

# SQL Server 2016

SQL Server 2016 CTP 3.2 自習書シリーズ No.1

---

SQL Server 2016 CTP 3.2 の新機能の概要

---

Published: 2016 年 1 月 31 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 CTP 3.2 の概要 .....	4
1.1	SQL Server 2016 CTP 3.2 のダウンロード .....	5
1.2	SQL Server 2016 CTP 3.2 で提供された主な新機能 .....	7
STEP 2.	Operational Analytics OLTP とデータ分析の両立 .....	21
2.1	Operational Analytics の概要 ～OLTP とデータ分析の両立～ .....	22
2.2	Operational Analytics の検証の詳細 .....	28
2.3	Operational Analytics の検証環境 .....	40
STEP 3.	セキュリティ強化 .....	41
3.1	セキュリティの強化 .....	42
3.2	動的データ マスクによる情報漏洩対策 .....	43
3.3	行レベル セキュリティによるセキュリティ強化 .....	51
3.4	Always Encrypted による列データの暗号化 .....	57
3.5	TDE (透過的なデータ暗号化) の性能向上、インメモリ OLTP 対応 .....	75
3.6	テンポラル テーブルによる Audit (監査証跡) .....	85
STEP 4.	注目の新機能を試してみよう .....	99
4.1	SQL Server R Services (R 統合) .....	100
4.2	JSON 対応 (FOR JSON、OPENJSON、JSON_VALUE など) .....	112
4.3	Stretch Database ～アクセス頻度の低いデータをクラウドへ～ .....	119
4.4	Azure バックアップ URL の性能向上 (ブロック BLOB 対応) .....	134
4.5	PolyBase で Hadoop アクセス (Hortonworks、HDInsight) .....	141
STEP 5.	既存機能の強化 .....	152
5.1	SQL Server 2014 からの主な変更点 .....	153
5.2	ライブ クエリ統計 (Live Query Statistics) .....	158
5.3	クエリ ストアで性能監視、プラン固定 .....	160
5.4	Transact-SQL (T-SQL) の強化 .....	180
5.5	AlwaysOn 可用性グループの拡張 .....	181
5.6	BI 関連の強化 .....	184
5.7	付録：サンプル データベース (NorthwindJ) の作成 .....	190

## STEP 1. SQL Server 2016 CTP 3.2 の概要

---

この STEP では、SQL Server の次期バージョンである「**SQL Server 2016**」の CTP 3.2 で提供された主な新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 CTP 3.2 のダウンロード
- ✓ SQL Server 2016 CTP 3.2 の主な新機能



## 1.1 SQL Server 2016 CTP 3.2 のダウンロード

SQL Server の次期バージョンである「**SQL Server 2016**」は、製品出荷前のプレビュー版である「**CTP 3.2**」が、2015 年 12 月から公開されています。この **CTP 3.2** は、次の URL からダウンロードすることができます。

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



**CTP** は、**Community Technology Preview** の略で、従来の Beta 版（プレビュー版）と同じ位置付けのものです。これまでのバージョンの SQL Server では、CTP 1、CTP 2、CTP 3 という形で CTP が提供されていましたが、今回の SQL Server 2016 からは CTP 2、CTP 2.1、…、CTP 2.4、CTP 3、CTP 3.1、CTP 3.2 のように数ヶ月ごとに最新の機能が盛り込まれた CTP が提供されるようになりました。この自習書は、2015 年 12 月に提供された CTP 3.2 をベースに記述していますが、CTP 3.3 などの最新版が登場した場合には、ぜひそちらで試してみてください。

この自習書では、**CTP 3.2** で提供された **SQL Server 2016 の新機能**を簡単に試せるように、ステップ バイ ステップ形式で画面ショット満載で紹介していますので、ぜひ、皆さんも実際に試しながらこの自習書を読み進めていただければと思います。

## ➡ Management Studio を別途ダウンロード可能に

SQL Server 2016 からは、Management Studio の最新版を単独でダウンロードできるようになりました。CTP が出た直後であれば、CTP に含まれている Management Studio が最新版になりますが、時間が空いた場合には、最新版が提供されていないかチェックしてみることをお勧めします。最新版の Management Studio は、次の URL からダウンロードできます。

Management Studio の最新版のダウンロード

<http://msdn.microsoft.com/ja-jp/library/mt238290.aspx>

MSDN ライブラリ

開発ツールと言語ドキュメント

デザインツール

モバイルおよび Embedded 開発

.NET 開発

Office の開発

Office ソリューション開発

patterns & practices

サーバー および エンタープライズ開発

Microsoft Azure

Web 開発

Windows 開発

Windows ドライバー開発

テクニカルドキュメント

## SQL Server Management Studio のダウンロード

SQL Server Management Studio (SSMS) は、SQL Server のすべてのコンポーネントを構成、管理、開発し、それらのコンポーネントへアクセスできるようにする統合環境です。SSMS では、さまざまなグラフィック ツールと、機能の豊富な多くのスクリプト エディターが用意されており、これらを使用して、あらゆるスキル レベルの開発者や管理者が SQL Server にアクセスできます。

このリリースは、SQL Server 2005 から SQL Server 2016 までをサポートし、多くの新機能を備えています。たとえば、SQL Server の以前のバージョンとの互換性の強化、軽量のスタンドアロン Web インストーラー、新しいリリースが入手可能になると SSMS 内で表示されるトースト通知などがあります。このリリースは、Azure SQL Database の最新のクラウド機能の使用も、最高レベルでサポートしています。

メモ

SQL Server Management Studio のこのバージョンは、同じコンピューターにインストールされている以前のリリース バージョンとの共存に対応しています。

### SQL Server Management Studio プレビュー

この最新プレビューのバージョン番号は **13.0.800.111** です  
SSMS プレビュー リリースは、入手可能な最新のサービス パックと併せて使用される場合、次のプラットフォームをサポートします：  
Windows 10、Windows 8、Windows 8.1、Windows 7、Windows Server 2012 (64 ビット)、  
Windows Server 2008 R2 (64 ビット)

SQL Server Management Studio November 2015 プレビューをダウンロードする

## 1.2 SQL Server 2016 CTP 3.2 で提供された主な新機能

SQL Server 2016 CTP 3.2 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"> <li>• <b>Operational Analytics</b> (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</li> <li>• <b>インメモリ OLTP の飛躍的な進化</b> 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など</li> <li>• <b>列ストア インデックスの大幅強化</b> 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など</li> </ul>
セキュリティ強化	<ul style="list-style-type: none"> <li>• <b>動的データ マスク</b>によるデータのマスク (情報漏洩対策)</li> <li>• <b>行レベル セキュリティ</b>による行レベルのアクセス制御</li> <li>• <b>Always Encrypted</b> による暗号化</li> <li>• <b>TDE</b> (透過的なデータ暗号化) の性能向上、インメモリ OLTP 対応</li> <li>• <b>テンポラル テーブル</b>による Audit</li> </ul>
注目の新機能	<ul style="list-style-type: none"> <li>• <b>R 統合</b> (SQL Server R Services)</li> <li>• <b>JSON 対応</b></li> <li>• <b>Stretch Database</b> (ストレッチ データベース)</li> <li>• <b>Azure バックアップ URL</b> の性能向上</li> <li>• <b>PolyBase</b> で <b>Hadoop アクセス</b> (HDFS)</li> </ul>
可用性	<ul style="list-style-type: none"> <li>• <b>AlwaysOn 可用性グループの強化</b> 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</li> </ul>
既存機能の強化	<ul style="list-style-type: none"> <li>• <b>DROP .. IF EXISTS</b> (オブジェクトが存在しているなら DROP を実行)</li> <li>• <b>tempdb</b> 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</li> <li>• <b>トレースフラグ 1117、1118、4199</b> が既定でオンに</li> <li>• <b>ライブ クエリ統計</b></li> <li>• <b>クエリ ストア</b>による性能監視、プラン固定</li> </ul>
BI 関連の強化	<ul style="list-style-type: none"> <li>• <b>Reporting Services の大幅強化</b> 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、Datazen 統合 (モバイル レポート対応、新しいレポート マネージャー) など</li> <li>• <b>Integration Services の強化</b> 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</li> <li>• <b>Analysis Services の強化</b> DirectQuery の新しい実装/大幅な性能向上、DBCC コマンドのサポート、Tabular Model Scripting Language (TMSL) の提供、AMO (Analysis Services Management Objects) の変更 など</li> <li>• <b>MDS の強化</b> 性能の向上、セキュリティの強化、各種の機能強化 など</li> </ul>

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。特に、**Operational Analytics** (OLTP とデータ分析の両立) 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。

**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。

この章では、これらの新機能の概要を説明します。

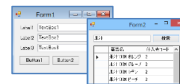
## ➡ 性能向上に関する新機能 ～目玉の新機能は Operational Analytics～

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ろりをして、**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



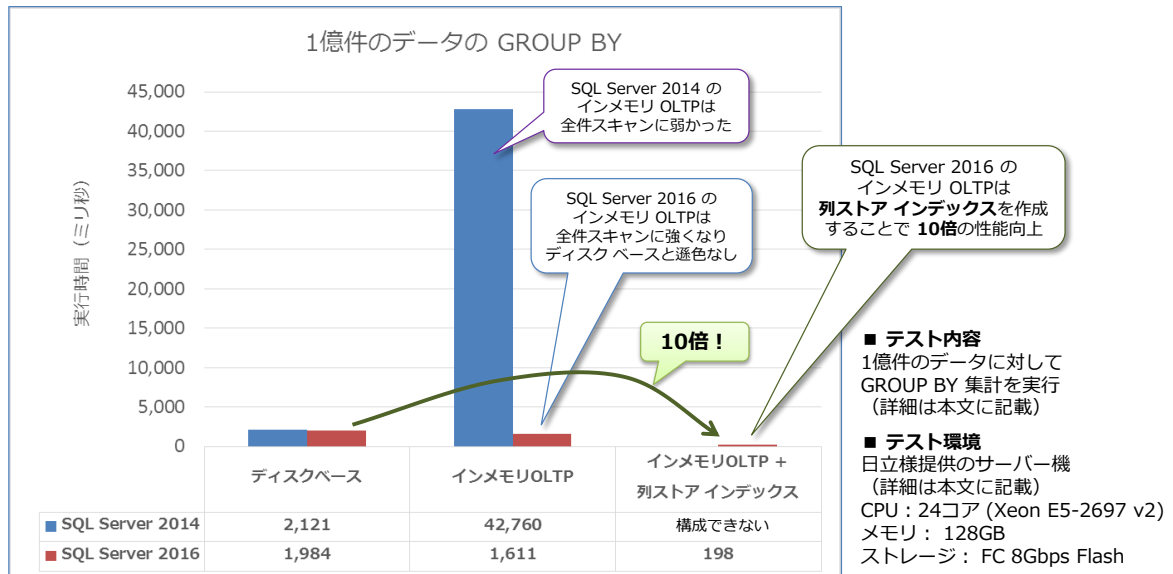
### SQL Server 2016 ではどちらも実現 (両立) 可能

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、どちらも両立できるようになりました。

実際に、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 (詳細は 2 章に記載しています)。



\* 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。

\* 検証は、SQL Server 2016 CTP 3.2 を利用して行ったので、RTM 版ではさらに性能が向上する可能性があります。

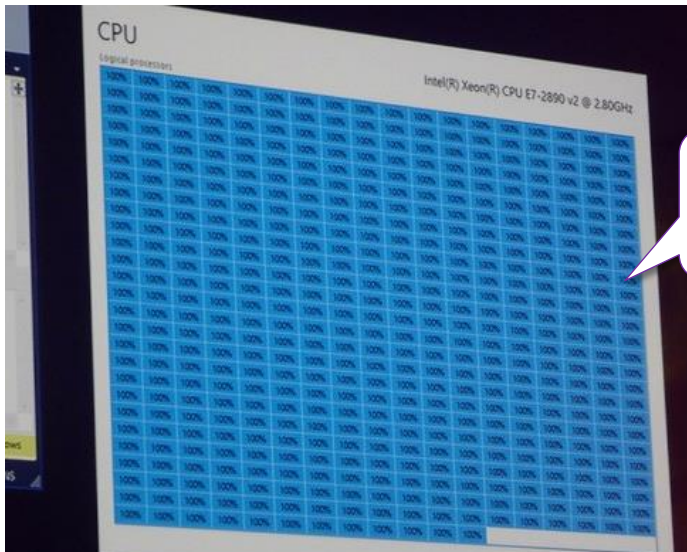
ディスクベースの通常テーブルでは **1.98 秒** かった集計処理が、インメモリ OLTP にすることで **1.6 秒** (18.8%の性能向上)、さらに**列ストア インデックスを作成**することで、わずか **198 ミリ秒**で処理できるようになり、**10 倍もの性能向上**を確認することができました。

このように、インメモリ OLTP と列ストア インデックスを組み合わせれば、**完全なインメモリ**で OLTP とデータ分析を両立できるようになります。もちろん、メモリをそこまで搭載できないという場合には、**列ストア インデックスのみを単独**で利用するという選択肢もあります。SQL Server 2016 の列ストア インデックスでは、今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**になったり、**クラスター化列ストア インデックスに追加の B-tree インデックスを作成**できるようになったり、**PRIMARY KEY 制約がサポート**されるようになったりしているので、既存環境から大きな変化を加えることなく利用できるようになりました。また、バッチ モード (列ストア インデックスの内部的な動作モード) の性能向上も実現しているので、より **Analytics ワークロード** (データ分析/集計) に強くなっています。

これらのインメモリ OLTP と列ストア インデックスの強化ポイントについては、2 章で詳しく説明しています。

## 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** **すべてのスレッドが 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

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。

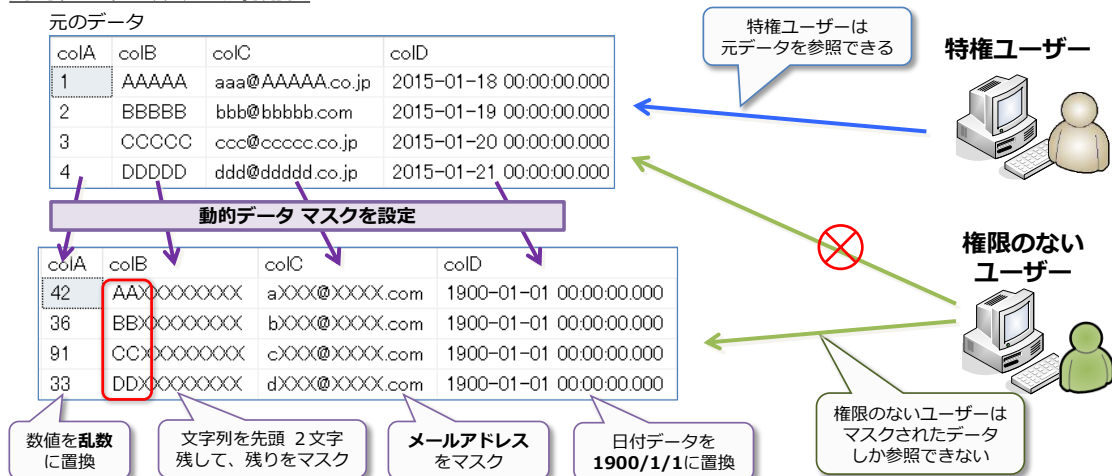
## ➡ セキュリティ強化に関する新機能

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ます。その主なものは、次のとおりです。

### 動的データ マスク (Dynamic Data Masking)

動的データ マスクは、クレジットカード番号やマイナンバーなどの顧客情報／機密情報をマスク (別の値に置換) して、情報漏洩を防止できる機能です (詳細は 3 章で説明)。

#### 動的データ マスクの利用例

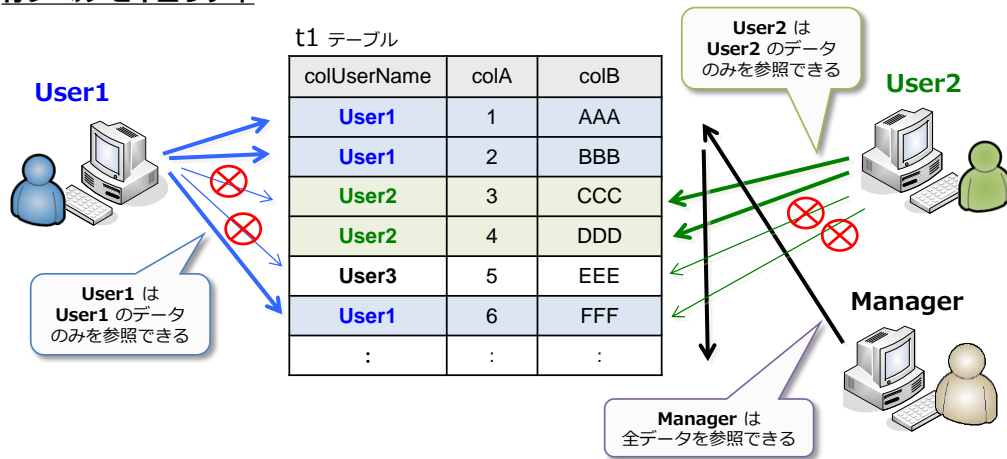




## 行レベル セキュリティ (Row Level Security)

行レベル セキュリティは、行レベルのアクセス制御を実現できる機能で、ユーザーごとに、参照できる行データを制限することができる機能です (詳細は 3 章で説明)。

### 行レベル セキュリティ



## Always Encrypted による列データの暗号化

Always Encrypted は、ネットワーク上を流れるデータも、データベース内に格納されるデータも、すべて暗号化して格納できる機能です (列データを暗号化して、アプリケーションも透過的に利用できます。詳細は 3 章で説明)。

-- Always Encrypted 設定後のデータの確認  
SELECT \* FROM aeTest1

100 %

結果 メッセージ

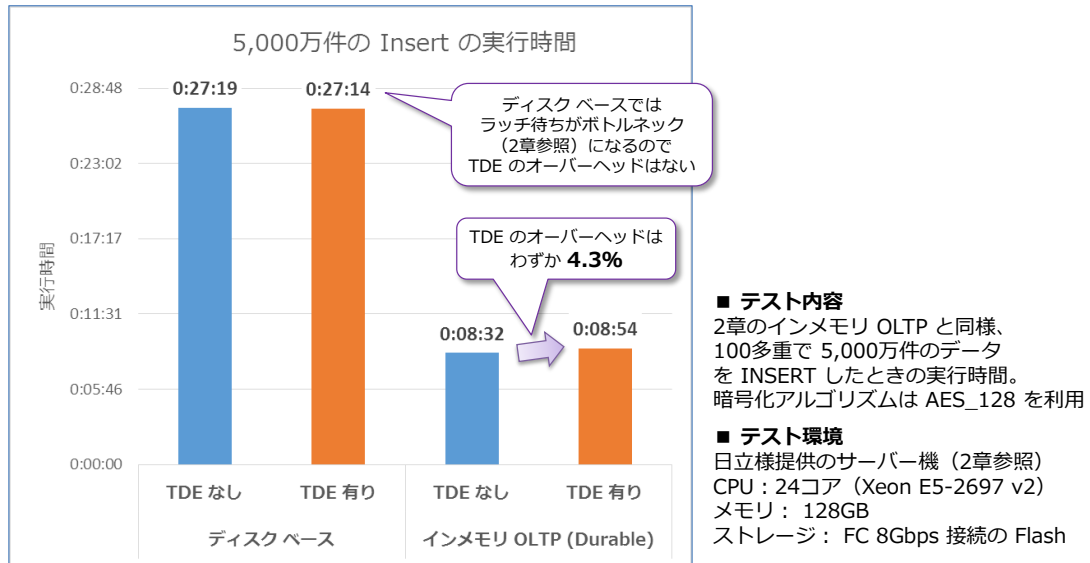
	colA	colB	colC
1	1	0x01B44A66DBF0B586803561098905B0EFEEA...	0x0112FD74802543520C46C5F9135E0B80DDDEA024AE4A000...
2	2	0x0137A2B5E14A4F1E8C760651254DA458349...	0x016747C02567D8F26EA69C61F35B6980715E95EFF0E9822F...

Always Encrypted で列データを暗号化

## TDE (透過的なデータ暗号化) の性能向上、インメモリ OLTP 対応

TDE (透過的なデータ暗号化) は、Intel の **AES-NI** (AES 暗号化の処理を高速化するための命令セット、ハードウェア アクセラレータ) に対応して、性能向上を実現しました。また、インメモリ OLTP を利用している場合にも、TDE が利用できるようになりました。

TDE を設定したことによる性能のオーバーヘッド (暗号化処理のオーバーヘッドなど) を検証したのが次のグラフです。

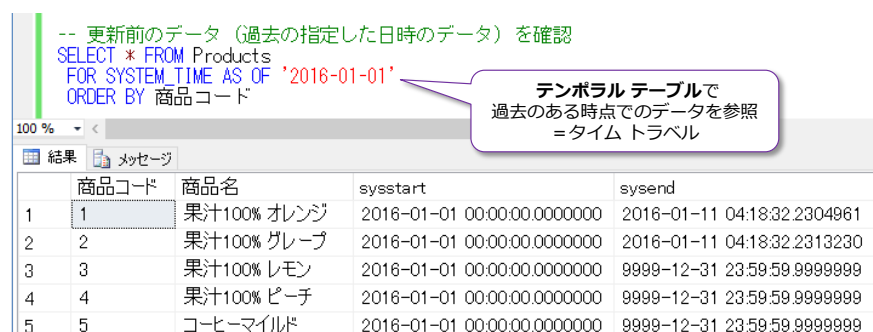


\* ベンチマークの結果の公表は、使用許諾契約書で禁止されていますが、その効果をわかりやすく表現するため、日本マイクロソフト株式会社の監修のもと、数値を掲載しております。

ディスク ベースの通常テーブルでは、ラッチ待ちがボトルネックになってしまうので、TDE を設定しても、設定しなくても**結果は変わらない** (オーバーヘッドは確認できない)、**インメモリ OLTP の Durable (SCHEMA\_AND\_DATA)** では、TDE を設定しない場合が **8 分 32 秒** であるのに対して、TDE を設定したとしても **8 分 54 秒** で完了、**わずか 4.3%** のオーバーヘッドであることを確認できました (詳細は 3 章で説明)。

### テンポラル テーブルを利用した Audit (コンプライアンスにおける監査証跡)

SQL Server 2016 では、過去の更新履歴を自動保存できる「**テンポラル テーブル**」機能を利用して、Audit (監査証跡) を実現できます。テンポラル テーブルは、操作ミス時のデータ復旧や、データ分析における SCD (緩やかに変化するディメンション) シナリオでも利用できます。テンポラル テーブルでは、次のように過去のデータを簡単に参照することができます。





## ➡ その他の注目の新機能

SQL Server 2016 では、R 統合や、JSON 対応、Hadoop 対応など、数多くの新機能が提供されています。その主なものは、次のとおりです。

### ● SQL Server R Services (R 統合)

**SQL Server R Services** は、Transact-SQL ステートメントを利用して、**R スクリプト** / **ライブラリ**を実行することができる機能です。

```
-- sum
EXEC sp_execute_external_script
  @language = N'R',
  @script = N'sum1 <- sum(input1[,1]);
             ret <- data.frame(sum1);',
  @input_data_1 = N'SELECT colA, colB FROM maskTestDB..maskTest1',
  @input_data_1_name = N'input1',
  @output_data_1_name = N'ret'
WITH RESULT SETS ( ( col1 int ) )
```

R スクリプトを記述

SQL Server 上のテーブルデータを R スクリプトでの処理データとして与えることができる

col1
6

R スクリプトで処理した結果

### ● JSON 対応

SQL Server 2016 では、**JSON** (JavaScript Object Notation) に対応して、SQL Server 上のテーブルデータを JSON 形式で出力したり、JSON データを SQL Server に取り込んだりできるようになりました (**FOR JSON** や **OPENJSON**、**ISJSON**、**JSON\_VALUE**、**JSON\_QUERY** など、JSON データを処理するための多数の関数を提供)。

```
DECLARE @jsonstr nvarchar(max) =
N'["arri": [
  [{"key1": "test1", "key2": 999}],
  [{"key1": "test2", "key2": 777}]
]']

SELECT * FROM
OPENJSON(@jsonstr, '$.arri')
WITH (
  key1 varchar(20),
  key2 int
)
```

JSON 形式のデータ

OPENJSON 関数で JSON データを取得

```
-- ISJSON 関数で CHECK 制約を設定
CREATE TABLE jsonTest1
( colA int PRIMARY KEY
, colB nvarchar(max) CONSTRAINT chkJSON CHECK ( ISJSON(colB) > 0 ) )

-- JSON データをテーブルに格納
INSERT INTO jsonTest1 VALUES(1, N'["key1": "test1", "key2": 999]')
INSERT INTO jsonTest1 VALUES(2, N'["key1": "test2", "key2": 777]')

-- テーブル内の JSON データをクエリ
SELECT JSON_VALUE(colB, '$.key1') AS key1
, JSON_VALUE(colB, '$.key2') AS key2
FROM jsonTest1
```

ISJSON 関数で JSON データかどうかをチェック

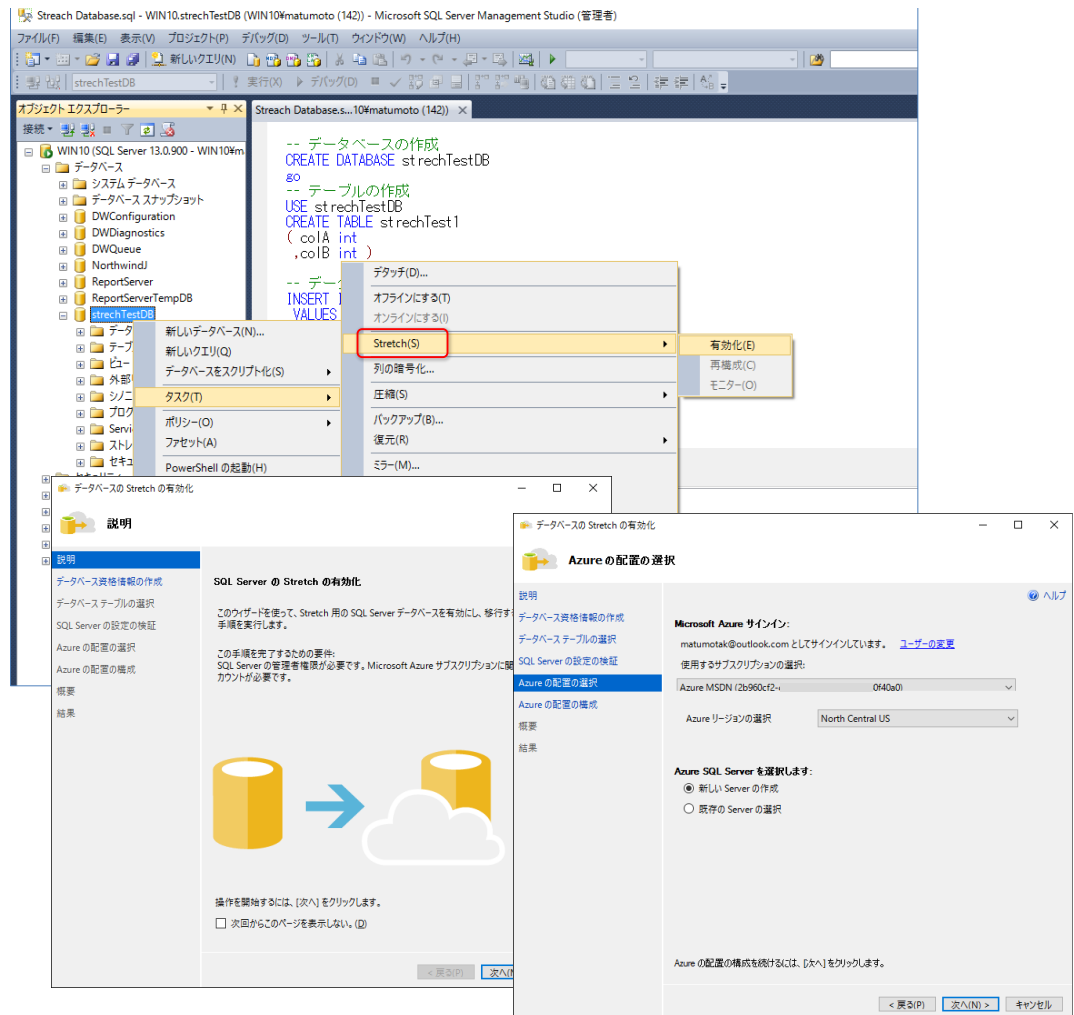
JSON データをテーブルに格納

JSON\_VALUE 関数で JSON データをクエリ

key1	key2
test1	999
test2	777

### ● Strech Database ～アクセス頻度の低いデータをクラウドへ～

Strech Database は、テーブルデータをクラウド(**Microsoft Azure SQL Database**)上に配置することができる機能です。



## ● Azure バックアップ URL の性能向上 (ブロック BLOB 対応)

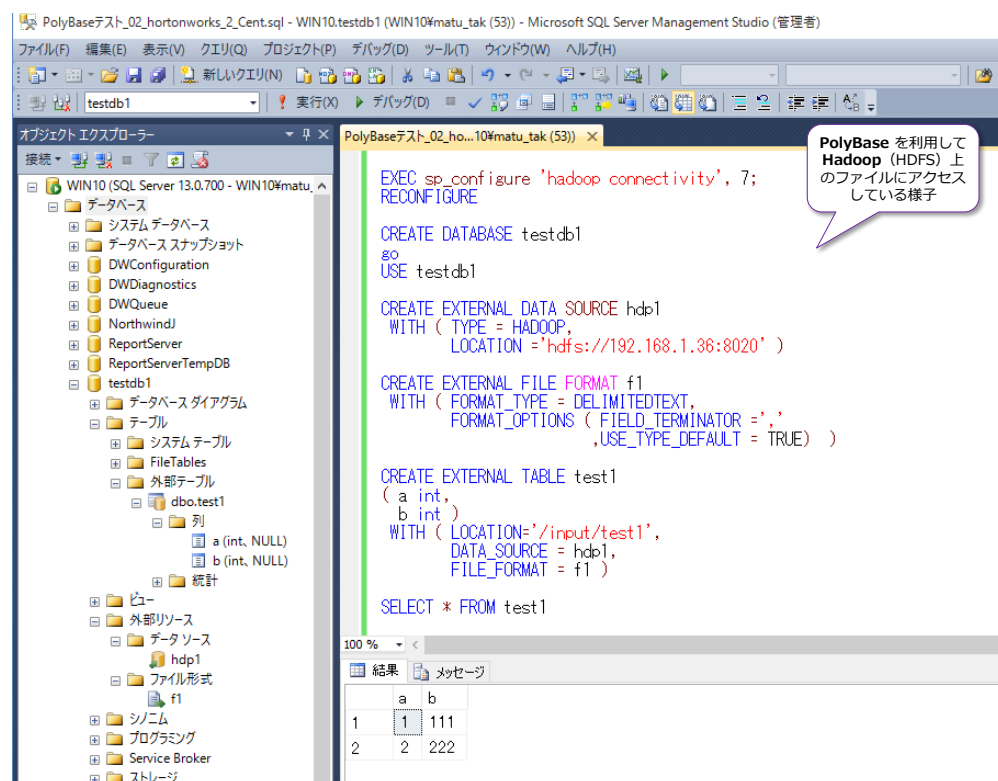
SQL Server では、バックアップを Microsoft Azure の BLOB ストレージ上に配置する機能が SQL Server 2012 の SP1 CU2 から利用することができましたが、これは BLOB ストレージ内の「ページ BLOB」にファイルを格納していました。SQL Server 2016 からは、従来と同様「ページ BLOB」に格納する方法に加えて、「**ブロック BLOB**」に格納する方法が追加されました。



## ● PolyBase で Hadoop アクセス (HDFS)

**PolyBase** は、**Hadoop ファイル システム** (HDFS : Hadoop File System) 上にあるデータを、SQL Server からアクセスできる機能です (この機能は、元々 SQL Server Parallel Data Warehouse : PDW で提供されていた機能ですが、SQL Server 2016 からは、通常の SQL Server でも利用できるようになりました)。

PolyBase を利用すれば、Microsoft Azure 上の Hadoop サービスである「**Microsoft Azure HDInsight**」や、Hortonworks の提供する「**HDP : Hortonworks Data Platform**」、Cloudera の提供する「**CDH : Cloudera's Distribution including Apache Hadoop**」上のデータに対して、SQL Server からアクセスできるようになります。また、PolyBase では、Microsoft Azure 上の「**BLOB ストレージ**」にアクセスすることもできます。



## ➡ 既存機能の強化

SQL Server 2016 では、既存機能の強化も怠っていません。その主なものは、次のとおりです。

### ● DROP .. IF EXISTS (オブジェクトが存在している場合に DROP を実行)

これまでの SQL Server では、オブジェクトを削除する場合には、次のようにオブジェクトの存在チェックを行う必要がありました。

```
-- 該当オブジェクトが存在するなら削除 (SQL Server 2014 以前の場合)
IF EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'obj1'))
DROP TABLE obj1
```

SQL Server 2016 からは、「**DROP .. IF EXISTS**」が提供されたことで、上の記述を、次のように記述することができます。

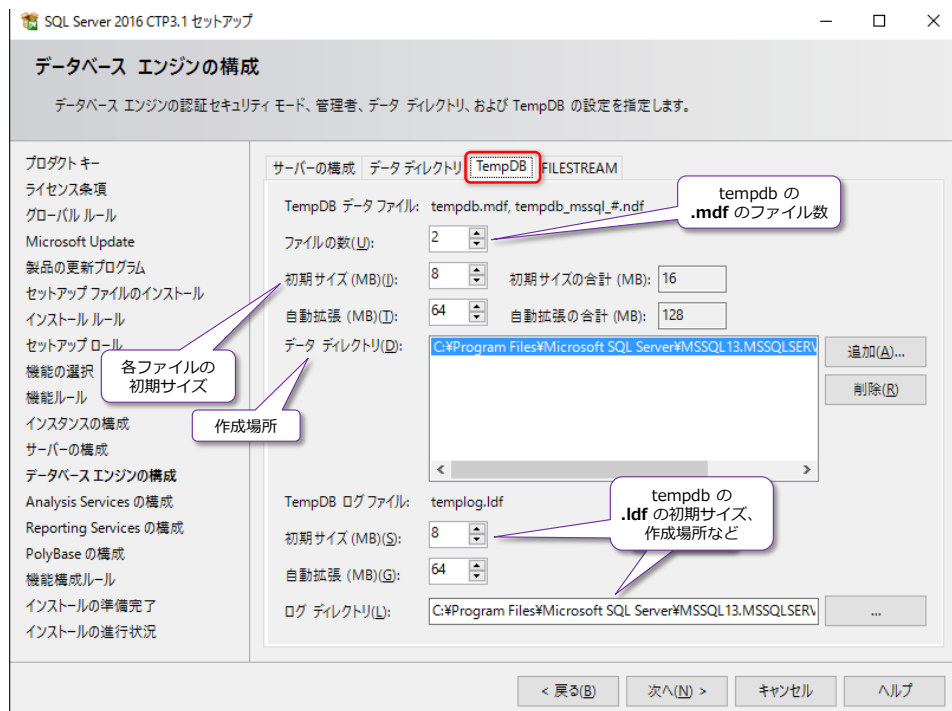
-- 該当オブジェクトが存在するなら削除 (SQL Server 2016 の場合)  
**DROP TABLE IF EXISTS** obj1

## ● インストール時に各種の設定が可能に

SQL Server 2016 では、これまでのバージョンではインストール後に行っていたような設定（ベスト プラクティス値など）を、インストール時に行えるようになりました。

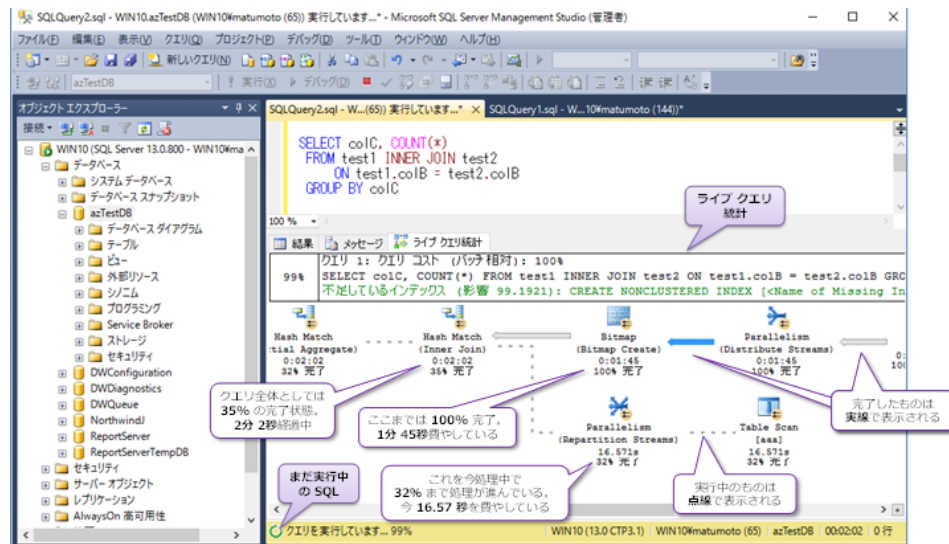


tempdb の複数ファイル／初期サイズもインストール時に設定できるようになりました。



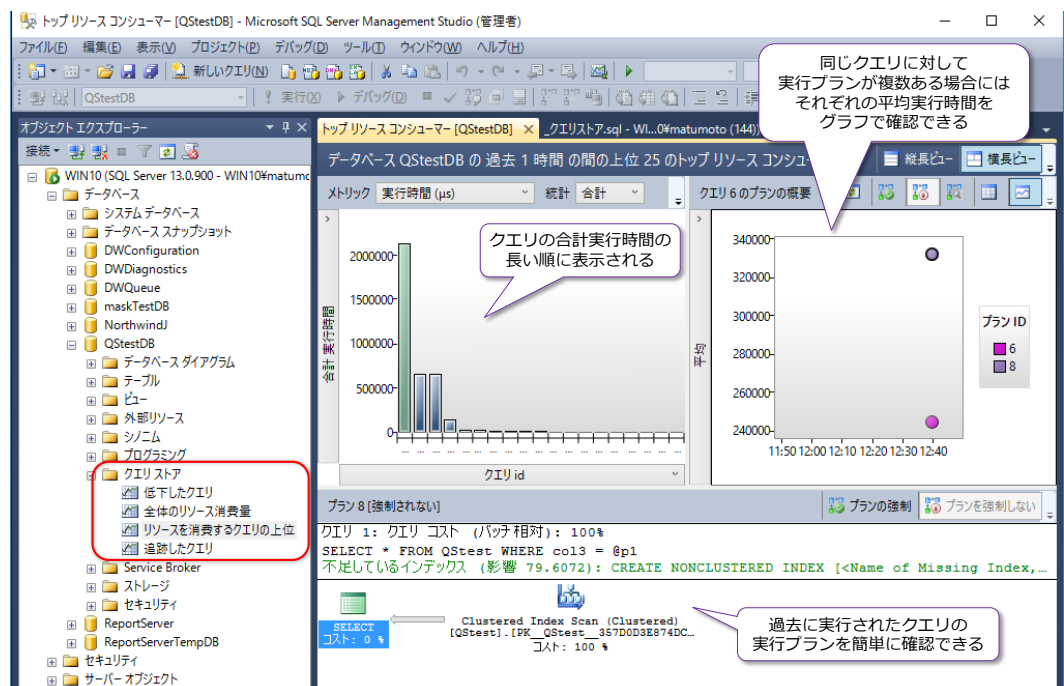
- **tempdb の性能**に関連するトレースフラグ **1117** と **1118** が既定でオンに
- **オプティマイザの動作**に関連するトレースフラグ **4199** が既定でオンに
- **Active Directory** のパスワード認証／統合認証を利用可能に
- **ライブ クエリ統計** (Live Query Statistics)

ライブ クエリ統計は、現在実行中の SQL ステートメントの実行プランのうち、どの部分を実行しているのかを確認できる機能です。



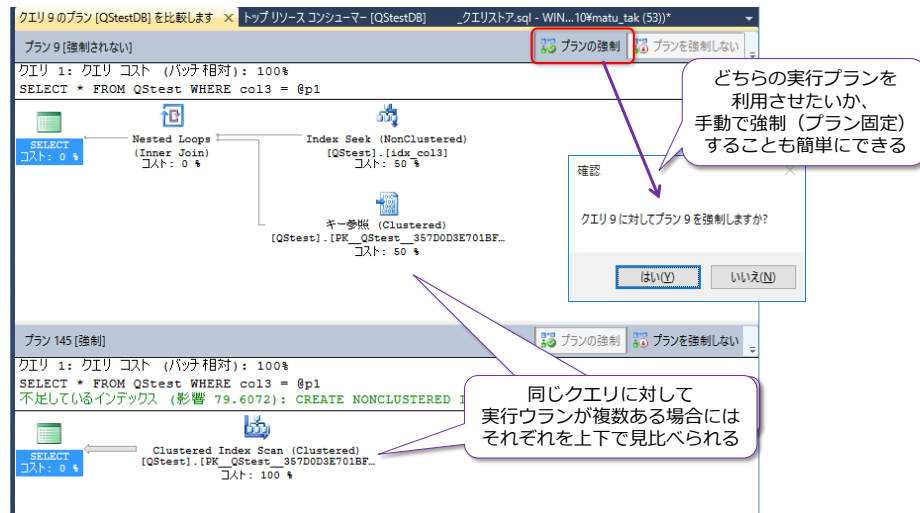
- **クエリ ストア (Query Store)** で性能監視、プラン固定

クエリ ストアは、**クエリの実行履歴** (実行プランを含む) を保存することができる機能です。性能に問題が出た場合に、過去に振り返って、こういった実行プランで実行されていたのかを確認することができます。



また、同じクエリに対して、複数の実行プランがある場合には、次のように簡単に見比べ

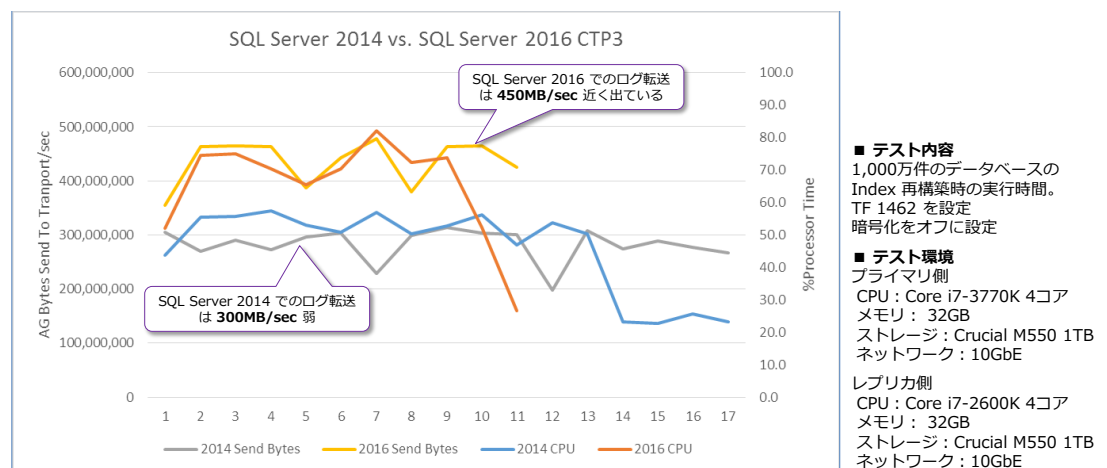
ことができ、かつどちらを利用させたいのかを変更（データベース管理者によるプラン固定）することも簡単にできます。



## ● AlwaysOn 可用性グループの拡張

SQL Server 2016 では、AlwaysOn 可用性グループも強化されました。自動フェールオーバーを構成できる台数が 2 から 3 に増加や、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（透過的なデータ暗号化）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利用可能（Windows Server 2016 を利用する場合）、Standard エディションのサポートなどが提供されました。

実際に、弊社環境で、ログ転送の性能を検証してみたところ、次のような結果を確認することができました。



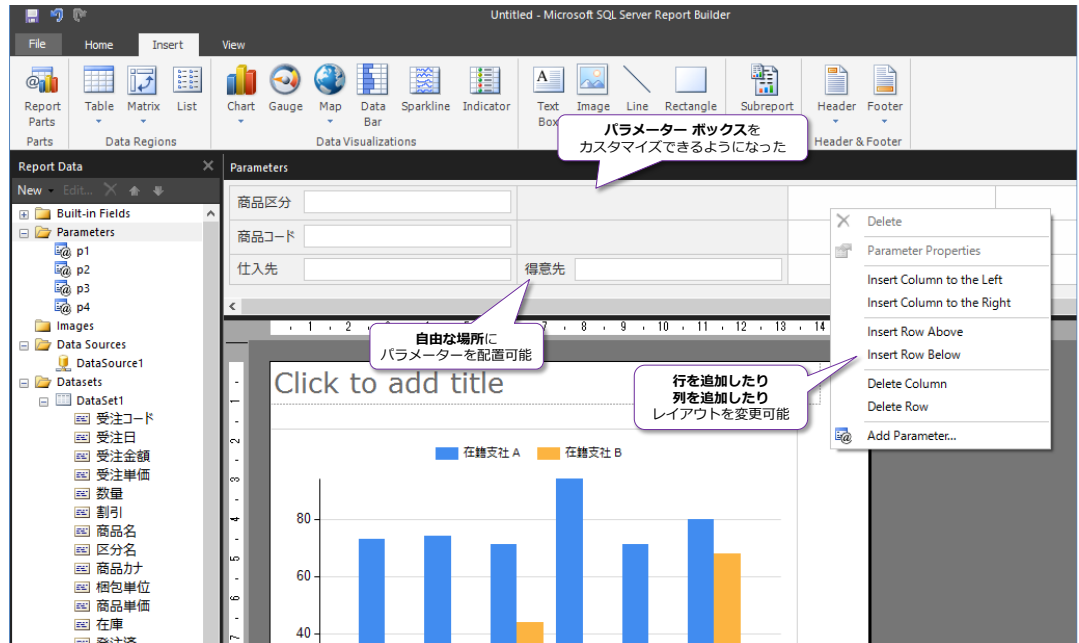
\* ベンチマークの結果の公表は、使用許諾契約書で禁止されていますが、その効果をわかりやすく表現するため、日本マイクロソフト株式会社の監修のもと、数値を掲載しております。  
また、検証は CTP 3.1 で行ったものであるため、RTM 版ではさらに性能が向上する可能性があります。

SQL Server 2014 では **300MB/sec 弱** のログ転送スピード（Bytes send to Transport/sec）であるところを、SQL Server 2016 では **450MB/sec 近く** 出ていることを確認することができました（詳細は 5 章に記載）。

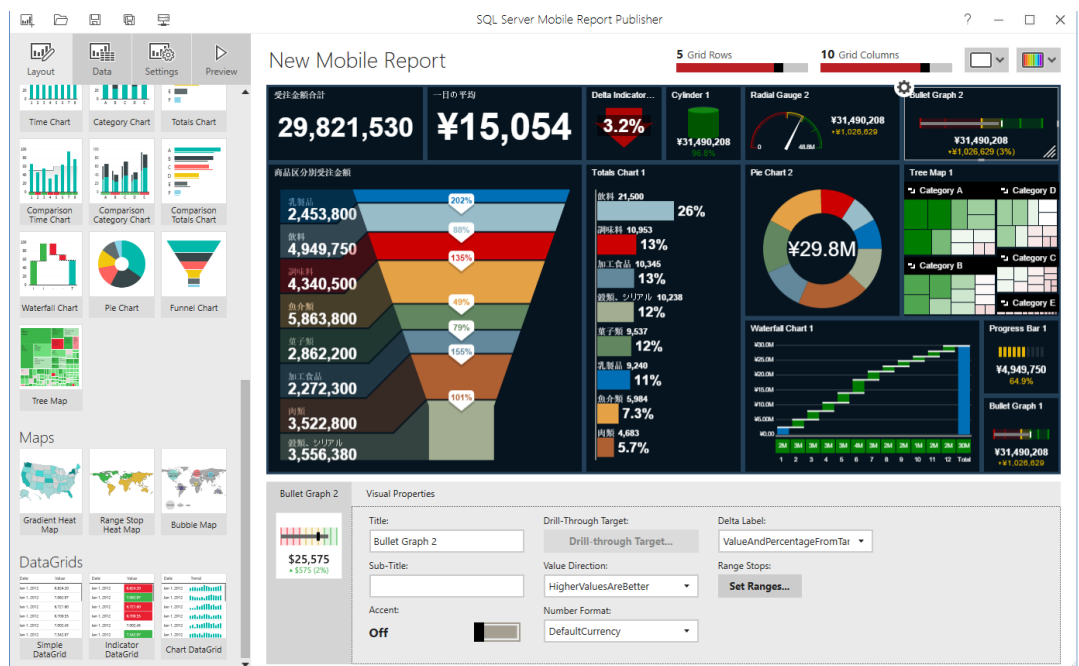


## ● Reporting Services (SSRS) の大幅強化

SQL Server 2016 では、Reporting Services も大幅に強化されました。パラメーターボックスのカスタマイズや、新しいグラフのサポート（サンバースト、ツリーマップ）、新しいレポートビルダー、PowerPoint レンダリング、印刷コントロールの変更（ActiveX コントロールが不要に）、Datazen 統合（モバイル レポート対応、新しいレポート マネージャー）などが提供されました。



Datazen 統合では、次のような見栄えの良いレポートを簡単に作成できるようになり、かつ**モバイル レポート**（スマートフォンやタブレット端末など向けのレポート）にも対応しました（詳細については、別途自習書を作成する予定です）。



- **Integration Services (SSIS) の強化**

Integration Services では、Execute SQL タスクでの **R スクリプト**のサポートや、**Azure Feature Pack** の提供 (Azure BLOB/Azure HDInsight を操作可能なタスク)、**Hadoop** (HDFS)のサポート、インポート/エクスポート ウィザードでの **Azure BLOB** 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** プロパティでのバッファ サイズの自動計算、OData v4/Excel 2013 データソースのサポート、新しい Custom Logging Level、パッケージの増分配置などが提供されました。

- **Analysis Services (SSAS) の強化**

Analysis Services では、DirectQuery の新しい実装や大幅な性能向上、DBCC コマンドのサポート、AMO (Analysis Services Management Objects) の変更、Tabular Model Scripting Language (TMSL) などが提供されました。

- **MDS (マスター データ サービス) の強化**

MDS では、性能の向上や、セキュリティの強化、各種の機能強化 (トランザクションのメンテナンス、多対多リレーションシップ、Excel アドインでのビジネス ルール管理など、多数の機能強化) などが提供されました。

以降では、これらの新機能について、ステップ バイ ステップ形式で画面ショット満載で紹介しているので、ぜひ、皆さんも実際に試しながらこの自習書を読み進めていただければと思います。



## STEP 2. Operational Analytics

### OLTP とデータ分析の両立

この STEP では、SQL Server 2016 の一番の目玉の新機能である「**インメモリ OLTP と列ストア インデックスの融合**」による OLTP とデータ分析の両立について、概要を説明します。

この STEP では、次のことを学習します。

- ✓ **Operational Analytics** (OLTP とデータ分析の両立)
- ✓ **インメモリ OLTP の飛躍的な進化** (データ分析でも利用可能に)
- ✓ **列ストア インデックスの飛躍的な進化** (OLTP でも利用可能に)

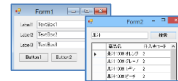
## 2.1 Operational Analytics の概要 ～OLTP とデータ分析の両立～

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 ではどちらも実現(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、どちらも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

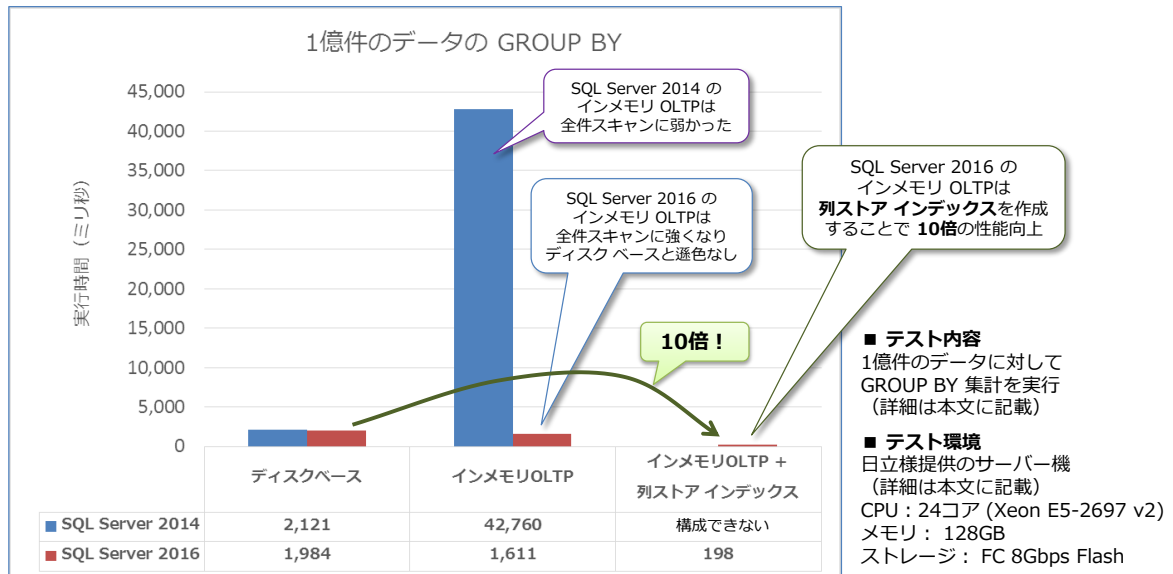
- **インメモリ OLTP で列ストア インデックスのサポート** (インメモリ OLTP のメモリ最適化テーブルにクラスター化列ストア インデックスを追加可能に)
- **最大サイズが 2TB に拡張** (大量データへの対応)
- **64 コア以上での性能向上** に対応 (スケールアップが可能)
- **並列プラン**への対応 (メニー コアの活用が可能に)
- **TDE** (透過的なデータ暗号化) のサポート (セキュリティの強化)
- **ALTER/BIN2 以外の照合順序**のサポート (既存環境からの移行しやすさが大幅に向上)

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。

\* 検証は、SQL Server 2016 CTP 3.2 を利用して行っただけで、RTM 版ではさらに性能が向上する可能性があります。

このグラフは、**SQL Server 2016 の CTP 3.2** 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）。CTP 3.2 にはデバッグ コードが含まれていたり、チューニングが不十分な状態であるにもかかわらず、大きな性能向上を確認することができました（チューニングが施された **RTM 版**ではさらに性能が向上する可能性があります）。

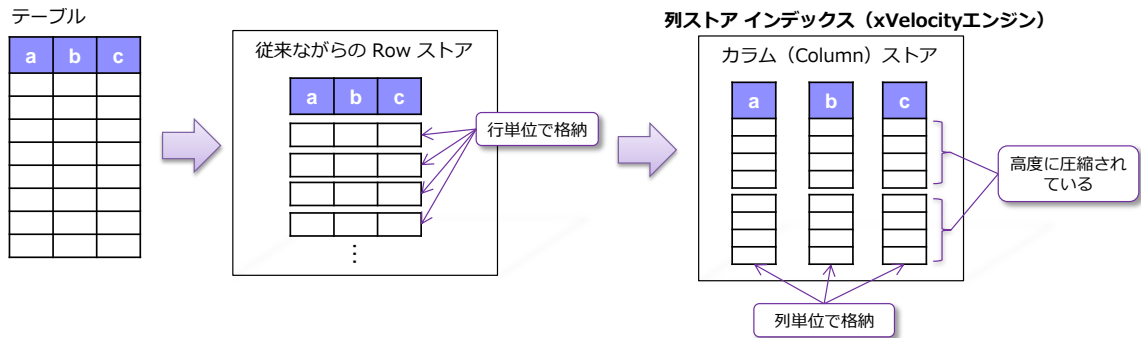
ディスク ベースの通常テーブルでは **1.98 秒**かかった集計処理が、**インメモリ OLTP** にすることで **1.6 秒**（18.8%の性能向上）、さらに**列ストア インデックス**（クラスター化列ストア インデックス）を作成することで、わずか **198 ミリ秒**で処理できるようになり、**10 倍もの性能向上**を確認することができました。これに対して、SQL Server 2014 では、インメモリ OLTP を利用することで、**2.12 秒**かかっていた処理が **42.76 秒**もかかってしまい、**20.2 倍**も遅い結果になってしまっています（SQL Server 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～

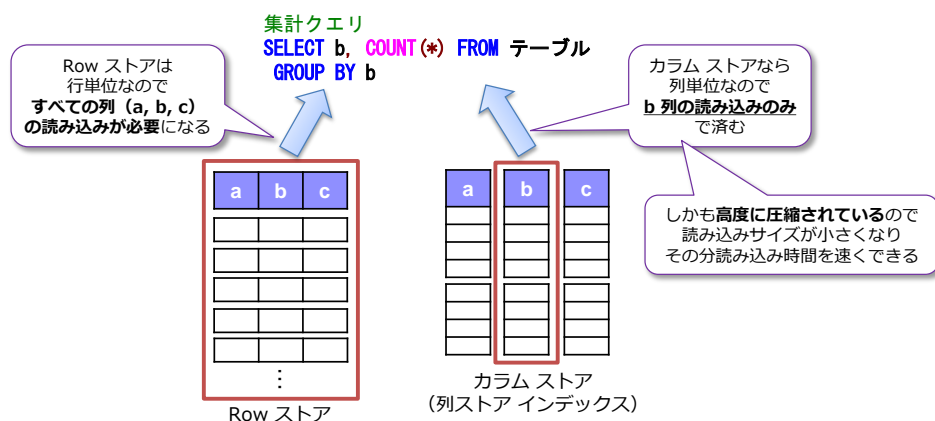
**列ストア インデックス**（Column-store Index）は、大量のデータを分析／集計するときに強さを発揮する**カラム指向データベースの SQL Server 実装**として、SQL Server 2012 のときに提供されたものです。列ストア インデックスは、「**xVelocity 列ストア インデックス**」と呼ばれることもあり、PowerPivot および Analysis Services の Tabular Mode（テーブル モード）で採用されている**インメモリの BI エンジン**である「**xVelocity エンジン**」を RDB（リレーショナル データベース）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 (Column 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** (Column ストア) へと、**約 1/6** のサイズにまで圧縮することができています。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

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 **1TB** 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、メモリに載りきら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）、メモリに載りきら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）。

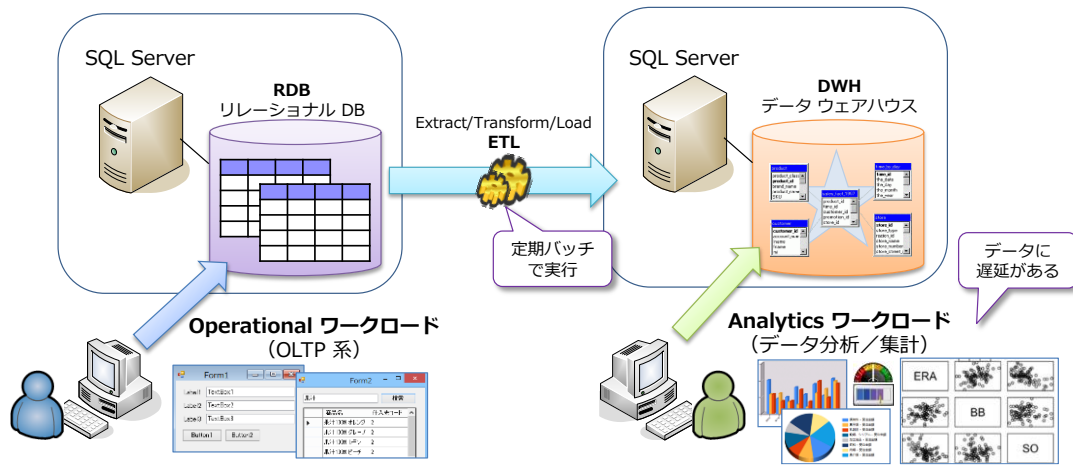
このように性能向上に関するオプションが増えたことは、本当にワクワクします（今回の SQL Server 2016 の進化は、今後のデータベースのあり方を左右するのではないかと思えるほど、非常に大きな可能性を感じています）。Operational Analytics に関しては、別途自習書を作成していま

すので、そちらもぜひご覧いただければと思います。

## ➡ 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。

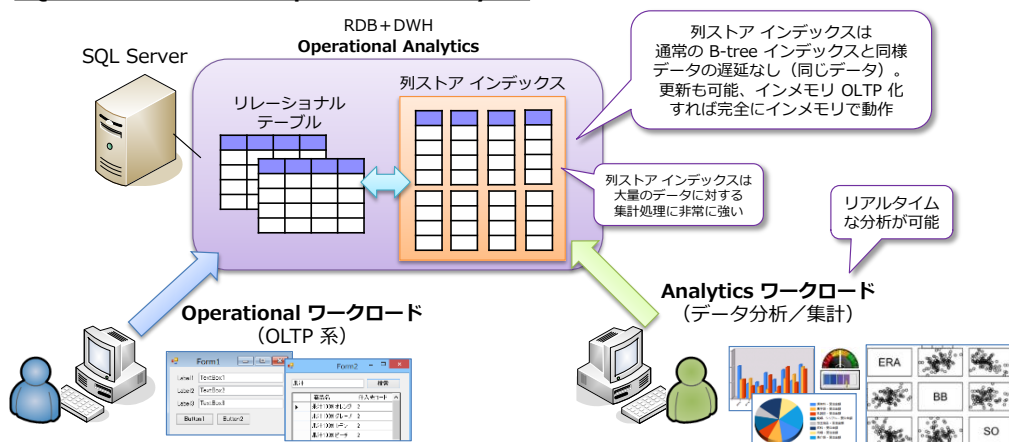
### 従来ながらの DWH では、データ分析に遅延がある



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がっています。

### SQL Server 2016 での Operational Analytics

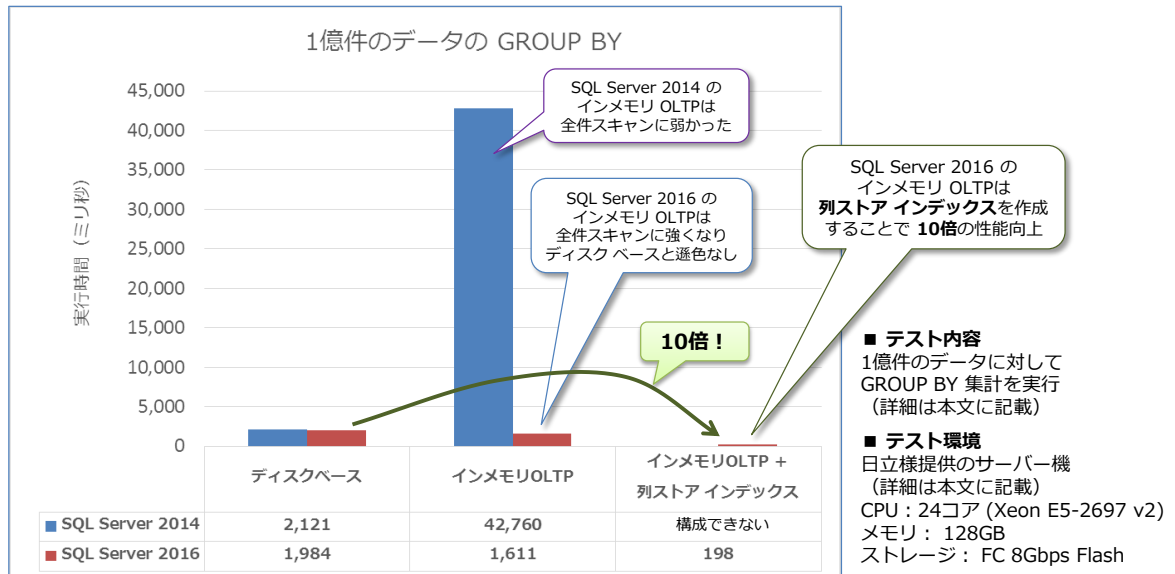


このように OLTP に DWH の役割も任せてしまえば、データ分析／集計をリアルタイムでできるようになり、このような実装に最適なのが SQL Server 2016 の Operational Analytics です。SQL Server 2016 では、インメモリ OLTP と列ストア インデックスが飛躍的に進化したことで、こうした動きがさらに加速していくように思っています。



## 2.2 Operational Analytics の検証の詳細

前項で紹介した 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。

\* 検証は、SQL Server 2016 CTP 3.2 を利用して行ったので、RTM 版ではさらに性能が向上する可能性があります。

### 検証の詳細

この検証で使したテーブルの構成と 1 億件のデータは、次のように作成しています。

#### 検証で使したテーブル構成 / 1億件のデータ

##### 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,T) 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

##### 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



**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、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) を格納しています。実際の 1 億件のデータは、次のようになっています。

SELECT \* FROM data100M

	col1	col2	col3	col4	col5
1	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	2011-01-53:00.000
5	5	6066207	697032	9903	2012:58:00.000
6	6	16121127	770155	233	2022:53:00.000
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	825	col3 int 0~999,999 の乱数	11-07-24 00:31:00.000	col5 datetime 2009/1/1 以降の日付の乱数
11	11	496	12-07-31 13:04:00.000		
12	12	124477	CONVERT(int, RAND() * 1000000)	00	CONVERT(int, RAND() * 2628000) + 1
13	13	12975051	00		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	65	2009-10-14 19:40:00.000	
17	17		CONVERT(int, RAND() * 20000000)	0	10:54:00.000
18	18			7	05:17:00.000

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ました。

-- 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

SELECT col4, COUNT(\*) AS cnt FROM InMem\_HASH GROUP BY col4 ORDER BY col4

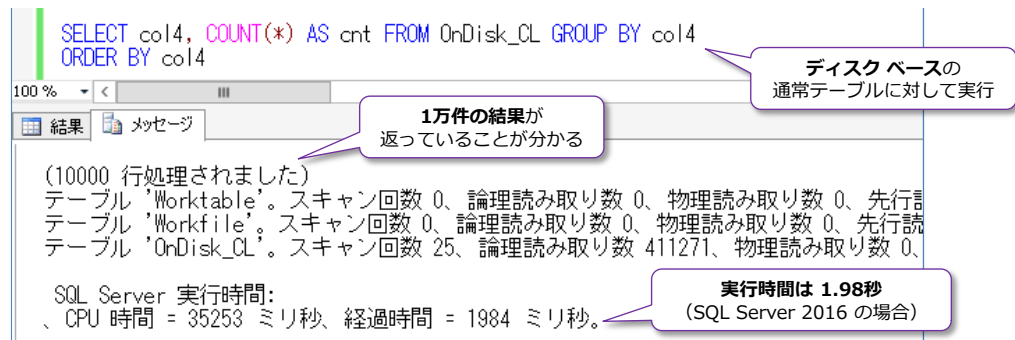
	col4	cnt
1	0	10210
2	1	9958
3	2	10122
4	3	10112
5	4	10114
6	5	10007

col4 で GROUP BY をすることで 1万件の結果が返る

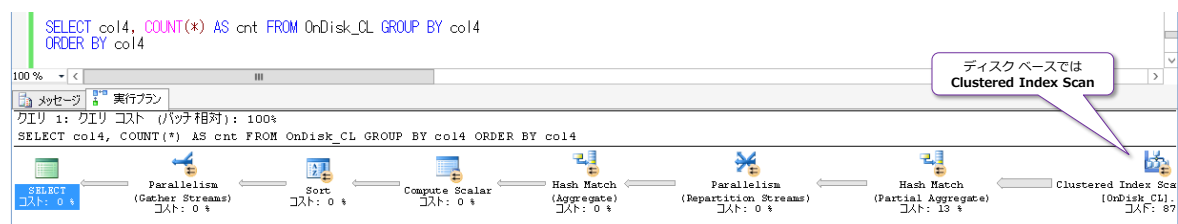
各値には約 1万件のデータがあり  
1万件 \* 1万件 = 1 億件のデータ

**col4** 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。ディス

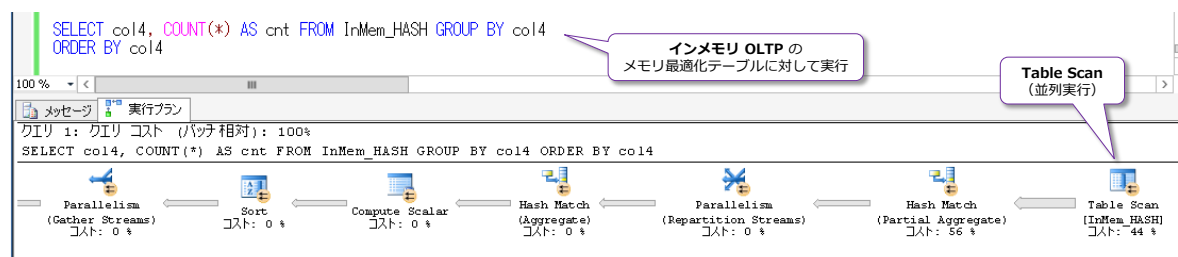
クベースの通常テーブルで、このクエリを実行したときの結果は、次のとおりです（**SET STATISTICS TIME/IO ON** で計測した実行時間と I/O 数）。



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。

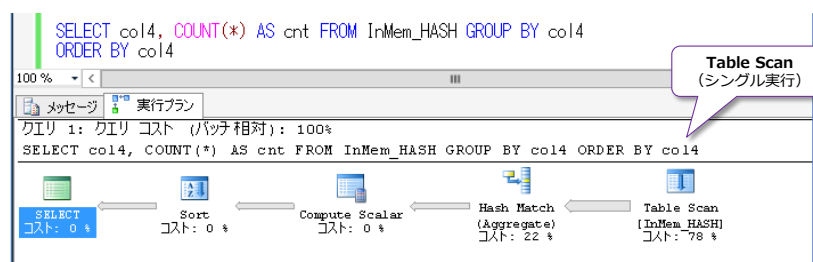


これに対して、**インメモリ OLTP のメモリ最適化テーブル**で同じクエリを実行したときの実行プランは、次のようになります。

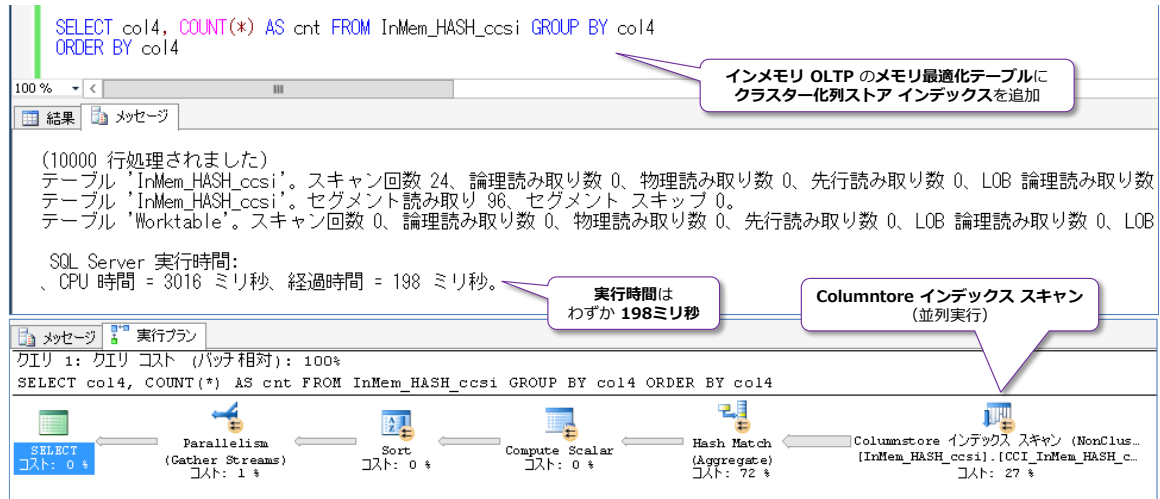


**Table San** で実行されて、**Parallelism** (並列実行) になっているのがポイントです (実行時間は、前述のグラフのとおり、**1.6 秒**です)。SQL Server 2016 からは並列プランに対応するようになったので、**Parallelism** で処理されています。

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 **198 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。

## ➡ インメモリ OLTP は、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) が目玉の新機能であると説明しましたが、インメモリ OLTP は、その名のとおり 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
<b>bwin 社</b> 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。	<b>約 16.7倍の性能向上</b> 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)
<b>SBIリクイディティ・マーケット株式会社</b> 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	<b>約 2.5倍の性能向上</b> 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
<b>Edgenet 社</b> 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	<b>約 8~11倍の性能向上</b> 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
<b>TPP 社</b> 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)	<b>約 7倍の性能向上</b> 34,700 Transaction/sec が 数十万 Transaction/sec へ向上
<b>弊社のお客様 A社</b> 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	<b>約 2.8倍の性能向上</b>

\* 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



## 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**

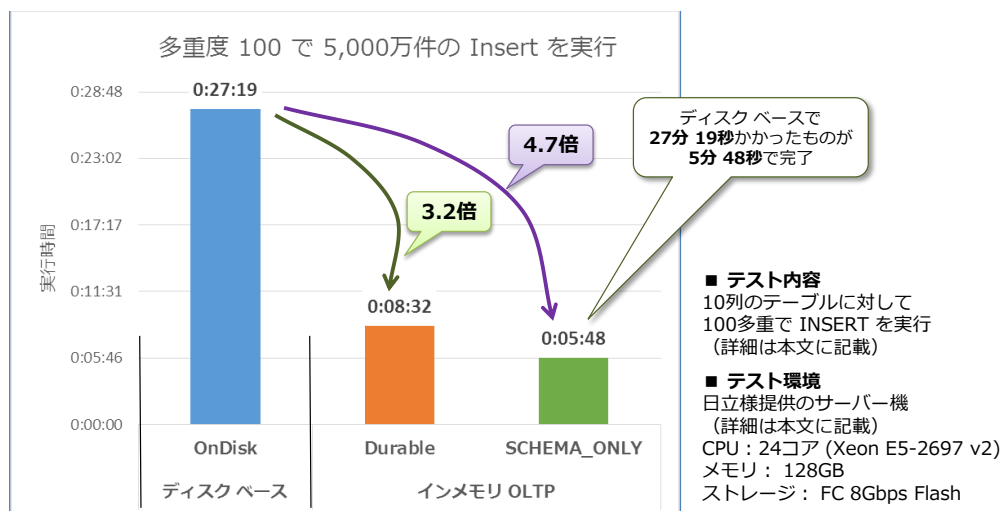


\* PASS Summit 2015 での「SQLCAT:SQL Server 2016 Early Adopter Experiences」セッションのスライドを引用。吹き出しは筆者が追加

**1 秒あたりに 66 万**ということは、**1 分** (60 秒) に換算すると  $66 \text{ 万} \times 60 \text{ 秒} = \mathbf{3,960 \text{ 万}}$ ものバッチ要求を処理できるような数値です。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 CTP 3.2 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。

\* 検証は、SQL Server 2016 CTP 3.2 を利用して行ったので、RTM 版ではさらに性能が向上する可能性があります。

ディスク ベースの通常テーブルでは **27 分 19 秒** 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 **8 分 32 秒** (3.2 倍の性能向上)、**SCHEMA\_ONLY** (永続化なし) にした場合はわずか **5 分 48 秒** で完了し、**4.7 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	30,506
インメモリ OLTP	Durable	97,657
	SCHEMA_ONLY	143,673

1秒あたり 14.4万件も  
INSERT できる

**SCHEMA\_ONLY** では、**1 秒あたり 14.4 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、そのテーブルへデータを **1 件ずつ INSERT** を行って、**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 )
```

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

```
-- Native Compile SP
CREATE PROC sp1_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
```

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	AAAA	EEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
2	756	83	AAAA	EEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
3	8160	267	AA	EEEE	CCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
4	4079	599	AAAAAAAAAAAAAAAAAAAA	EEEEEEEEEEEEEEEEEEEEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
5	657	723	AAAAAAAAAAAA	EEEEEEEEEEEEEEEEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
6	3779	690	AAAAAAAAAAAAAAAAAAAA	EEEEEEEEEEEEEEEEEEEEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
7	4012	416	AAAAAAAAAAAA	EEEEEEEEEEEEEEEEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
8	4907	38	AAA	EEEEEE	CCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
9	6998	190	AAAAAAAAAA	EEEEEEEEEEEEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
10	973	188	AAAAAA	EEEEEEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
11	6528	142	AAAAAAAAAAAA	EEEEEEEEEEEEEEEEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
12	3972	600	AAAA	EEEEEEEE	CCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
13	8508	96	AAAAAAAAAAAA	EEEEEEEEEEEEEEEEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
14	4354	732	AAAAAAAAAAAA	EEEEEEEEEEEEEEEEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
15	2427	102	AAAAAAAAAAAA	EEEEEEEEEEEEEEEEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
16	8437	654	AAAA	EEEEEEEE	CCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
17	9372	8	AAAAAAAAAAAA	EEEEEEEEEEEEEEEEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
18	3117	114	AAAAAAAAAAAA	EEEEEEEEEEEEEEEEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
19	2662	212	AAAA	EEEEEE	CCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
20	778	241	AAAA	EEEEEE	CCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
21	9372	8	AAAAAAAAAAAA	EEEEEEEEEEEEEEEEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
22	3117	114	AAAAAAAAAAAA	EEEEEEEEEEEEEEEEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
23	2662	212	AAAA	EEEEEE	CCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
24	778	241	AAAA	EEEEEE	CCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
25	9372	8	AAAAAAAAAAAA	EEEEEEEEEEEEEEEEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
26	3117	114	AAAAAAAAAAAA	EEEEEEEEEEEEEEEEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
27	2662	212	AAAA	EEEEEE	CCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
28	778	241	AAAA	EEEEEE	CCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
29	9372	8	AAAAAAAAAAAA	EEEEEEEEEEEEEEEEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0
30	3117	114	AAAAAAAAAAAA	EEEEEEEEEEEEEEEEEEEEEEEE	CCCCCCCCCCCCCCCCCCCCCCCCCCCC	2014-07-09 22:55:04.393	2015-07-09 22:55:04.393	1	0

col1 int  
連番

col2 int  
0~10000  
の乱数

col3 int  
0~1000  
の乱数

col4 nvarchar(20)  
0~19バイト (乱数) の文字

col5 nvarchar(50)  
0~38バイト (乱数) の文字

col6 nvarchar(100)  
0~95バイト (乱数) の文字

col7 datetime  
現在の日付

col8 datetime  
現在の日付+1年

col9 int  
1のみ

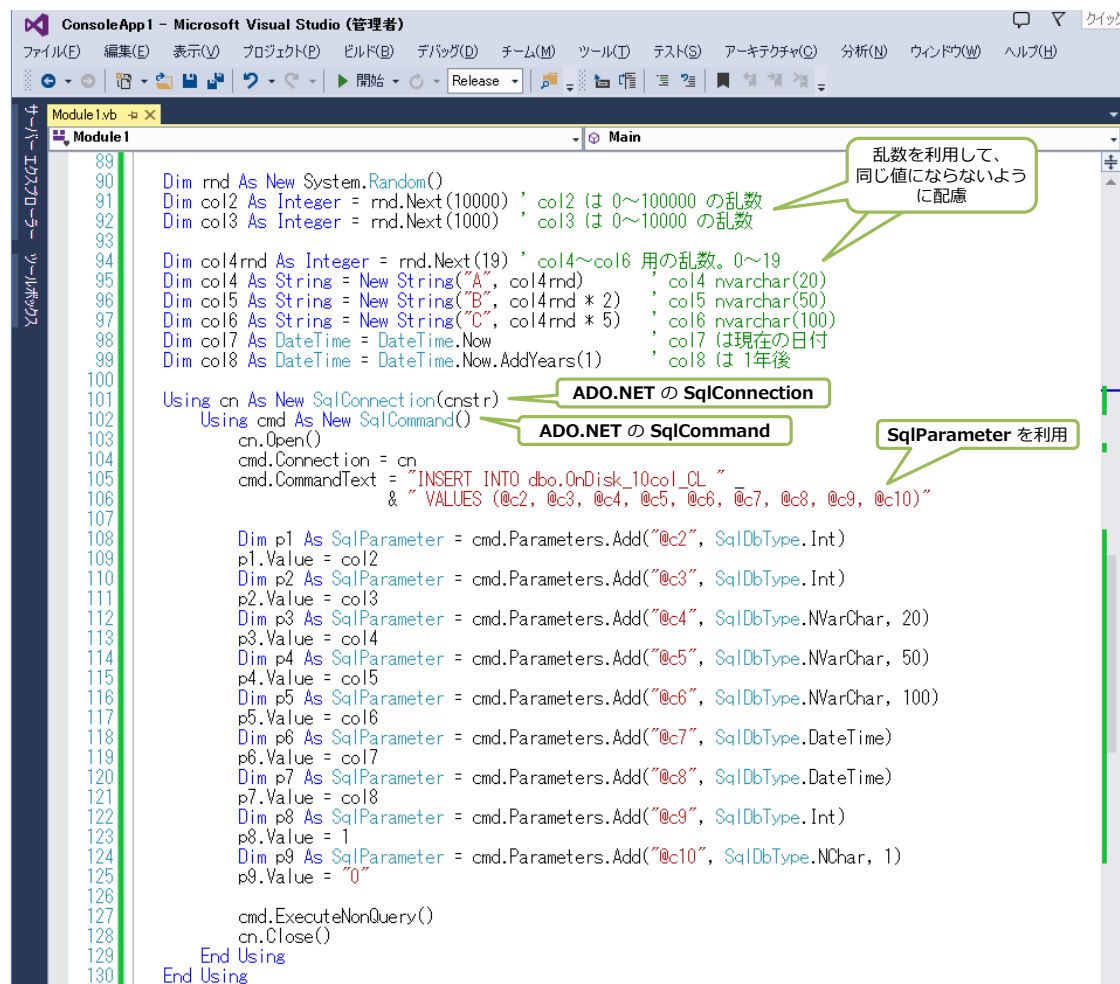
col10 nchar(1)  
"0"のみ



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）。



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.OnDisk_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

```

CommandType を変更

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

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（100 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。

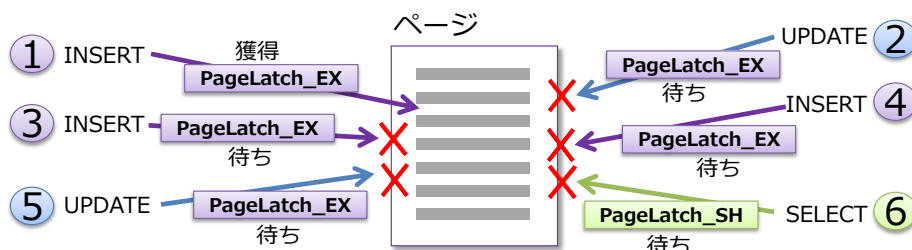
### ディスク ベースの通常テーブルで性能が向上しない理由 ～ラッチ待ち～

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

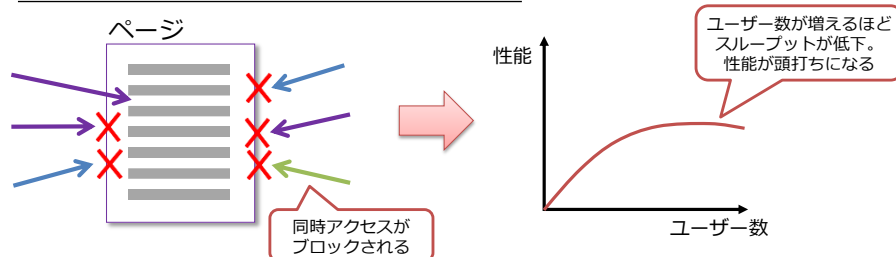




更新系のステートメント (INSERT/UPDATE/DELETE) では、**PAGELATCH\_EX** (排他ページラッチ。EX は Exclusive : 排他 の略) をかけにいき、SELECT ステートメントによる参照時には **PAGELATCH\_SH** (共有ページラッチ。SH は Shared : 共有 の略) をかけにいきます。このとき、先にアクセスしている処理がある場合は (ラッチが既にかかっている場合には)、それが解放されるまで、"待ち" が発生します。このような、同時アクセスによって待ちが発生する状態は、**ラッチ競合** (Latch Contention) 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

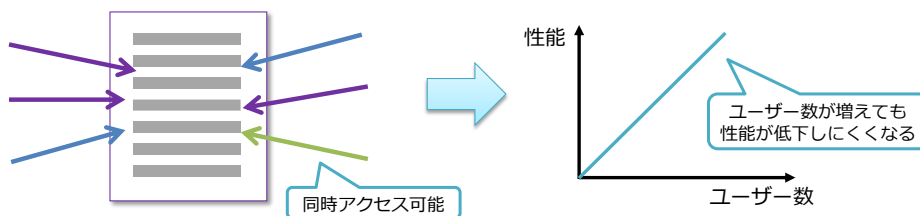
#### ラッチ待ちが発生するとスループットが低下



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。

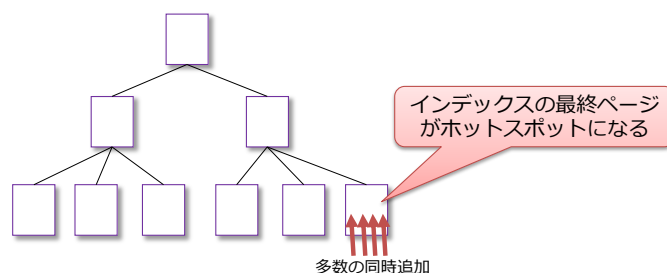
#### ラッチ フリーならユーザー数が増えても性能低下を抑えられる



#### 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。

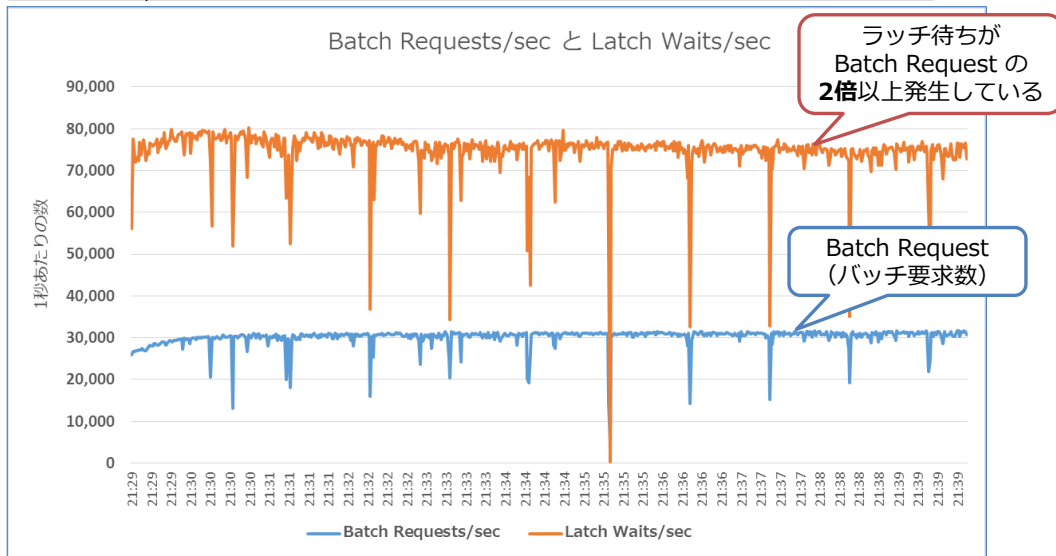
#### インデックスの最終ページでページ ラッチ待ちが多発する



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。

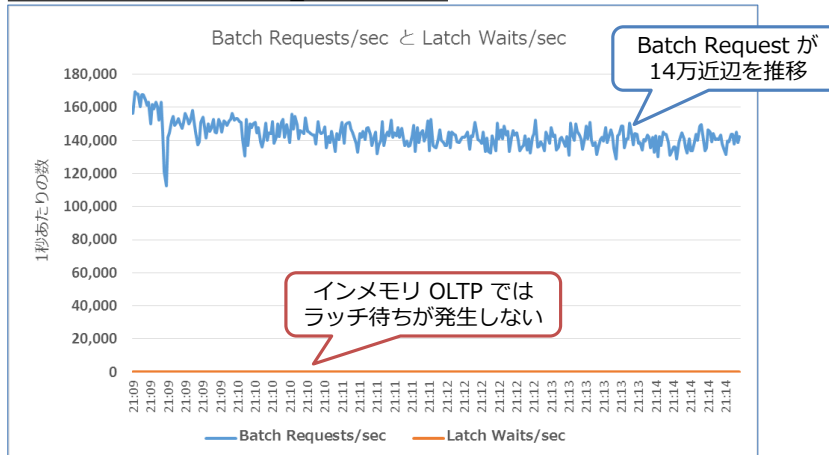
100多重で 5,000万件を INSERT しているときのラッチ待ちの様子 (ディスク ベース)



**Batch Request** (バッチ要求数) が **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 **7.5 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になります。

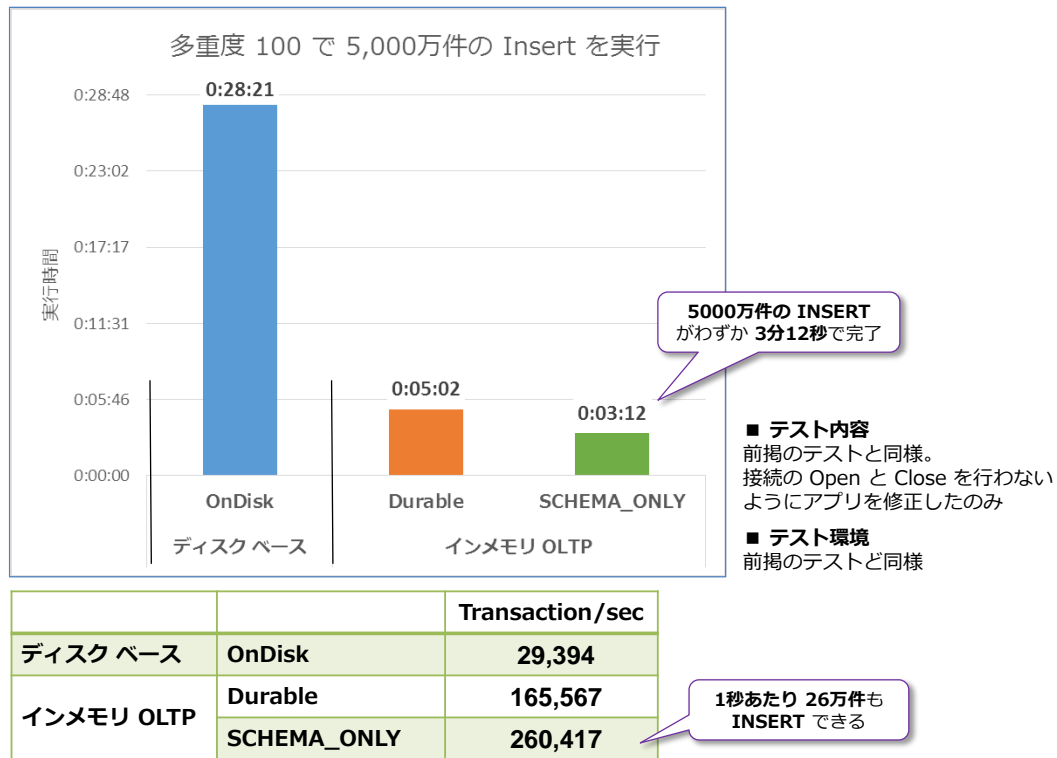
一方、インメモリ OLTP ではラッチ待ちは発生しないので、次のような性能が出ます。

インメモリ OLTP (SCHEMA ONLY) の場合



このように、インメモリ OLTP は、多数のユーザーからの INSERT および常に 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）であれば、次のような性能を出すことができます。



ディスク ベースでは、接続をキープしてクエリ負荷が上がる分だけ性能が逆に落ちるのに対して、インメモリ OLTP の SCHEMA\_ONLY では、**1 秒あたりに 26 万件ものトランザクション**を処理できることを確認できました。

インメモリ OLTP の基本的な利用方法については、別途作成している Operational Analytics の自習書に記載していますので、そちらもぜひご覧いただければと思います。

## 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)),  
ミレーション:OPEN-V×2、SASディスク×1

## STEP 3. セキュリティ強化

この STEP では、「動的データ マスク」や「行レベル セキュリティ」、「**Always Encrypted**」、「**TDE の強化**」、「**テンポラル テーブルによる Audit**」など、SQL Server 2016 CTP 3.2 で提供されたセキュリティ強化を実現できる機能を説明します。

この STEP では、次のことを学習します。

- ✓ 動的データ マスクによる情報漏洩対策
- ✓ 行レベル セキュリティによるセキュリティ強化
- ✓ Always Encrypted による列データの暗号化
- ✓ 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（監査証跡）

## 3.1 セキュリティの強化

---

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ます。その主なものは、次のとおりです。

- **動的データ マスク (Dynamic Data Masking)**

顧客情報（クレジット カード番号やマイナンバーなど）や機密情報をマスク（別の値に置換）して、情報漏洩を防止できる機能

- **行レベル セキュリティ (Row Level Security)**

行レベルのアクセス制御を実現できる機能で、ユーザーごとに、参照できる行データを制限することができる

- **Always Encrypted による列データの暗号化**

ネットワーク上を流れるデータも、データベース内に格納されるデータも、すべて暗号化して格納できる機能（列データを暗号化して、アプリケーションも透過的に利用可能）

- **TDE (透過的なデータ暗号化) の性能向上、インメモリ OLTP 対応**

Intel の AES-NI (AES 暗号化の処理を高速化するための命令セット) に対応して、TDE の性能を向上。インメモリ OLTP を利用している場合にも TDE が利用可能に。

- **テンポラル テーブルを利用した Audit (コンプライアンスにおける監査証跡)**

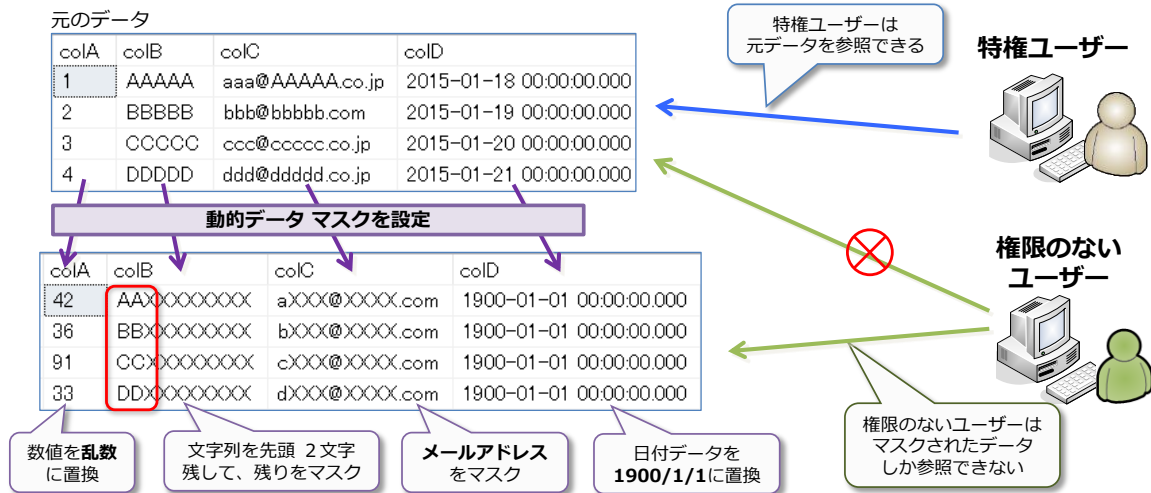
過去の更新履歴を自動保存できる「テンポラル テーブル」機能を利用して、Audit (監査証跡) が可能に

この章では、これらの機能について、1 つ 1 つ具体的な利用方法を説明します。

## 3.2 動的データ マスクによる情報漏洩対策

**動的データ マスク (Dynamic Data Masking)** は、クレジットカード番号やマイナンバー、メールアドレスなどの顧客情報／機密情報をマスク (異なる値に置換) して、セキュリティ強化 (情報漏洩の防止) を実現できる機能です。これは、次のように利用することができます。

### 動的データ マスクの利用例



このように、動的データ マスクでは、権限のないユーザーに対して、重要な情報を参照できないようにする (マスクされたデータしか参照できないようにする) ことができるので、顧客情報や機密情報 (クレジットカード番号やマイナンバーなど) のデータ流出を抑えることができるようになります。

動的データ マスクは、次のように **CREATE TABLE** ステートメントでのテーブル作成時または **ALTER TABLE** ステートメントでのテーブル変更時に、列に対して設定することができます。

```
-- 動的データ マスクの設定例
CREATE TABLE maskTest1
( colA int
, colB varchar(200) MASKED WITH (FUNCTION = 'default()')
)
```

列名、データ型に続けて、**MASKED WITH** を付けて、**FUNCTION** でどのようにマスクするかを指定します。上の例では「**default**」という関数を利用していますが、この場合は、文字列データを「**xxxx**」のようにマスク (置換) することができます。**FUNCTION** には、**default** の他に、数値を乱数化する「**random**」や、電子メール アドレスをマスクできる「**email**」、任意の文字列にマスクすることができる「**partial**」などがあります。

### ➡ Let's Try

それでは、これを試してみましょう。

1. まずは、動的データ マスクを試すためのデータベースとユーザーを作成します。



```
-- データベース「maskTestDB」の作成
CREATE DATABASE maskTestDB
go
-- データベース ユーザー「UserX」の作成
USE maskTestDB
CREATE USER UserX WITHOUT LOGIN
```

データベース名は「**maskTestDB**」、データベース ユーザーは、「**UserX**」という名前で作成し、**WITHOUT LOGIN** を付けることで、**簡易的なユーザー**にしています。このように作成したユーザーは、ログインするためのユーザーとしては利用することができないので、現実的な利用方法ではないのですが、セキュリティ機能を試すときにのみ便利なユーザーになります（このようなユーザーは、ログインをすることはできませんが、後述の **EXECUTE AS** ステートメントを利用して、接続をシミュレートして利用することができます。セキュリティ機能では、複数のユーザーでの動作を試したい場面が多々あるので、こういった場合に役立つユーザーの作成方法です）。

- 次に、**CREATE TABLE** ステートメントを利用してテーブルを作成します（テーブル名は **maskTest1** として、**colA**、**colB**、**colC**、**colD** の 4 つの列を作成）。このときに、**colB**、**colC**、**colD** の 3 つの列に対しては、次のように動的データ マスクを設定します。

```
-- テーブルの作成。colB、colC、colD 列にマスクを設定
USE maskTestDB
CREATE TABLE maskTest1
( colA int
, colB varchar(200) MASKED WITH (FUNCTION = 'default()')
, colC int          MASKED WITH (FUNCTION = 'default()')
, colD datetime     MASKED WITH (FUNCTION = 'default()')
)
```

**colA** はデータ型「**int**」の通常の列、**colB** は「**varchar(200)**」に続けて **MASKED WITH** を記述し、**FUNCTION** には **default** を指定してマスクを設定、**colC** は「**int**」で同様にマスクを設定、**colD** は「**datetime**」で同様にマスクを設定します。**default** を指定した場合には、文字データは「**xxxx**」、数値データは「**0**」、日付データは「**1900-01-01 00:00:00.000**」にマスクできるようになります。

- 次に、データを 2 件追加します（データは適当なものでかまいません）。

```
-- データを 2件追加
INSERT INTO maskTest1
VALUES (1, 'AAAAA', 111, '2015/10/30')
, (2, 'BBBBB', 222, '2015/11/30')
```

- 続いて、追加したデータを確認しておきます。

```
-- データの確認
SELECT * FROM maskTest1
```

```
-- データの確認
SELECT * FROM maskTest1
```

	colA	colB	colC	colD
1	1	AAAAA	111	2015-10-30 00:00:00.000
2	2	BBBBB	222	2015-11-30 00:00:00.000

動的データ マスクでは、特権ユーザー（管理者アカウント）に関しては、通常どおりにデータを参照することができます。

- 次に、一般ユーザーからのデータ アクセスを試すために、最初に作成したユーザー「UserX」に対して **SELECT** 権限を付与します。

```
-- データベース ユーザーに、テーブルに対する SELECT 権限を付与
GRANT SELECT ON maskTest1 TO UserX
```

- 次に、**EXECUTE AS** ステートメントを利用して、「UserX」でログインしたときの動作をシミュレートします（**EXECUTE AS** ステートメントは、**USER=** で指定したユーザーでログインしたときをシミュレートすることができ、**REVERT** ステートメントを実行するまでの間、そのユーザーでのログインをシミュレートできます）。

```
-- EXECUTE AS で UserX をシミュレート
EXECUTE AS USER = 'UserX'
SELECT * FROM maskTest1
REVERT
```

```
-- EXECUTE AS で UserX をシミュレート
EXECUTE AS USER = 'UserX'
SELECT * FROM maskTest1
REVERT
```

	colA	colB	colC	colD
1	1	xxxx	0	1900-01-01 00:00:00.000
2	2	xxxx	0	1900-01-01 00:00:00.000

マスクされたデータが表示される

このように、一般ユーザーがテーブルにアクセスした場合は、動的データ マスクが有効になって、データがマスクされていることを確認できます。今回は、**colB**、**colC**、**colD** に対して、**FUNCTION**(マスクの関数)で **default** を指定しているので、**colB**(文字データ)は「xxxx」、**colC**(数値データ)は「0」、**colD**(日付データ)は「1900-01-01 00:00:00.000」にマスクされていることを確認できます。

このように、動的データ マスクを利用すれば、一般ユーザーから簡単に情報を隠せるようになるので大変便利です。

## ➡ FUNCTION に email を指定して電子メール アドレスをマスク

次に、マスクの関数として「**email**」を利用して、電子メール アドレスをマスクしてみましょう。

1. これを試すために、既存の「**maskTest1**」テーブルに対して、**ALTER TABLE** ステートメントで列を追加してみましょう（**colE** という名前の列を追加します）。

```
ALTER TABLE maskTest1
ADD colE varchar(200) MASKED WITH (FUNCTION = 'email()')
```

このように、動的データ マスクは、**ALTER TABLE** ステートメントで列を追加するときに指定することもできます。マスクの関数に、「**FUNCTION = 'email()'**」と指定することで、電子メール アドレスをマスクできるようになります。

2. 次に、データを 1 件追加（**colE** 列には電子メール アドレス形式の文字データを指定）してみましょう。

```
-- データを 1件追加
INSERT INTO maskTest1
VALUES (3, 'CCCCC', 333, '2015/11/30', 'ccc@test.local')

-- データの確認
SELECT * FROM maskTest1
```

-- データの確認  
SELECT \* FROM maskTest1

100 %

結果 メッセージ

	colA	colB	colC	colD	colE
1	1	AAAAA	111	2015-10-30 00:00:00.000	NULL
2	2	BBBBB	222	2015-11-30 00:00:00.000	NULL
3	3	CCCCC	333	2015-11-30 00:00:00.000	ccc@test.local

電子メール アドレス形式の文字データ

3. 次に、一般ユーザー「**UserX**」からアクセスして、データがマスクされることを確認します。

```
EXECUTE AS USER = 'UserX'
SELECT * FROM maskTest1
REVERT
```

EXECUTE AS USER = 'UserX'  
SELECT \* FROM maskTest1  
REVERT

100 %

結果 メッセージ

	colA	colB	colC	colD	colE
1	1	xxxx	0	1900-01-01 00:00:00.000	NULL
2	2	xxxx	0	1900-01-01 00:00:00.000	NULL
3	3	xxxx	0	1900-01-01 00:00:00.000	cXXX@XXXX.com

cXXX@XXXX.comに変換されている

「ccc@test.local」というデータが、「cXXX@XXX.com」という形でマスクされていることと確認できます。このように、電子メール アドレスをマスクしたい場合には、**email** 関数を利用すると便利です。

## ➡ FUNCTION に random を指定して数値データを乱数化

次に、マスクの関数として「**random**」を利用して、数値データを**乱数**にマスクしてみましょう。

- これを試すために、既存の **colC** 列 (**int** データ型で作成した列) を利用します。既存の列に対して、動的データ マスクを設定/変更するには、次のように **ALTER TABLE** ステートメントの **ALTER COLUMN** で、**ADD MASKED WITH** を利用します。

```
ALTER TABLE maskTest1
ALTER COLUMN colC
ADD MASKED WITH (FUNCTION = 'random(1, 100)')
```

**random** 関数では、2 つの引数に、**乱数の範囲**を指定します。上の例のように「**(1, 100)**」と指定した場合は、1~100 の間の乱数にマスクすることができます。

- 設定が完了したら、一般ユーザー「**UserX**」からアクセスして、データがマスクされることを確認してみましょう。

```
EXECUTE AS USER = 'UserX'
SELECT * FROM maskTest1
REVERT
```

	colA	colB	colC	colD	colE
1	1	xxxx	19	1900-01-01 00:00:00.000	NULL
2	2	xxxx	78	1900-01-01 00:00:00.000	NULL
3	3	xxxx	1	1900-01-01 00:00:00.000	cXXX@XXX.com

## ➡ FUNCTION に partial を指定して文字データを任意の文字列にマスク

次に、マスクの関数として「**partial**」を利用して、文字データを**任意の文字列**にマスクしてみます。

- これを試すために、既存の **colB** 列 (**varchar(200)** データ型で作成した列) を利用してみましょう。**ALTER TABLE .. ALTER COLUMN** ステートメントを次のように実行します。

```
ALTER TABLE maskTest1
ALTER COLUMN colB
ADD MASKED WITH (FUNCTION = 'partial(1, "zzzzz", 1)')
```

**partial** 関数は、3つの引数で「接頭辞, padding, 接尾辞」という形で利用します。上の例のように「(1, "zzzzz", 1)」と指定した場合は、接頭辞が 1、接尾辞も 1 なので、先頭と最後の 1 文字はそのまま残して、間の文字を **zzzzz** で埋めて（パディングして）マスクすることができます。

2. 設定が完了したら、一般ユーザー「UserX」からアクセスして、データがマスクされることを確認してみましょう。

```
EXECUTE AS USER = 'UserX'
SELECT * FROM maskTest1
REVERT
```

	colA	colB	colC	colD	colE
1	1	AzzzzzA	68	1900-01-01 00:00:00.000	NULL
2	2	BzzzzzB	88	1900-01-01 00:00:00.000	NULL
3	3	CzzzzzC	40	1900-01-01 00:00:00.000	cXXX@XXXX.com

3. 次に、**partial** 関数の引数を次のように変更（接尾辞を 1 から 0 に変更）してみましょう。

```
ALTER TABLE maskTest1
ALTER COLUMN colB
ADD MASKED WITH (FUNCTION = 'partial(1, "zzzzz", 0)')
```

4. 設定後、一般ユーザー「UserX」からアクセスして、結果を確認してみます。

```
EXECUTE AS USER = 'UserX'
SELECT * FROM maskTest1
REVERT
```

	colA	colB	colC	colD	colE
1	1	Azzzzz	16	1900-01-01 00:00:00.000	NULL
2	2	Bzzzzz	20	1900-01-01 00:00:00.000	NULL
3	3	Czzzzz	45	1900-01-01 00:00:00.000	cXXX@XXXX.com

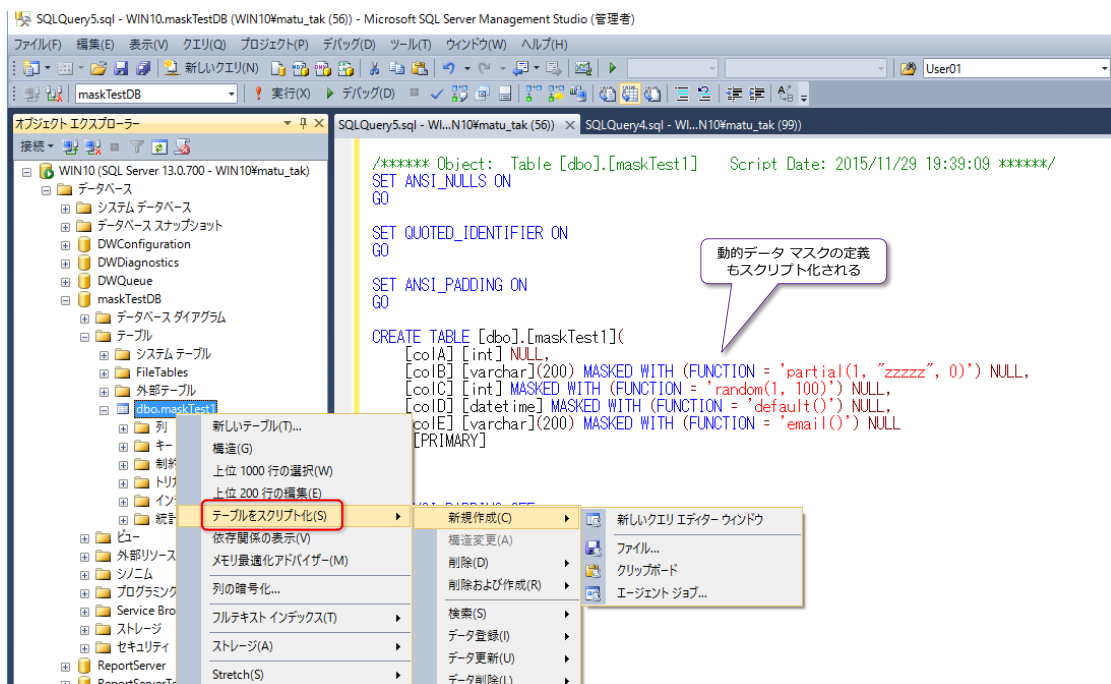
今度は、接尾辞を **0** に指定しているので、最後の文字は残っていないことを確認できますと思います。

このように **partial** 関数を利用すると、文字列データを任意の文字列にマスクすることがで

きるので大変便利です。

## ➡ 設定の確認 ～スクリプト生成、masked\_columns カタログ ビュー～

動的データ マスクの設定を確認するには、**masked\_columns** カタログ ビューを利用する方法がありますが、これよりも簡単なのが、Management Studio のスクリプト生成機能を利用する方法です。これを利用するには、次のように、該当テーブルを右クリックして、[テーブルをスクリプト化] から [新規作成] → [新しいクエリ エディター ウィンドウ] をクリックします。



これで、**CREATE TABLE** ステートメントが生成されて、動的データ マスクの定義も確認することができます。

**masked\_columns** カタログ ビューを利用する場合は、次のように「**masking\_function**」列で定義を確認することができます。

```
SELECT name, masking_function, *
FROM sys.masked_columns
WHERE object_id = OBJECT_ID('maskTest1')
```

<pre>SELECT name, masking_function, * FROM sys.masked_columns WHERE object_id = OBJECT_ID('maskTest1')</pre>							
	name	masking_function	object_id	name	column_id	system_type_id	user_type_id
1	colB	partial(1, "zzzzz", 0)	645577338	colB	2	167	167
2	colC	random(1, 100)	645577338	colC	3	56	56
3	colD	default()	645577338	colD	4	61	61
4	colE	email()	645577338	colE	5	167	167

このように、**動的データ マスク**を利用すれば、データを簡単にマスク（違う値に変換）することができるので、**クレジットカード番号やマイナンバー、メール アドレスなどの顧客情報／機密情報をマスク**する目的で利用することができます（セキュリティの強化、情報漏洩／データ流出の防止を実現することができます）。

その他、動的データ マスクに関する最新情報は、オンライン ブックの次のトピックが参考になると思います。

## Dynamic Data Masking

<https://msdn.microsoft.com/en-us/library/mt130841.aspx>

...
  
Strong Passwords
  
Row-Level Security
  
▶ SQL Server Encryption
  
**Dynamic Data Masking**
  
SQL Server Certificates and Asymmetric Keys
  
▶ SQL Server Audit (Database Engine)

# Dynamic Data Masking

SQL Server 2016

Updated: November 19, 2015

Applies To: SQL Server 2016 Preview

**Topic Status:** Some information in this topic is preview and subject to change in future releases. Preview information describes new features or changes to existing features in SQL Server 2016 Community Technology Preview 3 (CTP 3.0).

Dynamic data masking limits sensitive data exposure by masking it to non-privileged users. Dynamic data masking helps prevent unauthorized access to sensitive data by enabling customers to designate how much of the sensitive data to reveal with minimal impact on the application layer. It's a data protection feature that hides the sensitive data in the result set of a query over designated database fields, while the data in the database is not changed. Dynamic data masking is easy to use with existing applications, since masking rules are applied in the query results. Many applications can mask sensitive data without modifying existing queries.

For example, a call center support person may identify callers by several digits of their social security number or credit card number, but those data items should not be fully exposed to the support person. A masking rule can be defined that masks all but the last four digits of any social security number or credit card number in the result set of any query. For another example, by using the appropriate data mask to protect personally identifiable information (PII) data, a developer can query production environments for troubleshooting purposes without violating compliance regulations.

The purpose of dynamic data masking is to limit exposure of sensitive data, preventing users who should not have access to the data from viewing it. Dynamic data masking does not aim to prevent database users from connecting directly to the database and running exhaustive queries that expose pieces of the sensitive data. Dynamic data masking is complementary to other SQL Server security features (auditing, encryption, row level security...) and it is highly recommended to use this feature in conjunction with them in addition in order to better protect the sensitive data in the database.

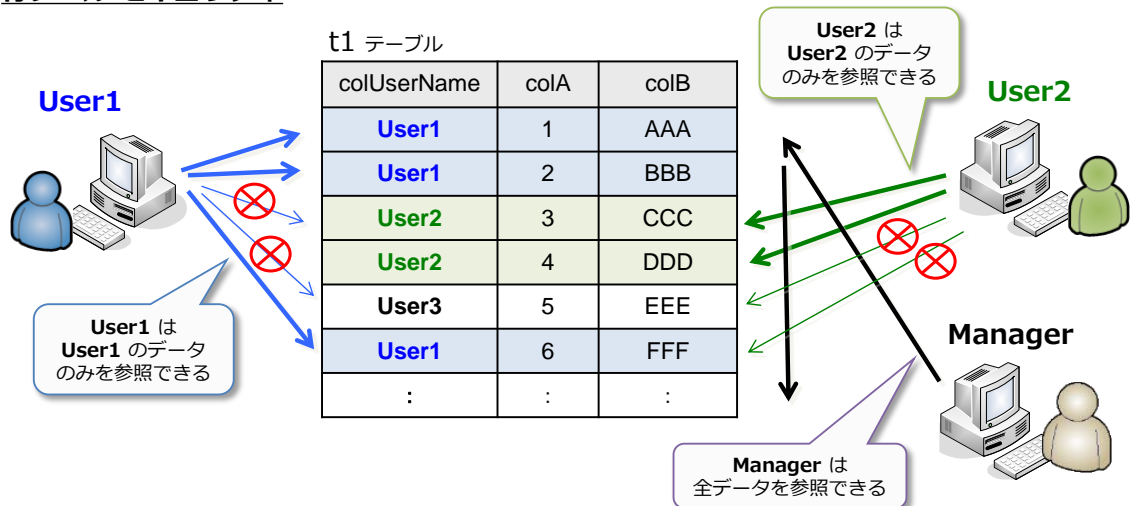
Dynamic data masking is available in SQL Server 2016 Community Technology Preview 3 (CTP 3.0). For Azure SQL Database, see [Get started with SQL Database Dynamic Data Masking \(Azure Preview portal\)](#).



### 3.3 行レベル セキュリティによるセキュリティ強化

行レベル セキュリティは、行レベルのアクセス制御を実現できる機能です。これを利用すれば、ユーザーに対して、参照できる「行」を制限できるようになるので、次のようにユーザーごとに、見せたい行を制限できるようになります。

#### 行レベル セキュリティ



このように行レベル セキュリティを利用すれば、**User1 は User1 のデータのみ**、**User2 は User2 のデータのみ**しか参照できないように制限することができます。また、すべてのデータを参照できるユーザー（図では **Manager**）を設定することもできます。

#### ➡ Let's Try

それでは、これを試してみましょう。

1. まずは、行レベル セキュリティを試すためのデータベースとユーザーを作成します。

```
-- データベース「rsTestDB」の作成
CREATE DATABASE rsTestDB
go

-- データベース ユーザー User1、User2、Manager の作成
USE rsTestDB
CREATE USER User1 WITHOUT LOGIN
CREATE USER User2 WITHOUT LOGIN
CREATE USER Manager WITHOUT LOGIN
```

データベース名は「rsTestDB」、データベース ユーザーは「User1」、「User2」、「Manager」という名前で作成します（**WITHOUT LOGIN** は、現実的なシナリオでは利用しない使い方になりますが、行レベル セキュリティを試すための簡易的なユーザーの作成方法になります）。

2. 次に、**CREATE TABLE** ステートメントを利用してテーブルを作成します。

```
-- テーブル「rsTest1」の作成
CREATE TABLE rsTest1
( colUserName sysname
, colA int
, colB varchar(200) )
```

テーブル名は「rsTest1」とし、「colUserName」列は、アクセス可能なユーザーの名前を格納するために利用します。「colA」と「colB」には、特に意味はないので、適当な列の名前で大丈夫です。

3. 次に、データを 4 件追加します。

```
-- データを 4件追加
INSERT rsTest1
VALUES ('User1', 1, 'AAA')
      , ('User1', 2, 'BBB')
      , ('User2', 3, 'CCC')
      , ('User2', 4, 'DDD')
```

1、2 件目は、colUserName 列に **User1** を格納して、**User1** のみが参照できるようにし、  
3、4 件目には **User2** を格納するようにします。

4. 次に、追加したデータを確認します。

```
-- データの確認
SELECT * FROM rsTest1
```

-- データの確認  
SELECT \* FROM rsTest1

100 % <

結果 メッセージ

	colUserName	colA	colB
1	User1	1	AAA
2	User1	2	BBB
3	User2	3	CCC
4	User2	4	DDD

5. 続いて、データベース ユーザー (**User1**、**User2**、**Manager**) に対して、テーブルに対する **SELECT** 権限を付与します。

```
-- データベース ユーザーに、テーブルに対する SELECT 権限を付与
GRANT SELECT ON rsTest1 TO User1
GRANT SELECT ON rsTest1 TO User2
GRANT SELECT ON rsTest1 TO Manager
```

## ➡ 行レベル セキュリティのためのユーザー定義関数とセキュリティ ポリシーの作成

行レベル セキュリティは、**ユーザー定義関数**と**セキュリティ ポリシー**（フィルター）を作成することによって実現できます。

6. ユーザー定義関数は、次のように作成します。

```
-- 行レベル セキュリティを実現するためのユーザー定義関数の作成例
CREATE FUNCTION rsFunc1 (@UserName AS sysname)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN
SELECT 1 AS result
WHERE USER_NAME() = @UserName
OR USER_NAME() = 'Manager'
```

ユーザー定義関数の名前は **rsFunc1**（任意の名前を設定可能）、引数に **@UserName** を追加して、**ユーザー名**を受け取れるようにします。関数の中では、**USER\_NAME()** 関数を利用して接続中のユーザー名を取得し、引数として受け取ったユーザー名と等しいかどうか、あるいは **Manager** かどうかを判断して、ユーザー名が等しいまたは Manager の場合に **1** を返すようにしています。

7. ユーザー定義関数を作成した後は、**FILTER PREDICATE** を指定した**セキュリティ ポリシー**（フィルター）を作成します。これを行うには、次のように **CREATE SECURITY POLICY** ステートメントを記述します。

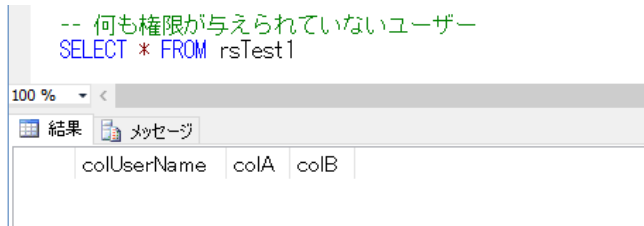
```
CREATE SECURITY POLICY rsFilter1
ADD FILTER PREDICATE dbo.rsFunc1(colUserName)
ON dbo.rsTest1
WITH (STATE = ON)
```

セキュリティ ポリシーの名前は **rsFilter1**（任意の名前を設定可能）、**ADD FILTER PREDICATE** で「**dbo.rsFunc1(colUserName) ON dbo.rsTest1**」と指定することで、該当テーブル（**rsTest1**）に対して、**rsFunc1** 関数（前の手順で作成したユーザー定義関数）でフィルターをかけて、関数へ与える引数に **colUserName** 列を与えることができるようになります。末尾の **WITH (STATE = ON)** では、このセキュリティ ポリシーを有効化することができます。

以上で、行レベル セキュリティの設定が完了です。

8. 次に、**通常のユーザー**（何も権限が与えられていないユーザー）で、テーブルを **SELECT** してみます。

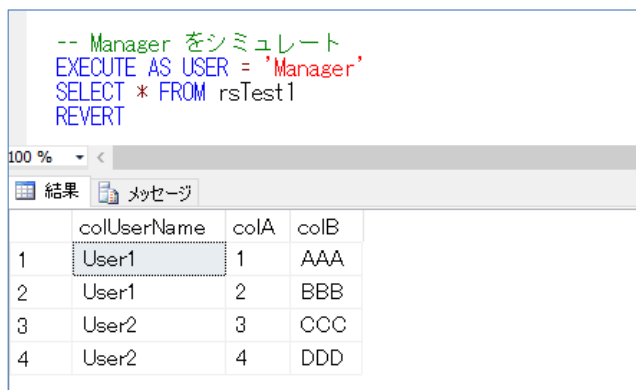
```
-- 何も権限が与えられていないユーザー
SELECT * FROM rsTest1
```



結果には、何も返ってきません。このように、テーブルに対して**フィルター**を設定している場合には、ユーザー定義関数で設定したユーザー以外は、テーブル データを参照できない形になります。

9. 次に、**Manager** ユーザーでアクセスしてみます (**EXECUTE AS USER** ステートメントで別のユーザーでの接続をシミュレートします)。

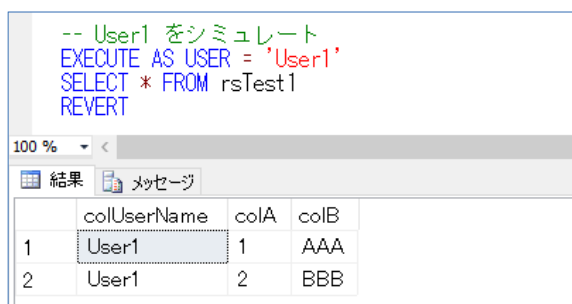
```
-- Manager をシミュレート
EXECUTE AS USER = 'Manager'
SELECT * FROM rsTest1
REVERT
```



**Manager** ユーザーは、すべてのデータを参照することができます。内部的には、フィルターで指定したユーザー定義関数内の「**USER\_NAME() = 'Manager'**」という部分が利用されていて、**Manager** ユーザーであれば、すべてのデータが返る形になります。

10. 次に、**User1** ユーザーでアクセスしてみます。

```
-- User1 をシミュレート
EXECUTE AS USER = 'User1'
SELECT * FROM rsTest1
REVERT
```

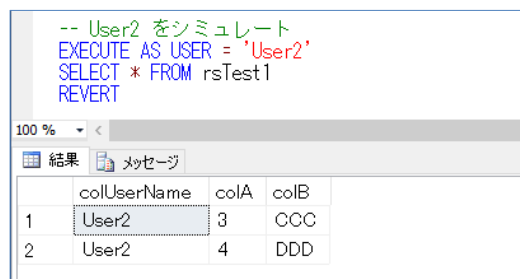


**User1** ユーザーは、**colUserName** 列が **User1** のデータしか参照できないことを確認できます。

内部的には、フィルターで指定したユーザー定義関数内の「**USER\_NAME() = @UserName**」という部分が利用されていて、**USER\_NAME()** は **User1**（接続しているユーザーの名前）を返し、**@UserName** には、**colUserName** 列に格納されているデータ(**User1** や **User2**) が与えられるので、データが **User1** の場合には、その結果を返すという形です。

11. 次に、**User2** ユーザーでアクセスしてみます。

```
-- User2 をシミュレート
EXECUTE AS USER = 'User2'
SELECT * FROM rsTest1
REVERT
```



	colUserName	colA	colB
1	User2	3	CCC
2	User2	4	DDD

**User2** ユーザーは、**colUserName** 列が **User2** のデータしか参照できないことを確認できます。

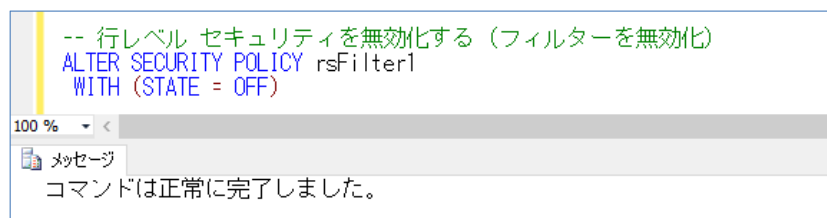
このように、行レベル セキュリティを利用すれば、行レベルのアクセス制御を実現することができ、ユーザーごとに、見せたい行を制限できるようになります。

## ➡ 行レベル セキュリティを無効化したい場合

行レベル セキュリティを無効化したい場合には、作成した**フィルター（セキュリティ ポリシー）**を無効化するだけで完了します。これも試してみましょう。

1. フィルターを無効化するには、次のように **ALTER SECURITY POLICY** ステートメントで、**STATE** を **OFF** に設定します。

```
-- 行レベル セキュリティを無効化する（フィルターを無効化）
ALTER SECURITY POLICY rsFilter1
WITH (STATE = OFF)
```



```
-- 行レベル セキュリティを無効化する（フィルターを無効化）
ALTER SECURITY POLICY rsFilter1
WITH (STATE = OFF)
```

メッセージ  
コマンドは正常に完了しました。

無効化をした後は、**通常のユーザー**（何も権限が与えられていないユーザー）でも、データを参照できるようになります。

```
-- 何も権限が与えられていないユーザー（無効化後は参照可能）
SELECT * FROM rsTest1
```

	colUserName	colA	colB
1	User1	1	AAA
2	User1	2	BBB
3	User2	3	CCC
4	User2	4	DDD

その他、行レベル セキュリティに関する最新情報は、オンライン ブックの次のトピックが参考になると思います。

### Row-Level Security

<https://msdn.microsoft.com/en-us/library/dn765131.aspx>

...
Permissions or Securables Page
Password Policy
Strong Passwords
Row-Level Security
SQL Server Encryption
Dynamic Data Masking
SQL Server Certificates and Asymmetric Keys
SQL Server Audit (Database Engine)

## Row-Level Security

SQL Server 2016 | Other Versions ▾

Updated: November 30, 2015

Applies To: Azure SQL Database, SQL Server 2016 Preview

Row-Level Security enables customers to control access to rows in a database table based on the characteristics of the user executing a query (e.g., group membership or execution context).

Row-Level Security (RLS) simplifies the design and coding of security in your application. RLS enables you to implement restrictions on data row access. For example ensuring that workers can access only those data rows that are pertinent to their department, or restricting a customer's data access to only the data relevant to their company.

The access restriction logic is located in the database tier rather than away from the data in another application tier. The database system applies the access restrictions every time that data access is attempted from any tier. This makes your security system more reliable and robust by reducing the surface area of your security system.

Implement RLS by using the [CREATE SECURITY POLICY](#) Transact-SQL statement, and predicates created as [inline table valued functions](#).

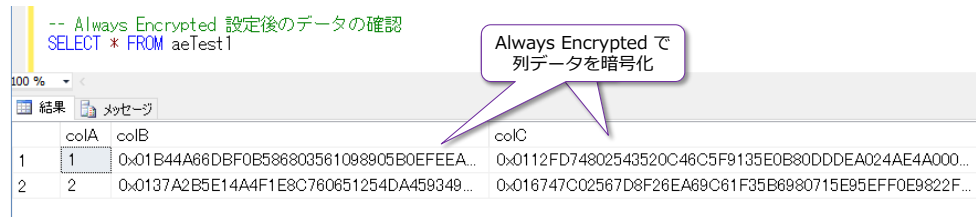
**Applies to:** SQL Server (SQL Server 2016 Community Technology Preview 3 (CTP 3.0) through [current version](#)), SQL Database V12 ([Get it](#)).

### In this Topic

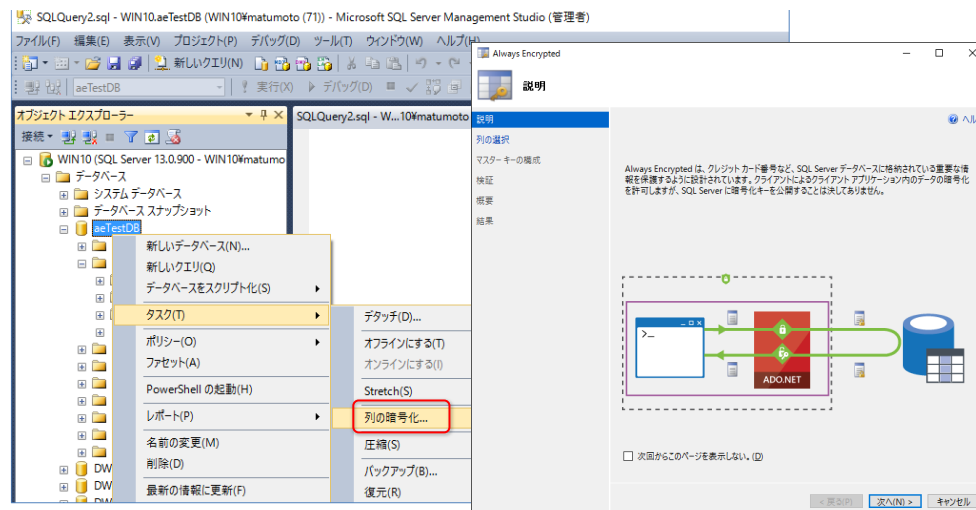
- Description
- Use Cases

### 3.4 Always Encrypted による列データの暗号化

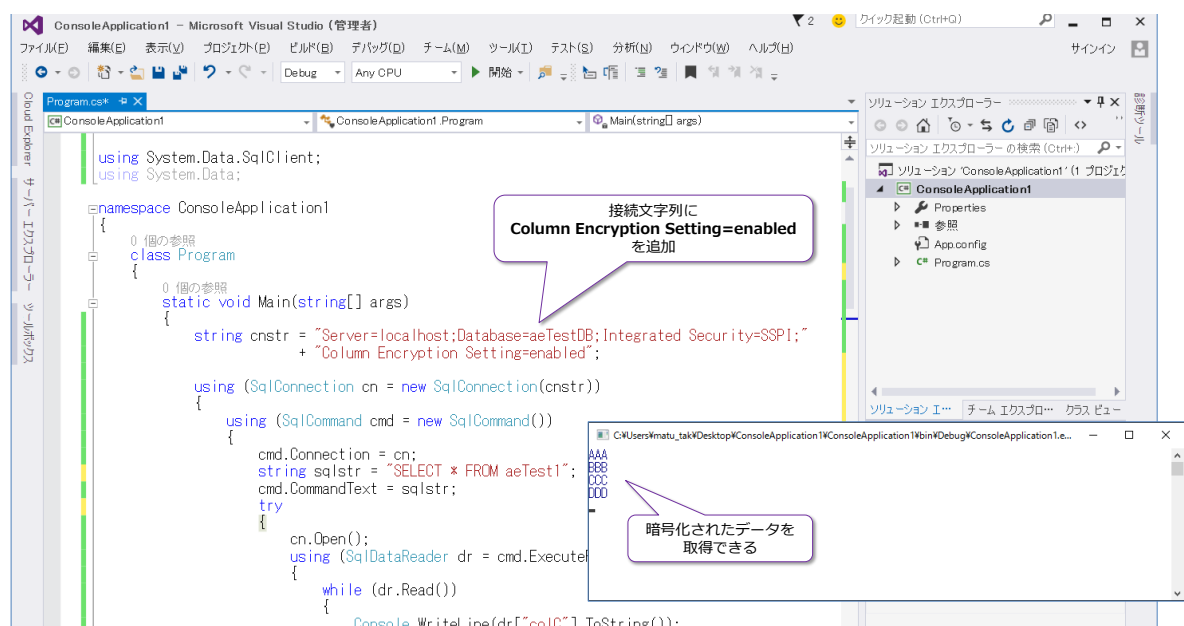
**Always Encrypted** は、ネットワーク上を流れるデータも、データベース内に格納されるデータも、すべて暗号化して格納できる機能です。内部的には、次のように**列データを暗号化**して格納することで、これを実現しています（アプリケーションからも透過的に利用できます）。



設定には、ウィザードが用意されているので、簡単に設定することができます。



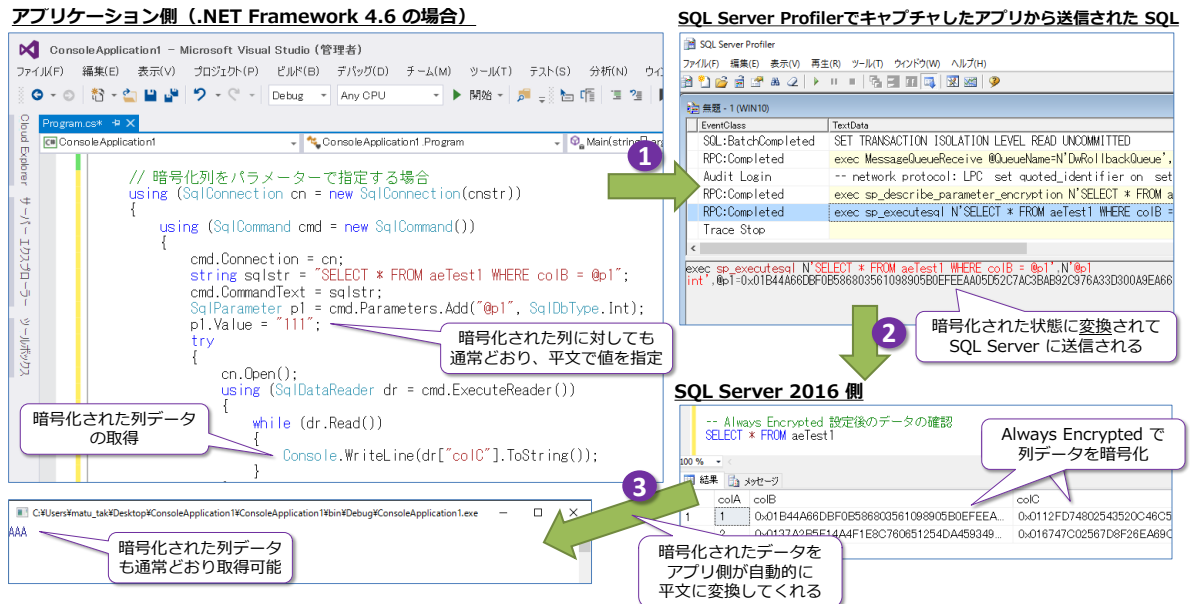
**Always Encrypted** を利用するために、アプリケーション側で変更する必要があるのは接続文字列のみで、そのほかのプログラムは修正する必要はありません（接続文字列に **Column Encryption Setting=enabled** を追加するのみ。ただし、**.NET Framework 4.6** 以上が必要）。





## ➡ Always Encrypted での基本動作（アプリ側が 暗号 ↔ 復号 を自動で行う）

Always Encrypted での基本動作は、次のようになります。



アプリケーションの接続文字列に「**Column Encryption Setting=enabled**」を追加しておくことで、アプリケーションのプログラム内に記述された平文（上の例では **colB** 列に与える **111** という値）は、アプリケーションによって、自動的に暗号化されて（**0x01B44A66DB~** 形式に自動変換されて）SQL Server に送信されます。SQL Server 側では、暗号化された状態でデータが格納されている（**0x01B44A66DB~** という形で格納されている）ので、この該当データをアプリケーションに返す、という流れです。

また、アプリケーション側では、**colC** 列のデータ（暗号化された **0x0112FD7480~** 形式のデータ）を取得していますが、このデータについても、アプリケーションが自動的に復号化を行って、**AAA** というデータに戻しています。

このように、**Always Encrypted** を利用すれば、アプリケーション コードを変更することなく（変更するのは接続文字列だけで）、SQL Server 側の列データおよびネットワーク上を流れるデータを暗号化することができるようになります = セキュリティを大幅に向上させることができます。

## ➡ Let's Try

それでは、これを試してみましょう。

1. まずは、Always Encrypted を試すためのデータベースとテーブルを作成します。

```
-- データベースの作成
CREATE DATABASE aeTestDB
go
```

```

-- テーブルの作成
USE aeTestDB
CREATE TABLE aeTest1
( colA int PRIMARY KEY
, colB int
, colC varchar(20))

-- データを 2件追加
INSERT INTO aeTest1 VALUES(1, 111, 'AAA')
INSERT INTO aeTest1 VALUES(2, 222, 'BBB')

-- データの確認
SELECT * FROM aeTest1

```

-- データの確認  
SELECT \* FROM aeTest1

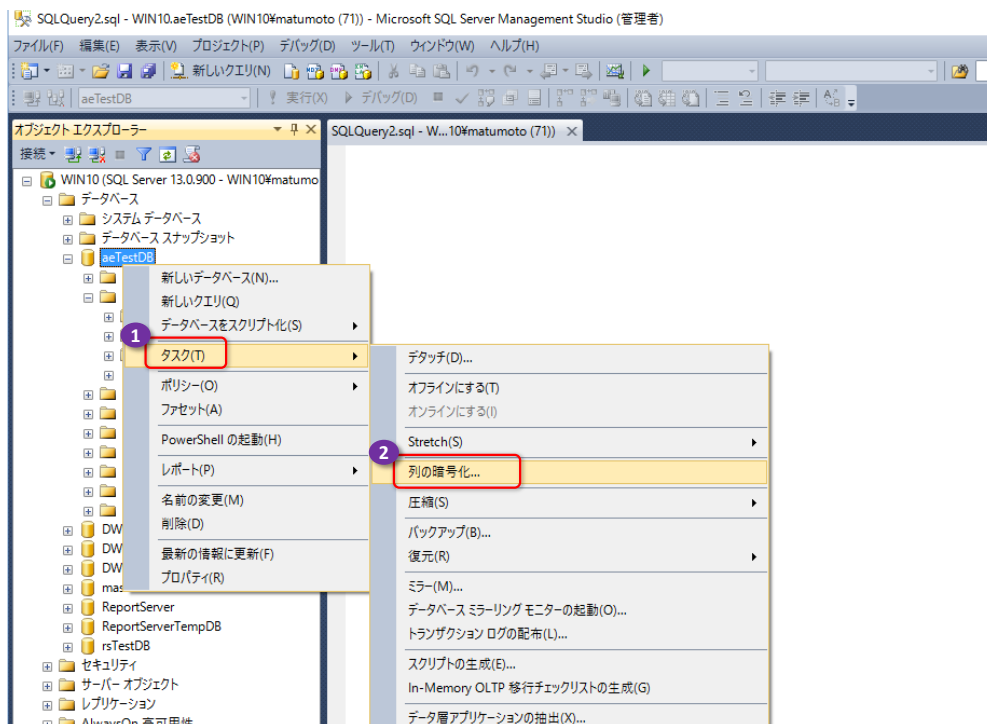
100 %

結果 メッセージ

	colA	colB	colC
1	1	111	AAA
2	2	222	BBB

データベース名は「aeTestDB」、この中に「aeTest1」という名前のテーブルを作成して、データを 2 件追加しておきます。テーブルには、「colA」と「colB」、「colC」の 3 列を作成していますが、この後の手順で「colB」と「colC」列を暗号化していきます。

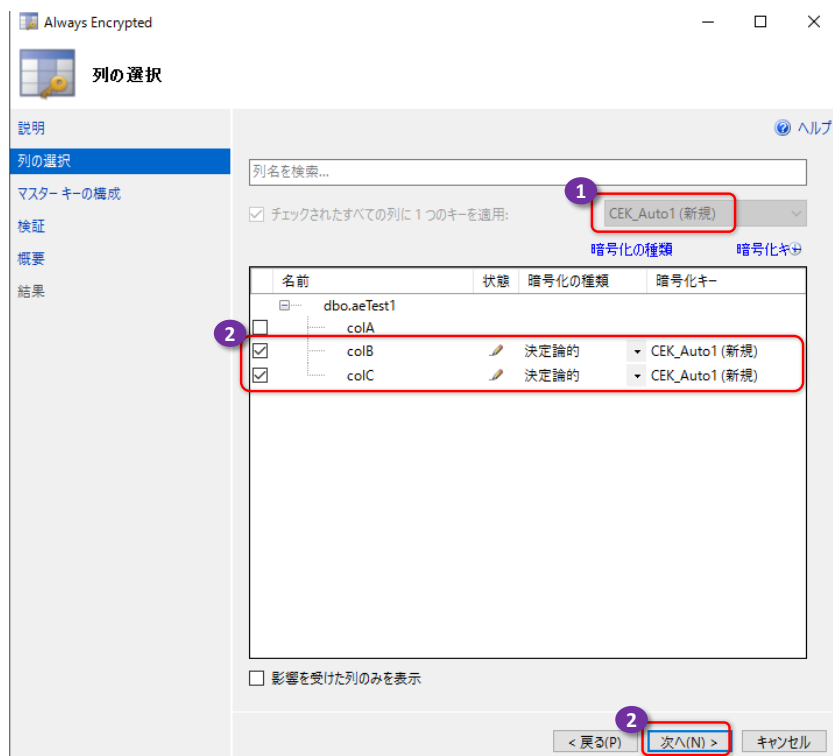
- 次に、オブジェクト エクスプローラーで「aeTestDB」データベースを右クリックして、[タスク] メニューの [列の暗号化] をクリックします。



- [Always Encrypted] ウィザードが起動したら、最初のページでは [次へ] ボタンをクリックします。

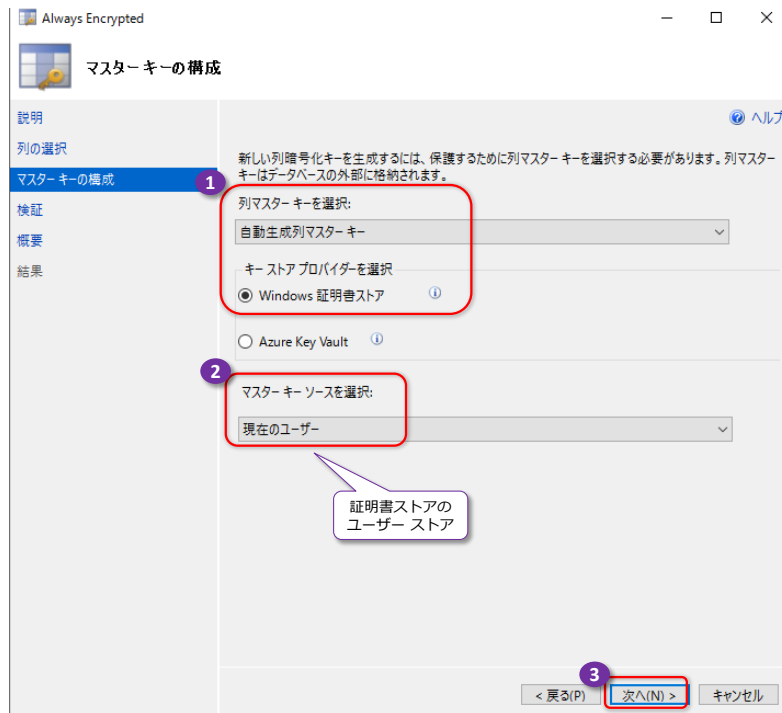


3. 次の「列の選択」ページでは、「CEK\_Auto1 (新規)」が選択されていることを確認します。これは、列を暗号化するための「列の暗号化キー」(COLUMN ENCRYPTION KEY)になるもので、ウィザードによって自動的に作成されます。



「aeTest1」テーブルの列の一覧からは、「colB」と「colC」列をチェックして、これを暗号化するようにします。「暗号化の種類」では、「決定論的」または「ランダム化」を任意に設定します（暗号化をランダム化するかどうかを設定します。画面は「決定論的」を選択）。

4. 次の「マスター キーの構成」ページでは、列マスター キー (COLUMN MASTER KEY) をどこに作成するのかを設定します。



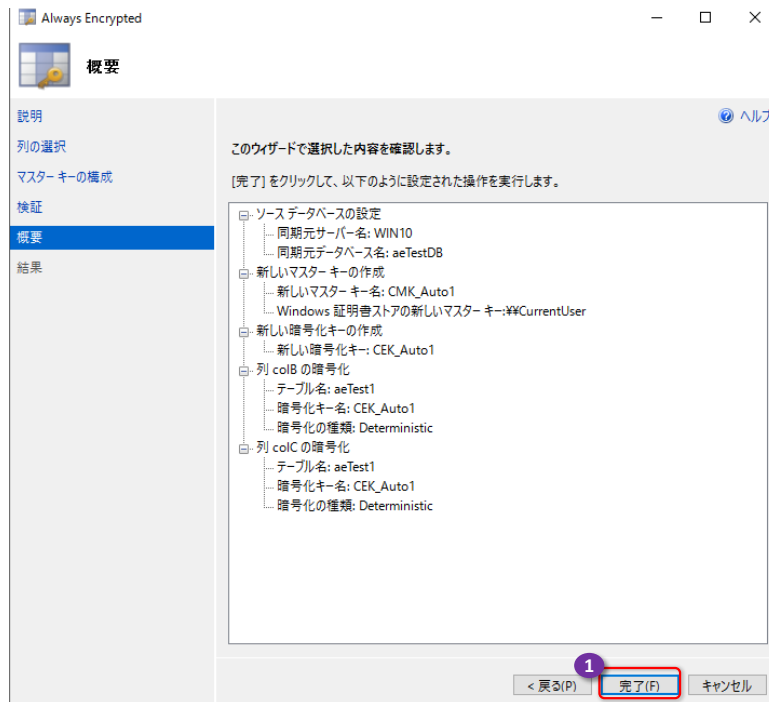
ここでは、「列マスター キーを選択」で「自動生成列マスター キー」、「キー ストア プロバイダーを選択」で「Windows 証明書ストア」、「マスター キー ソースを選択」で「現在のユーザー」が選択されていること（すべて既定値）を確認します。これで、ユーザーの証明書ストアに列マスター キーを作成することができます。

5. 次の「検証ページ」では、Always Encrypted を設定するためのスクリプトを生成するのか、今すぐ実行するのかを設定します。



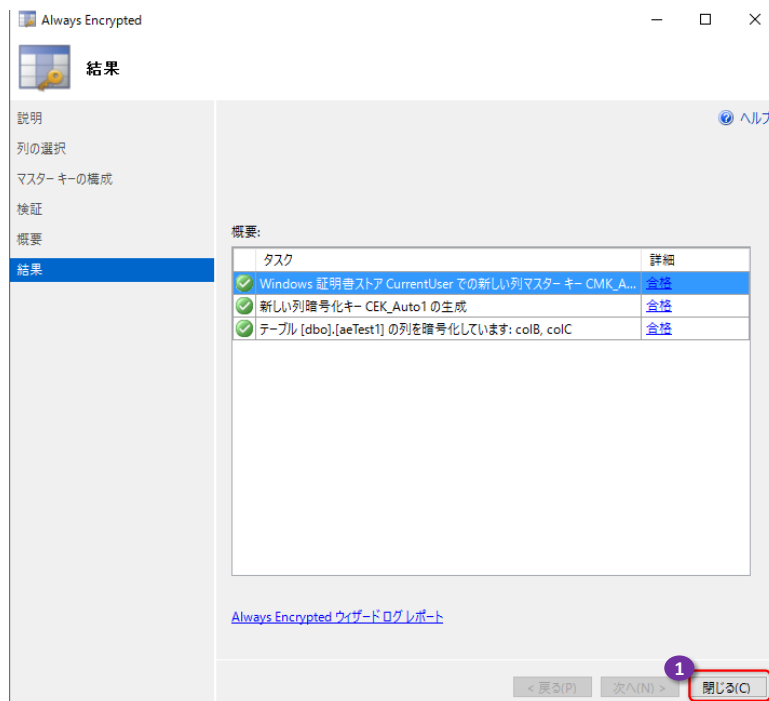
ここでは、**「続行して今すぐ完了」**を選択して、すぐに実行するようにします。

6. 最後の**「概要」**ページでは、設定内容を確認して、**「完了」**ボタンをクリックします。



以上で、**Always Encrypted のすべての設定（列マスターキーや、列の暗号化キーの作成、colB と colC 列に対する暗号化の実行）**が完了です。

7. すべての実行が成功すると（暗号化が完了すると）、次のように**「結果」**ページで**「合格」**と表示されるので、**「閉じる」**ボタンをクリックします。



## ➡ 暗号化されていることの確認

次に、**Always Encrypted** によって、列データが暗号化されたことを確認してみましょう。

1. **SELECT** ステートメントを実行して、「aeTest1」テーブルの中身を参照してみます。

```
SELECT * FROM aeTest1
```

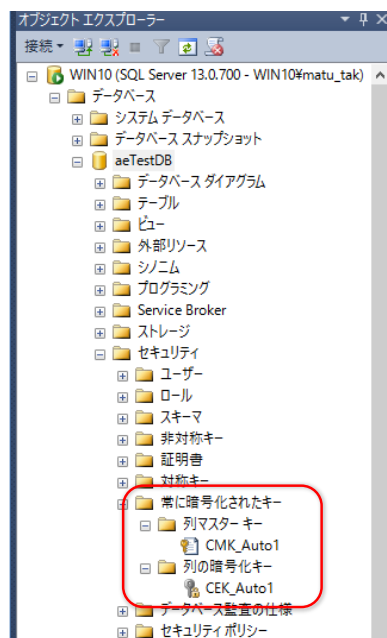
-- データの確認  
SELECT \* FROM aeTest1

暗号化されていることを確認

	colA	colB	colC
1	1	0x01B44A66DBF0B586803561098905B0EFEEA...	0x0112FD74802543520C46C5F9135E0B80DDDEA024AE4A000...
2	2	0x0137A2B5E14A4F1E8C760651254DA459349...	0x016747C02567D8F26EA69C61F35B6980715E95EFF0E9822F...

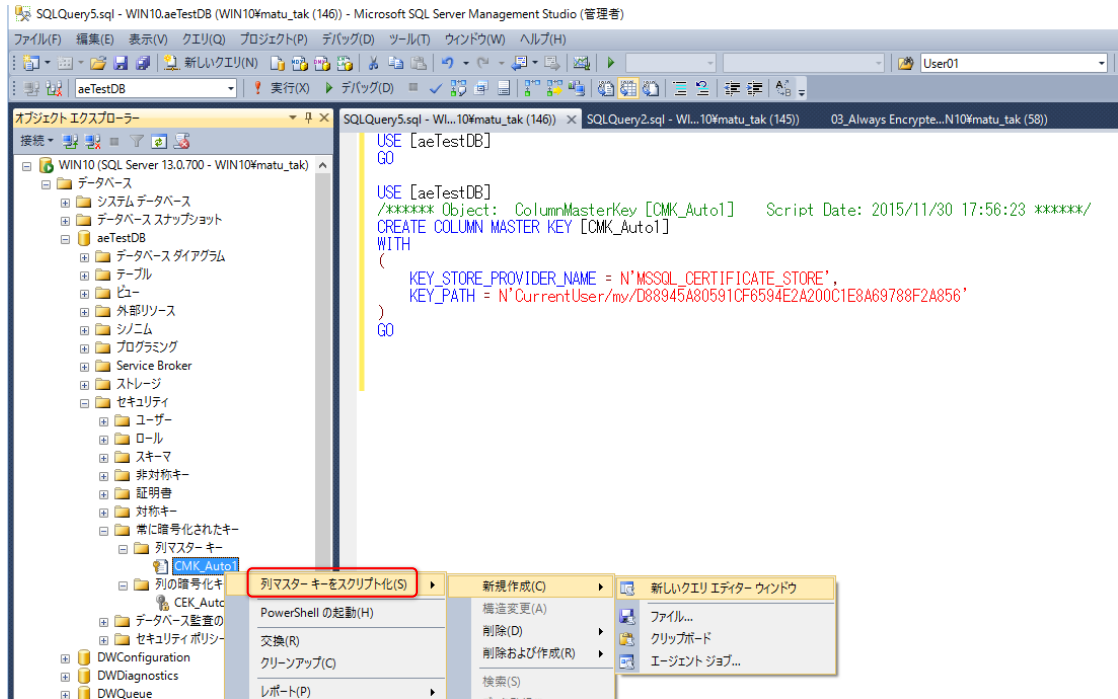
**colB** と **colC** 列が暗号化されていることを確認できます。このように、**Always Encrypted** を設定すると、列データを暗号化することができます。

2. 次に、ウィザードによって自動作成された**列マスター キー (COLUMN MASTER KEY)** と **列の暗号化キー (COLUMN ENCRYPTION KEY)** を確認してみましょう。次のようにオブジェクト エクスプローラーで **[セキュリティ]** の **[常に暗号化されたキー]** フォルダを展開します。

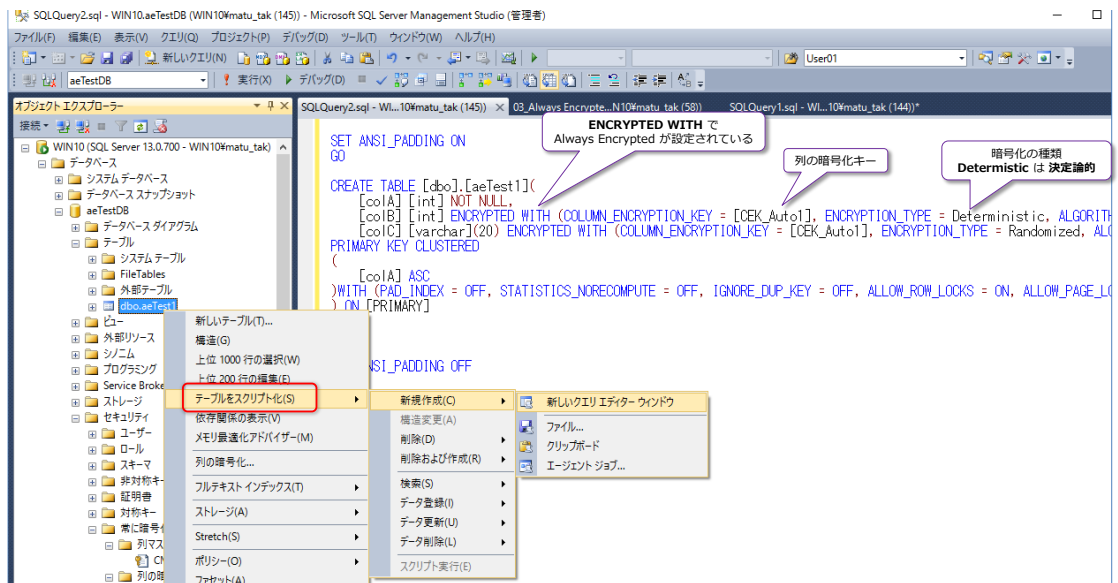


「**CMK\_Auto1**」という名前の**列マスター キー**、「**CEK\_Auto1**」という名前の**列の暗号化キー**が作成されていることを確認できます。

これらのキーは、次のように右クリックして、スクリプトを生成することで、ウィザードによってどのようにキーが作成されたのかを確認することもできます。



また、テーブル「aeTest1」を、次のようにスクリプト化すれば、colB と colC 列に対する暗号化の設定も確認することができます。



生成された CREATE TABLE ステートメントの colB と colC では、ENCRYPTED WITH があり、COLUMN\_ENCRYPTION\_KEY（列の暗号化キー）で CEK\_Auto1 が指定され、ENCRYPTION\_TYPE（暗号化の種類）で Deterministic なら「決定論的」、Randomized なら「ランダム化」になります。

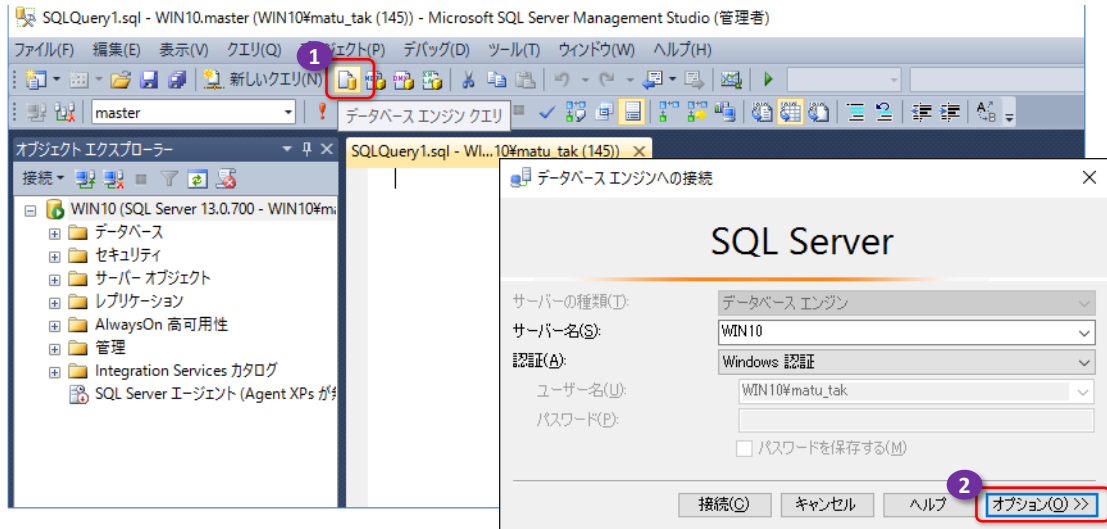
## ➡ 接続パラメーターに Column Encryption Setting=Enabled を追加

Management Studio では、接続時のパラメーターに「Column Encryption Setting=Enabled」を追加することで、Always Encrypted で暗号化されたデータを参照することができます。これも



試してみましょう。

1. まずは、ツールバーの「データベース エンジン クエリ」ボタンをクリックして、新しい接続を追加します。



「データベース エンジンへの接続」ダイアログが表示されたら、「オプション」ボタンをクリックします。

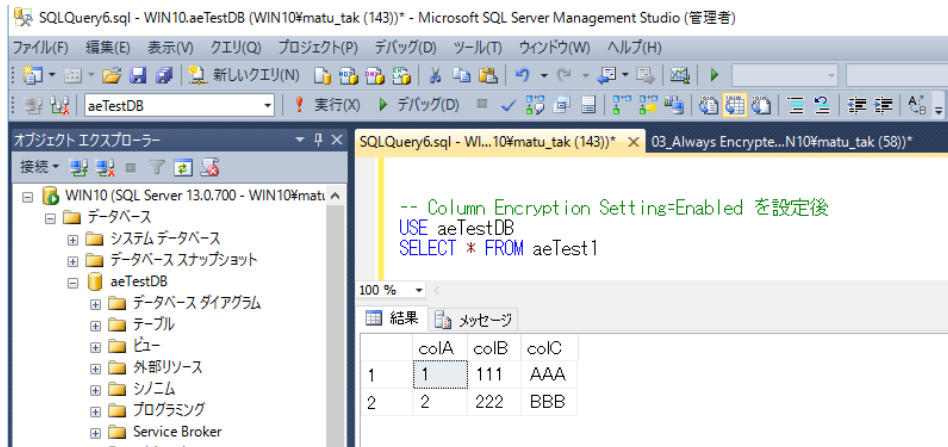
2. オプション設定ダイアログでは、「追加の接続パラメーター」タブをクリックして、「Column Encryption Setting=Enabled」と入力し、「接続」ボタンをクリックします。



これで、暗号化されたデータを参照できるようになります。

3. 接続完了後、**SELECT** ステートメントを実行して、「aeTest1」テーブルを参照し、暗号化されたデータを参照できることを確認しておきましょう。

```
USE aeTestDB
SELECT * FROM aeTest1
```



なお、[追加の接続パラメーター] で設定した「**Column Encryption Setting=Enabled**」は、Management Studio を終了しても設定が残っている（次の接続時も Enabled で接続になる）ので、元に戻したい場合には、[追加の接続パラメーター] を「空」にするようにします。

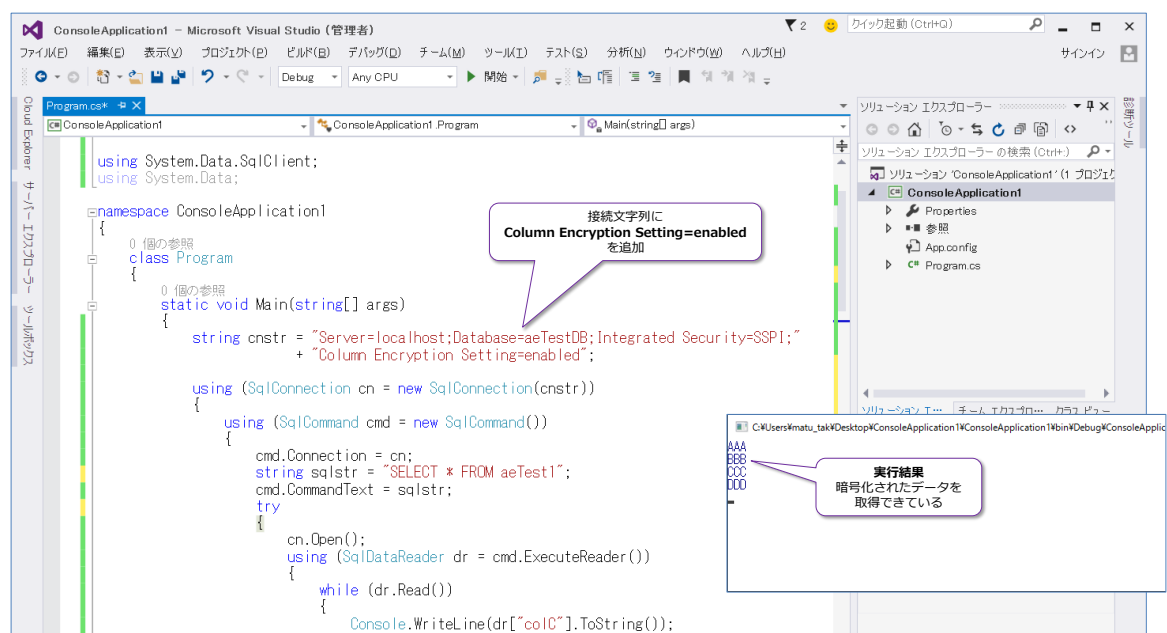
## ➡ アプリケーションからの接続（.NET Framework 4.6 の場合）

次に、暗号化されたデータをアプリケーションから参照してみましょう。ここでは、**Visual Studio 2015 (C#)** および **.NET Framework 4.6** を例に説明します。

### ADO.NET の場合

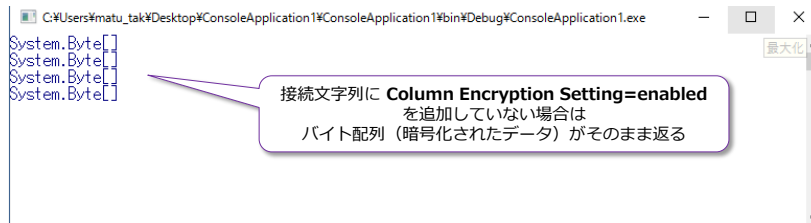
.NET Framework アプリケーション(**ADO.NET**)から Always Encrypted を利用するには、**.NET Framework 4.6** 以上が必要になります。**.NET Framework 4.6** 以上を利用している場合には、次のように**接続文字列**に「**Column Encryption Setting=enabled**」を追加します。

```
string cnstr = "Server=localhost;Database=aeTestDB;Integrated Security=SSPI;"
              + "Column Encryption Setting=enabled";
```

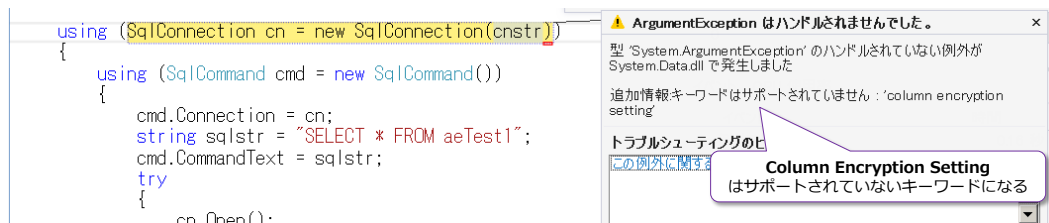


そのほかのコードは修正する必要はなく、これだけで暗号化されたデータを取得することができます。

なお、接続文字列に「**Column Encryption Setting=enabled**」を追加していない場合には、実行結果は次のようになります。

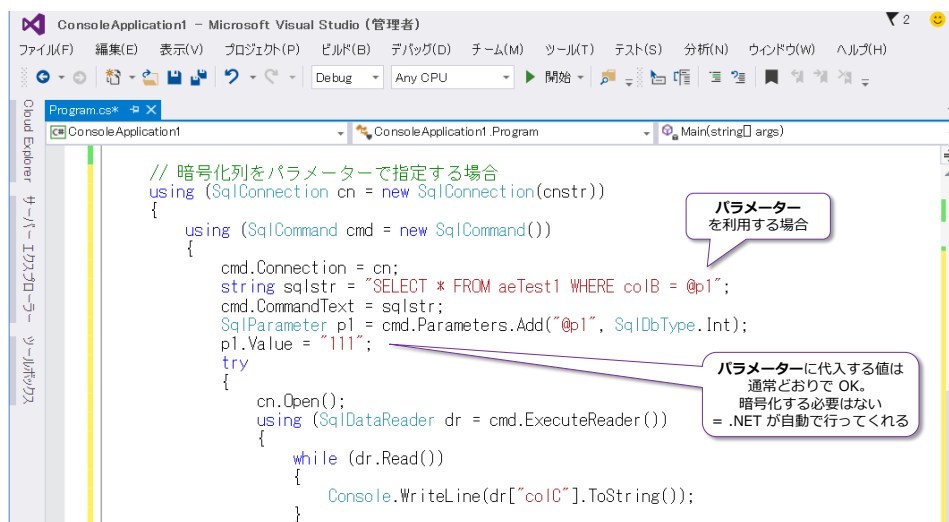


復号化は行われず、暗号化されたデータをそのまま取得する形になってしまいます。また、古いバージョンの .NET Framework をターゲット フレームワークに指定している場合には、次のようにサポートされていないキーワードとして扱われます。

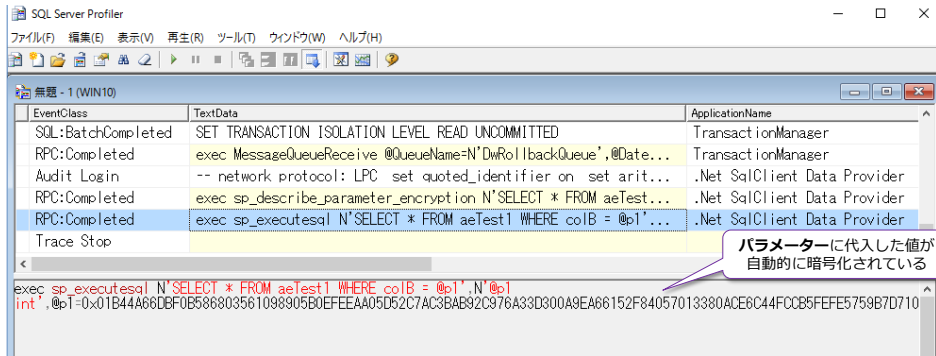


## ADO.NET でのパラメーター (SqlParameter) 場合

ADO.NET のパラメーター (SqlParameter) も、通常と同じように利用することができます。



パラメーターに代入する値を暗号化する必要はなく、通常どおりで平文で（そのまま）指定します。これで、内部的には .NET Framework が自動的に暗号化を行ってくれます（接続文字列に **Column Encryption Setting=enabled** 指定されている場合の動作）。この動作は、**SQL Server Profiler** ツールを利用して確認することができます。次の画面は、上のアプリケーションを実行したときに、SQL Server Profiler ツールでキャプチャしたものです。

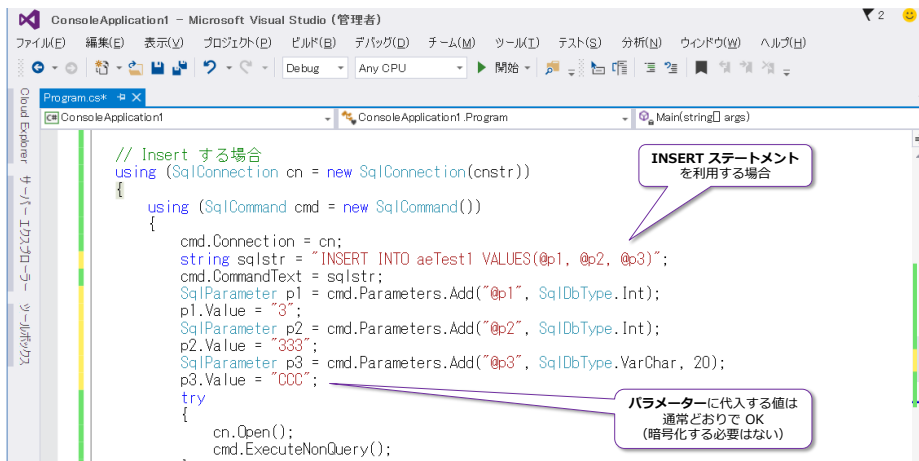


パラメーターに代入した値（コードでは **colB** 列に対して **111** を代入）が、.NET Framework によって自動的に暗号化されて、**0x01B44A66DB~** という値に変換されて、SQL Server に送信されています。SQL Server 側では、暗号化された状態でデータが格納されている（**0x01B44A66DB~** という形式で格納されている）ので、この該当データをアプリケーションに返すことができます。

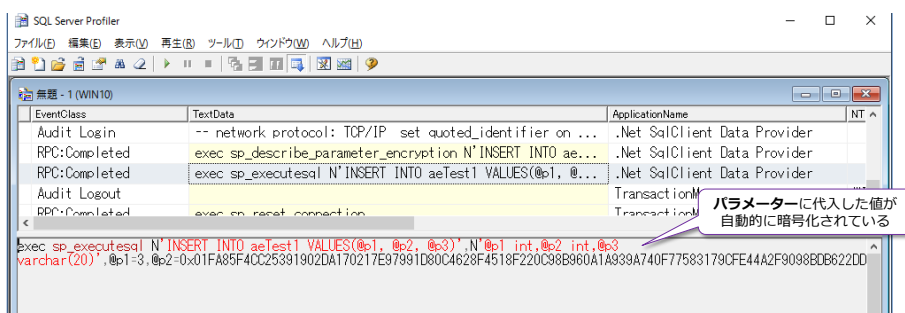
このように、Always Encrypted では、アプリケーション側で（内部的に）暗号化と複合化を行ってくれることで、アプリケーション コードを変更することなく、列データを暗号化できるようになっています。

## ADO.NET での INSERT の場合

INSERT ステートメントなどの更新系のステートメントについても、通常の ADO.NET アプリケーションと同様に利用することができます。

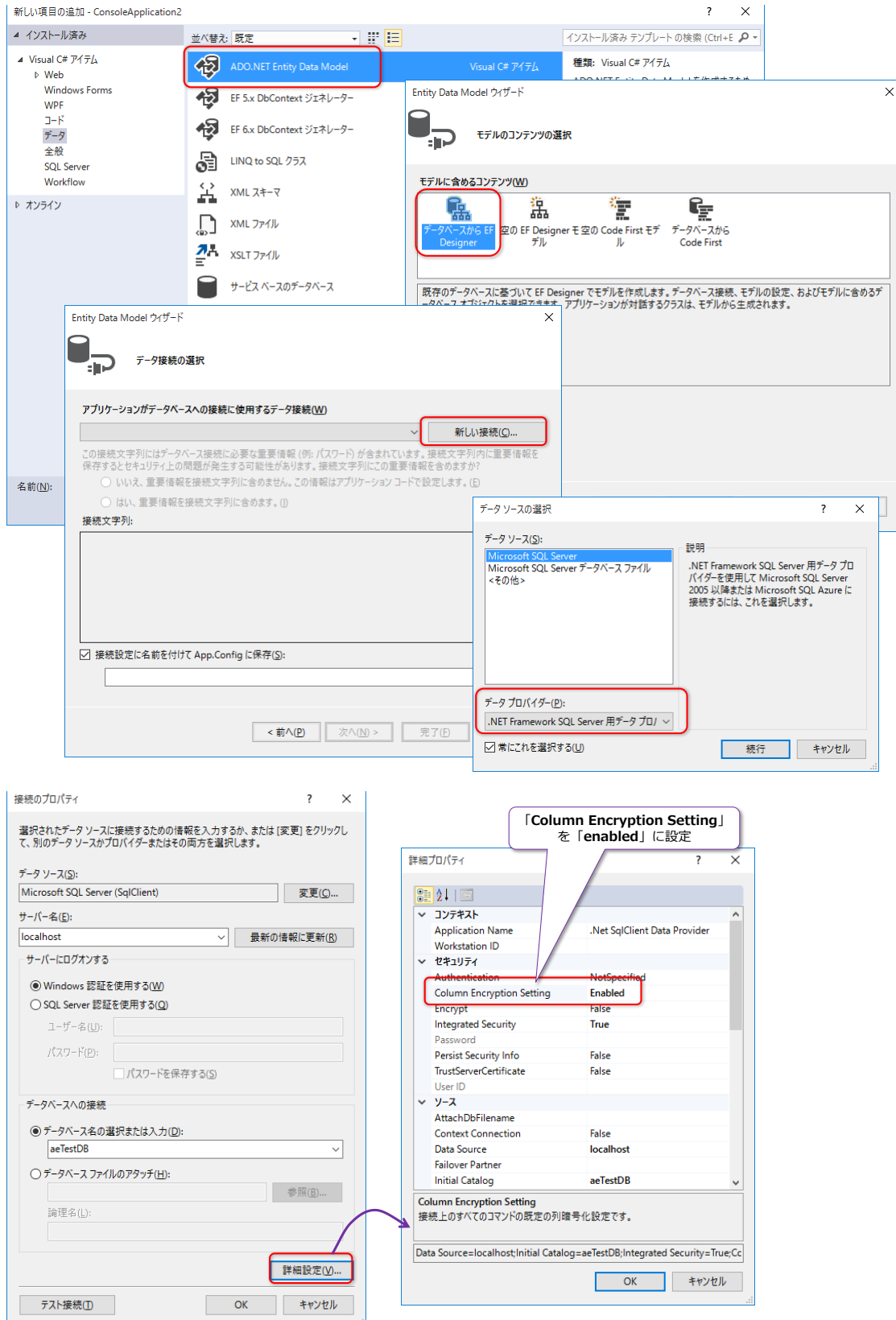


パラメーターに代入した値は、次のように自動的に暗号化されて SQL Server に送信されます。



## Entity Framework を利用する場合

Entity Framework を利用する場合には、次のように「**接続のプロパティ**」(.NET Framework SQL Server 用データ プロバイダー) で、「**詳細設定**」ボタンをクリックして、「**Column Encryption Setting**」を「**enabled**」に設定します。

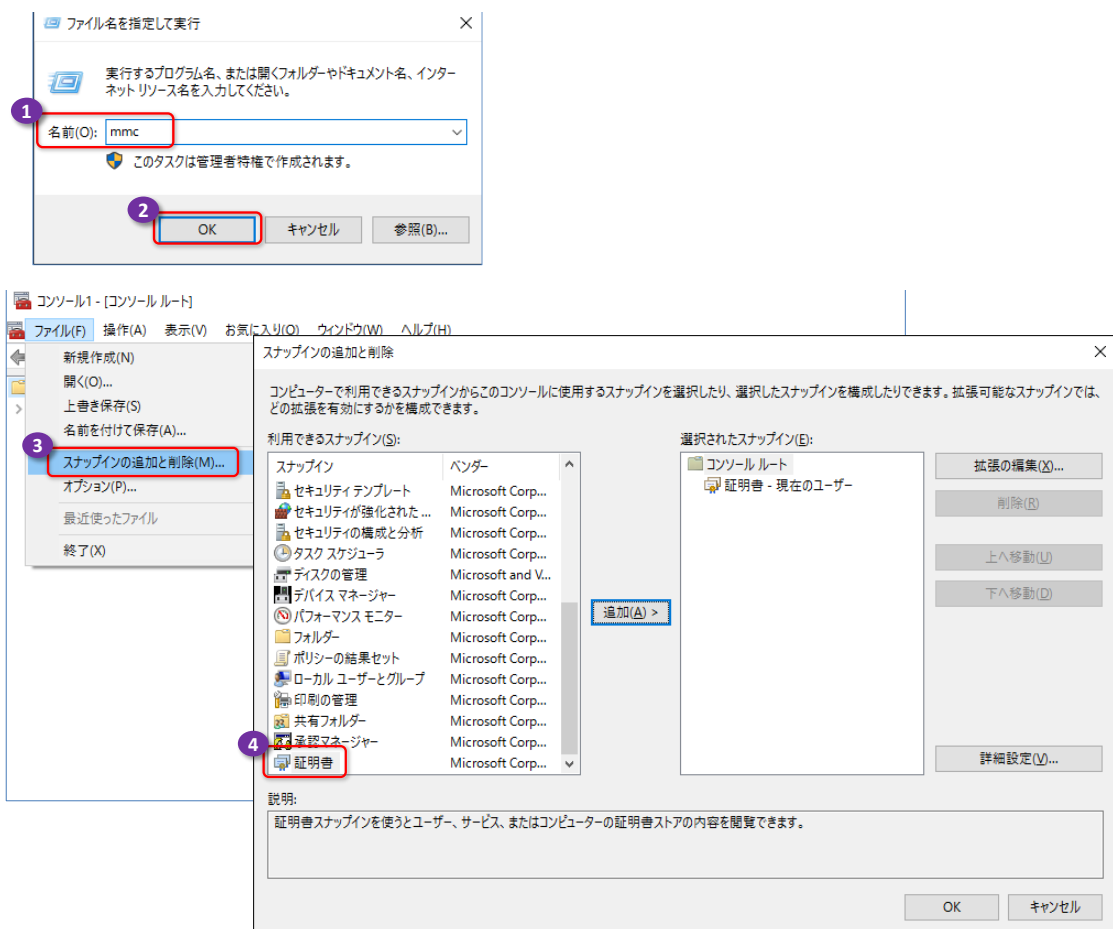


## ➡ SQL Server とは別のマシンからの接続（証明書のエクスポート／インポート）

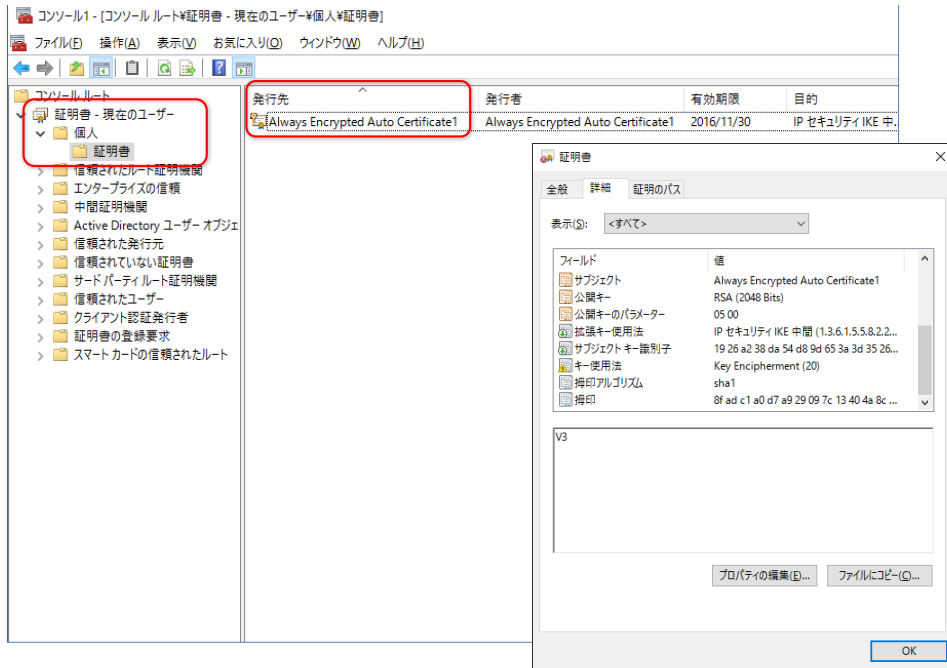
Always Encrypted では、（内部的には）証明書を利用して暗号化を行っているので、SQL Server とは別のマシンのアプリケーションから暗号化データにアクセスするには、証明書を複製しておく 必要があります。これを行っていない場合には、別のマシンからアクセスした場合に、次のように「暗号化解除できませんでした」エラーが通達されます。



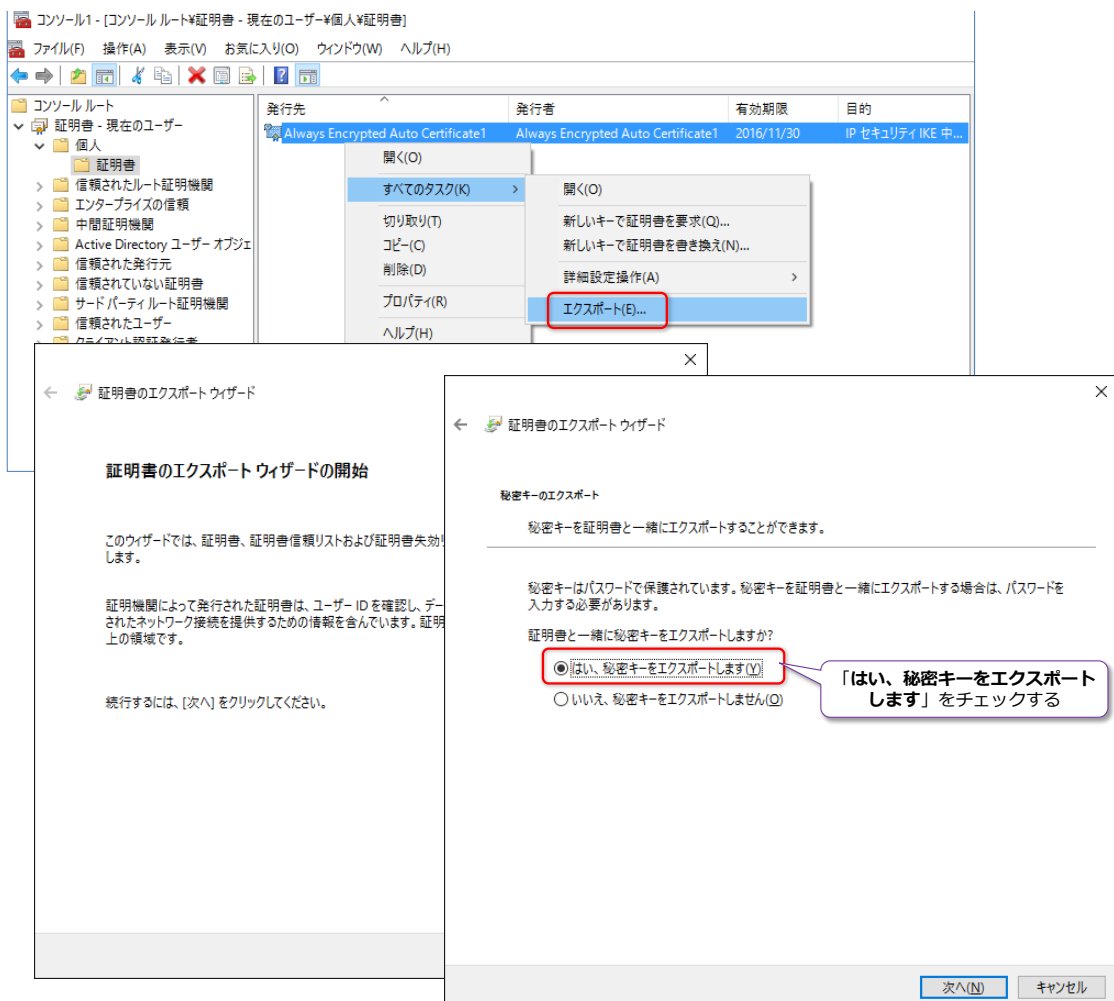
これを回避するには、証明書を複製しておくようにします。証明書は、Always Encrypted をウィザードの既定値で設定した場合には、「**Windows 証明書ストア**」の「**ユーザー ストア**」に作成されています。これを確認するには、[スタート]メニューの[ファイル名を指定して実行]から「mmc」と入力して、管理コンソールを起動し、[ファイル]メニューの[スナップインの追加と削除]で[証明書]を追加します。



[証明書] スナップインを追加したら、[証明書 - 現在のユーザー] の[個人] → [証明書] フォルダを展開します。



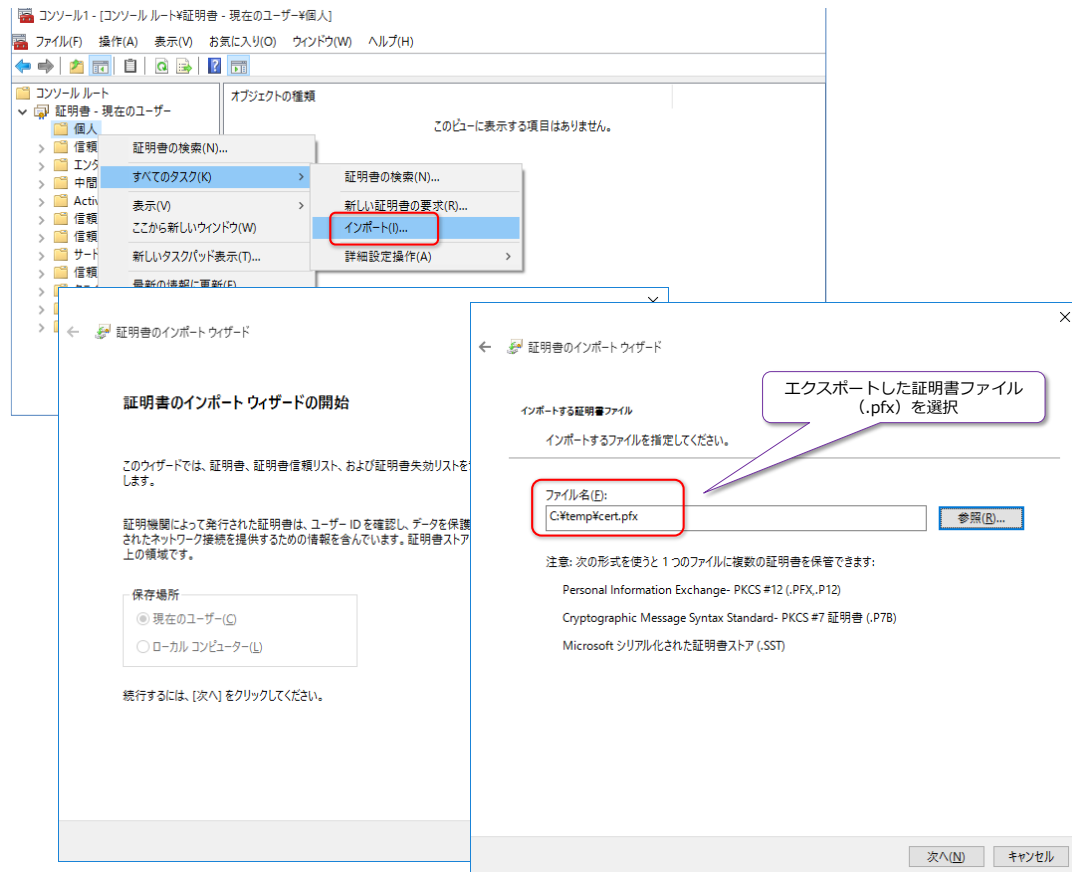
ここに「**Always Encrypted Auto Certificate1**」という名前の証明書が作成されていることを確認できます。この証明書をエクスポートするには、次のように証明書を右クリックして、[すべてのタスク] メニューから [エクスポート] をクリックします。





エクスポート時は、「はい、秘密キーをエクスポートします」をチェックして、秘密キーも含めてエクスポートするようにします（そのほかの設定は既定値、パスワードの部分は任意のパスワードを設定するようにします）。

エクスポートした証明書（既定では .pfx 形式のファイル）は、別マシンにコピーして、別マシン上で同じく「証明書」スナップインを利用して、次のようにインポートすることができます。

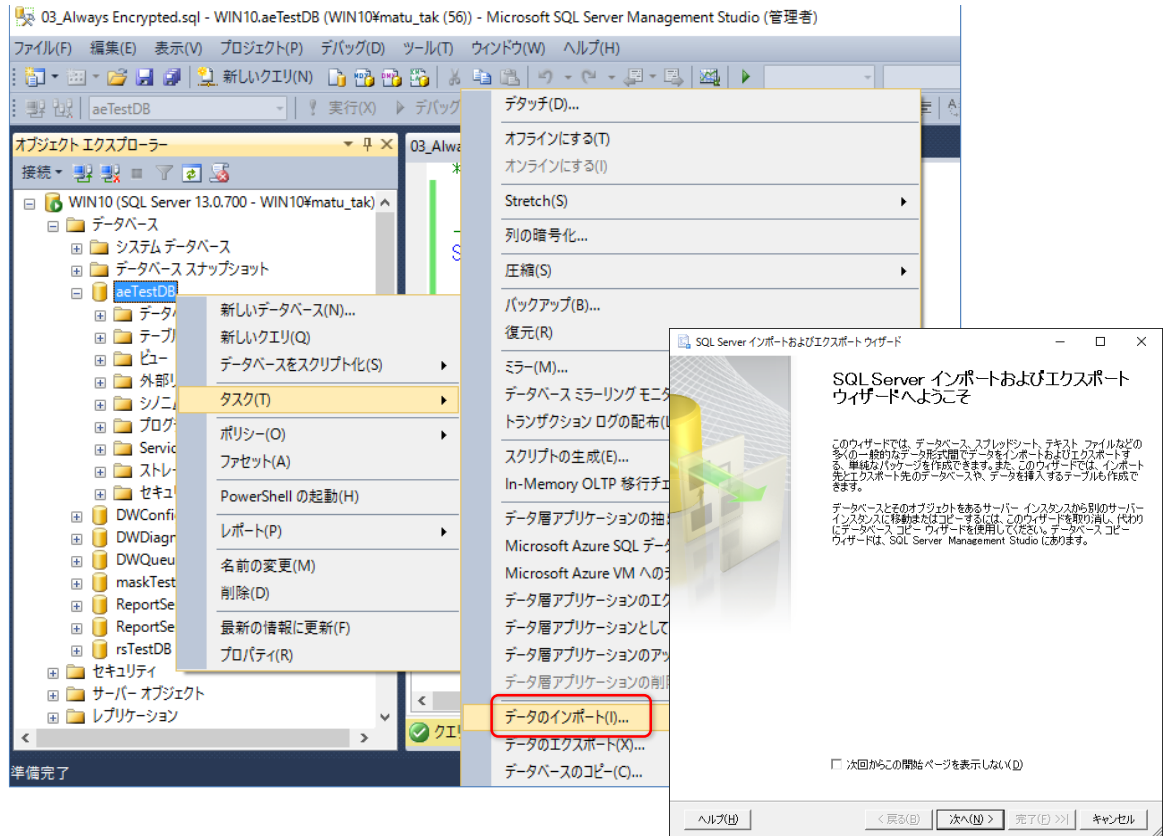


後は、エクスポート時に設定したパスワードを入力すれば、証明書のインポートが完了です。インポートが完了した後は、暗号化データを処理できるようになります。

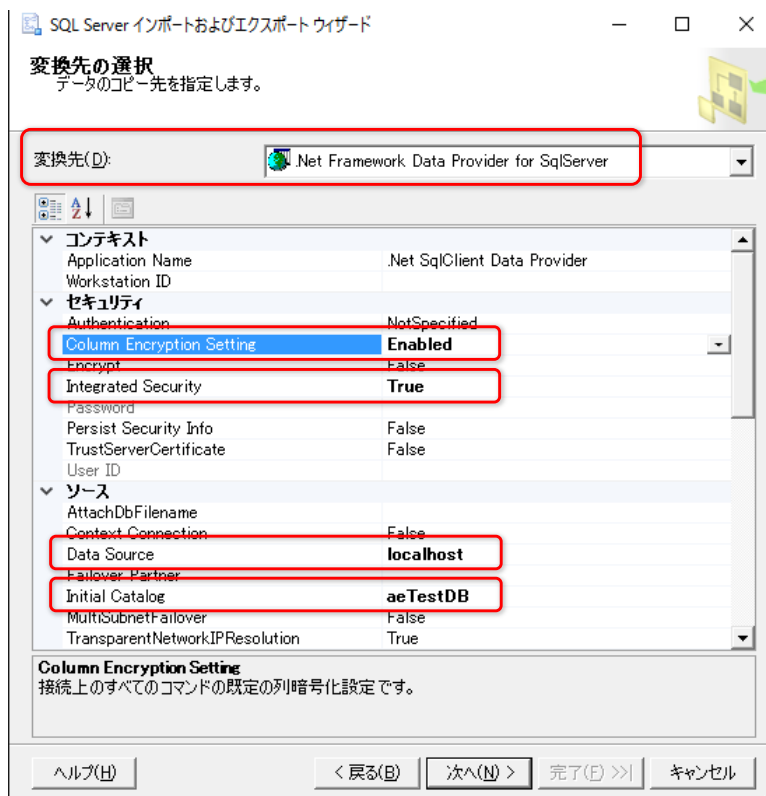
このように、ネットワークをまたがったとしても、**同じ証明書**（証明書の複製）を持っていれば、暗号化／復号化を行うことができるので、**Always Encrypted** では、**ネットワーク上を流れるデータについても、暗号化されること**になります（データベース内のデータは、常に暗号化されていて、復号化あるいは暗号化はアプリケーション側で実行することになるので、ネットワーク上を流れるデータは暗号化されたデータのみになります）。

## ➡ インポート ウィザードでデータのインポート（既存のテーブルからのデータ移行 etc）

既存のテーブルやテキスト ファイルなどから、Always Encrypted を設定したテーブルにデータをインポートしたい場合には、Integration Services の「**SQL Server インポートおよびエクスポート ウィザード**」を利用することができます。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「**データのインポート**」から起動できます。



このウィザードでは、[**変換先の選択**] ページで、次のように [**変換先**] で [**.NET Framework Data Provider for SqlServer**] を選択することがポイントです。



これで、詳細プロパティが表示されるようになるので、[**Data Source**] に SQL Server の名前、[**Initial Catalog**] に取り込み先のデータベース名、認証方法は [**Integrated Security**]

(Windows 認証での接続の場合は **True** に設定) または **[User ID]** と **[Password]** を設定 (SQL Server 認証での接続の場合) して、**「Column Encryption Setting」** を **「Enabled」** に設定します。これで、Always Encrypted が設定されたテーブルでも、データをインポートすることができます。

このように、**Always Encrypted** を利用すれば、ネットワーク上を流れるデータも、データベース内に格納されるデータも、すべて暗号化して格納できるようになります。データの移行も、インポート/エクスポート ウィザードを利用して簡単に行うことができます。

その他、**Always Encrypted** に関する最新情報は、オンライン ブックの次のトピックが参考になると思います。

Always Encrypted (Database Engine)

<https://msdn.microsoft.com/en-us/library/mt163865.aspx>

...
Choose an Encryption Algorithm

- Transparent Data Encryption (TDE)
- SQL Server and Database Encryption Keys
- Always Encrypted**
  - Always Encrypted (client development)
  - Always Encrypted Wizard
  - Migrate Sensitive Data Protected by Always Encrypted
  - Column Master Key Rotation

## Always Encrypted (Database Engine)

**SQL Server 2016**

Updated: November 25, 2015

Applies To: Azure SQL Database, SQL Server 2016 Preview

Always Encrypted is a feature designed to protect sensitive data, such as credit card numbers or national identification numbers (e.g. U.S. social security numbers), stored in Azure SQL Database or SQL Server databases. Always Encrypted allows clients to encrypt sensitive data inside client applications and never reveal the encryption keys to the Database Engine (SQL Database or SQL Server). As a result, Always Encrypted provides a separation between those who own the data (and can view it) and those who manage the data (but should have no access). By ensuring on-premises database administrators, cloud database operators, or other high-privileged, but unauthorized users, cannot access the encrypted data, Always Encrypted enables customers to confidently store sensitive data outside of their direct control. This allows organizations to encrypt data at rest and in use for storage in Azure, to enable delegation of on-premises database administration to third parties, or to reduce security clearance requirements for their own DBA staff.

Always Encrypted makes encryption transparent to applications. An Always Encrypted-enabled driver installed on the client computer achieves this by automatically encrypting and decrypting sensitive data in the client application. The driver encrypts the data in sensitive columns before passing the data to the Database Engine, and automatically rewrites queries so that the semantics to the application are preserved. Similarly, the driver transparently decrypts data, stored in encrypted database columns, contained in query results.

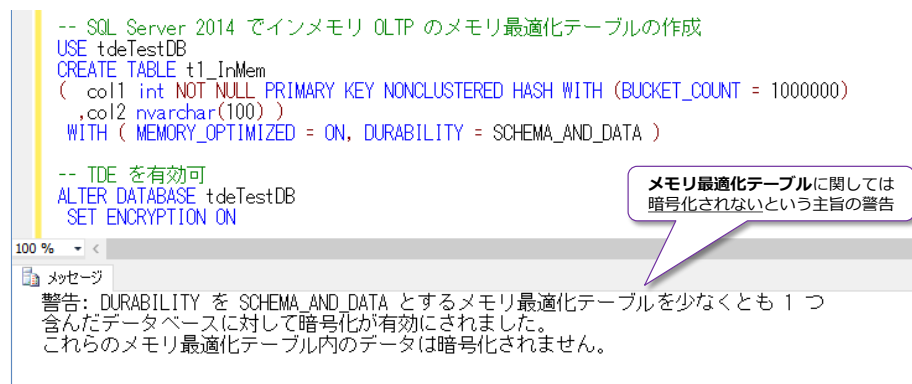
Always Encrypted is available in SQL Server 2016 Community Technology Preview 3 (CTP 3.0) and in preview in SQL Database. For Channel 9 presentations that includes Always Encrypted, see [SQL Server 2016 Always Encrypted and Overview and Roadmap for Microsoft SQL Server Security](#).

### 3.5 TDE（透過的なデータ暗号化）の性能向上、インメモリ OLTP 対応

TDE（Transparent Data Encryption: 透過的なデータ暗号化）は、データベースの実体となるファイル（.mdf/.ldf）およびバックアップ ファイルを暗号化することができる機能で、SQL Server 2008 から提供されました。SQL Server 2016 では、TDE のパフォーマンスを向上させるために、Intel の **AES-NI (AES-New Instructions: AES 暗号化の処理を高速化するための命令セット)** に対応しました。AES-NI は、**ハードウェア アクセラレーター**で、これを利用することで、**TDE における CPU のオーバーヘッドを軽減**することができます。

#### ➡ インメモリ OLTP への対応

SQL Server 2016 からは、TDE（透過的なデータ暗号化）がインメモリ OLTP 機能に対応するようになりました。SQL Server 2014 を利用している場合には、データベース内に、インメモリ OLTP を利用したテーブル（メモリ最適化テーブル）がある場合には、次のようなメッセージが表示されていました。



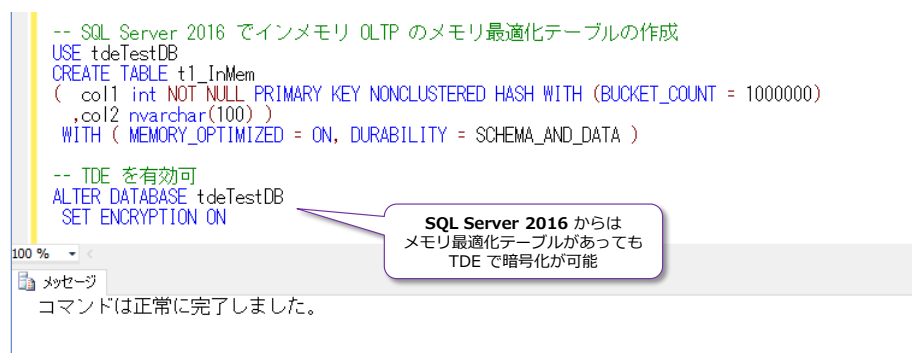
```
-- SQL Server 2014 でインメモリ OLTP のメモリ最適化テーブルの作成
USE tdeTestDB
CREATE TABLE t1_InMem
( col1 int NOT NULL PRIMARY KEY NONCLUSTERED HASH WITH (BUCKET_COUNT = 1000000)
, col2 nvarchar(100) )
WITH ( MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA )

-- TDE を有効可
ALTER DATABASE tdeTestDB
SET ENCRYPTION ON
```

メモリ最適化テーブルに関しては暗号化されないという主旨の警告

警告: DURABILITY を SCHEMA\_AND\_DATA とするメモリ最適化テーブルを少なくとも 1 つ含んだデータベースに対して暗号化が有効にされました。これらのメモリ最適化テーブル内のデータは暗号化されません。

TDE は設定できるものの、メモリ最適化テーブルに関しては暗号化ができないという状態でした。これが SQL Server 2016 からは、インメモリ OLTP にも対応するようになったので、メモリ最適化テーブルでも暗号化ができるようになりました。



```
-- SQL Server 2016 でインメモリ OLTP のメモリ最適化テーブルの作成
USE tdeTestDB
CREATE TABLE t1_InMem
( col1 int NOT NULL PRIMARY KEY NONCLUSTERED HASH WITH (BUCKET_COUNT = 1000000)
, col2 nvarchar(100) )
WITH ( MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA )

-- TDE を有効可
ALTER DATABASE tdeTestDB
SET ENCRYPTION ON
```

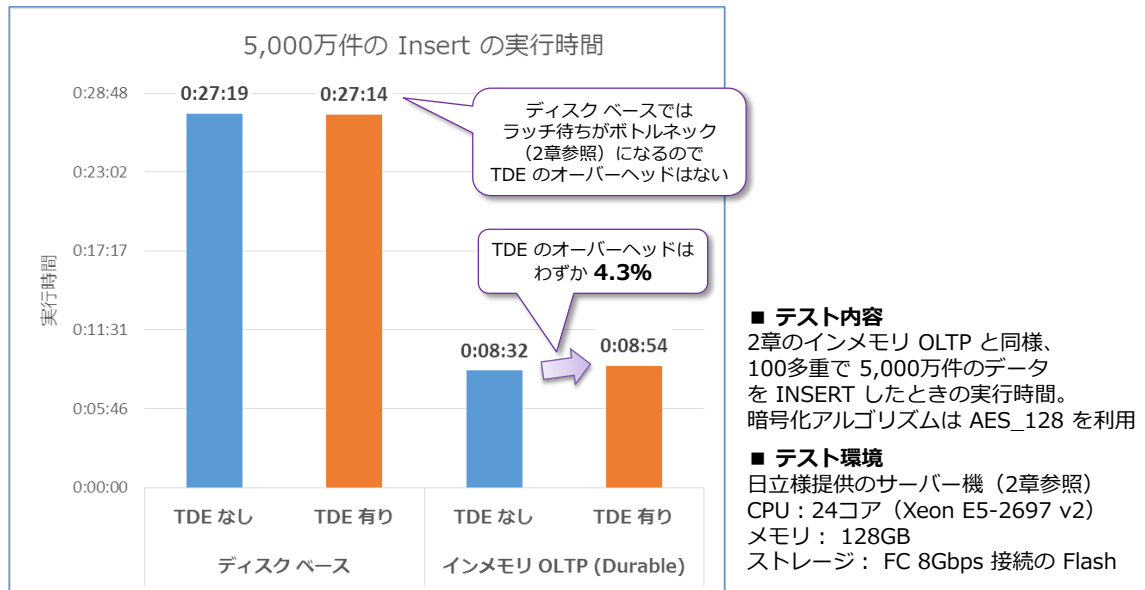
SQL Server 2016 からはメモリ最適化テーブルがあっても TDE で暗号化が可能

メッセージ  
コマンドは正常に完了しました。

#### ➡ TDE のオーバーヘッド ～性能検証結果～

TDE（透過的なデータ暗号化）を設定したことによる性能のオーバーヘッド（暗号化処理のオーバ

ーヘッドなど）が気になる方は多いと思います。SQL Server 2016 での TDE（透過的なデータ暗号化）のオーバーヘッドを検証してみた結果が、次のグラフです（2 章の インメモリ OLTP での検証と同様、100 多重で 5,000 万件の INSERT を行った場合の検証）。



\* 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、ラッチ待ちがボトルネックになってしまうので、TDE を設定しても、設定しなくても**結果は変わらない**（オーバーヘッドは確認できない）、**インメモリ OLTP の Durable**（SCHEMA\_AND\_DATA）では、TDE を設定しない場合が **8 分 32 秒**であるのに対して、TDE を設定したとしても **8 分 54 秒**で完了、**わずか 4.3%** のオーバーヘッドであることを確認できました。

なお、SQL Server 2014 と SQL Server 2016 でのオーバーヘッドの差（AES-NI の効果）を検証しようとも試みたのですが、前述したように SQL Server 2014 では インメモリ OLTP で TDE を設定することができないので、SQL Server 2014 のときにどれぐらいのオーバーヘッドであったのかを知ることはできませんでした。また、ディスクベースの通常テーブルの場合は、SQL Server 2014 でも、TDE なしでも有りでも、SQL Server 2016 と同様、約 27 分の結果になり、ラッチ待ちがボトルネックになって、TDE のオーバーヘッドを確認することはできませんでした。

このように、TDE のオーバーヘッドを検証することは難しく、ディスクベースの通常テーブルを利用している場合には、ラッチ待ちやロック待ち、ログ書き込み待ち、チェックポイント負荷、インデックス更新のオーバーヘッドなど、さまざまな TDE 以外のオーバーヘッドが発生するので、それらがボトルネックになっている環境であれば、TDE を設定しても、設定しなくても、それらに足を引っ張られて、TDE のオーバーヘッドが存在しないように見えるというぐらい、オーバーヘッドが軽いものになっています。

TDE は、ここ数年、弊社へのお問い合わせが多くなっている機能です（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が大きな要因だと思っています）。後述しますが、TDE の設定は非常に簡単で、SQL Server 2016 からはインメモリ OLTP にも対応し、前掲のグラフのようにオーバーヘッドも軽く、アプリケーションは全く変更する必要がないので、

ぜひ活用してみてください。

## ➡ TDE（透過的なデータ暗号化）の設定方法

TDE（透過的なデータ暗号化）の利用方法は、以前のバージョンと変わりませんが、まだ試したことがない方も多いと思うので、ここではこれも試してみましょう。

TDE を設定するおおまかな流れは、次のとおりです。

1. **master** データベースに接続して、**データベース マスター キー**を作成（**CREATE MASTER KEY** ステートメントを利用）
2. **サーバー証明書**を作成（**CREATE CERTIFICATE** ステートメントを利用）
3. Management Studio のオブジェクト エクスプローラーで、該当データベースを右クリックして、[タスク] メニューの [データベース暗号化の管理] をクリックし、TDE を有効化する（または **CREATE DATABASE ENCRYPTION KEY** と **ALTER DATABASE .. SET ENCRYPTION ON** ステートメントを利用して TDE を有効化する）

## ➡ Let's Try

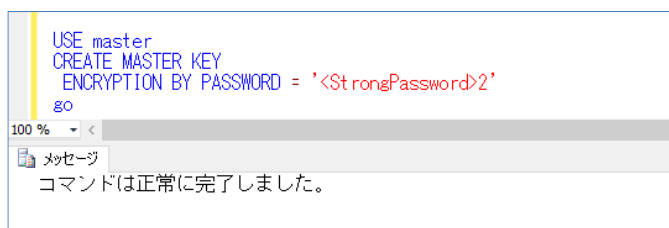
それでは試してみましょう。

1. まずは、TDE を試すためのデータベースを「**tdeTestDB**」という名前で作成します。

```
-- データベースの作成
USE master
CREATE DATABASE tdeTestDB
```

2. 次に、**CREATE MASTER KEY** ステートメントを利用して、**master** データベース内に**データベース マスター キー**を作成します。

```
-- データベース マスター キーの作成
USE master
CREATE MASTER KEY
ENCRYPTION BY PASSWORD = '<StrongPassword>'
```

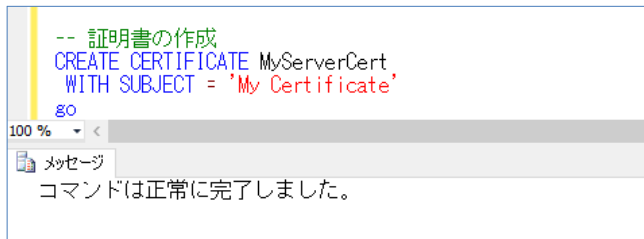


**ENCRYPTION BY PASSWORD** には、任意のパスワードを設定しますが、強固なパスワード

ド（大文字と小文字、@ などの特殊文字を含むなど）を設定するようにします。

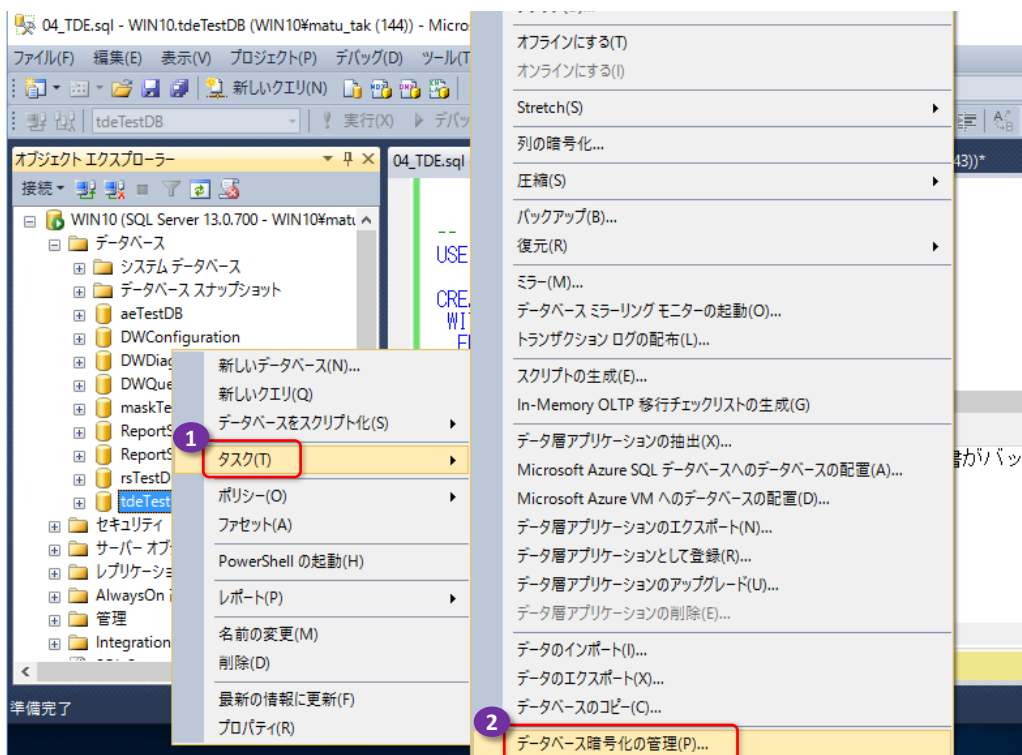
- 次に、**CREATE CERTIFICATE** ステートメントを利用して、**サーバー証明書**を作成します（これも **master** データベース内に作成します）。

```
-- サーバー証明書の作成
CREATE CERTIFICATE MyServerCert
WITH SUBJECT = 'My Certificate'
```



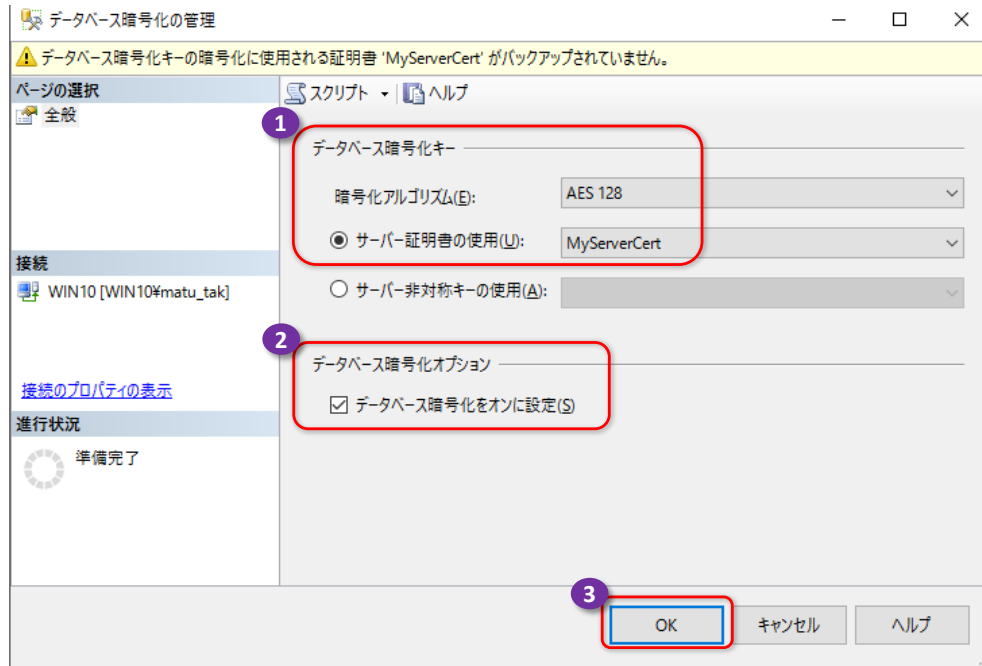
「**MyServerCert**」は、サーバー証明書の名前なので、任意の名前を設定できます。

- 次に、Management Studio のオブジェクト エクスプローラーで、次のようにデータベースを右クリックして、[タスク] メニューから [データベース暗号化の管理] をクリックします。



- [データベース暗号化の管理] ダイアログが表示されたら、[データベース暗号化キー] セクションで [暗号化アルゴリズム] に任意のアルゴリズム（画面は **AES\_128**）を選択して、[サーバー証明書を使用] で上の手順で作成した [MyServerCert] を選択します。





暗号化アルゴリズムでは、**AES\_128**、**AES\_192**、**AES\_256** などを選択できますが、キーの長さに応じて（128bit より 192bit、192bit より 256bit に変更することで）、セキュリティの強度を上げることができます（その反面、性能とのトレードオフがあります）。

後は、[**データベース暗号化をオンに設定**] をチェックして、[**OK**] ボタンをクリックすれば、TDE の設定が完了です。

なお、Management Studio ではなく、ステートメントで TDE を設定する場合には、次のように、**CREATE DATABASE ENCRYPTION KEY** ステートメントを利用して**データベース暗号化キー**を作成し、**ALTER DATABASE** ステートメントで **SET ENCRYPTION ON** を指定して、TDE を有効化します。

```
USE tdeTestDB

-- データベース暗号化キーの作成
CREATE DATABASE ENCRYPTION KEY
WITH ALGORITHM = AES_128
ENCRYPTION BY SERVER CERTIFICATE MyServerCert
go

-- データベースに対して暗号化を有効
ALTER DATABASE tdeTestDB
SET ENCRYPTION ON
go
```

以上で設定が完了です。

## ➡ 暗号化されていることの確認

TDE（透過的なデータ暗号化）によって、データやバックアップ ファイルが暗号化されていることを確認するには、次のように行えます。

```
USE tdeTestDB
-- テーブルの作成
CREATE TABLE t1
( a int , b varchar(100) )

-- データを 2件追加
INSERT INTO t1
VALUES (1, 'aaaaaaaaaa')
      , (2, 'あいうえお')
SELECT * FROM t1

-- データベースのバックアップ
BACKUP DATABASE tdeTestDB
TO DISK = 'C:¥temp¥tdeTestDB.bak'
```

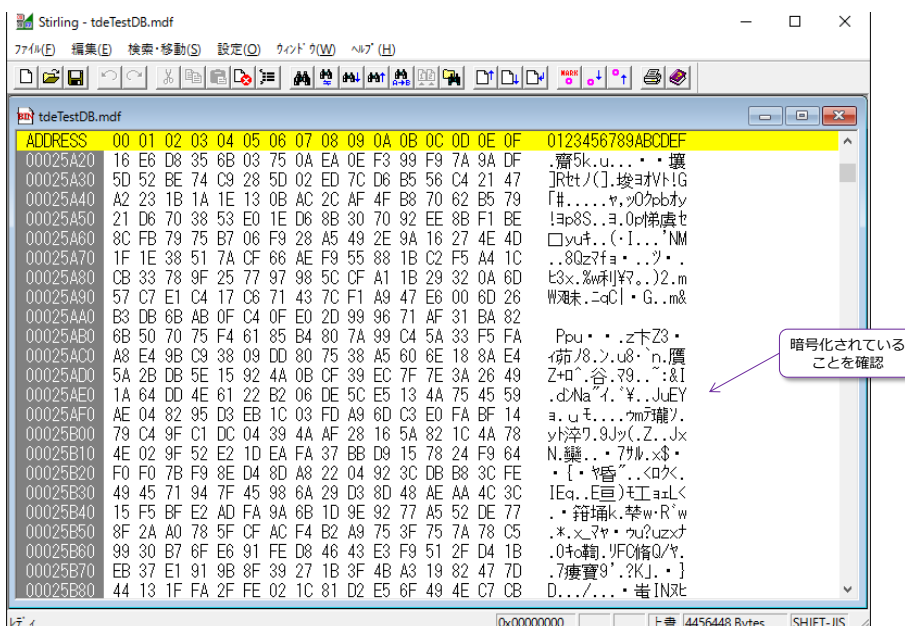
SELECT \* FROM t1

100 %

結果 メッセージ

	a	b
1	1	aaaaaaaaaa
2	2	あいうえお

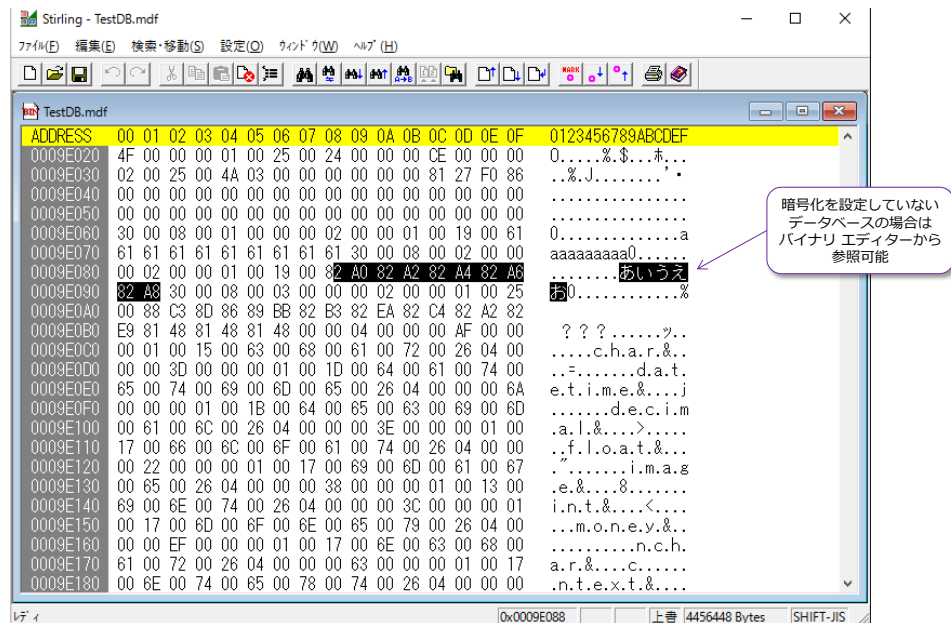
正しく暗号化されたかどうかは、SQL Server サービスを停止して、停止が完了した後に、「tdeTestDB.mdf」（データ ファイル）を任意のバイナリ エディター（Stirling など）で開くことで確認できます（.mdf は、既定では **C:¥Program Files¥Microsoft SQL Server¥MSSQL13.MSSQLSERVER¥MSSQL¥DATA** フォルダに格納されています）。



バイナリ エディターで開いても、こういったデータが格納されているかを読み取ることはできず、きちんと暗号化されていることを確認できます。同じように、バックアップ ファイル「tdeTestDB.bak」についても暗号化がされていることを確認することができます。

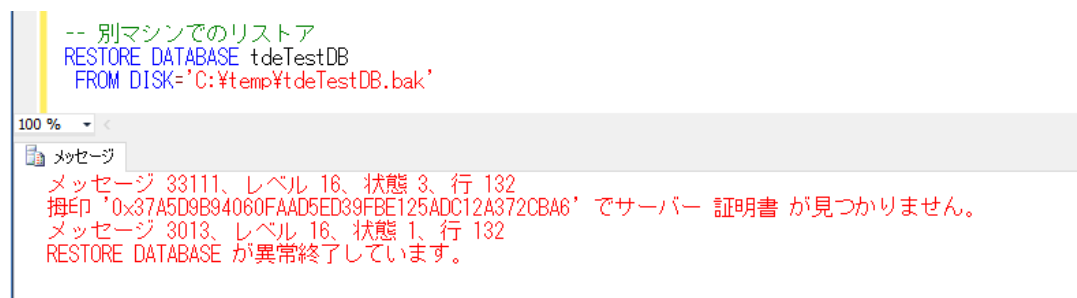
このように、TDE を利用すれば、物理ファイル（.mdf やバックアップ ファイル）を暗号化することができるのがメリットです。

なお、暗号化されていないデータベース（やバックアップ ファイル）をバイナリ エディターで開いた場合には、次のようにデータを読み取ることが可能です。



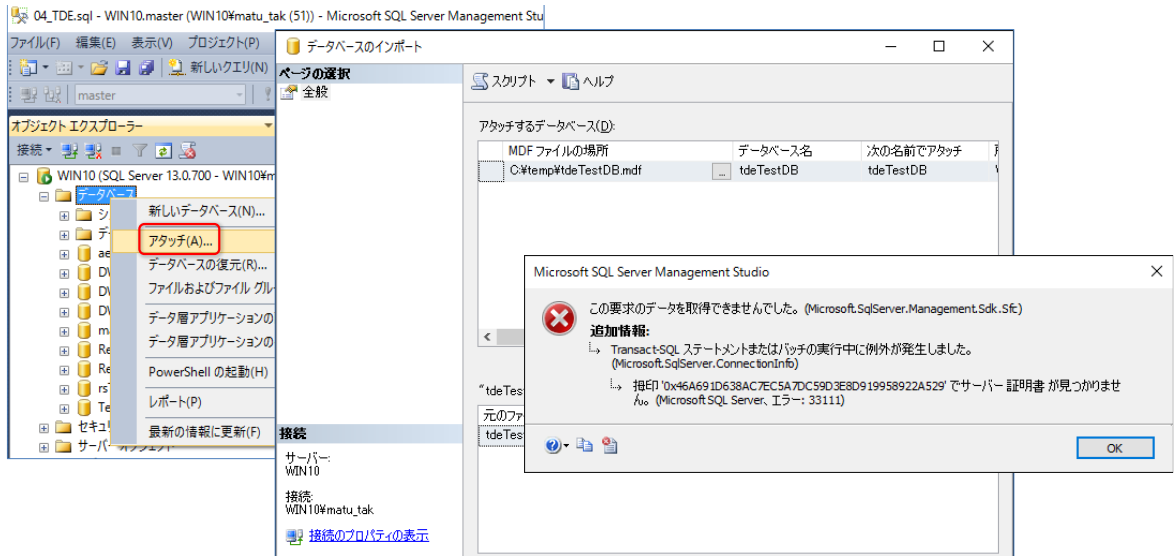
## ➡ 別マシンでのリストアはエラー

TDE（透過的なデータ暗号化）を設定したデータベースのバックアップは、別のマシンでリストアしようすると、次のようにエラーが発生します。



サーバー証明書が存在しないので、リストアができないという主旨です。TDE では、バックアップ ファイルが持ち出されたとしても、簡単にはリストアできないようになっています（かつ中身が暗号化されているので、安全性が高いものとなっています）。

また、データ ファイル（.mdf）を別マシンでアタッチしようとしても、次のように同じエラーが発生します。

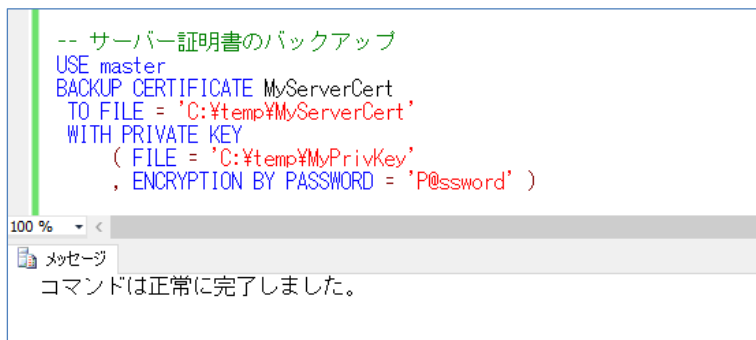


このように、データ ファイルについても、簡単にアタッチできないように、セキュリティが高いものになっています。

## ➡ サーバー証明書のバックアップとリストア

開発機から本番機へ、あるいはその逆へなど、別マシンへデータベースを移動させたい場合には、次のように **BACKUP CERTIFICATE** ステートメントを利用して、サーバー証明書（と秘密キー）のバックアップを実行しておく必要があります。

```
BACKUP CERTIFICATE MyServerCert
TO FILE = 'C:\%temp%\MyServerCert'
WITH PRIVATE KEY
( FILE = 'C:\%temp%\MyPrivKey'
, ENCRYPTION BY PASSWORD = 'P@ssword' )
```

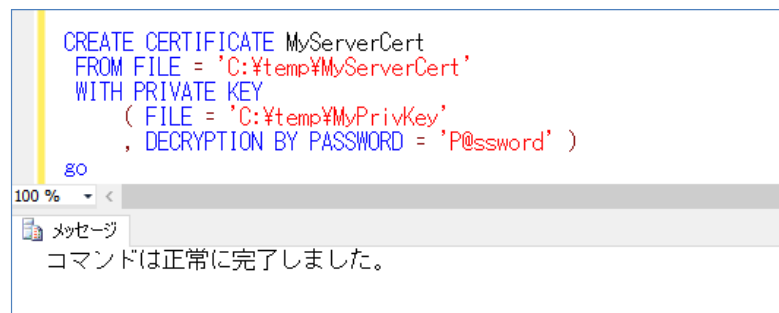


このステートメントでは、サーバー証明書をファイルへバックアップすることができ、**WITH PRIVATE KEY** オプションを付けることで、秘密キーもバックアップできます。このときに **ENCRYPTION BY PASSWORD** で指定するパスワードには、推測されにくい強固なパスワードを設定して、そのパスワードが簡単に漏れないように注意しておく必要があります。

バックアップしたファイル（サーバー証明書と秘密キー）は、別マシンに物理的にコピーして、次のように **CREATE CERTIFICATE** ステートメントを利用して、リストア（インポート）すること

ができます。

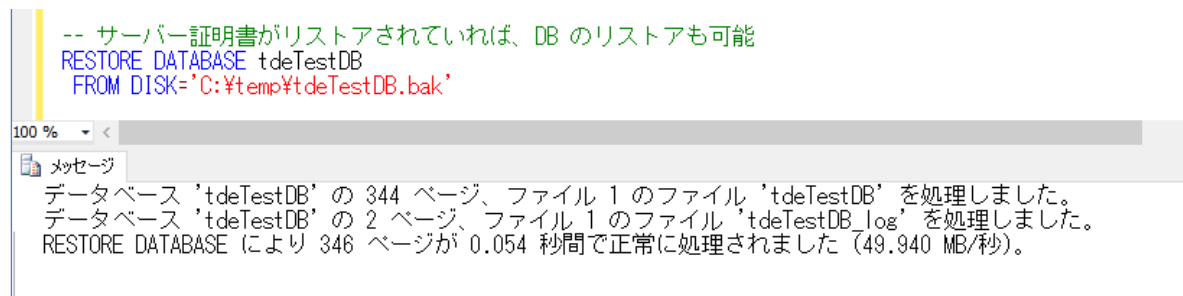
```
-- データベース マスター キーの作成
USE master
CREATE MASTER KEY
    ENCRYPTION BY PASSWORD = '<StrongPassword>'
go
-- サーバー証明書のリストア（インポート）
CREATE CERTIFICATE MyServerCert
    FROM FILE = 'C:¥temp¥MyServerCert'
    WITH PRIVATE KEY
        ( FILE = 'C:¥temp¥MyPrivKey'
          , DECRYPTION BY PASSWORD = 'P@ssword' )
```



サーバー証明書のリストア（ファイルからの作成）には、**データベース マスター キー**を作成しておく必要があるため、まずデータベース マスター キーを作成しています。

サーバー証明書のリストア時（**CREATE CERTIFICATE** ステートメントの実行時）には、**DECRYPTION BY PASSWORD** に、複合化のためのパスワードを、バックアップ時に指定したのと同じものを指定します。これで、サーバー証明書のリストアが完了です。

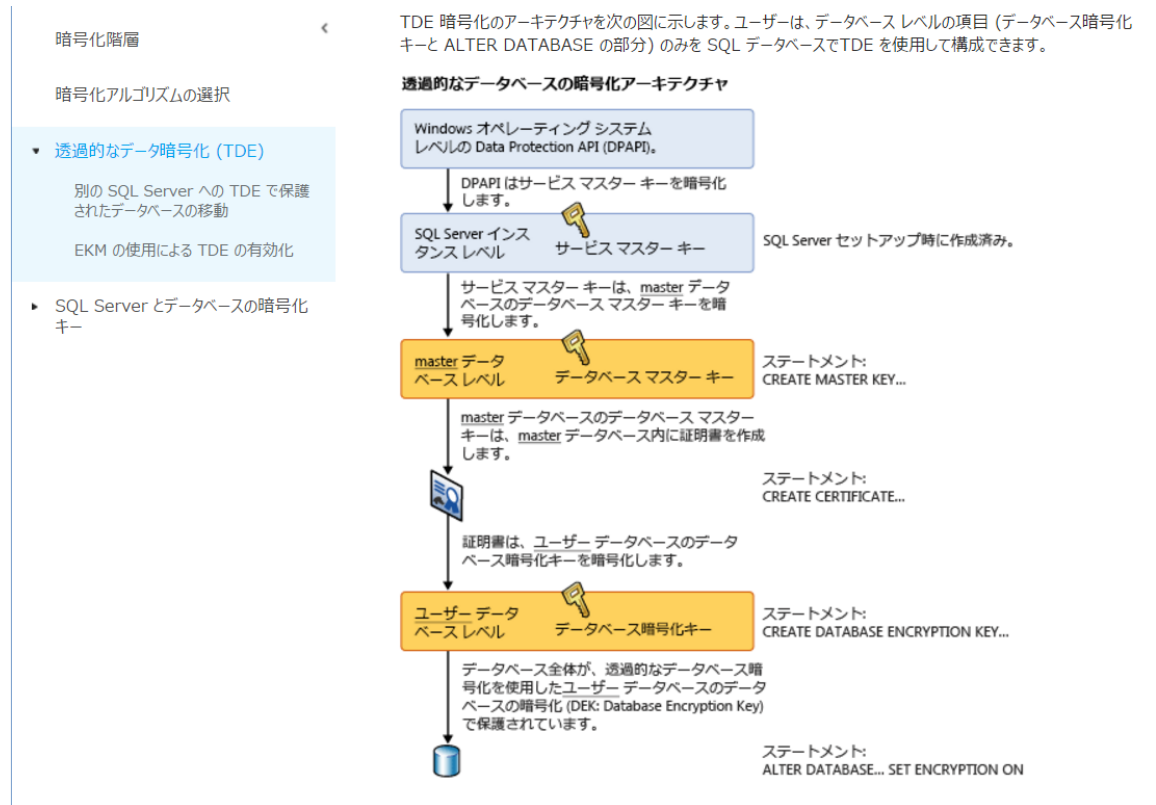
このように同じサーバー証明書がリストアされている環境であれば、透過的なデータ暗号化（TDE）を設定したデータベースをリストアして、データを参照することができるようになります。



その他、TDE の詳細（アーキテクチャなど）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。

### 透過的なデータ暗号化（TDE）

<http://msdn.microsoft.com/ja-jp/library/bb934049.aspx>



### 3.6 テンポラル テーブルによる Audit (監査証跡)

**テンポラル テーブル** (Temporal Tables) は、テーブルに対する過去の更新履歴を自動的に履歴テーブルに保存できる機能です。これを利用すれば、次のような用途で利用することができます。

- タイム トラベル
- **行レベルでのデータ修復** (操作ミスなどへの対応)
- **Audit** (コンプライアンスにおける監査証跡)
- **SCD** (Slowly Changing Dimension : 緩やかに変化するディメンション)

テンポラル テーブルを利用すれば、タイム マシンのように、過去のデータに簡単にアクセスすることができるので、**操作ミスが発生した場合のデータ修復**や、**コンプライアンスにおける監査証跡** (Audit Log : 監査ログ)、**データ分析における SCD** (緩やかに変化するディメンション) として利用することができます。

テンポラル テーブルを Audit 目的で利用すれば、**J-SOX 法** (日本版 SOX 法) や**内部統制**、**PCI DSS** (クレジット カード情報のセキュリティ基準) などの**コンプライアンス (法令遵守)** を実現するために利用できます (なお、これらを実現するために、従来からある **SQL Server Audit** 機能を利用することも、SQL Server 2016 ではもちろん可能です。違いについては以下の Note 参照)。

#### Note : CDC や Change Tracking との違いは「使いやすさ」、SQL Server Audit との違い

これまでの SQL Server には、**CDC** (Change Data Capture : 変更データ キャプチャ) や **Change Tracking** (変更追跡) という、テンポラル テーブルと似たような機能 (いずれも更新履歴を自動保存できる機能) がありました。実際、CDC や Change Tracking を利用して、タイム トラベルや行レベルのデータ修復、Audit、SCD を実現している場合もあると思います。今回提供された「**テンポラル テーブル**」と **CDC/Change Tracking** との大きな違いは、今回のテンポラル テーブルは、**CDC/Change Tracking よりも使いやすい** (履歴テーブルを操作しやすい) という点です。**CDC/Change Tracking** では、タイム トラベルがしづらかったり、行レベルでのデータ修復時に、該当データを探すのが大変だったりしましたが、今回のテンポラル テーブルではそういったことはありません。テンポラル テーブルを利用すれば、簡単に過去のデータを参照することができます。

テンポラル テーブルと **SQL Server Audit** 機能との比較は、SQL Server Audit 機能が「**監査**」目的に特化した機能で、非常に細かく監査イベントを設定できる (データの更新履歴だけでなく、ログインの成功/失敗や、アカウントのパスワード変更、データベースそのものに対する変更など、いろいろな種類のイベントをきめ細かく設定できる) ことが大きな違いです。テンポラル テーブルで行える Audit は、テンポラル テーブルを設定したテーブルに対して、過去の更新履歴を記録/追跡できるようになることで、ログインの成功/失敗などを追跡する目的では利用できません。テンポラル テーブルを利用するメリットは、Audit 目的で利用できるだけでなく、その他の用途 (操作ミス時のデータ修復や SCD シナリオなど) でも利用できる点です。

#### ➡ テンポラル テーブルの作成方法

テンポラル テーブルは、次のように作成することができます。

```
-- テンポラル テーブルの作成例
CREATE TABLE Products
( 商品コード int PRIMARY KEY, --PRIMARY KEY は必須
```



```

商品名      nvarchar(40),
sysstart    datetime2 GENERATED ALWAYS AS ROW START HIDDEN NOT NULL,  --開始時刻用
sysend      datetime2 GENERATED ALWAYS AS ROW END HIDDEN NOT NULL,      --終了時刻用
            PERIOD FOR SYSTEM_TIME ( sysstart, sysend )
)
WITH ( SYSTEM_VERSIONING = ON
      ( HISTORY_TABLE = dbo.ProductsHistory ) )  --更新履歴を格納するテーブルの指定

```

テンポラル テーブルでは、**PRIMARY KEY 制約 (主キー制約)** が必須になるので、PRIMARY KEY 制約を設定した列を設けるようにします。また、テンポラル テーブルでは、履歴／更新日時を管理するための列が 2 つ必要になります (上の例では **sysstart** と **sysend** という名前で作成した列で、任意の名前を付けることが可能)。この 2 つの列には、「**GENERATED ALWAYS AS**」で「**ROW START**」および「**ROW END**」を指定するようにして、「**PERIOD FOR SYSTEM\_TIME**」で列の名前を指定するようにします。また、**HIDDEN** 属性を付けることで、列データを隠すことができるようになります (**SELECT \*** で検索しても、表示されない列に設定できるようになります)。

そして **WITH** 句で「**SYSTEM\_VERSIONING = ON**」を指定することで、テンポラル テーブルを有効化して、「**HISTORY\_TABLE =**」で更新履歴を格納するテーブルの名前を指定すれば、テンポラル テーブルの作成が完了です。

#### 既存のテーブルをテンポラル テーブルに変更する場合 ～ALTER TABLE～

テンポラル テーブルを既存のテーブルに対して設定したい場合には、次のように **ALTER TABLE** ステートメントを実行します。

```

-- 日時を格納するための列を 2つ追加 (データがある場合には DEFAULT 制約が必須)
ALTER TABLE Products
ADD sysstart datetime2 GENERATED ALWAYS AS ROW START HIDDEN
    DEFAULT CONVERT(datetime2, GETUTCDATE()), -- DEFAULT 制約
    sysend    datetime2 GENERATED ALWAYS AS ROW END HIDDEN
    DEFAULT CONVERT(datetime2, '9999-12-31 23:59:59.9999999'), -- DEFAULT 制約
    PERIOD FOR SYSTEM_TIME ( sysstart, sysend )

-- テンポラリ テーブルを有効化と履歴テーブルの指定
ALTER TABLE Products
SET ( SYSTEM_VERSIONING = ON
      ( HISTORY_TABLE = dbo.ProductsHistory ) )

```

既存のテーブルのデータが空(データ件数が 0)の場合には、**DEFAULT 制約**は必要ありませんが、データが存在している場合には、更新日時を管理するための列に対して **DEFAULT 制約**を設定する必要があります。上の例では、開始日時 (**sysstart** 列) は **GETUTCDATE** 関数で、現在時刻を UTC (協定世界時) で指定していますが、「**2016-01-01**」のように特定の開始日時を指定することも可能です。終了時刻 (**sysend** 列) は、**datetime2** データ型の最大値である「**9999-12-31 23:59:59.9999999**」を指定しておくようにします。

2 つの列 (**sysstart**、**sysend**) を追加した後は、**ALTER TABLE** ステートメントで「**SYSTEM\_VERSIONING = ON**」を指定することで、テンポラル テーブルを有効化できます。

## ➡ Let's Try

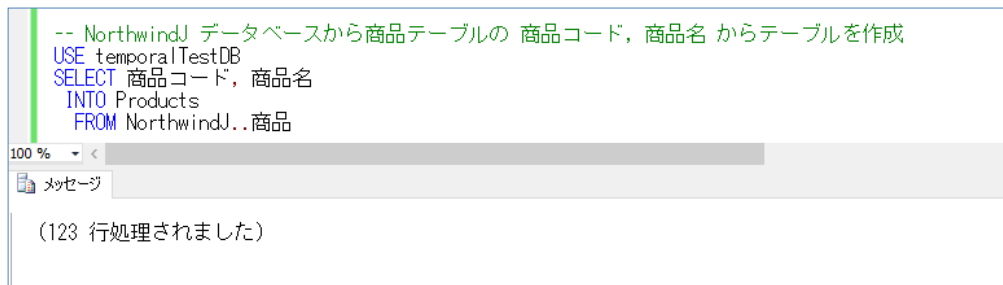
それでは、これを試してみましょう。

1. まずは、テンポラル テーブルを試すためのデータベースを「**temporalTestDB**」という名前で作成します。

```
-- データベースの作成
CREATE DATABASE temporalTestDB
```

2. 次に、「**NorthwindJ**」データベース内から「**商品**」テーブルから「**商品コード**」と「**商品名**」を抜き出して、テーブルを作成します（なお、**NorthwindJ** データベースの作成方法は、巻末の付録に記載しています）。

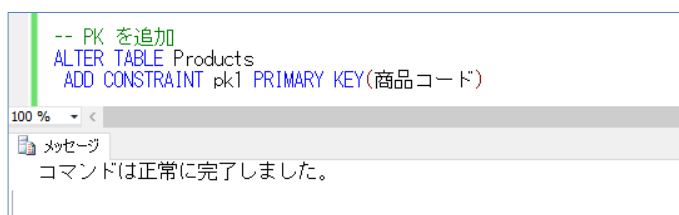
```
USE temporalTestDB
SELECT 商品コード, 商品名
INTO Products
FROM NorthwindJ..商品
```



作成するテーブル名は「**Products**」にしていますが、任意の名前でもかまいません。

3. 次に、作成した **Products** テーブルを**テンポラル テーブル**に変更していきます。テンポラル テーブルの作成には、**PRIMARY KEY 制約（主キー制約）が必須**になるので、次のように **ALTER TABLE .. ADD CONSTRAINT** ステートメントを実行して、「**商品コード**」列に **PRIMARY KEY 制約**を設定します。

```
-- PK を追加
ALTER TABLE Products
ADD CONSTRAINT pk1 PRIMARY KEY(商品コード)
```



4. 次に、**ALTER TABLE** ステートメントを利用して、テンポラル テーブルで必要になる 2 つの列（日付を格納するための列）を追加します。

```
ALTER TABLE Products
ADD sysstart datetime2 GENERATED ALWAYS AS ROW START
    DEFAULT CONVERT(datetime2, '2016-01-01'),
sysend datetime2 GENERATED ALWAYS AS ROW END
    DEFAULT CONVERT(datetime2, '9999-12-31 23:59:59.9999999'),
PERIOD FOR SYSTEM_TIME ( sysstart, sysend )
```

列の名前は「**sysstart**」と「**sysend**」にしていますが、任意の列名を付けることができます。

「**sysstart**」列には「**GENERATED ALWAYS AS**」で「**ROW START**」、「**sysend**」列には「**ROW END**」を指定して、「**PERIOD FOR SYSTEM\_TIME (sysstart, sysend)**」を記述することで、テンポラル テーブル用の列（更新日時／履歴を制御するための日付列）にすることができます。また、「**sysstart**」には **DEFAULT 制約**で「**2016-01-01**」を指定していますが、前述したように **GETUTCDATE** 関数を利用して現在時刻を指定してもかまいません。

### HIDDEN 付きがお勧め

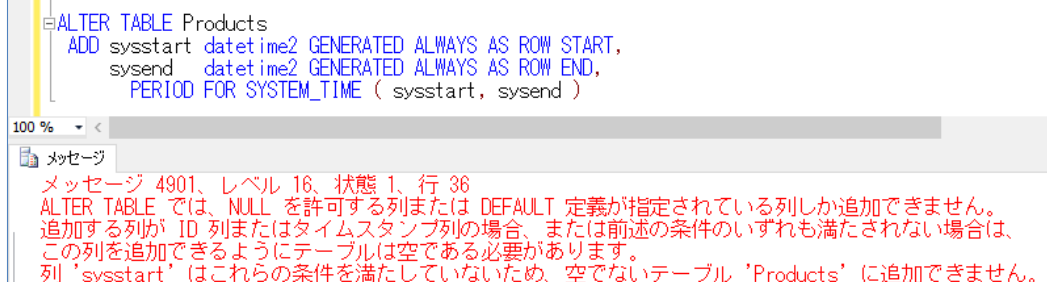
上の手順では、「**sysstart**」と「**sysend**」列に **HIDDEN 属性**をあえて付けませんでした。が、実際の運用では、次のように **HIDDEN 属性**を付けるのがお勧めです（**HIDDEN 属性**を付ければ、**SELECT \*** で検索しても、表示されない列に設定することができます）。

```
ALTER TABLE Products
ADD sysstart datetime2 GENERATED ALWAYS AS ROW START HIDDEN
    DEFAULT CONVERT(datetime2, '2016-01-01'),
sysend datetime2 GENERATED ALWAYS AS ROW END HIDDEN
    DEFAULT CONVERT(datetime2, '9999-12-31 23:59:59.9999999'),
PERIOD FOR SYSTEM_TIME ( sysstart, sysend )
```

テンポラル テーブルの内部動作を理解するには、**HIDDEN 属性**を外しておいた方が分かりやすいので、以降の手順では、**HIDDEN 属性**を外した状態で説明していきます。

#### Note : DEFAULT 制約を設定していない場合

2つの列（**sysstart**、**sysend**）を追加する際に、**DEFAULT 制約**を指定しないで **ALTER TABLE** ステートメントを実行した場合は、次のようにエラーになります（データが存在している場合はエラーになります）

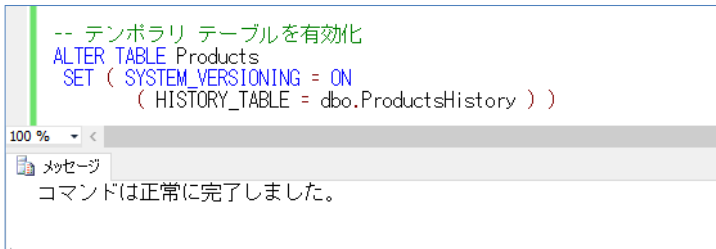


```
ALTER TABLE Products
ADD sysstart datetime2 GENERATED ALWAYS AS ROW START,
sysend datetime2 GENERATED ALWAYS AS ROW END,
PERIOD FOR SYSTEM_TIME ( sysstart, sysend )
```

メッセージ 4901、レベル 16、状態 1、行 36  
 ALTER TABLE では、NULL を許可する列または DEFAULT 定義が指定されている列しか追加できません。  
 追加する列が ID 列またはタイムスタンプ列の場合、または前述の条件のいずれも満たされない場合は、  
 この列を追加できるようにテーブルは空である必要があります。  
 列 'sysstart' はこれらの条件を満たしていないため、空でないテーブル 'Products' に追加できません。

- 次に、テンポラル テーブルとして有効化するために、次のように **ALTER TABLE** ステートメントを実行します。

```
ALTER TABLE Products
SET ( SYSTEM_VERSIONING = ON
      ( HISTORY_TABLE = dbo.ProductsHistory ) )
```

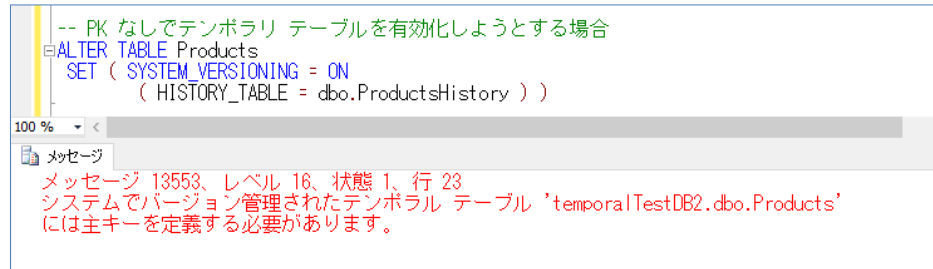


「**SYSTEM\_VERSIONING = ON**」を指定することで、テンポラル テーブルとして有効化することができ、「**HISTORY\_TABLE = dbo.ProductsHistory**」と指定することで、過去のデータを保存しておく履歴テーブルの名前を「**ProductsHistory**」に設定しています（任意の名前に設定することができます）。

以上で、テンポラル テーブルの設定が完了です。

#### Note : PRIMARY KEY 制約を設定していない場合

テーブルに PRIMARY KEY 制約を設定していない場合には、次のようにエラーが表示されます。



## ➡ データを更新してみよう

次に、**Products** テーブルのデータを更新して、更新履歴が「**ProductsHistory**」テーブルに格納されていくことを確認してみましょう。

1. まずは、データを更新する前のデータを確認しておきます。

```
SELECT * FROM Products
```

The screenshot shows a SQL query window with the command: `-- データの確認`  
`SELECT * FROM Products`  
 Below the query window, the Results pane displays a table with 5 rows and 5 columns: `商品コード`, `商品名`, `sysstart`, and `sysend`.

	商品コード	商品名	sysstart	sysend
1	1	果汁100% オレンジ	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999
2	2	果汁100% グレープ	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999
3	3	果汁100% レモン	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999
4	4	果汁100% ピーチ	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999
5	5	コーヒーマイルド	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999

**sysstart** と **sysend** 列には、**DEFAULT 制約**で指定した値が格納されていることを確認できます。

なお、**sysstart** と **sysend** 列に **HIDDEN 属性**を付けている場合には、**SELECT** ステートメントでデータを検索しても、次のように、これらの列が表示されない形になります。

```

ALTER TABLE Products
ADD sysstart datetime2 GENERATED ALWAYS AS ROW START HIDDEN
    DEFAULT CONVERT(datetime2, '2016-01-01'),
    sysend datetime2 GENERATED ALWAYS AS ROW END HIDDEN
    DEFAULT CONVERT(datetime2, '9999-12-31 23:59:59.9999999'),
    PERIOD FOR SYSTEM_TIME ( sysstart, sysend )

ALTER TABLE Products
SET ( SYSTEM_VERSIONING = ON
    ( HISTORY_TABLE = dbo.ProductsHistory ) )

SELECT * FROM Products
  
```

	商品コード	商品名
1	1	果汁100% オレンジ
2	2	果汁100% グレープ
3	3	果汁100% レモン
4	4	果汁100% ピーチ
5	5	コーヒーマイルド
6	6	コーヒーピター

2. 次に、**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 Products SET 商品名 = 'AAAA' WHERE 商品コード = 1
UPDATE Products SET 商品名 = 'BBB' WHERE 商品コード = 2
  
```

商品コードが「1」の商品名（果汁 100%オレンジ）を「AAAA」、商品コードが「2」の商品名（果汁 100%グレープ）を「BBB」に変更します。

3. データの更新が完了したら、**SELECT** ステートメントを実行して、データを確認します。

```
SELECT * FROM Products
```

```

-- データの確認
SELECT * FROM Products
  
```

	商品コード	商品名	sysstart	sysend
1	1	AAAA	2016-01-11 04:18:32.2304961	9999-12-31 23:59:59.9999999
2	2	BBB	2016-01-11 04:18:32.2313230	9999-12-31 23:59:59.9999999
3	3	果汁100% レモン	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999
4	4	果汁100% ピーチ	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999
5	5	コーヒーマイルド	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999
6	6	コーヒーピター	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999

**sysstart** 列の値が、データを更新した日時に変更されていることを確認できます。なお、日時は **UTC**（協定世界時）で記録されるので、日本の場合は、+9 時間（上のように **1 月 11 日 04 時 18 分** なら **1 月 11 日 13 時 18 分**）した値になります。

4. 次に、もう一度 **UPDATE** ステートメントを実行して、さらに異なる値にデータを更新してみ

ます。

```
UPDATE Products SET 商品名 = 'AAAA_2nd' WHERE 商品コード = 1
UPDATE Products SET 商品名 = 'BBB_2nd' WHERE 商品コード = 2
```

商品コードが「1」の商品名を「AAAA\_2nd」、商品コードが「2」の商品名を「BBB\_2nd」に変更します。

5. データの更新後、**SELECT** ステートメントを実行して、データを確認します。

```
SELECT * FROM Products
```

-- データの確認 (現在のデータ)  
SELECT \* FROM Products

	商品コード	商品名	sysstart	sysend
1	1	AAAA_2nd	2016-01-11 04:23:27.6951657	9999-12-31 23:59:59.9999999
2	2	BBB_2nd	2016-01-11 04:23:27.6971647	9999-12-31 23:59:59.9999999
3	3	果汁100% レモン	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999
4	4	果汁100% ピーチ	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999
5	5	コーヒーマイルド	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999
6	6	コーヒービター	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999

**sysstart** 列の値が、データを更新した日時に変更されていることを確認できます。

## 履歴テーブル確認

次に、更新履歴が格納されている履歴テーブル (**ProductsHistory**) を確認してみましょう。

1. まずは、**SELECT** ステートメントを実行して、中身を参照してみます。

```
SELECT * FROM ProductsHistory
```

-- 履歴データの確認 (更新履歴)  
SELECT \* FROM ProductsHistory

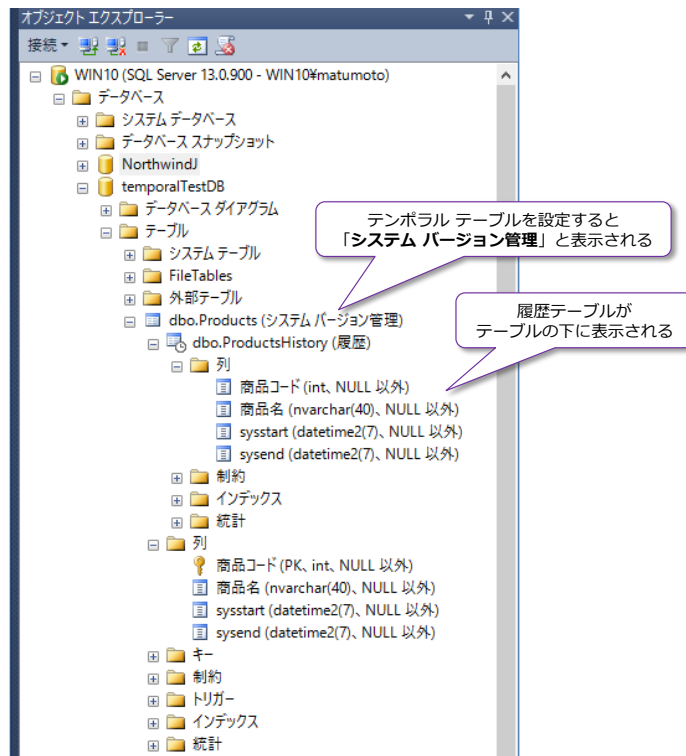
	商品コード	商品名	sysstart	sysend
1	1	果汁100% オレンジ	2016-01-01 00:00:00.0000000	2016-01-11 04:18:32.2304961
2	2	果汁100% グレープ	2016-01-01 00:00:00.0000000	2016-01-11 04:18:32.2313230
3	1	AAAA	2016-01-11 04:18:32.2304961	2016-01-11 04:23:27.6951657
4	2	BBB	2016-01-11 04:18:32.2313230	2016-01-11 04:23:27.6971647

複数のデータ更新がある場合には  
**sysend** と、次の **sysstart** が同じ値になる

更新前の過去のデータ (商品コード 1 は、果汁 100%オレンジと AAAA、商品コード 2 は、果汁 100%グレープと BBB) が表示されて、**sysstart** (開始日時) と **sysend** (終了日時) で有効期間 (いつからいつまでの値だったのか) が記録されていることを確認できます。また、複数のデータ更新がある場合には、**sysend** と、次の **sysstart** が同じ値 (終了日時が、次の

開始日時) になります。

- 次に、オブジェクト エクスプローラーで、履歴テーブルを確認してみましょう。



テンポラル テーブルを設定すると、テーブル名 (**Products**) の隣に「**システム バージョン管理**」と表示されて、テーブルを展開すると、「**(履歴)**」と表示される履歴テーブル (**ProductsHistory**) を確認することができます。

このように、テンポラル テーブルを設定すると、自動的に履歴テーブルが作成されて、このテーブルに更新履歴が格納されていくようになります。

## ➡ 過去のデータへのアクセス ～FOR SYSTEM\_TIME AS OF～

テンポラル テーブルの一番のメリットは、過去のデータ (履歴) に簡単にアクセスできる点にあります (従来の CDC や Change Tracking 機能よりも簡単に過去データを参照することができます)。これも試してみましょう。

- まずは、過去の任意の日時を指定して、データにアクセスしてみましょう。これを行うには、**SELECT** ステートメントで、次のように、テーブル名の後に「**FOR SYSTEM\_TIME AS OF** '戻りたい日時'」を付けます。

```
SELECT * FROM Products
FOR SYSTEM_TIME AS OF '2016-01-01'
ORDER BY 商品コード
```



-- 更新前のデータ（過去の指定した日時のデータ）を確認  
 SELECT \* FROM Products  
 FOR SYSTEM\_TIME AS OF '2016-01-01'  
 ORDER BY 商品コード

	商品コード	商品名	sysstart	sysend
1	1	果汁100% オレンジ	2016-01-01 00:00:00.0000000	2016-01-11 04:18:32.2304961
2	2	果汁100% グレープ	2016-01-01 00:00:00.0000000	2016-01-11 04:18:32.2313230
3	3	果汁100% レモン	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999
4	4	果汁100% ピーチ	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999
5	5	コーヒーマイルド	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999
6	6	コーヒーピター	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999

日時には、「**2016-01-01**」を指定していますが、この値は、**DEFAULT 制約**で **sysstart** 列に設定した日付で、この時点（最初の時点）でのデータを参照することができます。したがって、商品コード **1** には **果汁 100%オレンジ**、商品コード **2** には、**果汁 100%グレープ**（いずれも更新する前のデータ）が表示されています。

なお、「**FOR SYSTEM\_TIME AS OF**」は、**WHERE** 句を利用する場合には、次のように **WHERE** 句の前に記述するようにします。

-- WHERE を付ける場合  
 SELECT \* FROM Products  
 FOR SYSTEM\_TIME AS OF '2016-01-01'  
 WHERE 商品コード = 1  
 ORDER BY 商品コード

FOR SYSTEM\_TIME は WHERE の前に記述する

	商品コード	商品名	sysstart	sysend
1	1	果汁100% オレンジ	2016-01-01 00:00:00.0000000	2016-01-11 04:18:32.2304961

2. 次に、「**FOR SYSTEM\_TIME AS OF**」で指定する日時に、**2015 年の日付（2015-12-01** など）を指定してデータを参照してみます。

```
SELECT * FROM Products
FOR SYSTEM_TIME AS OF '2015-12-01'
ORDER BY 商品コード
```

-- 2015年の日付を指定  
 SELECT \* FROM Products  
 FOR SYSTEM\_TIME AS OF '2015-12-01'  
 ORDER BY 商品コード

	商品コード	商品名	sysstart	sysend
--	-------	-----	----------	--------

結果は、何もデータが返ってきません。**sysstart** 列の **DEFAULT 制約**で「**2016-01-01**」を指定しているので、初期データは「**2016-01-01**」時点以降のデータという扱いになり、それ以前のデータは存在しない、ということでこのような動作になります。

3. 次に、1 つ目の更新データ（商品コード **1** の **AAAA**）を参照するために、**ProducttsHistory** テーブルの **sysstart** 列の値を確認してみます。

```
SELECT * FROM ProductsHistory
```

-- 更新日時を確認  
SELECT \* FROM ProductsHistory

	商品コード	商品名	sysstart	sysend
1	1	果汁100% オレンジ	2016-01-01 00:00:00.0000000	2016-01-11 04:18:32.2304961
2	2	果汁100% グレープ	2016-01-01 00:00:00.0000000	2016-01-11 04:18:32.2313230
3	1	AAAA	2016-01-11 04:18:32.2304961	2016-01-11 04:23:27.6951657
4	2	BBB	2016-01-11 04:18:32.2313230	2016-01-11 04:23:27.6971647

**sysstart** (開始日時) と **sysend** (終了日時) の間が、そのデータの**有効期間** (いつからいつまでの値だったのか) になります。

4. 上で調べた「AAAA」データの有効期間内の値を指定して、データを参照してみましょう。

```
SELECT * FROM Products
FOR SYSTEM_TIME AS OF '2016-01-11 04:19:00'
ORDER BY 商品コード
```

-- AAAA の範囲内の日時を指定  
SELECT \* FROM Products  
FOR SYSTEM\_TIME AS OF '2016-01-11 04:19:00'  
ORDER BY 商品コード

	商品コード	商品名	sysstart	sysend
1	1	AAAA	2016-01-11 04:18:32.2304961	2016-01-11 04:23:27.6951657
2	2	BBB	2016-01-11 04:18:32.2313230	2016-01-11 04:23:27.6971647
3	3	果汁100% レモン	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999
4	4	果汁100% ピーチ	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999

画面では、**sysstart** は「2016-01-11 04:18:32.2304961」、**sysend** は「2016-01-11 04:23:27.6951657」なので、その間の「2016-01-11 04:19:00」を指定することで、「AAAA」データを参照できています。

5. 次に、**sysend** (2016-01-11 04:23:27.6951657) よりも大きい値を指定して、データを参照してみます。

```
SELECT * FROM Products
FOR SYSTEM_TIME AS OF '2016-01-11 04:25:00'
ORDER BY 商品コード
```

-- AAAA の sysend より大きい時刻を指定  
SELECT \* FROM Products  
FOR SYSTEM\_TIME AS OF '2016-01-11 04:25:00'  
ORDER BY 商品コード

	商品コード	商品名	sysstart	sysend
1	1	AAAA_2nd	2016-01-11 04:23:27.6951657	9999-12-31 23:59:59.9999999
2	2	BBB_2nd	2016-01-11 04:23:27.6971647	9999-12-31 23:59:59.9999999
3	3	果汁100% レモン	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999

今度は、最後に更新した値である「AAAA\_2nd」を参照できることを確認できます。

このように、「**FOR SYSTEM\_TIME AS OF**」を利用することで、ある特定の時点でのデータを簡単に参照することができます。なお、前述したように、更新日時は **UTC**（協定世界時）で記録されているので、日本の場合は **+9 時間**を加算した日時を指定することで、任意の時点のデータを参照できるようになります。

## ➡ その他の演算子

テンポラル テーブルでは、「**FOR SYSTEM\_TIME**」の「**AS OF**」の他に、「**ALL**」や「**BETWEEN .. AND**」、「**FROM .. TO**」、「**CONTAINED IN(..)**」という演算子を利用して、過去のデータを参照することもできます。これも試してみましょう。

1. まずは、**ALL** を利用してみましょう。

```
SELECT * FROM Products
FOR SYSTEM_TIME ALL
ORDER BY 商品コード
```

```
-- ALL
SELECT * FROM Products
FOR SYSTEM_TIME ALL
ORDER BY 商品コード, sysstart
```

	商品コード	商品名	sysstart	sysend
1	1	果汁100% オレンジ	2016-01-01 00:00:00.0000000	2016-01-11 04:18:32.2304961
2	1	AAAA	2016-01-11 04:18:32.2304961	2016-01-11 04:23:27.6951657
3	1	AAAA_2nd	2016-01-11 04:23:27.6951657	9999-12-31 23:59:59.9999999
4	2	果汁100% グレープ	2016-01-01 00:00:00.0000000	2016-01-11 04:18:32.2313230
5	2	BBB	2016-01-11 04:18:32.2313230	2016-01-11 04:23:27.6971647
6	2	BBB_2nd	2016-01-11 04:23:27.6971647	9999-12-31 23:59:59.9999999
7	3	果汁100% レモン	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999
8	4	果汁100% ピーチ	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999

このように、**ALL** を指定した場合は、過去の更新データをすべて表示することができます。

2. 次に、**BETWEEN .. AND** を利用してみましょう。

```
SELECT * FROM Products
FOR SYSTEM_TIME BETWEEN '2016-01-02' AND '2016-01-13'
ORDER BY 商品コード, sysstart
```

```
-- BETWEEN .. AND
SELECT * FROM Products
FOR SYSTEM_TIME BETWEEN '2016-01-02' AND '2016-01-13'
ORDER BY 商品コード, sysstart
```

	商品コード	商品名	sysstart	sysend
1	1	果汁100% オレンジ	2016-01-01 00:00:00.0000000	2016-01-11 04:18:32.2304961
2	1	AAAA	2016-01-11 04:18:32.2304961	2016-01-11 04:23:27.6951657
3	1	AAAA_2nd	2016-01-11 04:23:27.6951657	9999-12-31 23:59:59.9999999
4	2	果汁100% グレープ	2016-01-01 00:00:00.0000000	2016-01-11 04:18:32.2313230
5	2	BBB	2016-01-11 04:18:32.2313230	2016-01-11 04:23:27.6971647
6	2	BBB_2nd	2016-01-11 04:23:27.6971647	9999-12-31 23:59:59.9999999
7	3	果汁100% レモン	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999

**BETWEEN** では、**AND** で複数の日時を指定して、その間のデータをすべて表示することができます。

3. 次に、**BETWEEN** の **AND** で指定する日時を、「AAAA」データの **sysend** と同じ日時にして、データを参照してみます。

```
SELECT * FROM Products
FOR SYSTEM_TIME BETWEEN '2015-10-21 14:24' AND '2016-01-11 04:23:27.6951657'
ORDER BY 商品コード
```

-- BETWEEN .. AND で AAAA の sysend を指定

```
SELECT * FROM Products
FOR SYSTEM_TIME BETWEEN '2016-01-02' AND '2016-01-11 04:23:27.6951657'
ORDER BY 商品コード, sysstart
```

	商品コード	商品名	sysstart	sysend
1	1	果汁100% オレンジ	2016-01-01 00:00:00.0000000	2016-01-11 04:18:32.2304961
2	1	AAAA	2016-01-11 04:18:32.2304961	2016-01-11 04:23:27.6951657
3	1	AAAA_2nd	2016-01-11 04:23:27.6951657	9999-12-31 23:59:59.9999999
4	2	果汁100% グレープ	2016-01-01 00:00:00.0000000	2016-01-11 04:18:32.2313230
5	2	BBB	2016-01-11 04:18:32.2313230	2016-01-11 04:23:27.6971647
6	3	果汁100% レモン	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999
7	4	果汁100% ピーチ	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999

「AAAA」データの **sysend** は、「AAAA\_2nd」の **sysstart** 値でもあるので、両方のデータを参照できています。

4. 次に、**BETWEEN** を **FROM**、**AND** を **TO** に変更して、同じように参照してみます。

```
SELECT * FROM Products
FOR SYSTEM_TIME FROM '2015-10-21 14:24' TO '2016-01-11 04:23:27.6951657'
ORDER BY 商品コード
```

-- BETWEEN を FROM、AND を TO に変更して検索

```
SELECT * FROM Products
FOR SYSTEM_TIME FROM '2015-10-21 14:24' TO '2016-01-11 04:23:27.6951657'
ORDER BY 商品コード
```

	商品コード	商品名	sysstart	sysend
1	1	果汁100% オレンジ	2016-01-01 00:00:00.0000000	2016-01-11 04:18:32.2304961
2	1	AAAA	2016-01-11 04:18:32.2304961	2016-01-11 04:23:27.6951657
3	2	BBB	2016-01-11 04:18:32.2313230	2016-01-11 04:23:27.6971647
4	2	果汁100% グレープ	2016-01-11 04:18:32.2313230	2016-01-11 04:18:32.2313230
5	3	果汁100% レモン	2016-01-11 04:18:32.2313230	9999-12-31 23:59:59.9999999
6	4	果汁100% ピーチ	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999
7	5	コーヒーマイルド	2016-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999

FROM TO では  
AAAA\_2nd はヒットしない

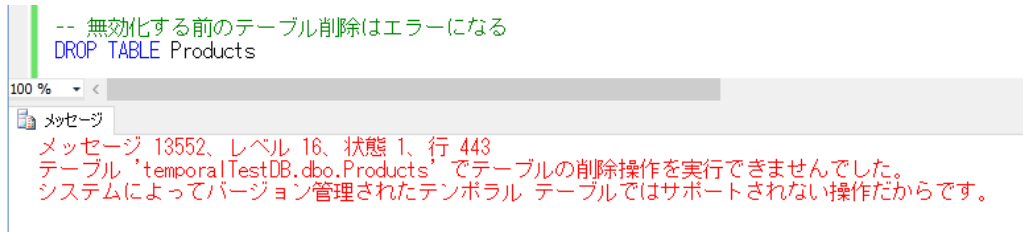
今度は、「AAAA」は参照できるものの、「AAAA\_2nd」はヒットしないことが分かります。このように、「**BETWEEN date1 AND date2**」と「**FROM date1 TO date2**」の違いは、「**sysstart <= date2**」になるか、「**sysstart < date2**」になるか（= を含むかどうか）です。**BETWEEN** の場合は **= を含む**、**FROM** の場合は **= を含みません**。

このように、テンポラル テーブルでは、過去の履歴データに対して、いろいろなデータ参照方

法が用意されているので、過去のデータを非常に扱いやすくなっています。

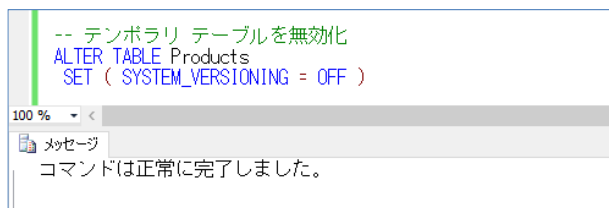
## ➡ テンポラル テーブルを無効化したい場合（通常テーブルに戻したい場合）

1. テンポラル テーブルを設定したテーブルは、**DROP TABLE** ステートメントで削除しようとしても、次のようにエラーが表示されます。

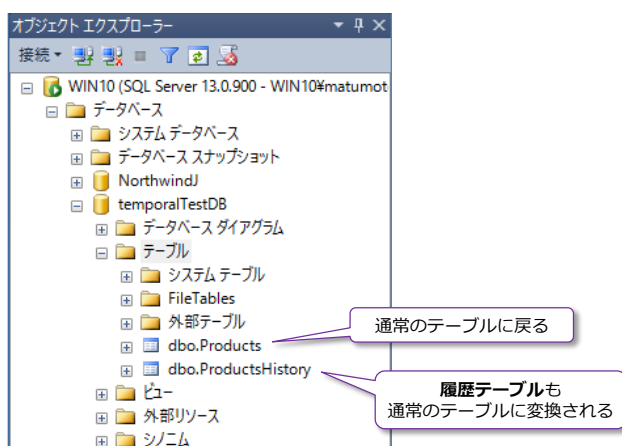


2. テンポラル テーブルを無効化したい場合には、次のように **ALTER TABLE** ステートメントで「**SYSTEM\_VERSIONING = OFF**」に設定します。

```
ALTER TABLE Products
SET ( SYSTEM_VERSIONING = OFF )
```



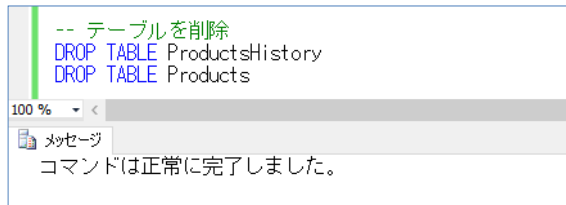
テンポラル テーブルを無効化すると、オブジェクト エクスプローラーでは、次のように表示されます。



テンポラル テーブルの場合は「システム バージョン管理」と表示されていましたが、通常のテーブルに戻って、履歴テーブルに関しても、通常のテーブルに変換される（履歴テーブルも残る）形になります。

3. このように通常テーブルに戻った後は、通常と同じように **DROP TABLE** ステートメントで削除することができるので、不要な場合は削除しておいてください。

```
DROP TABLE ProductsHistory
DROP TABLE Products
```



以上のように、**テンポラル テーブル**を利用すれば、テーブルに対する更新履歴を自動的に履歴テーブルに保存、および履歴データを簡単に参照することができるので、**操作ミスが発生した場合のデータ修復**や、**コンプライアンスにおける監査証跡** (Audit Log: 監査ログ)、**データ分析における SCD** (緩やかに変化するディメンション) として利用することができます。

その他、テンポラル テーブルについては、オンライン ブックの以下のトピックがお勧めです。

## Temporal Tables

<https://msdn.microsoft.com/en-us/library/dn935015.aspx>

...

- ▶ Primary and Foreign Key Constraints
- ▶ Unique Constraints and Check Constraints
- Table Column Properties (SQL Server Management Studio)
- ▼ **Temporal Tables**
  - ▶ Getting Started with System-Versioned Temporal Tables
    - Temporal Table System Consistency Checks
    - Partitioning with Temporal Tables
    - Temporal Table Considerations and Limitations
    - Temporal Table Security
  - ▶ System-Versioned Temporal Tables with Memory-Optimized Tables
    - Temporal Table Metadata Views and Functions

# Temporal Tables

SQL Server 2016

Updated: January 6, 2016

Applies To: SQL Server (starting with 2016 CTP3)

**Applies to:** SQL Server (SQL Server 2016 Community Technology Preview 3.2 (CTP 3.2) through [current version](#)).

SQL Server 2016 introduces support for system-versioned temporal tables as a database feature that brings built-in support for providing information about data stored in the table at any point in time rather than only the data that is correct at the current moment in time. Temporal is a database feature that was introduced in ANSI SQL 2011 and is now supported in SQL Server 2016.

**Quick Start**

- **Download CTP:** To download SQL Server 2016 Community Technology Preview 3.2 (CTP 3.2), go to [Evaluation Center](#).
- **Create Azure Virtual Machine:** Have an Azure account? [Spin up a virtual machine](#) with SQL Server 2016 already installed.
- **Download AdventureWorks sample database:** To get started with Temporal Tables download [AdventureWorks database for SQL Server 2016 CTP3](#) with script samples and follow the instructions in the folder 'Temporal'
- **Getting started:**
  - [Getting Started with System-Versioned Temporal Tables](#)
  - [System-Versioned Temporal Tables with Memory-Optimized Tables](#)
  - [Temporal Table Usage Scenarios](#)
- **Examples:**

## STEP 4. 注目の新機能を試してみよう

---

この STEP では、「**SQL Server R Services**」や「**JSON 対応**」、「**Stretch Database**」、「**Azure バックアップ**」、「**PolyBase**」など、SQL Server 2016 CTP 3.2 で提供された注目の新機能を試してみましょう。

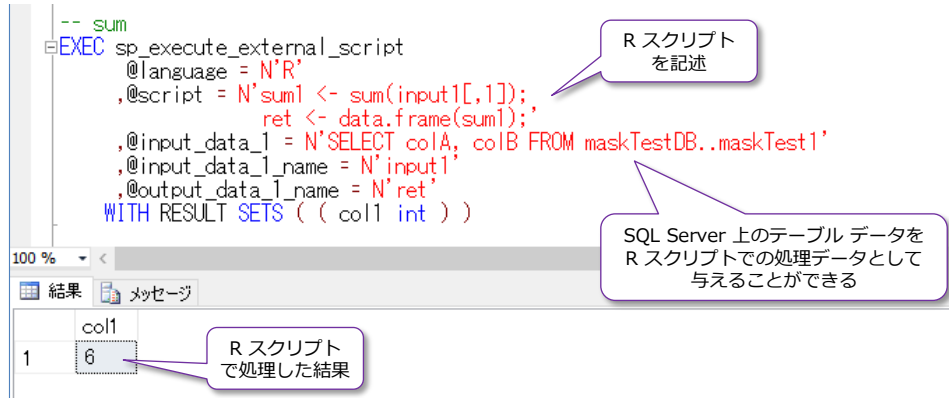
この STEP では、次のことを学習します。

- ✓ SQL Server R Services
- ✓ JSON 対応 (FOR JSON、OPENJSON、JSON\_VALUE など)
- ✓ Stretch Database
- ✓ Azure バックアップ (Backup To URL) の性能向上
- ✓ PolyBase で Hadoop アクセス (Hortonworks や HDInsight)



## 4.1 SQL Server R Services (R 統合)

**SQL Server R Services** は、Transact-SQL ステートメントを利用して、**R スクリプト/ライブラリ**を実行することができる、**R 統合機能**です。



R 言語は、オープンソースの統計解析向けのプログラミング言語で、これを SQL Server 環境に統合（ビルトイン）して、実行できるようにしたのが「**SQL Server R Services**」です。この R 統合機能は、内部的には「**Revolution R Enterprise 7.5.0**」を利用していて、「**Revolution R**」は、オープンソースの R の性能的な欠点を補うために、Enterprise 向けの性能強化を計ったプラットフォームです（Revolution R は、2015 年にマイクロソフトが買収し、SQL Server R Services として SQL Server 2016 に統合、および 2016 年 1 月 1 日以降は「**Microsoft R Server**」という名称に変更して、Linux 環境でも利用できる単体製品としても販売されています）。

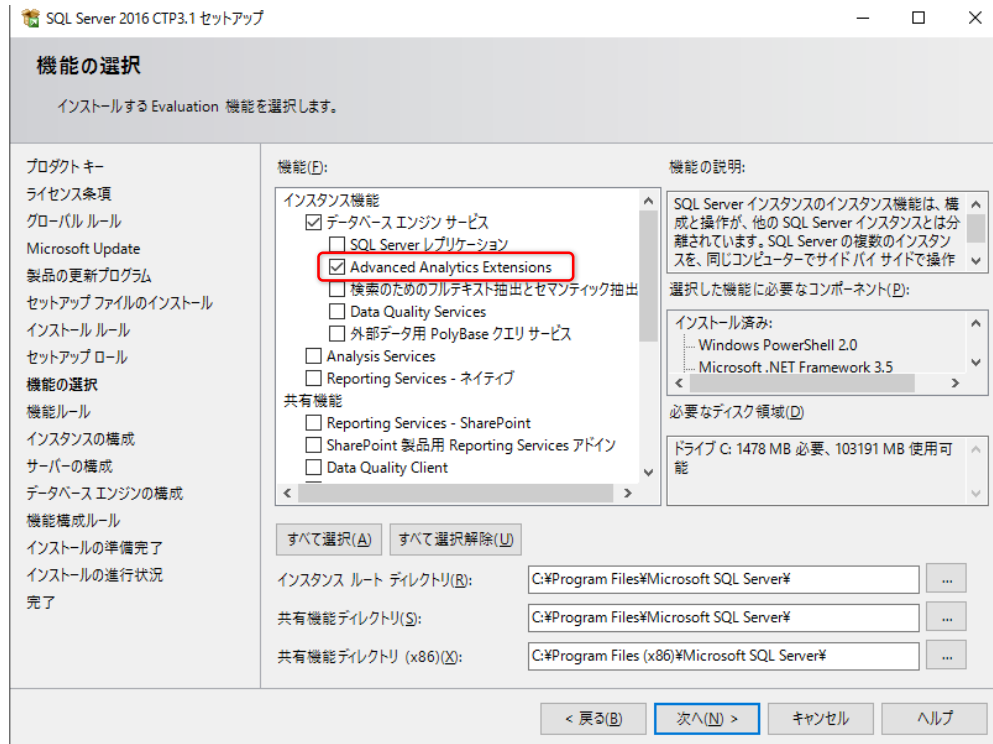
SQL Server 2016 CTP 3.2 で、SQL Server R Services を利用するための流れは、次のとおりです（CTP 3.2 では、SQL Server のインストーラーに統合されていませんが、RTM 版ではインストーラーに統合される予定です）。

- SQL Server のインストール時に「**Advanced Analytics Extensions**」をチェック
- **Revolution R Open 3.2.2 for Revolution R Enterprise 7.5.0** のインストール  
<http://go.microsoft.com/fwlink/?LinkId=626650>
- **Revolution R Enterprise 7.5** のインストール  
<http://go.microsoft.com/fwlink/?LinkId=626652>
- sp\_configure で「**external scripts enabled**」を「**1**」に変更
- Revolution R Open 3.2.2 の **RegisterRExt.exe /install** を実行

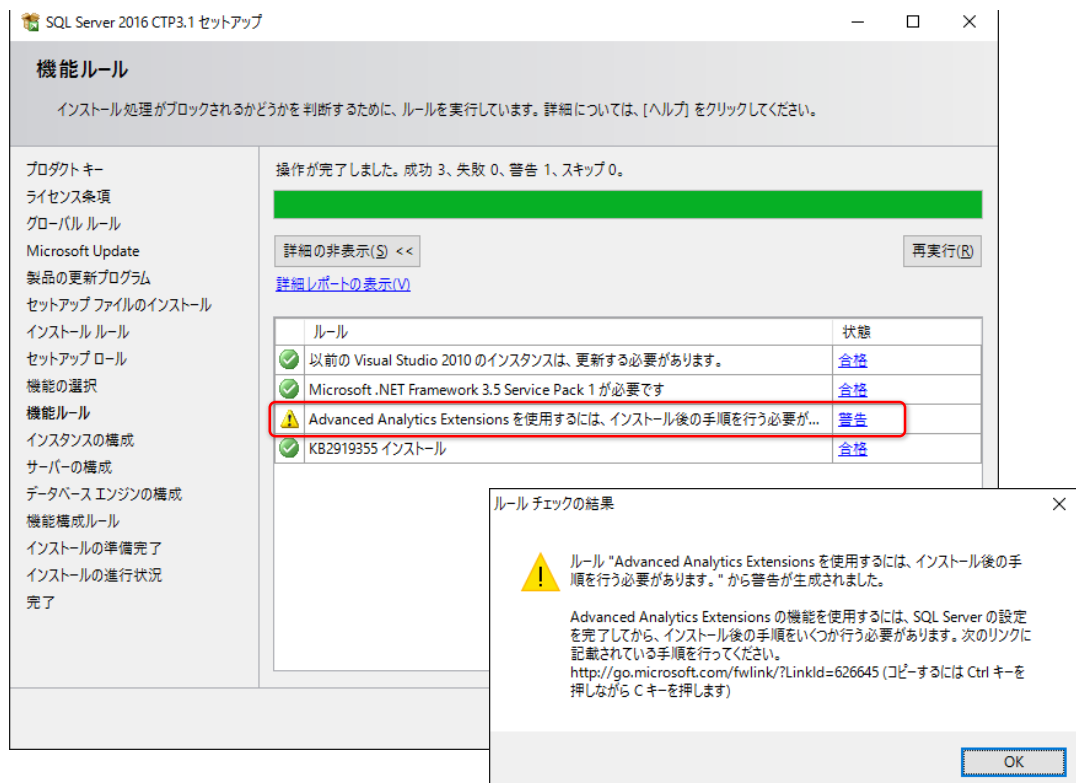
### ➡ SQL Server R Services のインストール

それでは、SQL Server R Services をインストールしてみましょう。

1. まずは、SQL Server 2016 CTP 3.2 のインストール時に、次のように「**機能の選択**」ページで、「**データベース エンジン サービス**」の「**Advanced Analytics Extensions**」をチェックします。



この **Advanced Analytics Extensions** を選択した場合は、次の「機能ルール」ページで、次のように警告が表示されますが（追加の手順が必要になるという主旨）、これは無視してインストールを継続して大丈夫です。



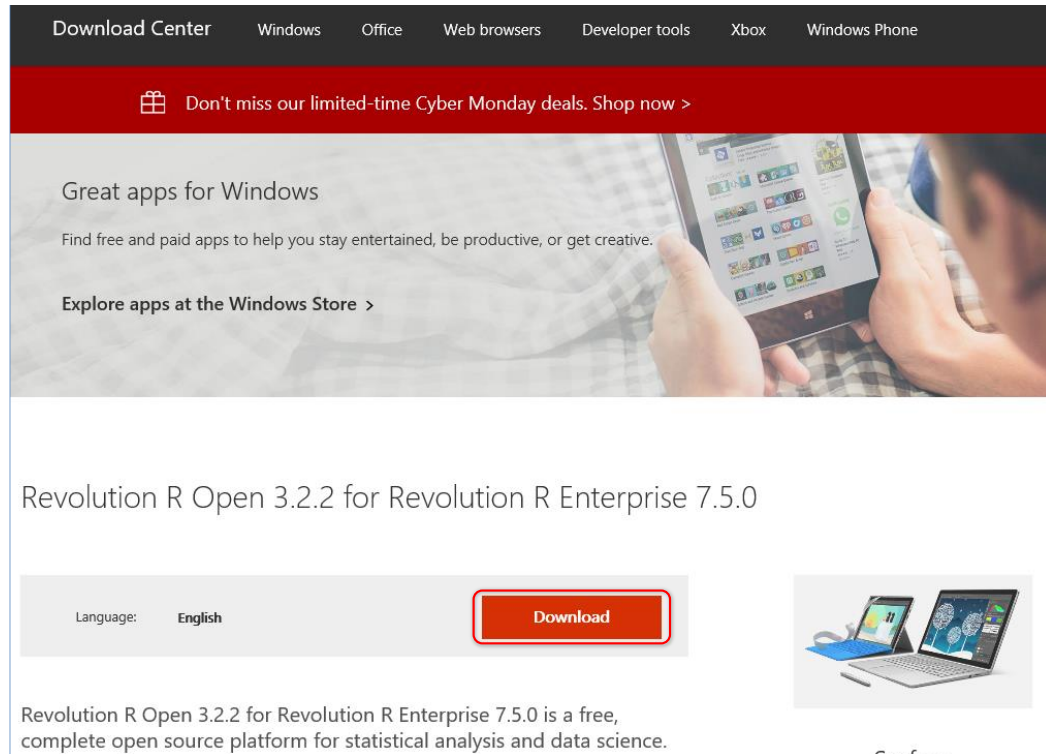
残りの項目は、通常の SQL Server のインストールと同様進めます。

- SQL Server 2016 CTP 3.2 のインストールが完了した後は、「**Revolution R Open 3.2.2 for Revolution R Enterprise 7.5.0**」をインストールします。これは、次の URL からダウンロード

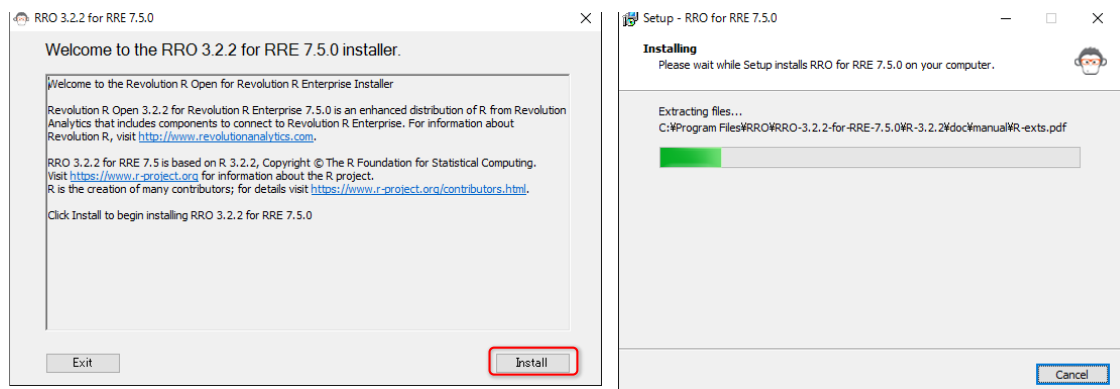
ンロードすることができます。

Revolution R Open 3.2.2 for Revolution R Enterprise 7.5.0 のダウンロード

<http://go.microsoft.com/fwlink/?LinkId=626650>



ダウンロード後は、「**RRO-3.2.2-for-RRE-7.5.0-Windows.exe**」をダブルクリックして、インストールを行います。

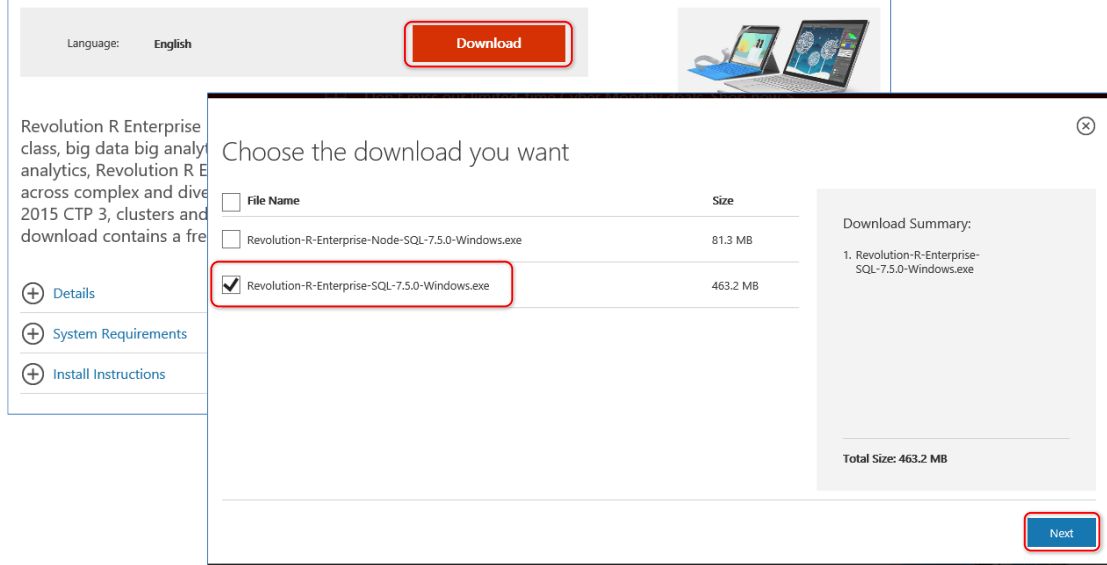


- 次に、「**Revolution R Enterprise 7.5**」をインストールします。これは、次の URL からダウンロードすることができます。

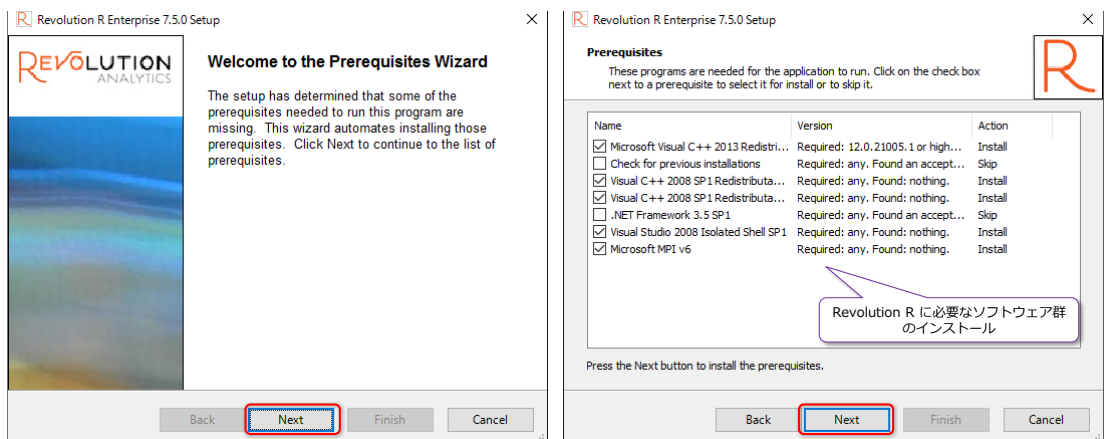
Revolution R Enterprise 7.5 のダウンロード

<http://go.microsoft.com/fwlink/?LinkId=626652>

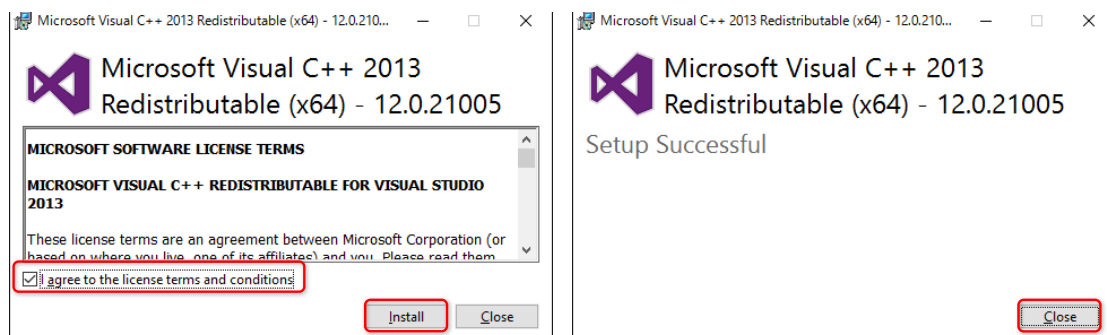
## Revolution R Enterprise 7.5.0 (RRE-7.5.0)

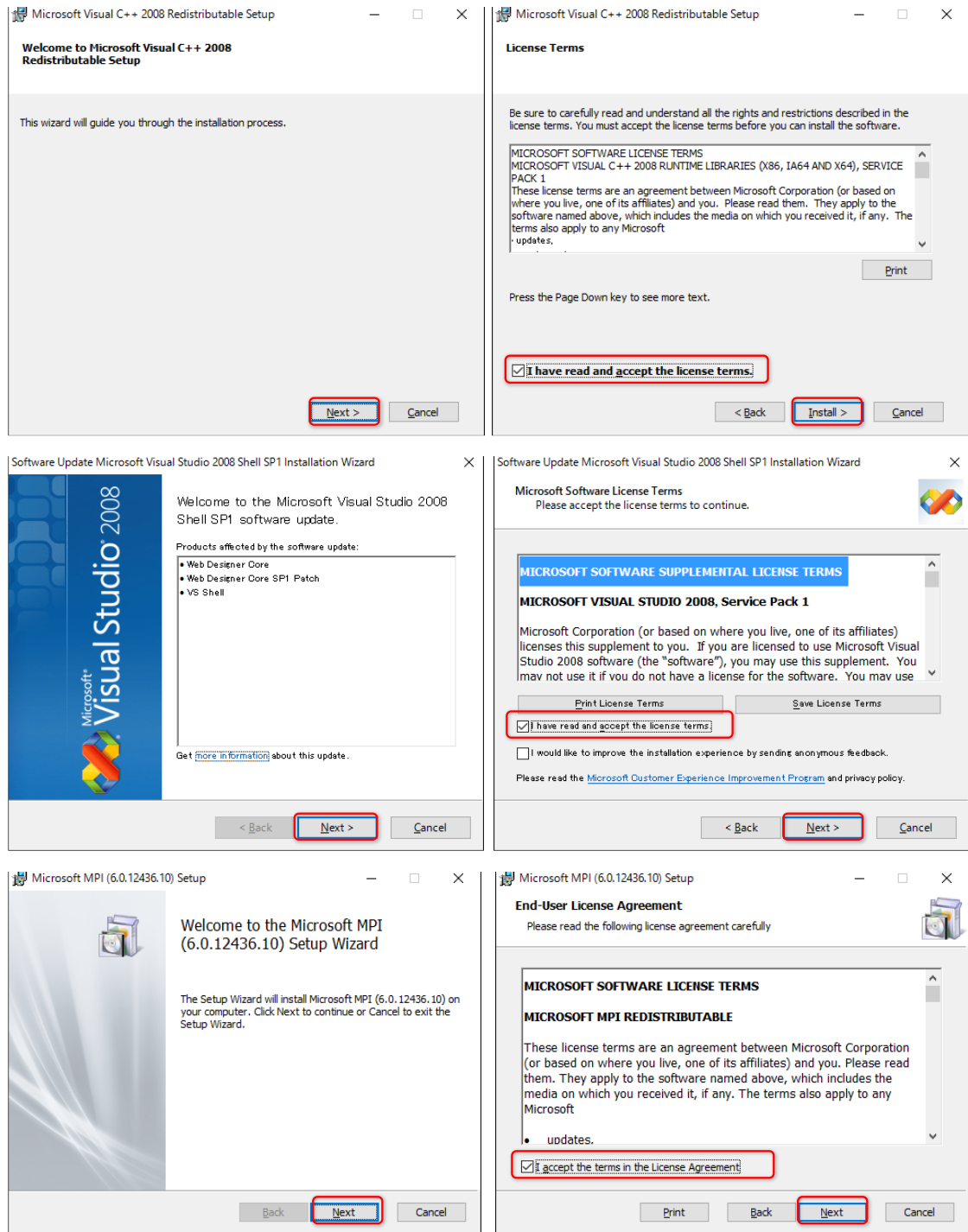


ダウンロード後は、「**Revolution-R-Enterprise-SQL-7.5.0-Windows.exe**」をダブルクリックして、インストールを行います。

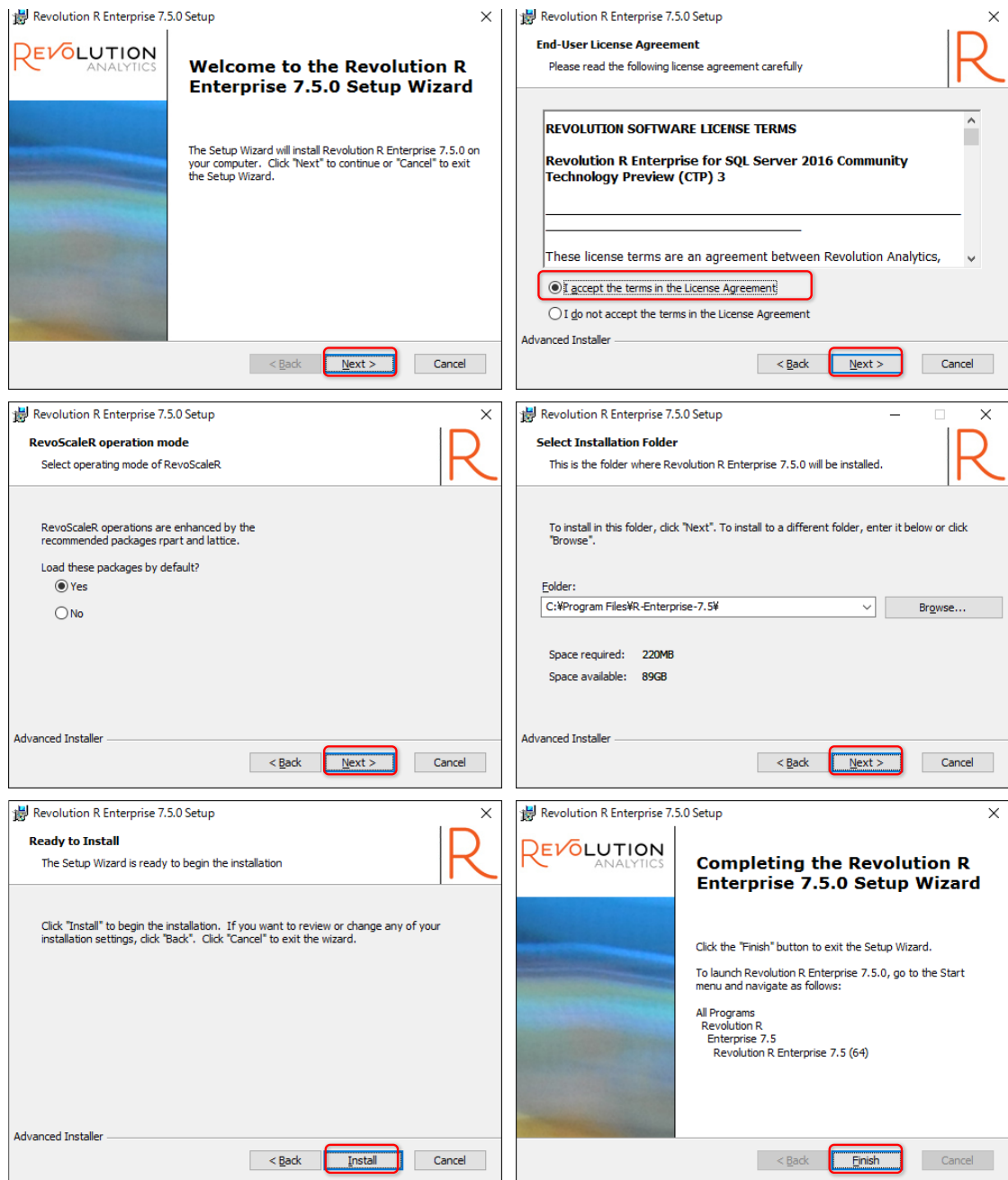


Revolution R には、Visual C++ 2013 Redistributable や Visual C++ 2008 Redistributable、Microsoft MPI などが必須コンポーネントになるので、次のようにセットアップが促されます（それぞれ使用許諾契約書の内容を確認した上で、I agree をチェックして、Install をクリックします。設定項目は、すべて既定値のままで大丈夫です）。



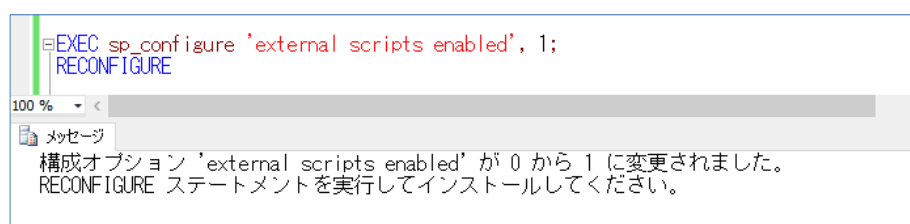


必須コンポーネントのインストールが完了した後は、次のように Revolution R のセットアップが始まります（こちらもすべて既定値でのインストールで大丈夫です）。



4. Revolution R のインストールが完了した後は、SQL Server 2016 CTP 3.2 の Management Studio を起動して、クエリ エディターで、**sp\_configure** を利用して「**external scripts enabled**」を「**1**」に変更します。

```
EXEC sp_configure 'external scripts enabled', 1;
RECONFIGURE
```





5. 次に、Revolution R Open 3.2.2 の「**RegisterRExt.exe /install**」を実行します。これは、コマンド プロンプトから、次のように実行します。

```
"%programfiles%\R\R-3.2.2-for-RRE-7.5.0\R-3.2.2\library\RevoScaleR\libs\x64\RegisterRExt.exe" /install
```

```

C:\>"%programfiles%\R\R-3.2.2-for-RRE-7.5.0\R-3.2.2\library\RevoScaleR\libs\x64\RegisterRExt.exe" /install
Source directory to pick the R extension binaries determined to be "C:\Program Files\R\R-3.2.2-for-RRE-7.5.0\R-3.2.2\library\RevoScaleR\libs\x64".
Connecting to SQL server...
Sql server bin directory is "C:\Program Files\Microsoft SQL Server\MSSQL13.MSSQLSERVER\MSSQL\Binn".
Sql server log directory is "C:\Program Files\Microsoft SQL Server\MSSQL13.MSSQLSERVER\MSSQL\Log".
Creating user account pool...
Creating user accounts.
Processing 20/20
Adding firewall rule for user account pool.
***WARNING: For security reasons, it is recommended to block network access for R processes executing under local user accounts. It appears that Windows Firewall is currently disabled for the Network Profiles DOMAIN, PRIVATE, PUBLIC. Please enable Windows Firewall or use another firewall to block network access.
Creating working directory for user accounts.
Saving user account configuration.
User account pool with 20 users created.
R installation found at C:\Program Files\R\R-3.2.2-for-RRE-7.5.0\R-3.2.2.
MPI installation found at C:\Program Files\Microsoft MPI.
Settings file C:\Program Files\Microsoft SQL Server\MSSQL13.MSSQLSERVER\MSSQL\Binn\rlauncher.config created.
Stopping service MSSQLLaunchpad...
Stopping service MSSQLSERVER...
Copied xp_callrre.dll from C:\Program Files\R\R-3.2.2-for-RRE-7.5.0\R-3.2.2\library\RevoScaleR\libs\x64 to C:\Program Files\Microsoft SQL Server\MSSQL13.MSSQLSERVER\MSSQL\Binn.
Copied Rlauncher.dll from C:\Program Files\R\R-3.2.2-for-RRE-7.5.0\R-3.2.2\library\RevoScaleR\libs\x64 to C:\Program Files\Microsoft SQL Server\MSSQL13.MSSQLSERVER\MSSQL\Binn.
Starting service MSSQLLaunchpad...
Connecting to SQL server...
Created role db_rrole.
Added extended stored procedure xp_ScaleR_init_job.
Granted Execute for extended stored procedure xp_ScaleR_init_job to db_rrole.
Added extended stored procedure xp_ScaleR_queue_job.
Granted Execute for extended stored procedure xp_ScaleR_queue_job to db_rrole.
Added extended stored procedure xp_ScaleR_retrieve_results.
Granted Execute for extended stored procedure xp_ScaleR_retrieve_results to db_rrole.
Added extended stored procedure xp_ScaleR_query_status.
Granted Execute for extended stored procedure xp_ScaleR_query_status to db_rrole.
Added extended stored procedure xp_ScaleR_cancel_job.
Granted Execute for extended stored procedure xp_ScaleR_cancel_job to db_rrole.
Added extended stored procedure xp_ScaleR_cleanup.
Granted Execute for extended stored procedure xp_ScaleR_cleanup to db_rrole.
R extensibility installed successfully.

C:\>

```

最後の行に「**R extensibility installed successfully**」と表示されれば、インストールが完了です。これで、SQL Server に R が統合（ビルトイン）されて、クエリ エディターから R スクリプトを実行できるようになります（スクリプトの実行には、後述の **sp\_execute\_external\_script** システム ストアド プロシージャを利用します）。

## ➡ R スクリプトの実行 ～sp\_execute\_external\_script～

インストールが完了した後は、クエリ エディターを利用して、R スクリプトを実行してみましょう。これは、**sp\_execute\_external\_script** システム ストアド プロシージャを利用して、次のように記述します。

```

EXEC sp_execute_external_script
    @language = N' R'
    , @script = N' 実行したい R スクリプト'
    , @input_data_1 = N' R で処理したいデータ'
    , @input_data_1_name = N' input_data_1 に対して設定する名前'
    , @output_data_1_name = N' R スクリプトで処理した結果'
    WITH RESULT SETS ((列名 データ型 null/not null, ...));

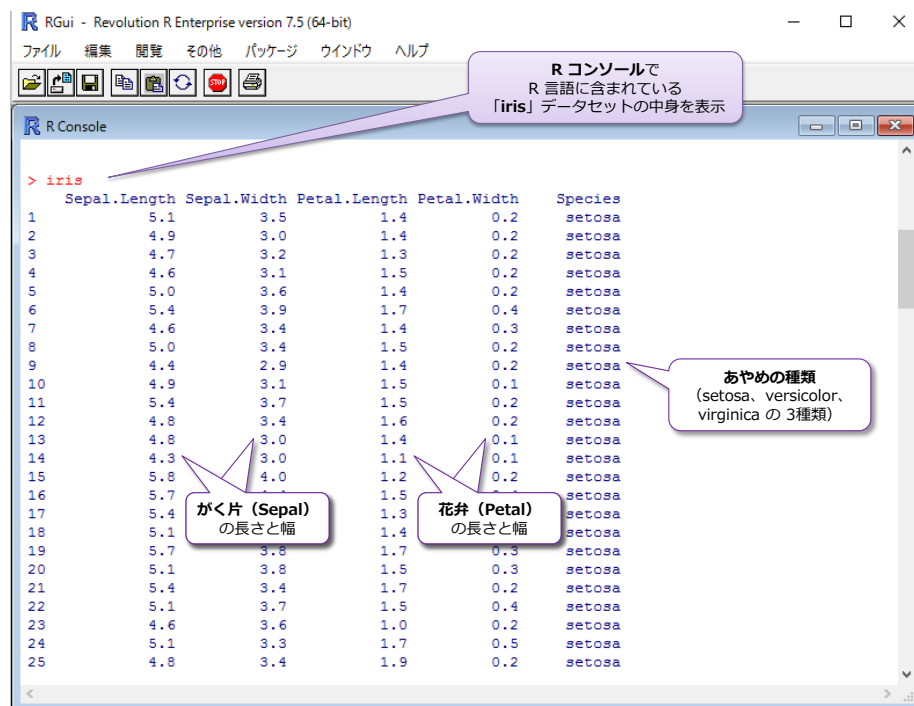
```



@language で「R」を指定、@script に「R スクリプト」を記述することで、任意の R スクリプトを実行できるようになります。また、R スクリプトに与えたい入力データは「@input\_data\_1」および「@input\_data\_1\_name」で指定し、R スクリプトで処理した結果（出力データ）は「@output\_data\_1\_name」で指定することができます。ただし、output に指定できるのは、R のデータ フレーム形式のデータ (data.frame 関数で変換可能なもの) である必要があります。

## ➡ Let's Try

それでは、これを試してみましょう。ここでは、R 言語に含まれる、サンプルのデータセットである「iris」（アヤメのがく片と花弁の長さや幅が格納されたデータ）を取得してみます。このデータは、R のツールである「R コンソール」で「iris」と入力することで、次のように内容を確認できます（R コンソールは、[スタート] メニューの [すべてのアプリ] → [PRO-for-RRE] の [PRO for RRE 7.5.0] をクリックして起動することができます）。

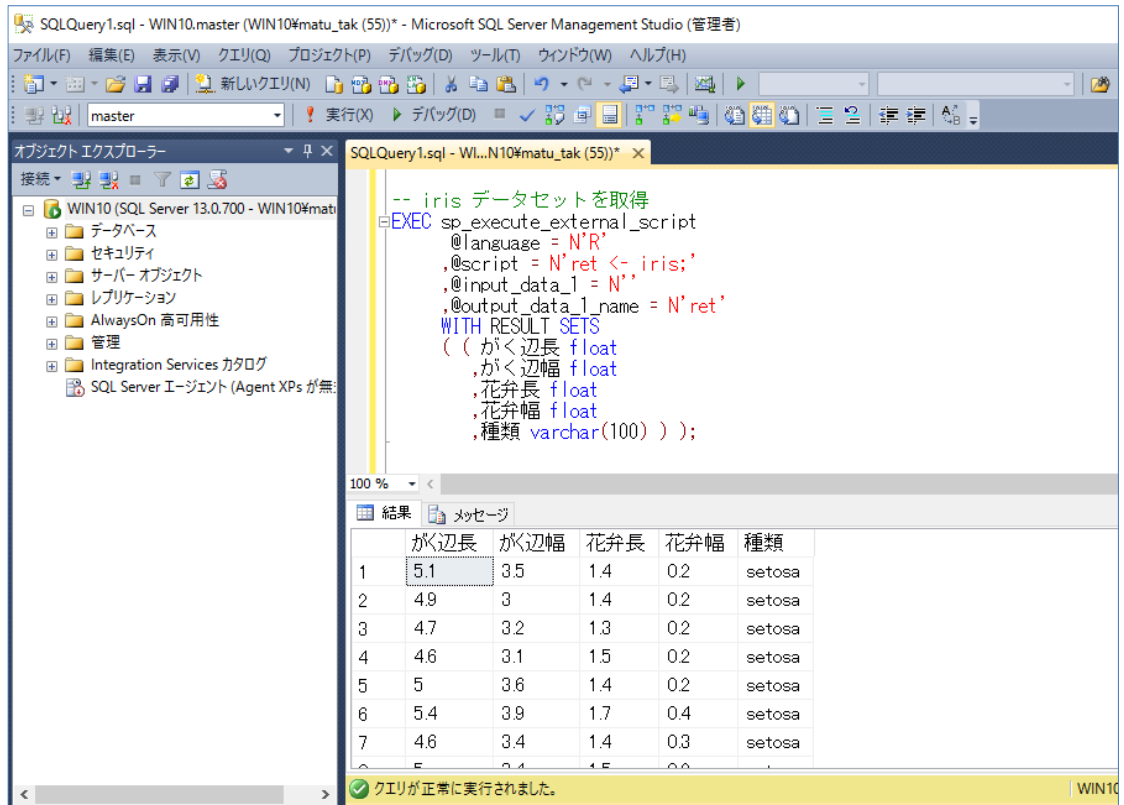


このデータは、植物生物学者である Edgar Anderson 氏が 50 の花（アヤメ : iris）から収集した「がく片」（Sepal）と「花弁」（Petal）の長さや幅になっています。Species 列には、アヤメの種類（品種）が格納されています。

1. このデータを **sp\_execute\_external\_script** システム ストアド プロシージャを利用して取得するには、次のように記述します（Management Studio のクエリ エディターで記述）。

```
-- iris データセットを取得
EXEC sp_execute_external_script
    @language = N'R'
    ,@script = N'ret <- iris;'
    ,@input_data_1 = N''
```

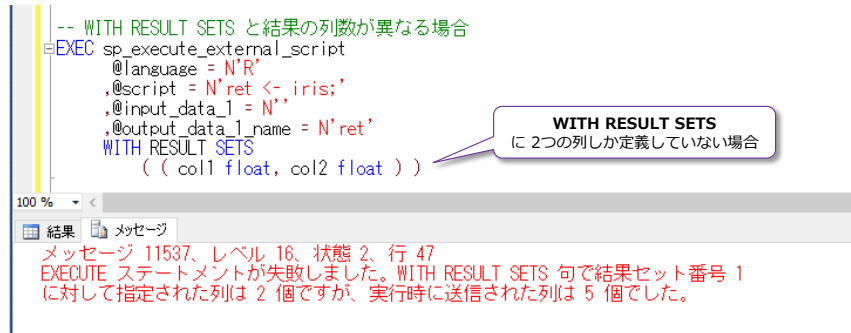
```
,@output_data_1_name = N'ret'
WITH RESULT SETS
( ( がく辺長 float
  , がく辺幅 float
  , 花弁長 float
  , 花弁幅 float
  , 種類 varchar(100) ) )
```



@script (R スクリプトを記述する場所) には、「ret <- iris;」と記述して、iris データセットを「ret」変数 (データ フレーム) に代入しています (R 言語では <- で値の代入ができます)。  
@output\_data\_1\_name (出力データの名前) には、この「ret」データ フレームを指定することで、SQL Server 側で結果を取得できるようになります。

@input\_data\_1 には、R スクリプトに与えたい入力データを指定しますが、今回は入力データを利用しないので、空文字を指定しています (この引数は、必須の引数になるので、値を指定しない場合は空文字を与えるようにします)。

WITH RESULT SETS では、出力結果に対して、SQL Server 側で利用する際の**列名とデータ型、NULL の可否** (NULL または NOT NULL を指定。省略時は NULL) を記述します。今回の iris データは、がく片の長さ (Sepal.Length) と幅 (Sepal.Width)、花弁の長さ (Petal.Length) と幅 (Petal.Width)、アヤメの種類 (Species) の 5 つの列が返るので、WITH RESULT SETS に 5 個分の列定義を記述しています。もし、結果セットの列数と、WITH RESULT SETS で定義した列数が異なる場合には、次のようにエラーが返されます。



このように、SQL Server R Services を利用すると、R スクリプトを実行して、その実行結果を SQL Server 側で処理できるようになるので、大変便利です。

## ➡ SQL Server のデータを input に与える ～@input\_data\_1～

次に、SQL Server のデータベース内のデータを、入力値として R スクリプトに与えてみましょう。これを行うには、@input\_data\_1 引数に **SELECT** ステートメントを記述するようにします。ここでは、動的データ マスクのところで利用した「**maskTestDB**」データベースの「**maskTest1**」テーブルのデータで試してみましょう。

	colA	colB	colC	colD	colE
1	1	AAAAA	111	2015-10-30 00:00:00.000	NULL
2	2	BBBBB	222	2015-11-30 00:00:00.000	NULL
3	3	CCCCC	333	2015-11-30 00:00:00.000	ccc@test.local

1. SQL Server のデータを入力値として与えるには、次のように記述します。

```
EXEC sp_execute_external_script
    @language = N'R'
    ,@script = N'ret <- input1;'
    ,@input_data_1 = N'SELECT colA, colB FROM maskTestDB..maskTest1'
    ,@input_data_1_name = N'input1'
    ,@output_data_1_name = N'ret'
    WITH RESULT SETS ( (colA int, colB varchar(200) ) )
```

	colA	colB
1	1	AAAAA
2	2	BBBBB
3	3	CCCCC

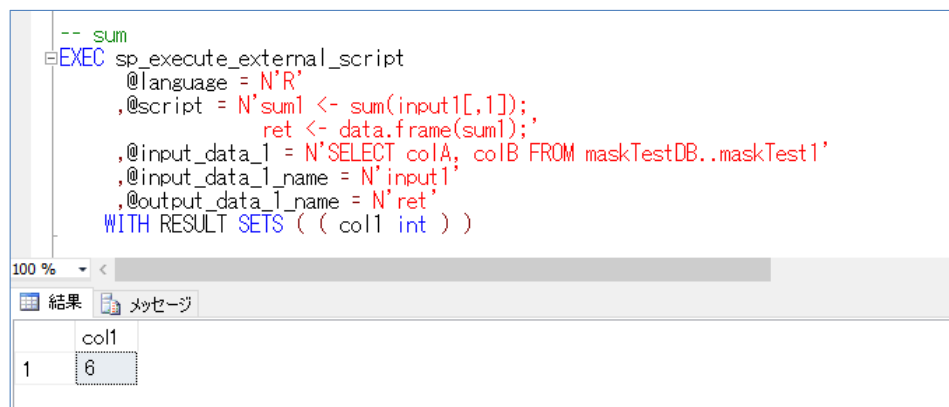
**@input\_data\_1** 引数に **SELECT** ステートメントを記述して、**@input\_data\_1\_name** 引数に **SELECT** ステートメントの結果に対して任意の名前（ここでは **input1** と指定）を設定します。ここで指定した名前では、R スクリプトを処理できるようになります（もし、**@input\_data\_1\_name** 引数を省略した場合は、**InputDataSet** という名前が補われます）。

**@script**（R スクリプトを記述する場所）には、「**ret <- input1;**」と記述しているので、**SELECT** ステートメントの結果をそのまま「**ret**」データ フレームに格納して、それを **WITH RESULT SETS** で定義した列定義に代入しています。

このように、SQL Server のデータを R スクリプトに与えることも簡単に行うことができます。

- 次に、R スクリプトの関数を利用して、入力データを処理してみましょう。ここでは、**sum** 関数を利用して、**colA** 列の合計を計算してみます。

```
EXEC sp_execute_external_script
    @language = N'R'
    ,@script = N'sum1 <- sum(input1[,1]);
                ret <- data.frame(sum1);'
    ,@input_data_1 = N'SELECT colA, colB FROM maskTestDB..maskTest1'
    ,@input_data_1_name = N'input1'
    ,@output_data_1_name = N'ret'
    WITH RESULT SETS ( ( col1 int ) )
```



```
-- sum
EXEC sp_execute_external_script
    @language = N'R'
    ,@script = N'sum1 <- sum(input1[,1]);
                ret <- data.frame(sum1);'
    ,@input_data_1 = N'SELECT colA, colB FROM maskTestDB..maskTest1'
    ,@input_data_1_name = N'input1'
    ,@output_data_1_name = N'ret'
    WITH RESULT SETS ( ( col1 int ) )
```

col1
6

手順 1 のスクリプトから変更したのは **@script** に記述した R スクリプトと **WITH RESULT SETS** で指定した列定義のみです。**@script** には、1 行目に「**sum1 <- sum(input1[,1]);**」と記述して、**input1** の 1 列目のデータ（SQL Server から取得した **colA** 列のデータ）に対して R 言語の **sum** 関数を利用して処理し、その結果を **sum1** 変数に代入しています。

次に、「**ret <- data.frame(sum1);**」で、**sum1** 変数を **data.frame** 関数でデータ フレームに変換して、その結果を **ret** 変数（データ フレーム）に格納しています。この結果は、1 列分なので、**WITH RESULT SETS** で指定する列定義も 1 列分（**col1 int** のみ）にしています。

このように、R スクリプトは、「`;`」で行の区切りを付けて、複数行実行することができます。また、スクリプト内では R 言語の関数や、ライブラリを利用することができるので、さまざまな統計解析処理を行っていくことができます。

なお、上の R スクリプトは、「`ret <- data.frame(sum(input1[1]));`」のように 1 行で記述することもできます。

その他、SQL Server R Services 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。

## SQL Server R Services

<https://msdn.microsoft.com/en-us/library/mt604845.aspx>

Microsoft | Developer Network Sign in MS

Technologies ▾ Downloads ▾ Programs ▾ Community ▾ Documentation ▾ Samples ▾

\*\*\* > Books Online for SQL Server 2016 > Database Engine > Database Engine Features and Tasks ▾

# SQL Server R Services

SQL Server 2016

Updated: October 27, 2015

SQL Server R Services provides a platform for writing intelligent applications that uncover new insights and create predictions on top of your data. It allows using the rich and powerful R language and package portfolio, while bringing the analytics close to the data and eliminating costs and security risks associated with data movement. SQL Server R Services provides a convenient, familiar Transact-SQL interface allowing your production applications to call the R runtime and retrieve predictions and visuals. With Enterprise Edition you also benefit from libraries that allow overcoming R's performance and scale limitations. You get to marry R with the comprehensive set of SQL Server tools and technologies that offer superior performance, security, reliability and manageability.

In SQL Server 2016, SQL Server R Services is comprised of server and client components. The server components include:

- **Advanced Analytics Extensions.** Install this feature during SQL Server setup to enable secure execution of R scripts on the SQL Server computer.
- **Revolution R Enterprise 7.5.0 and Revolution R Open 3.2.2 for Revolution R Enterprise 7.5.0** These downloads are prerequisites for using SQL Server R Services. These downloads include a set of high performance R packages, connectivity tools and the open-source R distribution.

After you have installed and configured the server components, you will be able to execute R code against your SQL Server data by using Transact-SQL in system stored procedures. The stored procedures can return predictions and plots generated by the R code to your application.

The client components of SQL Server R Services include these downloads: Revolution R Enterprise 7.5.0, and Revolution R Open 3.2.2 for Revolution R Enterprise 7.5.0. With these tools, R developers and data scientists can explore the data and build predictive models from a client workstation using any R IDE they like. After the R code is ready to be put into production, it can be embedded in Transact-SQL stored procedures.

## 4.2 JSON 対応 (FOR JSON、OPENJSON、JSON\_VALUE など)

SQL Server 2016 では、**JSON** (JavaScript Object Notation) に対応して、SQL Server 上のテーブル データを JSON 形式で出力したり、JSON データを SQL Server に取り込んだりできるようになりました。具体的には、次の機能が提供されています。

- **FOR JSON** で SQL Server 上のテーブル データを JSON 形式で出力

```
-- JSON 形式で出力
SELECT 商品コード, 商品名 FROM Northwind..商品
FOR JSON AUTO
```

テーブル データを JSON 形式で出力

結果	メッセージ
JSON_F52E2B61-18A1-11d1-B105-00805F49916B	
1	[{"商品コード":1,"商品名":"果汁100% オレンジ"}, {"商品コード":2,"商品名":"果汁100% グレープ"}]
2	り], [{"商品コード":69,"商品名":"乾燥バナナ"}, {"商品コード":70,"商品名":"乾燥アップル"}]

- **OPENJSON** 関数で JSON データの取り込み

```
DECLARE @jsonstr nvarchar(max) =
N'[{ "arr1": [
    { "key1": "test1", "key2": 999 }
    , { "key1": "test2", "key2": 777 }
] } ]'
```

```
SELECT * FROM
OPENJSON(@jsonstr, '$.arr1')
WITH (
    key1 varchar(20),
    key2 int
)
```

OPENJSON 関数で JSON データを取得

結果	メッセージ
1	key1 test1 key2 999
2	key1 test2 key2 777

- **ISJSON** 関数で JSON データかどうかの判別

JSON データを SQL Server 上に格納 (ISJSON 関数を CHECK 制約で利用)

- **JSON\_VALUE** と **JSON\_QUERY** で JSON データをクエリ

```
-- ISJSON 関数で CHECK 制約を設定
CREATE TABLE jsonTest1
( colA int PRIMARY KEY
, colB nvarchar(max) CONSTRAINT chkJSON CHECK ( ISJSON(colB) > 0 )
)

-- JSON データをテーブルに格納
INSERT INTO jsonTest1 VALUES(1, N'[{ "key1": "test1", "key2": 999 } ]')
INSERT INTO jsonTest1 VALUES(2, N'[{ "key1": "test2", "key2": 777 } ]')
```

```
-- テーブル内の JSON データをクエリ
SELECT JSON_VALUE(colB, '$.key1') AS key1
, JSON_VALUE(colB, '$.key2') AS key2
FROM jsonTest1
```

ISJSON 関数で JSON データかどうかをチェック

JSON データをテーブルに格納

JSON\_VALUE 関数で JSON データをクエリ

結果	メッセージ
1	key1 test1 key2 999
2	key1 test2 key2 777

## ➡ FOR JSON で SQL Server 上のテーブル データを JSON 形式で出力

**FOR JSON** を利用すれば、SQL Server 上のテーブル データを JSON 形式で出力できます。これも試してみましょう。これを行うには、次のように **SELECT** ステートメントの末尾に「**FOR JSON AUTO**」と付けるだけです。

```
-- テーブル データを JSON 形式で出力
SELECT 商品コード, 商品名 FROM NorthwindJ..商品
FOR JSON AUTO
```

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

```
-- JSON 出力
SELECT 商品コード, 商品名 FROM NorthwindJ..商品
FOR JSON AUTO
```

Below the query window, the results pane shows a JSON string: `JSON_F52E2B61-18A1-11d1-B105-00805F49916B`. The results are displayed in a table with two rows:

	結果	メッセージ
1	[{"商品コード":1,"商品名":"果汁100% オレンジ"}, {"商品コード":2,"商品名":"果汁100% グレープ"}, {"商品コード":3,"商品名":"果汁100% アップル"}, {"商品コード":4,"商品名":"果汁100% オレンジ"}, {"商品コード":5,"商品名":"果汁100% グレープ"}, {"商品コード":6,"商品名":"果汁100% アップル"}, {"商品コード":7,"商品名":"果汁100% オレンジ"}, {"商品コード":8,"商品名":"果汁100% グレープ"}, {"商品コード":9,"商品名":"果汁100% アップル"}, {"商品コード":10,"商品名":"果汁100% オレンジ"}]	
2		

全体が [] で囲まれた配列として出力されて、各行のデータ (1 行分のデータ) が {} で囲まれて、それぞれ「**列名 1**:"**データ 1**",**列名 2**:"**データ 2**"」(データが数値の場合は " なし) というペアになって出力されています。上の例では、「**商品コード**」と「**商品名**」列を指定しているので、{"**商品コード**":1, "**商品名**":"**果汁 100% オレンジ**"} という形で 1 行分のデータが出力されています。なお、出力データ量が多い場合 (出力する JSON 文字列が長くなる場合) には、上の結果のように複数行に分かれて出力されます。

## ➡ OPENJSON 関数で JSON データの取り込み

**OPENJSON** は、JSON データを取り込むことができる関数です。これも試してみましょう。まずは、次のように実行してみます。

```
SELECT * FROM
OPENJSON(N' [{"key1": "test1", "key2": 999}]')
```

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

```
-- OPENJSON
SELECT * FROM
OPENJSON(N' [{"key1": "test1", "key2": 999}]')
```

Below the query window, the results pane shows a table with the following data:

	key	value	type
1	key1	test1	1
2	key2	999	2

各キー (**key1**、**key2**) とそれぞれの値 (**test1**、**999**) が 1 行ずつ表示されていることを確認できます。**type** 列は、キー値の種類が返り、「1」は文字列 (**key1** は **test1** という文字データなので 1)、「2」は数値 (**key2** は **999** という数値データなので 2) になります。



次に、**OPENJSON** 関数で **WITH** キーワードを追加して、各キーの値を列として取得してみます（前の例は、各キー値を “行” として取得しましたが、今度は “列” として取得します）。

```
SELECT * FROM
OPENJSON(N'{"key1": "test1", "key2": 999}')
```

```
WITH (
    key1 varchar(20),
    key2 int
)
```

```
-- WITH でキーを指定
SELECT * FROM
OPENJSON(N'{"key1": "test1", "key2": 999}')
```

```
WITH (
    key1 varchar(20),
    key2 int
)
```

100 %

	key1	key2
1	test1	999

このように、**WITH** キーワードでは、キーの名前（**key1**、**key2**）とデータ型を指定することで、列として取得することもできます。

次に、**OPENJSON** 関数に、**文字列の変数**を与えてみます。

```
DECLARE @jsonstr nvarchar(max) =
    N'{"arr1": [
        {"key1": "test1", "key2": 999}
        , {"key1": "test2", "key2": 777}
    ]}'
```

```
SELECT * FROM
OPENJSON(@jsonstr)
```

```
-- OPENJSON に文字列変数を与える
DECLARE @jsonstr nvarchar(max) =
    N'{"arr1": [
        {"key1": "test1", "key2": 999}
        , {"key1": "test2", "key2": 777}
    ]}'
```

```
SELECT * FROM
OPENJSON(@jsonstr)
```

100 %

	key	value	type
1	arr1	[{"key1": "test1", "key2": 999}, {"key1": "test2", "key2": 777}]	4

JSON データを **@jsonstr** という**文字列変数**に格納して、それを **OPENJSON** 関数で取得しています。この JSON データは、**arr1** というキーに対して、**[ ]**（大カッコ）で囲んで配列を構成しているので、**type** が「**4**」（配列を表す値）で返ってきています。なお、JSON では **[要素 1, 要素 2, ...]** という形で、配列を表現できます。これに対して、**{ }**（中カッコ）で囲むものは、**{"**

キー名 1:"値 1", "キー名 2":"値 2"} という形で、キー名とその値が「:」で区切られてペアになり、これは**オブジェクト**と呼ばれています。

### OPENJSON 関数で Path を指定する ～OPENJSON(jsonstr, path)～

OPENJSON 関数では、第 2 引数に JSON データ内のパス（階層構造がある場合のパス）を記述することができます。これも試してみましょう。

```
DECLARE @jsonstr nvarchar(max) =
    N' { "arr1": [
        { "key1": "test1", "key2": 999 }
        , { "key1": "test2", "key2": 777 }
    ] }'

SELECT * FROM
    OPENJSON(@jsonstr, '$.arr1')
```

-- OPENJSON で Path を指定

```
DECLARE @jsonstr nvarchar(max) =
    N' { "arr1": [
        { "key1": "test1", "key2": 999 }
        , { "key1": "test2", "key2": 777 }
    ] }'
```

大カッコ内のカンマで区切られた配列

中カッコで囲まれて : でキー名と値が区切られたオブジェクト

パスを指定

```
SELECT * FROM
    OPENJSON(@jsonstr, '$.arr1')
```

	key	value	type
1	0	{"key1": "test1", "key2": 999}	5
2	1	{"key1": "test2", "key2": 777}	5

type の 5 は オブジェクトを表す

OPENJSON 関数の第 2 引数に「\$.arr1」と指定することで、「arr1」の値である []（大カッコ）で囲まれた配列データを取得できています。この配列の中には、中カッコで囲まれたオブジェクトが 2 つがあるので、2 行に分かれて結果を取得できています。**type** 列の「5」はオブジェクトを表す値になります。

オブジェクトに対しては、**WITH** キーワードでキー名を指定することで、キー値を“列”として出力することもできるので、次のように取得することもできます。

```
-- WITH でキーを指定
DECLARE @jsonstr nvarchar(max) =
    N' { "arr1": [
        { "key1": "test1", "key2": 999 }
        , { "key1": "test2", "key2": 777 }
    ] }'
```

```
SELECT * FROM
    OPENJSON(@jsonstr, '$.arr1')
    WITH (
        key1 varchar(20),
        key2 int
    )
```

WITH キーワードで キー名を指定

	key1	key2
1	test1	999
2	test2	777

各キーの値を 列として取得できる

このように、**OPENJSON** 関数を利用すれば、JSON データを簡単に取得することができます。

## ➡ ISJSON 関数で JSON データかどうかの判別

**ISJSON** は、JSON データかどうかをチェックできる関数です。JSON データであれば **1**、そうでなければ **0** を返すことができます。これも試してみましょう。

```
SELECT ISJSON(N'{"key1": "test1", "key2": 999}')
```

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

```
-- ISJSON
SELECT ISJSON(N'{"key1": "test1", "key2": 999}')
```

The results pane shows a single row with the value 1.

	(列名なし)
1	1

与えた文字列は JSON データなので、**1** が返ることを確認できます。次に、JSON データではない値を与えてみます。

```
SELECT ISJSON(N'aaa')
```

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

```
SELECT ISJSON(N'aaa')
```

The results pane shows a single row with the value 0.

	(列名なし)
1	0

今度は **0** が返ることを確認できます。このように、ISJSON 関数を利用すれば、JSON データかどうかをチェックできるので便利です。

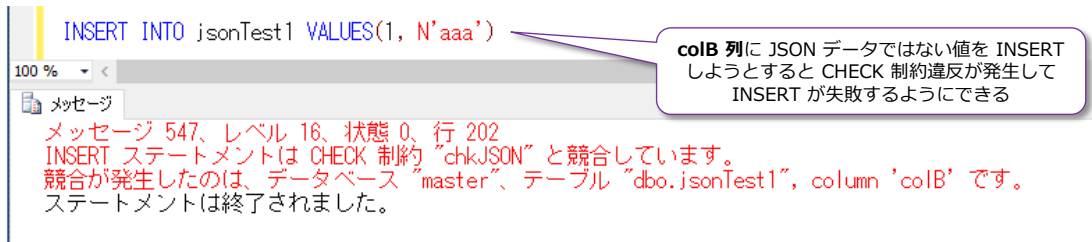
## ➡ JSON データを SQL Server 上に格納 (ISJSON 関数を CHECK 制約で利用)

SQL Server 内に JSON データを格納したい場合には、**ISJSON** 関数を利用して **CHECK 制約** を作成するのがお勧めです。これで、JSON データのみを確実に格納できるようになります。これも試してみましょう。

```
-- ISJSON 関数で CHECK 制約を設定
CREATE TABLE jsonTest1
( colA int PRIMARY KEY
, colB nvarchar(max) CONSTRAINT chkJSON CHECK ( ISJSON(colB) > 0 )
)
```

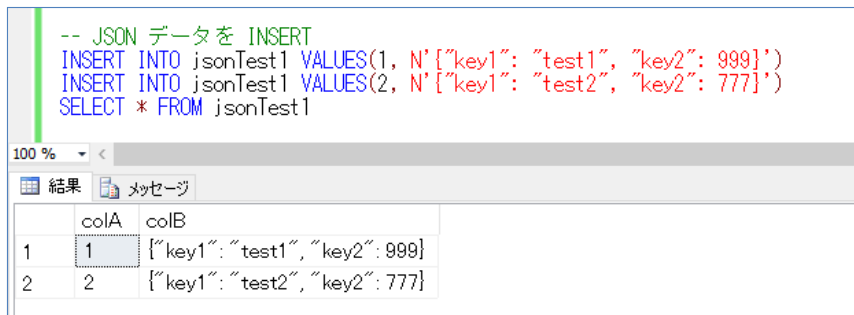
**colB** 列に対して、**CHECK 制約**で **ISJSON** 関数を利用することで、**colB** 列には、JSON データのみが格納できるようになります。

たとえば、次のように JSON 形式ではない値 (aaa) を **INSERT** しようすると、次のように **CHECK 制約違反**が発生して、データが INSERT されるのを防ぐようになります。



次に、**colB** 列に **JSON データ**を格納してみましょう。

```
INSERT INTO jsonTest1 VALUES(1, N'{"key1": "test1", "key2": 999}')
INSERT INTO jsonTest1 VALUES(2, N'{"key1": "test2", "key2": 777}')
SELECT * FROM jsonTest1
```

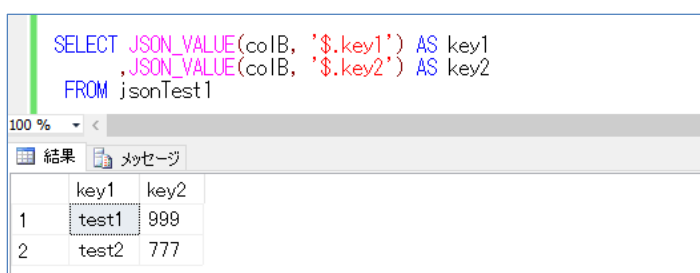


JSON データであれば、何の問題もなく **INSERT** することができます。このように、INSERT したデータは、後述の **JSON\_VALUE** や **JSON\_QUERY** 関数を利用して、簡単に取り出すことができます。

## ➡ JSON\_VALUE と JSON\_QUERY で JSON データをクエリ

**JSON\_VALUE** と **JSON\_QUERY** は、JSON データ (特定のキーやオブジェクト) をクエリできる関数です。これも試してみましょう。

```
SELECT JSON_VALUE(colB, '$.key1') AS key1
       , JSON_VALUE(colB, '$.key2') AS key2
FROM jsonTest1
```



**JSON\_VALUE** 関数では、第 1 引数に JSON 文字列 (ここでは colB 列に格納されたデータ)、

第 2 引数にキーへのパスを指定することで、該当キーの値 (value) を取得することができます。ここでは「\$.key1」と「\$.key2」を指定しているので、各キーの値 (test1 や 999 など) を取得することができます。

このように、SQL Server 内に格納した JSON データでも、**JSON\_VALUE** 関数を利用すれば、簡単に取得することができます。

**JSON\_QUERY** 関数は、オブジェクトのパスを指定して、該当オブジェクトを取得することができます。これは次のように試すことができます。

```
TRUNCATE TABLE jsonTest1
INSERT INTO jsonTest1 VALUES(1, '{ "obj1": { "key1": "test1", "key2": 999 } }')
INSERT INTO jsonTest1 VALUES(2, '{ "obj1": { "key1": "test2", "key2": 777 } }')

SELECT JSON_VALUE(colB, '$.obj1.key1') AS key1
      ,JSON_VALUE(colB, '$.obj1.key2') AS key2
      ,JSON_QUERY(colB, '$.obj1') AS obj1
FROM jsonTest1
```

```
TRUNCATE TABLE jsonTest1
INSERT INTO jsonTest1 VALUES(1, '{ "obj1": { "key1": "test1", "key2": 999 } }')
INSERT INTO jsonTest1 VALUES(2, '{ "obj1": { "key1": "test2", "key2": 777 } }')

SELECT JSON_VALUE(colB, '$.obj1.key1') AS key1
      ,JSON_VALUE(colB, '$.obj1.key2') AS key2
      ,JSON_QUERY(colB, '$.obj1') AS obj1
FROM jsonTest1
```

	key1	key2	obj1
1	test1	999	{ "key1": "test1", "key2": 999 }
2	test2	777	{ "key1": "test2", "key2": 777 }

JSON\_QUERY 関数で取得したオブジェクト

**JSON\_QUERY** は、多段階層になっている場合に、特定のオブジェクトのみを抜き出す場合に便利な関数です。

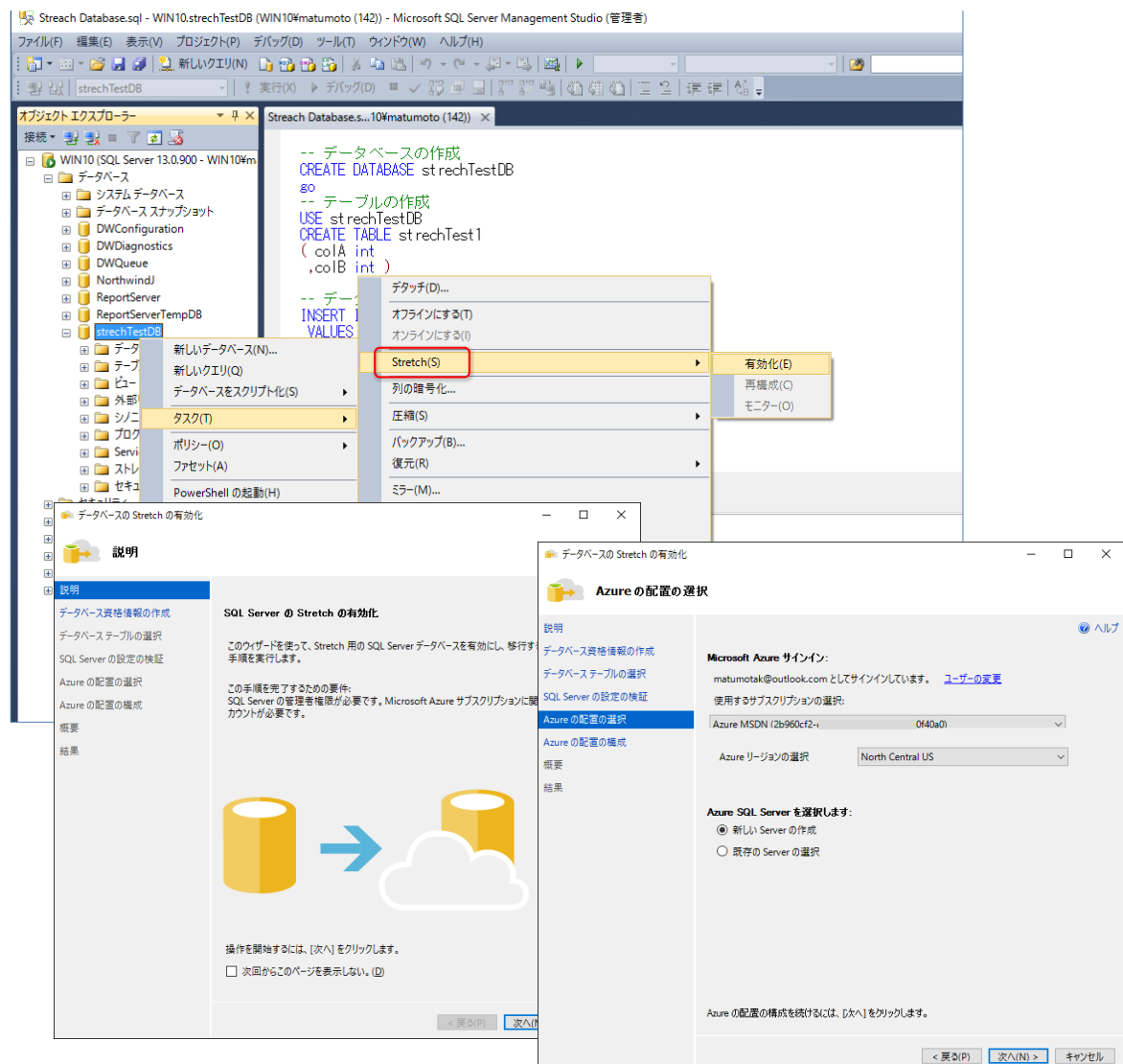
以上のように、SQL Server には、JSON データを扱うためのたくさんの関数が提供されて、JSON データを簡単に扱えるようになっています。

## 4.3 Strech Database ～アクセス頻度の低いデータをクラウドへ～

**Strech Database**（ストレッチ データベース）は、テーブル データをクラウド上（**Microsoft Azure SQL Database**）に配置することができる機能です。

これを利用すれば、アクセス頻度の低いデータ（アクセス ログや、3 年以上前の古いデータなど）や、コンプライアンス（法令遵守）目的でデータ蓄積しておく必要があるもの、ディスク容量を圧迫し得る大量データなどを、クラウド上に配置することで、運用コストを削減できる可能性があるものです。クラウド上に配置したデータは、SQL Server 上のローカル データと同じように操作することができるので、ユーザーは、データがクラウド上にあるのかどうかは意識することなく、透過的に利用することができます。

**Strech Database** は、次のように設定のためのウィザードが用意されているので、非常に簡単に設定することができます。

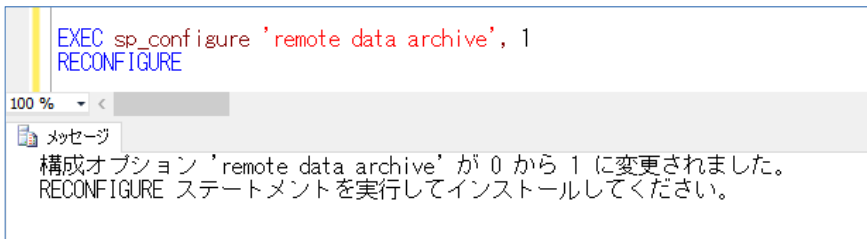


## ➡ Let's Try

それでは、これを試してみましょう。

1. **Strech Database** を利用するには、まず **sp\_configure** で「**remote data archive**」を「**1**」に設定しておく必要があります。

```
EXEC sp_configure 'remote data archive', 1
RECONFIGURE
```

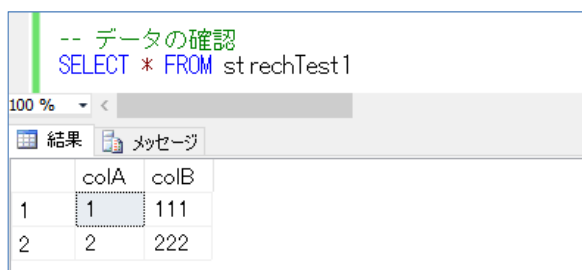


2. 次に、**Strech Database** を試すためのデータベースとテーブルを作成します。

```
-- データベースの作成
CREATE DATABASE strechTestDB
go
-- テーブルの作成
USE strechTestDB
CREATE TABLE strechTest1
( colA int
, colB int )

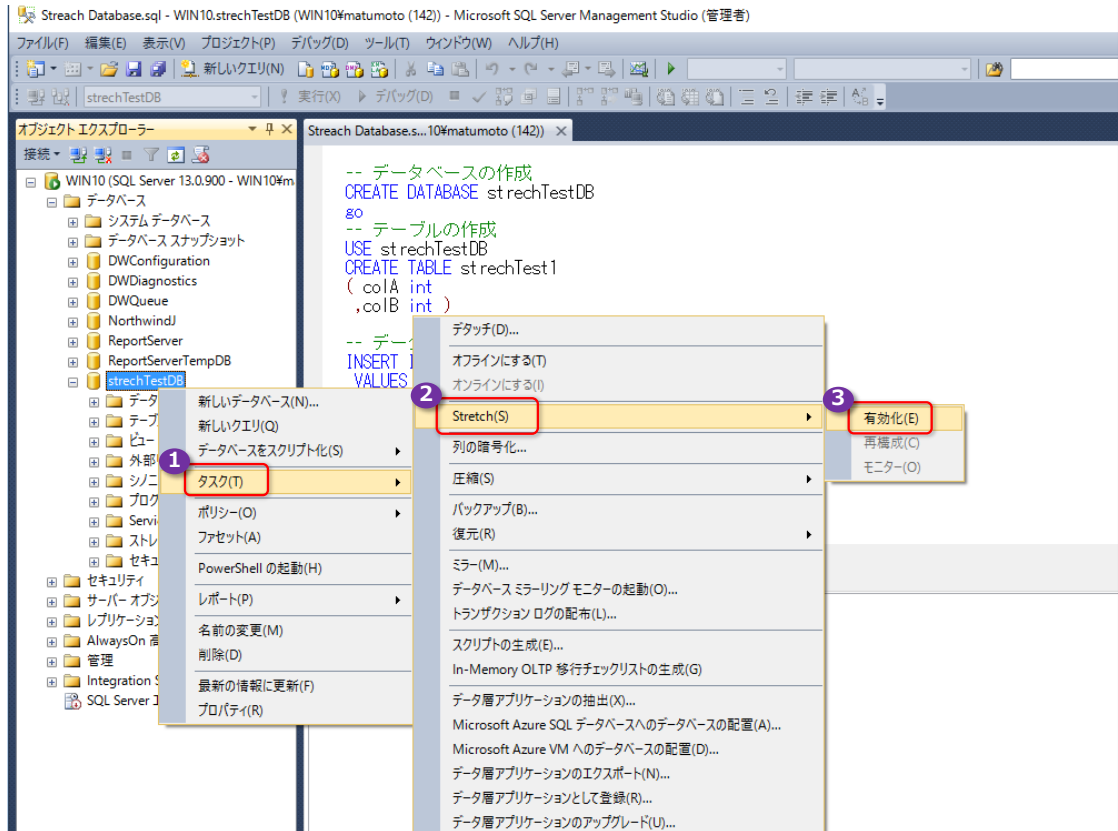
-- データを 2件追加
INSERT INTO strechTest1
VALUES (1, 111)
      , (2, 222)

-- データの確認
SELECT * FROM strechTest1
```

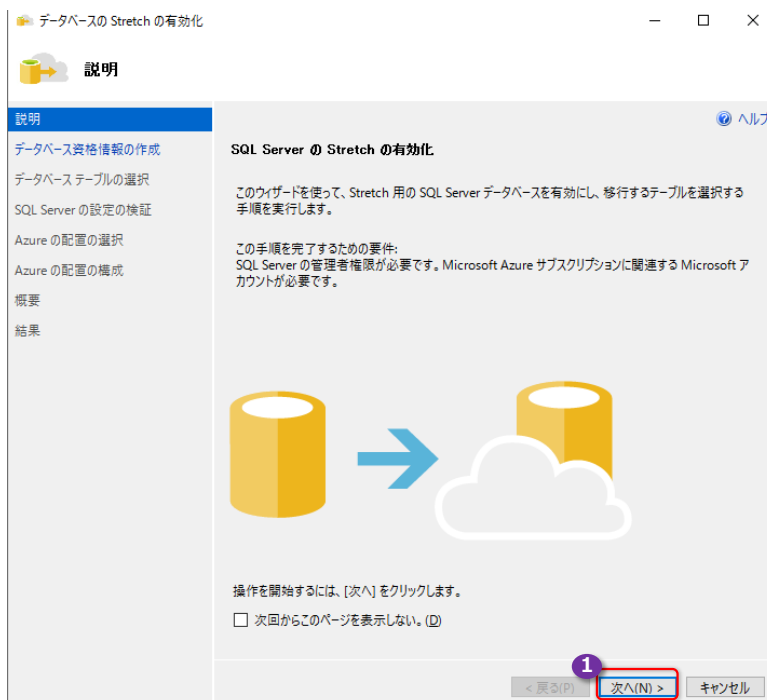


3. 次に、データベースを右クリックして、[タスク] メニューの [Strech] から [有効化] をクリックします。





4. 「データベースの Stretch の有効化」ウィザードが起動したら、「次へ」ボタンをクリックして、次のページに進みます。



5. 次の「データベース資格情報の作成」ページでは、データベースのマスター キーを作成するために、パスワードを入力します。

データベースの Stretch の有効化

データベース資格情報の作成

説明

データベース資格情報の作成

Stretch を構成するには、ソース SQL Server データベース資格情報をセキュリティで保護するデータベースマスター キーが必要です。

1 マスター キーのパスワードを入力してください。

パスワード \*\*\*\*\*

パスワードの確認 \*\*\*\*\*

強力なパスワードを作成するには、英数字の大文字、小文字および特殊文字を組み合わせて使用します。

2 次へ(N) >

ここでは、推測されにくい複雑なパスワードを入力するようにします。

6. 次の「**データベース テーブルの選択**」ページでは、どのテーブルをクラウド上に配置したいのかを選択します。

データベースの Stretch の有効化

データベース テーブルの選択

説明

データベース資格情報の作成

データベース テーブルの選択

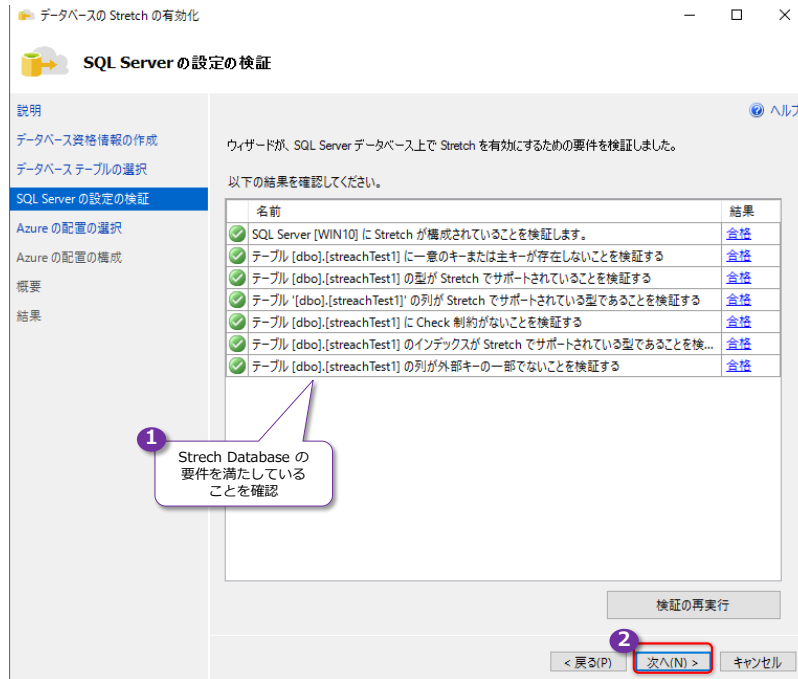
Azure にストレッチする SQL Server データベースからテーブルを選択します。

名前	Stretch が実行されました	行	サイズ (KB)	移行
<input checked="" type="checkbox"/> strechTest1	False	2	16	AllRows

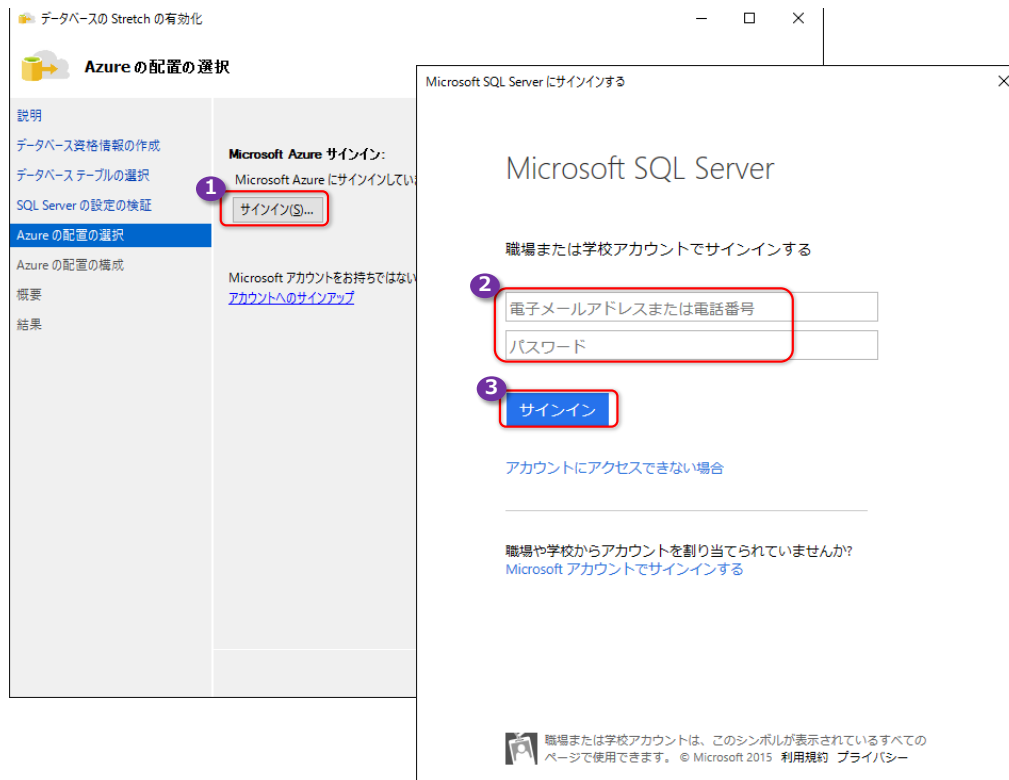
2 次へ(N) >

ここでは、前の手順で作成した「**strechTest1**」テーブルをチェックして、「**次へ**」ボタンをクリックします。

7. 次の「**SQL Server の設定の検証**」ページでは、該当テーブルが Stretch 可能かどうか（クラウド上に配置できるかどうか）がチェックされます。

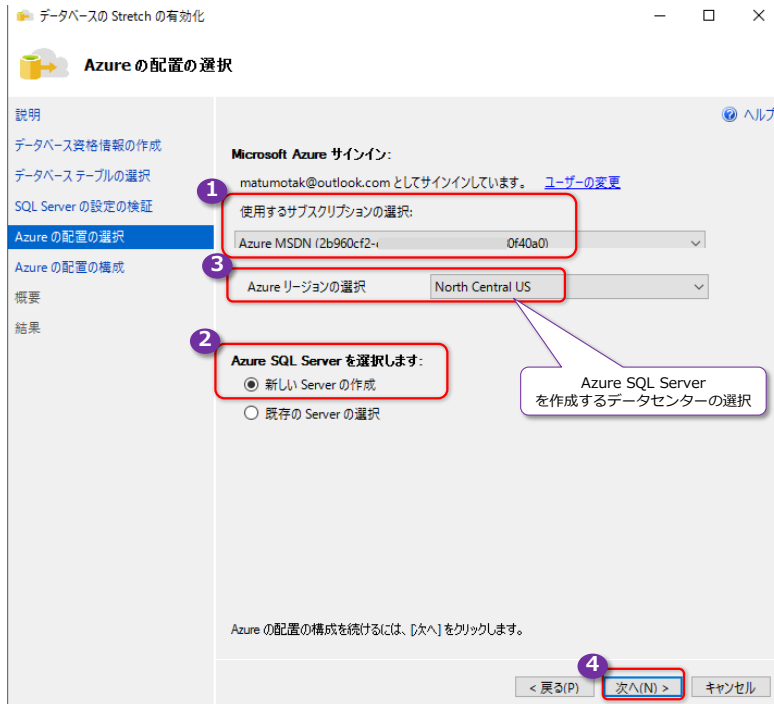


8. 次の「Azure の配置の選択」ページでは、「サインイン」ボタンをクリックします。



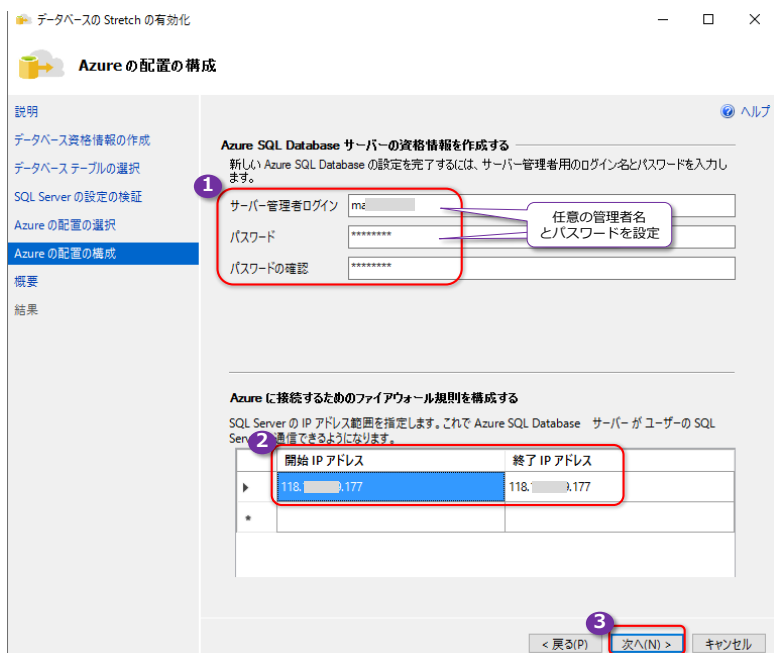
「Microsoft SQL Server にサインインする」ダイアログが表示されたら、Microsoft Azure を契約している Microsoft アカウントのメール アドレスとパスワードを入力して、「サインイン」ボタンをクリックします。

9. サインインが完了すると、次のように契約している Microsoft Azure のサブスクリプションが表示されます。



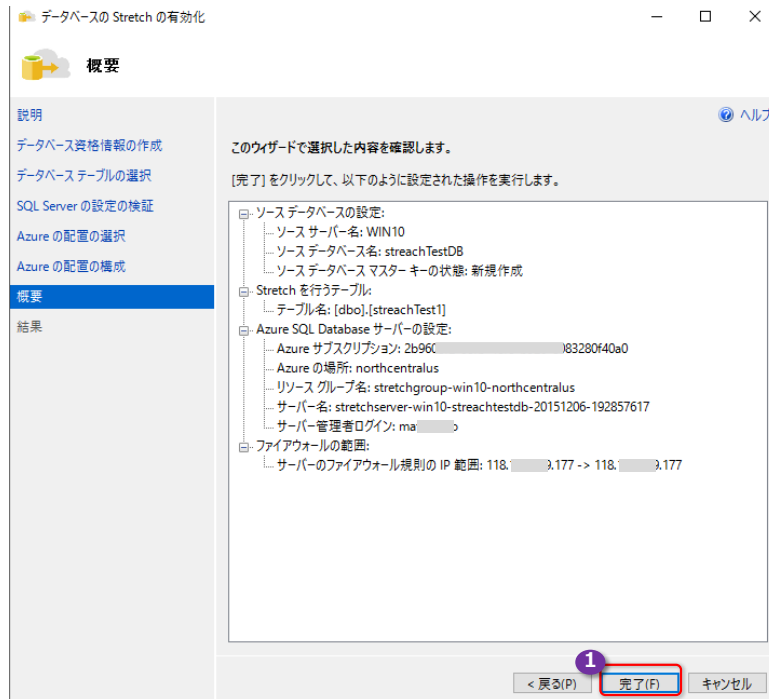
ここでは、Azure SQL Database 用のサーバー（Azure SQL Server）を新規作成するか、既存のサーバーを選択するかを選択します。サーバーを新規作成する場合は、**新しい Server の作成**、既存のサーバーを選択する場合は**既存の Server の選択**をクリックします。また、**Azure リージョンの選択**で、サーバーをどのデータセンターに作成するのか／既存のサーバーを利用する場合は、そのサーバーを作成しているデータセンターの場所を選択します（画面は **North Central US**：米国中北部 を選択）。なお、既定のデータセンターは「**Brazil South**」ですが、CTP 3.2 の日本語版では、このデータセンターを選択すると、作成が失敗してしまいます。ですので、**North Central US** などに変更しておくようにしてください。

10. 次の**「Azure の配置の構成」**ページでは、サーバー（Azure SQL Server）の管理者とパスワードを任意に設定します。



「**Azure に接続するためのファイアウォール規則を構成する**」では、現在利用している IP アドレスを「**開始 IP アドレス**」と「**終了 IP アドレス**」に入力して、「**次へ**」ボタンをクリックします（Azure SQL Database では、IP アドレスでのアクセス制限を行っていて、既定ではすべての IP アドレスからのアクセスが拒否されているので、許可をしたい IP アドレスをここで指定する形になります）。

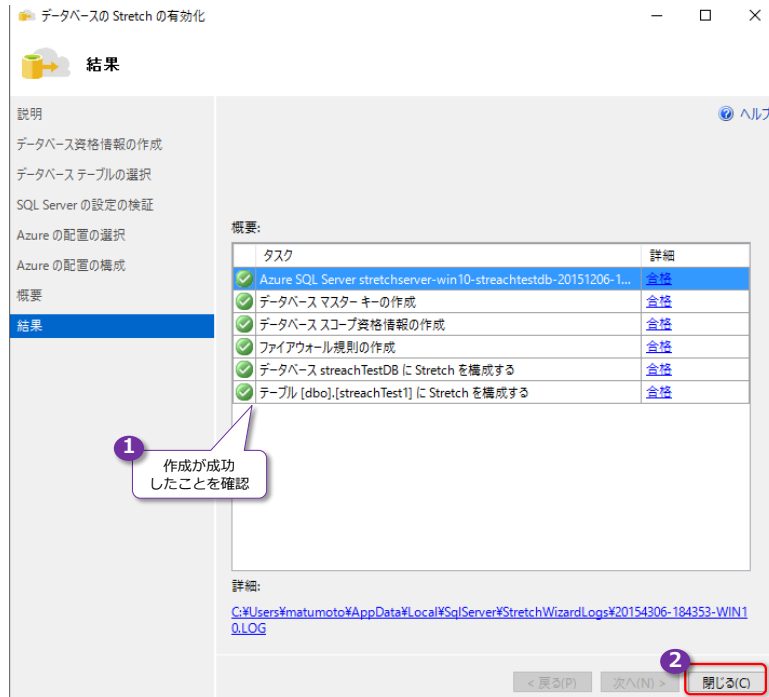
11. 次の「**概要**」ページでは、設定内容を確認して「**完了**」ボタンをクリックします。



これで、Stretch Database の作成（Azure SQL Database の作成や、Stretch Database の設定など）が始まり、設定中は、次のように「**配置状況**」ページが表示されます。



12. 設定が完了すると、次のように「**結果**」ページが表示されます。



すべてのタスクが **[合格]** と表示されれば、設定が完了です。

以上で、Strech Database の設定が完了です。これで、Microsoft Azure 上に Azure SQL Database（とサーバー）が自動的に作成されるので、以降では、これを確認していきましょう。

## ➡ Microsoft Azure 管理ポータルで自動作成されたデータベースの確認

まずは、Microsoft Azure 上に自動作成された Azure SQL Database を確認します。

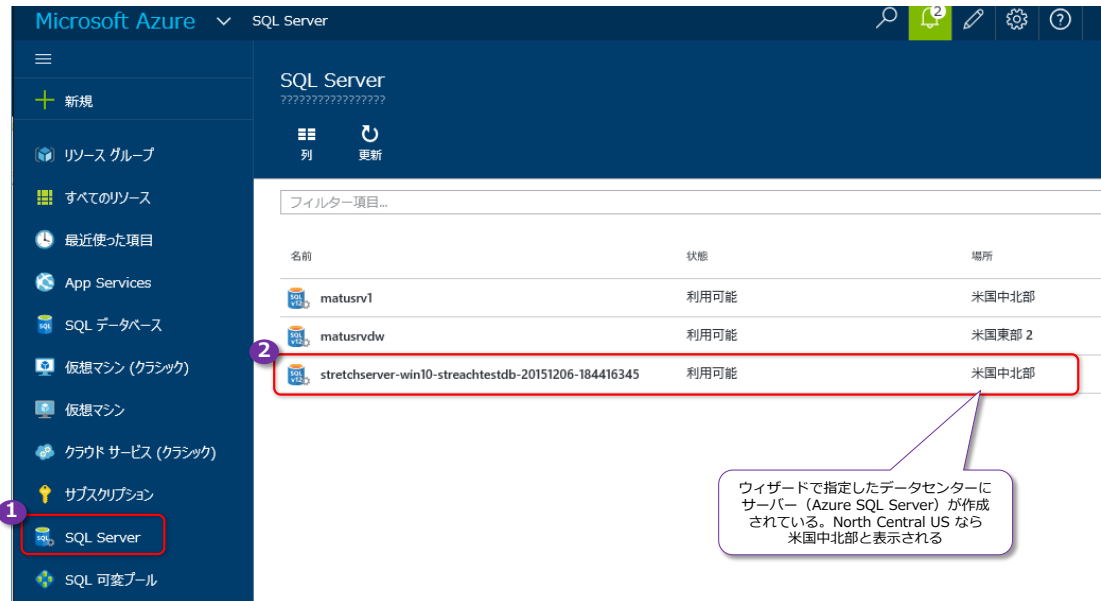
1. 自動作成された Azure SQL Database を確認するには、Azure 管理ポータルにアクセスします。これは、次の URL でアクセスできます。

Microsoft Azure 管理ポータル

<http://portal.azure.com/>

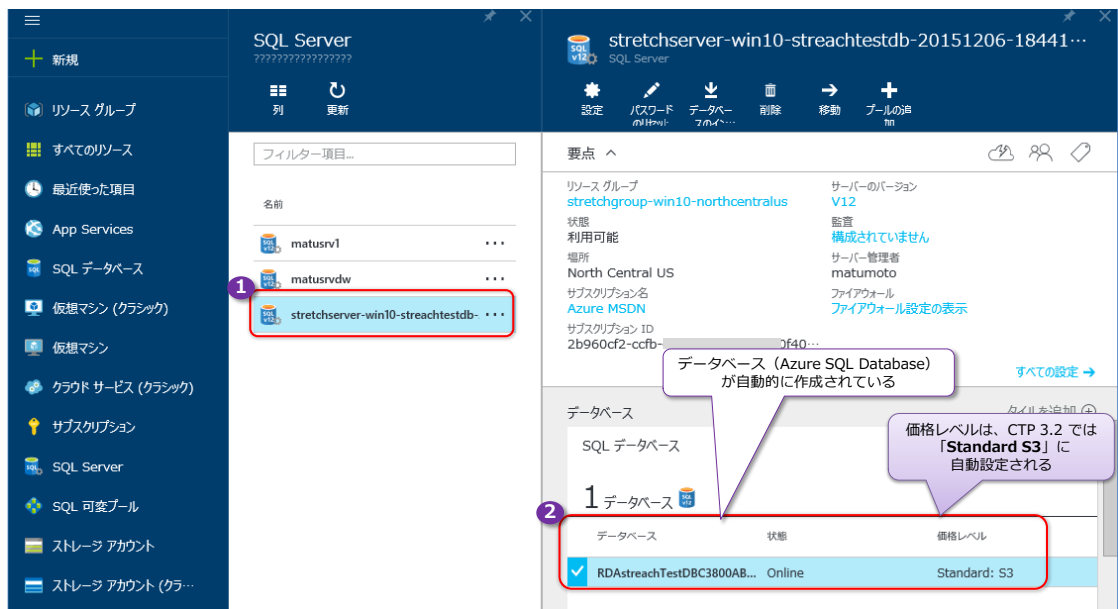


2. 管理ポータルが開いたら、[SQL Server] をクリックして、Azure SQL Database 用のサーバー（Azure SQL Server）を確認します。



「stretchserver-マシン名-データベース名-日付-～」という名前のサーバーが自動的に作成されていることを確認できます。

3. 次に、このサーバーをクリックすると、サーバー内に作成されたデータベース（Azure SQL Database）を確認することができます。



「RDA データベース名～」という名前のデータベースが自動的に作成されて、[価格レベル] が [Standard S3] に設定されていることを確認できます。Azure SQL Database では、1 時間単位での課金になり、価格レベルが「S3」で、データセンターが「米国中北部（North Central US）」の場合は、1 時間あたり 20.57 円です。価格は、データセンターの場所によって異なり、「東日本（Japan East）」を選択している場合は、S3 だと 1 時間あたり 23.28 円になります。



Azure SQL Database の価格については、次の URL で確認できます。

SQL Database の価格

<http://azure.microsoft.com/ja-jp/pricing/details/sql-database/>

価格の詳細

Elastic Database Single Database 「Single Database」をクリック

リージョン: 米国中北部 通貨: 日本円 (¥) データセンターを選択

Single database モデル

単一データベースは、パフォーマンスの需要がある程度予測できるワークロード用に最適化された、完全に分離されたデータベースです。シングル データベースは、Basic、Standard、Premium の各サービス レベルの間でスケールアップやスケールダウンが可能です。これにより、アプリのニーズに合わせたパフォーマンスと機能を、ちょうど必要なときに活用することができます。各レベルの主な違いは、データベース トランザクション ユニッ (DTU) によって測定されるパフォーマンスにあります。詳細については「サービス レベル」を参照してください。

Basic

	DTU	DB あたりの最大ストレージ 容量	料金 *
B	5	2 GB	¥0.69/時間 (~ ¥510/月)

Standard

	DTU	DB あたりの最大ストレージ 容量	料金 *
S0	10	250 GB	¥2.07/時間 (~ ¥1,530/月)
S1	20	250 GB	¥4.12/時間 (~ ¥3,060/月)
S2	50	250 GB	¥10.29/時間 (~ ¥7,650/月)
S3	100	250 GB	¥20.57/時間 (~ ¥15,300/月)

S3 の場合 の価格

「Single Database」をクリックして、「リージョン」でデータセンターを選択することで、価格を確認することができます。

4. Azure 管理ポータルでは、該当データベースをクリックして、次のように「価格レベル (DTU のスケール)」をクリックすると、価格を確認 (ひと月あたりの価格) することもできます。

RDAtreachTestDBC3800AB2-C618-467C-A464-4667...

設定 RDAtreachTestDBC3800AB2-C618-467C-A46...

価格レベルの変更

データベース スケールアップ (DTU) は、Basic データベース、Standard データベース、および Premium データベース のパフォーマンス レベルの増強を意味します。これは、CPU、メモリ、読み取り、および書き込みのパフォーマンスを総合し た測定に基づいています。詳細情報

	47,430.00 JPY/MONTH (ESTIMATED 31 P...)	94,860.00 JPY/MONTH (ESTIMATED 31 P...)	189,720.00 JPY/MONTH (ESTIMATED 31 P...)
P6 Premium	10... DTUs	P11 Premium	S0 Standard
Up to 500 GB	Up to 1024 GB	Up to 250 GB	
Active Geo-Replic...	Active Geo-Replic...	Standard Geo-Rep...	
Point In Time Res...	Point In Time Res...	Point In Time Res...	
Auditing	Auditing	Auditing	
379,440.00 JPY/MONTH (ESTIMATED 31 P...)	714,011.22 JPY/MONTH (ESTIMATED 31 P...)	1,530.09 JPY/MONTH (ESTIMATED 31 P...)	
S1 Standard	S2 Standard	S3 Standard	
20 DTUs	50 DTUs	100 DTUs	
Up to 250 GB	Up to 250 GB	Up to 250 GB	
Standard Geo-Rep...	Standard Geo-Rep...	Standard Geo-Rep...	
Point In Time Res...	Point In Time Res...	Point In Time Res...	
Auditing	Auditing	Auditing	
3,059.87 JPY/MONTH (ESTIMATED 31 S...)	7,652.04 JPY/MONTH (ESTIMATED 31 S...)	15,299.97 JPY/MONTH (ESTIMATED 31 S...)	

価格レベルが「Standard S3」の場合の金額を確認できる

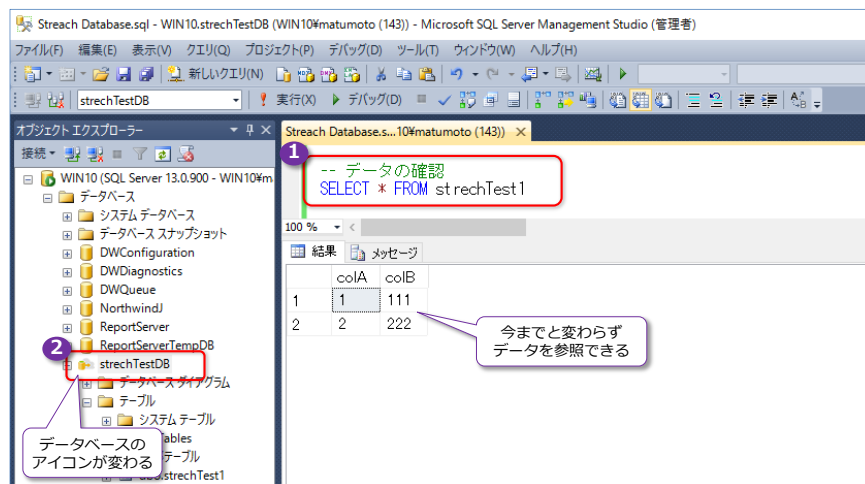
また、このページでは、価格レベルを変更することもできます。S3 の場合は、データセンターが「**米国中北部 (North Central US)**」の場合に、ひと月あたり 15,299.97 円かかるので、Strech Database を試した後は、必ずデータベースを削除しておくようにしておいてください。削除方法については後述します。なお、データベースを削除せずに、B (Basic) レベルに価格を変更した場合は、ひと月利用したとしても、約 510 円で済みます。

## 動作の確認

次に、Strech Database の動作を確認してみましょう。

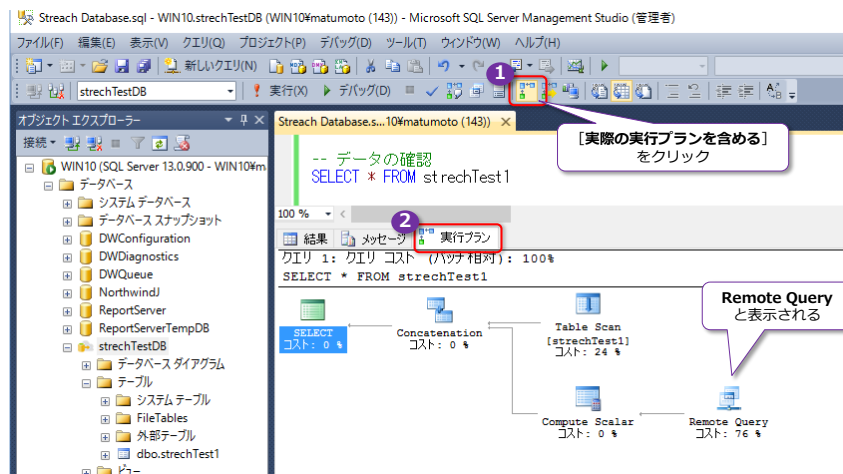
1. Strech Database では、テーブル データはクラウド上 (Azure SQL Database 内) に自動的に配置されますが、ユーザーは、それを意識することなく、(ローカルにデータがあるかのよう) に データを利用することができます。次のように **SELECT** ステートメントを実行して、データを参照してみましょう。

```
SELECT * FROM stretchTest1
```



Strech Database を設定する前と同様、同じデータを参照できることを確認できます。

2. 次に、ツールバーの**[実際の実行プランを含める]**をクリックして、実行プランを確認します。

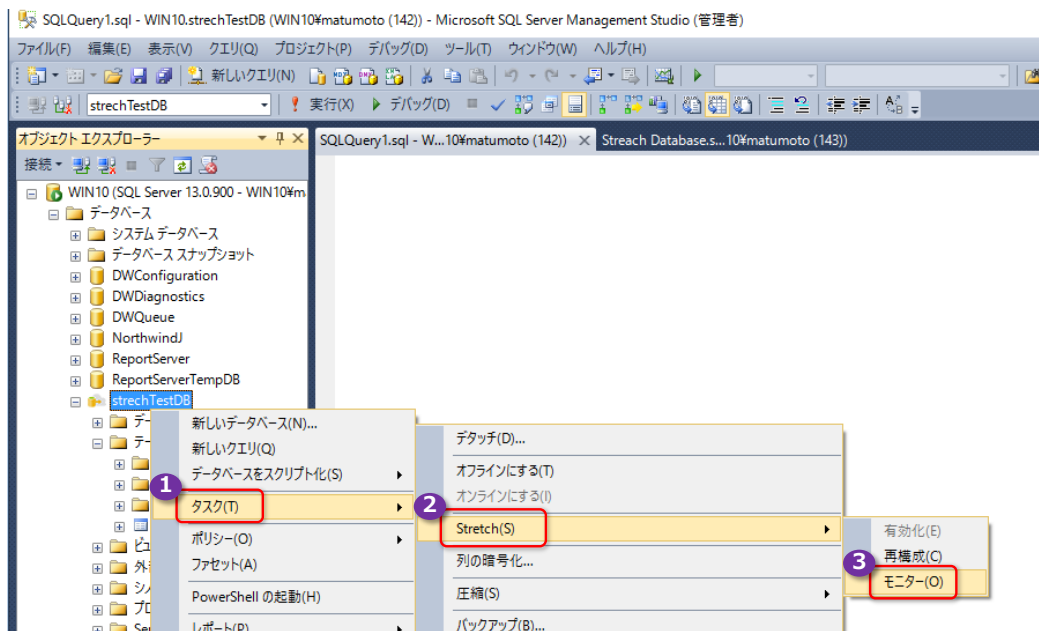


Strech Database を設定すると、[Remote Query] と表示されて、リモートのデータソースからデータを取得したことを確認できます。

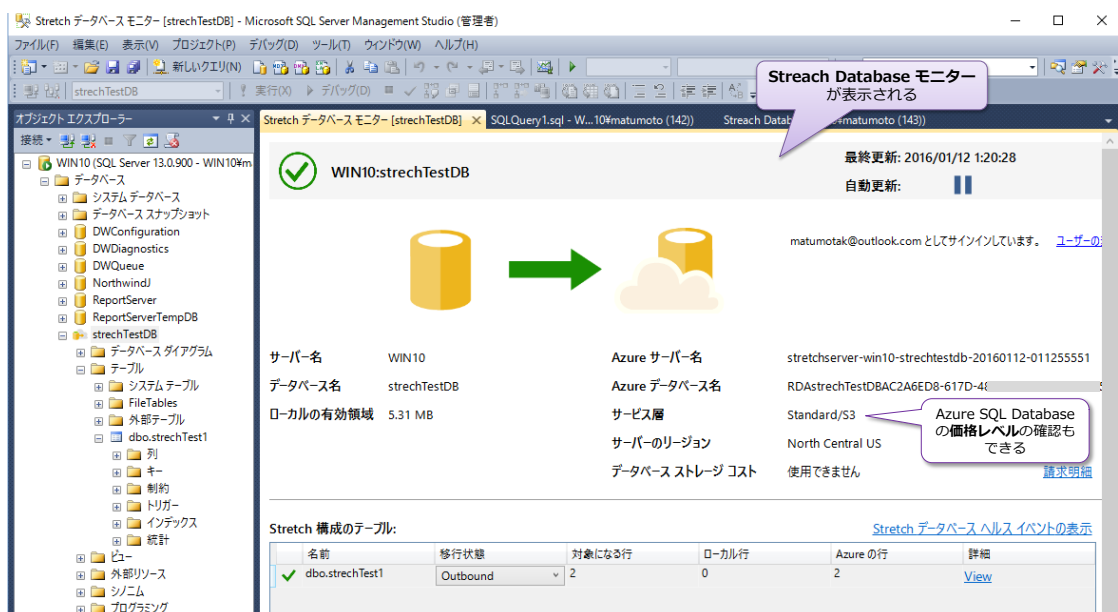
## ➡ Strech Database モニターで確認

Strech Database を設定すると、[Strech Database モニター] というツールが追加されて、状態（接続状況など）を確認することができます。

1. [Strech Database モニター] を開くには、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 [Strech] から [モニター] をクリックします。



2. [Strech Database モニター] は、次のように表示されます。

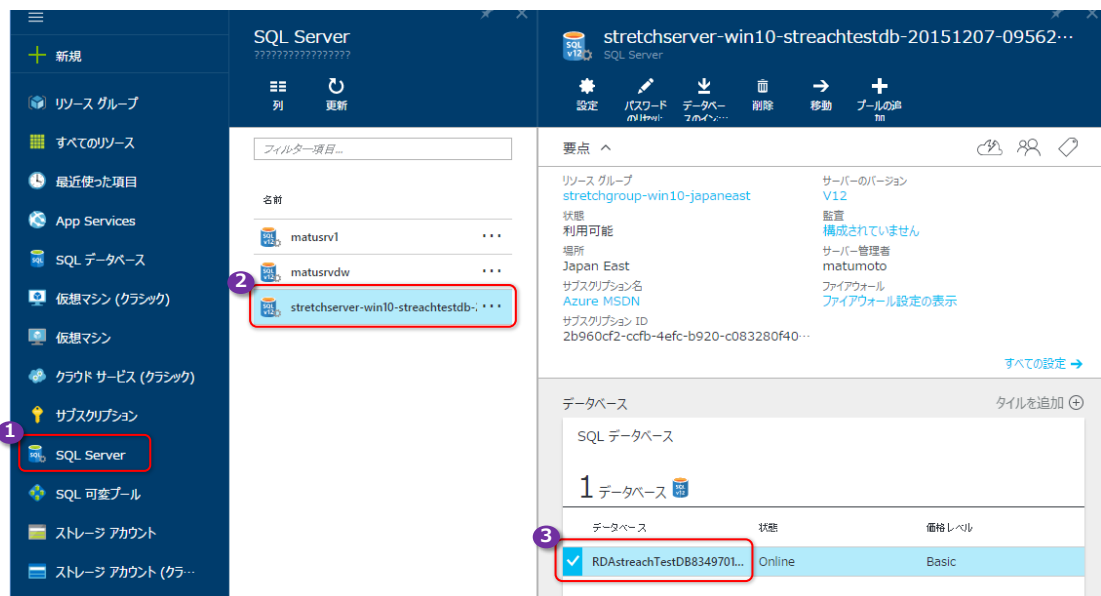


動作が正常の場合は、緑の矢印のアイコンが表示されます。

## ➡ Azure 管理ポータルでデータベースを削除

Stretch Database の動作を確認したら、Microsoft Azure 上からデータベース（Azure SQL Database）を削除しておきましょう（削除しておかないと、課金が続くことになるので注意してください）。

1. Azure SQL Database のデータベースを削除するには、まず、Azure の管理ポータル（<http://portal.azure.com/>）にアクセスします。
2. 管理ポータルが開いたら、[SQL Server] をクリックして、サーバー（Azure SQL Server）の一覧を表示して、Stretch Database 用のサーバー（stretchserver-マシン名-データベース名-日付-~ 形式のもの）をクリックします。



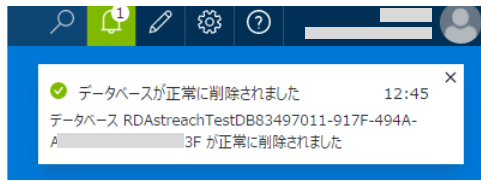
サーバー内のデータベースの一覧が表示されたら、Stretch Database 用のデータベース（RDA データベース名~ 形式のもの）をクリックします。

3. 次のようにデータベースの構成が表示されたら [削除] をクリックします。



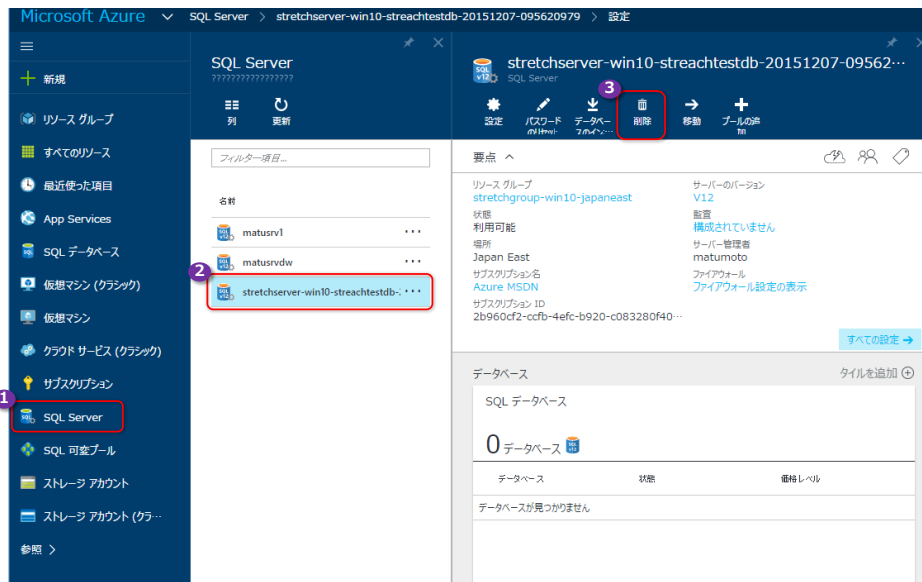
[このデータベースを完全に削除しますか?] と表示されたら、[はい] をクリックします。こ

れで削除が開始されて、削除が正常に完了すると、画面の右上に、次のように「データベースが正常に削除されました」と表示されます。

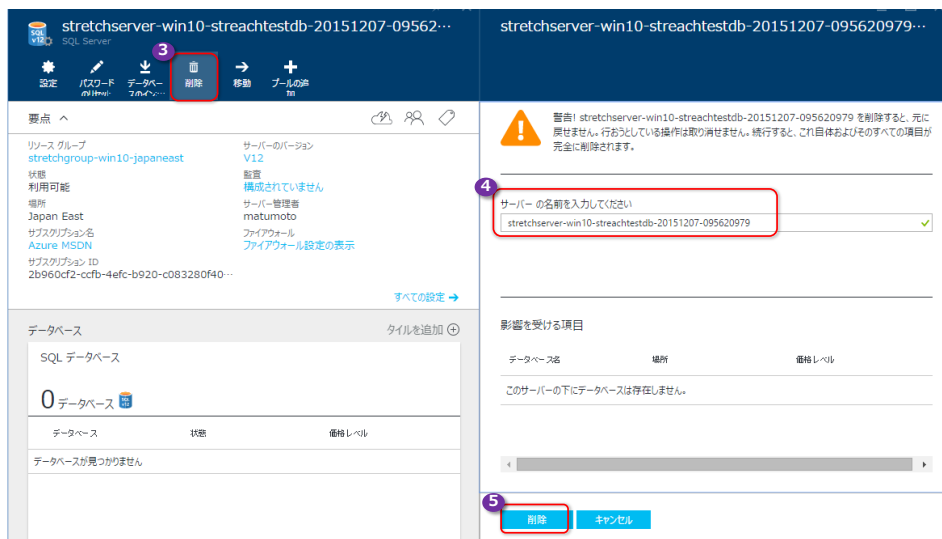


これが確認できれば、データベースが完全に削除されているので、これ以上課金されることはなくなります。

4. データベースの削除ができれば、サーバー（Azure SQL Server）が残っていても、課金されることはなくなりますが、サーバーも削除したい場合は、次のように操作します。サーバーを表示した画面で「削除」をクリックします。



サーバーの削除の場合は、確認のために、次のように「サーバー名」を入力して、「削除」ボタンをクリックします。



以上で、サーバー（Azure SQL Server）の削除が完了です。

その他、Stretch Database 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。

### Stretch Database

<https://msdn.microsoft.com/en-us/library/dn935011.aspx>

The screenshot shows the 'Stretch Database' page in the Microsoft Docs library. On the left is a navigation pane with a tree structure: 'Databases (Database Engine)' is expanded, and 'Stretch Database' is selected. Below it, a list of topics is shown, including 'Identify databases and tables for Stretch Database by running Stretch Database Advisor', 'Surface area limitations and blocking issues for Stretch Database', 'Enable Stretch Database for a database', 'Enable Database for Stretch Wizard', 'Reconfigure Stretch Database', 'Monitor Stretch Database', 'Enable Stretch Database for a table', 'Unmigrate remote data (Stretch Database)', 'Disable Stretch Database', 'Pause and resume Stretch Database', and 'Manage and troubleshoot Stretch Database'. The main content area has the title 'Stretch Database' and 'SQL Server 2016'. It states 'Applies To: SQL Server 2016 Preview'. The text explains that Stretch Database archives historical data transparently and securely in the Microsoft Azure cloud. It lists three benefits: reduced cost and complexity, faster local queries and database operations, and seamless access to both local and remote data. A section titled 'What does Stretch Database do?' describes the migration process and the retry logic for connection issues.

► Databases (Database Engine)

▼ Stretch Database

- Identify databases and tables for Stretch Database by running Stretch Database Advisor
- Surface area limitations and blocking issues for Stretch Database
- Enable Stretch Database for a database
- Enable Database for Stretch Wizard
- Reconfigure Stretch Database
- Monitor Stretch Database
- Enable Stretch Database for a table
- Unmigrate remote data (Stretch Database)
- Disable Stretch Database
- Pause and resume Stretch Database
- Manage and troubleshoot Stretch Database

## Stretch Database

### SQL Server 2016

Applies To: SQL Server 2016 Preview

Stretch Database archives your historical data transparently and securely. In SQL Server 2016 Community Technology Preview 3 (CTP 3.0), Stretch Database stores your historical data in the Microsoft Azure cloud. After you enable Stretch Database, it silently migrates your historical data to an Azure SQL Database.

Stretch Database provides the following benefits:

- You typically enjoy reduced cost and complexity.
- Your local queries and database operations against current data typically run faster.
- You don't have to change existing queries and client apps. You continue to have seamless access to both local and remote data.

### What does Stretch Database do?

After you enable Stretch Database for a local server instance, a database, and at least one table, it silently begins to migrate your historical data to an Azure SQL Database. You can pause data migration to troubleshoot problems on the local server or to maximize the available network bandwidth.

Stretch Database ensures that no data is lost if a failure occurs during migration. It also has retry logic to handle connection issues that may occur during migration. A dynamic management view provides the status of migration.

## 4.4 Azure バックアップ URL の性能向上（ブロック BLOB 対応）

SQL Server では、バックアップを Microsoft Azure の BLOB ストレージ上に配置する機能が SQL Server 2012 の SP1 CU2 から利用することができましたが、これは BLOB ストレージ内の「ページ BLOB」にファイルを格納していました。SQL Server 2016 からは、従来と同様「ページ BLOB」に格納する方法に加えて、「ブロック BLOB」に格納する方法が追加されました。

ページ BLOB は「ランダム読み取りと書き込みに最適化された 512 バイト単位のストレージ」、ブロック BLOB は「複数のブロックを並列操作することで大きなファイルを効率よく管理できるストレージ」という特徴があります。

### ➡ ページ BLOB を利用したバックアップ（従来ながらの利用方法）

従来の SQL Server と同様、ページ BLOB にバックアップを行う場合は、次のように利用します。

```
CREATE CREDENTIAL 資格情報名
WITH IDENTITY = 'ストレージ アカウント名',
SECRET = 'ストレージ アカウントのアクセス キー'

BACKUP DATABASE sampleDB
TO URL = 'https://ストレージ アカウント名.blob.core.windows.net/コンテナ名/~/bak'
WITH CREDENTIAL = '資格情報名', COMPRESSION
```

**CREATE CREDENTIAL** ステートメントで、**IDENTITY** に Microsoft Azure 上の**ストレージ アカウントの名前**、**SECRET** にストレージ アカウントにアクセスするための**アクセス キー**を指定して、資格情報を作成します。この資格情報を利用して（**CREDENTIAL** 句で資格情報名を指定して）、**BACKUP** ステートメントを実行すれば、**TO URL** 句で指定した Microsoft Azure 上の BLOB ストレージ（コンテナ）内にページ BLOB としてバックアップを行うことができます。

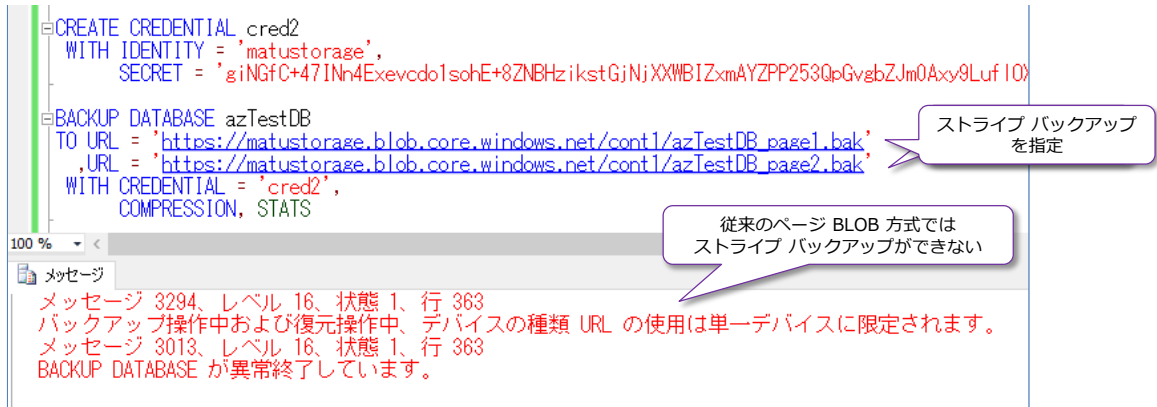
**WITH** で指定している **COMPRESSION** は、バックアップ圧縮を行うための設定で、必須ではありませんが、バックアップ ファイルを圧縮することでネットワーク上を流れるデータ量を減らすことができるので、バックアップにかかる実行時間を短縮することができます。

なお、ページ BLOB では、1 つのファイルの最大サイズが「**1TB**」までという制限があるので、バックアップ ファイルのサイズは 1TB 未満に抑える必要があります。

これに対して、後述のブロック BLOB を利用したバックアップでは、1 つのファイルあたりは「**200GB**」のサイズに制限されますが、**ストライプ バックアップ**（複数のファイルに並列でバックアップ処理）ができるので、1TB 以上のサイズでもバックアップが可能です（最大は **12.8TB** までのストライプ合計サイズがサポートされます）。

ページ BLOB では、ストライプ バックアップがサポートされていないので、これを行おうとすると、次のようにエラーになります。





## ➡ ブロック BLOB へのバックアップ（SQL Server 2016 からの新機能）

SQL Server 2016 からは、**ブロック BLOB** へのバックアップおよび**ストライプ バックアップ**がサポートされたことで、ページ BLOB にバックアップするよりも性能向上、より大きなサイズへの対応（最大サイズの上限が 1TB から 12.8TB へ）ができるようになりました。

ブロック BLOB へバックアップを行うには、**SAS**（Shared Access Signature：共有アクセス署名）を利用して、次のように実行します。

```
CREATE CREDENTIAL [https://ストレージ アカウント名.blob.core.windows.net/コンテナ名]
WITH IDENTITY='Shared Access Signature'
, SECRET='SAS'

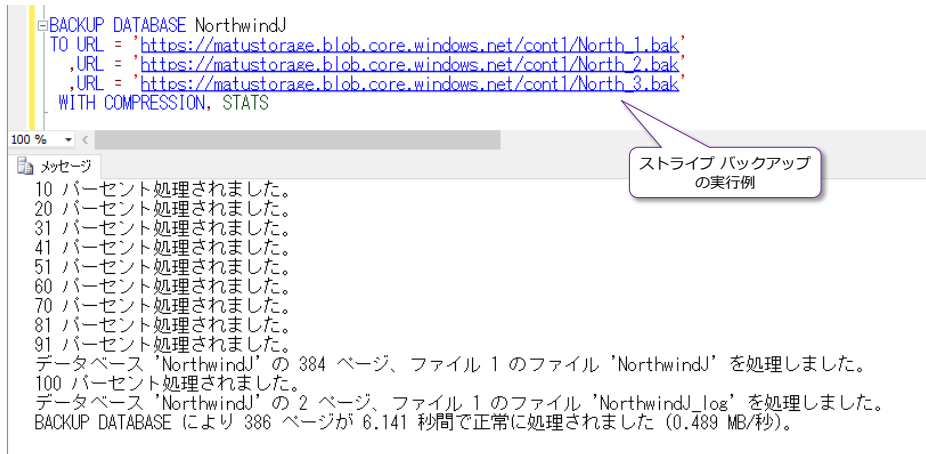
BACKUP DATABASE データベース名
TO URL = 'https://ストレージ アカウント名.blob.core.windows.net/コンテナ名/~.bak'
WITH COMPRESSION
```

**CREATE CREDENTIAL** ステートメントで、**IDENTITY** に「Shared Access Signature」と記述して、**SECRET** に **SAS**（Shared Access Signature：共有アクセス署名）を指定します（SAS の作成方法については後述します）。また、資格情報名には、ストレージ アカウントのコンテナ名までの URL を [ ] で囲んで指定します。

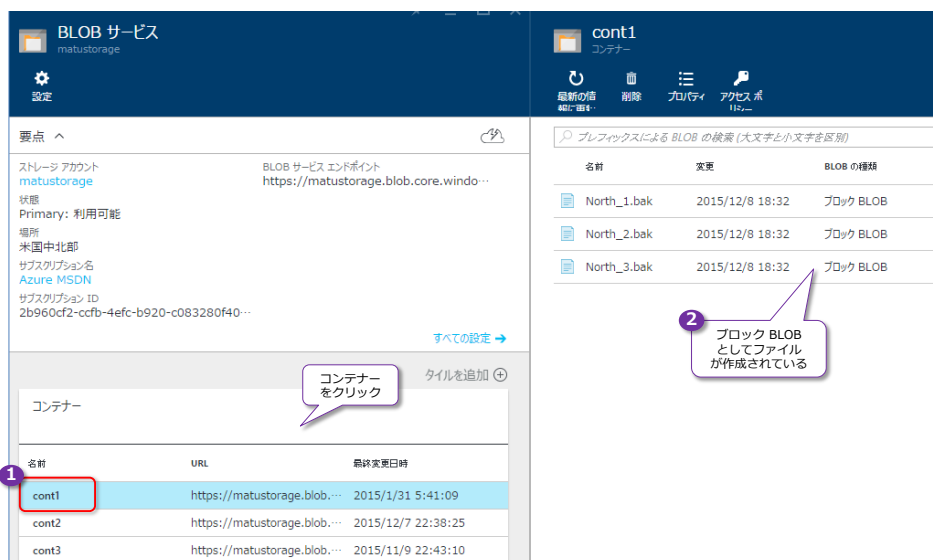
**BACKUP** ステートメントでは、**TO URL** に指定する BLOB の**コンテナ名**と **CREATE CREDENTIAL** で指定した**コンテナ名**を同じにすることで、SAS（共有アクセス署名）を利用したアクセスができます。また、このように、バックアップをすることで、ブロック BLOB にバックアップすることができます。

SAS を利用したバックアップでは、ストライプ バックアップもサポートされるので、次のように **TO URL** で指定するファイル パスを複数指定（カンマで区切って指定）することができます。

```
BACKUP DATABASE データベース名
TO URL = 'https://ストレージ アカウント名.blob.core.windows.net/コンテナ名/~_1.bak'
, URL = 'https://ストレージ アカウント名.blob.core.windows.net/コンテナ名/~_2.bak'
, URL = 'https://ストレージ アカウント名.blob.core.windows.net/コンテナ名/~_3.bak'
WITH COMPRESSION
```



ブロック BLOB としてバックアップされたことを確認するには、Azure の管理ポータルで、該当ストレージ アカウントのコンテナーを表示すれば、次のように確認することができます。



このように、ブロック BLOB にストライプ バックアップを行うことで、弊社環境では、2 億件のデータベースのバックアップにかかった時間が、ページ BLOB では **2 分 35 秒**だったところを、**1 分 43 秒**に短縮することができました (**33.5%** の性能向上)。簡易的な検証だったので、ストライプ数を 10 個~50 個に変更したりしましたが、ネットワーク帯域のボトルネックとの兼ね合いもあるので、最適なストライプ数 (何個のファイルに分割すれば良いか) は状況によって変わってくると思うので、ぜひいろいろなパターンを試してみてください。

## ➡ SAS (Shared Access Signature : 共有アクセス署名) の作成方法

SAS (共有アクセス署名) を作成する一番簡単な方法は、Codeplex で提供されている「**Azure Storage Explorer**」ツールを利用する方法です。これは、次の URL からダウンロードできます。

<http://azurestorageexplorer.codeplex.com/>

CodePlex Project Hosting for Open Source Software Register Sign In Search all projects

## Azure Storage Explorer

HOME SOURCE CODE DOWNLOADS DOCUMENTATION DISCUSSIONS ISSUES PEOPLE LICENSE

Page Info | Change History (all pages) ★ Follow (431) | Subscribe

**New: Azure Storage Explorer 6 Preview 3 (August 2014)**

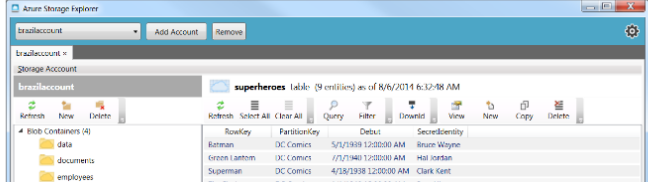
Now available in the Downloads section  
Modern code base, updated for latest cloud platform, APIs, and developer tools.  
Preview 3 has blob, queue and table support.  
Walk-through: <http://davidpallmann.blogspot.com/2014/08/azure-storage-explorer-preview-3-now.html>

Search Wiki & Documentation

**downloads**

CURRENT Azure Storage Explorer 6 Preview 3  
DATE Fri Aug 15, 2014 at 5:00 PM  
STATUS Beta  
DOWNLOADS 138,634  
RATING ★★★★★ 32 ratings  
[Review this release](#)

**MOST HELPFUL REVIEWS**



**Azure Storage Explorer** ツールでは、次のように操作することで、SAS を生成することができます。

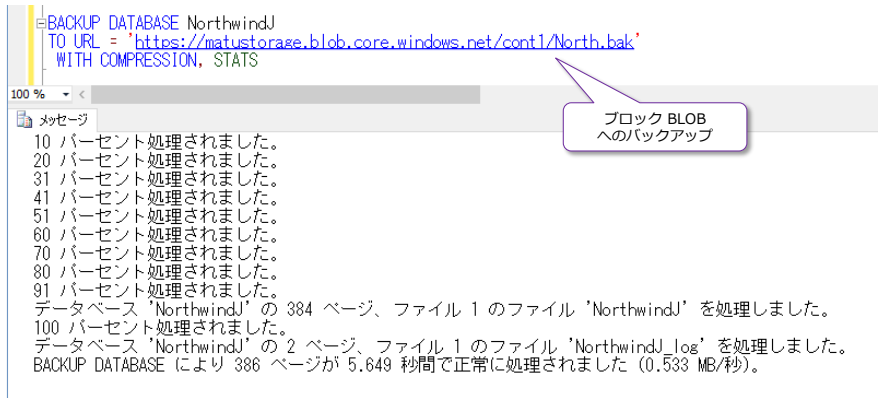
1. 「Add Account」でストレージアカウントに接続
2. バックアップ先となるコンテナを選択
3. 「Security」をクリック
4. 「Shared Access Signatures」をクリック
5. 「Read」と「Write」をチェックして書き込み権限を付与
6. 「Generate Signature」をクリック
7. sv= の部分をコピーする

【Generate Signature】で生成した **SAS** (sv= の部分) を、**CREATE CREDENTIAL** ステートメントの **SECRET** に貼り付ければ、次のようにブロック BLOB へのバックアップを行えるようになります。

```
-- 出力された sv=以降をコピーして SECRET に貼り付け
CREATE CREDENTIAL [https://matustorage.blob.core.windows.net/cont1]
WITH IDENTITY='Shared Access Signature'
, SECRET='sv=2014-02-14&sr=c&sig=KdUtNlp4a7%2F0HslzahdF1cPcf rzwipVJP81009gqZCI%3D&st=2014-07-15T15%3A00Z&se=2015-12-15T15%3A00Z&sp=rw'
```

メッセージ  
コマンドは正常に完了しました。

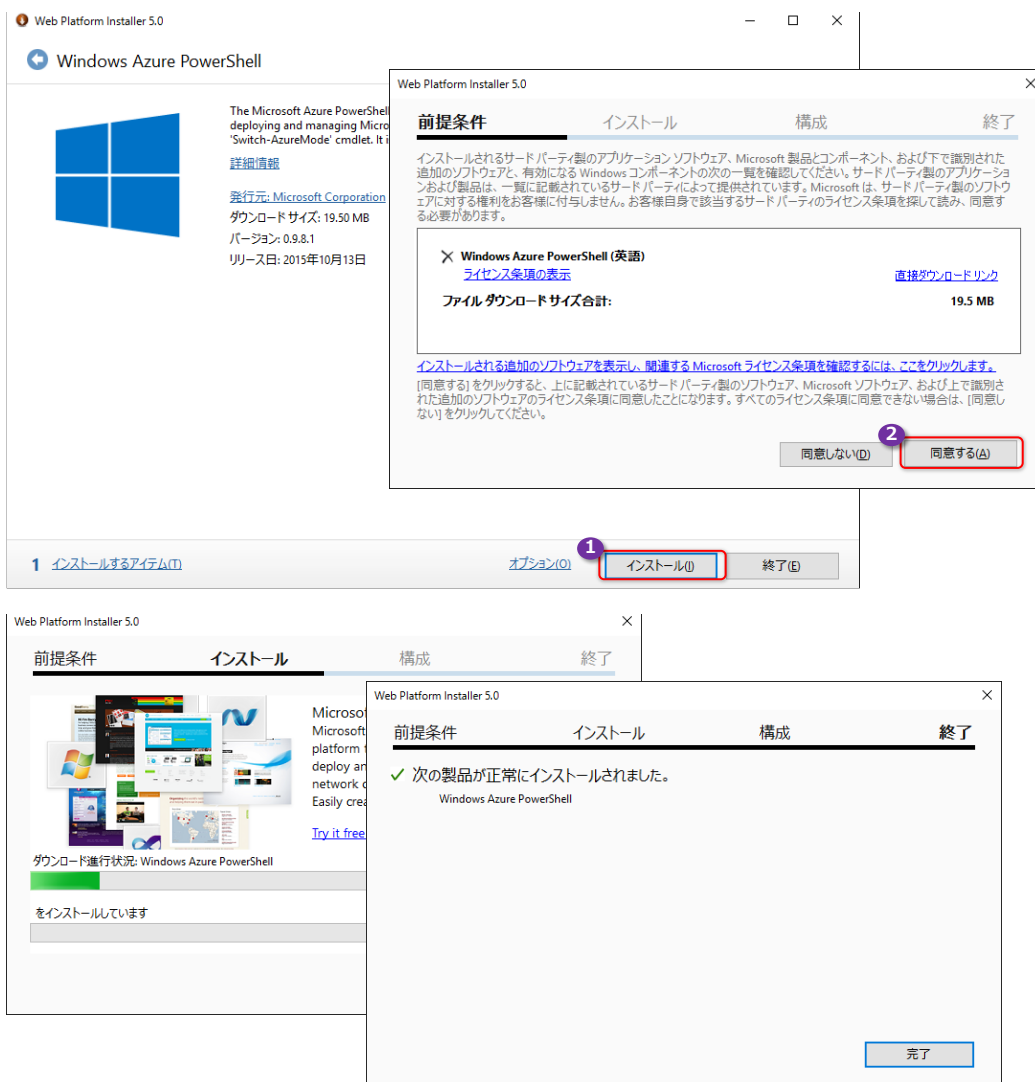
生成した SAS を貼り付け



➡ **Azure PowerShell** を利用して、**SAS** を生成する場合

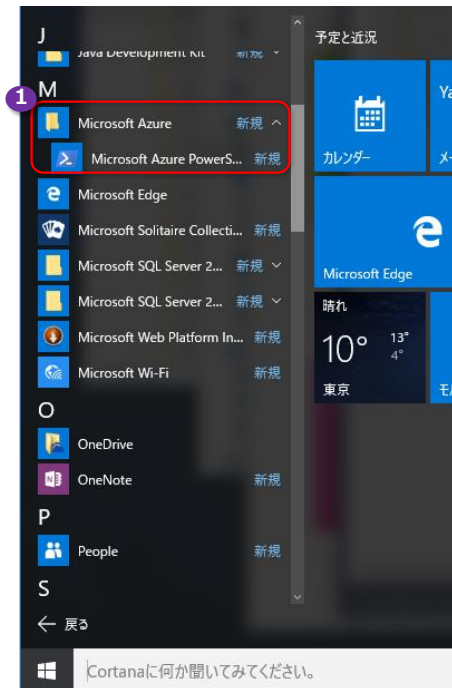
SAS は、**Azure PowerShell** を利用して生成することもできます。**Azure PowerShell** は、次の URL からダウンロード／インストールすることができます。

<http://go.microsoft.com/fwlink/p/?linkid=320376&clid=0x411>



インストール完了後は、[スタート] メニューの [すべてのアプリ] から [Microsoft Azure] の

[**Microsoft Azure PowerShell**] をクリックして、起動することができます。



Azure PowerShell が起動したら、次のようにコマンドを記述すれば、SAS を生成することができます。

```
$context = New-AzureStorageContext -StorageAccountName ストレージ アカウント名 -StorageAccountKey アクセスキー  
New-AzureStorageContainerSASToken -Name コンテナ名 -Permission rwdl -FullUri -Context $context
```

**New-AzureStorageContext** で、「**-StorageAccountName**」に**ストレージ アカウントの名前**を指定して、「**-StorageAccountKey**」に**ストレージ アカウントにアクセスするためのアクセス キー**を指定します。

次に、**New-AzureStorageContainerSASToken** で「**-Name**」でバックアップ先となるコンテナ名を入力します。

```
管理者: Microsoft Azure PowerShell
Windows PowerShell
Copyright (C) 2015 Microsoft Corporation. All rights reserved.

For a list of all Azure cmdlets type 'help azure'.
For a list of Windows Azure Pack cmdlets type 'Get-Command *wapack*'.
For Node.js cmdlets type 'help node-dev'.
For PHP cmdlets type 'help php-dev'.
PS C:\>
PS C:\> $context = New-AzureStorageContext -StorageAccountName matustorage -StorageAccountKey giNGfC+47...:ikstC...
PS C:\>
PS C:\> New-AzureStorageContainerSASToken -Name cont1 -Permission rwdl -FullUri -Context $context
https://matustorage.blob.core.windows.net/cont1?sv=2015-02-21&sr=c&sig=0rdKqo5LR0K0ncXY5N8ZHcJZZhc%2Bw50eKcApobF1zIg%3D&se=2015-12-
PS C:\>
```

Azure PowerShell  
での SAS の生成例

sv= の  
部分をコピーする

生成された SAS (sv= の部分) を、**CREATE CREDENTIAL** ステートメントの **SECRET** に貼り付ければ、ブロック BLOB へのバックアップを行えるようになります。

Azure PowerShell でのコマンド例や、バックアップ URL の詳細については、オンライン ブック

の以下のトピックが参考になると思います。

## SQL Server Backup to URL

<http://msdn.microsoft.com/en-us/library/dn435916.aspx>

- ▼ SQL Server Backup to URL
  - Use PowerShell to Backup Multiple Databases to Microsoft Azure Blob Storage Service
  - ▶ SQL Server Backup to URL Best Practices and Troubleshooting
- File-Snapshot Backups for Database Files in Azure
- ▶ Managed Backup to Microsoft Azure
- Restoring From Backups Stored in Microsoft Azure

### Create a Shared Access Signature

The following examples create Shared Access Signatures that can be used to create a SQL Server Credential on a newly created container. The first example creates an ad hoc Shared Access Signature and the second example creates a Shared Access Signature that is associated with a Stored Access Policy. For more information, see [Shared Access Signatures, Part 1: Understanding the SAS Model](#). These examples require Microsoft Azure PowerShell. For information about installing and using Azure PowerShell, see [How to install and configure Azure PowerShell](#).

- Ad hoc Shared Access Signature**

```
$context = New-AzureStorageContext -StorageAccountName <mystorageaccountname> -StorageAccountKey <mystorageaccountkey>
New-AzureStorageContainer -Name <mystorageaccountcontainername> -Context $context
New-AzureStorageContainerSASToken -Name <mystorageaccountcontainername> -Permission rwdl -Fu
```
- Shared Access Signature that is associated with a Stored Access Policy**

```
$SubscriptionName='mysubscriptionname'
$StorageAccountName='mystorageaccountname'
$ContainerName='mystorageaccountcontainername'
Select-AzureSubscription -SubscriptionName $SubscriptionName
$AccountKeys = Get-AzureStorageKey -StorageAccountName $StorageAccountName
$StorageContext = New-AzureStorageContext -StorageAccountName $StorageAccountName -StorageAccountKey $AccountKeys
$Container = New-AzureStorageContainer -Context $StorageContext -Name $ContainerName
$Cbc = $Container.CloudBlobContainer
$Permissions = $Cbc.GetPermissions();
$PolicyName = 'policy1'
$Policy = new-object 'Microsoft.WindowsAzure.Storage.Blob.SharedAccessBlobPolicy'
$Policy.SharedAccessStartTime = $(Get-Date).ToUniversalTime().AddMinutes(-5)
$Policy.SharedAccessExpiryTime = $(Get-Date).ToUniversalTime().AddYears(5)
$Policy.Permissions = "Read,Write,List,Delete"
$Permissions.SharedAccessPolicies.Add($PolicyName, $Policy)
$Cbc.SetPermissions($Permissions);
$Policy = new-object 'Microsoft.WindowsAzure.Storage.Blob.SharedAccessBlobPolicy'
$Sas = $Cbc.GetSharedAccessSignature($Policy, $PolicyName)
Write-Host 'Shared Access Signature= '$($Sas.Substring(1))'
Write-Host 'Credential T-SQL'
$TSQL = "CREATE CREDENTIAL [{0}] WITH IDENTITY='Shared Access Signature', SECRET='{1}'"
Write-Host $TSQL
```

## 4.5 PolyBase で Hadoop アクセス (Hortonworks、HDInsight)

**PolyBase** は、**Hadoop ファイル システム** (HDFS: Hadoop File System) 上にあるデータを、SQL Server からアクセスできる機能です (この機能は、元々 SQL Server Parallel Data Warehouse: PDW で提供されていた機能ですが、SQL Server 2016 からは、通常の SQL Server でも利用できるようになりました)。

PolyBase を利用すれば、Microsoft Azure 上の Hadoop サービスである「**Microsoft Azure HDInsight**」や、Hortonworks の提供する「**HDP: Hortonworks Data Platform**」、Cloudera の提供する「**CDH: Cloudera's Distribution including Apache Hadoop**」上のデータに対して、SQL Server からアクセスできるようになります。また、PolyBase では、Microsoft Azure 上の「**BLOB ストレージ**」にもアクセスすることができます。

### ➡ PolyBase の利用イメージ ～CREATE EXTERNAL TABLE～

PolyBase では、**CREATE EXTERNAL TABLE** ステートメントを利用することで、外部データ ソースである HDFS/Hadoop サービス上にあるファイル (1 つまたは複数のファイル) を、SQL Server 上のテーブルであるかのように操作できるようになります (外部データ ソースに対して、テーブル定義を割り当てることができ、SQL Server 上のテーブルのように扱えます)。

次の画面は、PolyBase を利用して、IP アドレス 192.168.1.36 のマシン (CentOS 7 をインストールして、Hortonworks Data Platform: HDP で HDFS を稼働中の Linux マシン) にアクセスしているときの様子です。

The screenshot shows the Microsoft SQL Server Management Studio interface. The left pane displays the 'Object Explorer' with the 'testdb1' database selected. The right pane shows the 'Query Editor' with the following SQL script:

```
EXEC sp_configure 'hadoop connectivity', 7;
RECONFIGURE

CREATE DATABASE testdb1
GO
USE testdb1

CREATE EXTERNAL DATA SOURCE hdp1
WITH ( TYPE = HADOOP,
      LOCATION = 'hdfs://192.168.1.36:8020' )

CREATE EXTERNAL FILE FORMAT f1
WITH ( FORMAT_TYPE = DELIMITEDTEXT,
      FORMAT_OPTIONS ( FIELD_TERMINATOR = ',',
                      ,USE_TYPE_DEFAULT = TRUE ) )

CREATE EXTERNAL TABLE test1
( a int,
  b int )
WITH ( LOCATION = '/input/test1',
      DATA_SOURCE = hdp1,
      FILE_FORMAT = f1 )

SELECT * FROM test1
```

At the bottom, the 'Results' pane shows the output of the query:

	a	b
1	1	111
2	2	222



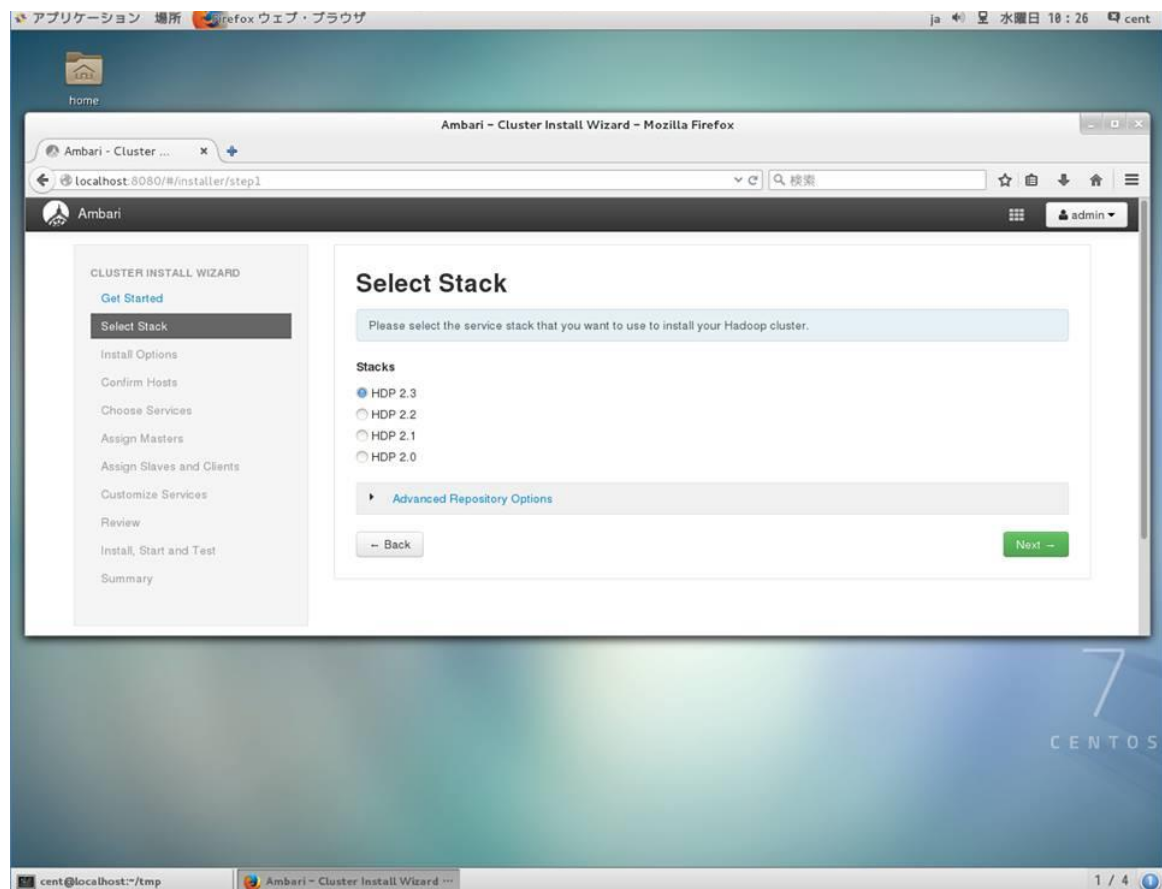
**CREATE EXTERNAL DATA SOURCE** ステートメントで「**TYPE = HADOOP**」と指定して、**LOCATION** に IP アドレスと HDFS アクセスのためのポート番号（HDP インストール時の既定値の **8020** を利用）して、外部データ ソースを定義しています。

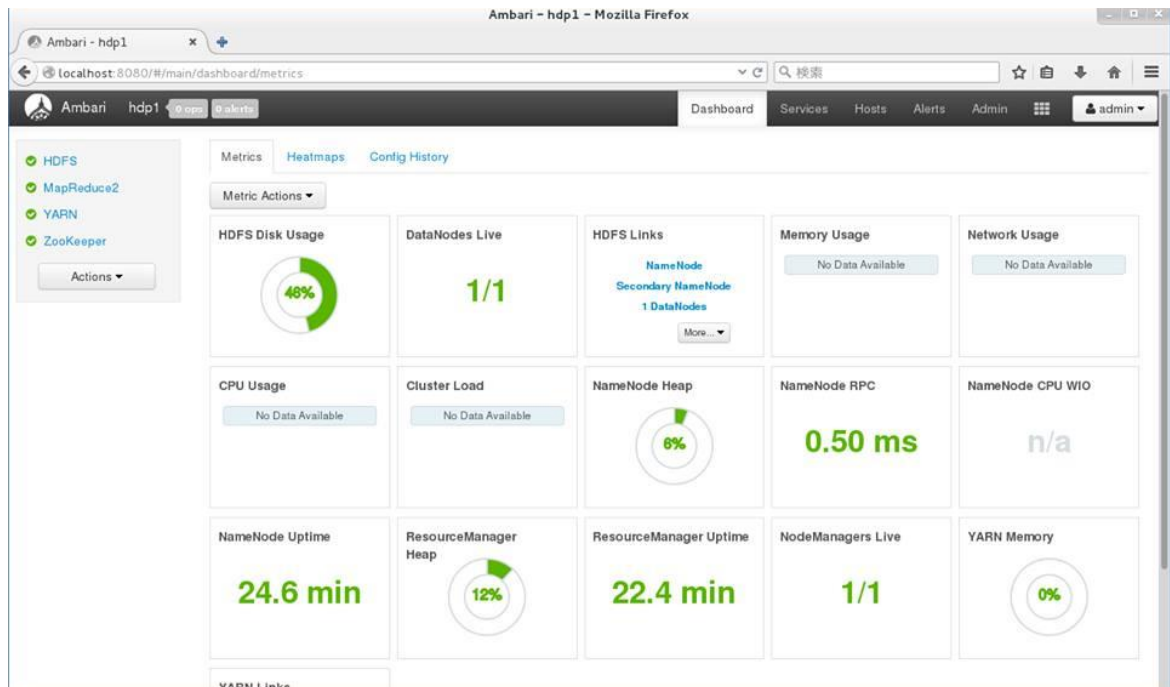
次に、**CREATE EXTERNAL FILE FORMAT** ステートメントでファイルのフォーマット（上の例では **FIELD\_TERMINATOR** に **,** を指定して **CSV ファイル**を指定）しています。

最後に、**CREATE EXTERNAL TABLE** ステートメントで、**DATA\_SOURCE** に最初に作成した外部データ ソース、**FILE\_FORMAT** に上で定義したファイル フォーマットを指定して、**LOCATION** に HDFS 上のファイルまたはフォルダーへのパス（上の例では **input** フォルダー配下の **test1** ファイル）を指定することで、HDFS 上のファイル（パスにフォルダーを指定した場合は、フォルダー配下のファイル群）を、SQL Server のテーブルであるかのように利用することができます。

このようにテーブル定義を設定することで、外部データ ソースであるにも関わらず、SQL Server 上に存在するテーブルであるかのように操作でき、「**SELECT \* FROM <外部データ ソース>**」のように **SELECT** ステートメントでアクセスできるようになります。

なお、この例で利用した Hortonworks Data Platform（HDP）は、次のようにインストールしています。





```

hdfs@localhost:~
ファイル(F) 編集(E) 表示(V) 検索(S) 端末(T) ヘルプ(H)
[hdfs@localhost ~]$ cat test1
1,111
2,222
[hdfs@localhost ~]$ hadoop fs -put test1 hdfs://192.168.1.36:8020/input/
[hdfs@localhost ~]$ hadoop fs -ls hdfs://192.168.1.36:8020/input/
Found 1 items
-rw-r--r--  3 hdfs hdfs          12 2015-11-12 03:44 hdfs://192.168.1.36:8020/in
put/test1
[hdfs@localhost ~]$

```

**test1 ファイル (CSV 形式)**

**HDFS (Hadoop ファイル システム) に test1 ファイルを配置 (put)**

**CentOS 7 ターミナル ツールで HDP を操作**

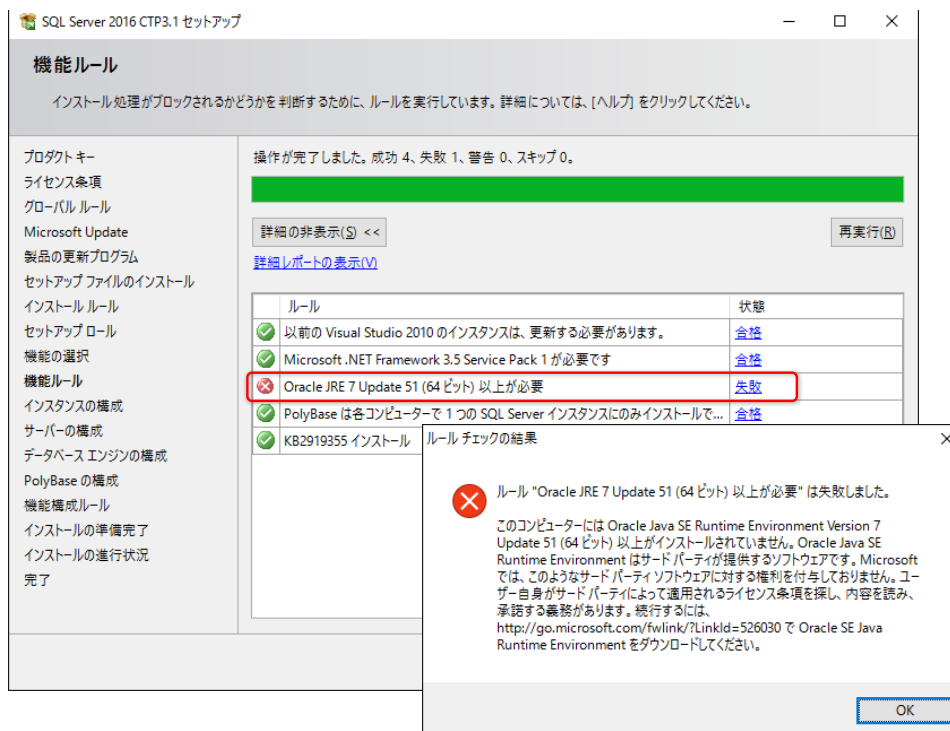
## ➡ PolyBase を利用する前提条件

SQL Server 2016 CTP 3.2 で PolyBase を利用するには、次の前提条件があります。

- **x64** 版の SQL Server 2016 CTP 3.2 (Evaluation edition)
- .NET Framework 4.0
- 64 ビット版の Oracle JRE (Java SE RunTime Environment) version 7.51 以上
- 最小メモリ要件 4GB、最小 HDD 容量要件 2GB

PolyBase は、**64 ビット環境**のみでサポートされて、**64 ビット版の JRE 7.51** (JRE 7 Update 51) 以上が必要になります。

**JRE 7.51** 以上をインストールしていない場合には、SQL Server 2016 のインストール時の「**機能ルール**」ページで、次のようにエラーが表示されて、先に進むことができません。



**JRE** (Java SE Runtime Environment) は、執筆時点での最新版である「**JRE 8u66**」や、Java の開発者キットである **JDK** (Java Development Kit) の「**JDK 8u66**」を入れておけば、PolyBase をインストール／動作させることができます (**64 ビット版の JRE 7 Update 51** 以上を利用することで動作させられます)。これらは、次の URL からダウンロードできます。

Java SE Runtime Environment のダウンロード URL (JRE 8u66 など)

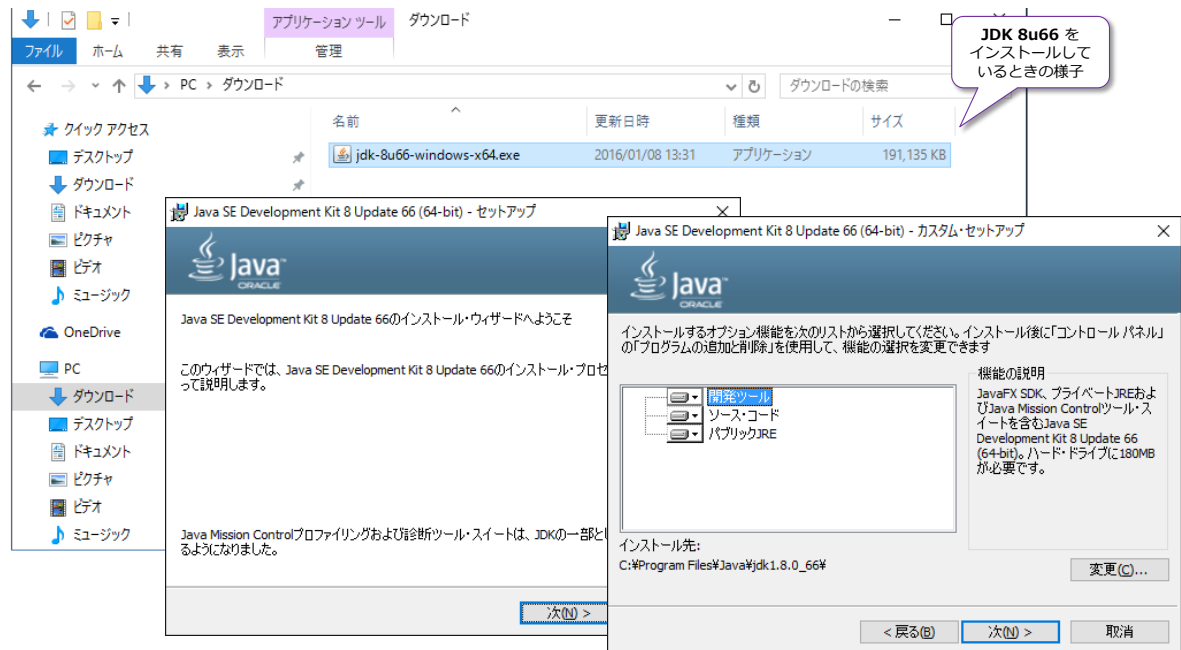
<http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html>

Java Development Kit のダウンロード URL (JDK 8u66 など)

<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

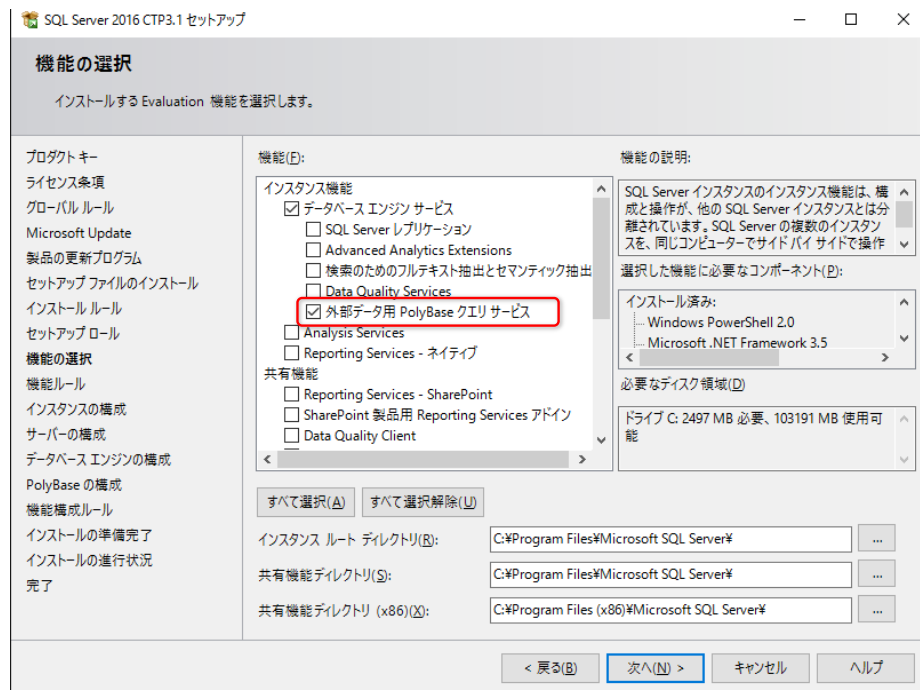
ダウンロード後は、.exe ファイルをダブルクリックすれば、JRE/JDK をインストールすること

ができます。

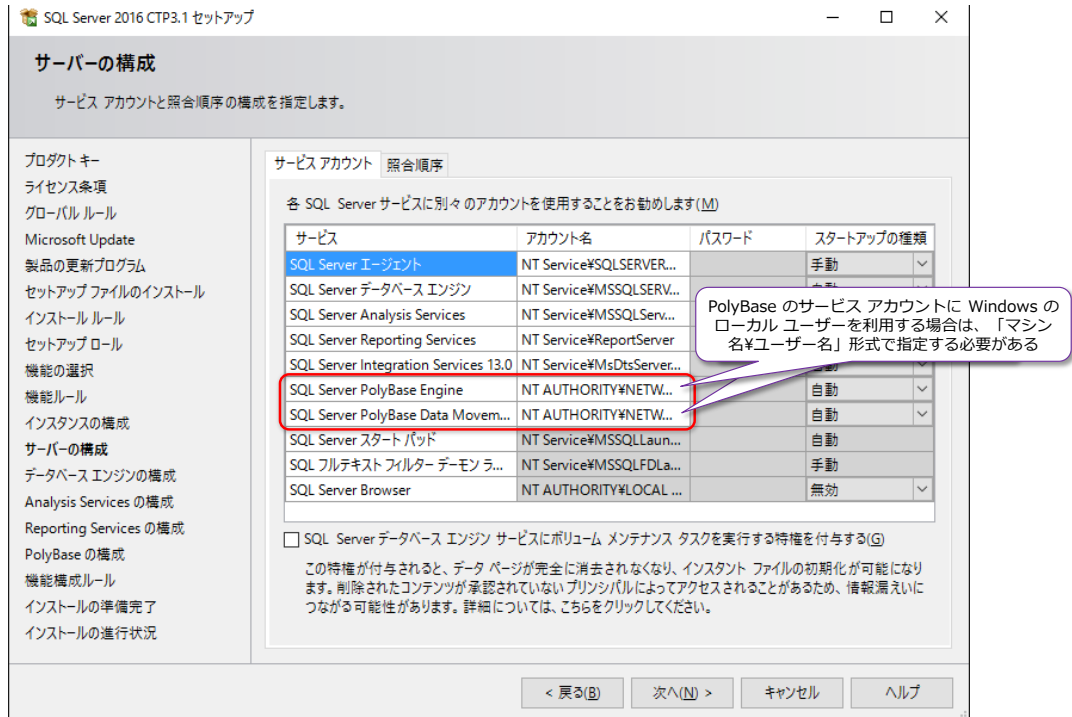


## ➡ PolyBase のインストール方法

PolyBase をインストールするには、SQL Server 2016 CTP 3.2 のインストール時に、次のように「機能の選択」ページで、「データベース エンジン サービス」の「外部データ用 PolyBase クエリ サービス」をチェックします。

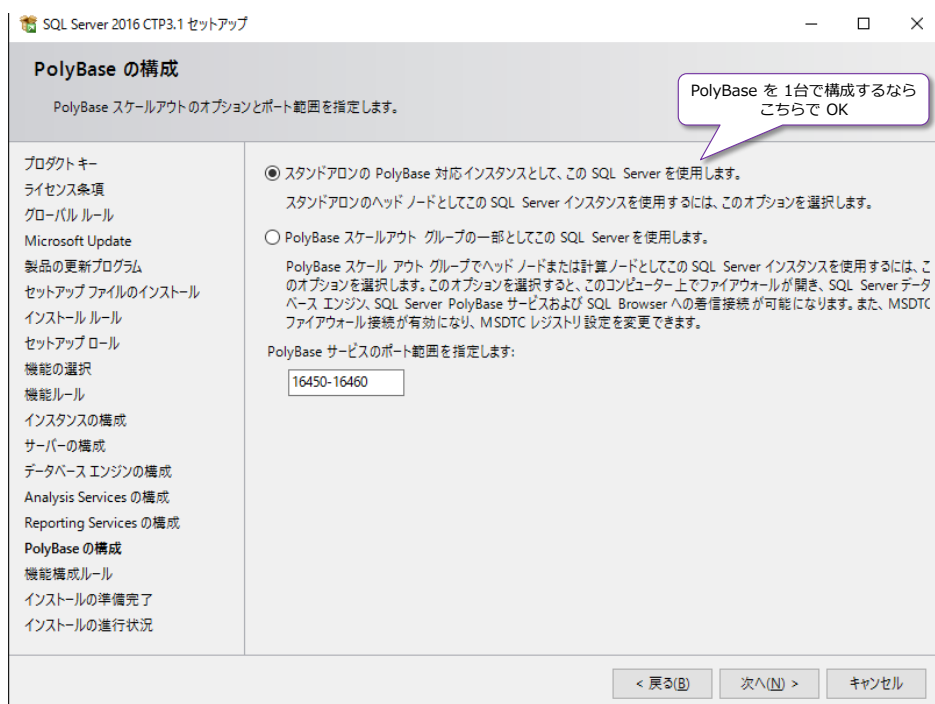


SQL Server 2016 のインストール時に PolyBase を選択した場合の注意点は、次の「サーバーの構成」ページの「サービス アカウント」タブでのサービス アカウントの設定画面です。



PolyBase は、2 つのサービスとして追加されますが、既定ではサービス アカウントとして「**NT AUTHORITY¥NETWORK SERVICE**」が選択されています。このサービス アカウントを、Windows のローカル ユーザーに変更する場合には、「**マシン名¥ユーザー名**」形式で（ユーザー名だけでなく、¥マーク付きでマシン名を付与して）指定する必要があります（ユーザー名だけだとインストールの最後のフェーズでエラーになってしまうので注意してください）。

また、PolyBase を選択している場合は、SQL Server 2016 のインストール時に、次のように「**PolyBase の構成**」ページが表示されます。



このページでは、PolyBase を 1 台で構成するのか、複数台（スケールアウト）で構成するのかを

設定しますが、1 台で構成する場合には[**スタンドアロンの PolyBase 対応インスタンスとして、SQL Server を使用します。**] をチェックすれば大丈夫です。

## ➡ PolyBase の利用方法

PolyBase を利用するには、まず **sp\_configure** で「**hadoop connectivity**」構成オプションを設定します。

```
EXEC sp_configure 'hadoop connectivity', 構成値
RECONFIGURE
```

構成値	接続可能な HDFS/Hadoop サービス
0	Hadoop 接続の無効化
1	Hortonworks HDP 1.3 on Windows Server、 Azure blob storage (WASB[S])
2	Hortonworks HDP 1.3 on Linux
3	Cloudera CDH 4.3 on Linux
4	Hortonworks HDP 2.0 on Windows Server、 Azure blob storage (WASB[S])
5	Hortonworks HDP 2.0 on Linux
6	Cloudera 5.1 on Linux
7	Hortonworks 2.1 and 2.2 on Linux、 Hortonworks 2.2 on Windows Server、 Azure blob storage (WASB[S])

PolyBase は、**HDP** (Hortonworks Data Platform) と **CDH** (Cloudera's Distribution including Apache Hadoop) の 2 つの Hadoop プロバイダーをサポートしています。この項の冒頭の例では、**HDP 2.3** (Hortonworks Data Platform) に接続するスクリプトを紹介しましたが、これは構成値を「**7**」に設定していました。

```
EXEC sp_configure 'hadoop connectivity', 7
RECONFIGURE
```

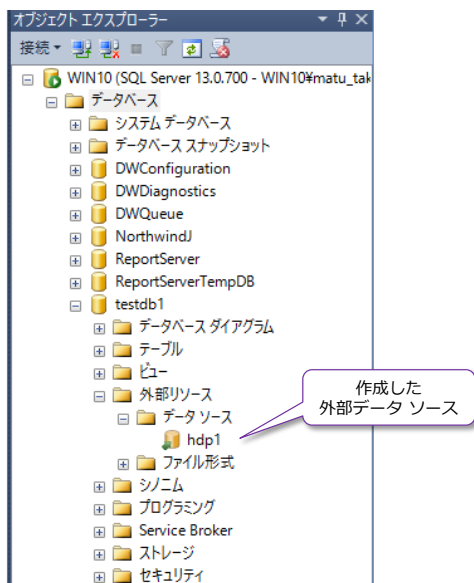
なお、PolyBase を利用して、**Azure BLOB ストレージ**に接続する場合も、「**7**」に設定しておけば接続することができます (**7** の他に、**1** と **4** でも接続可能)。

以降では、冒頭の例で **HDP 2.3** (Hortonworks Data Platform) に接続したスクリプトを、改めて 1 つ 1 つ説明していきます。

```
USE testdb1
/*
PolyBase で IP アドレス 192.168.1.36 のマシンに接続する例
(GentOS 7 上に HDP3.2 をインストールした Linux マシン)      */
CREATE EXTERNAL DATA SOURCE hdp1 WITH
(
  TYPE = HADOOP,
  LOCATION = 'hdfs://192.168.1.36:8020'
)
```

PolyBase では、**データベース内に外部データ ソース (EXTERNAL DATA SOURCE)** を作成するので、**USE** ステートメントでデータベースに接続してから、**CREATE EXTERNAL DATA SOURCE** ステートメントを実行しています。このステートメントでは、「**TYPE=HADOOP**」と指定して、**LOCATION** に IP アドレスと HDFS アクセスのためのポート番号 (**HDP 3.2** をインストールした時の既定値の **8020** ポートを利用) して、外部データ ソースを定義しています。

このように作成した外部データ ソースは、Management Studio のオブジェクト エクスプローラーで、次のように **外部リソース** の **データ ソース** を展開して確認することができます。



次に、**CREATE EXTERNAL FILE FORMAT** ステートメントでファイルのフォーマットを指定します。

```
CREATE EXTERNAL FILE FORMAT f1
WITH (
    FORMAT_TYPE = DELIMITEDTEXT,
    FORMAT_OPTIONS (FIELD_TERMINATOR = ',',
                    , USE_TYPE_DEFAULT = TRUE)
)
```

この例では **FIELD\_TERMINATOR** に **,** を指定して **CSV ファイル**を指定しています。この例でアクセスするファイルは、次のように CSV ファイルとして作成しています (HDP 3.2 の Hadoop ファイル システム内に **test1** という名前で、カンマ区切りのファイルとして作成)。

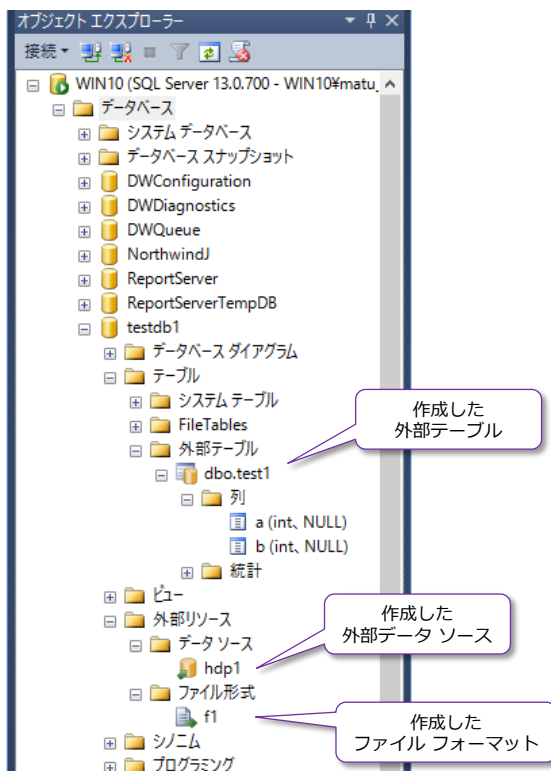




外部データ ソースとファイル フォーマットを定義した後は、**CREATE EXTERNAL TABLE** ステートメントで外部テーブルを定義します。

```
CREATE EXTERNAL TABLE dbo.test1 (
    a int NULL,
    b int NULL
)
WITH (LOCATION='/input/test1',
      DATA_SOURCE = hdp1,
      FILE_FORMAT = f1,
      REJECT_TYPE = VALUE,
      REJECT_VALUE = 0)
```

このステートメントでは、**DATA\_SOURCE** に最初に作成した外部データ ソース（ここでは **hdp1**）、**FILE\_FORMAT** に上で定義したファイル フォーマット（ここでは **f1**）を指定して、**LOCATION** に HDFS 上のファイルまたはフォルダーへのパス（ここでは **input** フォルダ配下の **test1** ファイル）を指定します。このように、作成した外部テーブルは、オブジェクト エクスプローラーでは次のように表示されます。



外部テーブルも、SQL Server のテーブルであるかのように表示されます。また、**SELECT** ステートメントで操作することもできるので、次のように検索することができます。

```
SELECT * FROM test1
```



```
SELECT * FROM test1
```

	a	b
1	1	111
2	2	222

HDFS 上の CSV ファイル (**input** フォルダの **test1** ファイル) の中身を参照できることを確認できます。このように、PolyBase を利用することで、HDFS 上のファイル（パスにフォルダーを指定した場合は、フォルダー配下のファイル群）を、SQL Server のテーブルであるかのように利用することができるのが大きなメリットです。

## ➡ 参考：HDP（Hortonworks Data Platform）を簡単に試す方法

HDP（Hortonworks Data Platform）を試したい場合は、簡単に試すことができる仮想マシン（すべてのソフトウェアがインストール／構成されて、起動するだけで利用できる状態のもの）が Sandbox（お試し環境）として提供されています。これは、次の URL からダウンロードすることができます。

HDP（Hortonworks Data Platform）の Sandbox（仮想マシン）のダウンロード URL

<http://hortonworks.com/products/hortonworks-sandbox/#install>

**Hortonworks Data Platform**  
企業向けに設計された、100%オープンソースの Apache Hadoop データプラットフォーム

概要 2.3 における新機能 **ダウンロード & インストール** ドキュメンテーション アドオン機能 アーカイブ

### ダウンロード & インストール

Try HDP 2.3.2 with Hortonworks Sandbox

HDP 2.3.2 on Hortonworks Sandbox  
Runs on VirtualBox or VMware

Try out the very latest features and functionality in Hadoop and its' ecosystem of projects with [HDP 2.3](#). Follow the [Step by Step Tutorials](#).

[System Requirements](#) | [Installation Steps](#) | [Release Notes](#)

for VirtualBox  
Mac & Windows

for VMware  
Mac & Windows

for VirtualBox  
(HDP 2.3.2 - 8.5 GB)

for VMware  
(HDP 2.3.2 - 8.7 GB)

### HDP 2.3.2: Ready for the enterprise

Automated (with Ambari 2.1)  
RHEL/CentOS/SLES/Ubuntu/Debian (64-bit)

The recommended way to set up HDP for a production environment. [Apache Ambari](#) simplifies the provisioning, management and monitoring of your cluster.

Automated Install Guide  
[PDF](#) | [HTML](#)

Upgrade Guide  
[PDF](#) | [HTML](#)

[Ambari Release Notes](#)

Download links within documentation  
[Go straight to the page](#)

## ➡ PolyBase の参考情報

Getting started with PolyBase (構成方法など)

<https://msdn.microsoft.com/en-us/library/mt163689.aspx>

Microsoft | Developer Network

Technologies ▾ Downloads ▾ Programs ▾ Community ▾ Documentation ▾ Samples

\*\*\* > Database Engine Features and Tasks > Database Features > PolyBase ▾

### Getting started with PolyBase

PolyBase groups for scale-out computation

PolyBase troubleshooting with dynamic management views

## Getting started with PolyBase

SQL Server 2016

Published: August 27, 2015

Updated: November 12, 2015

Applies To: SQL Server 2016 Preview

This topic contains the basics about installing and running PolyBase. After running the steps below, you will have:

- PolyBase installed and runnable on your server
- Examples of statements that create PolyBase objects
- An understanding of how to manage PolyBase objects in SQL Server Management Studio (SSMS)
- Examples of queries using PolyBase objects

### Install PolyBase

**Note**

PolyBase installs three user databases, DWConfiguration, DWDiagnosics, and DWQueue. These are necessary for PolyBase to run correctly, and are for internal use only.

### Prerequisites

In order to run PolyBase, the following is necessary:

PolyBase のヘルプのトップページ

<https://msdn.microsoft.com/en-us/library/mt143171.aspx>

\*\*\* > Database Engine > Database Engine Features and Tasks > Database Features ▾

► Databases (Database Engine)

► Stretch Database

▼ PolyBase

Getting started with PolyBase

PolyBase groups for scale-out computation

PolyBase troubleshooting with dynamic management views

► Tables

► In-Memory OLTP (In-Memory Optimization)

► Indexes (Database Engine)

► Partitioned Tables and Indexes

► Views

► Stored Procedures

► Search

► User-Defined Functions

► Statistics

► Plan Guides

## PolyBase

SQL Server 2016

Updated: October 28, 2015

Applies To: SQL Server 2016 Preview

PolyBase allows you to use T-SQL statements to access data stored in Hadoop or Azure Blob Storage and query it in an adhoc fashion. It also lets you query semi-structured data and join the results with relational data sets stored in SQL Server. PolyBase is optimized for data warehousing workloads and intended for analytical query scenarios.

Users → Queries → Results → SQL Server 2016 → PolyBase → Hadoop / Azure blob storage

### Challenges and solutions

PolyBase addresses one of the main customer pain points in data warehousing: accessing distributed datasets. With the increasing volumes of unstructured or semi-structured data sets, users are storing data sets in more cost-effective distributed and scalable systems,

## STEP 5. 既存機能の強化

---

この STEP では、「**SQL Server 2014 からの主な変更点**」や「**ライブ クエリ統計**」、「**クエリ ストア**」、「**AlwaysOn の拡張**」、「**T-SQL の強化**」、「**BI 関連機能の強化**」など、SQL Server の既存機能を強化する新機能について説明します。

この STEP では、次のことを学習します。

- ✓ SQL Server 2014 からの主な変更点
- ✓ ライブ クエリ統計 (Live Query Statistics)
- ✓ クエリ ストアで性能監視、プラン固定
- ✓ T-SQL の強化
- ✓ AlwaysOn 可用性グループの拡張 (自動フェールオーバー数が 2 から 3、ラウンドロビンレプリカ、ログ転送性能の向上、DTC 対応など)
- ✓ BI 関連の強化

Reporting Services の強化 (Datazen 統合など)、  
Analysis Services の強化、SSIS (Integration Services) の強化など

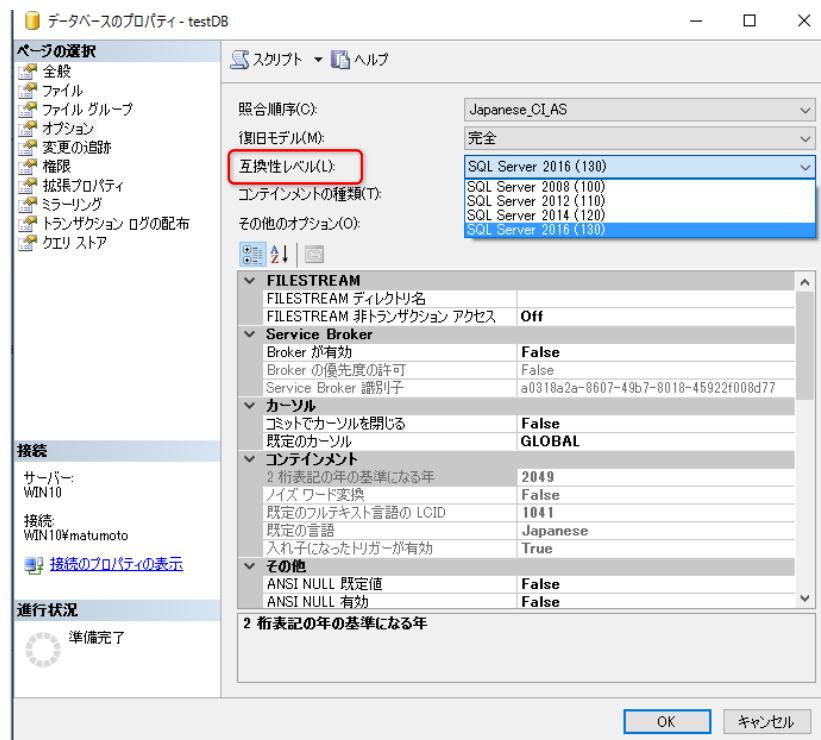
## 5.1 SQL Server 2014 からの主な変更点

SQL Server 2014 からの主な変更点は、次のとおりです。

- データベースの互換性レベル **130** の提供
- サービス アカウントに「**ボリュームの保守タスクを実行**」を簡単に付与可能に
- tempdb の拡張
  - tempdb の**複数ファイル／初期サイズ**をセットアップ時に選択できるようになった
  - tempdb の性能に関連する**トレースフラグ 1117 と 1118**が既定でオンになった
- **トレースフラグ 4199**が既定でオンになった
- Active Directory のパスワード認証／統合認証を利用可能に

### ➡ データベースの互換性レベル 130

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**」になりました（**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、リストア 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。

なお、互換性レベルの詳細については、オンライン ブックの以下のトピックががお勧めです。

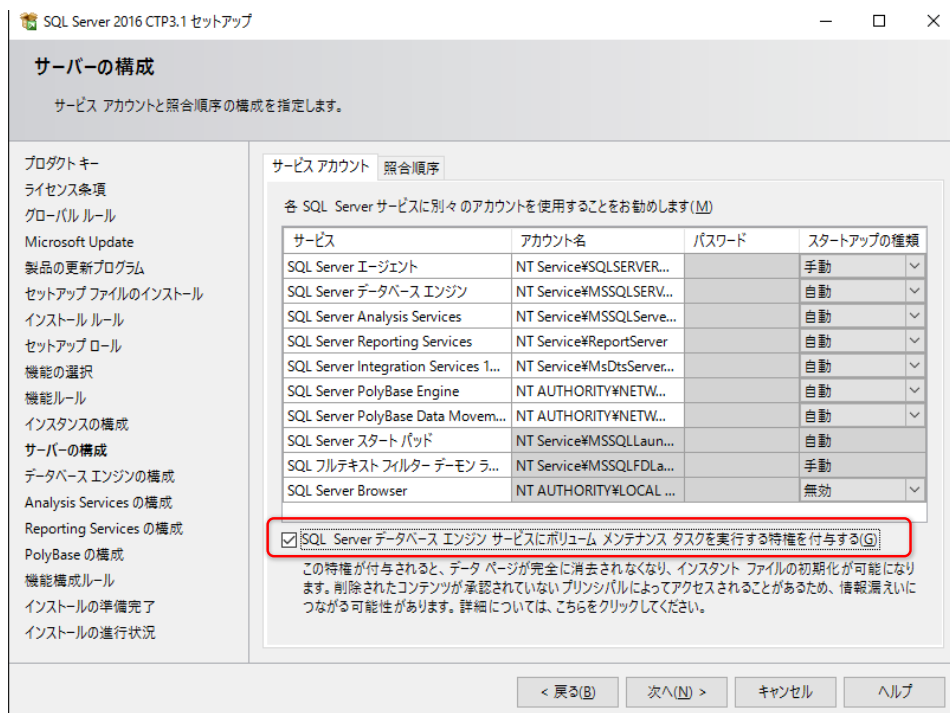
### ALTER DATABASE Compatibility Level (Transact-SQL)

<http://msdn.microsoft.com/en-us/library/bb510680.aspx>

Differences Between Compatibility Level 120 and Level 130	
This section describes new behaviors introduced with compatibility level 130.	
Compatibility-level setting of 120 or lower	Compatibility-level setting of 130
The Insert in an Insert-select statement is single-threaded..	The Insert in an Insert-select statement is multi-threaded or can have a parallel plan.
Memory Optimized Table queries execute single-threaded.	Memory Optimized Table queries can now have parallel plans
Introduced the SQL 2014 Cardinality estimator <b>CardinalityEstimationModelVersion="120"</b>	Further cardinality estimation ( CE) Improvements with the Cardinality Estimation Model 130 which is visible from a Query plan. <b>CardinalityEstimationModelVersion="130"</b>
Batch mode v/s Row Mode changes with Columnstore indexes <ul style="list-style-type: none"> <li>Sorts on a table with Columnstore index are in Row mode</li> <li>Windowing function aggregates operate in row mode such as</li> </ul>	Batch mode v/s Row Mode changes with Columnstore indexes <ul style="list-style-type: none"> <li>Sorts on a table with a Columnstore index are now in batch mode</li> <li>Windowing aggregates now operate in batch mode such as LAG/LEAD</li> </ul>

## ➡ サービス アカウントに「ボリュームの保守タスクを実行」を簡単に付与可能に

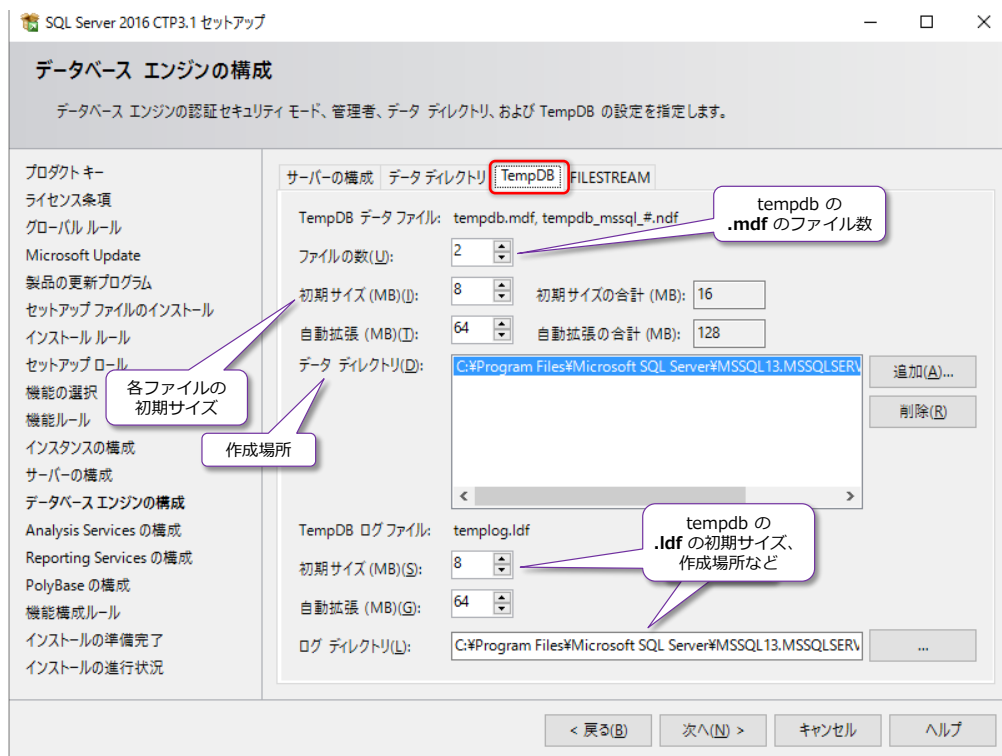
SQL Server 2016 からは、インストール時に、サービス アカウントに対して、ユーザーの権利である「ボリュームの保守タスクを実行」を付与できるようになりました。



この権利は、**瞬時初期化**（.mdf ファイルのサイズが拡張するときに、瞬間的にサイズを拡張できる機能）を有効化するために、以前のバージョンでは、手動で行っている作業でした（なお、サービス アカウントが Administrators グループのメンバーである場合には、この権利は自動的に付与されています）。

## ➡ tempdb の拡張 ～ファイル数や初期サイズをセットアップ時に設定可能に～

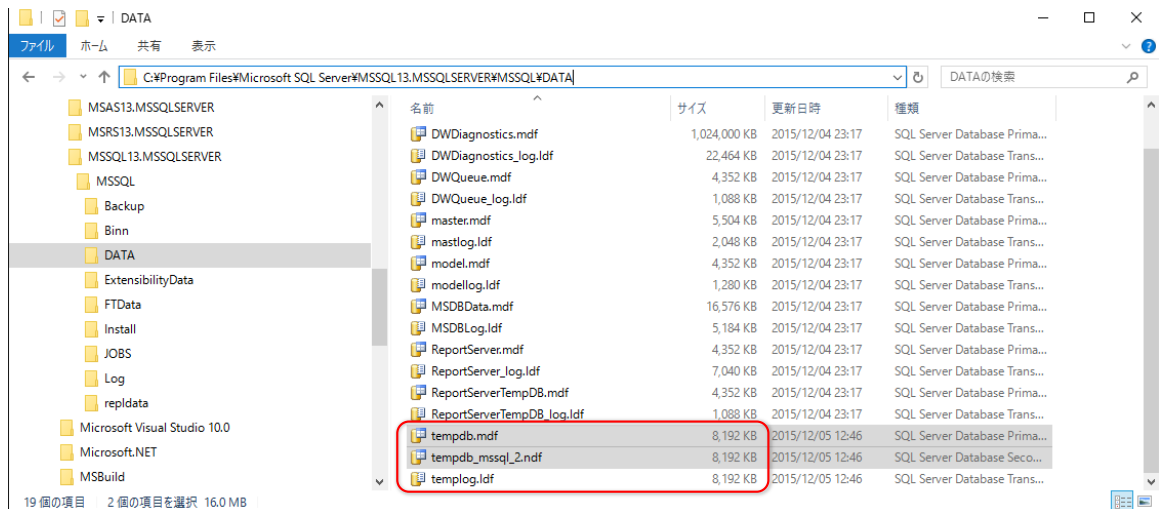
SQL Server 2016 からは、SQL Server のインストール時に、**tempdb データベースの設定**を変更できるようになりました（今までは、tempdb の設定を変更する際には、インストール後に別途行う必要がありましたが、SQL Server 2016 からはインストール時に設定変更できるようになりました）。これは、インストール時の「**データベース エンジンの構成**」ページで、次のように「**TempDB**」ページを開くことで設定できます。



**.mdf**（データ ファイル）に関しては、ファイル数や初期サイズ、作成場所、**.ldf**（ログ ファイル）に関しては、初期サイズ、作成場所などを細かく設定できるようになりました。

なお、.mdf のファイル数は、既定では搭載されている CPU のスレッド数または 8 スレッド以上の場合には 8 に設定されます（画面は 2 スレッドの場合の例）。また、既定の作成場所は、次のように SQL Server のインストール先の **DATA** フォルダー（以前のバージョンと同様、master などのシステム データベースが格納される場所）になります。

**C:\Program Files\Microsoft SQL Server\MSSQL13.MSSQLSERVER\MSSQL\DATA**



## ➡ tempdb の性能に関連するトレースフラグ 1117 と 1118 が既定でオンに

SQL Server 2016 からは、tempdb データベースの性能に関連するトレース フラグである **1117** と **1118** をオンにしたときと同等の動作になるように、既定の動作が変更になります。この 2 つは、tempdb でラッチ待ちが発生している場合の動作を改善できるトレース フラグで、SQL Server 2014 以前のバージョンを利用している場合のベスト プラクティスの 1 つでした。SQL Server 2016 からは、既定でこの 2 つのトレースフラグが設定されている状態になるので、これらを設定する必要がなくなりました。

なお、トレースフラグ **1117** と **1118** については、以下の **KB 2154845**（サポート技術情報）が参考になります。

Recommendations to reduce allocation contention in SQL Server tempdb database

<http://support.microsoft.com/en-us/kb/2154845>

## ➡ トレースフラグ 4199 がオンと同等の動作（クエリ オプティマイザーの動作）

SQL Server 2016 からは、データベースの互換性レベルで **130** を利用している場合に、**トレース フラグ 4199** をオンにしたときと同等の動作になるように、既定の動作が変更になります。これは、クエリ オプティマイザーがクエリを最適化する際の見積行数（予測行数）などに関する変更（hotfix）が含まれているものです。

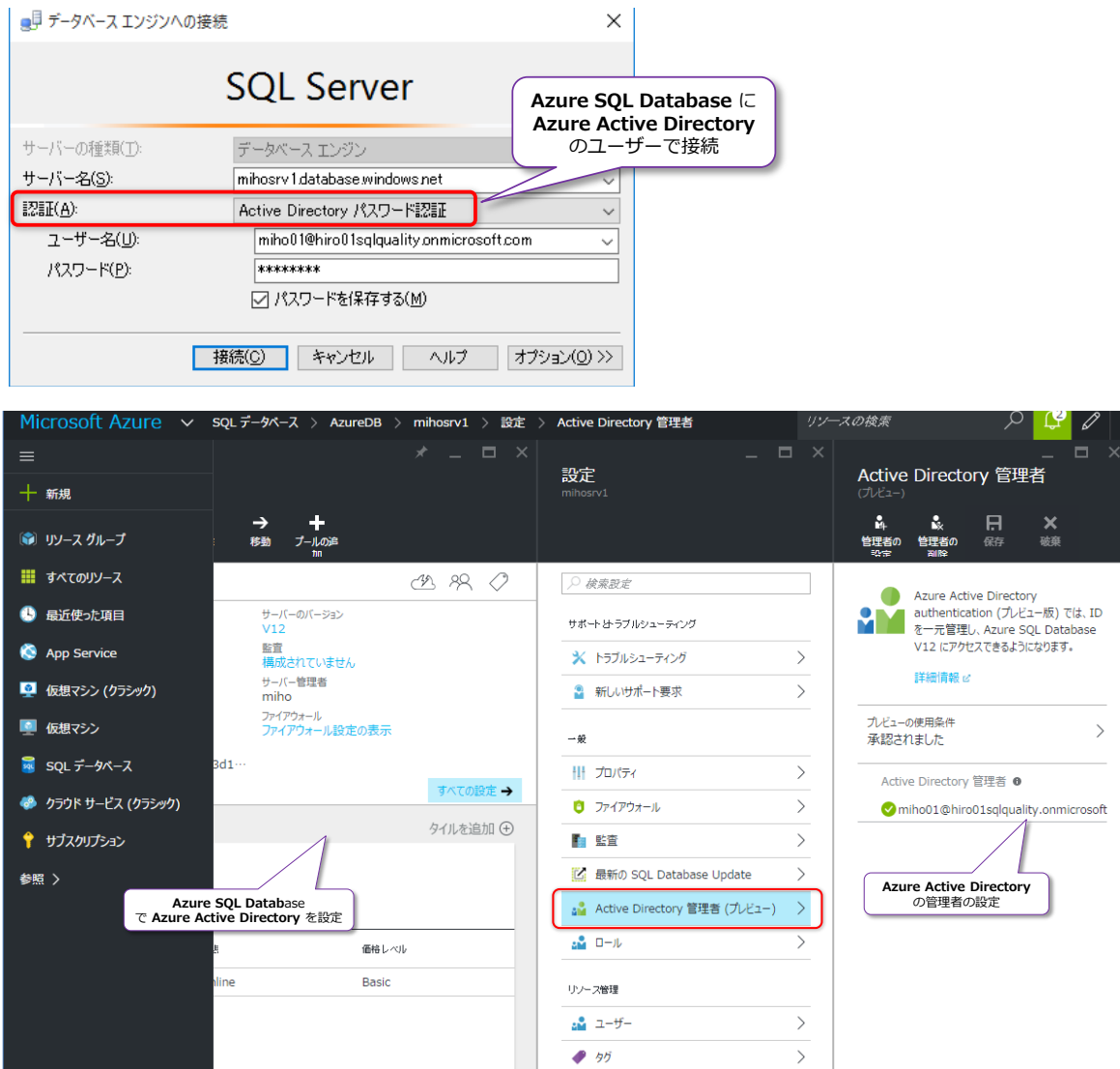
**トレースフラグ 4199**（やデータベースの互換性レベルによる動作の違いなど）については、以下の **KB 974006**（サポート技術情報）が参考になります。

SQL Server query optimizer hotfix trace flag 4199 servicing model

<http://support.microsoft.com/en-us/kb/974006>

## ➡ Active Directory パスワード認証／統合認証を利用可能に

SQL Server 2016 の Management Studio では、認証方法として「**Active Directory パスワード認証**」と「**Active Directory 統合認証**」が選択できるようになりました。これを利用すれば、次のように **Azure Active Directory** (Azure AD) を利用して、**Azure SQL Database** に接続できるようになります。



なお、Azure SQL Database で Azure Active Directory を利用する方法については、Azure ドキュメントの以下のトピックが参考になります。

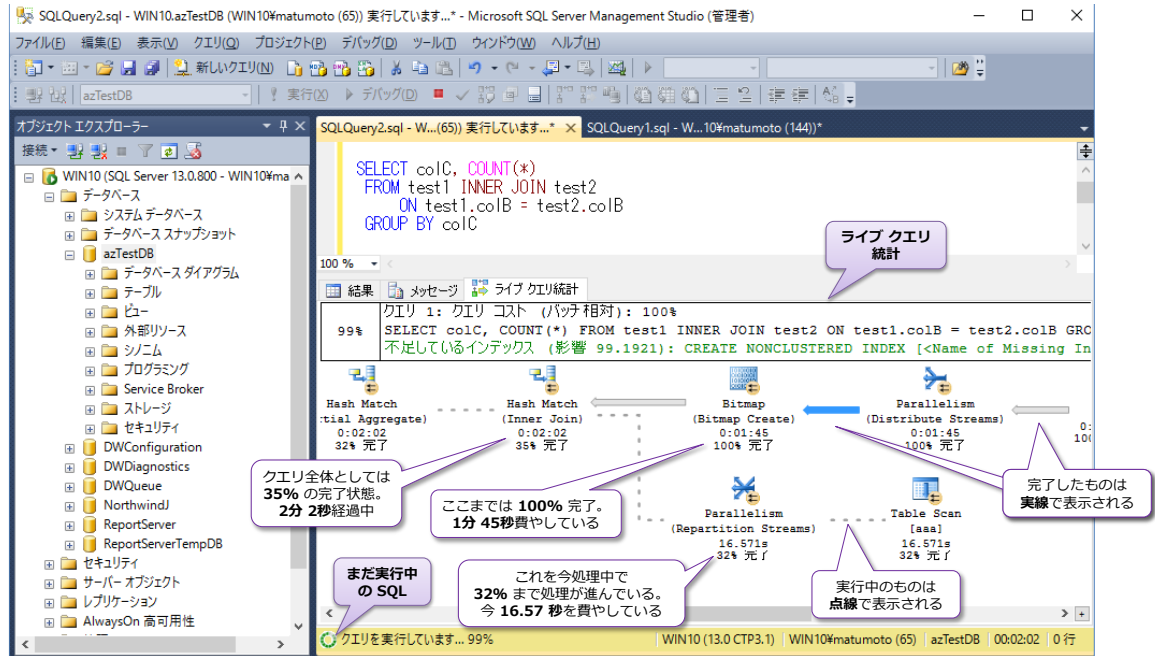
Azure Active Directory 認証を使用して SQL Database に接続する

<http://azure.microsoft.com/ja-jp/documentation/articles/sql-database-aad-authentication/>



## 5.2 ライブ クエリ統計 (Live Query Statistics)

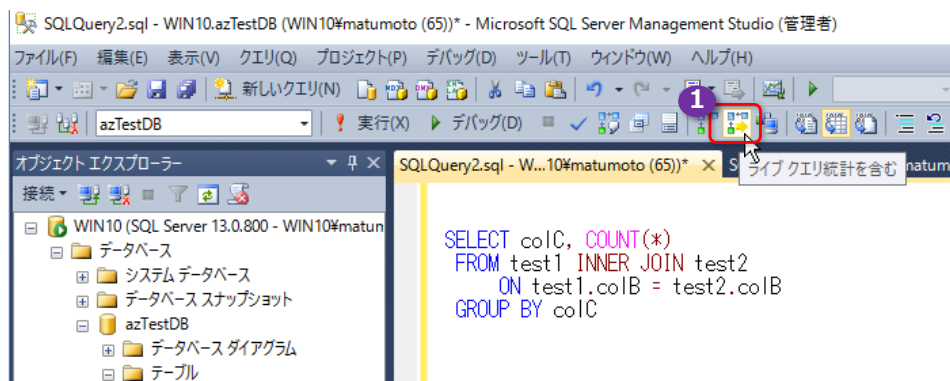
**ライブ クエリ統計**は、現在実行中の SQL ステートメントの実行プランのうち、どの部分を実行しているのかを確認できる、大変便利な機能です。



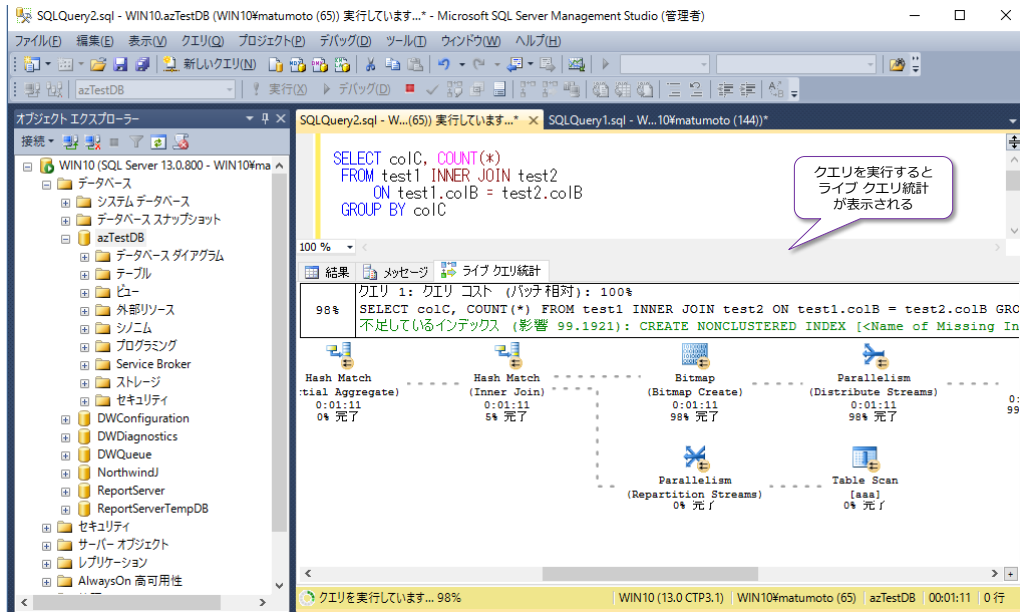
ライブ クエリ統計では、完了した部分が**実線**で表示されて、**実行中 (未完了)**の部分が**点線**で表示されます。未完了のものは、現在の完了状況が % で表示されて、**何秒 (s) 経過しているのか**が表示され、また、完了したものについては、100% と表示されて、どれぐらいの経過時間だったのかを確認することができます。これを見れば、どこがボトルネックでクエリが遅くなっているのかが一目瞭然なので、パフォーマンス チューニングをする際に本当に役立ちます。

### ➡ ライブ クエリ統計の利用方法 (1クリックのみ)

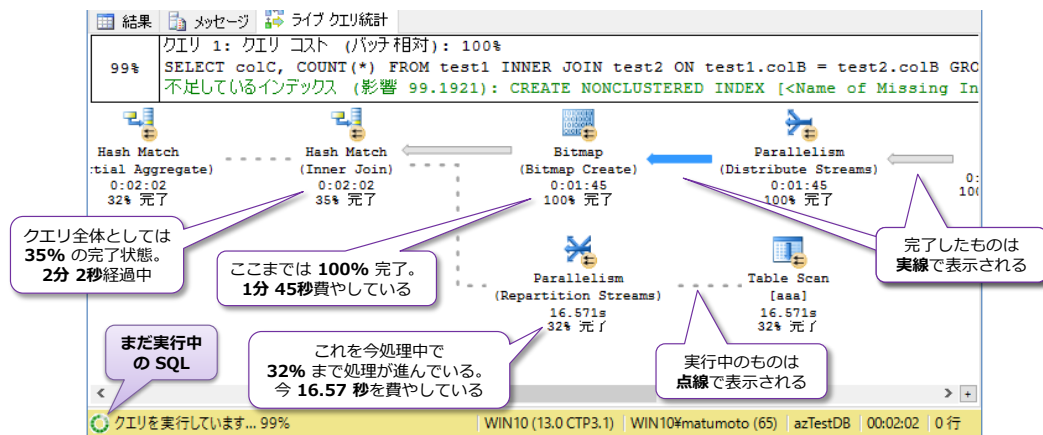
ライブ クエリ統計は、次のようにツールバーの「**ライブ クエリ統計を含む**」をクリックするだけで利用することができます。



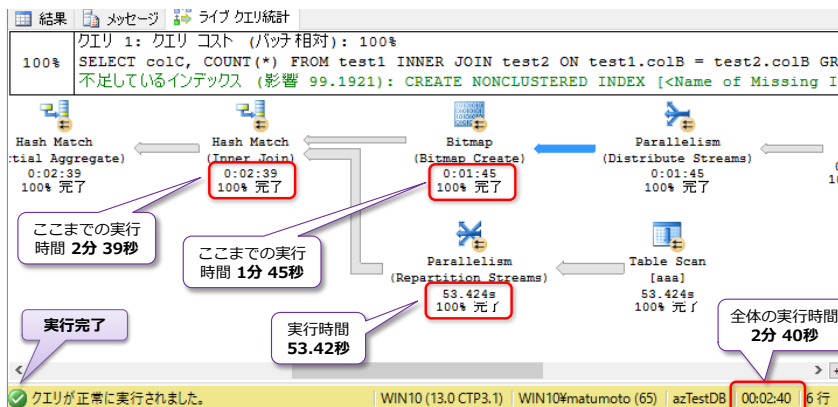
あとは、クエリを実行すれば、実行プランが表示されて、どこまで実行しているのかをグラフィカルに確認できるようになります。



実行中は、点線で表示されている部分が、完了後に実線に変わっていき、次のように表示されます。



実行が完了した後は、すべての実線に変わって、次のように表示されます。

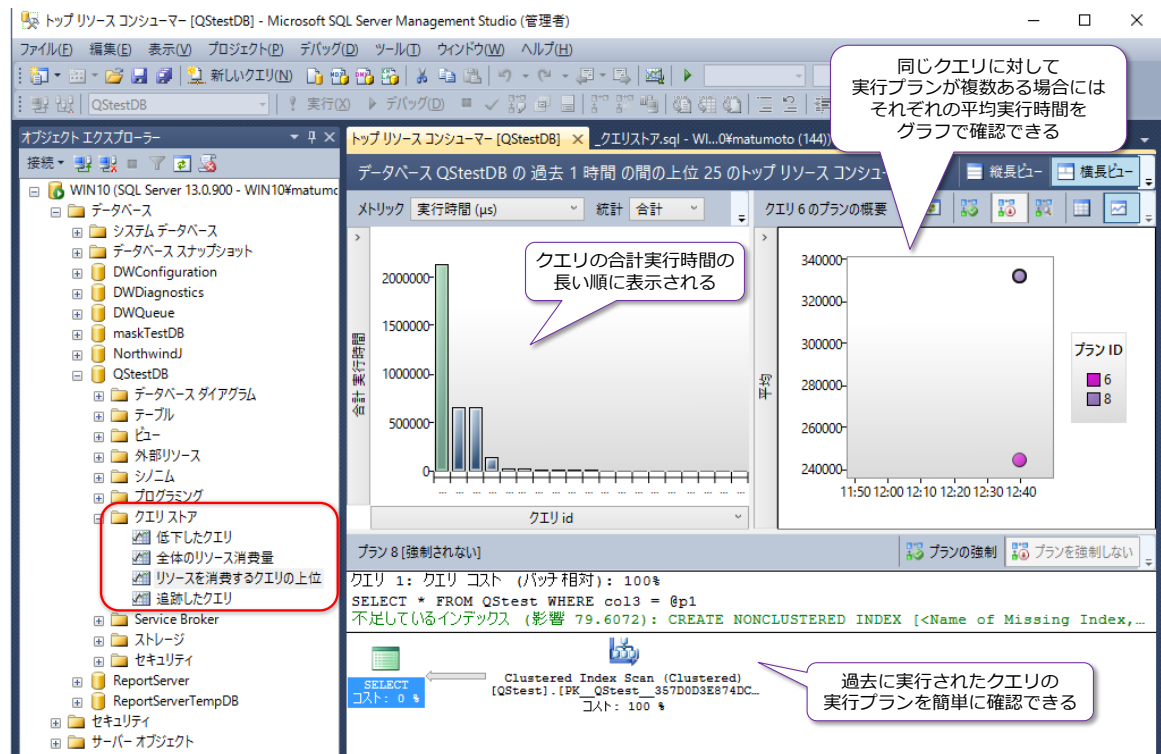


全体の実行時間のうち、どの処理にどれだけの実行時間がかかったのかが簡単に分かるようになっているので、ボトルネックとなっている処理をすぐに見つけることができます。

## 5.3 クエリ ストアで性能監視、プラン固定

クエリ ストア (Query Store) は、「昨日までは問題なかったのに、何故か急に遅くなった」や「遅くなった理由が分からない」といったクエリのトラブル (性能問題) を解決できる可能性がある、大変便利な機能です。

クエリ ストアを利用すると、Store (蓄積) という名のとおり、**クエリの実行履歴** (と実行プラン) を保存することができるようになります。

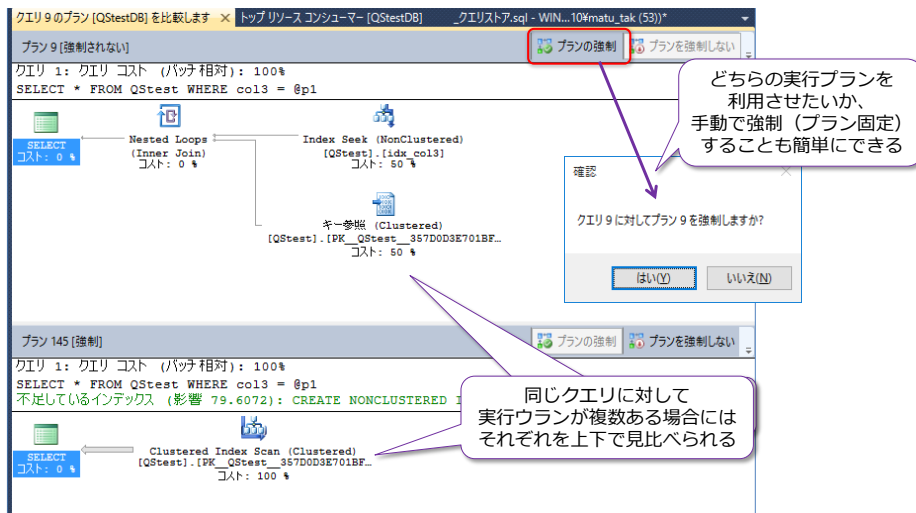


これまでのバージョンでも **query\_stats** という動的な管理ビュー (DMV) を参照すれば、クエリの実行履歴を確認することができましたが、この情報は、SQL Server を再起動したり、プロシージャ キャッシュのサイズが圧迫された場合には、参照できない (クリアされてしまう) という性質のものでした。これに対して、クエリ ストアであれば、クエリの実行履歴を永続化することができるので、SQL Server を再起動しても、過去に振り返って、クエリの実行履歴を確認できるようになります (遅くなった原因のクエリを過去に振り返って調べられるようになります)。

弊社のお客様の中には、SQL Server を定期的に再起動して運用しているという場合がありますが、このように運用していると、これまでのバージョンの SQL Server では過去のクエリ実行履歴を確認することができませんでした (‘query\_stats’ に履歴が残っていないため)。これに対応するには、アプリケーション側で対応したり (Log4net/Log4J などを利用して、アプリ側でログを記録したり)、SQL Server のトレース (sp\_trace\_create) 機能や拡張イベント (XEvents)、パフォーマンス データ コレクション機能を利用したりして、クエリの実行履歴を記録しておくという方法しかとれませんでした (弊社の過去のパフォーマンス チューニングではトレースをよく利用していました)。しかし、SQL Server 2016 からは、クエリ ストア機能が提供されるので、これを利用すれば、簡単に (数クリックで) クエリの実行履歴を保存できるようになるので、大変便利です。

## ➡ クエリ ストアの最大のメリット ～実行プランの比較／プラン固定～

クエリ ストア機能では、**クエリの実行プラン**も同時に記録しておくことができるので、そのクエリがどういった実行プランで実行されていたのかを、過去に振り返ることができます。これがクエリ ストア機能の最大のメリットです。また、同じクエリに対して、複数の実行プランがある場合には、次のように簡単に見比べることもでき、かつどちらを利用させたいのかを変更（DB 管理者によるプラン固定）することも簡単にできます。



このように、クエリ ストア機能を利用すれば、実行プランを簡単に確認／操作できるようになるので、次のような（多くの方々を悩ませてきた）**性能問題を解決**できる可能性が出てきます。

- 開発環境では問題がなかったのに、本番環境では性能が出ない
- 月イチの夜間バッチを実行した翌日にパフォーマンスが悪くなった
- 日によって性能が出るときと出ないときがある
- SQL Server を再起動した後に、性能が出なくなった

このような状況は、弊社のお客様でもよくあるのですが、これらの原因のほとんどは実行プランによるものです。健全な性能が出ているときは、効率の良い実行プランが選択されていて、性能が出ていないときには効率の悪い実行プランが選択されているといった状況です。

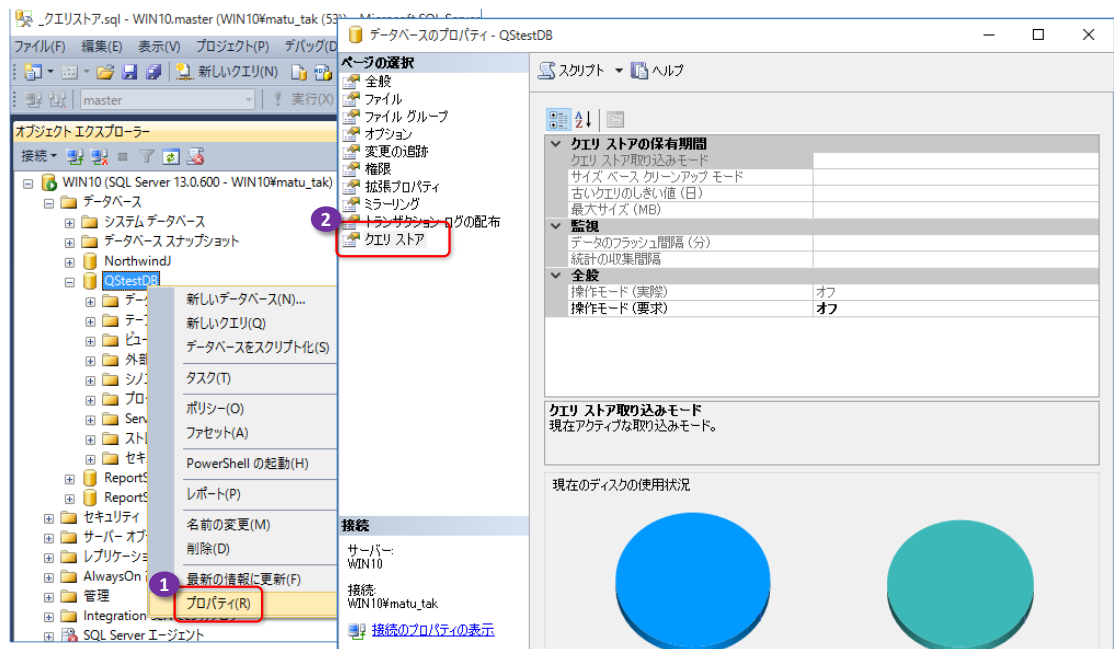
## ➡ Let's Try

それでは、クエリ ストアを試してみましょう。

1. まずは、クエリ ストアを試すためのデータベースを作成します。クエリ エディターで、次のように **CREATE DATABASE** ステートメントを実行して、「QStestDB」という名前のデータベースを作成します。

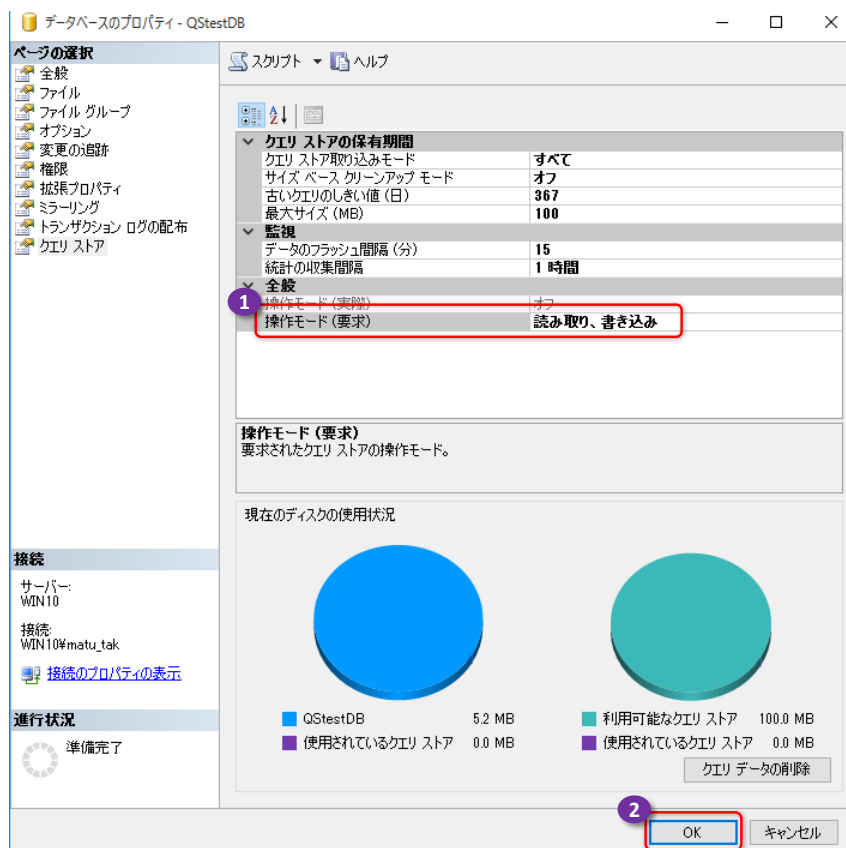
```
CREATE DATABASE QStestDB
```

2. 次に、クエリ ストアを有効化するために、データベースのプロパティを開きます。これを行うには、次のように「QStestDB」データベースを右クリックして、[プロパティ] をクリックします。



「データベースのプロパティ」ダイアログでは、[クエリ ストア] ページを開きます。

3. [クエリ ストア] ページでは、[操作モード (要求時)] を [読み取り/書き込み] に変更して [OK] ボタンをクリックします。



以上でクエリ ストアの設定が完了です（ここで設定できるプロパティ項目については後述します）。これでクエリ ストアが有効になって、クエリの実行履歴が自動的に記録されていくようになります。

**Note : SQL ステートメントでクエリ ストアを有効化する場合**

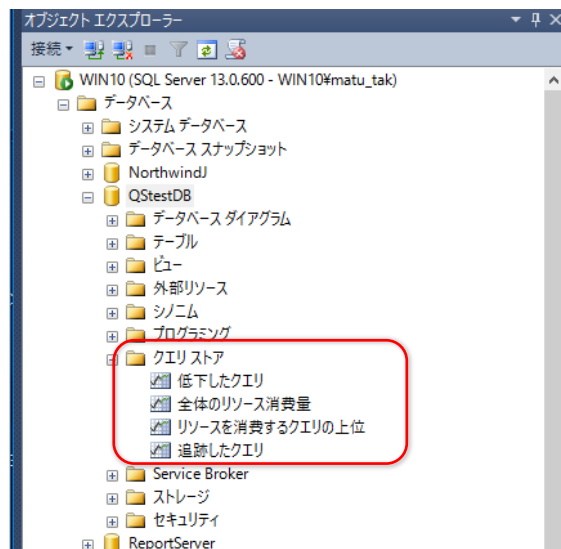
SQL ステートメントを利用して、クエリ ストアを有効化したい場合には、次のように **ALTER DATABASE** ステートメントを実行します。

— SQL でクエリ ストアを有効化する場合

```
ALTER DATABASE QStestDB
SET QUERY_STORE = ON
```

**SET** 句で「**QUERY\_STORE = ON**」と指定することで、クエリ ストアを有効化することができます。

4. クエリ ストアを有効化すると、次のようにデータベース内に「**クエリ ストア**」フォルダーが作成されるようになります。



5. 次に、データベース内にテーブルを作成します。

```
USE QStestDB
CREATE TABLE QStest
( col1 int IDENTITY PRIMARY KEY
, col2 int
, col3 int )
```

テーブル名は「**QStest**」として、**col1** ~ **col3** の 3 つの列（いずれも **int** データ型）を持つようにしています。

6. 次に、データを 10 万件追加します。この時点では、**col3** 列には「1」のデータのみを **INSERT** するようにします。

```
DECLARE @i int = 1
WHILE @i <= 100000
```



```
BEGIN
    INSERT INTO QStest VALUES(@i, 1)
    SET @i += 1
END
```

```
-- 10万件のデータを追加。col3 にはすべて「1」を INSERT
DECLARE @i int = 1
WHILE @i <= 100000
BEGIN
    INSERT INTO QStest VALUES(@i, 1)
    SET @i += 1
END
```

100 %

メッセージ

(1 行処理されました)

(1 行処理されました)

(1 行処理されました)

(1 行処理されました)

(1 行処理されました)

7. 次に、**col3** 列に「1」ではないデータを格納するために、**UPDATE** ステートメントを利用して、データを更新します。

```
UPDATE QStest SET col3 = 2 WHERE col1 = 99998
UPDATE QStest SET col3 = 3 WHERE col1 = 99999
```

8. 次に、**SELECT** ステートメントを実行して、データを確認しておきます。

```
SELECT * FROM QStest
```

```
-- データの個数の確認
SELECT col3, COUNT(*) FROM QStest
GROUP BY col3
```

100 %

結果 メッセージ

	col1	col2	col3
99992	99992	99992	1
99993	99993	99993	1
99994	99994	99994	1
99995	99995	99995	1
99996	99996	99996	1
99997	99997	99997	1
99998	99998	99998	2
99999	99999	99999	3
100000	100000	100000	1

99998 のデータの **col3** 列が「2」、99999 が「3」、その他は「1」であることを確認できます。

9. 次に、**GROUP BY** 演算を利用して、**col3** 列のデータの件数を確認しておきます。

```
SELECT col3, COUNT(*) FROM QStest
GROUP BY col3
```

```
-- データの個数の確認
SELECT col3, COUNT(*) FROM QStest
GROUP BY col3
```

	col3 (列名なし)	
1	3	1
2	1	99998
3	2	1

「3」と「2」はデータ件数が 1 件、「1」は 99,998 件であることを確認できます。

10. 次に、col3 列に非クラスター化インデックスを作成しておきます。

```
CREATE INDEX idx_col3 ON QStest(col3)
```

## クエリの実行

次に、クエリ ストアにデータを蓄積するために、クエリをいくつか実行しておきましょう。

- まずは、「WHERE col3 = @p1」という形のパラメーター クエリを利用して、「@p1」に「2」を与えて、col3 列が 2 のデータのみを取得してみます。

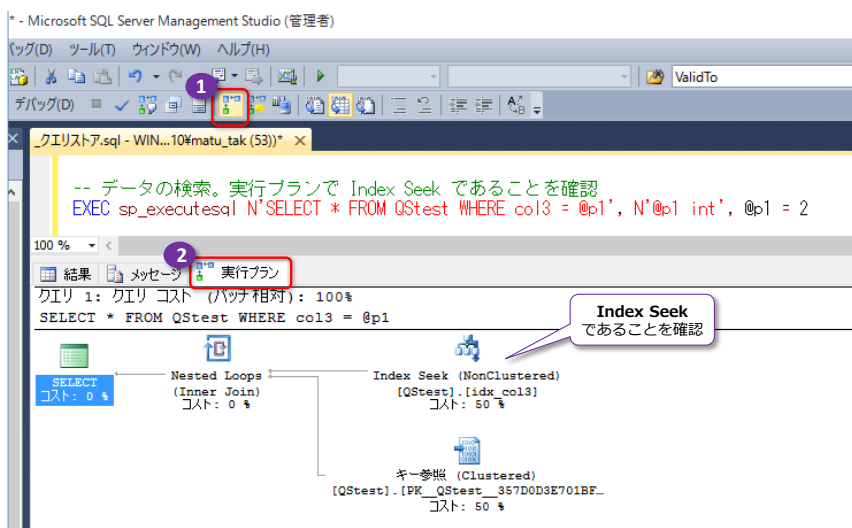
```
EXEC sp_executesql N'SELECT * FROM QStest WHERE col3 = @p1', N'@p1 int', @p1 = 2
```

```
EXEC sp_executesql N'SELECT * FROM QStest WHERE col3 = @p1', N'@p1 int', @p1 = 2
```

	col1	col2	col3
1	99998	99998	2

2 を与えてクエリを実行

- このクエリは、先ほど作成したインデックスを利用した **Index Seek** で実行されるので、これを確認しておきます。ツールバーの「**実際の実行プランを含める**」をクリックして、再度クエリを実行します。

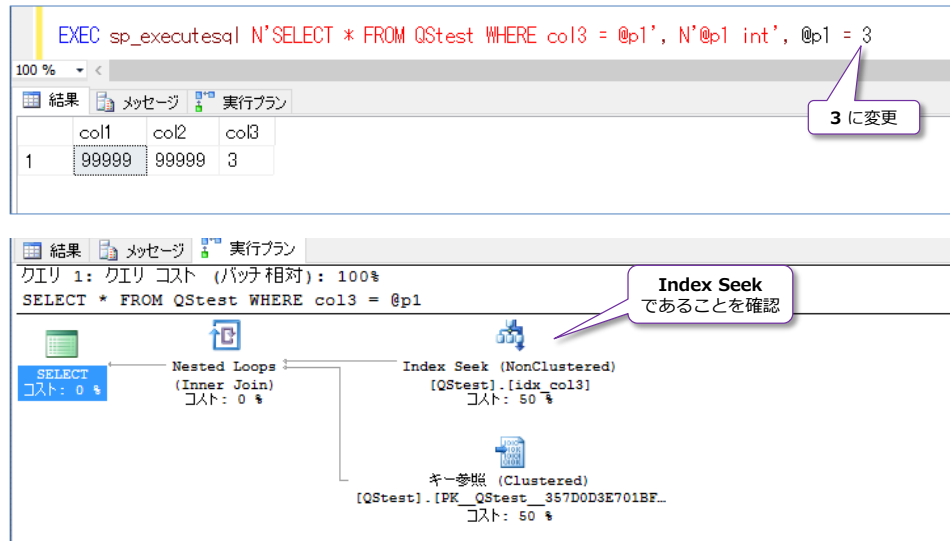




クエリの実行後、[実行プラン] タブで、**Index Seek** になっていることを確認します。

3. 次に、同じクエリで、「@p1」に与える値を「3」に変更して実行してみます。

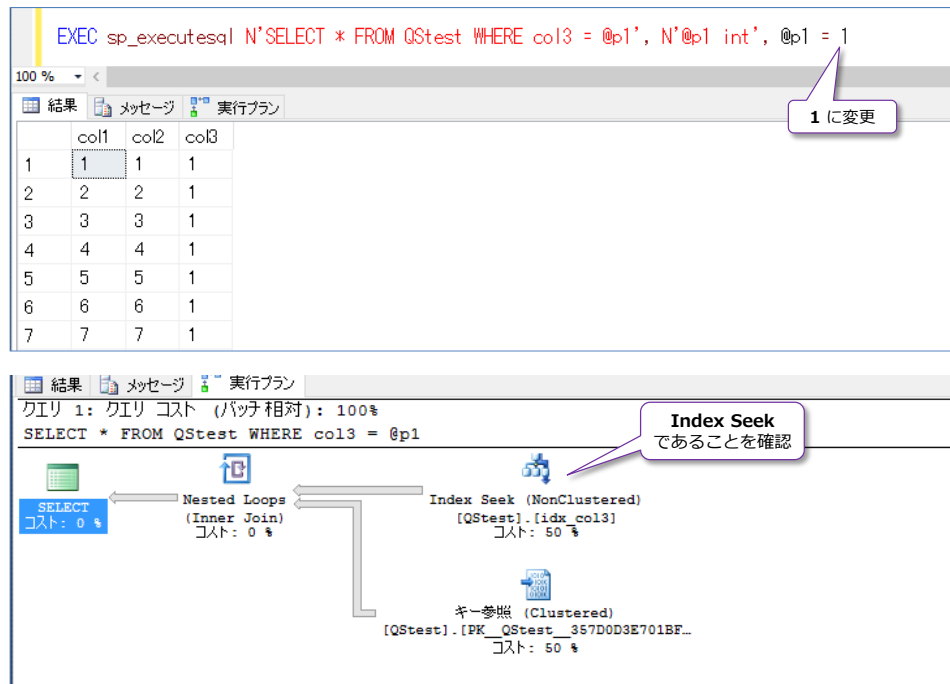
```
EXEC sp_executesql N'SELECT * FROM QStest WHERE col3 = @p1', N'@p1 int', @p1 = 3
```



これも **Index Seek** で実行されていることを確認できます。

4. 次に、同じクエリで、「@p1」に与える値を「1」に変更して実行してみます。

```
EXEC sp_executesql N'SELECT * FROM QStest WHERE col3 = @p1', N'@p1 int', @p1 = 1
```

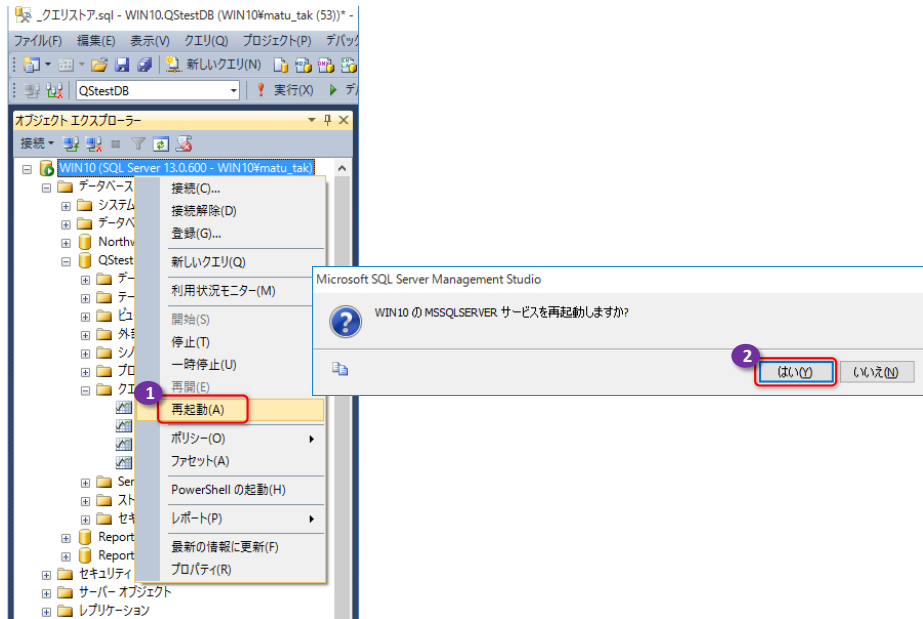


「1」に該当するデータ件数が多い（99,998 件）ので、実行には少し時間がかかります。これも **Index Seek** で実行されていることを確認できます。

## ➡ SQL Server 再起動後の同じクエリの実行プランの確認

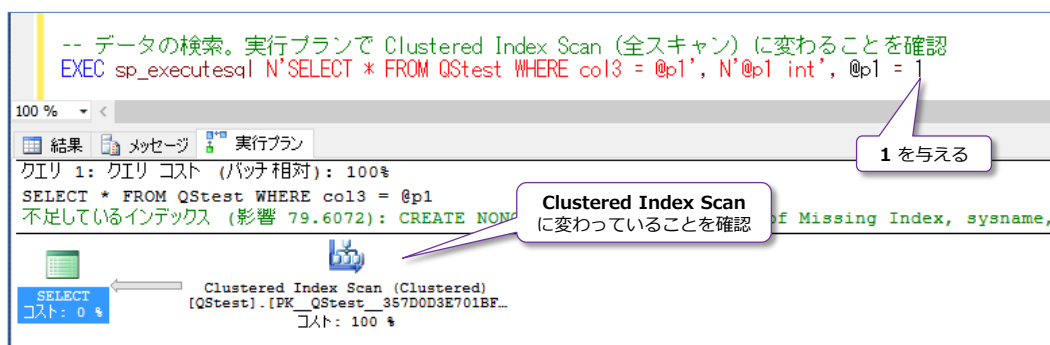
次に、SQL Server サービスを再起動して、再起動後に同じクエリを実行して、実行プランがどのように変化するのを確認してみましょう。

1. SQL Server サービスを再起動するには、次のように操作します。



2. 再起動が完了したら、再度、先ほどと同じクエリを実行します。このとき、「@p1」に与える値には「1」を指定します。

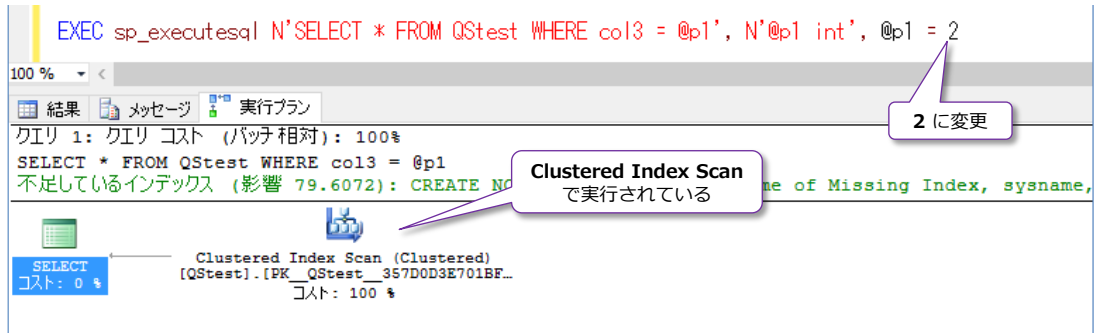
```
EXEC sp_executesql N'SELECT * FROM QStest WHERE col3 = @p1', N'@p1 int', @p1 = 1
```



今度は、**Clustered Index Scan** (全スキャン) で実行されるように変わっていることを確認できます。「1」に該当するデータ件数は多いので、SQL Server のクエリ オプティマイザー (クエリの最適化を行う内部エンジン) が全スキャンをしたほうが速いと判断して、このようになっています。

3. 次に、「@p1」に与える値を「2」に変更して実行します。

```
EXEC sp_executesql N'SELECT * FROM QStest WHERE col3 = @p1', N'@p1 int', @p1 = 2
```



これも、**Clustered Index Scan**（全スキャン）で実行されていることを確認できます。「2」に該当するデータ件数は **1 件のみ**なので、**Index Seek** を利用した方が速く実行できるにも関わらず、全スキャンが選択されてしまっています。

このように、パラメーター クエリの場合は、一度実行プランが選択されると、パラメーターに代入される値に関係なく、同じ実行プランが選択されてしまうという動作になります（そもそも、パラメーター クエリは、コンパイルの負荷を軽減する＝パラメーター違いのクエリでも同じ実行プランを再利用するための機能なので、これはごくごく普通の動作になります）。

## ➡ プロシージャ キャッシュのクリア後の実行プランの確認

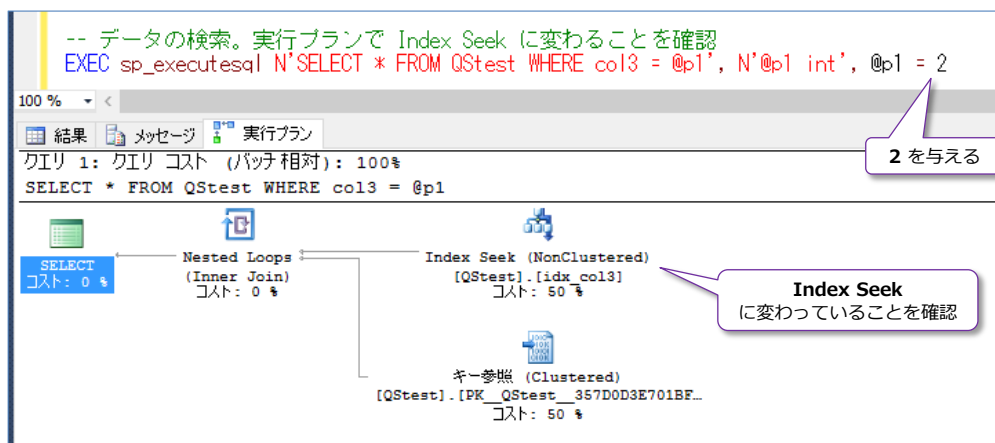
次に、**プロシージャ キャッシュをクリア**した場合に、同じクエリを実行して、実行プランがどのように変化するのを確認してみましょう。

1. プロシージャ キャッシュをクリアするには、次のように **DBCC FREEPROCCACHE** コマンドを実行します。

```
DBCC FREEPROCCACHE
```

2. クリアが完了したら、再度、先ほどと同じクエリを実行します。このとき、「@p1」に与える値には「2」を指定します。

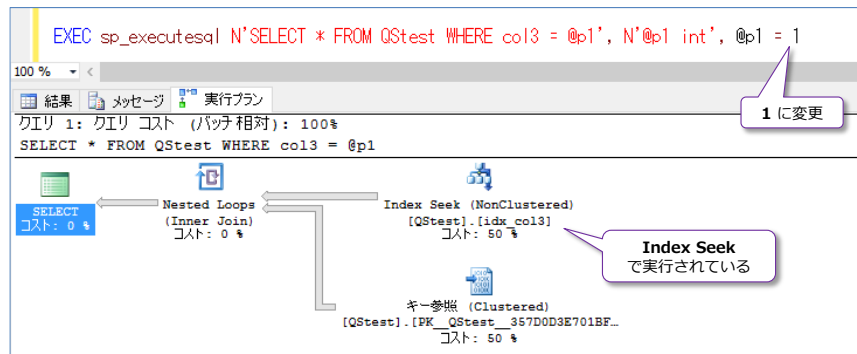
```
EXEC sp_executesql N'SELECT * FROM QStest WHERE col3 = @p1', N'@p1 int', @p1 = 2
```



今度は、**Index Seek**（インデックスのピンポイント検索）で実行されるように変わっていることを確認できます。このように、プロシージャ キャッシュをクリアした場合にも、SQL Server を再起動したときと同じ動作（クエリ オプティマイザーによる実行プランの選択し直し）になります。

3. 次に、「@p1」に与える値を「1」に変更して実行します。

```
EXEC sp_executesql N'SELECT * FROM QStest WHERE col3 = @p1', N'@p1 int', @p1 = 1
```



これも、Index Seek で実行されていることを確認できます。

ここまでの結果をまとめると、次のようになります。

	最初に col3 に与えた値	実行プラン	次に col3 に与えた値	次に col3 に与えた値
最初	2	Index Seek	3	1
SQL Server 再起動後	1	Clustered Index Scan	2	-
DBCC FREEPROCCACHE 後	2	Index Seek	1	-

col3 内の「2」のデータ件数は「1 件」、「1」のデータ件数は「99,998 件」なので、パラメーターに「2」を与えた場合は **Index Seek**（ピンポイント検索）のほうが効率が良いと判断され、「1」を与えた場合は **Clustered Index Scan**（全スキャン）のほうが効率が良いと判断されています。このように、データに“偏り”がある場合には、SQL Server の再起動やプロシージャ キャッシュのクリアによって、選択される実行プランが定まらない（ころころ変わってしまう、最初に与えた値を最適化するように実行プランが選択される）という状況になります（このような状況は、弊社のお客様でもたくさん遭遇してきました）。

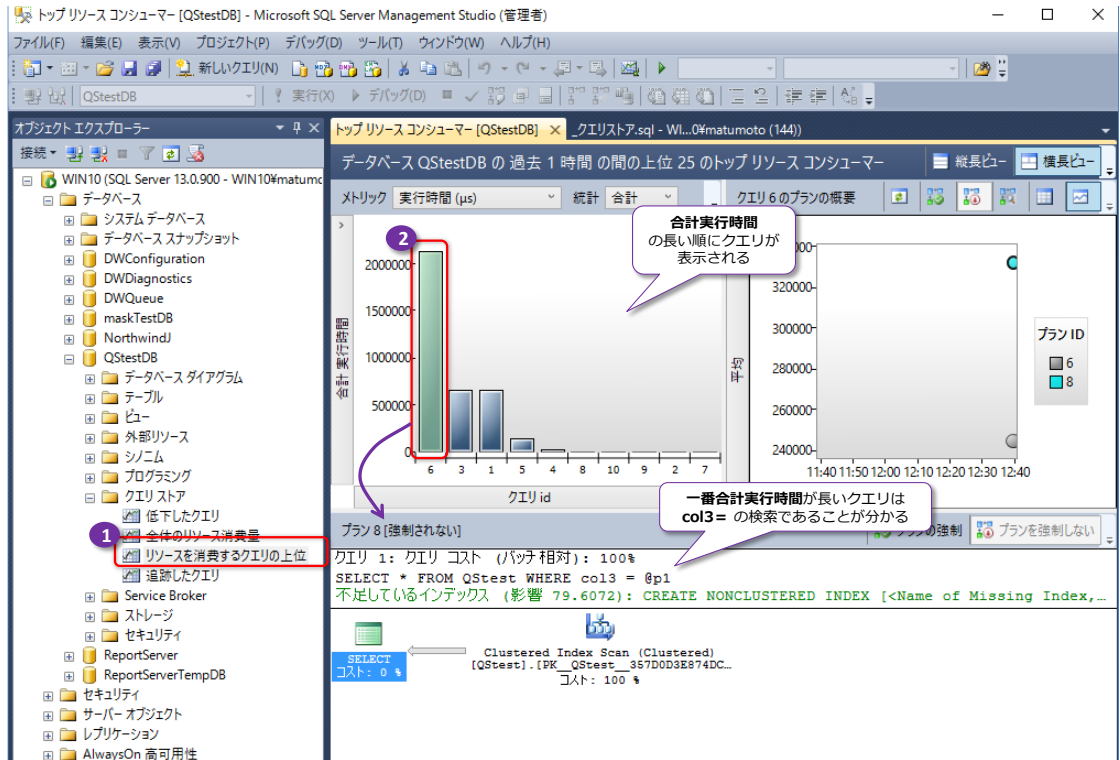
クエリ ストア機能を利用すれば、このような状況が発生したかどうかを簡単に確認できるようになり、かつデータベース管理者が意図した実行プラン（この実行プランを利用して欲しいというもの）に強制（プラン固定）していくということも簡単に行うことができます。以降では、これも試してみましょう。

## ➡ クエリ ストアを GUI で確認

まずは、クエリ ストアに蓄積されたデータを確認してみましょう。

1. クエリ ストアに蓄積されたデータを確認するには、次のようにオブジェクト エクスプローラ

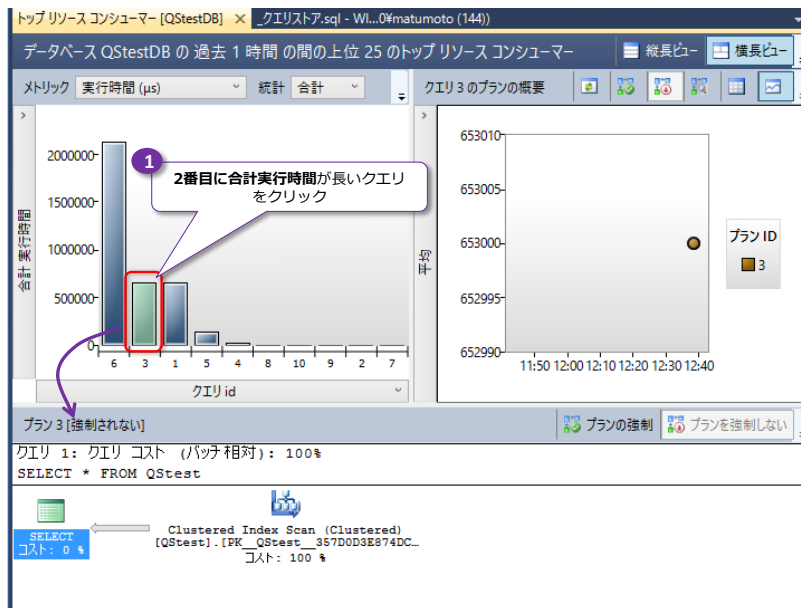
ーで、**[クエリ ストア]** フォルダを展開して、**[リソースを消費するクエリの上位]** をクリックします。



[合計 実行時間] の長い順にクエリが表示されていることを確認できます。

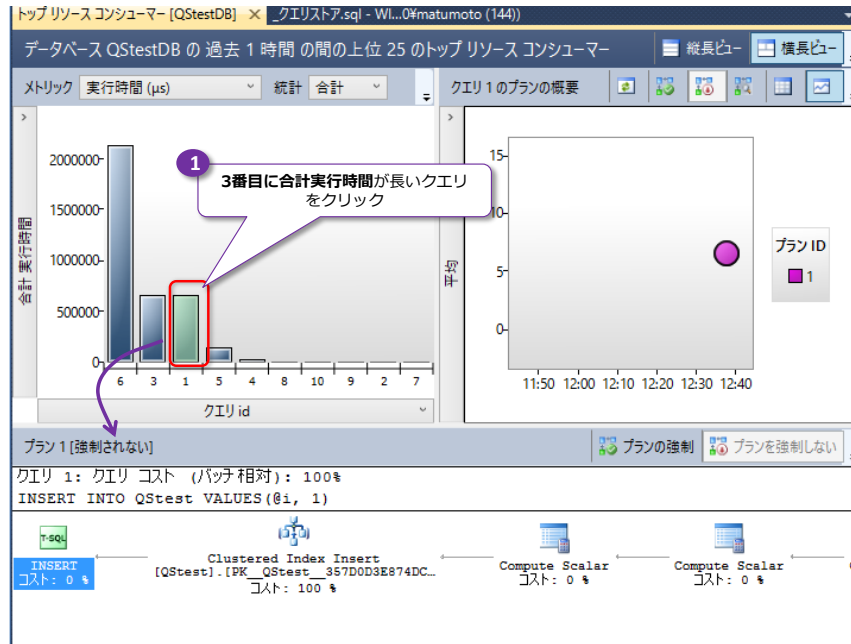
一番実行時間の長いクエリ（グラフの一番左に表示されるクエリ）は、前の手順で実行した「**WHERE col3 = @p1**」のクエリです。

2 番目に長いクエリは、「**SELECT \* FROM CStest**」（一番最初にデータを確認するために実行したもの）であることを確認できます。



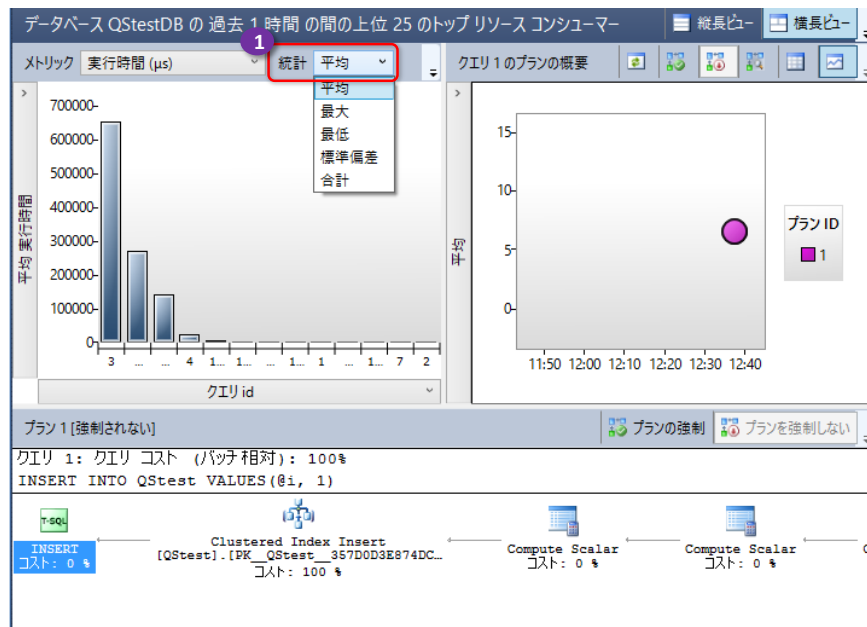
3 番目に長いクエリは、「**INSERT INTO CQtest VALUES(@i, 1)**」(10 万件のデータを

INSERT したときのもの) であることを確認できます。



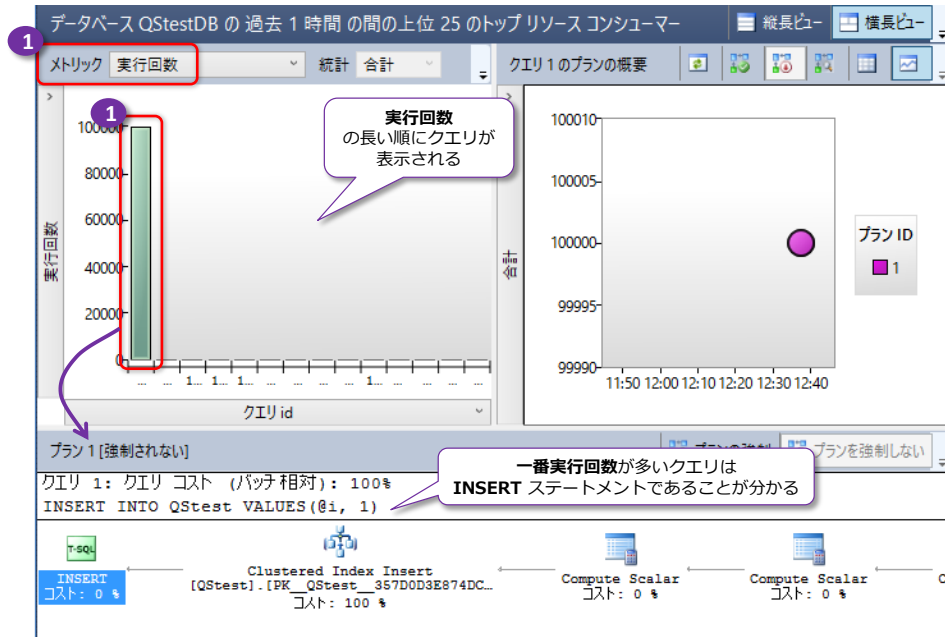
このように、クエリ ストア機能を有効化すると、実行時間の長いクエリを簡単に識別できるようになるので、大変便利です。なお、このグラフにリストされるのは、既定では「過去 1 時間」に実行されたものになっています（表示される期間を変更する方法については後述します）。

- 次に、[総計] で [平均] を選択します。



これで、合計の実行時間ではなく、**平均の実行時間**（例えばクエリが 3 回実行された場合は、その 3 回の実行時間を合計した値になるのではなく、3 回の平均の実行時間）でクエリを並べ替えることができるようになります。この [総計] では、[合計] と [平均] のほかに、[最大] や [最低]、[標準偏差] を選択することもできます。

- 次に、[メトリック] で [実行回数] を選択します。



これで、実行回数の多い順にクエリを並べ替えることができます。実行回数が多いのは、(10万回実行したので) **INSERT** ステートメントになることが分かると思います。このように、実行時間ではなく、実行回数で並べ替えることができるのも、クエリ ストアのメリットです。ここでは、[実行時間] と [実行回数] のほかに、[CPU 時間 (μs)] [論理読み取り]、[論理書き込み]、[メモリ消費量 (KB)]、[物理読み取り] を選択することもできるので、CPU の利用時間の長いクエリや、I/O (読み取り／書き込み) 数の多いクエリ、メモリ消費量の多いクエリで並べ替えることができるようになっています。

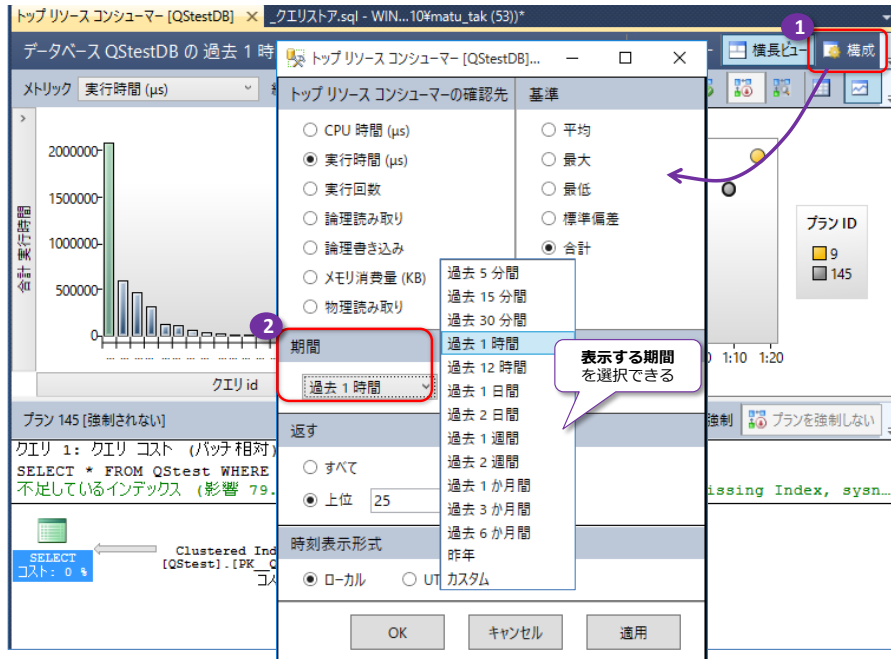
4. 確認後、[メトリック] で [実行時間 (μs)] を選択して、実行時間で並べ替えるように戻しておきます。

なお、CTP 3.2 では、[メトリック] で [実行時間 (μs)] を選択しても、実行時間で並び変わらないことがあり、その場合は、いったんウィンドウを閉じて、再度 [クエリ ストア] フォルダーの [リソースを消費するクエリの上位] をクリックしてみてください。

## ➡ 表示される期間の変更

1. 前述したように、グラフに表示されるクエリは、既定では「過去 1 時間」に実行されたものです。これを変更するには、次のようにツールバーの [構成] (一番右側にあるアイコン) をクリックします。

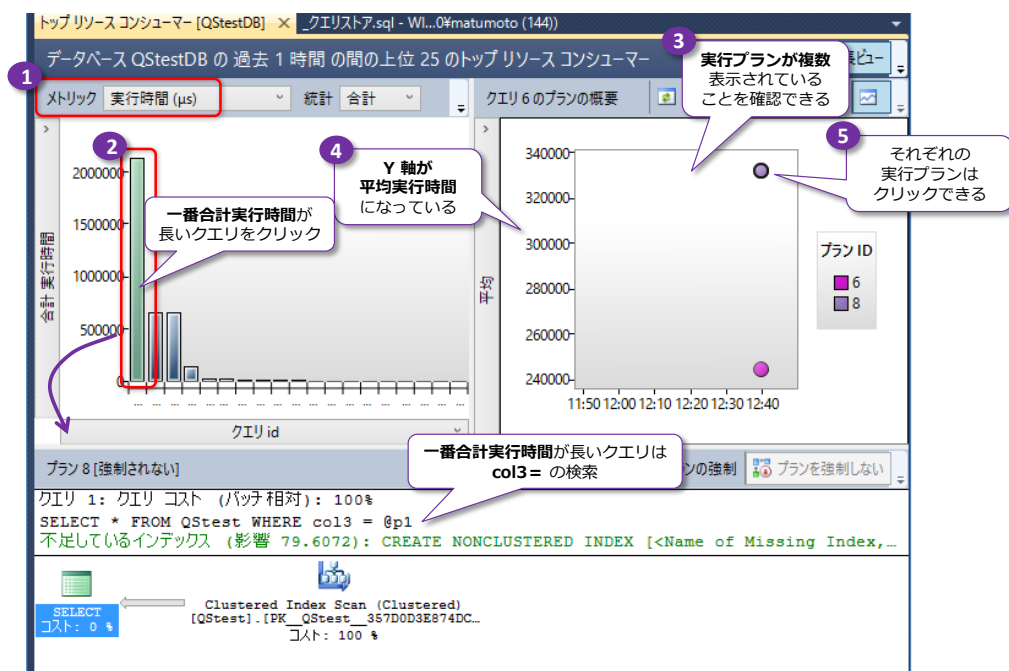




ここで「期間」を「過去 1 週間」や「過去 1 ヶ月間」などに変更することで、過去に戻ることができるようになります（どれだけ過去に戻れるかは、クエリ ストアの最大サイズによって変わってくるので、この設定方法については後述します）。また、この「構成」ダイアログでは、前述の「メトリック」や「総計」（基準）を変更したり、上位何件（既定では 25 件）を表示するのかを設定できたりします。

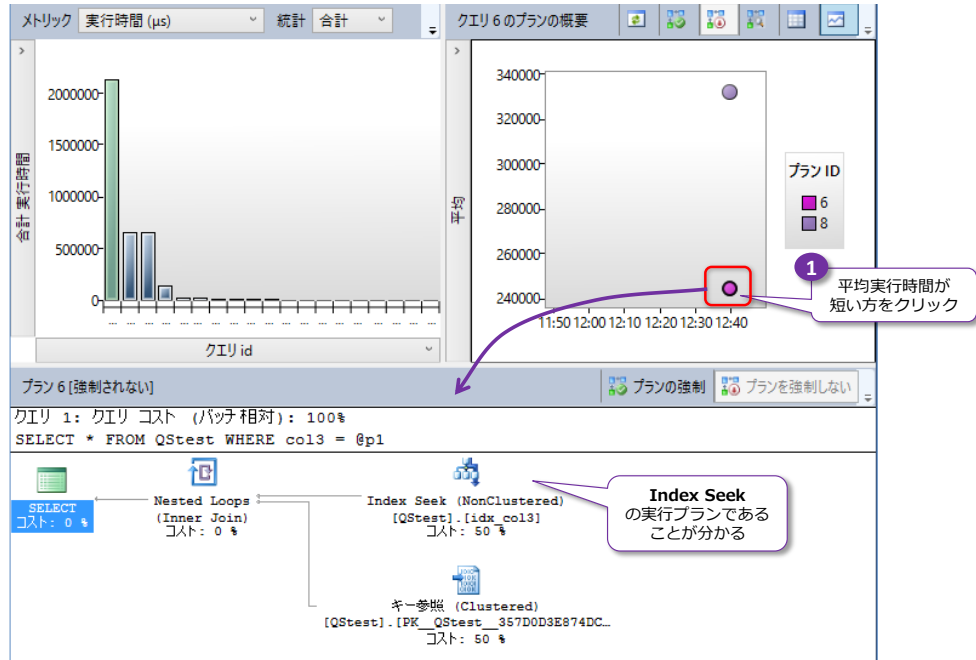
## ➡ 複数の実行プランの確認 ～ col3=@p1 のクエリ～

- 次に、一番実行時間の長いクエリである、前の手順で実行した「WHERE col3 = @p1」を選択します。

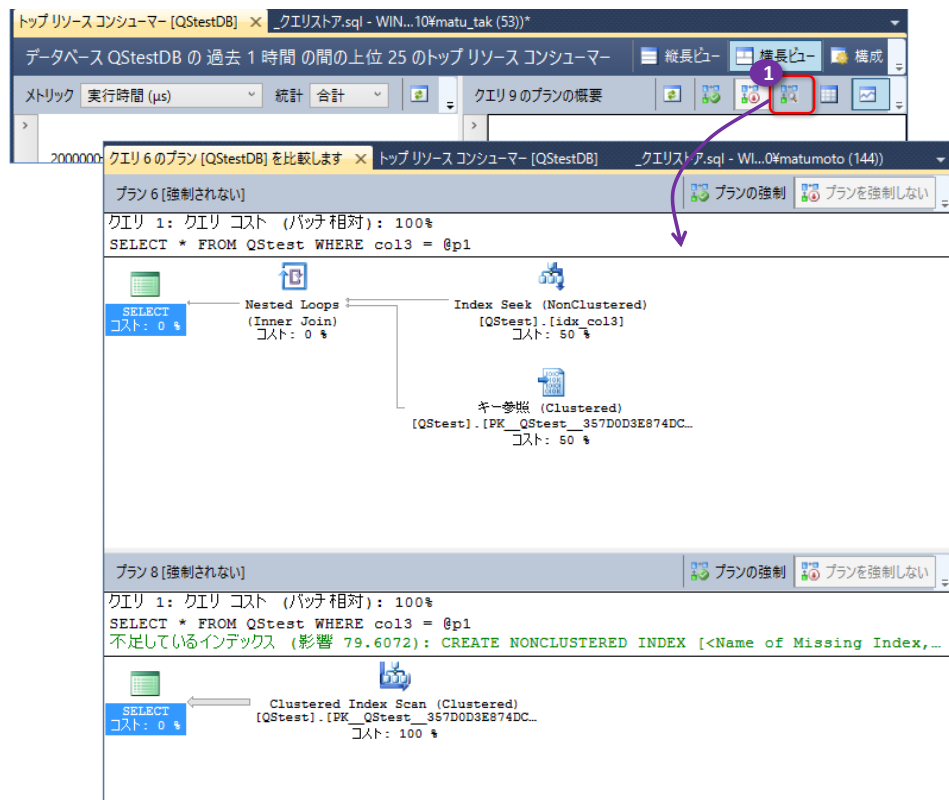




右側のグラフには、実行プランごとの平均実行時間が表示されますが、ここに **2 つの実行プラン**が表示されていることを確認できます。この右側のグラフは、**平均実行時間が Y 軸**になっていて、**平均実行時間が短いほうが Index Seek** の実行プラン、長いほうが **Clustered Index Scan**（全スキャン）の実行プランであることが分かります。



- 次に、ツールバーの「選択したクエリのプランを、別々のウィンドウで比較します」をクリックします。

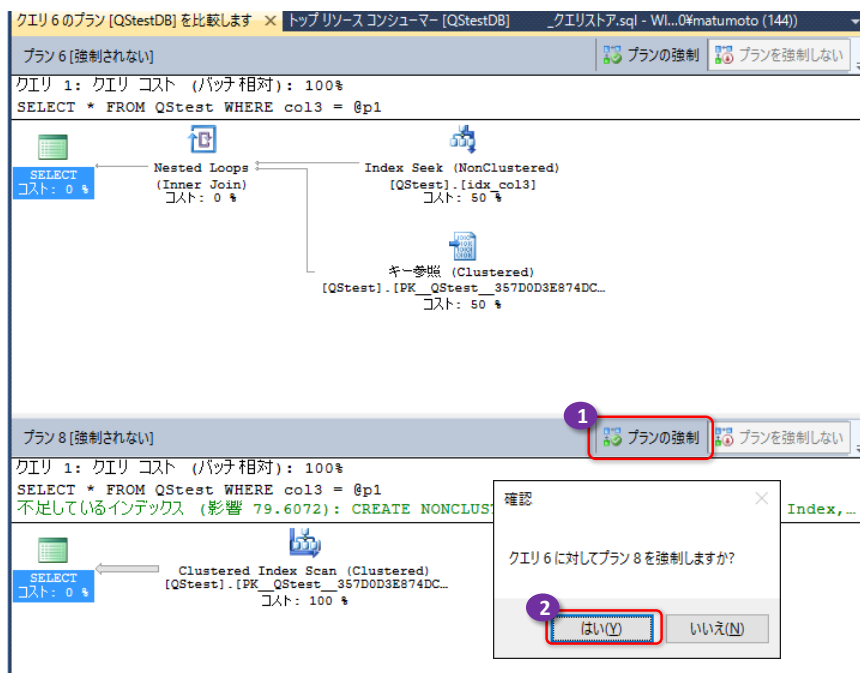


これで、2 つの実行プランを上下で確認できるようになります（比較しやすくなります）。

## ➡ 実行プランの強制

クエリに対して、複数の実行プランが表示される場合は、どちらかの実行プランで必ず実行されるように、実行プランを強制することができます。以前のバージョンの SQL Server では、実行プランの強制には、**プランガイド**という機能を利用していましたが、利用するにはいろいろと面倒なところがありました。しかし、クエリ ストアであれば、わずか数クリックするだけでこれを実現することができます。これも試してみましょう。

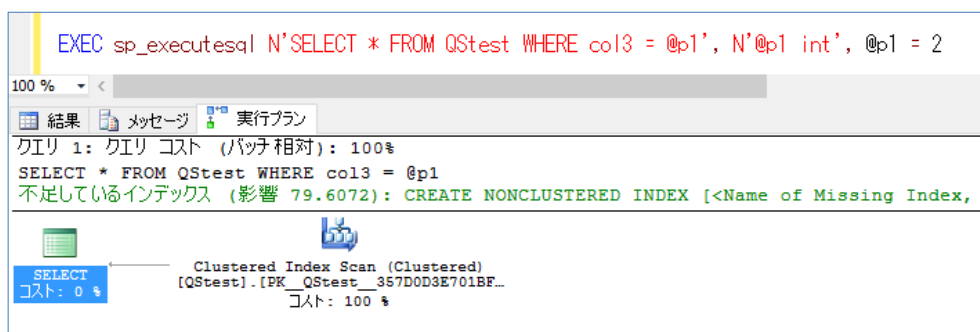
1. まずは、**col3=@p1** のクエリが **Clustered Index Scan** (全スキャン) で実行されるように、プランを強制してみましょう。次のように **Clustered Index Scan** の実行プランに対して、**[プランの強制]** をクリックします。



**[確認]** ダイアログが表示されたら、**[はい]** をクリックします。これで、「**WHERE col3 = @p1**」のクエリが、必ず **Clustered Index Scan** で実行されるようになります。

2. これを確認するために、クエリを実行してみます。このとき、「**@p1**」に与える値には「**2**」を指定します。

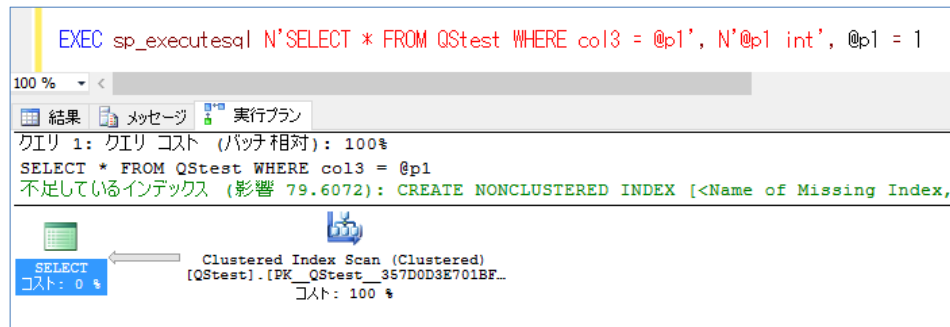
```
EXEC sp_executesql N'SELECT * FROM QStest WHERE col3 = @p1', N'@p1 int', @p1 = 2
```



**Clustered Index Scan** (全スキャン) で実行されるように変わっていることを確認できます。

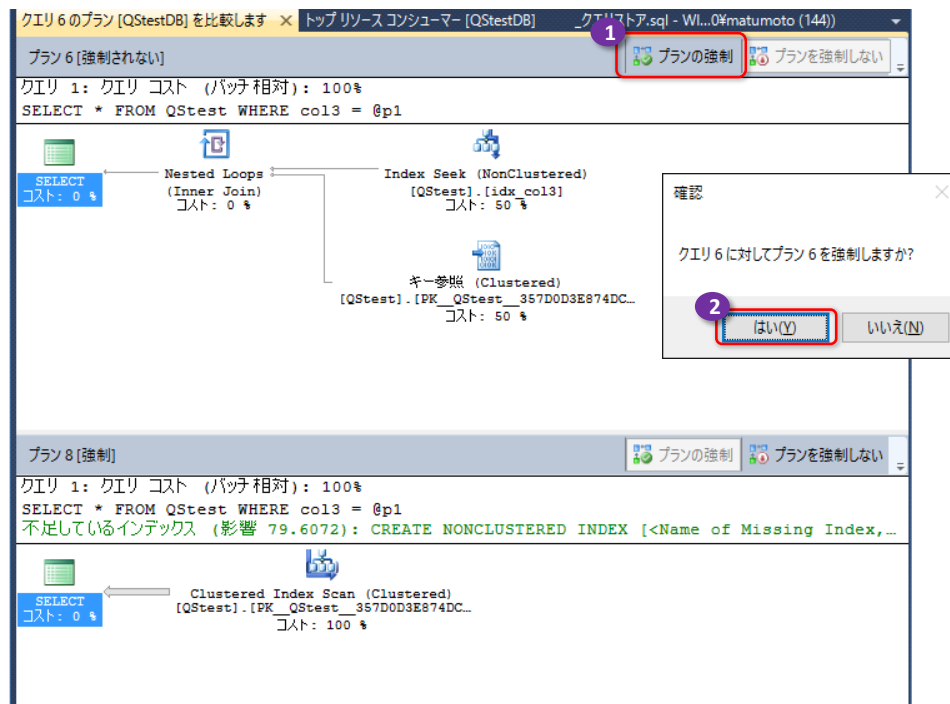
3. 次に、「@p1」に与える値を「1」に変更して実行します。

```
EXEC sp_executesql N'SELECT * FROM QStest WHERE col3 = @p1', N'@p1 int', @p1 = 1
```



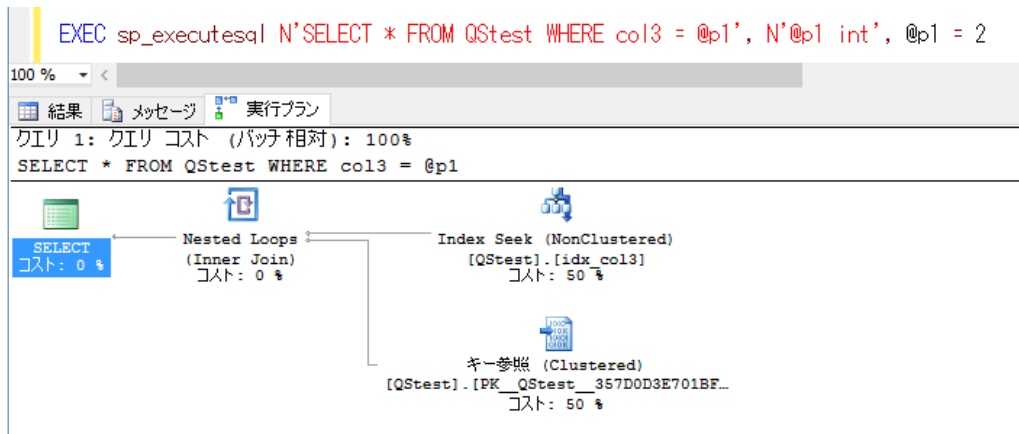
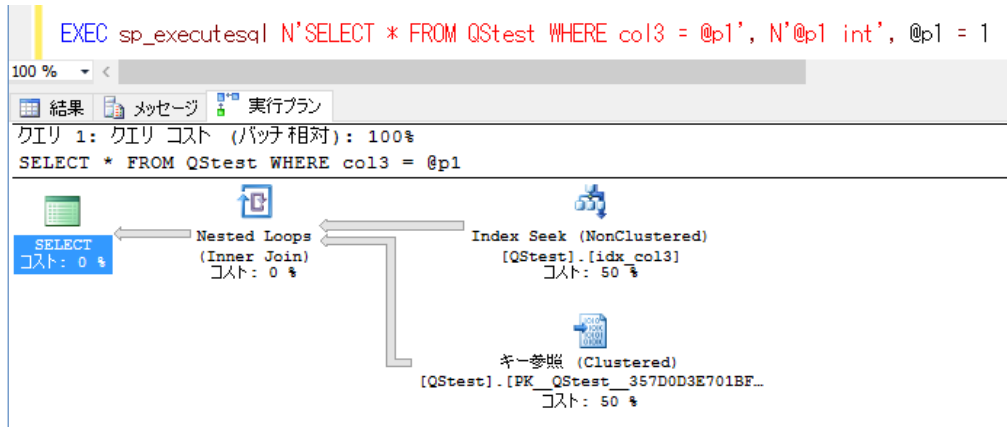
これも、**Clustered Index Scan** で実行されていることを確認できます。このように、クエリストアのプラン強制機能を利用すると、指定した実行プランで必ず実行するように強制できるようになります。

4. 次に、先ほどの実行プランを 2 つ表示しているウィンドウに戻って、今度は Index Seek の実行プランに対して、[プランの強制] をクリックします。



[確認] ダイアログが表示されたら、[はい] をクリックします。

これで、「**WHERE col3 = @p1**」のクエリが、必ず **Index Seek** で実行されるようになります。これも試しておきましょう。



このように、クエリ ストア機能を利用すれば、クエリに対して実行プランを指定することが簡単にできるようになるので大変便利です。

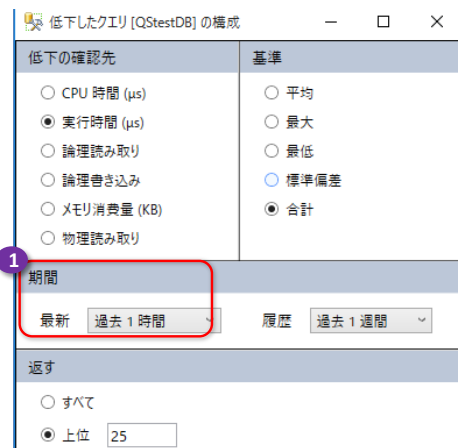
## ➡ 全体のリソース消費量

- 次に、[クエリ ストア] フォルダの [全体のリソース消費量] をクリックしてみます。



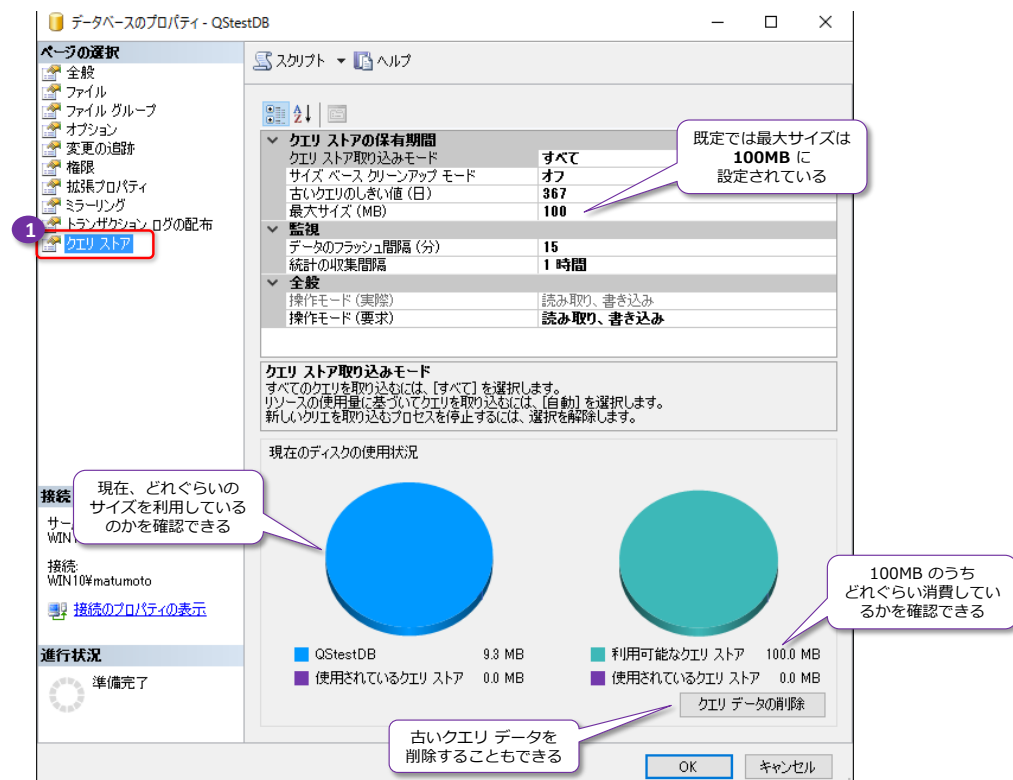
これで、「実行時間」や「CPU 時間」、「実行回数」、「論理読み取り」に関して、時間ごとに確認できるようになります（どの日のどの時間帯に、実行時間が長いクエリが実行されたのか、実行回数が跳ね上がったのはいつなのか、といった情報を確認できるようになります）。

グラフに何も表示されない場合は、ツールバーの「構成」ボタンをクリックして、「期間」を「過去 1 時間」に変更するなどしてみてください（既定値は、過去 1 ヶ月間に設定されているので、何も表示されていないように見えるため）。



## ➡ クエリ ストアの設定

クエリ ストアは、既定では、100MB のサイズになるまで蓄積できるようになっています。これは、[データベースのプロパティ] の [クエリ ストア] ページで確認／変更することができます。



[最大サイズ] を超えた場合には、クエリ ストアは、自動的に「読み取り専用モード」に変更され

て、これ以上クエリの実行履歴を記録しないようになります（サイズ拡大の要因となったものを追跡できるように、読み取り専用モードとして、過去のデータに振り返られるようになっています）。これ以上クエリを記録したい場合には、**[最大サイズ]** を変更するようにします。あるいは、クエリの実行履歴が完了（性能低下の原因となっているクエリを特定できたなど）した場合には、**[クエリデータの削除]** をクリックして、古いデータ（クエリの実行履歴）を削除することもできます。

その他、クエリ ストアについては、オンライン ブックの以下のトピックがお勧めです。

Database Properties (Query Store Page)

<http://msdn.microsoft.com/en-us/library/dn817825.aspx>

...
Database Properties (ChangeTracking Page)
Database Properties (Transaction Log Shipping Page)
Database Properties (Mirroring Page)
Database Properties (Query Store Page)

## Database Properties (Query Store Page)

SQL Server 2016

Access this page from the principal database, and use it to configure and to modify the properties of the database query store. These options can also be configure by using the [ALTER DATABASE SET options](#). For information about the query store, see [Monitoring Performance By Using the Query Store](#).

**Applies to:** SQL Server (SQL Server 2016 Community Technology Preview 2 (CTP2) through [current version](#)), SQL Database V12.

### Options

Enable

Enables and disables the query store.

Operation Mode

Valid values are READ\_ONLY and READ\_WRITE. In READ\_WRITE mode, the query store collects and persists query plan and runtime execution statistics information. In READ\_ONLY mode, information can be read from the query store, but new information is not added. If the maximum allocated space of the query store has been exhausted, the query store will change its operation mode to READ\_ONLY.

## 5.4 Transact-SQL (T-SQL) の強化

SQL Server 2016 では、Transact-SQL も強化されています。

- **R 統合** (4 章で説明)
- **JSON 対応** (4 章で説明)
- **DROP .. IF EXISTS** (オブジェクトが存在している場合に DROP を実行)
- DBCC CHECKDB/CHECKTABLE/CHECKFILEGROUP で **MAXDOP オプション** を利用可能に

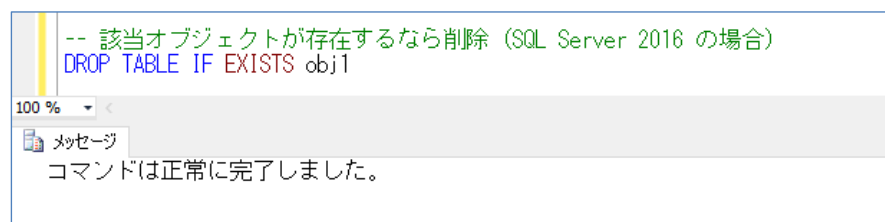
やはり、R 統合 (R スクリプトを実行できるようになった点) と JSON 対応は、大きな強化ポイントです。

もう 1 つ大変便利になったのが「**DROP .. IF EXISTS**」です。これまでの SQL Server では、オブジェクトを削除する場合には、次のようにオブジェクトの存在チェックを行う必要がありました。

```
-- 該当オブジェクトが存在するなら削除 (SQL Server 2014 以前の場合)
IF EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'obj1'))
DROP TABLE obj1
```

SQL Server 2016 からは、「**DROP .. IF EXISTS**」が提供されたことで、上の記述を、次のように記述できるようになります。

```
-- 該当オブジェクトが存在するなら削除 (SQL Server 2016 の場合)
DROP TABLE IF EXISTS obj1
```



テーブルに対して実行したい場合は、「**DROP TABLE IF EXISTS テーブル名**」と記述することで、該当テーブルが存在する場合には削除、そうでない場合には何もしない (エラーも発生しない) という使い方ができます。

そのほか、ストアド プロシージャの場合は「**DROP PROCEDURE IF EXISTS ストアド プロシージャ名**」、ビューの場合は「**DROP VIEW IF EXISTS ビュー名**」という形で利用することができます。また、CTP 3.2 では、**DROP .. IF** でサポートされるのは、AGGREGATE、ASSEMBLY、COLUMN、CONSTRAINT、DATABASE、DEFAULT、FUNCTION、INDEX、ROLE、RULE、SCHEMA、SECURITY POLICY、SEQUENCE、SYNONYM、TRIGGER、TYPE、USER です。



## 5.5 AlwaysOn 可用性グループの拡張

SQL Server 2016 では、AlwaysOn 可用性グループ (Availability Group) も大幅に強化されました。AlwaysOn 可用性グループでは、主に次の新機能が提供されています。

- 自動フェールオーバーを構成できる台数が 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 (透過的なデータ暗号化) 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利用可能 (Windows Server 2016 を利用する場合)
- Standard エディションでも利用可能

### ➡ 自動フェールオーバーを構成できる台数が 2 から 3 に増加

SQL Server 2016 からは、可用性グループで自動フェールオーバーを構成できる台数が **2** から **3** に増えました。

新しい可用性グループ

レプリカの指定

説明  
名前指定  
データベースの選択  
レプリカの指定  
データ同期の選択  
検証  
概要  
結果

ヘルプ

セカンダリレプリカをホストする SQL Server のインスタンスを指定します。

レプリカ エンドポイント バックアップの設定 リスナー

可用性レプリカ(0):

サーバー インスタンス	初期ロール	自動フェールオーバー (上限 2)	同期コミット (上限 3)	読み取り可能なセカンダリ
SERVER1	プライマリ	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	はい
SERVER2	セカンダリ	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	はい
SERVER3	セカンダリ	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	はい

< >

レプリカの追加(A)... Azure レプリカの追加(Z)... レプリカの削除(R)

SERVER3 によってホストされているレプリカの概要

レプリカ モード: 同期コミットと自動フェールオーバー  
レプリカは同期コミットの可用性モードを使用し、自動フェールオーバーと手動フェールオーバーの両方をサポートします。

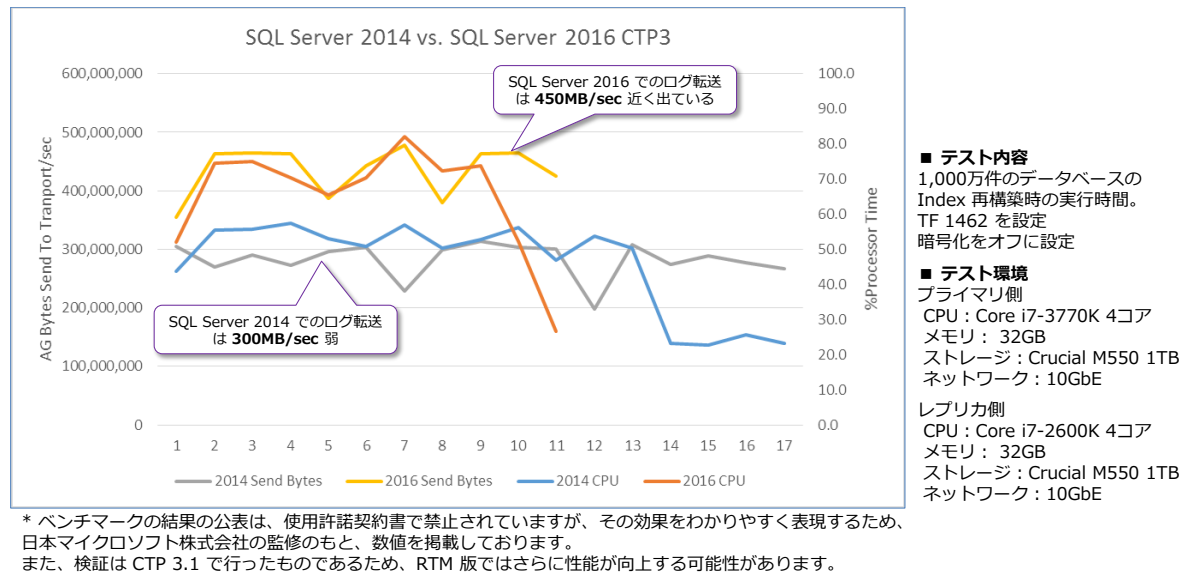
読み取り可能なセカンダリ (0/3)

< 戻る(P) 次へ(N) > キャンセル

## ➡ ログ転送性能の向上（マルチ スレッドで処理）

SQL Server 2016 からは、可用性グループでの**ログ転送の性能が向上**しています。ログ リーダー（ログ転送時のログを読み取るスレッド）や、REDO スレッド（レプリカ側でログを反映するスレッド）がマルチ スレッド対応になったことでこれを実現しています。

実際に、弊社環境で、ログ転送の性能を検証してみたところ、次のような結果を確認することができました。



SQL Server 2016 では、ログ転送スピード（Bytes send to Transport/sec）が **450MB/sec** 近く出ているのに対して、SQL Server 2014 では **300MB/sec 弱**を推移しているという状態でした（CPU 利用率に関しては、SQL Server 2014 が 50%前後を推移して、CPU 性能を活かし切れていない状態であるのに対して、SQL Server 2016 では 70%近くを推移して、CPU を活用している状態）。このグラフは、1,000 万件のデータベースのインデックスを再構築（REBUILD）しているときの状況（パフォーマンス カウンター）になりますが、実行時間は次のようになりました（3 回実行したときの実行時間を計測しました）。

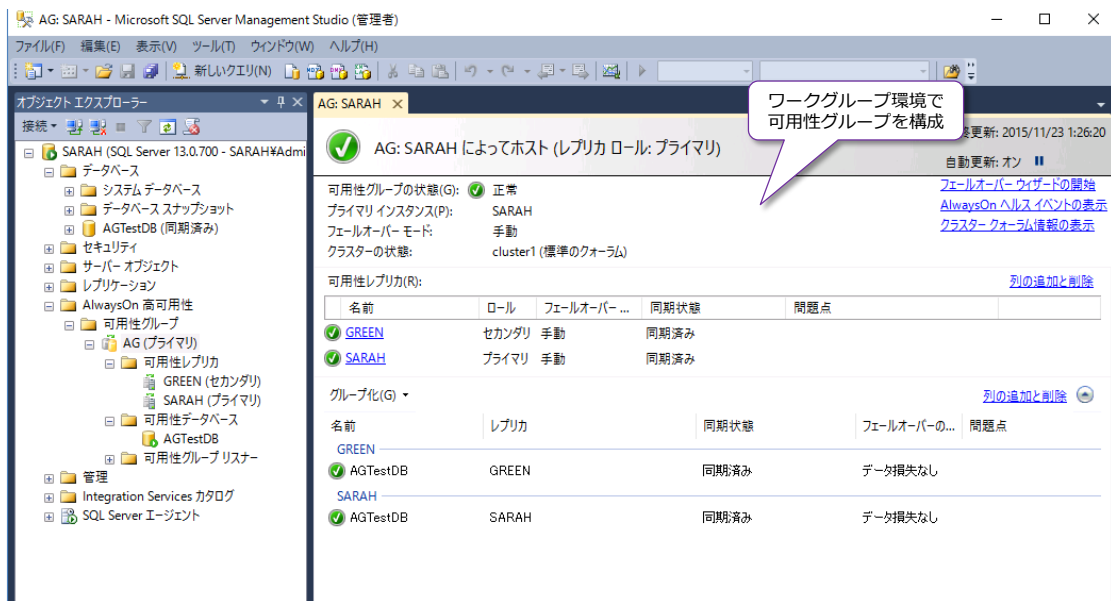
バージョン	平均	1st	2nd	3rd
SQL Server 2014	16.7	17	16	17
SQL Server 2016	11.0	11	11	11
差	5.7			
差%	34.0%			

SQL Server 2016 では 平均 **11 秒**で完了したのに対して、SQL Server 2014 では **16.7 秒**もかかっていて、**34%**の性能向上を確認することができました。

## ➡ ワークグループ環境で利用可能（Windows Server 2016 を利用する場合）

SQL Server 2016 からは、OS に **Windows Server 2016** を利用している場合には、**ワークグループ環境**でも可用性グループを利用できるようになりました（Active Directory が必須ではなく

になりました)。Windows Server 2016 では、ワークグループ環境でも WSFC（フェールオーバー クラスタリング）を構成できるようになったので、この機能を利用して、可用性グループを構成できるようになりました。



## 5.6 BI 関連の強化

SQL Server 2016 では、BI 関連の機能も強化されています。

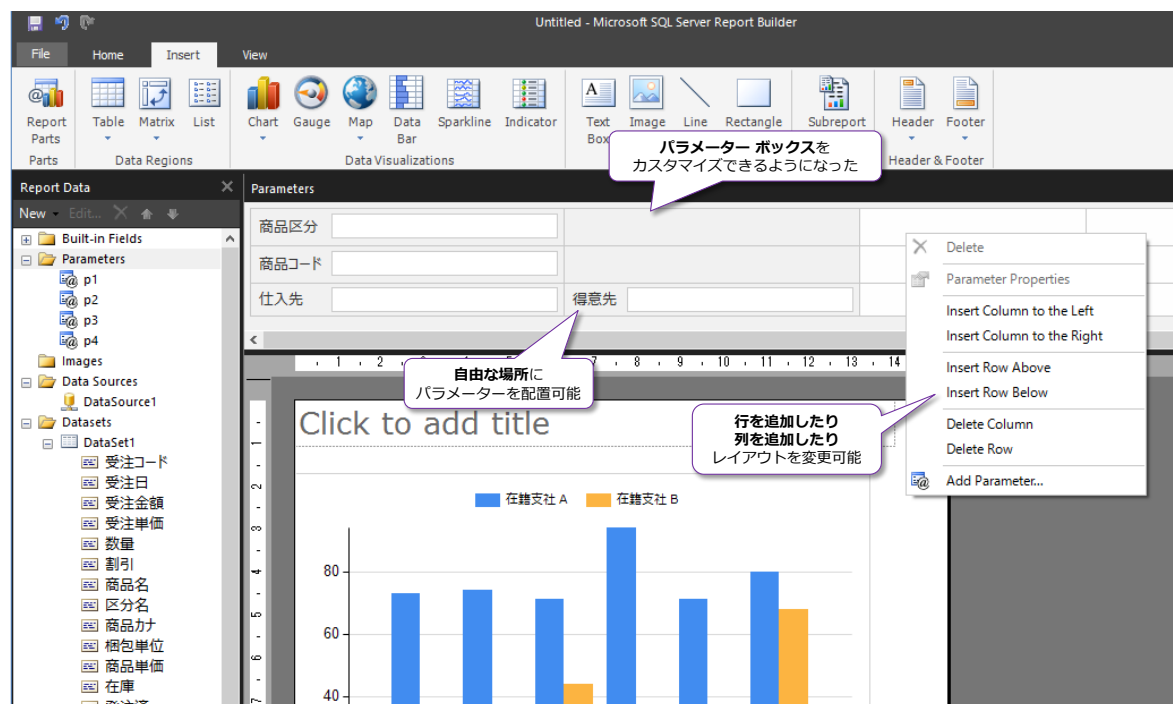
- **Reporting Services** (SSRS) の強化 (Datazen など)
- Integration Services (SSIS) の強化
- Analysis Services (SSAS) の強化
- **MDS** (マスター データ サービス) の強化

### ➡ Reporting Services の強化 (Datazen 統合など)

SQL Server 2016 の Reporting Services は、久しぶりの大きな変更が加わっていて、主に次の新機能が提供されています。

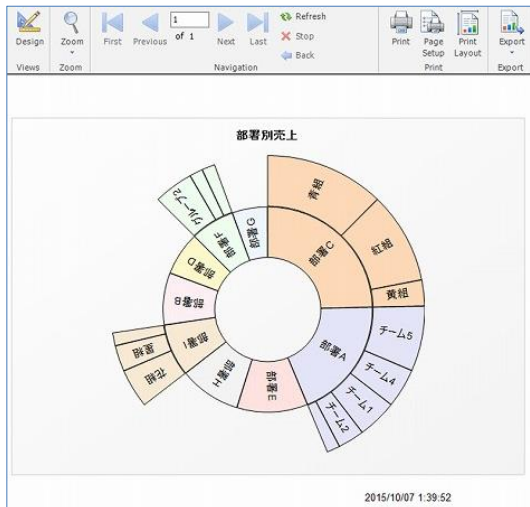
- パラメーター ボックスのカスタマイズ
- 新しいグラフのサポート (サンバースト、ツリーマップ)
- 新しいレポート ビルダー
- PowerPoint レンダリング (PowerPoint ファイルへのエクスポート)
- 印刷コントロールの変更 (ActiveX コントロールが不要に)
- Datazen 統合 (モバイル レポート対応、新しいレポート マネージャーなど)

パラメーター ボックスのカスタマイズは、待望の強化ポイントです。SQL Server 2016 からは、次のようにパラメーターを自由に配置できるようになりました。

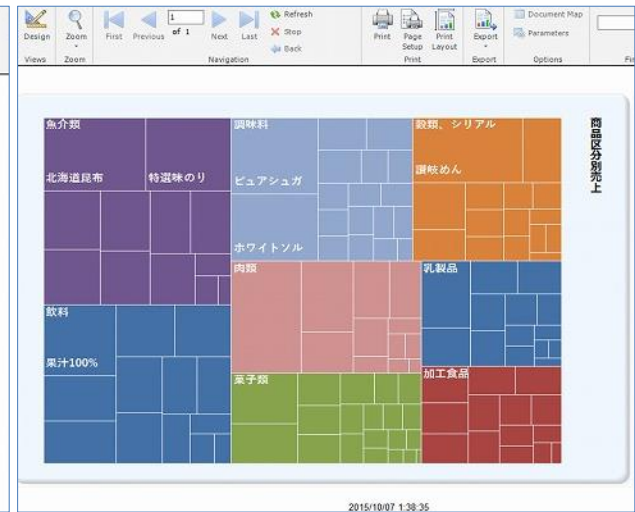


新しいグラフとしては、サンバースト グラフ、ツリー マップ グラフの作成が可能になりました。

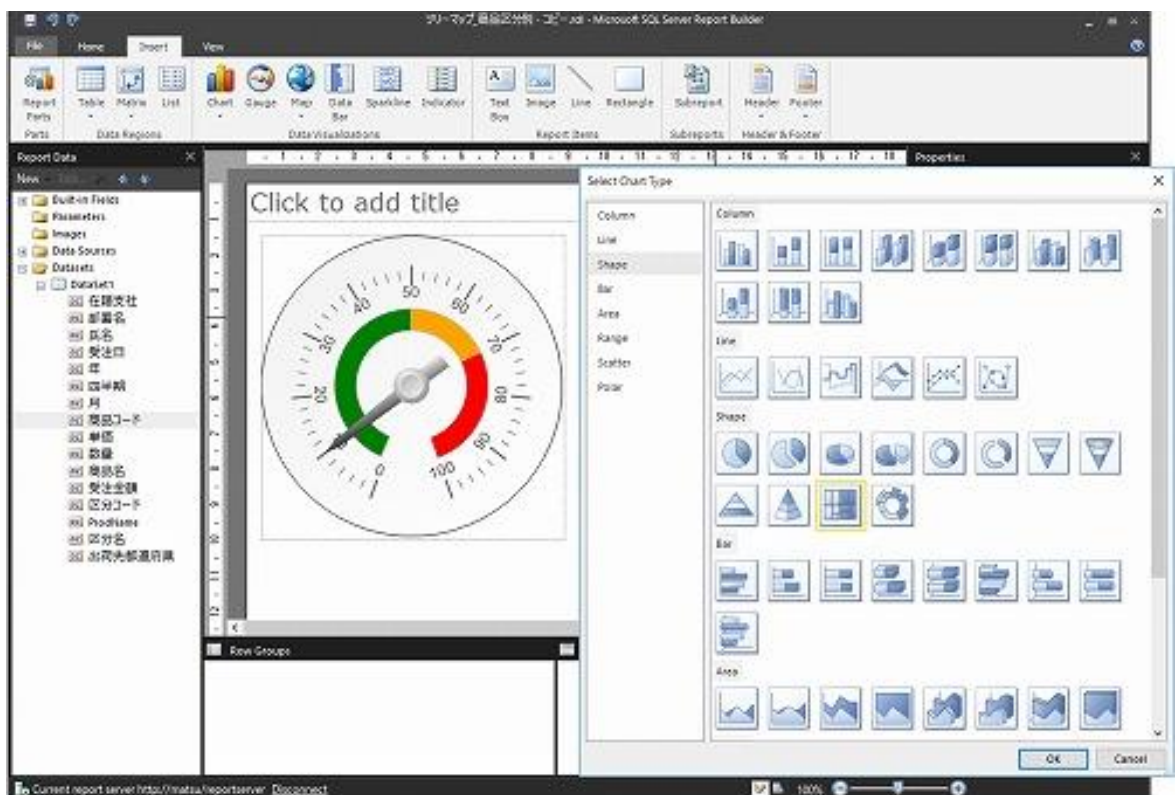
サンバースト グラフ



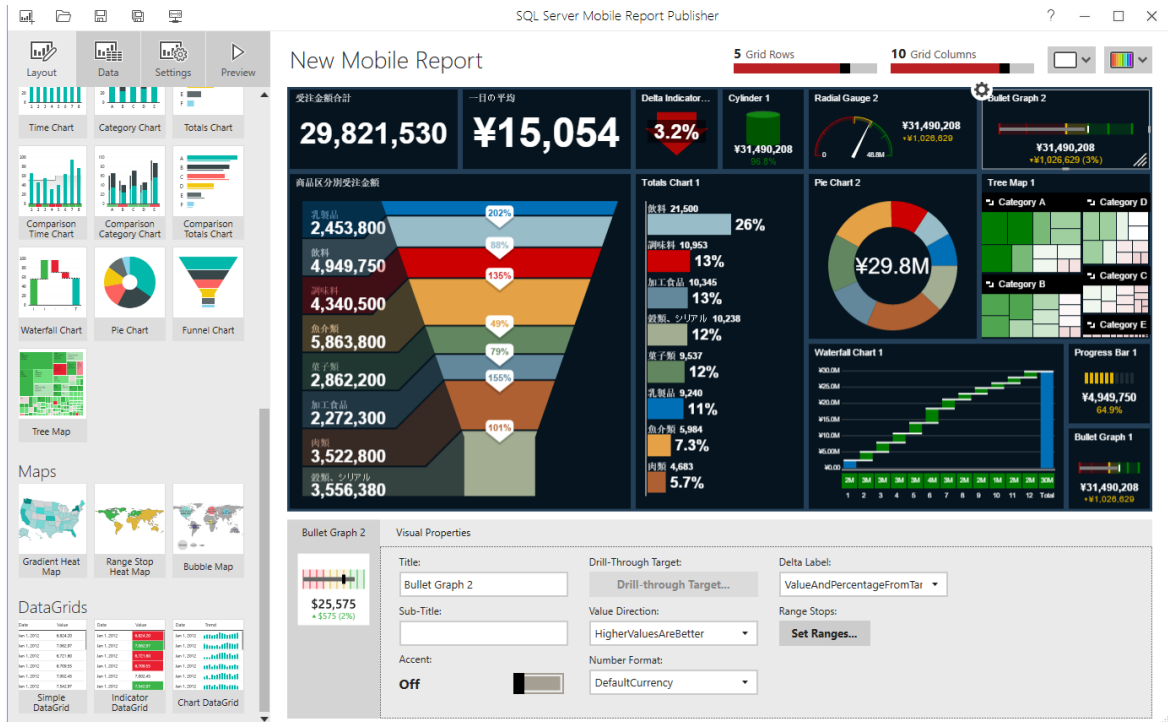
ツリーマップ グラフ



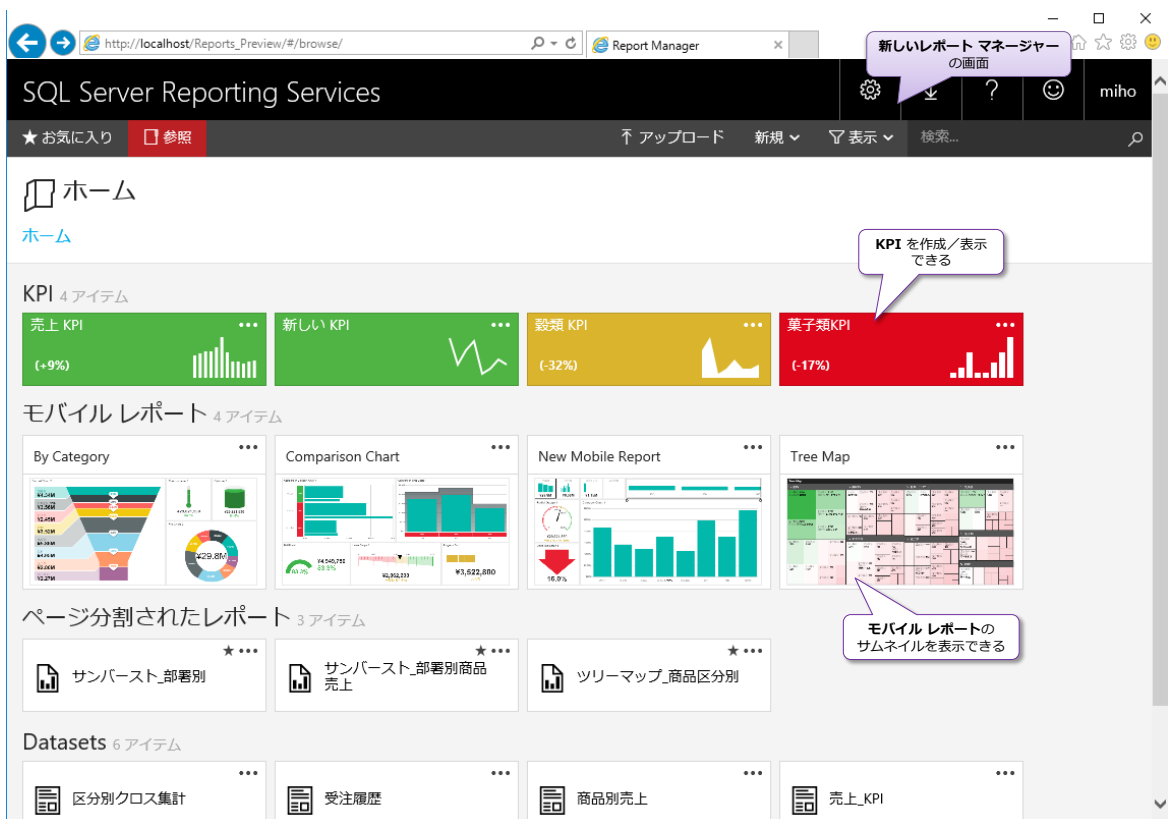
レポート ビルダーも新しくなりました。



そして、Reporting Services での最も大きな変化は、**Datazen の統合**です。**Datazen** は、2015 年にマイクロソフトが買収したレポーティング ソフトウェアで、これが Reporting Services に統合されることになりました。このソフトウェアの一番の特徴は、次のように見栄えの良いレポートを簡単に作成でき、かつ**モバイル レポート**（スマートフォンやタブレット端末など向けのレポート）にも対応している点です。



また、これを受けて、レポート マネージャーも、次のように変わりました。



このように Reporting Services は、SQL Server 2008 R2 以来の大きな進化を遂げているので、これらについては別途自習書を作成する予定です。



## ➡ Integration Services (SSIS) の強化

Integration Services は、主に次の新機能が提供されています。

- Execute SQL タスクで **R スクリプト**のサポート
- **Azure Feature Pack** の提供 (Azure BLOB/Azure HDInsight を操作可能なタスク)
- **Hadoop** (HDFS) のサポート
- インポート/エクスポート ウィザードで **Azure BLOB** 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** プロパティで、バッファ サイズの自動計算
- その他 (OData v4/Excel 2013 データソースのサポートや、新しい Custom Logging Level、パッケージの増分配置 など)

その他の Integration Services の新機能については、オンライン ブックの以下のトピックが参考になります。

What's New in Integration Services

<http://msdn.microsoft.com/en-us/library/bb522534.aspx>

## ➡ Analysis Services (SSAS) の強化

Analysis Services は、主に次の新機能が提供されています。

- DirectQuery の新しい実装/大幅な性能向上
- DBCC コマンドのサポート
- Tabular Model Scripting Language (TMSL) の提供
- AMO (Analysis Services Management Objects) の変更

その他の Analysis Services の新機能については、オンライン ブックの以下のトピックが参考になります。

What's New in Analysis Services

<http://msdn.microsoft.com/en-us/library/bb522628.aspx>



## ➡ MDS（マスター データ サービス）の強化

MDS（マスター データ サービス）は、主に次の新機能が提供されています。

- 性能の向上
- セキュリティの強化
- 各種の機能強化（トランザクションのメンテナンス、多対多リレーションシップ、Excel アドインでのビジネス ルール管理など、多数の機能強化）

MDS には、多数の機能強化があるので、詳細については、オンライン ブックの以下のトピックをぜひ参考に見てみてください。

What's New in Master Data Services (MDS)

<http://msdn.microsoft.com/en-us/library/ff929136.aspx>

## ➡ 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？ SQL Server 2016 CTP 3.2 には、たくさんの新機能が追加されていることを確認できたのではないのでしょうか。インメモリ OLTP とクラスタ化列ストア 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）。

今回は、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だ印象を持ちました（R 統合は、本当にびっくりしました）。

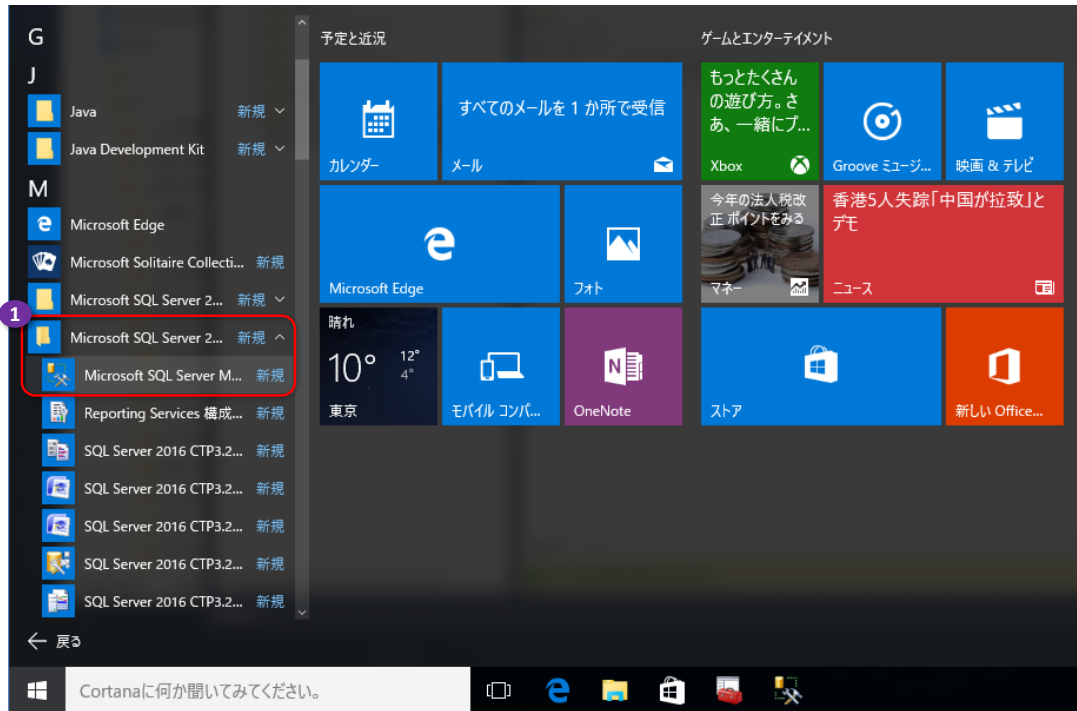
また、SQL Server 2016 では、セキュリティの強化や、既存機能の強化も怠ってはおらず、筆者が特に気に入ったのは「動的データ マスク」と「透過的なデータ暗号化の性能向上 & インメモリ OLTP 対応」、「可用性グループでのログ転送性能の向上」、「ライブ クエリ統計」、「クエリ ストア」です。

セキュリティの強化と性能向上は、トレードオフなところがありますが、透過的なデータ暗号化のインメモリ OLTP 対応は、その 1 つの答え（どちらも両立できる可能性）になるかもしれません。また、可用性グループの性能が向上した点も、大変気に入っています。これまで、弊社のお客様では、ログ転送部分で悩みを抱えることが多かったので、今回の性能向上は本当に期待できます（本文中で検証したものは、性能チューニングが施されていない CTP 3.2 のときに行ったものなので、RTM 版ではさらに性能が向上している可能性もあります）。気になった機能を、皆さんも、ぜひ検証してみてくださいねと思います。

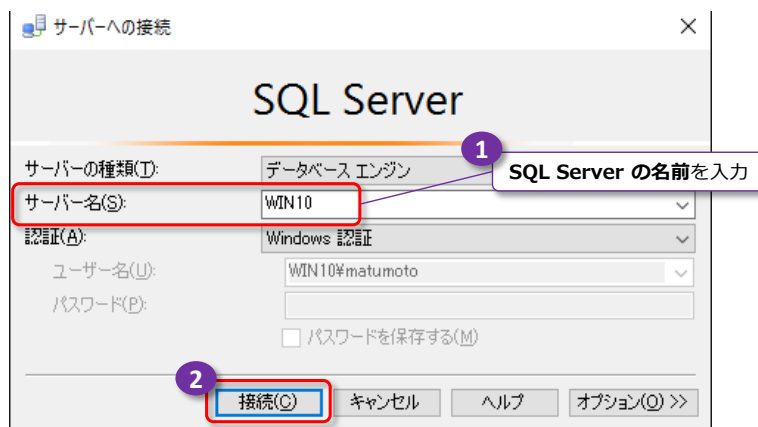
## 5.7 付録：サンプル データベース（NorthwindJ）の作成

この自習書で利用しているサンプル データベース「NorthwindJ」を作成する手順は、次のとおりです。

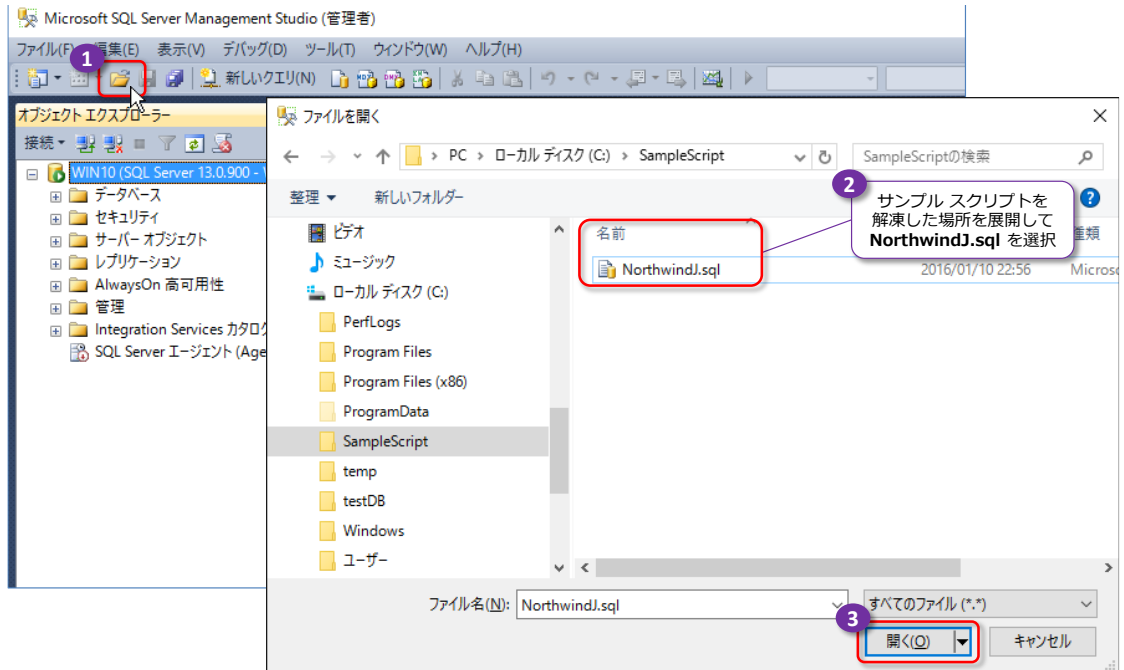
1. まずは、Management Studio を起動するために、[スタート] メニューの [すべてのアプリ] から [Microsoft SQL Server 2016] の [Microsoft SQL Server Management Studio] をクリックします。



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

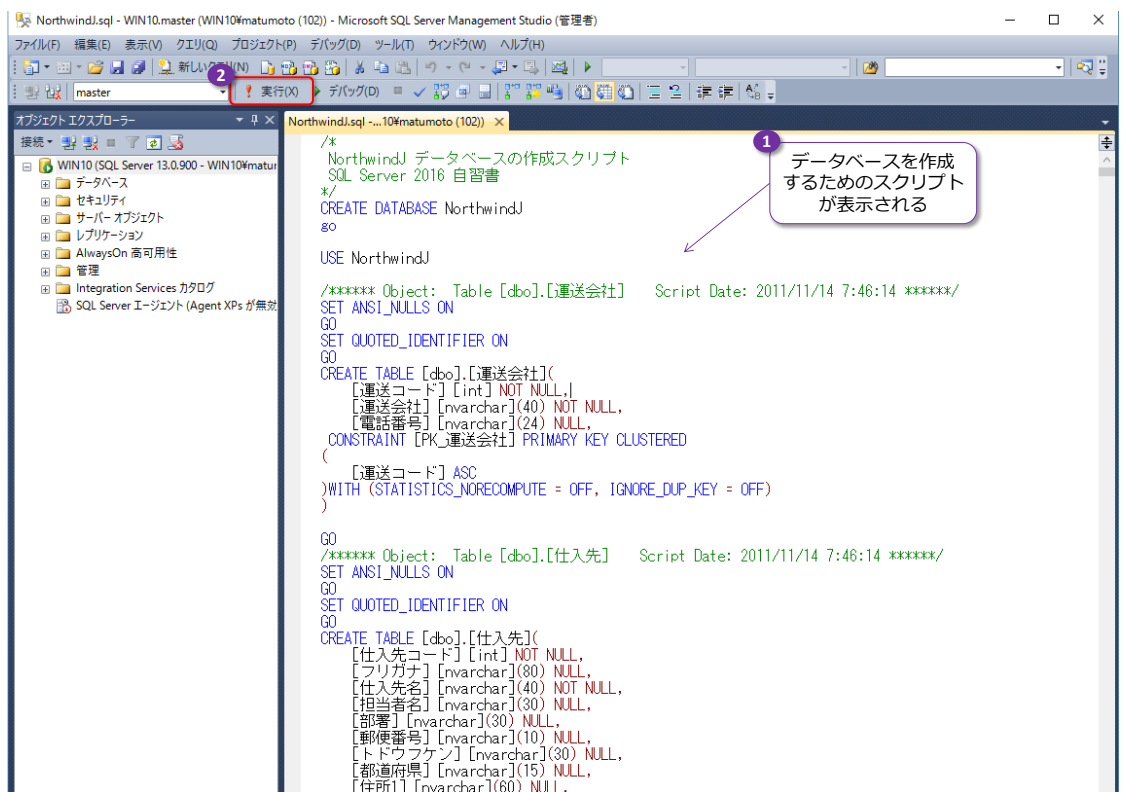


3. 接続完了後、次のようにツールバーの [ファイルを開く] ボタンをクリックします。

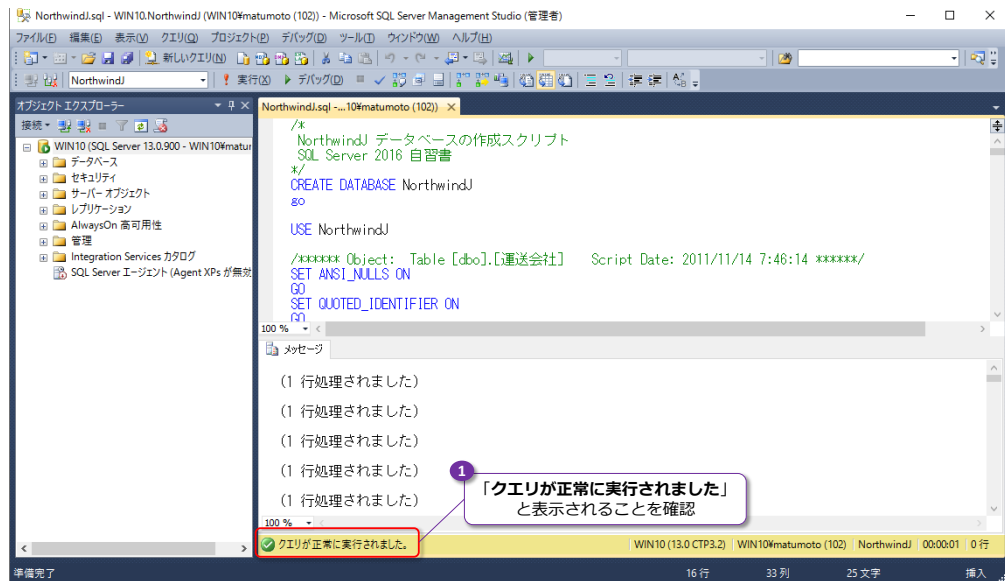


これにより、[ファイルを開く] ダイアログが表示されるので、サンプル スクリプトを解凍したフォルダーを展開して、「NorthwindJ.sql」ファイルを選択し、[開く] ボタンをクリックします。

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



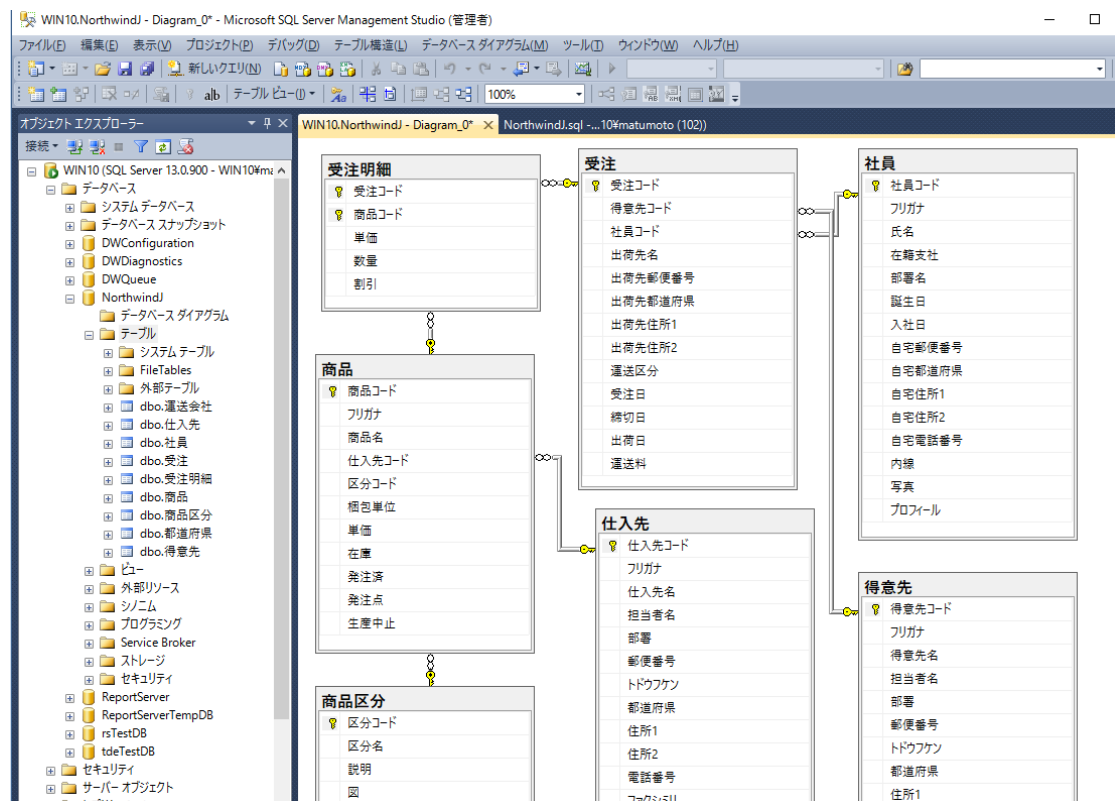
5. 数秒後に実行が完了して、次のように画面下に「クエリが正常に実行されました」と表示されることを確認します。



以上でデータベースの作成が完了です。

## ➤ NorthwindJ データベースの構成

NorthwindJ データベースは、Microsoft Access 2003 に付属のサンプル データベース「Northwind」を SQL Server 上へアップサイズし、この自習書の手順を試すために、一部のデータを加工したものです。具体的なスキーマ構成は次のとおりです。



このデータベースは、商品の販売管理を題材として、「商品」や「商品区分」、「受注」、「受注明細」テーブルなどが格納されています。

## 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算管理／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 2012 の教科書』(ソシム刊)がある。

### 松本崇博 (まつもと・たかひろ)

有限会社エスキューエル・クオリティ 取締役 Microsoft MVP for SQL Server (2004 年 4 月～)

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

SQL Server の BI システムとパフォーマンス チューニングを得意とするコンサルタント。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。