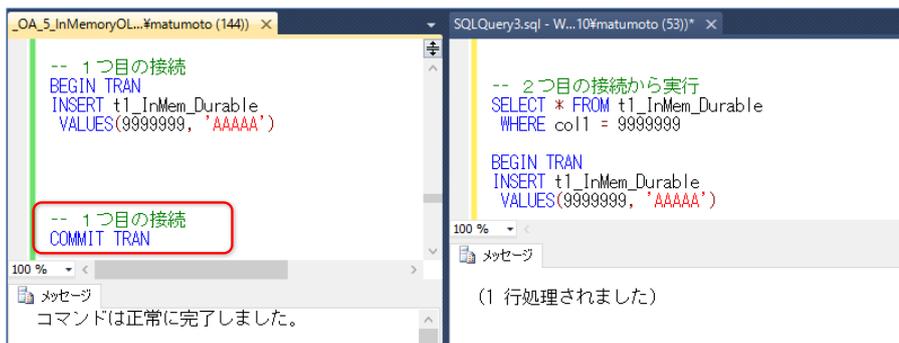
```
VALUES(9999999, 'AAAAA')
```



UPDATE ステートメントのときは、同じデータを更新すると、その時点でエラーが発生しましたが、INSERT ステートメントの場合は、この時点ではエラーが発生しません。

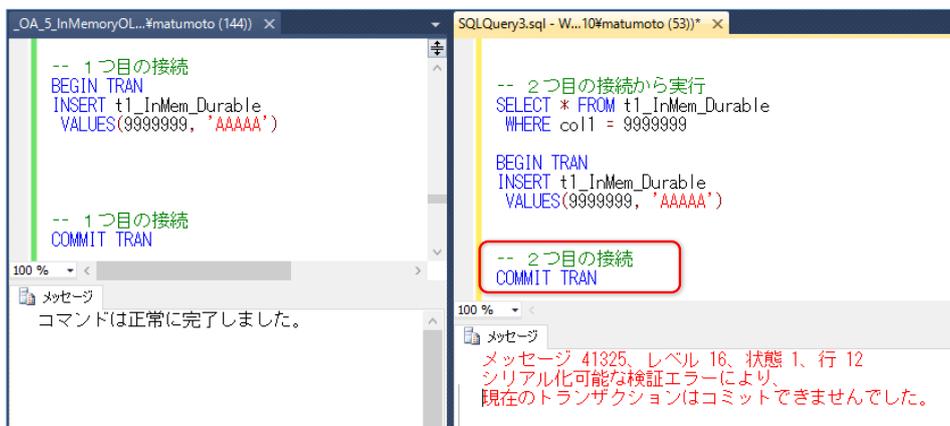
4. 次に、最初の接続側へ戻って、**COMMIT TRAN** を実行して、トランザクションを完了します。

```
COMMIT TRAN
```



5. 次に、2つ目の接続側で、**COMMIT TRAN** を実行します。

```
COMMIT TRAN
```



今度は、エラー「**41325**」が発生して同時更新が検出され、トランザクションが失敗していることを確認できます（これによって、トランザクションがロールバックされます）。このように、メモリ最適化テーブルでは、同じ値の INSERT が発生した場合は、先に追加した方（先に COMMIT した方）が勝ちます。

➡ おわりに

最後まで試された皆さん、いかがでしたでしょうか？ インメモリ OLTP とクラスター化列ストア インデックスの融合（Operational Analytics）は、今後のデータベースのあり方を左右するのではないかと思えるほど、非常に大きな可能性を感じています（弊社のお客様でも、インメモリ OLTP に移行できるケースが多々あるのではないかとワクワクしています）。

また、非クラスター化列ストア インデックスが更新可能になったことも、私たちは大変気に入っています。こちらはテーブルを移行することなく、インデックスを追加するだけで利用できるものなので、皆さんも、ぜひ試してみてくださいと思います。

SQL Server 2016 では、R 統合や JSON 対応、PolyBase、Hadoop 対応など、最近のマイクロソフト社の方向性と同様、SQL Server でもオープン化が非常に進んでいます。また、SQL Server 2016 では、セキュリティの強化や、既存機能の強化も怠ってはならず、私たちが特に気に入っているのは「動的データ マスク」と「透過的なデータ暗号化の性能向上&インメモリ OLTP 対応」、「可用性グループでのログ転送性能の向上」、「ライブ クエリ統計」、「クエリ ストア」です。

これらについては、SQL Server 2016 自習書シリーズ No.1「**SQL Server 2016 の新機能の概要**」編で説明しているので、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 20 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く、支援だけでなく実際の構築も行う。

主なコンサルティング実績/構築実績

- ▶ 大手製造業の「CAD 端末の利用状況の見える化」システム構築
Oracle や CSV (Notes)、TSV ファイル、Excel からデータを抽出し、SQL Server 2012 上に DWH を構築
見える化レポートには Reporting Services を利用
- ▶ 大手映像制作会社の BI システム構築 (会計/業務システムにおける予実管理/原価管理など)
従来 Excel で管理していたシートを Reporting Services のレポートへ完全移行。
Oracle や勘定奉行からデータを抽出して、SQL Server 上に DWH を構築
- ▶ 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/ABC 分析/ポイントカード分析)
- ▶ 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)
Reporting Services による勤怠状況の見える化レポートの作成、PostgreSQL/人事システムからのデータ抽出
- ▶ 外資系医療メーカーの BI システム構築支援 (Analysis Services と Excel による販売分析システム)
OLAP キューブによる売上および顧客データの多次元分析/自由分析 (ユーザーによる自由操作が可能)
- ▶ 大手流通系の DWH システムのパフォーマンス チューニング
データ量 100 億件の DWH、総ステップ数 2 万越えのストアド プロシージャのパフォーマンス チューニング
- ▶ ミッション クリティカルな金融システムでのトラブル シューティング/定期メンテナンス支援
- ▶ SQL Server の下位バージョンからの移行/アップグレード支援 (32 ビットから x64 への対応も含む)
- ▶ 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)
- ▶ 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
- ▶ 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)
- ▶ 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。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 2016 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。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。