Microsoft

\Orchestrating a brighter world NEC

SQL Server 2019 自習書

SQL Server 2019 を体験しよう！

2020 年 3 月

本書に記載した情報は、公開日時点での Microsoft Corporation（以下 Microsoft）、および NEC Corporation（以下 NEC）の考え方を表すものです。Microsoft および NEC は絶えず変化する市場に対応しなければならないため、ここに記載した情報に対していかなる責務を負うものではなく、記載の内容を約束するものではありません。本書は情報提供のみを目的としており、Microsoft および NEC は明示的または暗黙的を問わず、本書にいかなる保証も与えるものではありません。
また、本書に記載のいかなる部分についても、転載や検索システムへの格納または挿入を行うことは、どのような形式または手段（電子的、機械的、複写、レコーディング、その他）、および目的であっても禁じられています。これらは著作権保護された権利を制限するものではありません。
Microsoft、SQL Server、Visual Studio、Windows、Windows Server 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。
本書内での引用（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。
本書の情報やアドバイスは執筆時点での正確性に努めていますが、万が一誤りがあった場合にも法的責任を負うことはできません。
©Copyright 2020 Microsoft Corporation, NEC Corporation. All rights reserved.

◆ 目次

Introduction 6
まえがき 6
◆ 本書の対象者と目的 6
◆ 本書の構成 6
SQL Server 2019 7
◆ SQL Server 2019 の登場 7
◆ SQL Server 2019 へのアップグレード 12
自習の準備 16
◆ SQL Server 2019 評価版のインストール 16
◆ クライアントツールのインストール 16
◆ サンプルデータベースの作成 18
◆ サンプルコードのダウンロード 21
1. Intelligent Performance 22
1.1. Intelligent Query Processing 22
➢ Batch mode on rowstore 25
➢ Row mode memory grant feedback 30
➢ Table variable deferred compilation 40
➢ Approximate query processing with APPROX_COUNT_DISTINCT 48
➢ Scalar UDF Inlining 52
1.2. In-Memory Database 59
➢ Memory-Optimized TempDB Metadata 60
➢ Enhanced PMEM support（Hybrid Buffer Pool & Enlightenment） 67
1.3. Performance Enhancements 96
➢ Optimize for sequential key insert performance 97
➢ Reduced recompilations for workloads 103
➢ Concurrent PFS updates 108
➢ Indirect checkpoint scalability 116
➢ Scheduler worker migration 119
2. Mission-critical security 121
➢ Always Encrypted with secure enclaves 122
➢ Data Classification and auditing built-in 151
➢ TDE scan suspend and resume 160
➢ Certificate management in SQL Server Configuration Manager 166
3. High availability 176
➢ Accelerated Data Recovery 178

- Index Operation Enhancements 201
- Always On Availability Group Enhancements 208
- Database Snapshots Enhancements 225

4. User experience enhancement for DBA 227

4.1. Query Profiling 228
- Lightweight Query Profiling 228

4.2. Plan cache 237
- Plan cache granular control 237

4.3. Columnstore indexes 241
- Estimate compression for Columnstore indexes 241
- Columnstore stats in DBCC CLONEDATABASE 247

4.4. Error messages 254
- Verbose truncation warnings 254

4.5. Page Information 258
- sys.dm_db_page_info 258

4.6. Query Store 263
- Custom capture policy for Query Store（QUERY_CAPTURE_POLICY） 263

4.7. Statistics 265
- Diagnostics on statistics(WAIT_ON_SYNC_STATISTICS_REFRESH, SELECT (STATMAN)) 265

4.8. Resource Governor 266
- Memory grant percent in Resource Governor 266

4.9. Setup options 267
- MAXDOP and Memory Config during Setup 267

5. Enhanced user experience for Developers 269

5.1. Graph 269
- Edge constraint in Graph Database 271
- New graph function - SHORTEST_PATH 275
- Use derived table or view aliases in graph match query 278
- Using the T-SQL MERGE statement to support scenarios like upsert 284

5.2. Unicode support 288
- Support for UTF-8 character encoding 288

5.3. Language extensions 293
- Extensibility Framework SQL Server Language Extensions 293

5.4. Spatial 303
- New spatial reference identifiers (SRIDs) 303

6. Platform choice 304

6.1. SQL Server 2019 on Linux 304

➢ SQL Server 2019 on Linux の強化ポイント 306
➢ Machine Learning Services on Linux 312
6.2. SQL Server 2019 on Linux Container with Docker 325
➢ Why SQL Server Containers? 325
➢ コンテナーのライフサイクル 326
➢ SQL Server コンテナーの基本操作 327
➢ Non-root SQL Server Container 345
7. Scalable big data solution 353
7.1. BDC の機能 354
➢ データ仮想化 354
➢ データレイク 354
➢ データマート 354
➢ ビッグデータ処理と機械学習 355
➢ アプリケーション展開 355
➢ 運用管理 356
7.2. BDC のアーキテクチャ 357
➢ コントローラー 357
➢ コンピュートプール 358
➢ SQL Server マスターインスタンス 359
➢ データプール 360
➢ ストレージプール 361
➢ アプリケーションプール 363
7.3. やってみよう 364
➢ BDC の展開と管理 365
➢ データ仮想化（PolyBase）によるサイロ化されたデータの統合 375
➢ データレイク（Storage Pool）の展開と外部 HDFS との統合 388
➢ Jupyter と Spark MLlib による Python 機械学習 404
➢ スケールアウト可能なデータマート（Data Pool）の展開と PowerBI によるデータ可視化 413
➢ アプリケーションのデプロイと REST API によるアプリケーションの呼び出し 427
あとがき 434

Introduction

まえがき

◆ 本書の対象者と目的

本書は SQL Server の基本的な知識をお持ちのデータベース管理者や開発者向けに、SQL Server 2019 を包括的に知ってもらうことを目的に執筆しました。とくに、実際の現場が抱えている課題を SQL Server 2019 がどのように解決するのかをお伝えするために、私たちチームのこれまでの経験や知見を盛り込んでいます。また SQL Server 2019 に実際に触れて体験してもらうためのサンプルコードを公開しています。これらを通じて SQL Server がより活用されることを願っています。

2020 年 3 月
日本電気株式会社
NEC ソリューションイノベータ株式会社

◆ 本書の構成

本書は次のセクションに分けて構成しています。

No	セクション名	概要
1	Intelligent Performance（性能性）	パフォーマンス強化に関するトピックを取り上げます。
2	Mission-Critical Security（機密性）	セキュリティ強化に関するトピックを取り上げます。
3	High Availability（可用性）	可用性強化に関するトピックを取り上げます。
4	Enhanced user experience for DBAs	主にデータベース管理者向けの機能強化に関するトピックを取り上げます。
5	Enhanced user experience for Developers	主に開発者向けの機能強化に関するトピックを取り上げます。
6	Platform choice	Linux やコンテナーへの対応強化に関するトピックを取り上げます。
7	Scalable big data solution	従来の RDB の枠を超えた SQL Server 2019 の新機能「Big Data Cluster」に関するトピックを取り上げます。

各セクションは独立しているため、興味のあるトピックから順に読み進めることができるように構成しています。本章では各セクションの解説に入る前に SQL Server 2019 の概要とサンプルコードによるハンズオンの準備について解説します。

◆ SQL Server 2019 の登場

2019 年 11 月に SQL Server の最新バージョンとして SQL Server 2019 がリリースされました。
業界をリードするパフォーマンスとセキュリティを備えたデータベース製品であるとともに、蓄えたデータからインテリジェンスを引き出す製品としてリリースされました。

上図のペンギンのシンボルが示す通り、1 つ前のバージョンである SQL Server 2017 では Linux に対応するという驚きと興奮がありました。その Linux への対応により SQL Server のプラットフォームがコンテナーや Kubernetes にも広がり、そうして生まれたのが Big Data Cluster という SQL Server 2019 を象徴するひとつの大きな新機能です。

これは、長らくリレーショナルデータベースとして認知されてきた SQL Server が、データプラットフォームを提供する製品として進化したと言えるのではないでしょうか。Big Data Cluster では、データプラットフォームに必要とされる様々なコンポーネントが Kubernetes 上にワンパッケージで提供され、データをつなぎ、蓄え、そこからインテリジェンスを引き出すことをアシストします。

このように SQL Server の進化はますます加速している印象があります。下図は SQL Server 2019 に至るまでの進化の歴史です。

こうした進化の歴史の中には、Linux への対応や Big Data Cluster のような大きな新機能だけではなく、細かく地道な改善も多く取り込まれています。ここで今回の SQL Server 2019 を例に、 3 つの観点に絞って強化ポイントを取り上げてみます。

Intelligent Performance

最初は Intelligent Performance です。これは「アプリケーションをできる限り変更せずに、クエリのパフォーマンスを向上させること」をテーマに開発された機能群を示しています。本書のサンプルコードを通じて、その改善効果を実感してほしい機能群です。

Mission-critical Security

次に Mission-critical Security です。SQL Server 2008 で導入された TDE、SQL Server 2016 で導入された Always Encrypted、これらは強力なセキュリティ機能である一方で、導入の制約に引っかかってしまうケースがあったのではないでしょうか。こうした制約が SQL Server 2019 では大きく緩和され、導入の敷居が下がっています。

High Availability

最後は High Availability です。特に ADR の導入とオンラインインデックス処理の強化が注目のポイントです。こちらも実運用で悩まされていた問題を解決する機能です。

SQL Server が常に利用可能

課題

- ログが制御できないほど大きくなっている
- 長時間かかるロールバックによりアプリケーションがブロックされる
- 長時間かかる復旧によりダウンタイムが長引く
- インデックス操作に大量のリソースが必要

ソリューション

- 高速化されたデータベース復旧 (ADR)
- オンライン インデックス拡張機能
- 可用性グループ拡張機能

その他にも様々あります。例えば、次のような利用者の声（https://feedback.azure.com/forums/908035-sql-server）を元に導入された新たな強化もあります。地味でありながらもとても実用的、待ち望んでいた機能です。

お客様の声

ユーザー エクスペリエンス

- お客様からのフィードバック項目の第 1 位: 文字列の切り詰め
- 設定時の MAXDOP およびメモリの構成
- リソース ガバナーでのメモリ許可パーセント
- 列ストア インデックスの圧縮の推定

診断

- 新しい組み込みの T-SQL によるページ待機のトラブルシューティング
- 自動統計のブロックの診断
- クエリ ストア用のカスタム キャプチャ ポリシー
- DBCC CLONEDATABASE の列ストアの統計情報

パフォーマンス

- TempDB ワークロードの再コンパイルの削減
- 間接的チェックポイントのスケーラビリティ
- PFS 更新の同時実行
- ワーカーのスティール
- インラインのログ書き込み

```
String または binary データは切り詰められました。

テーブル'%.*ls' の列'%.*ls' で
String または binary データは切り詰められました。
切り詰められた値:  '%.*ls'
```

```
SELECT page_info.*
FROM sys.dm_exec_requests AS d
   CROSS APPLY
   sys.fn_PageResCracker(d.page_resource)
   AS r
   CROSS APPLY
   sys.dm_db_page_info(r.db_id,
   r.file_id, r.page_id,'DETAILED')
      AS page_info;
```

こうした SQL Server 2019 で導入される機能を可能な限り網羅的に本書で解説します。特に実際の現場が抱えている従来の課題をこうした機能がいかに解決できるかをお伝えすることに努めました。加えて SQL Server 2019 を実際に触って体験してもらうために、多くの機能に対してサンプルコードを作成しました。

ぜひ、現在の課題を解決しうる機能が無いか、本書で解説するすべての新機能に目を通していただけると幸いです。

具体的な機能の理解に入る前に SQL Server 2019 を検討するきっかけを知りたいという場合には次の情報が参考になるかもしれません。

◆ SQL Server 2019 へのアップグレード

ここでは既にお使いの SQL Server のアップグレードを検討しているユーザーに向けて「既存 SQL Server のライフサイクル」、「SQL Server 2019 のエディション」、「SQL Server 2019 のライセンス」の 3 点を補足します。

SQL Server のライフサイクルとアップグレード

SQL Server はバージョンごとに少なくとも 10 年間のサポートがあります。この 10 年間のライフサイクルには「メインストリームサポート」期間である 5 年間と「延長サポート」期間である 5 年間が含まれます。

「メインストリームサポート」では機能、パフォーマンス、スケーラビリティ、およびセキュリティ更新プログラムが含まれており、「延長サポート」ではセキュリティ更新プログラムのみが含まれます。

提供されるアップデート	メインストリームサポート	延長サポート
機能	●	–
パフォーマンス	●	–
スケーラビリティ	●	–
セキュリティ更新プログラム	●	●

SQL Server のライフサイクルの詳細は、次の Docs を確認ください。

◇ SQL Server のライフサイクルについて
https://docs.microsoft.com/ja-jp/sql/sql-server/end-of-support/sql-server-end-of-life-overview?view=sql-server-ver15

次の表は SQL Server の各バージョンのライフサイクルの日付を示しています。

バージョン	メインストリームサポート終了日	延長サポート終了日
SQL Server 2008 & 2008 R2	2014/07/08	2019/07/09
SQL Server 2012	2017/07/11	2022/07/12
SQL Server 2014	2019/07/09	2024/07/09
SQL Server 2016	2021/07/13	2026/07/14
SQL Server 2017	2022/10/11	2027/10/12
SQL Server 2019	2025/01/07	2030/01/08

最新の製品ライフサイクルの日付は次のサイトで確認することができます。

◇ 製品のライフサイクルの確認
https://support.microsoft.com/ja-jp/lifecycle/search?alpha=SQL%20Server

「メインストリームサポート」から「延長サポート」へ遷移し、「延長サポート」の終了によって該当バージョンの SQL Server のサポートは終了になります。サポートを終了する SQL Server を継続して利用するには次に示すいくつかのパスがあります。

1 つの選択肢は SQL Server との互換性の高い Azure SQL Database へ移行することです（上図の 1 番上）。

もう 1 つの選択肢は、既存のバージョンに対して追加の延長サポートを利用することです。追加の延長サポートの利用には有償によるもの（上図の 1 番下）と、Azure 仮想マシンへ移行することを条件とした無償によるもの（上図の上から 2 番目）があります。それぞれ詳細は次のサイトで確認してください。

◇ 有償の追加延長サポートについて
https://www.microsoft.com/ja-jp/cloud-platform/extended-security-updates

◇ Azure での無償の追加延長サポートについて
https://docs.microsoft.com/ja-jp/azure/virtual-machines/windows/sql/virtual-machines-windows-sql-server-2008-eos-extend-support

残りの選択肢は、より新しいバージョンへのアップグレードです（上図の下から 2 番目）。

本書がターゲットとするのは、新しいバージョンである SQL Server 2019 へのアップグレードになります。以降で SQL Server 2019 のエディションとライセンスを補足します。

SQL Server 2019 のエディション

従来と同様に利用シナリオに応じて 5 つのエディションが提供されています。

エディションの区別は主として「求められるスケーラビリティ」です。
従来は「求められるスケーラビリティ」に加えて「必要とされる機能」でも区別されていましたが、SQL Server 2016 SP1 のタイミングで大きく方針が変わりました。それは「すべてのエディションで一貫したプログラマビリティ エクスペリエンスを提供することで、成長に合わせてエディションやクラウドにまたがる高度なアプリケーションを構築できるようにすること」です。この方針に従い、上位のエディションでのみ提供されていた機能の多くが、下位のエディションにも提供されるようになりました。

エディションごとのスケーラビリティと機能の詳細は、次の Docs を確認ください。

◇ エディションごとのスケーラビリティと機能
https://docs.microsoft.com/ja-jp/sql/sql-server/editions-and-components-of-sql-server-version-15?view=sql-server-ver15

これまでに解説してきたように、製品としての機能強化だけではなく、エディションの方針も進化しています。

SQL Server 2019 のライセンス

ここでは基本的な「有償・無償の区分」と「ライセンスモデルの区分」についてのみ概要を解説します。

❖ 有償・無償の区分

Enterprise／Standard は有償、Express／Developer は無償です。なお、Developer エディションは 2016 年 3 月に無償となり、現在は Developer エディション（Enterprise エディション同等機能が提供される開発用途向けのエディション）を利用してすべての機能を無制限に評価することができるようになっています。

有償	Enterprise
	Standard
無償	Express
	Developer

❖ ライセンスモデルの区分

有償エディションにはライセンスモデルの区分があり、エディションごとに対応するライセンスモデルが異なります。なお、SQL Server の機能は All-in one であり利用する機能に対するライセンスの概念はありません。

ライセンスモデル	概要	Enterprise	Standard
コアベース	SQL Server を稼働させるマシン上のコア数をカウントするモデル	●	●
サーバー+CAL	SQL Server に対するサーバーライセンスと、そのサーバーにアクセスするユーザーまたはデバイス数をカウントするモデル	–	●

上記は基本的な考え方のみであるため、コア数や CAL 数のカウント方法、ケースごとのライセンス計上方法（仮想マシンやコンテナー上で稼働するケース、Always On による高可用性や Big Data Cluster など複数ノードで構成するケース、Software Assurance 特典を受けるケースなど）、もしくはライセンスの販売チャネルなど、ライセンスの詳細については次のガイドを確認ください。

◇ Licensing datasheet
https://download.microsoft.com/download/f/0/d/f0d7004e-9e39-4991-853b-2aa09e4ce456/SQL%20Server%202019%20%20Licensing%20Datasheet.pdf

◇ Licensing Guide
https://download.microsoft.com/download/6/6/0/66078040-86d8-4f6e-b0c5-e9919bbcb537/SQL%20Server%202019%20Licensing%20guide.pdf

自習の準備

本自習書で提供するサンプルコードを使用して SQL Server 2019 を自習するため、以降で解説する「SQL Server2019 評価版のインストール」、「クライアントツールのインストール」、「サンプルデータベースの作成」、「サンプルコードのダウンロード」を行ってください。

◆ SQL Server 2019 評価版のインストール

使い慣れたお好きなプラットフォームを選択し、次のサイトの手順に従って SQL Server 2019 をインストールしてください。

Windows	https://docs.microsoft.com/ja-jp/sql/database-engine/install-windows/install-sql-server-from-the-installation-wizard-setup?view=sql-server-ver15
Red Hat	https://docs.microsoft.com/ja-jp/sql/linux/quickstart-install-connect-red-hat?view=sql-server-ver15
SUSE	https://docs.microsoft.com/ja-jp/sql/linux/quickstart-install-connect-suse?view=sql-server-ver15
Ubuntu	https://docs.microsoft.com/ja-jp/sql/linux/quickstart-install-connect-ubuntu?view=sql-server-ver15
Red Hat コンテナ	https://docs.microsoft.com/ja-jp/sql/linux/sql-server-linux-configure-docker?view=sql-server-ver15#rhel
SUSE コンテナ	SUSE コンテナーは執筆時点で公式版が公開されていないため、以下を参考に Dockerfile を作成し、SUSE コンテナーイメージを作成してください。 https://github.com/microsoft/mssql-docker/tree/master/linux/preview/SLES
Ubuntu コンテナ	https://docs.microsoft.com/ja-jp/sql/linux/quickstart-install-connect-docker?view=sql-server-ver15&pivots=cs1-bash

◆ クライアントツールのインストール

本自習書のサンプルコードはいくつかのクライアントツールを利用します。サンプルコードの実行に必要なツールを適宜インストールして利用してください。

- SQL Server Management Studio（SSMS）
  SQL Server の GUI クライアントです。

  ◇ SQL Server Management Studio ダウンロード
  https://docs.microsoft.com/ja-jp/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver15

- Azure Data Studio（ADS）

  SSMS と同じく SQL Server の GUI クライアントです。

  ◇ Azure Data Studio ダウンロード
  https://docs.microsoft.com/ja-jp/sql/azure-data-studio/download-azure-data-studio?view=sql-server-ver15

  SSMS との主な相違は SSMS が Windows のみで動作する一方で、ADS はマルチプラットフォームに対応しており、Windows に加え macOS や Linux でも動作することです。
  本自習書では SSMS を必要とするケースを除き、基本的に ADS を利用します。
  ADS の使い方の詳細や、SSMS との相違の詳細は次の Docs を確認ください。

  ◇ ADS の使い方
  https://docs.microsoft.com/ja-jp/sql/azure-data-studio/quickstart-sql-server?view=sql-server-ver15

  ◇ SSMS と ADS の相違
  https://docs.microsoft.com/ja-jp/sql/azure-data-studio/what-is?view=sql-server-ver15#feature-comparison-with-sql-server-management-studio-ssms

- sqlcmd

  SQL Server の CUI クライアントです。sqlcmd の実行に必要な ODBC Driver も併せてインストールしてください。

  ◇ ODBC Driver ダウンロード
  https://www.microsoft.com/ja-jp/download/details.aspx?id=50420

  ◇ sqlcmd ダウンロード
  https://www.microsoft.com/ja-JP/download/details.aspx?id=52680

- ostress

  SQL Server 用の CUI ストレスツールです。本自習書ではいくつかのサンプルにおいて ostress を利用します。ostress は RML Utilities というユーティリティツールセットに含まれています。

  ◇ RML Utilities ダウンロード
  https://www.microsoft.com/en-us/download/details.aspx?id=4511

◆ サンプルデータベースの作成

本自習書のサンプルコードはマイクロソフト社が公開しているサンプルデータベースである「WideWorldImporters データベース」と「WideWorldImportersDW」を利用します。

WideWorldImporters データベースと WideWorldImportersDW データベースの作成

1. 以下の URL にアクセスし、サンプルデータベースのバックアップファイルである「WideWorldImporters-Full.bak」と「WideWorldImportersDW-Full.bak」をダウンロードします。

   ◇ サンプルデータベースのバックアップファイルのダウンロード
   https://github.com/microsoft/sql-server-samples/releases/tag/wide-world-importers-v1.0

2. ダウンロードしたバックアップファイルをリストアします。
   下記クエリ内の「C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLSERVER¥MSSQL¥DATA」は既定のデータベース格納先、「C:¥work」はバックアップファイルのダウンロード先です。これらは環境に応じて読み替えてください。

Introduction.sql

```
USE master;
GO
-- サンプルデータベース「WideWorldImporters」をリストア
```

```
RESTORE DATABASE WideWorldImporters
  FROM DISK = 'C:¥work¥WideWorldImporters-Full.bak'
  WITH MOVE 'WWI_Primary' TO 'C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLSE
RVER¥MSSQL¥DATA¥WideWorldImporters.mdf',
       MOVE 'WWI_UserData' TO 'C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLS
ERVER¥MSSQL¥DATA¥WideWorldImporters_UserData.ndf',
       MOVE 'WWI_Log' TO 'C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLSERVER
¥MSSQL¥DATA¥WideWorldImporters.ldf',
       MOVE 'WWI_InMemory_Data_1' TO 'C:¥Program Files¥Microsoft SQL Server¥MSSQL1
5.MSSQLSERVER¥MSSQL¥DATA¥WideWorldImporters_InMemory_Data_1';
GO
-- データベースの互換性レベルを SQL Server 2019 の既定値 (150) に設定
ALTER DATABASE WideWorldImporters SET COMPATIBILITY_LEVEL = 150;
GO

-- サンプルデータベース「WideWorldImportersDW」をリストア
RESTORE DATABASE WideWorldImportersDW
  FROM DISK = 'C:¥work¥WideWorldImportersDW-Full.bak'
  WITH MOVE 'WWI_Primary' TO 'C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLSE
RVER¥MSSQL¥DATA¥WideWorldImportersDW.mdf',
       MOVE 'WWI_UserData' TO 'C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLS
ERVER¥MSSQL¥DATA¥WideWorldImportersDW_UserData.ndf',
       MOVE 'WWI_Log' TO 'C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLSERVER
¥MSSQL¥DATA¥WideWorldImportersDW.ldf',
       MOVE 'WWIDW_InMemory_Data_1' TO 'C:¥Program Files¥Microsoft SQL Server¥MSSQL
15.MSSQLSERVER¥MSSQL¥DATA¥WideWorldImportersDW_InMemory_Data_1';
GO
-- データベースの互換性レベルを SQL Server 2019 の既定値 (150) に設定
ALTER DATABASE WideWorldImportersDW SET COMPATIBILITY_LEVEL = 150;
GO

-- リストアしたデータベースの存在を確認
SELECT name, compatibility_level FROM sys.databases
WHERE name LIKE N'WideWorldImporters%';
GO
```

画面下部分の［Results］タブに「WideWorldImporters」と「WideWorldImportersDW」の情報が表示されており、それぞれ「compatibility_level = 150」となっていれば正常にデータベースがリストアできています。

Results　Messages

	name	compatibility_level
1	WideWorldImporters	150
2	WideWorldImportersDW	150

WideWorldImportersDW データベースへの追加のデータ登録

1. 以下の URL にアクセスし、追加のデータ登録のためのスクリプトファイルである「extendwwidw.sql」をダウンロードします。表示されたページから、［Raw］を右クリックして［対象をファイルに保存］によってダウンロードします。

◇ 追加のデータ登録のためのスクリプトファイルのダウンロード

https://github.com/microsoft/bobsql/blob/d37fd129afb3f2810a8036fd6dd51fc40a9c9e2d/sqlbk/ch2_intelligent_performance/iqp/extendwwidw.sql

2. ダウンロードした「extendwwidw.sql」ファイルを ADS で開いて実行します。しばらく時間がかかるため［Messages］タブの末尾に「Total execution time: <所要時間>」というメッセージが表示されるまで待機してください。

3. 最後に、データが正しく登録されたことを確認します。

Introduction.sql

```
USE WideWorldImportersDW;
GO
SELECT 'Fact.OrderHistory' AS TableName, COUNT(*) AS Count
  FROM Fact.OrderHistory;
GO
SELECT 'Fact.OrderHistoryExtended' AS TableName, COUNT(*) AS Count
  FROM Fact.OrderHistoryExtended;
```

```
GO
```

［Results］タブで確認できる結果の通り、「Fact.OrderHistory」テーブルが 3702592 行、「Fact.OrderHistoryExtended」が 29620736 行と表示されていれば正常です。

Results　Messages

	TableName	Count
1	Fact.OrderHistory	3702592

	TableName	Count
1	Fact.OrderHistoryExtended	29620736

◆ サンプルコードのダウンロード

サンプルコードを評価環境へダウンロードしてください。サンプルの実行手順については、各セクションの中で解説します。

◇ サンプルコード
https://github.com/gho9o9/sql19selfstudybook

1. Intelligent Performance

このセクションは、あらゆるデータ操作に不可欠なパフォーマンスに関する機能強化を取り上げます。
「Intelligent Performance」とは、「アプリケーションをできる限り変更せずに、クエリのパフォーマンスを向上させること」をテーマに開発された機能群を示しており、SQL Server 2019 では次の機能テーマが導入・強化されています。

- **Intelligent Query Processing**
  インテリジェンスを兼ね備えたパフォーマンス最適化メカニズム群の提供

- **In-Memory Database**
  Persistent Memory を活用した機能強化と tempdb の In-Memory OLTP サポート

- **Performance Enhancements**
  実現場でのパフォーマンス問題経験がフィードバックされた各種機能強化

1.1. Intelligent Query Processing

SQL Server では常にパフォーマンスの改善が図られています。

例えば、SQL Server 2014 でカーディナリティ推定の精度を向上させる改良が入りました。クエリの性能を決定づけるクエリプランはクエリオプティマイザーによって作成され、主に次の 2 つの要素に基づき決定されています。

1. クエリプランの各オペレーターで処理されるカーディナリティ（≒行の総数）
2. オペレーターで指示されているアルゴリズムのコストモデル

最初のカーディナリティは 2 番目のコストモデルの入力となります。すなわち、カーディナリティの推定の精度向上は推定コストの精度向上に直結するため、より適切なオペレーター配置やアルゴリズム選定が可能になり、結果としてより望ましいパフォーマンスが得られることになります。

さらに SQL Server 2017 で Adaptive Query Processing（AQP）が導入されました。
先の解説の通り、最適なクエリプランの作成の材料となるカーディナリティは統計情報を頼りに推定されているため、統計情報の鮮度や精度が悪いと推定値は不適切になります。また、データの量や分布、密度が変化した場合は当初の推定値との乖離が大きくなるケースもあります。
一方、クエリプランはその作成に対してコストを要すため、データベースは出来る限りクエリプランを再利用しようとします。そのためパフォーマンスにシビアなシステムではクエリプランが最適ではなくなるタイミングをデータベース管理者が見極めて、明示的にクエリプランの再作成をスケジューリングしているケー

スがあります（データベース自身も再作成のタイミングを自動的に見極めていますが、見極めきれないケースもあります）。
AQP はクエリ実行に基づいて得られた「より信頼性のあるカーディナリティ」を、クエリプランを再作成することなく既存のクエリプランに適応させて上記の問題を回避することで、パフォーマンスの改善を図るメカニズム群です。

- **Batch Mode Adaptive Joins**
  「実際の実行で得られたカーディナリティ」を加味したテーブルの結合方法を既存のクエリプランへ適応することでパフォーマンスを改善します。

- **Multi-Statement Table-Valued Function（MSTVF）Interleaved Execution**
  従来固定値でハードコードされていた MSTVF のカーディナリティを、MSTVF を事前に実行して得られたカーディナリティに置き換えることで、より適切なクエリプランを作成することでパフォーマンスを改善します。

- **Batch Mode Memory Grant Feedback**
  「実際の実行で要したメモリサイズ」を加味したメモリ許可サイズを既存のクエリプラン（より厳密にはクエリプラン内のバッチモードオペレーター）へ適応することでパフォーマンスを改善します。

そして SQL Server 2019 では新たに 5 つの最適化メカニズムが追加されました。

- **Batch Mode on Rowstore**
  行ストアのヒープやインデックスに対するバッチモードが導入されます。これにより行ストアを選択した場合の DWH ワークロードへの影響が低減され、ハイブリッドなワークロード要件により対応しやすくなります。

- **Row Mode Memory Grant Feedback**
  SQL Server 2017 で導入されたバッチモードメモリ許可フィードバック（Batch mode memory grant feedback）機能が行モードに対応します。「実際の実行で要したメモリサイズ」を加味したメモリ許可サイズを、既存のクエリプラン（より厳密にはクエリプラン内の行モードオペレーター）へ適応することでパフォーマンスを改善します。

- **Table Variable Deferred Compilation**
  テーブル変数に対するクエリのコンパイルを遅延することで、実際の行数を踏まえたクエリプランを作成しパフォーマンスを改善します。

- **Approximate query processing with APPROX_COUNT_DISTINCT**
  一意な NULL でない値の数を取得する「COUNT(DISTINCT *expression*)」の実行について、「その結果値の正確性」と「結果値を得るまでの応答性」のトレードオフを選択することができるようになりました。

- **Scalar UDF Inlining**
  ユーザー定義のスカラー値関数の実行について、スカラー値関数のインライン化によりパフォーマンスを改善します。

これらは 3 つの既存機能と 5 つの新機能は“インテリジェンスを兼ね備えたパフォーマンス最適化メカニズム群”として Intelligent Query Processing（IQP）と名付けられました。

関数として機能が提供される APPROX_COUNT_DISTINCT を除く IQP の 7 つの機能は、すべて ALTER DATABASE SCOPED CONFIGURATION で有効（既定）・無効を設定可能です。また、現在の設定はシステムカタログ（sys.database_scoped_configurations）で確認することができます。

ここからは、SQL Server 2019 で新たに追加された、 5 つのパフォーマンス最適化メカニズムの詳細を個別に見ていきます。

➢ Batch mode on rowstore

1. 概要

行ストアのヒープやインデックスに対するバッチモードが導入されます。これにより行ストアを選択した場合の DWH ワークロードへの影響が低減され、ハイブリッドなワークロード要件により対応しやすくなります。

2. 利用シーン

SQL Server 2012 において、「分析クエリに最適化したデータ構造である列ストアインデックス」と「列ストアに最適化されたデータアクセス方法となるバッチモード」が導入されました。この列ストアインデックスはデータを列の単位で格納するため、例えば分析クエリ（≒DWH ワークロード）によく見られる売上の集計など、特定列の処理に対して行全体にアクセスする必要がなくなります。また、列内のデータは同じデータ型であるため圧縮効率が高く、I/O やメモリ負荷の大幅な削減と、それに関連して CPU 効率も改善されます。

こうした分析向けの処理として最適化されたバッチモードが SQL Server 2019 で行ストアにも対応します。前述の通り列ストアは分析クエリに対して最適である一方で、そのトレードオフとして行ストアと比較したいくつかの機能制約や更新処理に対する追加のオーバーヘッドを許容する必要がありました。そのため OLTP ワークロードを重視する場合には DWH ワークロードを犠牲に行ストアを選択するケースがありました。こうしたケースに対して行ストアに最適化されたバッチモードが導入されたことで行ストアを選択した場合にも DWH ワークロードへの影響が低減され、ハイブリッドなワークロード要件により対応しやすくなります。

3. やってみよう

それでは Batch mode on rowstore を利用することで得られるパフォーマンス改善効果を、SQL Server 2017 以前の動作と比較することで確認してみましょう。

✱ SQL Server 2017 以前の動作

ここでは行ストアに対する分析クエリが SQL Server 2017 以前の唯一の実行モードである「行モード（非バッチモード）」で動作した場合のパフォーマンスを確認します。

処理の実行

パフォーマンスを確認する処理を実行します。

Batch_mode_on_rowstore.sql

```
USE WideWorldImportersDW;
GO
-- 以前の動作を再現するためデータベースの互換性レベルを SQL Server 2017 (140) に設定
ALTER DATABASE WideWorldImportersDW SET COMPATIBILITY_LEVEL = 140;
GO
```

```
-- バッファキャッシュをクリア
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;
GO
-- このデータベースに関連するクエリプランをクリア
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
-- クエリ実行のパフォーマンス情報の表示を有効化
SET STATISTICS XML ON;
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
-- クエリ実行（レコード数が多い OrderHistoryExtended のスキャン操作）
SELECT [Tax Rate], [Lineage Key], SUM(Quantity) AS SUM_QTY,
       SUM([Unit Price]) AS SUM_BASE_PRICE, COUNT(*) AS COUNT_ORDER
FROM Fact.OrderHistoryExtended
WHERE [Order Date Key] <= DATEADD(dd, -73, '2015-11-13')
GROUP BY [Tax Rate], [Lineage Key]
ORDER BY [Tax Rate], [Lineage Key]
OPTION (MAXDOP 1);
GO
```

SQL Server 2017 以前の動作を再現させるために、スクリプトの冒頭で ALTER DATABASE により互換性レベルを「140」（= SQL Server 2017）へ変更していることに注意してください。

実行モードの確認

1. クエリの実行結果から、実行プランを表示するリンクをクリックします。

2. 新しいタブでクエリの実行プランが表示されるので、一番右端の Table Scan オペレーターに対してマウスのカーソルを合わせます。

3. カーソルを合わせるとプロパティ情報が表示されるので、「実際の実行モード」の値が「Row」（行モード）となっていることが確認できます。

パフォーマンス情報の確認

クエリを実行したシートに戻り、［メッセージ］タブをアクティブにしてパフォーマンスの情報を確認します。

	SQL Server 2017 以前の動作
処理時間（ミリ秒）	26,217
CPU 時間（ミリ秒）	24,656

✱ SQL Server 2019 の動作

ここでは行ストアに対する分析クエリが、SQL Server 2019 でバッチモードへ対応したことにより得られるパフォーマンス改善を確認します。

処理の実行

パフォーマンスを確認する処理を実行します。

Batch_mode_on_rowstore.sql

```
USE WideWorldImportersDW;
GO
-- データベースの互換性レベルを SQL Server 2019 (150) に設定 (既定値)
ALTER DATABASE WideWorldImportersDW SET COMPATIBILITY_LEVEL = 150;
GO
```

```
-- バッファキャッシュをクリア
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;
GO
-- このデータベースに関連するクエリプランをクリア
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
-- クエリ実行のパフォーマンス情報の表示を有効化
SET STATISTICS XML ON;
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
-- クエリ実行（レコード数が多い OrderHistoryExtended のスキャン操作）
SELECT [Tax Rate], [Lineage Key], SUM(Quantity) AS SUM_QTY,
       SUM([Unit Price]) AS SUM_BASE_PRICE, COUNT(*) AS COUNT_ORDER
FROM Fact.OrderHistoryExtended
WHERE [Order Date Key] <= dateadd(dd, -73, '2015-11-13')
GROUP BY [Tax Rate], [Lineage Key]
ORDER BY [Tax Rate], [Lineage Key]
OPTION (MAXDOP 1);
GO
```

先ほどの SQL Server 2017 以前と異なる点は、互換性レベルを SQL Server 2019 の既定値に設定している点のみです。

実行モードの確認

1. クエリの実行結果から、実行プランを表示するリンクをクリックします。

2. 新しいタブでクエリの実行プランが表示されるので、一番右端の Table Scan オペレーターに対してマウスのカーソルを合わせます。

3. カーソルを合わせるとプロパティ情報が表示されるので、「実際の実行モード」の値が「Batch」（バッチモード）となっていることが確認できます。

パフォーマンス情報の確認

クエリを実行したシートに戻り、［メッセージ］タブをアクティブにしてパフォーマンスの情報を確認します。

下記はパフォーマンス情報を SQL Server 2017 以前の動作の結果と合わせて整理したものです。

	SQL Server 2017 以前の動作	SQL Server 2019 の動作
処理時間（ミリ秒）	26,217	14,701
CPU 時間（ミリ秒）	24,656	10,531

行ストアがバッチモードに対応したことにより、サンプル分析クエリの処理時間と CPU 効率がともにおよそ 1/2 に改善されたことが確認できます。

➢ Row mode memory grant feedback

1. 概要

SQL Server 2017 でバッチモードメモリ許可フィードバック（Batch mode memory grant feedback）機能が導入され、「実際の実行で要したメモリサイズ」を加味したメモリ許可サイズを、既存のクエリプラン（より厳密にはクエリプラン内のバッチモードオペレーター）へ適応することで、パフォーマンスを改善しました。SQL Server 2019 ではさらに、行モードに対応した行モードメモリ許可フィードバック（Row mode memory grant feedback）機能が導入されています。

2. 利用シーン

クエリプランには、インデックスの選択や結合順序・方法など、処理のシーケンスが含まれるだけではなく、実行に必要な最小メモリサイズとメモリ許可サイズ（クエリ実行に対して許可されるメモリ割り当てサイズ）が含まれます。このメモリ許可サイズは推定に基づくため、実際のサイズと比較して下回っている場合はページングが発生しパフォーマンスが低下する、逆に上回っている場合は無駄なメモリ確保により同時実行性が低下することになるため、推定サイズの精度はとても重要です。

一方、クエリプランはその作成に対してコストを要すため、データベースは出来る限りクエリプランを再利用しようとします。そのため、パフォーマンスにシビアなシステムではクエリプランが最適ではなくなるタイミングをデータベース管理者が見極めて、明示的にクエリプランの再作成をスケジューリングしているケースがあります（データベース自身も再作成のタイミングを自動的に見極めていますが、見極めきれないケースもあります）。

この課題に対して、SQL Server 2019 は行モードメモリ許可フィードバック（Row mode memory grant feedback）機能を導入し、クエリプランを再作成することなく、既存のクエリプラン（より厳密にはクエリプラン内の行モードオペレーター）に対して「前回実行時の実際のメモリサイズ」を加味したサイズを反映することで、パフォーマンスを改善します。

なお、クエリプランが最適ではなくなるタイミングを見極めてきたデータベース管理者にはひとつ疑問が残るかもしれません。それは「実行時に与えられるパラメーターによって最適なクエリプランが異なるのと同様に、メモリ許可サイズにも同じことが言えるのでは？」ということです。答えは「Yes」です。そのため、SQL Server は不安定なメモリ要件を検知した場合は該当クエリに対するメモリ許可フィードバック機能を無効化するようインテリジェントに振る舞います（明示的に無効化することも可能です）。

3. やってみよう

それでは Row mode memory grant feedback を利用することで得らえるパフォーマンスの向上効果を確認してみましょう。

✱ クエリの初回実行

クエリ実行

テスト用のクエリを実行します。

Row_mode_memory_grant_feedback.sql

```
USE WideWorldImportersDW;
GO
-- 実際と乖離した行数でメタデータを更新
UPDATE STATISTICS Fact.OrderHistory WITH ROWCOUNT = 1;
GO
-- バッファキャッシュをクリア
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;
GO
-- このデータベースに関連するクエリプランをクリア
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
-- テストクエリ実行のパフォーマンス情報の表示を有効化
SET STATISTICS XML ON;
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
-- テストクエリ実行 (Fact.OrderHistory と Dimension.[Stock Item] をハッシュ結合)
SELECT fo.[Order Key], fo.Description, si.[Lead Time Days]
FROM Fact.OrderHistory AS fo
  INNER HASH JOIN Dimension.[Stock Item] AS si
    ON fo.[Stock Item Key] = si.[Stock Item Key]
WHERE fo.[Lineage Key] = 9 AND si.[Lead Time Days] > 19;
GO
```

冒頭の UPDATE STATISTICS により実際と乖離した行数でメタデータを更新している点に注目してください。これによりメモリ許可サイズの推定が実際と大きく乖離するように誘導しています。

推定行数の確認

1. クエリの実行結果から、実行プランを表示するリンクをクリックします。

2. 新しいタブでクエリの実行プランが表示されるので、一番右端の Table Scan オペレーターに対してマウスのカーソルを合わせます。

3. カーソルを合わせるとプロパティ情報が表示されます。「予測行数」は 1 行とみなされていますが、「実際の行数」は 3702592 行であることが確認でき、大きく乖離していることがわかります。

Table Scan

テーブルから行をスキャンします。

物理操作	Table Scan
論理操作	Table Scan
実際の実行モード	Row
推定実行モード	Row
ストレージ	RowStore
読み取った行数	3702592
実際の行数	3702592
実際のバッチ数	0
I/O の推定コスト	69.9461
操作の推定コスト	69.9462 (100%)
CPU の推定コスト	0.0001581
サブツリーの推定コスト	69.9462
実行回数	1
予測実行回数	1
予測行数	1
読み取り対象の予測行数	1
行の推定サイズ	127 B

不適切な推定行数による影響の確認

実行プランの中央にある Hash Match オペレーターに対してマウスのカーソルを合わせます。

ここで Hash Match のオペレーターのアイコンに警告マークが表示されていることが確認できます。カーソルを合わせるとプロパティ情報が表示されるので、「警告」の出力内容を確認します。

「警告」で示されている内容から、今回の処理で tempdb が使用されたことが確認できます。これはクエリオプティマイザーが、OrderHistory テーブルの行数を実際と比較して大きく下回る「1 行」と推定したことに伴ってメモリ許可サイズが小さくなったことで、該当オペレーターの実際の処理においてメモリ不足となり、tempdb に対する物理書き込みが発生したこと（≒パフォーマンスオーバーヘッドが発生していること）を示す警告です。

パフォーマンスの確認

クエリを実行したシートに戻り、［メッセージ］タブをアクティブにしてパフォーマンスの情報を確認します。

上記の図で、Workfile テーブルに対して出力されているメッセージの全文は以下の通りです。

```
テーブル 'Workfile'。スキャン数 8、論理読み取り数 52000、物理読み取り数 6489、ページ
サーバー読み取り数 0、先読みの読み取り数 45511、ページ サーバー先読みの読み取り数 0、L
OB 論理読み取り数 0、LOB 物理読み取り数 0、LOB ページ サーバー読み取り数 0、LOB 先読み
の読み取り数 0、LOB ページ サーバー先読みの読み取り数 0。
```

下記はパフォーマンス情報を整理したものです。ここでは Workfile テーブル（tempdb）に対する I/O 量にも注目しています。

		初回実行
処理時間（ミリ秒）		48,098
CPU 時間（ミリ秒）		2,454
Workfile テーブル	スキャン数	8
	論理読み取り数	52,000
	物理読み取り数	6,489
	先読みの読み取り数	45,511

✱ クエリの再実行

次に初回実行により判明した実際のメモリ許可サイズがフィードバックされることを確認します。

クエリ実行

テスト用のクエリを再実行します。

```
USE WideWorldImportersDW;
GO
-- バッファキャッシュをクリア
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;
GO

-- 初回実行時に作成されたクエリプランを再利用させるため、
-- クエリプランのクリアはコメントアウト
-- ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
-- GO

-- テストクエリ実行のパフォーマンス情報の表示を有効化
SET STATISTICS XML ON;
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
-- テストクエリ実行 (Fact.OrderHistory と Dimension.[Stock Item] をハッシュ結合)
SELECT fo.[Order Key], fo.Description, si.[Lead Time Days]
FROM Fact.OrderHistory AS fo
  INNER HASH JOIN Dimension.[Stock Item] AS si
    ON fo.[Stock Item Key] = si.[Stock Item Key]
WHERE fo.[Lineage Key] = 9 AND si.[Lead Time Days] > 19;
GO
```

```
--実際の行数でメタデータを更新
DBCC UPDATEUSAGE (WideWorldImportersDW,'Fact.OrderHistory') WITH COUNT_ROWS;
GO
```

初回実行時に作成されたクエリプランを再利用させるためにここではクエリプランのクリアは実行していないことに注目してください。それ以外は初回実行と同じクエリです。

推定行数の確認

1. クエリの実行結果から、実行プランを表示するリンクをクリックします。

2. 新しいタブでクエリの実行プランが表示されるので、一番右端の Table Scan オペレーターに対してマウスのカーソルを合わせます。

3. カーソルを合わせるとプロパティ情報が表示されます。初回実行時と同様に、「予測行数」は 1 行とみなされていますが、「実際の行数」は 3702592 行であることが確認でき、大きく乖離していることがわかります。

Table Scan
テーブルから行をスキャンします。

物理操作	Table Scan
論理操作	Table Scan
実際の実行モード	Row
推定実行モード	Row
ストレージ	RowStore
読み取った行数	3702592
実際の行数	3702592
実際のバッチ数	0
I/O の推定コスト	69.9461
操作の推定コスト	69.9462 (100%)
CPU の推定コスト	0.0001581
サブツリーの推定コスト	69.9462
実行回数	1
予測実行回数	1
予測行数	1
読み取り対象の予測行数	1
行の推定サイズ	127 B

不適切な推定行数による影響の確認

実行プランの中央にある Hash Match オペレーターに対してマウスのカーソルを合わせます。

初回実行時とは異なり、Hash Match のオペレーターのアイコンに警告マークが表示されていないことが確認できます。これは Row mode memory grant feedback が機能し、適切なメモリ許可サイズでクエリが実行されたことで、該当オペレーターの処理がメモリ内に収まったためです。その結果、tempdb への物理書き込みに伴うパフォーマンスオーバーヘッドが発生していないことがわかります。

出力一覧
[WideWorldImportersDW].[Fact].[OrderHistory].Order Key,
[WideWorldImportersDW].[Fact].[OrderHistory].Description,
[WideWorldImportersDW].[Dimension].[Stock Item].Lead Time Days
ハッシュ キーの probe
[WideWorldImportersDW].[Dimension].[Stock Item].Stock Item Key

「Row mode memory grant feedback」が機能したことの証跡確認

1. 実行プランが表示されているエリアの任意の場所を右クリックして［実行プランの XML の表示］を選択します。

2. 新しいタブで XML 形式の実行プランが表示されるので、「MemoryGrantInfo」タグを検索します。

3. 上記の図で「MemoryGrantInfo」タグに出力されている内容は以下の通りです。（見やすいように改行して記載しています。）

```
<MemoryGrantInfo
 SerialRequiredMemory="1024" SerialDesiredMemory="625072"
 RequiredMemory="1024" DesiredMemory="625072"
 RequestedMemory="625072" GrantWaitTime="0"
 MaxQueryMemory="1004880" GrantedMemory="625072"
 MaxUsedMemory="494344" LastRequestedMemory="1056"
 IsMemoryGrantFeedbackAdjusted="Yes: Adjusting">
```

```
</MemoryGrantInfo>
```

黄色で示した「IsMemoryGrantFeedbackAdjusted」の値が「Yes: Adjusting」であることにより、「Row mode memory grant feedback」が機能したことを確認できます。
また、拡張イベントの「memory_grant_updated_by_feedback」を取得することでも、証跡を取得することができるようになっています。

パフォーマンス情報の確認

1. クエリを実行したシートに戻り、[メッセージ] タブをアクティブにしてパフォーマンスの情報を確認します。

2. 上記の図で「Workfile」テーブルに対して出力されているメッセージの全文は以下の通りです。

```
テーブル 'Workfile'。スキャン数 0、論理読み取り数 0、物理読み取り数 0、ページ サーバー読み取り数 0、先読みの読み取り数 0、ページ サーバー先読みの読み取り数 0、LOB 論理読み取り数 0、LOB 物理読み取り数 0、LOB ページ サーバー読み取り数 0、LOB 先読みの読み取り数 0、LOB ページ サーバー先読みの読み取り数 0。
```

3. 初回実行と、メモリ許可サイズがフィードバックされた再実行の結果を整理すると、下記の通りとなります。

		初回実行	再実行
処理時間（ミリ秒）		48,098	3,823
CPU 時間（ミリ秒）		2,454	3,172
Workfile テーブル	スキャン数	8	0
	論理読み取り数	52,000	0
	物理読み取り数	6,489	0
	先読みの読み取り数	45,511	0

Row mode memory grant feedback 機能によって、初回実行時の結果を踏まえてより適切なメモリ許可サイズがクエリプランに反映されたことで、オペレーターの処理がメモリ内に収まりました。

これにより、Workfile テーブル（tempdb）への物理書き込みに伴うパフォーマンスオーバーヘッドが回避されたことで、結果として処理時間が大幅に改善されたことを確認できます。

➢ Table variable deferred compilation

1. 概要

テーブル変数に対するクエリのコンパイルを遅延することで、実際の行数を踏まえたクエリプランを作成し、パフォーマンスを改善します。

2. 利用シーン

これまでバッチやストアドプロシージャの内部で使われるケースの多いテーブル変数については、その実際の行数に関係なく、カーディナリティ推定を常に 1 行とみなしてクエリプランを作成していたため、これが原因となってパフォーマンスが劣化するケースがありました。

実際の行数に関係なく作成されたクエリプランがパフォーマンスに対して最適ではないケースの例として、テーブルの結合処理を考えてみます。SQL Server の結合処理の種類には大きくネステッドループ結合、マージ結合、ハッシュ結合がありますが、一般的に一方のテーブルの行数が極めて少ない場合はネステッドループ結合がより効率的であり、逆に両者のテーブルの行数が多い場合はマージ結合やハッシュ結合がより効率的な結合処理となります。

つまり、実際の行数に関係なく常に 1 行とみなされていたテーブル変数については、仮にその結合処理が必要となった場合、ネステッドループ結合が選択される可能性が高いため、実際の行数によってはパフォーマンスに対して望ましくない結合処理となることを意味しています。

この課題に対して SQL Server 2019 で実装された Table variable deferred compilation では、テーブル変数に対するクエリのコンパイルを遅延することで、実際の行数を踏まえたより望ましいクエリプランを作成するように改善されています。この機能は利用者が明示的に利用する機能ではなく SQL Server 内部の振る舞いの強化です。なお、テーブル変数と用途が近しい一時テーブルについては従来から遅延コンパイルが機能しています。

3. やってみよう

それでは Table variable deferred compilation を利用することで得らえるパフォーマンス改善の効果を、SQL Server 2017 以前と比較しながら確認してみましょう。

✱ SQL Server 2017 以前の動作

ここでは 2 つのことを確認します。ひとつは「SQL Server 2017 以前のテーブル変数では、その実際の行数に関係なくカーディナリティ推定の行数が 1 行とみなされること」、もうひとつは「テーブル変数の実際の行数がより大きい行数を持つ場合に、推定行数 1 行とみなして作成されたクエリプランを実行した場合のパフォーマンスを確認すること」です。

処理の実行

パフォーマンスを確認する処理を実行します。

```
Table_variable_deferred_compilation.sql
USE WideWorldImportersDW;
GO
-- 以前の動作を再現するためデータベースの互換性レベルを SQL Server 2017 (140) に設定
ALTER DATABASE WideWorldImportersDW SET COMPATIBILITY_LEVEL = 140;
GO
-- バッファキャッシュをクリア
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;
GO
-- このデータベースに関連するクエリプランをクリア
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
-- テーブル変数を定義
DECLARE @Order TABLE (
  OrderKey BIGINT NOT NULL,
  Quantity INT NOT NULL
);
-- テーブル変数にデータを投入
INSERT @Order SELECT [Order Key], Quantity FROM Fact.OrderHistory WHERE Quantity > 99;
-- テーブル変数の実際の行数を確認
SELECT COUNT(*) FROM @Order
-- クエリ実行のパフォーマンス情報の表示を有効化
SET STATISTICS XML ON;
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
-- クエリ実行 (テーブル変数 @Order と OrderHistoryExtended テーブルの結合)
SELECT oh.[Order Key], oh.[Order Date Key], oh.[Unit Price], o.Quantity
FROM Fact.OrderHistoryExtended AS oh
  INNER JOIN @Order AS o ON o.OrderKey = oh.[Order Key]
WHERE oh.[Unit Price] > 0.10
ORDER BY oh.[Unit Price] DESC
OPTION(MAXDOP 1);
GO
```

スクリプトの冒頭で ALTER DATABASE により互換性レベルを「140」(= SQL Server 2017) へ変更していることに注意してください。

推定行数の確認

1. クエリの実行結果から実行プランを表示するリンクをクリックします。(実行結果の最後に出力されているものが該当します。)

2. 新しいタブでクエリの実行プランが表示されるので一番右端の Table Scan オペレーターに対してマウスのカーソルを合わせます。

3. カーソルを合わせるとプロパティ情報が表示されます。「予測行数」は 1 行とみなされていますが、「実際の行数」は 490928 行であることが確認でき、大きく乖離していることがわかります。

Table Scan

テーブルから行をスキャンします。

物理操作	Table Scan
論理操作	Table Scan
実際の実行モード	Row
推定実行モード	Row
ストレージ	RowStore
読み取った行数	490928
実際の行数	490928
実際のバッチ数	0
操作の推定コスト	0.0032831 (15%)
I/O の推定コスト	0.003125
CPU の推定コスト	0.0001581
サブツリーの推定コスト	0.0032831
実行回数	1
予測実行回数	1
読み取り対象の予測行数	1
予測行数	1
行の推定サイズ	19 B

クエリプランの確認

今回実行したクエリではテーブル変数に対する結合処理を行っており、推定の行数を踏まえて作成されたクエリプランはネステッドループ結合が選択されていることがわかります。

一方で、テーブルの行数が極めて少ない場合にはネステッドループ結合が最適であることから、推定の行数が実際の行数に近しい場合には正しい選択であると考えられますが、そうでない場合にはネステッドループ結合は非効率な選択になりえます。

パフォーマンス情報の確認

クエリを実行したシートに戻り、［メッセージ］タブをアクティブにしてパフォーマンスの情報を確認します。

```
結果   メッセージ
(490928 行処理されました)
テーブル 'Worktable'。スキャン数 0、論理読み取り数 0、物理読み取り数 0、ページ サーバー
テーブル 'OrderHistoryExtended'。スキャン数 490928、論理読み取り数 1965862、物理読み
テーブル '#A06CE850'。スキャン数 1、論理読み取り数 1276、物理読み取り数 0、ページ サー

(1 行処理されました)

 SQL Server 実行時間:
、CPU 時間 = 6344 ミリ秒、経過時間 = 47884 ミリ秒。

完了時刻: 2020-02-26T17:34:21.9021115+09:00
```

上の図で OrderHistoryExtended テーブルに対して出力されているメッセージの全文は以下の通りです。

```
テーブル 'OrderHistoryExtended'。スキャン数 490928、論理読み取り数 1965862、物理読み取り数 9373、ページ サーバー読み取り数 0、先読みの読み取り数 0、ページ サーバー先読みの読み取り数 0、LOB 論理読み取り数 0、LOB 物理読み取り数 0、LOB ページ サーバー読み取り数 0、LOB 先読みの読み取り数 0、LOB ページ サーバー先読みの読み取り数 0。
```

下記はパフォーマンス情報を整理したものです。ここでは今回実行したクエリの処理時間と、テーブル変数との結合処理の対象となる OrderHistoryExtended テーブルに対する I/O 量に注目しています。この結果が SQL Server 2019 によってどのように改善されるかを次のパートで確認します。

		SQL Server 2017 以前の動作
処理時間（ミリ秒）		47,884
OrderHistoryExtended テーブル	スキャン数	490,928
	論理読み取り数	1,965,862
	物理読み取り数	9,373
	先読みの読み取り数	0

✱ SQL Server 2019 の動作

SQL Server 2017 以前との動作比較で SQL Server 2019 の Table variable deferred compilation の効果を確認します。

処理の実行

パフォーマンスを確認する処理を実行します。

Table_variable_deferred_compilation.sql

```
USE WideWorldImportersDW;
GO
-- データベースの互換性レベルを SQL Server 2019 (150) に設定 (既定値)
ALTER DATABASE WideWorldImportersDW SET COMPATIBILITY_LEVEL = 150;
GO
-- バッファキャッシュをクリア
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;
GO
-- このデータベースに関連するクエリプランをクリア
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
-- テーブル変数を定義
DECLARE @Order TABLE (
  OrderKey BIGINT NOT NULL,
  Quantity INT NOT NULL
);
-- テーブル変数にデータを投入
INSERT @Order SELECT [Order Key], Quantity FROM Fact.OrderHistory WHERE Quantity > 99;
-- テーブル変数の実際の行数を確認
SELECT COUNT(*) FROM @Order
-- クエリ実行のパフォーマンス情報の表示を有効化
SET STATISTICS XML ON;
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
-- クエリ実行 (テーブル変数 @Order と OrderHistoryExtended テーブルの結合)
SELECT oh.[Order Key], oh.[Order Date Key], oh.[Unit Price], o.Quantity
FROM Fact.OrderHistoryExtended AS oh
  INNER JOIN @Order AS o ON o.OrderKey = oh.[Order Key]
WHERE oh.[Unit Price] > 0.10
ORDER BY oh.[Unit Price] DESC
OPTION(MAXDOP 1);
GO
```

先の SQL Server 2017 以前の動作確認と異なる点は、互換性レベルを SQL Server 2019 の既定値に設定している点です。

推定行数の確認

1. クエリの実行結果から実行プランを表示するリンクをクリックします。(実行結果の最後に出力されているものが該当します。)

2. 新しいタブでクエリの実行プランが表示されるので一番右端の Table Scan オペレーターに対してマウスのカーソルを合わせます。

3. カーソルを合わせるとプロパティ情報が表示されます。SQL Server 2017 以前とは異なり「実際の行数」と「予測行数」の値がどちらも 490928 行で一致していることがわかります。

Table Scan
テーブルから行をスキャンします。

物理操作	Table Scan
論理操作	Table Scan
実際の実行モード	Batch
推定実行モード	Batch
ストレージ	RowStore
読み取った行数	490928
実際の行数	490928
実際のバッチ数	546
I/O の推定コスト	0.947569
操作の推定コスト	1.48775 (0%)
CPU の推定コスト	0.540178
サブツリーの推定コスト	1.48775
実行回数	1
予測実行回数	1
予測行数	490928
読み取り対象の予測行数	490928
行の推定サイズ	19 B

クエリプランの確認

テーブル変数の推定の行数が 490928 行と解釈されたことで、結合する両者のテーブルの行数が多い場合に最適な結合処理（今回はハッシュ結合）が選択され、SQL Server 2017 以前の動作とは異なるクエリプランが作成されていることがわかります。

パフォーマンス情報の確認

クエリを実行したシートに戻り、［メッセージ］タブをアクティブにしてパフォーマンスの情報を確認します。

上の図で OrderHistoryExtended テーブルに対して出力されているメッセージの全文は以下の通りです。

> テーブル 'OrderHistoryExtended'。スキャン数 1、論理読み取り数 755447、物理読み取り数 0、ページ サーバー読み取り数 0、先読みの読み取り数 754803、ページ サーバー先読みの読み取り数 0、LOB 論理読み取り数 0、LOB 物理読み取り数 0、LOB ページ サーバー読み取り数 0、LOB 先読みの読み取り数 0、LOB ページ サーバー先読みの読み取り数 0。

下記はパフォーマンス情報を SQL Server 2017 以前の動作の結果と合わせて整理したものです。

		SQL Server 2017 以前の動作	SQL Server 2019 の動作
処理時間（ミリ秒）		47,884	16,354
OrderHistoryExtended	スキャン数	490,928	1

テーブル	論理読み取り数	1,965,862	755,447
	物理読み取り数	9,373	0
	先読みの読み取り数	0	754,803

テーブル変数に対するクエリのコンパイルを遅延することで、実際の行数を踏まえたクエリプランが作成されたことを確認できました。そしてパフォーマンス情報を整理した結果から、今回のサンプルではより効率的なクエリプランによって、OrderHistoryExtended テーブルへのアクセスが最適化されたことで読み取り数が減少し、処理時間として約 3 倍のパフォーマンス改善が得られました。

➢ Approximate query processing with APPROX_COUNT_DISTINCT

1. 概要

一意な NULL でない値の数を取得する「COUNT(DISTINCT *expression*)」の実行について、「その結果値の正確性」と「結果値を得るまでの応答性」のトレードオフを選択することができるようになりました。

2. 利用シーン

例えばある Web サイトの Page View に対してユニークユーザー数を集計したい場合、返される数の正確性よりも応答性を重要したいケースに対して、Approximate query processing with APPROX_COUNT_DISTINCT が SQL Server 2019 で導入されています。APPROX_COUNT_DISTINCT は HyperLogLog アルゴリズムにより実装されており、正確な COUNT DISTINCT と比較してメモリフットプリントが軽く実行コストが低いことも特徴です。

◇ HyperLogLog
https://en.wikipedia.org/wiki/HyperLogLog

◇ Count-distinct problem
https://en.wikipedia.org/wiki/Count-distinct_problem

3. やってみよう

それでは「APPROX_COUNT_DISTINCT(*expression*)」の結果を、「COUNT(DISTINCT *expression*)」と比較して確認してみましょう。

✱ COUNT(DISTINCT *expression*) の動作

クエリの実行

パフォーマンスを確認する処理を実行します。

Approximate_query_processing_with_APPROX_COUNT_DISTINCT.sql

```
USE WideWorldImportersDW;
GO
-- バッファキャッシュをクリア
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;
GO
-- このデータベースに関連するクエリプランをクリア
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
-- テストクエリ実行のパフォーマンス情報の表示を有効化
SET STATISTICS XML ON;
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
```

```
-- テストクエリ実行 (COUNT(DISTINCT <col>))
SELECT COUNT(DISTINCT [WWI Order ID])
FROM Fact.OrderHistoryExtended
OPTION (MAXDOP 1, USE HINT(N'DISALLOW_BATCH_MODE'));
GO
```

クエリ結果の確認

「COUNT(DISTINCT *expression*)」に対して「29,620,736」の結果が得られています。

実行プランを表示するリンクをクリックします。

クエリプランには Hash Match（Aggregate）オペレーターが存在し、このオペレーターにより一意の NULL ではない値ごとのハッシュテーブルが作成されています。

パフォーマンス情報の確認

クエリを実行したシートに戻り、［メッセージ］タブをアクティブにしてパフォーマンスの情報を確認します。

	COUNT(DISTINCT *expression*)
COUNT の結果	29,620,736
処理時間（ミリ秒）	35,555
CPU 時間（ミリ秒）	34,281

✱ APPROX_COUNT_DISTINCT(*expression*) の動作

クエリの実行

パフォーマンスを確認する処理を実行します。

Approximate_query_processing_with_APPROX_COUNT_DISTINCT.sql

```
USE WideWorldImportersDW;
GO
-- バッファキャッシュをクリア
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;
GO
-- このデータベースに関連するクエリプランをクリア
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
-- テストクエリ実行のパフォーマンス情報の表示を有効化
SET STATISTICS XML ON;
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
-- テストクエリ実行（APPROX_COUNT_DISTINCT）
SELECT APPROX_COUNT_DISTINCT([WWI Order ID])
FROM Fact.OrderHistoryExtended
OPTION (MAXDOP 1, USE HINT(N'DISALLOW_BATCH_MODE'));
GO
```

COUNT に利用する関数が異なることを除いて、「COUNT(DISTINCT *expression*)」の動作確認のときと全く同じクエリです。

クエリ結果の確認

「APPROX_COUNT_DISTINCT(*expression*)」に対して「30,382,637」の結果が得られています。

実行プランを表示するリンクをクリックします。

「COUNT(DISTINCT *expression*)」で確認したクエリプランと比較して、Hash Match（Aggregate）オペレーターがなくなっていることに注目してください。

パフォーマンス情報の確認

クエリを実行したシートに戻り、［メッセージ］タブをアクティブにしてパフォーマンスの情報を確認します。

下記はパフォーマンス情報を「COUNT(DISTINCT expression)」の結果と合わせて整理したものです。

	COUNT (DISTINCT expression)	APPROX_COUNT_DISTINCT (expression)
COUNT の結果	29,620,736	30,382,637
処理時間（ミリ秒）	30,768	17,165
CPU 時間（ミリ秒）	28,766	15,719

「APPROX_COUNT_DISTINCT(expression)」による応答性を重視したことで、結果の正確性とのトレードオフにより応答性を得ることができ、このサンプルクエリについての処理時間はおよそ 1/2 に改善されました。

➢ Scalar UDF Inlining

1. 概要

ユーザー定義のスカラー値関数の実行について、スカラー値関数のインライン化によりパフォーマンスを改善します。

2. 利用シーン

スカラー値関数（単一のデータを戻り値とする関数）はコードの再利用や部品化により実装の品質を強化するための技法として重要な機能です。しかし、次に示す SQL Server 内部の振る舞いが関数実行のパフォーマンスを劣化させることがありました。

- **該当するタプルごとに反復的にユーザー定義関数が呼び出される**
  関数呼び出しによる反復的なコンテキスト切り替えの追加コストが発生する
- **ユーザー定義関数はコスト計算やクロスステートメントの最適化がサポートされない**
  不適切なクエリプランになり得る
- **ユーザー定義関数を呼び出すクエリでクエリ内の並列処理がサポートされない**
  スケーラビリティの制約となる

この課題に対応する機能として Scalar UDF Inlining（スカラー値関数のインライン化）が SQL Server 2019 で導入されています。Scalar UDF Inlining は、スカラー値関数をスカラー式またはスカラーサブクエリとして自動的に変換することで上記 3 点の問題を回避しパフォーマンスを改善します。

3. やってみよう

Scalar UDF Inlining によるパフォーマンス改善を確認してみましょう。

✱ 事前準備

スカラー値関数の定義

スカラー値関数を定義します。

Scalar_UDF_Inlining.sql

```
USE WideWorldImportersDW;
GO
-- サンプルクエリで使用する Scalar UDF を定義
CREATE OR ALTER FUNCTION dbo.ufn_customer_category(@CustomerKey int)
RETURNS char(10) AS
BEGIN
  DECLARE @total_amount decimal(18,2);
  DECLARE @category char(10);

  SELECT @total_amount = SUM([Total Including Tax])
    FROM Fact.OrderHistory
    WHERE [Customer Key] = @CustomerKey
    OPTION(MAXDOP 1);
```

```
  IF @total_amount < 500000
    SET @category = 'REGULAR';
  ELSE IF @total_amount < 1000000
    SET @category = 'GOLD';
  ELSE
    SET @category = 'PLATINUM';

  RETURN @category;
END
GO
```

✱ SQL Server 2017 以前の動作

まずは SQL Server 2017 以前の動作を見てみます。

処理の実行

パフォーマンスを確認する処理を実行します。

Scalar_UDF_Inlining.sql

```
USE WideWorldImportersDW;
GO
-- 以前の動作を再現するためデータベースの互換性レベルを SQL Server 2017 (140) に設定
ALTER DATABASE WideWorldImportersDW SET COMPATIBILITY_LEVEL = 140;
GO
-- バッファキャッシュをクリア
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;
GO
-- このデータベースに関連するクエリプランをクリア
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
-- クエリ実行のパフォーマンス情報の表示を有効化
SET STATISTICS XML ON;
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
-- クエリ実行 (ufn_customer_category()を使用)
SELECT TOP 100 [Customer Key], Customer,
       dbo.ufn_customer_category([Customer Key]) AS [Discount Price]
FROM Dimension.Customer
ORDER BY [Customer Key];
GO
```

スクリプトの冒頭で ALTER DATABASE により、互換性レベルを「140」(= SQL Server 2017) へ変更していることに注目してください。

関数の反復呼び出しの確認

1. クエリの実行結果から、実行プランを表示するリンクをクリックします。

2. 新しいタブでクエリの実行プランが表示されるので、左から二番目のコンピューティングスカラーオペレーターに対してマウスのカーソルを合わせます。

カーソルを合わせるとプロパティ情報が表示されます。「実際の行数」の値から、このオペレーターに 100 行の入力があったことがわかります。

3. 実行プランが表示されているエリアの任意の場所を右クリックして［実行プランの XML の表示］を選択します。

4. 新しいタブで XML 形式の実行プランが表示されるので、「ComputeScalar」タグを検索します。

5. 「ComputeScalar」タグ配下の「ScalarOperator」に出力されている内容は以下の通りです。

```
<ScalarOperator ScalarString="[WideWorldImportersDW].[dbo].[ufn_customer_category]
([WideWorldImportersDW].[Dimension].[Customer].[Customer Key])">
```

「ScalarString」の値が、事前準備で定義したユーザー定義関数であることが確認できます。このことから、コンピュートスカラーオペレーターへの入力となった 100 行分、行ごとにユーザー定義関数が呼び出されたことがわかります。

なお、次のように拡張イベントを利用してクエリ実行をトレースすると、より厳密に行ごとにユーザー定義関数がコールされていることを確認することができます。

Scalar_UDF_Inlinin...252855412970000.xel

フィルター選択された 100 個のイベントのうちの 100 個を表示しています (合計 556 個のイベント)

name	database_name	session_id	statement
sp_statement_completed	WideWorldImportersDW	89	SELECT @total_amount = SUM([Total Including Tax]) FROM Fact.OrderHistory WHERE [Customer Key] = @CustomerKey
sp_statement_completed	WideWorldImportersDW	89	SELECT @total_amount = SUM([Total Including Tax]) FROM Fact.OrderHistory WHERE [Customer Key] = @CustomerKey
sp_statement_completed	WideWorldImportersDW	89	SELECT @total_amount = SUM([Total Including Tax]) FROM Fact.OrderHistory WHERE [Customer Key] = @CustomerKey
sp_statement_completed	WideWorldImportersDW	89	SELECT @total_amount = SUM([Total Including Tax]) FROM Fact.OrderHistory WHERE [Customer Key] = @CustomerKey
sp_statement_completed	WideWorldImportersDW	89	SELECT @total_amount = SUM([Total Including Tax]) FROM Fact.OrderHistory WHERE [Customer Key] = @CustomerKey
sp_statement_completed	WideWorldImportersDW	89	SELECT @total_amount = SUM([Total Including Tax]) FROM Fact.OrderHistory WHERE [Customer Key] = @CustomerKey

パフォーマンス情報の確認

クエリを実行したシートに戻り、[メッセージ] タブをアクティブにしてパフォーマンスの情報を確認します。

結果 メッセージ

```
(100 行処理されました)
テーブル 'Customer'。スキャン数 1、論理読み取り数 6、物理読み取り数 1、ページ サーバー

(1 行処理されました)

 SQL Server 実行時間:
、CPU 時間 = 48828 ミリ秒、経過時間 = 49769 ミリ秒。

完了時刻: 2020-02-26T23:50:32.5637691+09:00
```

	SQL Server 2017 以前の動作
処理時間（ミリ秒）	49,769
CPU 時間（ミリ秒）	48,828

✱ SQL Server 2019 の動作

ここでは SQL Server 2019 の Scalar UDF Inlining の効果を確認します。

処理の実行

パフォーマンスを確認する処理を実行します。

```
USE WideWorldImportersDW;
GO
-- データベースの互換性レベルを SQL Server 2019 (150) に設定 (既定値)
ALTER DATABASE WideWorldImportersDW SET COMPATIBILITY_LEVEL = 150;
GO
-- バッファキャッシュをクリア
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;
GO
-- このデータベースに関連するクエリプランをクリア
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
-- クエリ実行のパフォーマンス情報の表示を有効化
SET STATISTICS XML ON;
```

```
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
-- クエリ実行 (ufn_customer_category() を使用)
SELECT TOP 100 [Customer Key], Customer,
       dbo.ufn_customer_category([Customer Key]) AS [Discount Price]
FROM Dimension.Customer
ORDER BY [Customer Key];
GO
```

先の SQL Server 2017 以前の動作確認と異なる点は、互換性レベルを SQL Server 2019 の既定値に設定している点のみです。

関数のインライン化

1. クエリの実行結果から、実行プランを表示するリンクをクリックします。

2. 新しいタブでクエリの実行プランが表示されます。
   以下の図のように SQL Server 2017 以前のクエリプランとは大きく異なることが確認できます。これはユーザー定義関数がインライン展開されているためです。例えばユーザー定義関数内で参照している OrderHistory テーブルに対する Table Scan オペレーターが存在していることが確認できます。

パフォーマンス情報の確認

クエリを実行したシートに戻り、［メッセージ］タブをアクティブにしてパフォーマンスの情報を確認します。

下記はパフォーマンス情報を SQL Server 2017 以前の動作の結果と合わせて整理したものです。

	SQL Server 2017 以前の動作	SQL Server 2019 の動作
処理時間（ミリ秒）	49,769	13,077
CPU 時間（ミリ秒）	48,828	12,547

Scalar UDF Inlining（スカラー値関数のインライン化）により、スカラー値関数におけるパフォーマンスの懸念が回避され、このサンプルクエリについての処理時間は 1/2 以下に改善されました。

1.2. In-Memory Database

SQL Server 2014 で In-Memory OLTP 機能という OLTP パフォーマンスを向上させるスケールアップ指向の In-Memory エンジンが導入されました。これはインメモリに最適化された新たなクエリエンジン、新たなストレージエンジンとデータ構造によって OLTP のパフォーマンスを向上させる機能です。新たなクエリエンジンは従来のようなクエリプランを逐一処理するのではなく、あらかじめクエリをマシンコード化することでクエリ処理による CPU インストラクションの軽量化を実現しています。また、新たなストレージエンジンは、ロックやラッチの待ちに伴うコンテキストスイッチによるオーバーヘッドや、スピンロックによる CPU 空転を排除し、CPU 効率を上げることによりマルチコアに対するスケーラビリティ向上を実現、さらにメインメモリに最適化したデータ構造やストリームベースの永続化による I/O 効率向上を実現しています。また SQL Server 2016 での改善として、既存アプリケーションからの移植性が強化されたことで活用シーンが広がっています。

その In-Memory OLTP 機能に加えて、SQL Server 2019 では新たに 2 つの新機能が追加され、これらは In-Memory Database 機能スイートとして括られました。

- **Memory-Optimized TempDB Metadata**
  tempdb 内のシステムテーブルに In-Memory OLTP 機能を適用しメモリ最適化テーブル化することで、tempdb 内のシステムテーブルに負荷をかけるワークロードの輻輳により発生するラッチ競合の問題を回避し、パフォーマンスを改善します。

- **Enhanced PMEM support（Hybrid Buffer Pool & Enlightenment）**
  Persistent Memory（PMEM）を活用する新たな機能として Hybrid Buffer Pool と Enlightenment（Linux 版のみ対応）が導入されます。
  Hybrid Buffer Pool 機能は PMEM 上に配置されたデータファイルに対する Memory-mapped I/O による直接アクセス（DAX）により、従来必要としていたバッファプールへのデータページのコピー処理のオーバーヘッドを排除します。この動作はクリーンページの読み取りを対象にしており、変更されるデータページ（ダーティページ）は従来どおりバッファプールにコピーされる動作になります。
  Enlightenment 機能はクリーンページの読み取りを対象とした Hybrid Buffer Pool 機能とは異なり、PMEM 上のデータファイルおよびログファイルに対する Read/Write にも DAX を利用しパフォーマンスを改善します。

➢ Memory-Optimized TempDB Metadata

1. 概要

tempdb 内のシステムテーブル（より厳密には一時テーブルのメタデータ管理に関連するシステムテーブル）に In-Memory OLTP 機能を適用し、メモリ最適化テーブル化することで、tempdb 内のシステムテーブルに負荷をかけるワークロードの輻輳により発生するラッチ競合の問題を回避し、パフォーマンスを改善します。

2. 利用シーン

tempdb 内で発生するラッチ競合によるパフォーマンス問題に直面し、戦ってきた開発者やデータベース管理者は多いのではないでしょうか。これは特に一時テーブルを多用するアプリケーションをホストする SQL Server において顕著に見られる問題でした。

そうした戦いを通じて得られた知見として、tempdb の設計には従来 3 つの掟がありました。

- tempdb のデータファイルは複数均等分割せよ
- データファイル拡張は全ファイル同時に拡張せよ
- 混合エクステントは無効化せよ

さらに、SQL Server 2016 ではデータベースエンジン側でいくつかの改善が入りました。

- tempdb データファイルの推奨分割数を、SQL Server をインストールするサーバーのスペックに応じて、インストーラーが自動決定
- tempdb データファイルの同時拡張の既定動作化
- tempdb に対する混合エクステント無効化の既定動作化
- 一時テーブルのキャッシュ管理に関する動作改善
  - ✓ 一時テーブルの削除処理の非同期化（削除リストを更新し、ヘルパースレッドによりリストが参照され削除される）
  - ✓ メタデータにアクセスするヘルパースレッドの削減と削除処理の一括化
  - ✓ メタデータへのアクセス時のラッチ戦略の最適化
- PFS ページを使用してスペースを割り当てるためのラウンドロビンアルゴリズムの改善

こうした改善でも解決しないケースに対しては、以下のようなアプリケーション側の改修に踏み込む必要がありました。

- ストアドプロシージャ内で作成した一時テーブルは明示的に削除しない（システム内部のクリーンアップ処理に任せる）
- 一時テーブルに対する ALTER は避ける
- 一時テーブルに対する TRUNCATE は避ける

- 一時テーブルへのインデックス作成ステートメントは、SQL Server 2014 でサポートされたインラインインデックス構文に書き換える
- 一時表へのレコード数が少ない場合（<100）はテーブル変数への置き換えを検討する（なお、テーブル変数は統計情報が作成されないため、テーブル変数へのクエリがカーディナリティの変更の影響を受けやすい場合は、トレースフラグ 2453 によりカーディナリティベースの再コンパイルを有効にすると良い）

しかし、SQL Server 2019 ではさらに大きな改善が入りました。それは In-Memory OLTP を tempdb 内のシステムテーブルに適用し、メモリ最適化テーブル化することです。先の解説の通り In-Memory OLTP は本質的にラッチを発生させないメカニズムであり、ラッチ競合問題の回避に最適です。また、メモリ最適化テーブルは永続化不要モードで利用すると、さらに性能に対するアドバンテージが得られます。一方、tempdb は再起動の都度作り直される仕様であり、永続化を必須としない特性を持ちます。このことから、In-Memory OLTP は tempdb のラッチ競合問題に対して、まさにベストフィットなソリューションと言えます。

3. やってみよう

それでは、Memory-Optimized TempDB Metadata による性能改善を確認してみましょう。

✱ 事前準備

再現用 SQL Server 環境の準備

ラッチ競合はスレッド間の競合であるため、その再現には複数の論理 CPU が割り当てられた SQL Server を用意してください。なお、本ハンズオンでは 4 つの論理コアを割り当てた SQL Server を利用しています。

ストアドプロシージャの作成

tempdb のラッチ競合を誘発させるワークロードとなる以下のストアドプロシージャを作成します。

Memory-Optimized_TempDB_Metadata.sql

```
USE WideWorldImportersDW;
GO
CREATE PROCEDURE dbo.TempTableProc AS
BEGIN
  CREATE TABLE #Temp (c1 int);
END;
GO
```

ostress ツールの準備

作成したストアドプロシージャの多重実行を再現するために、本ハンズオンでは ostress ツールを利用しています。
ostress ツールのセットアップについては、SQL Server 2019 自習書「Introduction」-「クライアントツールのインストール」を参照してください。

✱ Memory-Optimized TempDB Metadata 無効時の動作

まずは機能が無効な状態での性能を見ていきます。

Memory-Optimized TempDB Metadata の無効化

1. Memory-Optimized TempDB Metadata を無効化します。

Memory-Optimized_TempDB_Metadata.sql

```
USE master;
GO
ALTER SERVER CONFIGURATION SET MEMORY_OPTIMIZED TEMPDB_METADATA = OFF;
GO
```

2. SQL Server サービスを再起動します（この機能の ON/OFF の変更には本再起動が必須です）。

3. 再起動の完了後、「IsTempDBMetadataMemoryOptimized」が 0 （＝無効）であること確認します。

Memory-Optimized_TempDB_Metadata.sql

```
USE master;
GO
SELECT SERVERPROPERTY('IsTempDBMetadataMemoryOptimized')
         AS IsTempDBMetadataMemoryOptimized;
GO
```

Results　Messages

	IsTempDBMetadataMemoryOptimized
1	0

ラッチ待機の発生状況を確認

1. コマンドプロンプトから、以下のコマンドで ostress を実行します。

Memory-Optimized_TempDB_Metadata.bat.txt

```
"C:¥Program Files¥Microsoft Corporation¥RMLUtils¥ostress.exe" -Slocalhost -Usa -PPa
sswOrd -dWideWorldImportersDW -Q"EXEC dbo.TempTableProc" -mstress -quiet -n16 -r100
00 | FINDSTR "QEXEC Starting Creating elapsed"
```

スレッド数（-n16）は 16、スレッドごとに TempTableProc プロシージャの呼び出しを 10,000 回繰り返し実行（-r10000）するコマンドです。なお、サーバー名（-Slocalhost)、ユーザー名（-Usa)、パスワード（-PPassw0rd）は環境に合わせて適宜変更してください。

2. ostress ツールの実行中に、別の新しいセッションから現在のラッチ待機の一覧を確認します。

Memory-Optimized_TempDB_Metadata.sql

```
USE master;
GO
SELECT
  er.session_id, er.wait_type, er.wait_resource,
  OBJECT_NAME(page_info.object_id, page_info.database_id) as object_name,
  er.blocking_session_id, er.command,
  SUBSTRING(st.text, (er.statement_start_offset/2)+1,
    ((CASE er.statement_end_offset
        WHEN -1 THEN DATALENGTH(st.text)
        ELSE er.statement_end_offset
      END - er.statement_start_offset)/2) + 1) AS statement_text,
  page_info.database_id, page_info.file_id, page_info.page_id, page_info.object_id,
  page_info.index_id, page_info.page_type_desc
FROM sys.dm_exec_requests AS er
  CROSS APPLY sys.dm_exec_sql_text(er.sql_handle) AS st
  CROSS APPLY sys.fn_PageResCracker(er.page_resource) AS r
  CROSS APPLY
    sys.dm_db_page_info(r.db_id, r.file_id, r.page_id, 'DETAILED') AS page_info
WHERE er.wait_type like '%page%';
GO
```

Results　Messages　Query Plan　Top Operations

	session_id	wait_type	wait_resource	object_name	blocking_session_id	command	statement_text	database_id	file
1	66	PAGELATCH_EX	2:1:116	sysschobjs	72	CREATE TABLE	CREATE TABLE #Temp (c1 int)	2	1
2	67	PAGELATCH_EX	2:1:116	sysschobjs	66	EXECUTE	EXEC dbo. TempTableProc	2	1
3	68	PAGELATCH_EX	2:1:116	sysschobjs	66	CREATE TABLE	CREATE TABLE #Temp (c1 int)	2	1
4	69	PAGELATCH_EX	2:1:116	sysschobjs	71	EXECUTE	EXEC dbo. TempTableProc	2	1
5	70	PAGELATCH_EX	2:1:116	sysschobjs	66	CREATE TABLE	CREATE TABLE #Temp (c1 int)	2	1
6	71	PAGELATCH_EX	2:1:116	sysschobjs	80	EXECUTE	EXEC dbo. TempTableProc	2	1
7	72	PAGELATCH_EX	2:1:116	sysschobjs	0	CREATE TABLE	CREATE TABLE #Temp (c1 int)	2	1
8	75	PAGELATCH_EX	2:1:116	sysschobjs	66	CREATE TABLE	CREATE TABLE #Temp (c1 int)	2	1
9	76	PAGELATCH_EX	2:1:116	sysschobjs	70	CREATE TABLE	CREATE TABLE #Temp (c1 int)	2	1
10	77	PAGELATCH_EX	2:1:116	sysschobjs	0	EXECUTE	EXEC dbo. TempTableProc	2	1
11	78	PAGELATCH_EX	2:1:116	sysschobjs	72	CREATE TABLE	CREATE TABLE #Temp (c1 int)	2	1
12	79	PAGELATCH_EX	2:1:116	sysschobjs	0	CREATE TABLE	CREATE TABLE #Temp (c1 int)	2	1
13	81	PAGELATCH_SH	2:1:116	sysschobjs	79	CREATE TABLE	CREATE TABLE #Temp (c1 int)	2	1

上図の結果から、多数のラッチ待機が発生していることが確認できます。

もし、お使いの環境でラッチ待機が出力されない場合は、SQL Server に割り当てる論理 CPU 数や負荷スレッド数（-n16）を増やすなど調整してください。また、ostress が完了するまでの処理時間が長くなりすぎる場合は「-r10000」のパラメーターを減らして調整してください。

実行時間の確認

ostress コマンドの実行画面に戻り、ワークロードが完了するまで待機して実行時間を確認します。

今回の環境では、実行時間は「00:01:08.160」でした。

✱ Memory-Optimized TempDB Metadata 有効時の動作

次に機能が有効な状態での性能を見ていきます。

Memory-Optimized TempDB Metadata の有効化

1. Memory-Optimized TempDB Metadata を有効化します。

Memory-Optimized_TempDB_Metadata.sql

```
USE master;
GO
ALTER SERVER CONFIGURATION SET MEMORY_OPTIMIZED TEMPDB_METADATA = ON;
GO
```

2. SQL Server サービスを再起動します（この機能の ON/OFF の変更には本再起動が必須です）。

3. 再起動の完了後、「IsTempDBMetadataMemoryOptimized」が 1（＝有効）であること確認します。

Memory-Optimized_TempDB_Metadata.sql

```
USE master;
GO
SELECT SERVERPROPERTY('IsTempDBMetadataMemoryOptimized')
        AS IsTempDBMetadataMemoryOptimized;
GO
```

Results　Messages

	IsTempDBMetadataMemoryOptimized
1	1

ラッチ待機の発生状況を確認

1. コマンドプロンプトから、以下のコマンドで ostress を実行します。

Memory-Optimized_TempDB_Metadata.bat.txt

```
"C:¥Program Files¥Microsoft Corporation¥RMLUtils¥ostress.exe" -Slocalhost -Usa -PPa
ssw0rd -dWideWorldImportersDW -Q"EXEC dbo.TempTableProc" -mstress -quiet -n16 -r100
00 | FINDSTR "QEXEC Starting Creating elapsed"
```

先ほどと同様に、サーバー名（-Slocalhost)、ユーザー名（-Usa)、パスワード（-PPassw0rd）は環境に合わせて変更してください。

2. ostress ツールの実行中に、別の新しいセッションから現在のラッチ待機の一覧を確認します。

Memory-Optimized_TempDB_Metadata.sql

```
USE master;
GO
SELECT
  er.session_id, er.wait_type, er.wait_resource,
  OBJECT_NAME(page_info.object_id, page_info.database_id) as object_name,
  er.blocking_session_id, er.command,
  SUBSTRING(st.text, (er.statement_start_offset/2)+1,
    ((CASE er.statement_end_offset
        WHEN -1 THEN DATALENGTH(st.text)
        ELSE er.statement_end_offset
      END - er.statement_start_offset)/2) + 1) AS statement_text,
  page_info.database_id, page_info.file_id, page_info.page_id, page_info.object_id,
  page_info.index_id, page_info.page_type_desc
FROM sys.dm_exec_requests AS er
  CROSS APPLY sys.dm_exec_sql_text(er.sql_handle) AS st
  CROSS APPLY sys.fn_PageResCracker(er.page_resource) AS r
  CROSS APPLY
    sys.dm_db_page_info(r.db_id, r.file_id, r.page_id, 'DETAILED') AS page_info
WHERE er.wait_type like '%page%';
GO
```

Results　Messages

	session_id	wait_type	wait_resource	object_name	blocking_sessio…	command	statement_text	database_id	file

Memory-Optimized TempDB Metadata が無効の時とは異なり、ラッチ待機が発生していないことが確認できます。

処理時間の確認

ostress コマンドの実行画面に戻り、ワークロードが完了するまで待機して実行時間を確認します。

今回の環境では、実行時間は「00:00:22.268」でした。

以下のように Memory-Optimized TempDB Metadata が有効の場合は、無効の場合と比較して実行時間が改善していることが分かります。

Memory-Optimized TempDB Metadata	実行時間
無効	00:01:08.160
有効	00:00:22.268

➢ Enhanced PMEM support（Hybrid Buffer Pool & Enlightenment）

1. 概要

SQL Server 2019 では Persistent Memory（PMEM）を活用する新たな機能として、Hybrid Buffer Pool と Enlightenment が導入されます。

Hybrid Buffer Pool 機能は PMEM 上に配置されたデータファイルに対する Memory-mapped I/O による直接アクセス（DAX）により、従来必要としていたバッファプールへのデータページのコピー処理のオーバーヘッドを排除します。この動作はクリーンページの読み取りを対象にしており、変更されるデータページ（ダーティページ）は従来どおりバッファプールにコピーされる動作になります。

Enlightenment 機能はクリーンページの読み取りを対象とした Hybrid Buffer Pool 機能とは異なり、PMEM 上のデータファイルおよびログファイルに対する Read/Write にも DAX を利用しパフォーマンスを改善します。

2. 利用シーン

PMEM とはメインメモリ（DRAM）と同様に DIMM スロットに装填するフラッシュストレージデバイスです。その特徴は DRAM が揮発性メモリである一方で PMEM は不揮発性メモリであるためサーバーへの電源供給を失っても書き込まれたデータが失われることはありません。また従来のフラッシュストレージデバイス SSD と比較しレイテンシーが飛躍的に改善されています。

一般的に PMEM は DRAM に比較し高レイテンシーな代わりに容量単価が安く、SSD に比較し容量単価が高い代わりに低レイテンシーです。次の図は今回の検証に利用した Intel 社の Intel® Optane™ DC Persistent Memory の関連ページからの引用です。

SQL Server の最初の PMEM サポートは SQL Server 2016 で導入された Persisted Log Buffer 機能です。トランザクションによって生成されるログレコードはメモリ（DRAM）上のログバッファに順次書き出され、コミットレコードの検出もしくはログバッファ（ここではメインメモリ上のログバッファ）がいっぱい

になると即時、トランザクションログファイルに対して永続化保証のための物理 I/O が発行されます。これは従来の動作です。

Persisted Log Buffer 機能はログバッファを PMEM 上に配置します。DRAM と比較し PMEM へのログバッファへの書き出しは永続化されるため、コミットレコードの検出に伴うトランザクションログファイルへの物理 I/O が不要となり、従来比較で I/O 命令が削減され性能向上を図っています。

そして SQL Server 2019 では PMEM を活用する新たな機能として Hybrid Buffer Pool と Enlightenment（Enlightenment は Linux 版のみ）が導入されます。

Hybrid Buffer Pool 機能は PMEM 上に配置されたデータファイルに対する Memory-mapped I/O による直接アクセス（DAX アクセス）により、従来必要としていたバッファプールへのデータページのコピー処理のオーバーヘッドを排除します。この動作はクリーンページの読み取りを対象にしており、変更されるデー

タページ（ダーティページ）は従来どおりバッファプールにコピーされる動作になります。こうした仕組みから Read の割合が多いデータベースでより有効に機能すると言えます。

Enlightenment 機能はクリーンページの読み取りを対象とした Hybrid Buffer Pool 機能とは異なり、PMEM 上のデータファイルおよびログファイルに対する Read/Write にも DAX を利用しパフォーマンスを改善します。執筆時点で Enlightenment 機能は SQL Server on Linux に限定された機能です。

3. PMEM 検証

ここでは Hybrid Buffer Pool 機能と Enlightenment 機能の検証結果を共有します。
検証ではそれぞれの機能について次に示すワークロードと検証条件との組み合わせで性能差を比較しました。

- **Hybrid Buffer Pool（HBP）**
  - **検証目的**

    HBP 機能の性能改善効果を確認する。
  - **検証条件**

    ① HBP 無効（HBP 機能の性能改善効果を確認する）

    ② HBP 有効（HBP による新しいバッファ管理による性能測定）
  - **検証ワークロード**

    ✓ TPC-C（OLTP ワークロードの疑似）

    ✓ TPC-H（DWH ワークロードの疑似）

- **Enlightenment**
  - **検証目的**

    PMEM 導入による性能改善効果を確認する。
  - **検証条件**

    ① NVMe SSD 上にデータベースを配置（従来ストレージへのブロックアクセスによる性能測定）

    ② sector モードで構成した PMEM 上にデータベースを配置（PMEM へのブロックアクセス（Enlightenment 無効）による性能測定）

    ③ fsdax モードで構成した PMEM 上にデータベースを配置（PMEM への DAX アクセス（Enlightenment 有効）による性能測定）
  - **検証ワークロード**

    ✓ TPC-C（OLTP ワークロードの疑似）

    ✓ TPC-H（DWH ワークロードの疑似）

✱ 検証結果サマリ

Hybrid Buffer Pool : HBP 機能の性能改善効果を確認する

ワークロードをチューニングすることなく HBP を有効化することだけで TPC-C ワークロードがおよそ 5%の性能改善、TPC-H のワークロードがおよそ 25%の性能改善を得られています。

Hybrid Buffer Pool（HBP）	性能改善効果
TPC-C ワークロード（スループット）	5 %
TPC-H ワークロード（ターンアラウンドタイム）	25 %

Hybrid Buffer Pool 機能は、PMEM 上に配置されたデータファイルに対する Memory-mapped I/O による直接アクセス（DAX アクセス）により、従来必要としていたバッファプールへのデータページのコピー処理のオーバーヘッドを排除します。この動作はクリーンページの読み取りを対象にしており、変更されるデータページ（ダーティページ）は従来どおりバッファプールにコピーされる動作になります。こうした仕組みから Read の割合が多いデータベースでより有効に機能すると考えられ、その通り今回の検証では TPC-H のワークロードに対しより効果的である結果が得られています。

Enlightenment：PMEM 導入による性能改善効果を確認する

従来ストレージ（NVMe SSD）を fsdax モードで構成した PMEM に置き換えることで TPC-C ワークロードが 100%の性能改善、TPC-H のワークロードが 75%の性能改善を得られています。

Enlightenment	性能改善効果
TPC-C ワークロード	100 %
TPC-H ワークロード	75 %

Enlightenment 機能はクリーンページの読み取りを対象とした Hybrid Buffer Pool 機能とは異なり、PMEM 上のデータファイルおよびログファイルに対する Read/Write にも DAX を利用したパフォーマンス改善を図っており TPC-C、TPC-H いずれのワークロードに効果的である結果が得られています。

✱ 検証条件

検証環境

本検証はインテル株式会社のご協力をいただきインテル® データセンター CoE にて実施しました。

◇ インテル® データセンター CoE

https://www.intel.co.jp/content/www/jp/ja/data-center/data-center-coe.html

検証機

- **SQL Server 2019 サーバー**

Server	CLX S2600WF
CPU	Intel® Xeon® Platinum 8276L Processor@2.20GHz x2（28 コア×2CPU）
Memory	PC4-2666 32GBx12
AEP	Intel® Optane™ DC Persistent Memory 256GBx12
Disk	Intel(R) SSD DC P3700 400GB
Network	Intel(R) Ethernet Network Connection OCP X557-T2 10GbE
OS	Red Hat Enterprise Linux release 8.0 (Ootpa) Linux Version 4.18.0-80.el8.x86_64
SQL Server	Microsoft SQL Server 2019 RTM CU2 (15.0.4013.40) Enterprise Edition: Core-based Licensing (64-bit)

- **ベンチマーク用クライアント**

Server	BDW S2600WT
CPU	Intel® Xeon® Processor E5-2680 v4@2.40GHz x2（14 コア×2CPU）
Memory	PC4-2400 32Gx8
Disk	Intel(R) SSD DC P3700 400GB
Network	Intel(R) Ethernet Converged Network Adapter X550-T2
OS	Microsoft Windows Server 2019 Datacenter（10.0.17763）
ベンチマークツール	HammerDB（TPC-C と TPC-H に対応した OSS のベンチマークツールです） https://www.hammerdb.com/

データベース

- **配置**

HBP 検証	fsdax モードで構成した PMEM 上に配置
Enlightenment 検証	検証条件に応じて配置場所を変更 ・従来ストレージへのブロックアクセスによる性能測定 → NVMe SSD 上にデータベースを配置 ・PMEM へのブロックアクセス（Enlightenment 無効）による性能測定 → sector モードで構成した PMEM 上に配置 ・PMEM への DAX アクセス（Enlightenment 有効）による性能測定 → fsdax モードで構成した PMEM 上に配置

- **サイズ**

	TPC-C 用 DB	TPC-H 用 DB	tempdb
データファイル数	8	8	8
データファイルサイズ	16 GB	22 GB	8 GB
ログファイルサイズ	16 GB	16 GB	8 GB
データ量	100 GB	150 GB	–

SQL Server 2019 サーバーのパラメーター

- **カーネルパラメーター**

ファイルシステム	xfs
ファイルシステムブロックアロケーション	2MB
作成するファイルサイズ	2MB の倍数
Transparent Huge Pages (THP)	有効
XFS with shared copy-on-write data extents	無効（RHEL8.0 の機能で DAX 非対応のため無効化）

- **SQL Server インスタンスパラメーター**

トレースフラグ 843	有効
トレースフラグ 3979	検証条件に応じて変更 ・Memory-mapped I/O アクセス（fsdax モード PMEM） → 有効 ・ブロックアクセス（sector モード PMEM、NVMe SSD） → 無効
max server memory	ワークロードに応じて調整 TPC-C：既定値 TPC-H：16 GB（=データサイズの 1/10）
PROCESS AFFINITY	ワークロードに応じて調整 TPC-C：既定値（NUMANODE=0,1） TPC-H：DB を配置した NUMANODE を設定 ・fsdax モード PMEM：NUMANODE=1 ・sector モード PMEM：NUMANODE=0 ・NVMe SSD：NUMANODE=0
max degree of parallelism	ワークロードに応じて調整 TPC-C：1 TPC-H：56

ワークロードのパラメーター

- **TPC-C**

Number of Warehouses	1000（結果としてデータ量がおよそ 100GB のデータベースになります）
Minutes for Test Duration	3（ベンチマーク期間は 3 分間としています）
Virtual User	28（検証マシンの CPU スペックを元に設定しています）
その他操作	測定の都度 DB 再構築と SQL Server サービスの再起動を実施。

- **TPC-H**

Scale Factor	100（結果としてデータ量およそ 150GB の DB になります）
Clusterd Columnstore	無効（インデックスは B-Tree インデックスを設定しています）
MAXDOP	56（検証マシンの CPU スペックを元に設定しています）
その他操作	測定の都度 SQL Server サービスの再起動を実施。

データベースサイズ

- **TPC-C**

テーブル名	レコード件数	データサイズ（MB）	インデックスサイズ（MB）
customer	30000000	21307.14	4759.39
district	10000	78.13	78.55
history	30000000	1767.22	0.13
item	100000	9.20	0.10
new_order	9000000	156.60	0.42
orders	30000000	956.87	555.65
order_line	299996601	19211.10	46.51
stock	100000000	31250.00	65.99
warehouse	1000	7.81	7.98

- **TPC-H**

テーブル名	レコード件数	データサイズ（MB）	インデックスサイズ（MB）
customer	15000000	2620.06	5.81
lineitem	599999758	93538.83	30823.33
nation	25	0.01	0.02
part	20000000	2948.31	6.57
partsupp	80000000	12638.59	1429.16
region	5	0.01	0.01
supplier	1000000	155.97	14.38
orders	150000000	19938.17	3870.38

✱ 検証結果

Hybrid Buffer Pool（HBP）

- **TPC-C（28 Threads）**

TPC-C ワークロードに対するスループットを測定した結果です。

HBP 無効	26,290 [tps]
HBP 有効	27,392 [tps]

ワークロードをチューニングすることなく、HBP を有効化することだけでおよそ 5%の性能向上が得られています。

論理 Read I/O 量（Page lookups/sec）は大差がありませんが、物理 Read I/O 量（Page reads/sec）の大差が特徴的な相違です。

待機上位（wait_time_ms）：HBP 無効	待機上位（wait_time_ms）：HBP 有効
SOS_WORK_DISPATCHER　47739140）	SOS_WORK_DISPATCHER（45297530）
LOGMGR_QUEUE（1605370）	LOGMGR_QUEUE（1599256）
CXCONSUMER（1327019）	CXCONSUMER（1356890）
SLEEP_TASK（612957）	SLEEP_TASK（612943）
LAZYWRITER_SLEEP（483902）	LAZYWRITER_SLEEP（483824）
WRITELOG（300496）	WRITELOG（307498）
SP_SERVER_DIAGNOSTICS_SLEEP（299999）	XE_DISPATCHER_WAIT（273135）
XE_DISPATCHER_WAIT（274111）	REQUEST_FOR_DEADLOCK_SEARCH（245004）
CHECKPOINT_QUEUE（268363）	XE_TIMER_EVENT（244954）
XE_TIMER_EVENT（244955）	SQLTRACE_INCREMENTAL_FLUSH_SLEEP（244004）

- **参考：TPC-C（56 Threads）**

ベンチマークツールからの負荷を 2 倍（Virtual User 数を 28->56）にした結果は次の通り、それぞれスループットが上昇していることが確認できます。この条件においても HBP を有効化することだけでおよそ 5%の性能向上が得られています。

	28 Threads	56 Threads	上昇率
HBP 無効	26,290 [tps]	42,487 [tps]	60 [%]
HBP 有効	27,392 [tps]	44,319 [tps]	60 [%]

I/O（HBP無効）
15000000
12000000
9000000
6000000
3000000
0
180000
150000
120000
90000
60000
30000
0
0:00:01
0:00:11
0:00:21
0:00:31
0:00:41
0:00:51
0:01:01
0:01:11
0:01:21
0:01:31
0:01:41
0:01:51
0:02:01
0:02:11
0:02:21
0:02:31
0:02:41
0:02:51
0:03:01
Page lookups/sec
Page reads/sec
Readahead pages/sec
Page writes/sec

I/O（HBP有効）
15000000
12000000
9000000
6000000
3000000
0
180000
150000
120000
90000
60000
30000
0
0:00:01
0:00:11
0:00:21
0:00:31
0:00:41
0:00:51
0:01:01
0:01:11
0:01:21
0:01:31
0:01:41
0:01:51
0:02:01
0:02:11
0:02:21
0:02:31
0:02:41
0:02:51
0:03:01
Page lookups/sec
Page reads/sec
Readahead pages/sec
Page writes/sec

Memory（HBP無効）
300
250
200
150
100
50
0
0:00:01
0:00:10
0:00:19
0:00:28
0:00:37
0:00:46
0:00:55
0:01:04
0:01:13
0:01:22
0:01:31
0:01:40
0:01:49
0:01:58
0:02:07
0:02:16
0:02:25
0:02:34
0:02:43
0:02:52
0:03:01
Database Cache Memory (GB)
Free Memory (GB)
Stolen Server Memory (GB)
Total Server Memory (GB)

Memory（HBP有効）
300
250
200
150
100
50
0
0:00:01
0:00:10
0:00:19
0:00:28
0:00:37
0:00:46
0:00:55
0:01:04
0:01:13
0:01:22
0:01:31
0:01:40
0:01:49
0:01:58
0:02:07
0:02:16
0:02:25
0:02:34
0:02:43
0:02:52
0:03:01
Database Cache Memory (GB)
Free Memory (GB)
Stolen Server Memory (GB)
Total Server Memory (GB)

CPU & TPS（HBP無効）
100
90
80
70
60
50
40
30
20
10
0
60000
50000
40000
30000
20000
10000
0
0:00:01
0:00:18
0:00:35
0:00:52
0:01:09
0:01:26
0:01:43
0:02:00
0:02:17
0:02:34
0:02:51
%user
%system
Transactions/sec

CPU & TPS（HBP有効）
100
90
80
70
60
50
40
30
20
10
0
60000
50000
40000
30000
20000
10000
0
0:00:01
0:00:18
0:00:35
0:00:52
0:01:09
0:01:26
0:01:43
0:02:00
0:02:17
0:02:34
0:02:51
%user
%system
Transactions/sec

待機上位（wait_time_ms）：HBP 無効	待機上位（wait_time_ms）：HBP 有効
SOS_WORK_DISPATCHER（64120648）	SOS_WORK_DISPATCHER（71997677）
CXCONSUMER（2212164）	CXCONSUMER（2220761）
LOGMGR_QUEUE（1617750）	LOGMGR_QUEUE（1477330）
WRITELOG（690439）	WRITELOG（754047）
SLEEP_TASK（638144）	SLEEP_TASK（588919）
LAZYWRITER_SLEEP（505833）	LAZYWRITER_SLEEP（465819）
CHECKPOINT_QUEUE（360757）	CHECKPOINT_QUEUE（362092）
SP_SERVER_DIAGNOSTICS_SLEEP（300001）	XE_DISPATCHER_WAIT（316979）
XE_DISPATCHER_WAIT（274576）	LATCH_EX（312396）
XE_TIMER_EVENT（254951）	SP_SERVER_DIAGNOSTICS_SLEEP（299999）

- **TPC-H**

TPC-H ワークロードに対するターンアラウンドタイムを測定した結果です。

HBP 無効	281 [s]
HBP 有効	214 [s]

ワークロードをチューニングすることなく、HBP を有効化することだけでおよそ 25%の性能向上が得られています。

TPC-C と同じく物理 Read I/O 量（Page reads/sec）や CPU 使用の内訳に差があることがわかります。また、TPC-H は Read Only であることから、HBP 有効のケースではバッファプールが利用

されることがないためにバッファプールの空き（Free Memory）が変動していませんが、HBP 無効のケースではバッファプールが常に空きがない状態で推移しているのが特徴的です。

待機上位（wait_time_ms）：HBP 無効	待機上位（wait_time_ms）：HBP 有効
SOS_WORK_DISPATCHER（40214151）	SOS_WORK_DISPATCHER（42549088）
LOGMGR_QUEUE（1805109）	LOGMGR_QUEUE（2291895）
CXCONSUMER（1152366）	CXCONSUMER（1418774）
SLEEP_TASK（655877）	SLEEP_TASK（827759）
WAITFOR（481010）	WAITFOR（620393）
CXPACKET（454058）	CXPACKET（508385）
XE_DISPATCHER_WAIT（304367）	REQUEST_FOR_DEADLOCK_SEARCH（330026）
QDS_PERSIST_TASK_MAIN_LOOP_SLEEP（300013）	SQLTRACE_INCREMENTAL_FLUSH_SLEEP（328060）
PWAIT_EXTENSIBILITY_CLEANUP_TASK（300001）	DIRTY_PAGE_POLL（327342）
SP_SERVER_DIAGNOSTICS_SLEEP（300000）	HADR_FILESTREAM_IOMGR_IOCOMPLETION（327267）

Enlightenment

- **TPC-C（28 Threads）**

TPC-C ワークロードに対するスループットを測定した結果です。

NVMe SSD	13,828 [tps]
sector PMEM（Enlightenment 無効）	24,037 [tps]
fsdax PMEM（Enlightenment 有効）	26,290 [tps]
fsdax PMEM（Enlightenment 有効 & HBP 有効）	27,392 [tps]

従来ストレージ（NVMe SSD）を PMEM（Enlightenment 無効）に置き換えることでおよそ 75% の性能向上が得られています。

また、従来ストレージ（NVMe SSD）を PMEM（Enlightenment 有効）に置き換えることでおよそ 90%の性能向上が得られています。さらに HBP 有効と組み合わせた場合には 100%の性能向上が得られています。

Block アクセスとなる NVMe SSD と sector PMEM のケースはほぼリソース利用が似た波形をしており、Memory-mapped I/O による DAX アクセスとなる fsdax PMEM のケースでは先行読み取り（Readahead pages/sec）の機能が一切動作していない点が特徴的です。

待機上位（wait_time_ms）：NVMe SSD	待機上位（wait_time_ms）：sector PMEM	待機上位（wait_time_ms）：fsdax PMEM
SOS_WORK_DISPATCHER（50461856）	SOS_WORK_DISPATCHER（57794695）	SOS_WORK_DISPATCHER（47739140）
LOGMGR_QUEUE（1446522）	LOGMGR_QUEUE（1496394）	LOGMGR_QUEUE（1605370）
WRITELOG（978545）	CXCONSUMER（1300810）	CXCONSUMER（1327019）
PREEMPTIVE_OS_FLUSHFILEBUFFERS（784104）	SLEEP_TASK（600943）	SLEEP_TASK（612957）

CXCONSUMER（719777）	LAZYWRITER_SLEEP（475814）	LAZYWRITER_SLEEP（483902）
PAGEIOLATCH_SH（700053）	WRITELOG（399366）	WRITELOG（300496）
SLEEP_TASK（578801）	SOS_WORKER_MIGRATION（345681）	SP_SERVER_DIAGNOSTICS_SLEEP（299999）
LAZYWRITER_SLEEP（457580）	CHECKPOINT_QUEUE（331297）	XE_DISPATCHER_WAIT（274111）
SP_SERVER_DIAGNOSTICS_SLEEP（299999）	SP_SERVER_DIAGNOSTICS_SLEEP（299999）	CHECKPOINT_QUEUE（268363）
CHECKPOINT_QUEUE（292636）	SQLTRACE_INCREMENTAL_FLUSH_SLEEP（240005）	XE_TIMER_EVENT（244955）

- **参考：TPC-C（56 Threads）**

ベンチマークツールからの負荷を 2 倍（Virtual User 数を 28->56）にした結果は次の通り、それぞれスループットが上昇していることが確認できます。またケースごとの性能差の開きが大きくなっていることも確認できます。

	28 Threads	56 Threads	上昇率
NVMe SSD	13,828 [tps]	18,723 [tps]	35 [%]
sector PMEM（Enlightenment 無効）	24,037 [tps]	35,811 [tps]	50 [%]
fsdax PMEM（Enlightenment 有効）	26,290 [tps]	42,487 [tps]	60 [%]
fsdax PMEM（Enlightenment 有効 & HBP 有効）	27,392 [tps]	44,319 [tps]	60 [%]

Memory（NVMe SSD）
300
250
200
150
100
50
0
0:00:01
0:00:15
0:00:29
0:00:43
0:00:57
0:01:11
0:01:25
0:01:39
0:01:53
0:02:07
0:02:21
0:02:35
0:02:49
0:03:03
Database Cache Memory (GB)
Free Memory (GB)
Stolen Server Memory (GB)
Total Server Memory (GB)

Memory（sector PMEM）
300
250
200
150
100
50
0
0:00:01
0:00:15
0:00:29
0:00:43
0:00:57
0:01:11
0:01:25
0:01:39
0:01:53
0:02:07
0:02:21
0:02:35
0:02:49
0:03:03
Database Cache Memory (KB)
Free Memory (KB)
Stolen Server Memory (KB)
Total Server Memory (KB)

Memory（fsdax PMEM）
300
250
200
150
100
50
0
0:00:01
0:00:15
0:00:29
0:00:43
0:00:57
0:01:11
0:01:25
0:01:39
0:01:53
0:02:07
0:02:21
0:02:35
0:02:49
0:03:03
Database Cache Memory (GB)
Free Memory (GB)
Stolen Server Memory (GB)
Total Server Memory (GB)

CPU（NVMe SSD）
100
90
80
70
60
50
40
30
20
10
0
60000
50000
40000
30000
20000
10000
0
0:00:01
0:00:28
0:00:55
0:01:22
0:01:49
0:02:16
0:02:43
%system
%user
Transactions/sec

CPU（sector PMEM）
100
90
80
70
60
50
40
30
20
10
0
60000
50000
40000
30000
20000
10000
0
0:00:01
0:00:28
0:00:55
0:01:22
0:01:49
0:02:16
0:02:43
%system
%user
Transactions/sec

CPU（fsdax PMEM）
100
90
80
70
60
50
40
30
20
10
0
60000
50000
40000
30000
20000
10000
0
0:00:01
0:00:28
0:00:55
0:01:22
0:01:49
0:02:16
0:02:43
%system
%user
Transactions/sec

待機上位（wait_time_ms）：NVMe SSD	待機上位（wait_time_ms）：sector PMEM	待機上位（wait_time_ms）：fsdax PMEM
SOS_WORK_DISPATCHER（48213822）	SOS_WORK_DISPATCHER（72475743）	SOS_WORK_DISPATCHER（64120648）
WRITELOG（3025521）	CXCONSUMER（1934741）	CXCONSUMER（2212164）
LOGMGR_QUEUE（2067143）	LOGMGR_QUEUE（1609918）	LOGMGR_QUEUE（1617750）
PREEMPTIVE_OS_FLUSHFILEBUFFERS（1724653）	WRITELOG（1097345）	WRITELOG（690439）
PAGEIOLATCH_SH（1077813）	SLEEP_TASK（693430）	SLEEP_TASK（638144）
CXCONSUMER（971330）	LAZYWRITER_SLEEP（550292）	LAZYWRITER_SLEEP（505833）
SLEEP_TASK（821951）	PREEMPTIVE_OS_FLUSHFILEBUFFERS（432187）	CHECKPOINT_QUEUE（360757）
LAZYWRITER_SLEEP（649021）	CHECKPOINT_QUEUE（355189）	SP_SERVER_DIAGNOSTICS_SLEEP（300001）
XE_DISPATCHER_WAIT（393954）	XE_DISPATCHER_WAIT（321222）	XE_DISPATCHER_WAIT（274576）
CHECKPOINT_QUEUE（393667）	QDS_PERSIST_TASK_MAIN_LOOP_SLEEP（300003）	XE_TIMER_EVENT（254951）

- **TPC-H**

TPC-H ワークロードに対するターンアラウンドタイムを測定した結果です。

NVMe SSD	787 [s]

sector PMEM（Enlightenment 無効）	246 [s]
fsdax PMEM（Enlightenment 有効）	281 [s]
fsdax PMEM（Enlightenment 有効 & HBP 有効）	214 [s]

従来ストレージ（NVMe SSD）を PMEM（Enlightenment 無効）に置き換えることでおよそ 70%の性能向上が得られました。
また、従来ストレージ（NVMe SSD）を PMEM（Enlightenment 有効）に置き換えることでおよそ 65%の性能向上が得られました。さらに HBP 有効と組み合わせた場合には 75%の性能向上が得られています。

NVMe SSD は I/O ネックが見られます。sector PMEM と fsdax PMEM との大きな相違は「先行読み取り（Readahead pages/sec）」です。sector PMEM では「先行読み取り」が機能していますが、fsdax PMEM では「先行読み取り」が一切機能していないことがわかり、このことが sector PMEM（＝従来のブロックアクセス）の性能が fsdax PMEM（DAX アクセス）の性能を上回った要因と考えられます。また下の表（sys. dm_io_virtual_file_stats カタログ）が示す通り、読み取りデータ量（num_of_bytes_read）はほぼ同じですが、sector PMEM は fsdax PMEM と比較して読み取り命令回数（num_of_reads）が少なく、読み取り待機時間（io_stall_read_ms）も短いことが確認でき、「先行読み取り」が機能した結果として読み取り命令回数や読み取り待機時間が減少していると考えられます。

dm_io_virtual_file_stats	num_of_bytes_read	num_of_reads	io_stall_read_ms
sector PMEM（Enlightenment 無効）	1,682,476,998,656	25,989,583	2,385,543
fsdax PMEM（Enlightenment 有効）	1,653,912,739,840	201,893,645	4,695,602

I/O（NVMe SSD）
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
0:00:01
0:01:20
0:02:39
0:03:58
0:05:17
0:06:36
0:07:55
0:09:14
0:10:33
0:11:52
Page lookups/sec
Page reads/sec
Readahead pages/sec
Page writes/sec

IO（sector PMEM）
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
0:00:01
0:00:26
0:00:51
0:01:16
0:01:41
0:02:06
0:02:31
0:02:56
0:03:21
0:03:46
Page lookups/sec
Page reads/sec
Readahead pages/sec
Page writes/sec

I/O（fsdax PMEM）
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
0:00:01
0:00:29
0:00:57
0:01:25
0:01:53
0:02:21
0:02:49
0:03:17
0:03:45
0:04:13
Page lookups/sec
Page reads/sec
Readahead pages/sec
Page writes/sec

Memory（NVMe SSD）
30
25
20
15
10
5
0
0:00:01
0:01:20
0:02:39
0:03:58
0:05:17
0:06:36
0:07:55
0:09:14
0:10:33
0:11:52
Database Cache Memory (GB)
Free Memory (GB)
Stolen Server Memory (GB)
Total Server Memory (GB)

Memory（sector PMEM）
30
25
20
15
10
5
0
0:00:01
0:00:26
0:00:51
0:01:16
0:01:41
0:02:06
0:02:31
0:02:56
0:03:21
0:03:46
Database Cache Memory (GB)
Free Memory (GB)
Stolen Server Memory (GB)
Total Server Memory (GB)

Memory（fsdax PMEM）
30
25
20
15
10
5
0
0:00:01
0:00:29
0:00:57
0:01:25
0:01:53
0:02:21
0:02:49
0:03:17
0:03:45
0:04:13
Database Cache Memory (GB)
Free Memory (GB)
Stolen Server Memory (GB)
Total Server Memory (GB)

待機上位（wait_time_ms）：NVMe SSD	待機上位（wait_time_ms）：sector PMEM	待機上位（wait_time_ms）：fsdax PMEM
SOS_WORK_DISPATCHER（78530542）	SOS_WORK_DISPATCHER（48470245）	SOS_WORK_DISPATCHER（40214151）
PAGEIOLATCH_SH（25169771）	LOGMGR_QUEUE（2189027）	LOGMGR_QUEUE（1805109）
LOGMGR_QUEUE（6179266）	CXCONSUMER（1370114）	CXCONSUMER（1152366）
CXCONSUMER（5883916）	SLEEP_TASK（791698）	SLEEP_TASK（655877）
SLEEP_TASK（2308000）	WAITFOR（579463）	WAITFOR（481010）

CXPACKET（1758617）	CXPACKET（465512）	CXPACKET（454058）
WAITFOR（1696752）	SQLTRACE_INCREMENTAL_FLUSH_SLEEP（316068）	XE_DISPATCHER_WAIT（304367）
XE_DISPATCHER_WAIT（958075）	REQUEST_FOR_DEADLOCK_SEARCH（315036）	QDS_PERSIST_TASK_MAIN_LOOP_SLEEP（300013）
QDS_PERSIST_TASK_MAIN_LOOP_SLEEP（900054）	XE_TIMER_EVENT（314970）	PWAIT_EXTENSIBILITY_CLEANUP_TASK（300001）
PWAIT_EXTENSIBILITY_CLEANUP_TASK（899998）	HADR_FILESTREAM_IOMGR_IOCOMPLETION（312803）	SP_SERVER_DIAGNOSTICS_SLEEP（300000）

✱ 設定手順補足

PMEM の設定

- **ユーティリティのインストール**

```
#yum install ndctl
```

- **Namespace の作成**

✓ fsdax モード

```
# ndctl create-namespace -f --mode=fsdax
```

✓ sector モード

```
# ndctl create-namespace -f --mode=sector
```

- **Namespace の確認**

```
# ndctl list
```

✓ fsdax モード

```
{
  "dev":"namespace1.0",
  "mode":"fsdax",
  "map":"dev",
  "size":1598128390144,
  "uuid":"d2ca2d0f-9d21-4faa-8a78-7452b67c2a7a",
  "sector_size":512,
  "align":2097152,
  "blockdev":"pmem1"
}
```

✓ sector モード

```
{
    "dev":"namespace0.0",
```

```
    "mode":"sector",
    "size":1621910495232,
    "uuid":"54ff92f0-f8f2-4af4-bf7a-cc0271ac943a",
    "sector_size":4096,
    "blockdev":"pmem0s"
}
```

ファイルシステムの設定

- **ファイルシステム作成**

✓ fsdax モード

```
# mkfs.xfs -m reflink=0 -f /dev/pmem1
# mount -o dax,noatime /dev/pmem1 /mnt/dax
# xfs_io -c "extsize 2m" /mnt/dax
```

✓ sector モード

```
# mkfs.xfs -m reflink=0 -f /dev/pmem0s
# mount -o noatime /dev/pmem0s /mnt/block
# xfs_io -c "extsize 2m" /mnt/block
```

✓ マウント

```
# vi /etc/fstab
```

実行後、表示されるファイルの内容の末尾に以下を追記し、上書き保存して終了します。

・fsdax モード

```
/dev/pmem1              /mnt/dax                xfs     dax,noatime     0 0
```

・sector モード

```
/dev/pmem0s             /mnt/block              xfs     noatime         0 0
```

- **権限設定**

✓ fsdax モード

```
# chmod 755 /mnt/dax
# chown mssql:mssql /mnt/dax
```

✓ sector モード

```
# chmod 755 /mnt/block
# chown mssql:mssql /mnt/block
```

Hybrid Buffer Pool の設定

- **HBP 設定**

✓ インスタンスレベルの有効化

```
ALTER SERVER CONFIGURATION SET MEMORY_OPTIMIZED HYBRID_BUFFER_POOL = ON;
GO
```

✓ インスタンスレベルの無効化

```
ALTER SERVER CONFIGURATION SET MEMORY_OPTIMIZED HYBRID_BUFFER_POOL = OFF;
GO
```

✓ データベースレベルの有効化

```
ALTER DATABASE <対象データベース名> SET MEMORY_OPTIMIZED = ON;
```

✓ データベースレベルの無効化

```
ALTER DATABASE <対象データベース名> SET MEMORY_OPTIMIZED = OFF;
```

- **SQL Server 再起動**

```
sudo systemctl restart mssql-server
```

- **設定確認**

```
-- インスタンスレベルの設定
SELECT * FROM sys.server_memory_optimized_hybrid_buffer_pool_configuration;
GO

-- データベースレベルの設定
SELECT name, is_memory_optimized_enabled FROM sys.databases
GO
```

エラーログに以下の出力があれば、Hybrid Buffer Pool が有効であると判断できます。

（HybridBuffer Pool が無効の場合、以下のメッセージ自体が出力されません。）

```
/var/opt/mssql/log/errorlog
2020-02-20 09:27:08.93 Server      Hybrid buffer Pool: proposing 629155840 hashPage
s for 4801 GiB of 合計 Persistent Memory
2020-02-20 09:27:10.46 Server      Hybrid buffer pool started with memory-optimized
 metadata.
```

トレースフラグの設定

- **設定確認**

```
sudo /opt/mssql/bin/mssql-conf traceflag 834 on
sudo /opt/mssql/bin/mssql-conf traceflag 3979on
```

- **SQL Server 再起動**

```
sudo systemctl restart mssql-server
```

- **SQL Server 再起動**

```
sudo systemctl restart mssql-server
```

Intel、インテル、Intel ロゴ、Intel Inside ロゴ、Xeon は、アメリカ合衆国および/またはその他の国における Intel Corporation またはその子会社の商標です。

1.3. Performance Enhancements

実現場でのパフォーマンス問題経験がフィードバックされた各種機能強化が導入されています。

- **Optimize for sequential key insert performance**
  従来は特定のページに対するレコード挿入が高多重で発生した際に、性能が大きく劣化する問題がありました。Optimize for sequential key insert performance を有効にすると、こうしたホットスポットに対する同時実行制御のアルゴリズム改善が作動し、特定のページに対するレコード挿入のパフォーマンスを改善します。

- **Reduced recompilations for workloads**
  従来は外側スコープで作成された一時テーブルに対する DML ステートメント（SELECT、INSERT、UPDATE、DELETE）の実行のたびに、その DML ステートメントが再コンパイルされるオーバーヘッドが発生していました。Reduced recompilations for workloads は、SQL Server 内部の改善（追加の軽量チェックを行うことで不要な再コンパイルを回避）により、CPU 使用効率を改善するとともにパフォーマンスを改善します。

- **Concurrent PFS updates**
  従来はテーブルへのデータ登録などオブジェクトへの領域割り当てが同時多発するシステムにおいて、Page Free Space（PFS）ページへのアクセス集中に起因したパフォーマンス劣化が発生するケースがありました。SQL Server 2019 では PFS ページへの排他制御の改善により、該当ケースにおけるパフォーマンスを改善します。

- **Indirect checkpoint scalability**
  従来は間接チェックポイント（Indirect Checkpoint）の内部仕様に起因したパフォーマンス劣化が発生するケースが稀にありました。SQL Server 2019 ではこの内部仕様が改善され、間接チェックポイントの安定性が強化されています。

- **Scheduler worker migration**
  従来は SQL Server 内部のタスクは関連付けられたワーカーないしスケジューラーを移動することはないため、そのタスクの重みによってはスケジューラーに関連付けられた論理 CPU 間で一時的に負荷の不均一が生じるケースがありました。SQL Server 2019 ではアイドル状態のスケジューラーへタスクを移動させて負荷の不均一を解消し、システム全体の CPU 利用効率を上げることでパフォーマンスを改善します。

➢ Optimize for sequential key insert performance

1. 概要

従来は特定のページに対するレコード挿入が高多重で発生した際に、性能が大きく劣化する問題がありました。Optimize for sequential key insert performance を有効にすると、こうしたホットスポットに対する同時実行制御のアルゴリズム改善が作動し、特定のページに対するレコード挿入のパフォーマンスを改善します。

2. 利用シーン

これは比較的直面しやすい性能問題（"Last page insert contention"）を緩和するための機能です。
"Last page insert contention"とは、ある特定のページに対するレコード挿入が高多重で発生した際に性能が大きく劣化する問題で、特に以下の条件に該当するケースにおいてより直面しやすくなる問題です。

- シーケンシャル番号や現在のタイムスタンプなど、常に増加または減少する値をインデックスの先頭列に指定している
- 行サイズが小さく、単一ページに対して多数の行が格納される
- 該当インデックスへの同時挿入スレッドの数、がスケジューラー数（＝論理コア数）を大きく超過している

例えば、クラスター化インデックスのデータ構造を考えてみます。クラスター化インデックスのデータ構造はリーフレベルにインデックスキー順に並べられたデータ行そのものが格納される構造をしています。もしこのインデックスのキーが単純増減する ID 列や日時列であった場合は、該当のインデックスへの新たなレコード挿入は常に特定ページ（インデックスの最後のページ）が対象になります。一方で、レコード挿入に伴う物理メモリ上に展開されたページの更新は、更新損失を防ぐためにラッチという仕組みによって直列化されます。こうした内部の仕様により、該当インデックスに対して多数ユーザーからの同時挿入が発生すると、特定ページの更新に対してラッチによる長い待ち行列が発生することになり、これが性能劣化を引き起こします。

従来はこの問題を回避するためにはアプリケーションやインデックス設計の見直しが必要で、さらにトレードオフを考えながら最適解を導かねばならない難しい問題でした。

この問題に対して、SQL Server 2019 で導入されたこの機能を有効にする（インデックスの定義で OPTIMIZE_FOR_SEQUENTIAL_KEY オプションを有効にする）と、同時実行数の増加に耐えるためのアルゴリズム改善（新しいスレッドがラッチを要求できるレートを制御し、スループットを高く保つ可能性の高いスレッドを優先する）が動作するようになります。これによりアプリケーションやインデックス設計に手を入れることなく、"Last page insert contention" 問題を緩和する選択肢が得られました。

3. やってみよう

それでは、Optimize for sequential key insert performance による性能改善を確認してみましょう。

✱ 事前準備

サンプルテーブルの作成

OPTIMIZE_FOR_SEQUENTIAL_KEY オプションが無効と有効のテーブルをそれぞれ作成します。

Optimize_for_sequential_key_insert_performance.sql

```
USE WideWorldImportersDW;
GO
-- テーブルと主キー作成（OPTIMIZE_FOR_SEQUENTIAL_KEY は無効に設定）
CREATE TABLE dbo.Order_SequentialKey(
  [Order Key] bigint IDENTITY(1,1) NOT NULL,
  [City Key] int NOT NULL,
  [Customer Key] int NOT NULL,
  [Stock Item Key] int NOT NULL,
  [Order Date Key] date NOT NULL,
  [Picked Date Key] date NULL,
  [Description] nvarchar(100) NOT NULL
CONSTRAINT PK_Fact_Order PRIMARY KEY CLUSTERED (
  [Order Key] ASC
));
GO

-- テーブルと主キー作成（OPTIMIZE_FOR_SEQUENTIAL_KEY は有効に設定）
CREATE TABLE dbo.Order_SequentialKey_Optimized(
  [Order Key] [bigint] IDENTITY(1,1) NOT NULL,
  [City Key] [int] NOT NULL,
  [Customer Key] [int] NOT NULL,
  [Stock Item Key] [int] NOT NULL,
  [Order Date Key] [date] NOT NULL,
  [Picked Date Key] [date] NULL,
  [Description] [nvarchar](100) NOT NULL
CONSTRAINT [PK_Fact_Order_Optimized] PRIMARY KEY CLUSTERED (
  [Order Key] ASC
) WITH (OPTIMIZE_FOR_SEQUENTIAL_KEY=ON));
GO
```

ostress ツールの準備

同時実行を再現するために、本ハンズオンでは ostress ツールを利用しています。ostress ツールのセットアップについては、SQL Server 2019 自習書「Introduction」-「クライアントツールのインストール」を参照してください。

✱ Optimize for sequential key insert performance 無効時の動作

まずは機能が無効な状態での性能を見ていきます。

クエリの実行

コマンドプロンプトから、以下のコマンドで ostress を実行します。（ここでは以下のコマンドを実行する bat ファイルを、コマンドプロンプトから実行します。）

Optimize_for_sequential_key_insert_performance.bat.txt

```
ECHO OFF

REM カレントディレクトリを任意のワークフォルダに移動
cd C:¥work

REM 待機の統計情報をリセット
sqlcmd -S localhost -U sa -P Passw0rd -Q "DBCC SQLPERF(""sys.dm_os_wait_stats"", CLEA
R);"

REM 128 スレッドから INSERT を繰り返し 1024 回実行
"C:¥Program Files¥Microsoft Corporation¥RMLUtils¥ostress.exe" -Slocalhost -Usa -PPassw
0rd -dWideWorldImportersDW -Q"SET NOCOUNT ON; INSERT INTO dbo.Order_SequentialKey([Cit
y Key],[Customer Key],[Stock Item Key],[Order Date Key],[Picked Date Key],[Descriptio
n]) VALUES (1,1,1,GETDATE(),GETDATE(),N'XXXXX');" -mstress -quiet -n128 -r1024 | FINDS
TR "QEXEC Starting Creating elapsed"

REM 待機の統計情報を取得（結果を wait_info_optimize_off.log に出力）
sqlcmd -S localhost -U sa -P Passw0rd -Q "SELECT * FROM sys.dm_os_wait_stats WHERE wai
t_time_ms > 0 ORDER BY wait_time_ms desc;" -o waitstats_optimize_off.log -W -s ,
```

スレッド数（-n128）は 128、スレッドごとに INSERT を 1,024 回繰り返し実行（-r1024）するコマンドです。なお、サーバー名（-Slocalhost）、ユーザー名（-Usa）、パスワード（-PPassw0rd）は環境に合わせて変更してください。

コマンドの完了を待ちます。

待機の統計情報（waitstats_optimize_off.log）を確認

出力された待機の統計情報「waitstats_optimize_off.log」を確認します。

出力内容から、ラッチ系の待機時間（wait_time_ms）を確認すると、以下のようになります。PAGELATCH や LATCH が上位に存在することも確認できます。

待機の種類	待機時間（ミリ秒）
PAGELATCH_EX	664,560
PAGELATCH_SH	179,058
LATCH_SH	147,945
LATCH_EX	103,869

実行時間の確認

ostress コマンドの実行画面に戻り、ワークロードが完了するまで待機して実行時間を確認します。

今回の環境では、実行時間は「00:00:13.630」でした。

✱ Optimize for sequential key insert performance 有効時の動作

次に機能が有効な状態での性能を見ていきます。

クエリの実行

コマンドプロンプトから、以下のコマンドで ostress を実行します。（ここでは以下のコマンドを実行する bat ファイルを実行します。）

```
Optimize_for_sequential_key_insert_performance.bat.txt
ECHO OFF

REM カレントディレクトリを任意のワークフォルダに移動
cd C:¥work

REM 待機の統計情報をリセット
sqlcmd -S localhost -U sa -P Passw0rd -Q "DBCC SQLPERF(""sys.dm_os_wait_stats"", CLEA
R);"

REM 128 スレッドから INSERT を繰り返し 1024 回実行
```

```
"C:¥Program Files¥Microsoft Corporation¥RMLUtils¥ostress.exe" -Slocalhost -Usa -PPassw
Ord -dWideWorldImportersDW -Q"SET NOCOUNT ON; INSERT INTO dbo.Order_SequentialKey_Opti
mized([City Key],[Customer Key],[Stock Item Key],[Order Date Key],[Picked Date Key],[D
escription]) VALUES (1,1,1,GETDATE(),GETDATE(),N'XXXXX')" -mstress -quiet -n128 -r1024
 | FINDSTR "QEXEC Starting Creating elapsed"

REM 待機の統計情報を取得（結果を wait_info_optimize_on.log に出力）
sqlcmd -S localhost -U sa -P PasswOrd -Q "SELECT * FROM sys.dm_os_wait_stats WHERE wai
t_time_ms > 0 ORDER BY wait_time_ms desc;" -o waitstats_optimize_on.log -W -s ,
```

先ほどと同様に、サーバー名（-Slocalhost）、ユーザー名（-Usa）、パスワード（-PPassw0rd）は環境に合わせて変更してください。

コマンドの完了を待ちます。

待機の統計情報（waitstats_optimize_on.log）を確認

出力された待機の統計情報「waitstats_optimize_on.log」を確認します。

Optimize for sequential key insert performance 無効時の「waitstats_optimize_off.log」で確認した待機の種類に、有効時の結果を当てはめると、以下の通り PAGELATCH や LATCH による待機が大幅に削減されていることが確認できます。

待機の種類	機能無効時の待機時間（ミリ秒）	機能有効時の待機時間（ミリ秒）
PAGELATCH_EX	664,560	34,805
PAGELATCH_SH	179,058	735
LATCH_SH	147,945	550
LATCH_EX	103,869	241

実行時間の確認

ostress コマンドの実行画面に戻り、ワークロードが完了するまで待機して実行時間を確認します。

今回の環境では、実行時間は「00:00:11.200」でした。

以下のように Optimize for sequential key insert performance が有効の場合は無効の場合と比較して実行時間が改善していることが分かります。

Optimize for sequential key insert performance	実行時間
無効	00:00:13.630
有効	00:00:11.200

今回の結果はラッチ競合の度合いが軽く決定的なボトルネックにはなっておらず、実行時間への影響はそれほど大きくありませんが、交通渋滞のメカニズムと同様に一度ボトルネックが発生してしまうとあっという間に要求がスタックされスループットが大幅に低下し、新たな要求が止まない限りスループットは回復しない「コンボイ現象」に陥ります。Optimize for sequential key insert performance は新しいスレッドがラッチを要求できるレートを制御し、スループットを高く保つ可能性の高いスレッドを優先することで、「コンボイ現象」が起こりうる状況を極力回避しています。

◇ 「コンボイ現象」
https://jimgray.azurewebsites.net/papers/Convoy%20Phenomenon%20RJ%202516.pdf

➢ Reduced recompilations for workloads

1. 概要

従来は外側スコープで作成された一時テーブルに対する DML ステートメント（SELECT、INSERT、UPDATE、DELETE）の実行のたびに、その DML ステートメントが再コンパイルされるオーバーヘッドが発生していました。Reduced recompilations for workloads は SQL Server 内部の改善（追加の軽量チェックを行うことで不要な再コンパイルを回避すること）により、CPU 使用効率を改善するとともにパフォーマンスを改善します。

2. 利用シーン

一次テーブルを介して複数のモジュールがデータを操作することは、モジュール結合度の高い作りであり、ソフトウェアの設計としては望ましくありませんが、実際にはこうした実装は少なくありません。こうした背景を考慮し、SQL Server 内部の改善が入り、外側スコープで作成された一時テーブルに対する DML ステートメント（SELECT、INSERT、UPDATE、DELETE）の実行のたびに発生していた DML ステートメントの再コンパイルが回避され、CPU 使用効率を改善するとともにパフォーマンスを改善します。

3. やってみよう

それでは Reduced recompilations for workloads による性能改善を確認してみましょう。

✱ 事前準備

SQL Server 2017 環境の準備

SQL Server 2017 以前の動作を確認する場合には、SQL Server 2017 を別途用意します。

◇ SQL Server のインストール ガイド
https://docs.microsoft.com/ja-jp/sql/database-engine/install-windows/install-sql-server?view=sql-server-2017

ストアドプロシージャの作成

SQL Server 2017 と SQL Server 2019 それぞれの環境にストアドプロシージャを作成します。

Reduced_recompilations_for_workloads.sql

```
USE WideWorldImportersDW;
GO
-- ストアドプロシージャ（InnerProc）を定義
CREATE PROC dbo.InnerProc
AS
BEGIN
  INSERT #t1 (ID) VALUES (CONVERT(int, RAND() * 1000));
  SELECT ID FROM #t1;
```

```
END;
GO
-- ストアドプロシージャ (OuterTroc) を定義
CREATE PROC dbo.OuterTroc
AS
BEGIN
  CREATE TABLE #t1 (ID INT);
  EXEC dbo.InnerProc;
END;
GO
```

パフォーマンスモニターの起動

再コンパイルとバッチリクエストの状況を確認するため、パフォーマンスモニターを起動します。

パフォーマンスモニターを起動したら以下のカウンターを設定します。

- SQL Statistics¥Batch Requests/sec
- SQL Statistics¥SQL Re-Compilations/sec

なお、この設定は SQL Server 2017 以前と SQL Server 2019 それぞれで行います。

ostress ツールの準備

本ハンズオンでは ostress ツールを利用しています。ostress ツールのセットアップについては SQL Server 2019 自習書「Introduction」-「クライアントツールのインストール」を参照ください。

✱ SQL Server 2017 以前（Reduced recompilations for workloads 無効時）の動作

まずは機能が無効な SQL Server 2017 以前での性能を見ていきます。

処理の実行

コマンドプロンプトから、以下のコマンドで ostress を実行します。

```
Reduced_recompilations_for_workloads.bat.txt
"C:¥Program Files¥Microsoft Corporation¥RMLUtils¥ostress.exe" -E -Slocalhost¥SQL2017 -
Usa -PPassw0rd -dWideWorldImportersDW -Q"EXEC dbo.OuterTroc" -mstress -quiet -n8 -r512
 | FINDSTR "QEXEC Starting Creating elapsed"
```

スレッド数（-n8）は 8、スレッドごとに OuterTroc ストアドプロシージャを 512 回繰り返し実行（-r512）するコマンドです。なお、サーバー名（-Slocalhost¥SQL2017）、ユーザー名（-Usa）、パスワード（-PPassw0rd）は環境に合わせて変更してください。

コマンドの完了を待ちます。

再コンパイルの発生状況を確認

「SQL Statistics¥SQL Re-Compilations/sec」より、再コンパイルが多発していることが確認できます。

実行時間の確認

ostress コマンドの実行画面に戻り、ワークロードが完了するまで待機して実行時間を確認します。

今回の検証での実行時間は「00:00:04.221」でした。

✱ SQL Server 2019（Reduced recompilations for workloads 有効時）の動作

次に機能が有効な SQL Server 2019 での性能を見ていきます。

処理の実行

コマンドプロンプトから以下のコマンドで ostress を実行します。

Reduced_recompilations_for_workloads.bat.txt

```
"C:¥Program Files¥Microsoft Corporation¥RMLUtils¥ostress.exe" -E -Slocalhost -Usa -PPa
sswOrd -dWideWorldImportersDW -Q"EXEC dbo.OuterTroc" -mstress -quiet -n8 -r512 | FINDS
TR "QEXEC Starting Creating elapsed"
```

先ほどと同様に、サーバー名（-Slocalhost）、ユーザー名（-Usa）、パスワード（-PPassw0rd）は環境に合わせて変更してください。

コマンドの完了を待ちます。

再コンパイルの発生状況を確認

「SQL Statistics¥SQL Re-Compilations/sec」より、再コンパイルが全く発生していないことが確認できます。

実行時間の確認

ostress コマンドの実行画面に戻り、ワークロードが完了するまで待機して実行時間を確認します。

今回の検証での実行時間は「00:00:01.737」でした。

Reduced recompilations for workloads	実行時間
無効	00:00:04.221
有効	00:00:01.737

Reduced recompilations for workloads が有効の場合は無効の場合と比較して実行時間が改善していることが分かります。

➢ Concurrent PFS updates

1. 概要

従来はテーブルへのデータ登録などオブジェクトへの領域割り当てが同時多発するシステムにおいて、Page Free Space（PFS）ページへのアクセス集中に起因したパフォーマンス劣化が発生するケースがありました。SQL Server 2019 では PFS ページへの排他制御の改善により該当ケースにおけるパフォーマンスを改善します。

2. 利用シーン

PFS ページはデータファイル内の各ページの空き領域を管理する 8KB のシステムページです。PFS には該当ページが割り当て済みか、ページ内データの装填率、該当ページの属性といった情報が記録されており、SQL Server がテーブルなどのオブジェクトへ領域を割り当てるときに参照・更新されます。
ひとつの PFS ページは 8088 ページを管理しておりラッチによってアクセスが排他制御されます。そのため、オブジェクトへの領域割り当てが同時多発で発生するとラッチ競合が発生します。
PFS ページでのラッチ競合は tempdb においてより発生しやすい課題（詳細は SQL Server 2019 自習書「1. Intelligent Performance」-「Memory-Optimized TempDB Metadata」の項目を参照ください）ですが、ユーザーデータベースでも同様に発生しえます。
SQL Server 2019 では PFS ページへの排他制御の改善により該当ケースにおけるパフォーマンスを改善します。

3. やってみよう

それでは Concurrent PFS updates で得られるパフォーマンス向上効果を確認してみましょう。

✱ 事前準備

SQL Server 2017 環境の準備

SQL Server 2017 以前の動作を確認する場合には、SQL Server 2017 を別途用意します。

◇ SQL Server のインストール ガイド
https://docs.microsoft.com/ja-jp/sql/database-engine/install-windows/install-sql-server?view=sql-server-2017

ostress ツールの準備

本ハンズオンでは ostress ツールを利用しています。ostress ツールのセットアップについては SQL Server 2019 自習書「Introduction」-「クライアントツールのインストール」を参照してください。

✱ SQL Server 2017 以前の動作

ここでは SQL Server 2017 を利用して性能を測定します。

tempdb データファイル数の調整

今回は検証用に PFS ページのラッチ競合を起こりやすくするため、tempdb のデータファイルを 1 つで構成しています。tempdb のデータファイルを削除するためには対象のデータファイルを空にした後に削除します。以下に例を示しますがデータファイル論理名は環境に応じて読み替えてください。

Concurrent_PFS_updates.sql

```
USE tempdb;
GO
-- データファイルを空にする
DBCC SHRINKFILE (N'temp2', EMPTYFILE);
GO
-- データファイルを削除する
ALTER DATABASE tempdb REMOVE FILE temp2;
GO
-- データファイルを確認する
SELECT DB_NAME(database_id) AS db_name, file_id, type_desc, name, physical_name
FROM sys.master_files WHERE database_id = 2 AND type = 0;
GO
```

Results　Messages

	db_name	file_id	type_desc	name	physical_name
1	tempdb	1	ROWS	tempdev	C:\Program Files\Microsoft SQL Server\MSSQL14.SQL2017\MSSQL\DATA\tempdb.mdf

待機統計情報のクリア

性能測定前に待機統計情報をリセットします。

Concurrent_PFS_updates.sql

```
USE master;
GO
DBCC SQLPERF("sys.dm_os_wait_stats", CLEAR);
GO
```

ラッチ待機の発生状況を確認

1. コマンドプロンプトから、以下のコマンドで ostress を実行します。

Concurrent_PFS_updates.bat.txt

```
"C:¥Program Files¥Microsoft Corporation¥RMLUtils¥ostress.exe" -Slocalhost¥SQL2017 -
Usa -PPassw0rd -mstress -quiet -n128 -r1 -iPFS.sql | FINDSTR "QEXEC Starting Creati
ng elapsed"
```

スレッド数（-n128）は 128、スレッドごとに「PFS.sql」を一度だけ実行するコマンドです。なお、サーバー名（-Slocalhost¥SQL2017）、ユーザー名（-Usa）、パスワード（-PPassw0rd）は環境に合わせて変更してください。

「PFS.sql」の内容は以下の通り、一時テーブル（#Order_PFS）の作成と 100,000 行分の INSERT を行います。

Concurrent_PFS_updates.sql

```
CREATE TABLE #Order_PFS(
  [Order Key] bigint IDENTITY(1,1) NOT NULL,
  [City Key] int NOT NULL,
  [Customer Key] int NOT NULL,
  [Stock Item Key] int NOT NULL,
  [Order Date Key] date NOT NULL,
  [Picked Date Key] date NULL,
  [Description] nvarchar(100) NOT NULL
);
DECLARE @i int = 0;
WHILE(@i < 10000)
BEGIN
  INSERT INTO #Order_PFS([City Key],[Customer Key],[Stock Item Key],[Order Date Ke
y],[Picked Date Key],[Description]) VALUES(1,1,1,GETDATE(),GETDATE(),N'XXXXX');
  SET @i = @i + 1;
END;
```

2. ostress ツールの実行中に、別の新しいセッションから現在のラッチ待機の一覧を確認します。

Concurrent_PFS_updates.sql

```
USE master;
GO
SELECT
  er.session_id, er.wait_type, er.wait_resource,
  er.blocking_session_id, er.command,
  SUBSTRING(st.text, (er.statement_start_offset/2)+1,
    ((CASE er.statement_end_offset
        WHEN -1 THEN DATALENGTH(st.text)
        ELSE er.statement_end_offset
      END - er.statement_start_offset)/2) + 1) AS statement_text
FROM sys.dm_exec_requests AS er
  CROSS APPLY sys.dm_exec_sql_text(er.sql_handle) AS st
WHERE er.wait_type like '%latch%';
GO
```

Results　Messages

	session_id	wait_type	wait_resource	blocking_session_id	command	statement_text
1	64	PAGELATCH_UP	2:1:1	164	INSERT	INSERT INTO #Order_PFS([City…
2	65	PAGELATCH_UP	2:1:1	164	INSERT	INSERT INTO #Order_PFS([City…
3	67	PAGELATCH_UP	2:1:1	164	INSERT	INSERT INTO #Order_PFS([City…
4	68	PAGELATCH_UP	2:1:1	164	INSERT	INSERT INTO #Order_PFS([City…
5	69	PAGELATCH_UP	2:1:1	164	INSERT	INSERT INTO #Order_PFS([City…
6	70	PAGELATCH_UP	2:1:1	164	INSERT	INSERT INTO #Order_PFS([City…
7	71	PAGELATCH_UP	2:1:1	164	INSERT	INSERT INTO #Order_PFS([City…
8	72	PAGELATCH_UP	2:1:1	164	INSERT	INSERT INTO #Order_PFS([City…
9	73	PAGELATCH_UP	2:1:1	164	INSERT	INSERT INTO #Order_PFS([City…
10	74	PAGELATCH_UP	2:1:1	164	INSERT	INSERT INTO #Order_PFS([City…

上図のように wait_resource に対して RAGELATCH_UP 待機が発生しています。

wait_resource で示されている 2:1:1 は database_id:file_id:page_id に対応し、page_id=1 は PFS ページを指します。よって PFS ページに対して RAGELATCH_UP 待機が発生していることがわかります。

実行時間の確認

ostress コマンドの実行画面に戻り、ワークロードが完了するまで待機して実行時間を確認します。

```
管理者: C:¥Windows¥system32¥cmd.exe
ating elapsed"

c:¥work>"C:¥Program Files¥Microsoft Corporation¥RMLUtils¥ostress.exe" -Slocalhost¥SQL201
7 -Usa -PPassw0rd -mstress -quiet -n128 -r1 -iPFS.sql | FINDSTR "QEXEC Starting Creating
 elapsed"
02/28/20 13:25:39.223 [0x000017E0] Starting query execution...
02/28/20 13:25:39.229 [0x000017E0] Creating 128 thread(s) to process queries
02/28/20 13:25:39.241 [0x000017E0] PROGRESS: Creating thread 100
02/28/20 13:25:57.493 [0x000017E0] OSTRESS exiting normally, elapsed time: 00:00:18.440

c:¥work>
```

SQL Server 2017 での実行時間は「00:00:18.440」でした。

待機の統計情報を採取

待機の統計情報を採取します。

Concurrent_PFS_updates.sql

```
USE master;
GO
SELECT * FROM sys.dm_os_wait_stats WHERE wait_time_ms > 0 ORDER BY wait_time_ms desc;
GO
```

Results　Messages

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	PAGELATCH_UP	40044	2127002	134	18494
2	SLEEP_TASK	1527	1107216	2015	40
3	XE_DISPATCHER_WAIT	7	467541	119987	0
4	WAIT_XTP_OFFLINE_CKPT_NEW_LOG	88	440700	5015	3
5	REQUEST_FOR_DEADLOCK_SEARCH	88	440678	5017	440678
6	XE_TIMER_EVENT	124	440537	5014	435927
7	HADR_FILESTREAM_IOMGR_IOCOMPLETION	860	437403	529	23
8	DIRTY_PAGE_POLL	4031	437113	126	22
9	LAZYWRITER_SLEEP	437	436991	1016	12
10	SQLTRACE_INCREMENTAL_FLUSH_SLEEP	109	436825	4015	0

PAGELATCH_UP が待機時間の最も多かった待機であること確認でき、この待機時間は「2127002 ms」でした。

✱ SQL Server 2019 の動作

SQL Server 2017 で実行したものと同じワークロードを SQL Server 2019 に実行し性能測定します。

tempdb データファイル数の調整

SQL Server 2017 での実施と同様に、PFS ページのラッチ競合を起こりやすくするため tempdb のデータファイルを 1 つで構成しています。tempdb のデータファイルを削除するためには対象のデータファイルを空にした後に削除します。以下に例を示しますがデータファイル論理名は環境に応じて読み替えてください。

Concurrent_PFS_updates.sql

```
USE tempdb;
GO
-- データファイルを空にする
DBCC SHRINKFILE (N'temp2', EMPTYFILE);
GO
-- データファイルを削除する
ALTER DATABASE tempdb REMOVE FILE temp2
GO
-- データファイルを確認する
SELECT DB_NAME(database_id) AS db_name, file_id ,type_desc, name, physical_name
FROM sys.master_files WHERE database_id = 2 AND type = 0
GO
```

Results　Messages

	db_name	file_id	type_desc	name	physical_name
1	tempdb	1	ROWS	tempdev	C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\MSSQL\DATA\tempdb.mdf

待機統計情報のクリア

性能測定前に待機統計情報をリセットします。

Concurrent_PFS_updates.sql

```
USE master;
GO
DBCC SQLPERF("sys.dm_os_wait_stats", CLEAR);
GO
```

ラッチ待機の発生状況を確認

1. コマンドプロンプトから、以下のコマンドで ostress を実行します。

Concurrent_PFS_updates.bat.txt

```
"C:¥Program Files¥Microsoft Corporation¥RMLUtils¥ostress.exe" -Slocalhost -Usa -PPa
ssw0rd -mstress -quiet -n128 -r1 -iPFS.sql | FINDSTR "QEXEC Starting Creating elaps
ed"
```

スレッド数（-n128）は 128、スレッドごとに「PFS.sql」を一度だけ実行するコマンドです。なお、サーバー名（-Slocalhost）、ユーザー名（-Usa）、パスワード（-PPassw0rd）は環境に合わせて変更してください。

2. ostress ツールの実行中に、別の新しいセッションから現在のラッチ待機の一覧を確認します。

Concurrent_PFS_updates.sql

```
USE master;
GO
SELECT
  er.session_id, er.wait_type, er.wait_resource,
  OBJECT_NAME(page_info.object_id, page_info.database_id) as object_name,
  er.blocking_session_id,er.command,
  SUBSTRING(st.text, (er.statement_start_offset/2)+1,
    ((CASE er.statement_end_offset
        WHEN -1 THEN DATALENGTH(st.text)
        ELSE er.statement_end_offset
      END - er.statement_start_offset)/2) + 1) AS statement_text,
  page_info.database_id, page_info.file_id, page_info.page_id, page_info.object_id,
  page_info.index_id, page_info.page_type_desc
FROM sys.dm_exec_requests AS er
  CROSS APPLY sys.dm_exec_sql_text(er.sql_handle) AS st
  CROSS APPLY sys.fn_PageResCracker (er.page_resource) AS r
  CROSS APPLY
    sys.dm_db_page_info(r.db_id, r.file_id, r.page_id, 'DETAILED') AS page_info
WHERE er.wait_type like '%latch%';
GO
```

Results　Messages

session_id	wait_type	wait_resource	object_name	blocking_sessio…	command	st

SQL Server 2017 での実行とは異なり、PAGELATCH_UP のラッチ待機が発生していないことが確認できます。

実行時間の確認

ostress コマンドの実行画面に戻り、ワークロードが完了するまで待機して実行時間を確認します。

```
管理者: C:¥Windows¥system32¥cmd.exe
d"

c:¥work>"C:¥Program Files¥Microsoft Corporation¥RMLUtils¥ostress.exe" -Slocalhost -Usa -
PPassw0rd -mstress -quiet -n128 -r1 -iPFS.sql | FINDSTR "QEXEC Starting Creating elapsed
"
02/28/20 13:39:35.211 [0x00001708] Starting query execution...
02/28/20 13:39:35.213 [0x00001708] Creating 128 thread(s) to process queries
02/28/20 13:39:35.221 [0x00001708] PROGRESS: Creating thread 100
02/28/20 13:39:42.356 [0x00001708] OSTRESS exiting normally, elapsed time: 00:00:07.338

c:¥work>
```

SQL Server 2019 での実行時間は「00:00:07.338」でした。

Concurrent PFS updates	実行時間	ラッチ待機時間（ミリ秒）
SQL Server 2017	00:00:18.440	2,127,002
SQL Server 2019	00:00:07.338	0

ワークロード完了までの所要時間が大きく短縮されています（待機時間は並列で実行されている各スレッドの累積値であるため上記のように実行時間＜待機時間は起こりえます）。

待機の統計情報を採取

最後に待機の統計情報を採取します。

Concurrent_PFS_updates.sql

```
USE master;
GO
SELECT * FROM sys.dm_os_wait_stats WHERE wait_time_ms > 0 ORDER BY wait_time_ms desc;
GO
```

Results　Messages

	wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_ms
1	SOS_WORK_DISPATCHER	1006	142324768	1394053	20914
2	CHECKPOINT_QUEUE	2	1386466	1383040	1
3	SOS_SCHEDULER_YIELD	6435	758474	185	758470
4	SLEEP_TASK	227	159053	2008	16
5	WAIT_XTP_OFFLINE_CKPT_NEW_LOG	26	130185	5015	1
6	LOGMGR_QUEUE	5651	125117	143	5047
7	QDS_PERSIST_TASK_MAIN_LOOP_S…	2	120003	60003	0
8	XE_DISPATCHER_WAIT	2	120002	72290	0
9	REQUEST_FOR_DEADLOCK_SEARCH	13	65063	5013	65063
10	SQLTRACE_INCREMENTAL_FLUSH_S…	16	64081	4015	0

SQL Server 2017 での実行で最も多かった「PAGELATCH_UP」の待機が存在していないことがわかり Concurrent PFS updates が効いていることが確認できます。

➢ Indirect checkpoint scalability

1. 概要

従来は間接チェックポイント（Indirect Checkpoint）の内部仕様に起因したパフォーマンス劣化が発生するケースが稀にありました。SQL Server 2019 ではこの内部仕様が改善され間接チェックポイントの安定性が強化されています。

2. 利用シーン

データベースは Write Ahead Log という仕組みにより、データベースへの変更をトランザクションログへ先行して記録します。これにより、データが変更される都度データファイルへの同期書き込みを行うことなく、メモリ上のバッファプール内のデータページのみ変更することでパフォーマンス向上を図り、もしもデータベースがクラッシュした場合は、最後のチェックポイント以降のトランザクションログを分析することでデータファイルに未反映のデータの復旧処理を行います。

こうした仕組みを踏まえて、SQL Server のデータベース管理者はクラッシュリカバリの所要時間やそれに伴うシステム負荷をできるだけ予測できるものにするために、自動チェックポイント（recovery interval）の実行頻度を調整してきました。

そして SQL Server 2012 では新たなオプションとして間接チェックポイント（target_recovery_time）が導入されました。従来の自動チェックポイントはトランザクションレコードの生成量をしきい値として実行頻度を調整していましたが、間接チェックポイントは復旧時間との相関がより高いダーティページの量をしきい値として実行頻度を調整します。これによりクラッシュリカバリの処理時間やそれに伴うシステム負荷がこれまでよりも予測しやすいものになりました。

そして SQL Server 2016 では間接チェックポイントが既定で有効化される機能になりましたが、一部の特徴的なシステムにおいてはパフォーマンスに不利に働くケースもありました。

この問題に対して SQL Server 2019 では間接チェックポイントの内部仕様が改善されより安定性が強化されています。

3. 深堀りしよう

ここでは SQL Server 2017 以前における間接チェックポイントの振る舞いと課題について補足します。

間接チェックポイントは 3 つの主要コンポーネントがあります。

コンポーネント	役割
ダーティページリスト（DPList）	データベースごとのダーティページを追跡するためのリストです。 間接チェックポイントが有効化されるとデータベースエンジンによって作成され、データベースごとにパーティション化されて管理されます。
リカバリライター	定期的に起動して DPList をポーリングするバックグラウンドスレッドです。 ダーティページ数がしきい値を超えていることを検知すると DPList を最長の DPList から順に反復し始め、ページ ID を収集して DPList 内のページを

	ソートし、書き込みを最適化してまとめ、最後に非同期書き込み要求をポストします。
IO completion (IOC) ルーチン	ダーティページの IO を行うバックグラウンドスレッドです。 書き込み最適化された DPList を一度の IO 書き込み操作で最大 128 個の連続したページまで取得して IO を行い、IO が完了すると DPList からページを削除します。

「リカバリライター」による“書き込み最適化のための DPList アクセス”と「IOC ルーチン」による“IO 完了済みページを削除するための DPList アクセス”にはそれぞれでスピンロック（スレッドがロックを取得できるまで CPU を解放せずに待機するロック）が取得されるため、互いに競合し得ますが通常であればこれが問題になることはありません。しかしワークロードによっては特に tempdb の DPList で過度のスピンロック競合と指数関数的なバックオフが発生することがありました。こうした問題の発生はパフォーマンスの劣化とともに SQL Server のエラーログ、Dump、DMV、XEvent を通じて識別されます。以下はエラーログの例です。

```
Process 0:0:0 (0x34c0) Worker 0x000000FCD89BC160 appears to be non-yielding on Scheduler 9.
Thread creation time: 13140343858116. Approx Thread CPU Used: kernel 0 ms, user 60968 ms.
Process Utilization 13%. System Idle 86%. Interval: 70234 ms.
Process 0:0:0 (0x34c0) Worker 0x000000FCD89BC160 appears to be non-yielding on Scheduler 9.
Thread creation time: 13140343858116. Approx Thread CPU Used: kernel 0 ms, user 120718 ms.
Process Utilization 12%. System Idle 86%. Interval: 135452 ms.
Process 0:0:0 (0x34c0) Worker 0x000000FCD89BC160 appears to be non-yielding on Scheduler 9.
Thread creation time: 13140343858116. Approx Thread CPU Used: kernel 0 ms, user 172343 ms.
Process Utilization 12%. System Idle 87%. Interval: 195671 ms.
Process 0:0:0 (0x34c0) Worker 0x000000FCD89BC160 appears to be non-yielding on Scheduler 9.
Thread creation time: 13140343858116. Approx Thread CPU Used: kernel 0 ms, user 228828 ms.
Process Utilization 12%. System Idle 87%. Interval: 255890 ms.
```

```
A time-out occurred while waiting for buffer latch -- type 3, bp 0000010C1900C6C0, page
4:323520, stat 0x10f, database id: 2, allocation unit Id: 6488064, task 0x000000FE6076D088 :
0, waittime 300 seconds, flags 0x19, owning task 0x00000102ED0C8108. Not continuing to wait.
A time-out occurred while waiting for buffer latch -- type 3, bp 0000010C1900C6C0, page
4:323520, stat 0x10f, database id: 2, allocation unit Id: 6488064, task 0x000001033B2D7C28 :
0, waittime 300 seconds, flags 0x39, owning task 0x00000102ED0C8108. Not continuing to wait.
```

このような問題に直面した場合は間接チェックポイントの利用を中止することが対処になるため、クラッシュリカバリの所要時間を予測しにくい従来の自動チェックポイントで運用を行う必要がありました。

この問題に対して SQL Server 2019 では内部仕様が改善され間接チェックポイントの安定性が強化されています。

➢ Scheduler worker migration

1. 概要

従来、SQL Server 内部のタスクは関連付けられたワーカーないしスケジューラーを移動することはないため、そのタスクの重みによってはスケジューラーに関連付けられた論理 CPU 間で一時的に負荷の不均一が生じるケースがありました。SQL Server 2019 ではアイドル状態のスケジューラーへタスクを移動させて負荷の不均一を解消し、システム全体の CPU 利用効率を上げることでパフォーマンスを改善します。

2. 利用シーン

執筆時点では「可用性グループの並列 REDO タスク」が Scheduler worker migration に対応しています。

可用性グループの並列 REDO は SQL Server 2016 によって導入された機能です。SQL Server 2014 以前では、セカンダリデータベースへの REDO 処理はデータベースごとにシングルスレッドで処理されていたため、この処理がパフォーマンスのボトルネックとなるケースがありましたが、並列 REDO の導入により REDO 処理にスケーラビリティがもたらされました。

この並列 REDO はメインタスクと複数のワーカータスクで構成され、メインタスクは着信したログレコードの解析と各ワーカータスクへのディスパッチ、ワーカータスクは割り当てられたログレコードを処理し、データページに変更を適用する動きをします。

このメインタスクとワーカータスクの両方が CPU を比較的長時間利用するタスクであり、場合によってはこれらが同一のスケジューラーに割り当てられるケースがありえました。そうした場合同じスケジューラーに複数の長時間実行タスクが存在することになり、これに起因してシステム全体の CPU 使用効率が悪化しパフォーマンスが伸び悩むケースがありました。また、複数の可用性データベースが構成されている場合にはさらに REDO タスクが増えるため、より表面化しやすい問題でした。

こうした課題に対して SQL Server 2019 では Scheduler worker migration により長時間実行タスクを他のアイドル状態のスケジューラーに処理させることで、負荷の均一化をはかりシステム全体の CPU 使用効率を向上させます。

3. 深堀りしよう

ここでは SQL Server のタスクスケジューリングについて補足します。
SQL Server は各 CPU の使用率のバランスをとるために各スケジューラーが担当しているタスク数を追跡し、最も負荷が少ないスケジューラーに新たなタスクを割り当てます。

- **タスク**

  要求を満たすために完了する必要がある作業の単位。

- **ワーカー**

  オペレーティング システムのスレッドを論理的に表現したもの。タスクと関連付けられスケジューラー上でタスクを処理する。

- **スケジューラー**

  個々の論理 CPU ごとにスケジューラーがマップされワーカーを管理する。ワーカーがスケジューラー上でアクティブな状態を維持できる時間はクォンタムで管理されクォンタム時間が経過したワーカーは、CPU リソースへのアクセスを必要とする他のワーカーに時間を明け渡しその状態を変更する。 ワーカーの状態の変化はそのワーカーに関連付けられたタスクとタスクに関連付けられた要求に伝達される。

このタスク割当はタスクの重みを考慮しないため条件によっては長時間を要す複数のタスクが特定のスケジューラーに配置されるケースがあり、この場合スケジューラーごとすなわち論理 CPU ごとに負荷が不均一となり、またタスクとスケジューラーの関連付けは変更されることがないため、こうした長時間タスクが終了するまで不均一状態が解消されません。

この課題に対して SQL Server 2019 では Scheduler worker migration により、アイドル状態のスケジューラーは同じ NUMA ノード上の別のスケジューラーの実行可能キューからワーカーを移行し該当ワーカーのタスクをすぐに再開させるよう振る舞うことで、負荷の均一化をはかりシステム全体の CPU 使用効率を向上させます。

2. Mission-critical security

このセクションは、データ管理に不可欠なセキュリティに関する機能強化を取り上げます。SQL Server は米国標準技術局（NIST）による National Vulnerability Database で公開される脆弱性情報が極めて少ないデータベースとしても知られていますが、さらに SQL Server 2019 では次のセキュリティ機能が導入・強化されています。

- **Always Encrypted with Secure Enclaves**
  新しいセキュリティの仕組みである Secure Enclaves により、Always Encrypted の機能制限が緩和され、従来制限されていた暗号化列に対する比較検索や LIKE 検索が可能となり使い易さが向上しています。

- **Data Classification and auditing built-in**
  「データの検出と分類」機能が強化され、SQL Server 監査との統合や操作性が改善されました。これにより、機密性の高いデータの管理がこれまでよりもさらに容易になります。

- **TDE scan suspend and resume**
  Transparent Data Encryption（TDE）の有効化による暗号化処理の中断と再開ができるようになりました。これにより暗号化処理のスケジューリングが制御できるようになり、TDE が導入しやすくなります。

- **Certificate management in SQL Server Configuration Manager**
  SQL Server 構成マネージャーの改善により、SQL Server との通信を暗号化するための証明書管理操作が簡素化されました。また Always On 構成など、複数の SQL Server インスタンスへの展開も容易になっており、僅かな操作でより短期間に証明書の管理や展開が可能となります。

➢ Always Encrypted with secure enclaves

1. 概要

新しいセキュリティの仕組みである Secure Enclaves により、Always Encrypted の機能制限が緩和され、従来制限されていた暗号化列に対する比較検索や LIKE 検索が可能となり使い易さが向上しています。

2. 利用シーン

SQL Server 2016 は動的データマスク、行レベルセキュリティといった、セキュリティに関する新機能が多く導入されたバージョンでした。Always Encrypted も同じく SQL Server 2016 で新たに導入された暗号化機能で、これは数多くのリレーショナルデータベースの中でも、SQL Server のユニークな機能の一つです。

Always Encrypted によるデータの暗号化・復号化は、クライアントアプリケーションのドライバー層でアプリケーションに対し透過的に処理されます。すなわち、通信経路やデータベースファイルのみならず、メモリ上のデータベースバッファを含めて常時暗号化状態を維持しているため、データベースの特権ユーザーでさえもプレーンテキストを参照することができず、唯一、クライアントアプリケーション内でのみプレーンテキストとして復元されます。これにより、データベース管理者とそのデータベースの中身であるデータの管理者を切り離すことができ、例えばデータセンターオペレーターやクラウド業者に対してデータベース管理を委任しやすくなりました。

しかし、こうした強力なセキュア状態を実現することのトレードオフとして、データベース内の暗号化された列に対する操作が大きく制限されることになりました。具体的には大きく２つあり、ひとつは「暗号化列に対する検索に比較演算や LIKE 演算を利用することはできない」、もうひとつは「既にデータベースに存在するデータを暗号化するためにはデータをデータベースの外に取り出し、暗号化したのち再度戻すというような煩わしい手順を踏む必要がある」というもので、既存システムからの移行において大きな機能制限でした。

こうした課題に対して SQL Server 2019 は新しい仕組みとして Secure Enclaves を導入し、以下のように機能制限を緩和しています。

	Always Encrypted（2016～）		with Secure Enclaves（2019～）	
暗号化方式	ランダム暗号化	決定論的暗号化	ランダム暗号化	決定論的暗号化
クラスター化インデックス付与	×	△	×	△
非クラスター化インデックス付与	×	△	○	△
等価検索	×	○	○	○
比較検索	×	×	○	×
LIKE 検索	×	×	○	×
既存データの暗号化	×		○	
既存データの再暗号化	×		○	
既存データの復号	×		○	

新たに対応したランダム暗号化に対する等価検索、比較検索、LIKE 検索は“高度な計算”と呼ばれ、既存データに対する暗号化、再暗号化、復号は“インプレース暗号化”と呼ばれています。これらは SQL Server サーバー上にセキュリティで保護されたエンクレーブを構築し、そのエンクレーブ内でのみプレーンテキストに対する計算を許可することにより実現されています。 エンクレーブは SQL Server プロセス内のメモリの保護された領域となり、他のプロセスに対してはブラックボックスでデバッガーを使用してもエンクレーブ内のデータを参照することはできないセキュアな領域であるため、Always Encrypted のコンセプトである“データベース管理者とそのデータベースの中身であるデータの管理者を切り離す”が失われることはありません。

こうした“高度な計算”と“インプレース暗号化”のような、エンクレーブ処理を必要とする暗号化クエリの詳細処理フローを下図に示します。

上図の通りクライアントドライバーは、送信するクエリがエンクレーブ内での計算を伴うクエリである場合、クエリの送信先となる SQL Server のエンクレーブが、セキュリティで保護された正規のエンクレーブであることを事前に検証します。この検証プロセスをエンクレーブの構成証明と呼び、「ホストガーディアンサービス（HGS）」に照会をかける実装になっています。そのため Always Encrypted with Secure Enclaves を利用するためには HGS サーバーを構成する必要があります。

次にクライアントドライバーは SQL Server に対して「列マスターキーのストア場所」と「列マスターキーで暗号化された列暗号化キー値」を要求します。SQL Server は事前に設定済みの「列マスターキーのストア場所」と「列マスターキーで暗号化された列暗号化キー」を返却します。クライアントドライバーは返却

された「列マスターキーのストア場所」にある「列マスターキー」で「列マスターキーで暗号化された列暗号化キー値」を「列暗号化キー」として復号します。
SQL Server はクライアントドライバーから渡されたクエリがエンクレーブ処理を必要とする場合には「列暗号化キー」を共有するようクライアントドライバーに要求し、共有された「列暗号化キー」を利用してエンクレーブの境界でデータの暗号化・復号を行います。SQL Server サーバー内のセキュリティで保護されたエンクレーブは「Virtualization-based Security （VBS）」の Memory Enclaves で実現されています。そのため SQL Server サーバー上には「Host Guardian Hyper-V Support」機能を事前に構成しておく必要があります。

HGS や VBS の詳細については、次の Docs を確認ください。

◇ ホスト ガーディアン サービスの構成証明の計画
https://docs.microsoft.com/ja-jp/sql/relational-databases/security/encryption/always-encrypted-enclaves-host-guardian-service-plan?view=sql-server-ver15

◇ Virtualization-based security (VBS) memory enclaves: Data protection through isolation
https://www.microsoft.com/security/blog/2018/06/05/virtualization-based-security-vbs-memory-enclaves-data-protection-through-isolation/

3. やってみよう

それでは、Always Encrypted with secure enclaves を試してみましょう。「サーバーのセットアップ」、「クライアントのセットアップ」、「新機能 “インプレース暗号化” の動作確認」、「新機能 “高度な計算” の動作確認」の４つにわけて解説します。

＊ サーバーのセットアップ

ここでは以下の要件を満たすサーバー２台（「HGS をホストするためのサーバー」と「Always Encrypted with secure enclaves を利用する SQL Server をホストするためのサーバー」）が準備済みであることを前提に手順解説を始めます。

◇ ホスト ガーディアン サービスの構成証明の計画
…前提条件
https://docs.microsoft.com/ja-jp/sql/relational-databases/security/encryption/always-encrypted-enclaves-host-guardian-service-plan?view=sql-server-ver15#prerequisites

セットアップの流れは次の通りです。

No	対象サーバー	実施内容

1	HGS サーバー	HGS を構築する
2	SQL Server サーバー	SQL Server サーバーに構成証明用の機能をインストールする
3		構成証明用の HGS ホストキーを作成する
4		HGS ホストキーの公開キーをエクスポートする
5	HGS サーバー	公開キーを HGS サーバーにインポートする
6	SQL Server サーバー	SQL Server サーバーに構成証明を行う HGS サーバーを設定する
7		Always Encrypted with secure enclaves を有効にする
8		Always Encrypted with secure enclaves の高度な計算を有効にする
9		「列マスターキー」を作成する
10		「列暗号化キー」を作成する

1）HGS を構築する

@HGS サーバー

1. 管理者アカウントとしてサインイン後、[スタート] を右クリックして [Windows PowerShell (管理者)] を選択します。管理者 PowerShell コンソールで次のコマンドを実行し、HGS サーバーにホストガーディアンサービスの役割を追加します。

```
Always_Encrypted_with_Secure_Enclaves.ps.txt
Install-WindowsFeature -Name HostGuardianServiceRole -IncludeManagementTools -Restart
```

役割のインストールが完了すると、HGS サーバーが自動的に再起動されます。

2. HGS サーバーが再起動した後、再度、管理者 PowerShell コンソール画面を起動し、次のコマンドを実行してホストガーディアンサービスのインストールとそのドメインを構成します。

```
Always_Encrypted_with_Secure_Enclaves.ps.txt
$adminPassword = ConvertTo-SecureString -AsPlainText 'PassW0rd' -Force
Install-HgsServer -HgsDomainName 'bastion.local' -SafeModeAdministratorPassword $adminPassword -Restart
```

ここで指定したパスワードは、Active Directory のディレクトリ サービスの修復モードパスワードにのみ利用されるパスワードです。また、HGS サーバー上に作成される Active Directory ドメインのドメイン名を示している「bastion.local」は環境に応じて読み替えてください。
機能のインストールが完了すると、HGS サーバーが自動的に再起動されます。

3. HGS サーバーが再起動した後、再度、管理者 PowerShell コンソール画面を起動し、次のコマンドを実行して HGS でホストキーによる構成証明を使用できるようにします。

```
Always_Encrypted_with_Secure_Enclaves.ps.txt
Initialize-HgsAttestation -HgsServiceName 'hgs' -TrustHostKey
```

4. 管理者 PowerShell コンソール画面で次のコマンドを実行し、後の手順で使用する HGS サーバーの IP アドレスを取得します。

```
Always_Encrypted_with_Secure_Enclaves.ps.txt
Get-NetIPAddress
```

2）SQL Server サーバーに構成証明用の機能をインストールする

@SQL Server サーバー

1. 管理者アカウントとしてサインイン後、[スタート] を右クリックして [Windows PowerShell (管理者)] を選択します。管理者 PowerShell コンソール画面で次のコマンドを実行し、「Host

Guardian Hyper-V Support」機能をインストールします。もし SQL Server サーバー上に Hyper-V がインストールされていない場合は、併せて Hyper-V もインストールされます。

```
Always_Encrypted_with_Secure_Enclaves.ps.txt
Enable-WindowsOptionalFeature -Online -FeatureName HostGuardian -All
```

サーバーを再起動するように求められた場合は再起動します。

2. 使用している SQL Server サーバーが「仮想マシンである場合」もしくは「UEFI セキュアブートをサポートしていないか IOMMU が搭載されていない物理マシンの場合」は、プラットフォームのセキュリティ機能に関する VBS 要件を削除する必要があります。管理者 PowerShell コンソールで次のコマンドを実行して該当の VBS 要件を削除します。いずれにも当てはまらない場合はこの手順を飛ばして次に進みます。

```
Always_Encrypted_with_Secure_Enclaves.ps.txt
Set-ItemProperty -Path HKLM:¥SYSTEM¥CurrentControlSet¥Control¥DeviceGuard -Name Req
uirePlatformSecurityFeatures -Value 0
```

3. 管理者 PowerShell コンソールで次のコマンドを実行してコンピューターを再起動します。

```
Always_Encrypted_with_Secure_Enclaves.ps.txt
Restart-Computer
```

3）構成証明用の HGS ホストキーを作成する　@SQL Server サーバー

SQL Server サーバーが再起動した後、再度、管理者 PowerShell コンソール画面を起動し、次のコマンドを実行して、構成証明用のホストキー（非対称キー）を作成します。

```
Always_Encrypted_with_Secure_Enclaves.ps.txt
Set-HgsClientHostKey
```

4）HGS ホストキーの公開キーをエクスポートする　@SQL Server サーバー

1. 作成したホストキーの公開キーをエクスポートします。

```
Always_Encrypted_with_Secure_Enclaves.ps.txt
Get-HgsClientHostKey -Path $HOME¥Desktop¥hostkey.cer
```

```
管理者: Windows PowerShell
PS C:¥Users¥Administrator> Get-HgsClientHostKey -Path $HOME¥Desktop¥hostkey.cer

    ディレクトリ: C:¥Users¥Administrator¥Desktop

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        2020/02/28     22:35            816 hostkey.cer

PS C:¥Users¥Administrator>
```

2. SQL Server サーバーのデスクトップにエクスポートされたファイルを HGS サーバーのデスクトップにコピーします。

5）公開キーを HGS サーバーにインポートする　@HGS サーバー

管理者 PowerShell コンソール画面で次のコマンドを実行し、SQL Server サーバーでエクスポートした公開キーをインポートして HGS で管理される「保護されたホスト」に、SQL Server をホストするサーバーを登録します。

Always_Encrypted_with_Secure_Enclaves.ps.txt

```
Add-HgsAttestationHostKey -Name SERVER01 -Path $HOME¥Desktop¥hostkey.cer
```

```
管理者: Windows PowerShell
PS C:¥Users¥Administrator> Add-HgsAttestationHostKey -Name SERVER01 -Path $HOME¥Desktop¥hostkey.cer

Name     PublicKey
----     ---------
SERVER01 System.Security.Cryptography.X509Certificates.PublicKey

PS C:¥Users¥Administrator>
```

コマンド内で SQL Server サーバーのコンピューター名を示している「SERVER01」は環境に応じて読み替えてください。

6）SQL Server サーバーに構成証明を行う HGS サーバーを設定する　@SQL Server サーバー

管理者 PowerShell コンソール画面で次のコマンドを実行し、SQL Server サーバーに自身の構成証明を行う HGS サーバーの場所を設定します。

Always_Encrypted_with_Secure_Enclaves.ps.txt

```
Set-HgsClientConfiguration -AttestationServerUrl http://<HGS IP address or DNS name>/A
ttestation -KeyProtectionServerUrl http://<HGS IP address or DNS name>/KeyProtection/
```

コマンド内の 2 か所の<HGS IP address or DNS name>は HGS サーバーの IP アドレスまたは DNS 名に書き換えてください。

コマンドの結果として、「AttestationStatus = Passed」と表示されれば正常です。そうでない場合は、次の Docs を参考にトラブルシューティングします。

◇ ホスト ガーディアン サービスにコンピューターを登録する

…手順 5:ホストが正常に証明できることを確認する

https://docs.microsoft.com/ja-jp/sql/relational-databases/security/encryption/always-encrypted-enclaves-host-guardian-service-register?view=sql-server-ver15#step-5-confirm-the-host-can-attest-successfully

7）Always Encrypted with secure enclaves を有効にする　@SQL Server サーバー

1. Always Encrypted with secure enclaves を有効化します。

Always_Encrypted_with_Secure_Enclaves.sql

```
EXEC sys.sp_configure 'column encryption enclave type', 1;
RECONFIGURE;
```

2. SQL Server のサービスを再起動します。

Always_Encrypted_with_Secure_Enclaves.bat.txt

```
net stop mssqlserver
net start mssqlserver
```

8）Always Encrypted with secure enclaves の高度な計算を有効にする　@SQL Server サーバー

高度な計算を利用するためには 2 つの要件があり、ひとつは「トレースフラグ（127）を有効化すること」、もうひとつは「照合順序に BIN2 照合順序を利用すること」です。この要件を満たすための設定を実施します。

1. トレースフラグ（127）を設定し、高度な計算（比較検索、LIKE 検索）を有効にします。
   既定では暗号化された列に対する高度な計算は無効になっており、トレースフラグ（127）を設定することで高度な計算を使用することができるようになります。

   トレースフラグの詳しい設定手順については、次の Docs を確認ください。

   ◇ SCM サービス - サーバー起動オプションを構成する
   https://docs.microsoft.com/ja-jp/sql/database-engine/configure-windows/scm-services-configure-server-startup-options?view=sql-server-ver15

2. SQL Server のサービスを再起動します。

Always_Encrypted_with_Secure_Enclaves.bat.txt

```
net stop mssqlserver
net start mssqlserver
```

3. Always Encrypted with secure enclaves の動作確認に利用するデータベースとテーブルを作成してデータを投入します。高度な計算を利用するため、作成するデータベースの照合順序に BIN2 を指定しています。

Always_Encrypted_with_Secure_Enclaves.sql

```
CREATE DATABASE [ContosoHR] COLLATE Japanese_BIN2;
GO
USE [ContosoHR];
GO
CREATE TABLE dbo.Employees (
  [EmployeeID] [int] IDENTITY(1,1) NOT NULL,
  [SSN] [char](11) NOT NULL,
  [FirstName] [nvarchar](50) NOT NULL,
  [LastName] [nvarchar](50) NOT NULL,
  [Salary] [money] NOT NULL
) ON [PRIMARY];
INSERT INTO [dbo].[Employees]
        ([SSN]
        ,[FirstName]
        ,[LastName]
        ,[Salary])
    VALUES
        ('795-73-9838'
        , N'Catherine'
        , N'Abel'
        , $31692);
INSERT INTO [dbo].[Employees]
        ([SSN]
        ,[FirstName]
        ,[LastName]
        ,[Salary])
    VALUES
        ('990-00-6818'
        , N'Kim'
        , N'Abercrombie'
        , $55415);
GO
```

9)「列マスターキー」を作成する

@SQL Server サーバー

1. SSMS を使用し、sysadmin 権限を持つユーザーで SQL Server に接続します。

2. まず、列マスターキーを作成します。オブジェクトエクスプローラーで［データベース］-［ContosoHR］-［セキュリティ］-［Always Encrypted キー］を展開します。

3. ［列マスターキー］を右クリックし、［新しい列マスターキー］を選択します。

4. 「新しい列マスターキー」画面で以下の内容を入力します。
今回はキーストアに「Windows 証明書ストア - 現在のユーザー」を利用します。ほかに「Windows 証明書ストア - ローカル コンピューター」、「Azure Key Vault」、「キー記憶域プロバイダー（CNG）」が選択可能です。

項目	設定値
名前	CMK1
キーストア	Windows 証明書ストア – 現在のユーザー
エンクレーブ計算を許可する	チェック

5. 今回は［証明書の生成］ボタンをクリックします。なお、証明書が生成されるまで少しかかるため、一度のみボタンを押してしばらく待機してください。

6. 証明書が生成されたら、生成された証明書を選択して［OK］をクリックし、列マスターキーを作成します。指定したキーストアに既に登録済みの証明書を選択することも可能です。

上記の列マスターキー作成の GUI 操作は次の T-SQL 相当の処理になります。

```
Always_Encrypted_with_Secure_Enclaves.sql
USE ContosoHR;
GO
CREATE COLUMN MASTER KEY [CMK1] WITH (
  KEY_STORE_PROVIDER_NAME = N'MSSQL_CERTIFICATE_STORE',
  KEY_PATH = N'CurrentUser/My/C3D7CDDD73EB2804911C531D36FCA019559F468B',
```

```
  ENCLAVE_COMPUTATIONS (SIGNATURE = 0x740F47285CE2AB9EF80ADBEFBAA4F6F1E2052981FF0D5
E2AF25D755D7A5340DD9F372313F8CF1762849F0823339B6A08F413BF8A7A4F617A0D01EE48AEEB2593
00CEE9D5E9D619352D484A15F0C50147AF1E3C8F5D6123A876D0F1F2AEE1FEF7FF2C5AE3C29F7A9DD94
1AB1B75E2BE1932C8D31BCEF9672ABDF6A12FCCCCB9E4A7FBB3826C311C25818CFDB1A4B1464CC9547D
4B49A3073D8C8A18A235D3A6E4400999D86C007EDC16D6BE695B3726941AC01087A294E9BE50A44EFD2
CA64108B28276D6E63868B995371BBD2BE9D667B4F73E5F58080FE86829B07EB7B62E45CB88DA32C0C1
7CF5C7EC9CD2368EE71711EA988CA018CBEAD2C328773FE16987)
);
GO
```

ここでの要点は、列マスターキーの実体はデータベース内には保存されず、列マスターキーを格納するキーストアの種類と場所のみが保存される点です。SIGNATURE はキーストアの場所とセキュアエンクレーブが有効であることを示すデジタル署名です。

10）「列暗号化キー」を作成する　@SQL Server サーバー

1. 列暗号化キーを作成します。オブジェクトエクスプローラーで［列の暗号化キー］を右クリックし、［新しい列の暗号化キー］を選択します。

2. 「新しい列の暗号化キー」画面で以下の内容を入力して［OK］をクリックします。

項目	設定値
名前	CEK1
列マスターキー	CMK1（先ほど作成した列マスターキー）

上記の列暗号化キー作成の GUI 操作は以下の T-SQL 相当の処理になります。

Always_Encrypted_with_Secure_Enclaves.sql

```
USE ContosoHR;
GO
CREATE COLUMN ENCRYPTION KEY [CEK1] WITH VALUES (
  COLUMN_MASTER_KEY = [CMK1],
  ALGORITHM = 'RSA_OAEP',
  ENCRYPTED_VALUE = 0x016E000001630075007200720065006E007400750073006500720020F006D0
079002F006300330064003700630064006400640037003300650062003200380030003400390031003100
6300350033003100640033003600660063006100300031003900350035003900660034003600380062008
392BAC80338AB4D22B7AB1BF29CCB8D097C2C56EAA7BB9FEFA05D7E037FDB1F883A1088F58C0C1232AFD10
C224C7E0CAD5FBA0B42DF4B6D4781DB6A581D0E1E094AADA863AC9F472A116AD5C6770F57D50E3E0E408
A1B6CF61749B24BE2055740485931D6521EBFB9C2D4BC24BC2CA613C5EF3BC853E03A12B67DF0CE394A
739076309C7969B79D2E90F67AAE2774478EFDF6D899CCA710CE5F27287C388741EE247F53AD0D7964AC
FFAC6DB35863AF3C0B8D927922219FA0E3A58D30F0C4EB2277B635F6904C3062218BAACC18BDCD0E10A
8D31ED288093365104BD7F07C2E0DFE5C59FBD3FD3F2B7AF6DECFEC31FA1E2A8BFEE1F4A42B410F01D
1D982600E28036B8AC831D065C53C5E587B67C048BDC291DEDEBC32363CE50814D0E06942A1445C4C46
ADB18E7A22827721B52CB2335D2E5A0FA068E25C71FE7D64DEC6DFA52B223F6654F97077C3A801ABF5B
F5AF8A159583C20812537162A77D876518D08CB9F5AEB58879B5A9296C4946624012A2BEB2E55B732F
DAEDA12F6FCF8B4C479D5B01BEE48F372191A1689DD44124A10A739D55F52451314AB046AE55CAD313
981728EDCFD8CE27978B36E547B56885A3E11FD1336957DD6F9D62D8A782955EDA0A93D2A703AE37BC
97C91D3F71C1B1F66443B6721C2405D789DA5B043153E911761431B67CBF64D16AFA8FDDA9F51AD9FEA
C34035D3A0C8F5B5FBCC7EF8F9
);
GO
```

ここでの要点は、実際に列データの暗号化と復号に使用する列暗号化キーが、列マスターキーによって暗号化された ENCRYPTED_VALUE としてデータベースに保存される点です。暗号列の操作を必要とする際、クライアントは列マスターキーの場所と ENCRYPTED_VALUE をサーバーから入手し、入手した列マスターキーと ENCRYPTED_VALUE で列暗号化キーを復号します。

なお、作成した列暗号化キーやその付帯情報はシステムカタログを参照することで入手できます。

Always_Encrypted_with_Secure_Enclaves.sql

```
SELECT
  cek.name AS [CEK],
  cekv.encrypted_value AS [CMK で暗号化された CEK 値],
  cekv.encryption_algorithm_name AS [CEK 値の暗号化に使用されるアルゴリズム],
  cmk.key_store_provider_name AS [CMK ストアのプロバイダー],
  cmk.key_path AS [CMK ストアのパス],
  cmk.signature AS [Secure Enclaves 用の署名]
FROM sys.column_encryption_keys AS cek
LEFT JOIN sys.column_encryption_key_values AS cekv
  ON cek.column_encryption_key_id = cekv.column_encryption_key_id
LEFT JOIN sys.column_master_keys AS cmk
  ON cekv.column_master_key_id = cmk.column_master_key_id;
GO
```

Results　Messages

	CEK	CMKで暗号化されたCEK値	CEK値の暗号化に使用されるアルゴリズム	CMKストアのプロバイダー	CMKストアのパス	Secure Enclaves用の署名
1	CEK1	0x016E0000016300750…	RSA_OAEP	MSSQL_CERTIFICATE_STORE	CurrentUser/My/37…	0x1ED3D399A296B8C05021…

✱ クライアントのセットアップ

「サーバーのセットアップ」の手順で解説した通り、今回の検証ではキーストアに Windows 証明書ストアを利用しています。Windows 証明書ストアのようなローカルキーストアの場合はそのローカルキーストアを含むコンピューター上の SQL Server クライアントアプリケーションでのみ Always Encrypted および Secure Enclaves が使用できる構成です。つまり、ローカルキーストアの場合は SQL Server クライアントアプリケーションを実行するコンピューターごとに列マスターキーをインポートする必要があります。一方で、Azure Key Vault など集中型キーストアの場合は、列マスターキーをクライアントごとに展開する必要が無い点でキー管理が容易になります。

ここでは「サーバーのセットアップ」の手順でローカルキーストアに作成した列マスターキーを別のクライアントにインポートする手順を解説します。

作成した列マスターキーをエクスポートする

「サーバーをセットアップする」の「9)「列マスターキー」を作成する」手順を実行したキーストアとしたコンピューターから列マスターキーをエクスポートします。

1. キーストアのあるコンピューターに管理者としてサインインします。スタートメニューを右クリックして［ファイル名を指定して実行］をクリックし、「certmgr.msc」と入力して「OK」をクリックします。

2. 「証明書マネージャーツール」が起動するので、表示された画面で［個人］-［証明書］を選択します。

3. 列マスターキーの作成の際に生成した証明書を右クリックし、［すべてのタスク］-［エクスポート］をクリックします。

4. 「証明書のエクスポートウィザード」が表示されるので、［次へ］をクリックします。

5.　［はい、秘密キーをエクスポートします］を選択して、［次へ］をクリックします。

6.　既定の設定のまま、［次へ］をクリックします。

7. ［パスワード］にチェックを入れて、任意のパスワードを入力した後、［次へ］をクリックします。このパスワードは後ほど別のクライアントマシンに証明書をインポートする際に使用するため、控えておいてください。

8. 証明書の出力先を指定して［次へ］をクリックします。ここではデスクトップに「AlwaysEncryptedCert.pfx」というファイル名で出力するよう設定しています。

9. ［完了］をクリックします。

10. 「正しくエクスポートされました。」と表示されるので、「OK」をクリックして画面を閉じます。

11. 指定した場所に出力された pfx ファイルを、新しいクライアントにコピーします。

SQL Server クライアントを実行するコンピューターへマスターキーをインポートする

次に Always Encrypted および Secure Enclaves を利用するクライアントコンピューターへ、列マスターキーをインポートします。

1. 該当クライアントのコンピューターに管理者としてサインインし、コピーした pfx ファイルを右クリックして［PFX のインストール］をクリックします。

2. 「証明書のインポートウィザード」が表示されるので、［保存場所］は「サーバーをセットアップする」手順と同様に、［現在のユーザー］を選択して［次へ］をクリックします。

3. ［ファイル名］に選択した証明書が表示されていることを確認し、［次へ］をクリックします。

4. ［パスワード］に証明書のエクスポート時に指定したパスワードを入力し、［次へ］をクリックします。

5.　既定の設定のまま、［次へ］をクリックします。

6.　［完了］をクリックします。

7. 「正しくインポートされました。」と表示されるので、[OK] をクリックして画面を閉じます。

これで新しいクライアントコンピューターへの証明書のインポートは完了です。

✱ 新機能 “インプレース暗号化” の動作確認

Secure Enclaves により実現できるようになった「列のインプレース暗号化」の動作確認をします。

列をインプレース暗号化する

1. 列マスターキーをインポート済みのコンピューター上で SSMS を起動し、接続画面で接続情報を入力した後 [オプション] をクリックします。

2. [Always Encrypted] タブを開き、以下の内容を入力して [接続] をクリックします。<HGS IP address or DNS name>は HGS サーバーの IP アドレスまたは DNS 名に書き換えてください。

項目	設定値
Always Encrypted を有効にする（列の暗号化）	チェック
エンクレーブ構成証明 URL	http://<HGS IP address or DNS name>/Attestation

3. 暗号化前のデータを確認します。

```
Always_Encrypted_with_Secure_Enclaves.sql
USE ContosoHR;
GO
SELECT * FROM dbo.Employees;
GO
```

すべての列値が復号されプレーンテキストで表示されます。

	EmployeeID	SSN	FirstName	LastName	Salary
1	1	795-73-9838	Catherine	Abel	31692.00
2	2	990-00-6818	Kim	Abercrombie	55415.00

なお、ここで以下のような Always Encrypted クエリのパラメーター化を有効にすることの確認を求められた場合、「有効化」を選択します。

4. 今回は「Employee」テーブルの「SSN」列と「Salary」列の既存データをインプレースで暗号化します。

Always_Encrypted_with_Secure_Enclaves.sql

```
USE ContosoHR;
GO
-- SSN 列の既存データを暗号化
ALTER TABLE dbo.Employees ALTER COLUMN SSN char(11) ENCRYPTED WITH (
  COLUMN_ENCRYPTION_KEY = CEK1,
  ENCRYPTION_TYPE = Randomized,
  ALGORITHM = 'AEAD_AES_256_CBC_HMAC_SHA_256'
) NOT NULL WITH (ONLINE = ON);
GO
-- Salary 列の既存データを暗号化
ALTER TABLE dbo.Employees ALTER COLUMN Salary money ENCRYPTED WITH (
  COLUMN_ENCRYPTION_KEY = CEK1,
  ENCRYPTION_TYPE = Randomized,
  ALGORITHM = 'AEAD_AES_256_CBC_HMAC_SHA_256'
) NOT NULL WITH (ONLINE = ON);
GO
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
```

暗号化データを復号する

1. 列マスターキーをインポート済みのコンピューター上で SSMS を起動し、接続画面で接続情報を入力した後、［オプション］をクリックします。

2. ［Always Encrypted］タブを開き、［Always Encrypted を有効にする］にチェックを入れ、エンクレーブ構成証明の URL が入力済みであることを確認して［接続］をクリックします。

3. 暗号化データにアクセスします。

```
Always_Encrypted_with_Secure_Enclaves.sql
USE ContosoHR;
GO
SELECT * FROM dbo.Employees;
GO
```

先ほど暗号化した「SSN」列と「Salary」列のデータが復号され、プレーンテキストで表示されることが確認できました。

なお、列マスターキーをインポート済みのコンピューターであっても、［Always Encrypted を有効にする］のチェックを OFF にして暗号化データへアクセスした場合は、以下の図のようにデータは復号化されず、暗号化状態で表示されます。

結果 メッセージ

	EmployeeID	SSN	FirstName	LastName	Salary
1	1	0x01C590ABCB3B67171E1C8B7B14...	Catherine	Abel	0x01A376E1DE4F8695746236070378703...
2	2	0x01CFEE410DD9CFF61B04145020...	Kim	Abercrombie	0x014D030AAC94969A2A3383322E512E...

また、列マスターキーをインポートしていないコンピューターで、[Always Encrypted を有効にする] のチェックを ON にして暗号化データへアクセスした場合は、以下のエラーとなります。

```
メッセージ 0、レベル 11、状態 0
列 'SSN' を暗号化解除できませんでした。
メッセージ 0、レベル 11、状態 0
キー ストア プロバイダー 'MSSQL_CERTIFICATE_STORE' を使用して列暗号化キーを暗号化解除できませんでした。暗号化された列暗号化キーの最後の 10 バイトは 'XX-XX-XX-XX-XX-XX-XX-XX-XX-XX' です。
メッセージ 0、レベル 11、状態 0
拇印 'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX' 付きの証明書が、証明書の場所 'CurrentUser' の証明書ストア 'My' に見つかりませんでした。データベースにある列マスター キー定義の証明書パスが正しいことと、証明書が証明書の場所/ストアに正しくインポートされていることを確認してください。
パラメーター名:masterKeyPath
```

✱ 新機能 “高度な計算” の動作確認

Secure Enclaves によって新たに実現できるようになった「高度な計算」の動作確認をします。

高度な計算（比較検索と LIKE 検索）を行う

1. 列マスターキーをインポート済みのコンピューター上で SSMS を起動し、接続画面で接続情報を入力した後、[オプション] をクリックします。

2. [Always Encrypted] タブを開き、以下の内容を入力して [接続] をクリックします。<HGS IP address or DNS name>は HGS サーバーの IP アドレスまたは DNS 名に書き換えてください。

項目	設定値
Always Encrypted を有効にする（列の暗号化）	チェック
エンクレーブ構成証明 URL	http://<HGS IP address or DNS name>/Attestation

3. 前の手順で暗号化した「SSN」列と「Salary」列に対して高度な計算（比較検索と LIKE 検索）を実行します。

Always_Encrypted_with_Secure_Enclaves.sql

```
USE ContosoHR;
GO
DECLARE @SSNPattern char(11) = '%6818';
DECLARE @MinSalary money = 1000;
SELECT * FROM dbo.Employees WHERE SSN LIKE @SSNPattern AND Salary >= @MinSalary;
GO
```

暗号化列された「SSN」列に対する LIKE 検索と、「Salary」列に対する比較検索の結果が正しく表示されます。

➢ Data Classification and auditing built-in

1. 概要

「データの検出と分類」機能が強化され、SQL Server 監査との統合や操作性が改善されました。これにより、機密性の高いデータの管理がこれまでよりもさらに容易になります。

2. 利用シーン

SSMS（バージョン 17.5 以降）の機能である「データの検出と分類」を使用することで、SQL Server 2008 以降のデータベースに対して機密性の高いデータの検出と分類を行うことができるようになりました。このような機能が実装された背景の一つは、企業や規制機関におけるプライバシーに対する傾向の高まりにあります。例えば 2018 年 5 月 25 日より EU（欧州連合）において施行された個人情報保護に関する法律「一般データ保護規制（GDPR）」への準拠に向けた取り組みです。GDPR に準拠するためには、個人データを扱うデータベースに対して適切な安全対策を講じるとともに、適切な維持管理を行っていく必要があります。そのため、SQL Server 上のデータベース内に格納されている機密性の高いデータを検出・分類し、これらのデータへのアクセス状況を把握することは、組織の情報保護の達成に重要になりますが、従来の「データの検出と分類」には大きく 2 つの課題がありました。

1. 分類情報などのメタデータは、拡張プロパティという拡張情報を管理するための汎用的な機能を利用しており、機密データの管理に最適化されたものではないため管理効率が悪い。
2. 監査機能とは統合されていない（例えば GDPR に求められる機密データへのアクセス監査のために機密データの分類と監査が統合されていると望ましい）。

こうした課題に対して SQL Server 2019 では「データの検出と分類」機能が強化され、機密データ管理のための新しいコマンド（ADD SENSITIVITY CLASSIFICATION、DROP SENSITIVITY CLASSIFICATION）やカタログビュー（sys.sensitivity_classifications）が追加され、また SQL Server 監査機能とも統合されました。

3. やってみよう

それでは「データの検出と分類」機能と「SQL Server 監査統合」の動作を試してみましょう。

✱ SSMS を使用して機密データを自動検出して分類する

まずは SSMS を使用した機密データの分類を行います。

GUI 操作で機密データの分類を行う

1. SSMS のオブジェクト エクスプローラーで［データベース］を展開し、［WideWorldImporters］を右クリックして［タスク］-［データ検出と分類］-［データの分類］をクリックします。

2. SSMS の右側に、以下の図のような画面が表示されたことを確認し、上部に表示されている「62 columns with classification recommendations (click to view)」のメッセージが表示されているエリアをクリックします。

「62 columns with classification recommendations (click to view)」のメッセージが表示されているエリアをクリックすると、データベースの分析結果に基づいて、画面下部に推奨される機密データの候補と、そのデータの情報タイプ（Information Type）や機密ラベルが（Sensitivity Label）リストされます。

例えば、People テーブルの FullName カラムは情報タイプ「名前」、機密ラベル「Confidential - GDPR」として取り扱うことが推奨として示されています。これはあくまでも「推奨」であり、実際にどう取り扱うかはユーザーが決定します。

3. 推奨事項のリストに表示された分類から、承諾する項目にチェックを入れて［Accept selected recommendations］ボタンをクリックすることで、分類として承諾されます。
今回は以下の 2 項目に対してチェックを入れて［Accept selected recommendations］ボタンをクリックします。［Information Type］および［Sensitivity Label］はプルダウンから任意に変更することもできますが、今回は推奨のまま承諾します。

Schema	Table	Column	Information Type	Sensitivity Label
Application	People	EmailAddress	Contact Info	Confidential
Application	People	FullName	Name	Confidential - GDPR

4. 分類の承諾後は、画面上部にあるメニューから［Save］をクリックし、情報タイプと機密ラベルをデータベース内に保存します。なお［Add Classification］によって独自分類を追加することもできます。

画面上部に「The classification changes have been updated successfully.」というメッセージが表示されれば、保存は正常に完了です。

設定した分類情報を参照する

1. 分類設定の保存完了後、画面上部にあるメニューから［View Report］をクリックします。

これにより、設定した分類の情報を以下のようなグラフィカルなレポートで参照することができます。

2. レポートの下部分に表示されている表の「Schema」列に表示されている「Application」の左側にある［+］をクリックします。

これにより、設定されている分類一覧をすべて確認することができます。

Schema	Table	Column	Information Type	Sensitivity Label
⊟ Application	People	EmailAddress	Contact Info	Confidential
	People	FullName	Name	Confidential - GDPR
	People_Archive	EmailAddress	Contact Info	Confidential
	People_Archive	FullName	Name	Confidential - GDPR

なお、実行結果の「Table」列に WideWorldImporters データベースには存在していない「People_Archive」というテーブルへの設定情報が表示されていますが、これは「People」テーブルに対してテンポラルテーブルが設定されているためです。テンポラルテーブルによって保持されている履歴データにも機密データが存在するため、テンポラルテーブルを設定しているテーブルの列に分類・ラベル付けすると、自動的にテンポラルテーブルの対応する列にも同様の設定が実施されます。

✱ コマンドを使用し手動で機密データを分類する

先の手順は SSMS による GUI 操作でしたが、ここではコマンドを使用して機密データを分類してみます。

クエリ操作で機密データの分類を行う

「Confidential - GDPR」ラベルと「Name」タイプを設定する新しい分類を追加します。

Data_Classification_and_auditing_built-in.sql

```
USE WideWorldImporters;
GO
-- Application スキーマの People テーブルに定義している LogonName 列に対して、
--「Confidential - GDPR」ラベルと「Name」タイプを設定する新しい分類を作成
ADD SENSITIVITY CLASSIFICATION TO Application.People.LogonName WITH (
  LABEL = 'Confidential - GDPR',
  LABEL_ID = '989ADC05-3F3F-0588-A635-F475B994915B',
  INFORMATION_TYPE = 'Name',
  INFORMATION_TYPE_ID = '57845286-7598-22F5-9659-15B24AEB125E'
);
GO
```

設定した分類情報を参照する

追加した分類をカタログビューで確認します。

Data_Classification_and_auditing_built-in.sql

```
USE WideWorldImporters;
GO
SELECT o.name AS TableName, c.name AS ColumnName, information_type,
       information_type_id, label, label_id
FROM sys.sensitivity_classifications AS sc
  LEFT OUTER JOIN sys.objects AS o ON sc.major_id = o.object_id
  LEFT OUTER JOIN sys.columns AS c
    ON o.object_id = c.object_id AND sc.minor_id = c.column_id;
GO
```

先の SSMS での GUI 操作による分類や、テンポラルテーブルへの自動分類と合わせて、計 6 件の分類が確認できます。

Results　Messages

	TableName	ColumnName	information_type	information_type_id	label	label_id
1	People	FullName	Name	57845286-7598-22f5-9659-15b2…	Confidential - GDPR	989adc05-3f3f-0588-a635-f475…
2	People	LogonName	Name	57845286-7598-22F5-9659-15B2…	Confidential - GDPR	989ADC05-3F3F-0588-A635-F475…
3	People	EmailAddress	Contact Info	5c503e21-22c6-81fa-620b-f369…	Confidential	331f0b13-76b5-2f1b-a77b-def5…
4	People_Archive	FullName	Name	57845286-7598-22f5-9659-15b2…	Confidential - GDPR	989adc05-3f3f-0588-a635-f475…
5	People_Archive	LogonName	Name	57845286-7598-22F5-9659-15B2…	Confidential - GDPR	989ADC05-3F3F-0588-A635-F475…
6	People_Archive	EmailAddress	Contact Info	5c503e21-22c6-81fa-620b-f369…	Confidential	331f0b13-76b5-2f1b-a77b-def5…

✱ 機密データへのアクセスを監査する

これまでのデータの分類設定をもとに機密データへのアクセスを監査します。監査の設定は SSMS を用いた GUI 操作、Transact-SQL を用いたクエリ操作のいずれでも実施することができますが、ここでは Transact-SQL を使用してクエリから監査の作成を行う方法を実践します。

監査の設定を行う

1. 機密データへのアクセスを監査するための「監査」と「データベース監査の仕様」を定義します。

Data_Classification_and_auditing_built-in.sql

```
USE master;
GO
-- 新しい「監査」を作成 (FILEPATH には任意の場所を指定可能)
CREATE SERVER AUDIT Sensitivity_Audit TO FILE (
  FILEPATH = 'C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLSERVER¥MSSQL¥DATA'
);
GO
-- 上記で作成した「監査」を有効化
ALTER SERVER AUDIT Sensitivity_Audit WITH (STATE = ON);
GO

USE WideWorldImporters;
GO
-- Application スキーマの People テーブルに対する SELECT アクセスを監査するための
-- 「データベース監査の仕様」を作成
CREATE DATABASE AUDIT SPECIFICATION Sensitivity_Audit_Specification
  FOR SERVER AUDIT Sensitivity_Audit ADD (SELECT ON Application.People BY public)
  WITH (STATE = ON);
GO
```

2. 続けて、作成した監査が登録されていることを確認します。

Data_Classification_and_auditing_built-in.sql

```
USE master;
GO
SELECT * FROM sys.server_audits;
GO
```

Results　Messages

	audit_id	name	audit_guid	create_date	modify_date	prin
1	65536	Sensitivity_Audit	933ae2b0-0c71-4884-9ad4-107e…	2020-02-29 21:31:05.627	2020-02-29 21:31:05.627	1

3. 続けて、データベース監査の仕様が登録されていることを確認します。

Data_Classification_and_auditing_built-in.sql

```
USE WideWorldImporters;
GO
SELECT * FROM sys.database_audit_specifications;
GO
```

Results　Messages

	database_specification_id	name	create_date	modify_date	audit_
1	65536	Sensitivity_Audit_Specification	2020-02-29 21:31:05.637	2020-02-29 21:31:05.637	933ae

機密データへのアクセス監査ログを確認する

1. 監査をテストするため、Application スキーマの People テーブルを SELECT します。

Data_Classification_and_auditing_built-in.sql

```
USE WideWorldImporters;
GO
SELECT * FROM Application.People;
GO
```

2. sys.fn_get_audit_file で監査ログを一覧します。実行結果の「data_sensitivity_information」列のセルをクリックすると、新しいタブに XML で詳細が表示されます。

Data_Classification_and_auditing_built-in.sql

```
SELECT
  event_time, session_id, server_principal_name, database_name, object_name,
  CAST(data_sensitivity_information AS XML) AS data_sensitivity_information,
  client_ip, application_name
FROM sys.fn_get_audit_file ('C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLSER
VER¥MSSQL¥DATA¥*.sqlaudit', DEFAULT, DEFAULT);
GO
```

Results　Messages

	event_time	session_id	server_principal_name	database_name	object_name	data_sensitivity_information	client_ip	application_name
1	2020-02…	73	SERVER01\Administr…				local …	azdata-Query
2	2020-02…	73	SERVER01\Administr…	WideWorldI…	People	<sensitivity_attributes><…	local …	azdata-Query

上記の図に表示されている XML の全文は以下の通りです。（見やすいように改行しています。）

```
<sensitivity_attributes>
    <sensitivity_attribute
        label="Confidential - GDPR"
        label_id="989adc05-3f3f-0588-a635-f475b994915b"
        information_type="Name"
        information_type_id="57845286-7598-22f5-9659-15b24aeb125e" />
    <sensitivity_attribute
        label="Confidential"
        label_id="331f0b13-76b5-2f1b-a77b-def5a73c73c2"
        information_type="Contact Info"
        information_type_id="5c503e21-22c6-81fa-620b-f369b8ec38d1" />
</sensitivity_attributes>
```

People テーブルの３つの機密情報（「FullName」「EmailAddress」「LogonName」）へのアクセスを記録することができています。

環境をクリーンアップする

最後にこの検証で作成したオブジェクトを削除して環境をクリーンアップします。

Data_Classification_and_auditing_built-in.sql

```
USE WideWorldImporters;
GO
-- 分類を削除
DROP SENSITIVITY CLASSIFICATION FROM
  Application.People.FullName,
  Application.People.EmailAddress,
  Application.People.LogonName;
GO
-- 「データベース監査の仕様」の無効化
ALTER DATABASE AUDIT SPECIFICATION Sensitivity_Audit_Specification WITH (STATE = OFF);
GO
-- 「データベース監査の仕様」を削除
DROP DATABASE AUDIT SPECIFICATION Sensitivity_Audit_Specification;
GO

USE master;
GO
-- 「監査」の無効化
ALTER SERVER AUDIT Sensitivity_Audit WITH (STATE = OFF);
GO
-- 「監査」を削除
DROP SERVER AUDIT Sensitivity_Audit;
GO
```

➢ TDE scan suspend and resume

1. 概要

Transparent Data Encryption（TDE）の有効化に伴い、暗号化処理の中断と再開ができるようになりました。これにより暗号化処理のスケジューリングが制御できるようになり、TDE が導入しやすくなります。

2. 利用シーン

データベースには外部には漏らせない重要な情報が格納されていることがほとんどで、TDE による暗号化は重要です。またデータベースの運用開始当初に比較しセキュリティ要件が強化されたことで既に運用中のデータベースに対して TDE を有効化したいケースもあります。

一方、TDE を有効化すると暗号化対象となる全てのページを読み込み、暗号化し、書き戻すといった比較的高負荷な処理が内部的に行われるため、運用中のデータベースに対して TDE を有効化するには業務影響に留意してスケジュールを検討する必要がありました。さらに、開始した暗号化処理は中断ができないため、データベースの規模によっては業務時間外のメンテナンス時間内では処理を完了できず、一時的にサービスレベルの変更を検討せねばならないといった課題もありました。

こうした課題に対して SQL Server 2019 では暗号化処理の中断と再開ができるようになりました。これにより暗号化処理のスケジューリングが制御できるようになり、TDE が導入しやすくなります。中断と再開は暗号化処理に限らず、TDE 無効化にともなう復号処理や暗号キー更新に伴う再暗号化処理にも同じく適用されます。

3. やってみよう

それでは TDE による暗号化処理の中断と再開を試してみましょう。

＊ TDE 有効化のための事前準備

TDE に必要なオブジェクトを作成する

TDE 有効化するための事前準備をします。

TDE_scan_suspend_and_resume.sql

```
USE master;
GO
--データベースマスターキーを作成（すでに作成済みであれば不要）
IF NOT EXISTS
  (SELECT * from sys.symmetric_keys WHERE name = '##MS_DatabaseMasterKey##')
BEGIN
  CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'PsswOrd';
END;
GO
-- サーバー証明書を作成
CREATE CERTIFICATE MyServerCert WITH SUBJECT = 'My DEK Certificate';
GO
```

```
USE WideWorldImportersDW;
GO
-- 透過的な暗号化に使用する暗号化キーの作成
CREATE DATABASE ENCRYPTION KEY WITH ALGORITHM = AES_128
  ENCRYPTION BY SERVER CERTIFICATE MyServerCert;
GO
```

✱ TDE 有効化による暗号化処理の中断と再開

まずは TDE の暗号化処理を中断する操作を行ってみましょう。

暗号化処理を開始する

1. TDE による暗号化を開始します。

TDE_scan_suspend_and_resume.sql

```
USE WideWorldImportersDW;
GO
ALTER DATABASE WideWorldImportersDW SET ENCRYPTION ON;
GO
```

Messages

```
22:43:58    Started executing query at Line 1
            Commands completed successfully.
22:43:58    Started executing query at Line 3
            Commands completed successfully.
            Total execution time: 00:00:00.036
```

ALTER DATABASE コマンド自体はすぐに完了しますが、TDE の暗号化処理はバックグラウンドで動作します。

2. 暗号化処理の状態を sys.dm_database_encryption_keys システムビューで確認します。

TDE_scan_suspend_and_resume.sql

```
USE master;
GO
SELECT database_id, DB_NAME(database_id) AS database_name,
       encryption_state_desc, encryption_scan_state_desc
FROM sys.dm_database_encryption_keys;
GO
```

Results　Messages

	database_id	database_name	encryption_state_desc	encryption_scan_state_desc
1	2	tempdb	ENCRYPTED	COMPLETE
2	6	WideWorldImportersDW	ENCRYPTION_IN_PROGRESS	RUNNING

暗号化対象のデータベース「WideWorldImportersDW」の「encryption_state_desc」列の値が「ENCRYPTION_IN_PROGRESS」、「encryption_scan_state_desc」列の値が「RUNNING」となっている場合、暗号化処理が実行中であることを示します。

暗号化処理を中断する

1. 実行中の暗号化処理を中断します。

TDE_scan_suspend_and_resume.sql

```
USE WideWorldImportersDW;
GO
ALTER DATABASE WideWorldImportersDW SET ENCRYPTION SUSPEND;
GO
```

2. 暗号化処理の状態を確認します。

TDE_scan_suspend_and_resume.sql

```
USE master;
GO
SELECT database_id, DB_NAME(database_id) AS database_name,
       encryption_state_desc, encryption_scan_state_desc
FROM sys.dm_database_encryption_keys;
GO
```

Results　Messages

	database_id	database_name	encryption_state_desc	encryption_scan_state_desc
1	2	tempdb	ENCRYPTED	COMPLETE
2	6	WideWorldImportersDW	ENCRYPTION_IN_PROGRESS	SUSPENDED

暗号化対象のデータベース「WideWorldImportersDW」の「encryption_state_desc」列の値が「ENCRYPTION_IN_PROGRESS」、「encryption_scan_state_desc」列の値が「SUSPENDED」となっている場合、暗号化処理が中断状態であることを示します。

では引き続き、中断した処理を再開する操作を実施してみましょう。

暗号化処理を再開する

1. 中断した暗号化処理を再開します。

TDE_scan_suspend_and_resume.sql

```
USE WideWorldImportersDW;
GO
ALTER DATABASE WideWorldImportersDW SET ENCRYPTION RESUME;
GO
```

2. 暗号化処理完了の頃合いで暗号化処理の状態を確認します。

TDE_scan_suspend_and_resume.sql

```
USE master;
GO
SELECT database_id, DB_NAME(database_id) AS database_name,
       encryption_state_desc, encryption_scan_state_desc
FROM sys.dm_database_encryption_keys;
GO
```

Results　Messages

	database_id	database_name	encryption_state_desc	encryption_scan_state_desc
1	2	tempdb	ENCRYPTED	COMPLETE
2	6	WideWorldImportersDW	ENCRYPTED	COMPLETE

暗号化対象のデータベース「WideWorldImportersDW」の「encryption_state_desc」列の値が「ENCRYPTED」、「encryption_scan_state_desc」列の値が「COMPLETE」となっている場合、暗号化処理が完了していることを示します。

✱ TDE 無効化による復号処理

最後に、有効化した TDE を無効化し環境を元に戻します。

暗号化設定を解除する

1. TDE による暗号化を解除します。

TDE_scan_suspend_and_resume.sql

```
USE WideWorldImportersDW;
GO
ALTER DATABASE WideWorldImportersDW SET ENCRYPTION OFF;
GO
```

もし、TDE 有効化による暗号化処理が完了していない場合はエラーになります。無効化するには暗号化処理の完了を待つ必要があります。

```
Msg 33109, Level 16, State 2
暗号化、暗号化解除、またはキー変更のスキャンの実行中に、データベース暗号化を 無効化
 することはできません。
Msg 5069, Level 16, State 1
ALTER DATABASE ステートメントが失敗しました。
```

2. 暗号化解除の状況を確認します。

TDE_scan_suspend_and_resume.sql

```
USE master;
GO
SELECT database_id, DB_NAME(database_id) AS database_name,
       encryption_state_desc, encryption_scan_state_desc
FROM sys.dm_database_encryption_keys;
GO
```

Results　Messages

	database_id	database_name	encryption_state_desc	encryption_scan_state_desc
1	2	tempdb	ENCRYPTED	COMPLETE
2	6	WideWorldImportersDW	DECRYPTION_IN_PROGRESS	RUNNING

復号対象のデータベース「WideWorldImportersDW」の「encryption_state_desc」列の値が「DECRYPTION_IN_PROGRESS」、「encryption_scan_state_desc」列の値が「RUNNING」となっている場合、復号処理が実行中であることを示します。

3. 復号処理完了の頃合いであらためて復号処理の状態を確認します。

TDE_scan_suspend_and_resume.sql

```
USE master;
GO
SELECT database_id, DB_NAME(database_id) AS database_name,
       encryption_state_desc, encryption_scan_state_desc
FROM sys.dm_database_encryption_keys;
GO
```

Results　Messages

	database_id	database_name	encryption_state_desc	encryption_scan_state_desc
1	2	tempdb	ENCRYPTED	COMPLETE
2	6	WideWorldImportersDW	UNENCRYPTED	COMPLETE

復号対象のデータベース「WideWorldImportersDW」の「encryption_state_desc」列の値が「UNENCRYPTED」、「encryption_scan_state_desc」列の値が「COMPLETE」となっている場合、復号処理が完了していることを示します。

➢ Certificate management in SQL Server Configuration Manager

1. 概要

SQL Server 構成マネージャーの改善により、SQL Server との通信を暗号化するための証明書管理操作が簡素化されました。また Always On 構成など、複数の SQL Server インスタンスへの展開も容易になっており、僅かな操作でより短期間に証明書の管理や展開が可能となります。

2. 利用シーン

SQL Server との通信の暗号化で SSL/TLS を使用するには、SQL Server へ SSL/TLS 証明書を設定する必要があり、SQL Server 2017 以前のバージョンでは構成に応じて以下のような作業を行う必要がありました。

スタンドアロン構成	設定ツール
1．証明書をインストールする	Microsoft 管理コンソール
2．SQL Server で使用する証明書を選択する	SQL Server 構成マネージャー
Always On AG 構成	**設定ツール**
1．証明書をインストールする	Microsoft 管理コンソール
2．SQL Server で使用する証明書を選択する。	SQL Server 構成マネージャー
3．上記手順を各ノードで繰り返す。	–
Always On FCI 構成	**設定ツール**
1．証明書をインストールする	Microsoft 管理コンソール
2．以下のレジストリキーの値を変更する HKLM¥SOFTWARE¥Microsoft¥Microsoft SQL Server¥<instance>¥MSSQLServer¥SuperSocketNetLib¥Certificate	レジストリエディター
3．上記手順を各ノードで繰り返す。	–

SQL Server 2019 では SQL Server 構成マネージャーを使用した GUI 操作での証明書管理が強化され、以下の通り証明書管理に関わる操作が簡素化されます。

スタンドアロン構成	設定ツール
1．証明書の設定を行う	SQL Server 構成マネージャー
Always On AG 構成／FCI 構成	**設定ツール**
1．AG 構成の場合はプライマリノード、FCI 構成の場合はアクティブノードで証明書の設定を行う	SQL Server 構成マネージャー

3. やってみよう

それでは SQL Server 構成マネージャーを使用した証明書管理によって SSL/TLS 暗号化接続を構成してみましょう。

証明書の準備

お手元にテスト用の証明書が無い場合は、以下の PowerShell コマンドで自己証明書を作成してください。

TDE_scan_suspend_and_resume.sql

```
# 自己証明書の作成
$cert = New-SelfSignedCertificate `
            -Type SSLServerAuthentication `
            -Subject "SERVER01" `
            -DnsName "SERVER01" `
            -KeySpec "KeyExchange" `
            -CertStoreLocation "cert:¥LocalMachine¥My" `
            -KeyDescription "MyCert" `
            -KeyExportPolicy Exportable `
            -NotAfter (Get-Date).AddYears(30)

# 証明書パスワードを設定
$Password = ConvertTo-SecureString `
            -String "Passw0rd" `
            -Force `
            -AsPlainText

# 証明書のエクスポート
Export-PfxCertificate `
            -Cert $cert `
            -Password $Password `
            -FilePath "C:¥MyCert.pfx"

# クリーンアップ（この PC に登録された自己証明書の削除）
Remove-Item $cert.PSPath
```

上記コマンドの「SERVER01（証明書を作成するサーバー名）」、「Passw0rd（証明書のパスワード）」、「C:¥MyCert.pfx（証明書のエクスポート先）」は環境に合わせて読み替えてください。

証明書の設定

前の手順で作成した自己証明書を利用し証明書の設定を行います。

1. ［スタート］メニューから、［Microsoft SQL Server 2019］-［SQL Server 2019 構成マネージャー］を起動し、SQL Server 構成マネージャーの左ペインから［SQL Server ネットワークの構成］-［MSSQLSERVER のプロトコル］を右クリックし、［プロパティ］を選択します。

2. ［MSSQLSERVER のプロパティ］画面の［証明書］タブで［インポート］ボタンをクリックします。

3. ［証明書の選択］画面が表示されますので、証明書の種類として［PFX］を選択し、［参照］をクリックし後、「証明書の準備」で作成した証明書ファイル（C:¥MyCert.pfx）を指定して［次へ］をクリックします。

4. 「証明書の準備」で指定したパスワード（Passw0rd）を入力して［次へ］をクリックします。

5. 指定した証明書が自己証明書である場合は警告が表示されますが、今回はそのまま［次へ］をクリックします。

6. 以下の警告ダイアログが表示されますので、［OK］をクリックします。

7. 証明書のインポートが完了すると、以下のような画面が表示されますので、［完了］をクリックして終了します。

8. 「SQL Server 構成マネージャー」の「MSSQLSERVER のプロパティ」画面に戻るので［OK］をクリックしてプロパティ画面を閉じます。

9. 設定の変更を反映させるため、SQL Server サービスを再起動します。「SQL Server 構成マネージャー」の左ペインから［SQL Server のサービス］を選択し、右ペインに表示された一覧から［SQL Server（MSSQLSERVER)］を右クリックして［再起動］を選択します。

暗号化通信による接続

SSL/TLS による暗号化通信で SQL Server に接続してみましょう。

1. クライアント PC で SSMS を起動し、［サーバーへの接続］ダイアログで SQL Server への接続情報を入力後、［オプション］をクリックします。

2. ［暗号化接続］と［サーバー証明書を信頼する］にチェックを入れて、［接続］をクリックします。

3. この接続によりクライアントと SQL Server の通信は暗号化されます。ネットワークキャプチャ等を使用して通信をキャプチャすると、以下のように通信が暗号化されていることが確認できます。

証明書の解除

設定した証明書を解除する場合は以下の手順です。

1. ［スタート］メニューから、［Microsoft SQL Server 2019］-［SQL Server 2019 構成マネージャー］を起動し、「SQL Server 構成マネージャー」の左ペインから［SQL Server ネットワークの構成］-［MSSQLSERVER のプロトコル］を右クリックし、［プロパティ］を選択します。

2. ［MSSQLSERVER のプロパティ］画面の［証明書］タブで登録した証明書（SERVER01）が選択されていることを確認し［クリア］ボタンを押下します。

3. ［証明書］欄に表示されていた証明書が削除されたことを確認し［適用］をクリックします。

4. 以下の警告ダイアログが表示されますので［OK］をクリックします。

5. ［OK］をクリックして画面を閉じます。

6. 設定の変更を反映させるため、SQL Server サービスを再起動します。「SQL Server 構成マネージャー」の左ペインから［SQL Server のサービス］を選択し、右ペインに表示された一覧から［SQL Server（MSSQLSERVER)］を右クリックして［再起動］を選択します。

3. High availability

このセクションは SQL Server の可用性に関する機能強化を取り上げます。SQL Server 2019 では既存機能の強化に加え新機能「Accelerated Data Recovery」が導入されています。

- **Accelerated Data Recovery**

  Accelerated Database Recovery（ADR）というデータベースエンジンの新しい復旧プロセスを選択できるようになります。この新しい復旧プロセスでは、長時間トランザクションの影響を最小化するメカニズムによって「クラッシュリカバリの高速化」、「トランザクションロールバックの高速化」、「トランザクションログ肥大化の回避（積極的なログの切り捨て）」を実現しています。

- **Index Operation Enhancements**

  インデックス処理に関して 2 つの機能強化があります。

  1. 「インデックス作成と再構築のオンライン処理」は従来のインデックス（＝B ツリーインデックス）に加え非クラスター化列ストアインデックスに対応していましたが、SQL Server 2019 でクラスター化列ストアインデックスにも対応します。
  2. 「オンラインインデックス処理の再開と中断」はインデックス（＝B ツリーインデックス）の再構築処理に対応していましたが、SQL Server 2019 ではインデックス（＝B ツリーインデックス）の作成処理にも対応します。

- **Always On Availability Group Enhancements**

  Always On Availability Group に関して 3 つの機能強化があります。

  1. 同期レプリカの最大数が、従来の 3 台から 5 台（プライマリレプリカ 1 台、同期セカンダリレプリカ 4 台）に拡張されます。
  2. 「クラスターレス読み取りスケール可用性グループ」や「クロスプラットフォームクラスターによる分散型可用性グループ」の構成は高可用性を目的とした構成ではないため、自動フェールオーバーやプライマリレプリカへの透過接続は提供されません。そのため手動フェールオーバーを行う場合はそのたびにクライアントからの接続先をフェールオーバー先の新しいプライマリレプリカへ変更する必要がありました。この課題に対して SQL Server 2019 では Secondary-to-primary replica connection redirection 機能により、クライアントで指定された接続先レプリカに関係なく、接続を常にプライマリレプリカへリダイレクトすることができます。
  3. SQL Server 内部のタスクは、関連付けられたワーカーないしスケジューラーを移動することはないため、そのタスクの重みによってはスケジューラーに関連付けられた論理 CPU 間で一時的に負荷の不均一が生じるケースがありました。この課題に対して SQL Server 2019 は Scheduler worker migration のメカニズムを導入し、アイドル状態のスケジューラーへタスクを移動させて負荷の不均一を解消し、システム全体の CPU 利用効率を上げることでパフォーマ

ンス改善を図っています。執筆時点では「REDO 処理のパラレル化」が Scheduler worker migration に対応しています。

- **Database Snapshot Enhancements**

  In-Memory OLTP 機能によるインメモリ化されたテーブルに対する Database Snapshots がサポートされました。レポートのためのデータの履歴保持、オペレーションミスへの備え、テスト用途でのコピーなど、ある時点のインメモリ化されたテーブルデータのコピー（スナップショット）をコピー元の大きさに依存することなく高速に作成し、格納効率よく保持できるようになります。

➢ Accelerated Data Recovery

1. 概要

Accelerated Database Recovery（ADR）というデータベースエンジンの新しい復旧プロセスを選択できるようになります。この新しい復旧プロセスでは長時間トランザクションの影響を最小化するメカニズムによって「クラッシュリカバリの高速化」、「トランザクションリカバリの高速化」、「トランザクションログ肥大化の回避（積極的なログの切り捨て）」を実現しています。

2. 利用シーン

トランザクションログが肥大化すると、クラッシュリカバリの長期化やディスク領域の圧迫、最悪の場合は一切のトランザクションを実行できない状態になるなど、大きな問題を伴います。データベース管理者はこうした問題を避けるために、ログのバックアップによるログの切り捨てを欠かすことなく定期的に行ってきました。

しかし、従来の復旧プロセスの設計上の理由により、システム内に長時間トランザクションが存在する場合は、該当トランザクションが終了するまでログの切り捨てを行うことができず、ログが肥大化し続けるという問題に直面するケースがありました。

SQL Server 2019 ではこの長時間トランザクションの影響を最小化するための新しい復旧プロセスである「ADR」が導入されています。この新しい復旧プロセスは "長時間トランザクションが発生しているシステム" や、"クラッシュリカバリの所要時間をできる限り一定にしたいシステム" で特に有効です。

なお、後者の "クラッシュリカバリの所要時間をできる限り一定にしたいシステム" とは「クラッシュリカバリ後、長時間待たされることなくデータベースの利用を早期に再開することができるシステム」とも言えます。例えば、このデータベースを PaaS として提供しているケースなどでは、ユーザーの都合とは関係なく不定期にメンテナンスを入れたい場合があります。もし、膨大な作業量のユーザートランザクションの実行中にメンテナンスを発生させたとすると、従来の復旧プロセスではデータベースが再度利用できるまでに長時間を要すことになります。ユーザーからすると、たくさんの処理が強制キャンセルされただけではなく、さらに再開するまでに長い時間を待たされることになるため、非常に望ましくない状況といえます。実際、クラウド上の SQL Server である Azure SQL Database では、新しい復旧プロセスである ADR が常に有効化されています。

ここからは、「従来の復旧プロセス」と「新しい復旧プロセス」それぞれの詳細を解説します。

従来の復旧プロセス

ARIES（Algorithms for Recovery and Isolation Exploiting Semantics）というデータベースの伝統的な復旧手法をベースに構築されています。

❖ キーとなるコンポーネント

▶ ダーティページテーブル

ダーティページの一覧と、その付帯情報を示します。

▶ アクティブトランザクションテーブル

完了していないアクティブなトランザクションの一覧と、その付帯情報を示します。

❖ 復旧プロセス

▶ Phase 1 : Analysis

トランザクションログの最後のチェックポイント開始点 LSN（Last Checkpoint）を始点に、ログの終端（End of Log）までスキャンし、「ダーティページテーブル」と「アクティブトランザクションテーブル」を SQL Server が停止した時点の状態で構築します。

* LSN（Log Sequence Number）：
  データベースで発生した事象を記録したログレコードに割り当てられる一意な通し番号です。

▶ Phase 2 : Redo

アクティブトランザクションの中で最も古い開始点 LSN（Oldest uncommitted Tx）を始点に Redo を実行し、SQL Server が停止した時点にデータベースを戻します。Redo の始点は、ダーティページテーブル内の最も古いダーティページの LSN（Last Checkpoint から End of Log の間に位置づけられる Oldest Dirty page LSN）ではなく、アクティブトランザクションテーブル内の最も古い開始点をもつアクティブトランザクションの開始点 LSN（Oldest uncommitted Tx）になります。これはページの更新だけではなく、トランザクションに伴うロックを再現する必要があるためです。なお、SQL Server 2005 で導入された Fast Recovery 機能（次の Undo フェーズの完了を待たずに Redo フェーズの完了によってデータベースをオン

ラインにし、ユーザーリクエストを開始する機能）は、アクティブトランザクションのロックを再現することによって実現されています。

▶ Phase 3 : Undo

ログ終端（End of Log）を始点に、アクティブトランザクションテーブル内のトランザクションをひとつずつロールバックします。

ロールバック処理をイメージしやすいようにひとつ例を示します。次のサンプルはトランザクション内でテーブルのすべてのレコード（1000 行）を DELETE した後に、その DELETE をロールバックした際のトランザクションレコードを表示したものです。

Accelerated_Data_Recovery.sql

```
SET NOCOUNT ON;
GO

-- DB 作成と ADR 無効化（既定は無効）
USE master;
GO
DROP DATABASE IF EXISTS DemoDB;
GO
CREATE DATABASE DemoDB;
GO
--検証用に復旧モデルを「単純」に設定（トランザクションログを確認しやすくするため）
ALTER DATABASE DemoDB SET RECOVERY SIMPLE;
GO
ALTER DATABASE DemoDB SET ACCELERATED_DATABASE_RECOVERY = OFF;
GO

-- テーブル作成と 1000 行の INSERT
USE DemoDB;
GO
DROP TABLE IF EXISTS DemoTABLE;
GO
CREATE TABLE DemoTABLE (id int, contents char(100) not null);
GO
INSERT INTO DemoTABLE VALUES (1, 'Demo with ADR disabled')
GO 1000

--ログの切り捨て（トランザクションログを確認しやすくするため）
CHECKPOINT;
GO
-- 1000 行の DELETE の実行とロールバック
BEGIN TRANSACTION;
  DELETE FROM DemoTABLE;
ROLLBACK TRANSACTION;
GO

-- トランザクションログの確認
SELECT * FROM sys.fn_dblog(NULL, NULL);
GO

-- 後始末（DemoDB の削除）
USE master;
```

```
GO
DROP DATABASE DemoDB;
GO
```

Results　Messages

	Current LSN	Operation	Context	Transaction ID	LogBlockGeneration	Tag Bits
1058	00000027:00000308:006e	LOP_DELETE_ROWS	LCX_HEAP	0000:0000074c	0	0x0000
1059	00000027:00000308:006f	LOP_DELETE_ROWS	LCX_HEAP	0000:0000074c	0	0x0000
1060	00000027:00000308:0070	LOP_DELETE_ROWS	LCX_HEAP	0000:0000074c	0	0x0000
1061	00000027:00000308:0071	LOP_SET_FREE_SPACE	LCX_PFS	0000:00000000	0	0x0000
1062	00000027:00000308:0072	LOP_SET_FREE_SPACE	LCX_PFS	0000:00000000	0	0x0000
1063	00000027:00000308:0073	LOP_INSERT_ROWS	LCX_HEAP	0000:0000074c	0	0x0000
1064	00000027:00000308:0074	LOP_INSERT_ROWS	LCX_HEAP	0000:0000074c	0	0x0000
1065	00000027:00000308:0075	LOP_INSERT_ROWS	LCX_HEAP	0000:0000074c	0	0x0000
1066	00000027:00000308:0076	LOP_INSERT_ROWS	LCX_HEAP	0000:0000074c	0	0x0000

2 ⚠ 0　Ln 44, Col 3 (796 selected)　Spaces: 4　UTF-8　CRLF　SQL　MSSQL　2119 rows　00:00:01

上図のように、1000 行の DELETE（LOP_DELETE_ROWS）レコードが続いた後、ロールバックによってその逆の操作となる 1000 行の INSERT（LOP_INSERT_ROWS）が実行されていることが確認できます。

❖ 従来の復旧プロセスの設計上の留意点

長時間トランザクションが発生する場合、いくつかの課題がありました。

▶ クラッシュリカバリが長期化する

クラッシュリカバリの所要時間はトランザクションの作業量と存続期間に比例するため、長時間トランザクションの発生はクラッシュリカバリを長期化させます。クラッシュリカバリ中（最低限 Redo フェーズが完了するまで）はデータベースが利用できないため、ユーザーへのサービスの再開が大きく遅延します。

なお、クラッシュリカバリは、SQL Server が異常終了した場合の再開時に限らず、即時終了した場合の再開時にも実行されるリカバリ処理です。また、SQL Server の即時終了とは、ユーザーによる SQL Server サービスの通常停止に限らず、OS の通常シャットダウンの延長でも要求されるため、クラッシュリカバリは障害時に限らず日々の運用のなかでも起こり得るリカバリ処理です。

▶ トランザクションリカバリ（オンライントランザクションのロールバック）が長期化する

クラッシュリカバリと同様に、トランザクションリカバリについてもトランザクションの作業量と存続期間に比例するため、長時間トランザクションの発生はクラッシュリカバリを長期化させます。

▶ **ログが切り捨てできずファイルサイズが肥大化する**

完了していないトランザクションのログを切り捨てることはできないため、長時間トランザクションの発生は、トランザクションログファイルを肥大化させます。もしトランザクションログファイルが利用者の許容する最大サイズに達した場合は、一切のトランザクションを実行できない状態になります。

ADR の復旧プロセス

SQL Server 2019 では、従来の課題の解決を目指して長時間トランザクションの影響を最小化する新しい復旧プロセスが導入されています。

❖ キーとなるコンポーネント

▶ **Aborted Transaction Map（ATM）**

トランザクションの状態を管理するマップです。

▶ **Persisted Version Store (PVS)**

PVS には生成された行バージョンデータとして、変更前イメージと変更を与えたトランザクション ID が格納されます。PVS は行バージョンを保持する tempdb バージョンストアとは異なりユーザーデータベースへ永続的に格納されます。

データへのアクセスは ATM を利用し、PVS 内のバージョンチェーンを追跡します。具体的にはもし行のバージョンが ABORTED トランザクションに関連付けられている場合、該当バージョンを無視して追跡を継続することで有効なバージョンを辿っていきます。これによりクラッシュリカバリにおける Undo 処理やトランザクションリカバリにおけるロールバックは単にそのトランザクションを ABORTED としてマークするだけで良くなるため、処理を短時間で完了さ

せることができるようになります。

なお ADR を利用することの主たるトレードオフは「PVS へ記録していく行バージョンを生成することのオーバーヘッド」と「PVS を永続するための追加のストレージ領域が必要となること」の 2 点です。

▶ **sLog**

sLog はシステムによる非バージョン化操作（ロック取得やメタデータキャッシュ無効化、ページ割当と解除、ページ分割、統計情報更新など）のログレコードを格納するメモリ内のログストリームです。システムによる操作は長時化しうるユーザートランザクションとは異なり、短命で制御されたものであるためログのサイズは小さく、メモリ内で保持されます。ログのサイズが小さくメモリ内に配置されているため、Redo や Undo によるスキャンは迅速に完了します。なおメモリ内の sLog はチェックポイントによりディスクへ永続化され、復旧プロセスの中で再構築されます。

▶ **Logical Revert**

バージョンベースの復元を行レベルで実行するための非同期スレッドです。Logical Revert はアクティブなトランザクションを追跡し、順次 ABORTED としてマークするとともに、関連付いたすべてのロックを解放します。また有効なバージョンの行をデータページに戻します。この有効なバージョンに戻す操作は、従来のようにトランザクションログを物理的にスキャンして一度に 1 つずつ変更を元に戻すのではなく、PVS 内のコミットされたバージョンを取り出すことで迅速に操作されます。この操作の後 PVS 内の不要データはクリーンアップされます。

▶ **Cleaner**

定期的に起き上がる非同期スレッドで、Logical Revert による不要な行バージョンのクリーンアップを行います。

❖ 復旧プロセス

▶ **Phase 1: Analysis**

従来と同様の Analysis 処理に加えて、sLog を再構築します。

▶ **Phase 2: Redo**

まず Oldest uncommitted Tx を始点に Last Checkpoint まで sLog による Redo を実行します。メモリ内の sLog のスキャンと非バージョン化操作のみの Redo は迅速に完了します。次に Last Checkpoint を始点に End of Log までトランザクションログによる Redo を実行します。

▶ **Phase 3: Undo**

sLog による非バージョン化操作の Undo と Logical Revert による Undo を実行します。

非バージョン化操作の Undo は Redo と同様に迅速に完了します。Logical Revert による Undo はコミットされていないトランザクションを ABORTED としてマークするだけで済むため、これも迅速に完了します。

従来のロールバック処理との相違をイメージしやすいように、ADR を有効化した状態で先に解説したサンプルと同じ処理を流してみます。

Accelerated_Data_Recovery.sql

```
SET NOCOUNT ON;
GO

-- DB 作成と ADR 有効化
USE master;
GO
DROP DATABASE IF EXISTS DemoDB;
GO
CREATE DATABASE DemoDB;
GO
--検証用に復旧モデルを「単純」に設定（トランザクションログを確認しやすくするため）
ALTER DATABASE DemoDB SET RECOVERY SIMPLE;
GO
ALTER DATABASE DemoDB SET ACCELERATED_DATABASE_RECOVERY = ON;
GO

-- テーブル作成と 1000 行の INSERT
USE DemoDB;
GO
DROP TABLE IF EXISTS DemoTABLE;
GO
CREATE TABLE DemoTABLE (id int, contents char(100) not null);
GO
INSERT INTO DemoTABLE VALUES (1, 'Demo with ADR enabled');
GO 1000

--ログの切り捨て（トランザクションログを確認しやすくするため）
CHECKPOINT;
GO
-- テーブル作成と 1000 行の INSERT
BEGIN TRANSACTION;
  DELETE FROM DemoTABLE;
ROLLBACK TRANSACTION;
GO

-- トランザクションログの確認
SELECT * FROM sys.fn_dblog(NULL, NULL);
GO

-- 後始末（DemoDB の削除）
USE master;
GO
DROP DATABASE DemoDB;
GO
```

Results　Messages

	Current LSN	Operation	Context	Transaction ID	LogBlockGeneration	Tag Bits
1015	00000027:00000318:00f3	LOP_DELETE_ROWS	LCX_HEAP	0000:00000751	0	0x0000
1016	00000027:00000318:00f4	LOP_DELETE_ROWS	LCX_HEAP	0000:00000751	0	0x0000
1017	00000027:00000318:00f5	LOP_DELETE_ROWS	LCX_HEAP	0000:00000751	0	0x0000
1018	00000027:00000318:00f6	LOP_DELETE_ROWS	LCX_HEAP	0000:00000751	0	0x0000
1019	00000027:00000318:00f7	LOP_DELETE_ROWS	LCX_HEAP	0000:00000751	0	0x0000
1020	00000027:00000318:00f8	LOP_DELETE_ROWS	LCX_HEAP	0000:00000751	0	0x0000
1021	00000027:00000318:00f9	LOP_DELETE_ROWS	LCX_HEAP	0000:00000751	0	0x0000
1022	00000027:00000318:00fa	LOP_ABORT_XACT	LCX_CTR_ABORTED	0000:00000751	0	0x0000

⊗ 8 ⚠ 0　Ln 45, Col 1 (784 selected)　Spaces: 4　UTF-8　CRLF　SQL　MSSQL　1022 rows　00:00:01

上図のように、従来のロールバックとは異なり、1000 行の DELETE（LOP_DELETE_ROWS）レコード後の 1000 行の INSERT（LOP_INSERT_ROWS）は実行されておらず、トランザクションアボート（LOP_ABORT_XACT）のレコード 1 件のみが記録されていることが確認できます。

以上のように、ADR は長時間トランザクションの影響を最小化する新しいメカニズムにより「クラッシュリカバリの高速化」、「トランザクションリカバリの高速化」、「トランザクションログ肥大化の回避（積極的なログの切り捨て）」を実現することで、従来の復旧プロセスにおける課題を解決します。

3. やってみよう

次の 3 つの観点それぞれで、ADR が有効に機能することを確認します。

クラッシュリカバリの高速化	システム内のアクティブなトランザクションの作業量や存続期間に依存することなく、高速にクラッシュリカバリが実行されること
トランザクションリカバリの高速化	トランザクションの作業量や存続期間に依存することなく、瞬時にロールバックが実行されること
トランザクションログ肥大化の回避（積極的なログの切り捨て）	長時間トランザクションが存在する場合でもトランザクションログの切り捨てができること

＊ 事前準備

データベースの作成

ADR を無効化した「DB01」データベースと ADR を有効化した「DB02」データベースを作成します。

Accelerated_Data_Recovery.sql

```
USE master;
GO
-- 「DB01」を作成
CREATE DATABASE DB01;
```

```
GO
-- ADR の機能を OFF に設定
ALTER DATABASE DB01 SET ACCELERATED_DATABASE_RECOVERY = OFF;
GO
--  「DB02」を作成
CREATE DATABASE DB02;
GO
-- ADR の機能を ON に設定
ALTER DATABASE DB02 SET ACCELERATED_DATABASE_RECOVERY = ON;
GO
```

データの登録

1. 続けて「DB01」に「OrderHistoryADR」テーブルを作成し、1157060 件のレコードを投入します。

Accelerated_Data_Recovery.sql

```
SET NOCOUNT ON;
GO
USE DB01;
GO
--  「dbo.OrderHistoryADR」テーブルが存在する場合は削除
DROP TABLE IF EXISTS dbo.OrderHistoryADR;
GO
--  「WideWorldImportersDW」データベースの「Fact.Order」テーブルから
--  「dbo.OrderHistoryADR」テーブルを作成してデータをロード
SELECT [Order Key], [City Key], [Customer Key], [Stock Item Key],
       [Order Date Key], [Picked Date Key], [Salesperson Key], [Picker Key],
       [WWI Order ID], [WWI Backorder ID], Description, Package, Quantity,
       [Unit Price], [Tax Rate], [Total Excluding Tax], [Tax Amount],
       [Total Including Tax], [Lineage Key]
INTO dbo.OrderHistoryADR
FROM [WideWorldImportersDW].[Fact].[Order];
GO
-- 約 100 万件のデータを生成
INSERT dbo.OrderHistoryADR (
  [City Key], [Customer Key], [Stock Item Key], [Order Date Key],
  [Picked Date Key], [Salesperson Key], [Picker Key], [WWI Order ID],
  [WWI Backorder ID], Description, Package, Quantity, [Unit Price], [Tax Rate],
  [Total Excluding Tax], [Tax Amount], [Total Including Tax], [Lineage Key])
SELECT [City Key], [Customer Key], [Stock Item Key], [Order Date Key],
  [Picked Date Key], [Salesperson Key], [Picker Key], [WWI Order ID],
  [WWI Backorder ID], Description, Package, Quantity, [Unit Price], [Tax Rate],
  [Total Excluding Tax], [Tax Amount], [Total Including Tax], [Lineage Key]
FROM [WideWorldImportersDW].[Fact].[Order];
GO 4
--  「dbo.OrderHistoryADR」テーブルに主キーを作成
ALTER TABLE dbo.OrderHistoryADR ADD CONSTRAINT PK_Fact_OrderHistoryADR
PRIMARY KEY NONCLUSTERED ([Order Key] ASC, [Order Date Key] ASC)
WITH (DATA_COMPRESSION = PAGE);
GO
--  「dbo.OrderHistoryADR」テーブルの件数を確認
SELECT COUNT(*) FROM dbo.OrderHistoryADR;
GO
```

2. 同様に、「DB02」に「OrderHistoryADR」テーブルを作成し、1157060 件のレコードを投入します。

```
Accelerated_Data_Recovery.sql
SET NOCOUNT ON;
GO
USE DB02
GO
-- 「dbo.OrderHistoryADR」テーブルが存在する場合は削除
DROP TABLE IF EXISTS dbo.OrderHistoryADR;
GO
-- 「WideWorldImportersDW」データベースの「Fact.Order」テーブルから
-- 「dbo.OrderHistoryADR」テーブルを作成してデータをロード
SELECT [Order Key], [City Key], [Customer Key], [Stock Item Key],
       [Order Date Key], [Picked Date Key], [Salesperson Key], [Picker Key],
       [WWI Order ID], [WWI Backorder ID], Description, Package, Quantity,
       [Unit Price], [Tax Rate], [Total Excluding Tax], [Tax Amount],
       [Total Including Tax], [Lineage Key]
INTO dbo.OrderHistoryADR
FROM [WideWorldImportersDW].[Fact].[Order];
GO
-- 約 100 万件のデータを生成
INSERT dbo.OrderHistoryADR (
  [City Key], [Customer Key], [Stock Item Key], [Order Date Key],
  [Picked Date Key], [Salesperson Key], [Picker Key],
  [WWI Order ID], [WWI Backorder ID], Description, Package, Quantity,
  [Unit Price], [Tax Rate], [Total Excluding Tax], [Tax Amount],
  [Total Including Tax], [Lineage Key])
SELECT [City Key], [Customer Key], [Stock Item Key], [Order Date Key],
  [Picked Date Key], [Salesperson Key], [Picker Key], [WWI Order ID],
  [WWI Backorder ID], Description, Package, Quantity, [Unit Price],
  [Tax Rate], [Total Excluding Tax], [Tax Amount], [Total Including Tax],
  [Lineage Key]
FROM [WideWorldImportersDW].[Fact].[Order];
GO 4
-- 「dbo.OrderHistoryADR」テーブルに主キーを作成
ALTER TABLE dbo.OrderHistoryADR ADD CONSTRAINT PK_Fact_OrderHistoryADR
PRIMARY KEY NONCLUSTERED ([Order Key] ASC, [Order Date Key] ASC)
WITH (DATA_COMPRESSION = PAGE);
GO
-- 「dbo.OrderHistoryADR」テーブルの件数を確認
SELECT COUNT(*) FROM dbo.OrderHistoryADR;
GO
```

✱ クラッシュリカバリの高速化

長時間トランザクション実行中の障害を疑似的に発生させるため、単一のトランザクション内で大量のデータ更新を行ったのち、該当トランザクションを完了させずにインスタンスを強制終了させます。これを DB01（ADR 無効）と DB02（ADR 有効）のそれぞれのデータベースに対して実行し、次回の SQL Server 起動時に処理されるクラッシュリカバリの所要時間を DB01 と DB02 で比較してみます。

トランザクションを実行する

1. 「DB01」でトランザクションを開始します。約 100 万件の DELETE と約 100 万件の INSERT を実行したのち、トランザクションをコミットせずに実行中のまま放置します。

Accelerated_Data_Recovery.sql

```
SET NOCOUNT ON;
GO
USE DB01;
GO
-- トランザクションを開始
BEGIN TRANSACTION;

-- 「dbo.OrderHistoryADR」テーブルを全件削除
DELETE FROM dbo.OrderHistoryADR;

-- 「dbo.OrderHistoryADR」テーブルに約 100 万件登録
INSERT dbo.OrderHistoryADR (
  [City Key], [Customer Key], [Stock Item Key],
  [Order Date Key], [Picked Date Key], [Salesperson Key], [Picker Key],
  [WWI Order ID], [WWI Backorder ID], Description, Package, Quantity,
  [Unit Price], [Tax Rate], [Total Excluding Tax], [Tax Amount],
  [Total Including Tax], [Lineage Key])
SELECT [City Key], [Customer Key], [Stock Item Key], [Order Date Key],
  [Picked Date Key], [Salesperson Key], [Picker Key], [WWI Order ID],
  [WWI Backorder ID], Description, Package, Quantity, [Unit Price],
  [Tax Rate], [Total Excluding Tax], [Tax Amount], [Total Including Tax],
  [Lineage Key]
FROM [WideWorldImportersDW].[Fact].[Order];
GO 5
-- トランザクションを完了させず放置
```

2. 上記とは別の新しいクエリシートを表示して、今度は「DB02」でトランザクションを開始します。先ほどと同様に約 100 万件の DELETE と約 100 万件の INSERT を実行したのち、トランザクションをコミットせず実行中のまま放置します。

Accelerated_Data_Recovery.sql

```
SET NOCOUNT ON;
GO
USE DB02;
GO
-- トランザクションを開始
BEGIN TRANSACTION;

-- 「dbo.OrderHistoryADR」テーブルを全件削除
DELETE FROM dbo.OrderHistoryADR;

-- 「dbo.OrderHistoryADR」テーブルに約 100 万件登録
INSERT dbo.OrderHistoryADR([City Key], [Customer Key], [Stock Item Key],
  [Order Date Key], [Picked Date Key], [Salesperson Key], [Picker Key],
  [WWI Order ID], [WWI Backorder ID], Description, Package, Quantity,
  [Unit Price], [Tax Rate], [Total Excluding Tax], [Tax Amount],
  [Total Including Tax], [Lineage Key])
SELECT [City Key], [Customer Key], [Stock Item Key], [Order Date Key],
  [Picked Date Key], [Salesperson Key], [Picker Key], [WWI Order ID],
```

```
  [WWI Backorder ID], Description, Package, Quantity, [Unit Price],
  [Tax Rate], [Total Excluding Tax], [Tax Amount], [Total Including Tax],
  [Lineage Key]
FROM [WideWorldImportersDW].[Fact].[Order];
GO 5
-- トランザクションを完了させず放置
```

SQL Server を強制終了する

トランザクションを実行した 2 つとは別の新しいクエリシートを表示し、下記のスクリプトを実行してインスタンスを強制終了します。

Accelerated_Data_Recovery.sql

```
-- インスタンスを強制終了
SHUTDOWN WITH NOWAIT;
```

SQL Server を起動する

1. 強制終了したインスタンスを再起動するため、[スタート] メニューから、[Microsoft SQL Server 2019] - [SQL Server 2019 構成マネージャー] を選択します。

2. [Sql Server Configuration Manager] 画面が表示されるので、左カラムから [SQL Server のサービス] をクリックします。

3. 右ペインに表示されたサービスの一覧から、［SQL Server (MSSQLSERVER)］を右クリックし、［開始］を選択します。

4. 正常に SQL Server サービスが開始できたことを確認します。

クラッシュリカバリの所要時間を確認する

SQL Server エラーログ（ERRORLOG）を確認し、以下のメッセージからクラッシュリカバリの所要時間を確認します。

```
Recovery completed for database <データベース名> … in <復旧時間> second(s) …
```

ADR 無効の DB01 の復旧には時間がかかるため、上記のメッセージが出力されていない場合はしばらく待ってからあらためて確認してください。
なお、ERRORLOG の場所が分からない場合は以下のクエリで確認してください。

Accelerated_Data_Recovery.sql

```
USE master;
GO
-- ERRORLOG の出力場所を確認
SELECT SERVERPROPERTY('ErrorLogFileName') AS ERRORLOG_PATH;
GO
```

Results　Messages

	ERRORLOG_PATH
1	C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\MSSQL\Log\ERRORLOG

以下に、今回の検証における ERRORLOG の出力内容を抜粋します。

DB01（ADR 無効）

```
Recovery completed for database DB01 (database ID 7) in 324 second(s) (analysis 11512 ms, redo 19927 ms, undo 292210 ms [system undo 0 ms, regular undo 292094 ms].)
```

DB02（ADR 有効）

```
Recovery completed for database DB02 (database ID 8) in 10 second(s) (analysis 3865 ms, redo 5932 ms, undo 48 ms [system undo 0 ms, regular undo 0 ms].)
```

上記出力内容から、「DB01（ADR 無効）」は 324 秒を要しているのに対して、「DB02（ADR 有効）」は 10 秒で完了しており、クラッシュリカバリの高速化による効果が確認できます。

	DB01（ADR 無効）の クラッシュリカバリ所要時間	DB02（ADR 有効）の クラッシュリカバリ所要時間
処理時間（秒）	324	10

＊ トランザクションリカバリ（オンライントランザクションのロールバック）の高速化

先と同じく単一のトランザクション内で大量のデータ更新を行ったのち、該当トランザクションをロールバックします。これを「DB01（ADR 無効）」と「DB02（ADR 有効）」のそれぞれのデータベースに対して実行し、ロールバックの所要時間を DB01 と DB02 で比較してみます。

DB01 のロールバック所要時間を確認する

「DB01」でトランザクションを開始します。約 100 万件の DELETE と約 100 万件の INSERT を実行したのちトランザクションをロールバックさせ、その所要時間を確認します。

Accelerated_Data_Recovery.sql

```
SET NOCOUNT ON;
GO
USE DB01;
GO
-- トランザクションを開始
BEGIN TRANSACTION;

-- 「dbo.OrderHistoryADR」テーブルを全件削除
DELETE FROM dbo.OrderHistoryADR;

-- 「dbo.OrderHistoryADR」テーブルに約 100 万件登録
INSERT dbo.OrderHistoryADR (
  [City Key], [Customer Key], [Stock Item Key],
  [Order Date Key], [Picked Date Key], [Salesperson Key], [Picker Key],
  [WWI Order ID], [WWI Backorder ID], Description, Package, Quantity,
  [Unit Price], [Tax Rate], [Total Excluding Tax], [Tax Amount],
  [Total Including Tax], [Lineage Key])
SELECT [City Key], [Customer Key], [Stock Item Key], [Order Date Key],
  [Picked Date Key], [Salesperson Key], [Picker Key], [WWI Order ID],
  [WWI Backorder ID], Description, Package, Quantity, [Unit Price],
  [Tax Rate], [Total Excluding Tax], [Tax Amount], [Total Including Tax],
  [Lineage Key]
FROM [WideWorldImportersDW].[Fact].[Order];
GO 5
-- ROLLBACK の処理時間を確認するためステートメントの処理時間の統計を ON に設定
SET STATISTICS TIME ON;
-- トランザクションを ROLLBACK
ROLLBACK TRANSACTION;
```

Messages

```
20:08:12        Started executing query at Line 25

                SQL Server 実行時間:
                、CPU 時間 = 23125 ミリ秒、経過時間 = 77100 ミリ秒。

                Total execution time: 00:02:25.696
```

実行結果より、「DB01（ADR 無効）」でのロールバック所要時間は以下の通りです。

	DB01（ADR 無効）の ロールバック所要時間
処理時間（ミリ秒）	77,100
CPU 時間（ミリ秒）	23,125

DB02 のロールバック所要時間を確認する

上記とは別の新しいクエリシートを表示して今度は「DB02」でトランザクションを開始します。先ほどと同様に約 100 万件の DELETE と約 100 万件の INSERT を実行したのちトランザクションをロールバックさせ、その所要時間を確認します。

Accelerated_Data_Recovery.sql

```
SET NOCOUNT ON;
GO
USE DB02;
GO
-- トランザクションを開始
BEGIN TRANSACTION;

-- 「dbo.OrderHistoryADR」テーブルを全件削除
DELETE FROM dbo.OrderHistoryADR;

-- 「dbo.OrderHistoryADR」テーブルに約 100 万件登録
INSERT dbo.OrderHistoryADR (
  [City Key], [Customer Key], [Stock Item Key],
  [Order Date Key], [Picked Date Key], [Salesperson Key], [Picker Key],
  [WWI Order ID], [WWI Backorder ID], Description, Package, Quantity,
  [Unit Price], [Tax Rate], [Total Excluding Tax], [Tax Amount],
  [Total Including Tax], [Lineage Key])
SELECT [City Key], [Customer Key], [Stock Item Key], [Order Date Key],
  [Picked Date Key], [Salesperson Key], [Picker Key], [WWI Order ID],
  [WWI Backorder ID], Description, Package, Quantity, [Unit Price],
  [Tax Rate], [Total Excluding Tax], [Tax Amount], [Total Including Tax],
  [Lineage Key]
FROM [WideWorldImportersDW].[Fact].[Order];
GO 5
-- ROLLBACK の処理時間を確認するためステートメントの処理時間の統計を ON に設定
SET STATISTICS TIME ON;
-- トランザクションを ROLLBACK
ROLLBACK TRANSACTION;
```

Messages

```
20:15:13        Started executing query at Line 25

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

                Total execution time: 00:00:58.793
```

「DB01（ADR 無効）」は復旧におよそ 77 秒を要しているのに対して「DB02（ADR 有効）」は 0 秒未満で完了しており、トランザクションリカバリ（オンライントランザクションのロールバック）の高速化による効果が確認できます。

	DB01（ADR 無効）の ロールバック所要時間	DB02（ADR 有効）の ロールバック所要時間
処理時間（ミリ秒）	77,100	0
CPU 時間（ミリ秒）	23,125	0

✱ トランザクションログ肥大化の回避（積極的なログの切り捨て）

アクティブなトランザクション（＝長時間トランザクション）が存在する場合のトランザクションログの切り捨てを「DB01（ADR 無効）」と「DB02（ADR 有効）」それぞれのデータベースに対して実行し、その結果を比較してみます。

DB01（ADR 無効）：アクティブトランザクションが存在する状態でログを切り捨てる

1. クエリシート（「クエリウィンドウ１」）を表示して、「DB01」の事前準備を行います。

Accelerated_Data_Recovery.sql

```
-- 以下はクエリウィンドウ１ で実行
USE DB01;
GO
-- チェックポイント発行
CHECKPOINT;
GO
-- 完全バックアップを取得するまで単純復旧モデル相当の動作となるため、
-- 完全バックアップを取得(ファイルは捨てる)
BACKUP DATABASE DB01 TO DISK = N'NUL';
GO
-- 検証前に一度トランザクションログを切り捨て
BACKUP LOG DB01 TO DISK = N'NUL';
GO
```

2. 上記とは別の新しいクエリシート（「クエリウィンドウ２」）を表示して「DB01」でトランザクションを開始します。単一のトランザクション内でデータ更新を行ったのち該当トランザクションを完了させずに放置します。

Accelerated_Data_Recovery.sql

```
-- 以下はクエリウィンドウ２で実行
SET NOCOUNT ON;
GO
USE DB01;
GO
-- トランザクションを開始
BEGIN TRANSACTION;
```

```
-- 約 100 万件 DELETE
DELETE FROM dbo.OrderHistoryADR;

-- 約 20 万件 INSERT
INSERT dbo.OrderHistoryADR (
  [City Key], [Customer Key], [Stock Item Key],
  [Order Date Key], [Picked Date Key], [Salesperson Key], [Picker Key],
  [WWI Order ID], [WWI Backorder ID], Description, Package, Quantity,
  [Unit Price], [Tax Rate], [Total Excluding Tax], [Tax Amount],
  [Total Including Tax], [Lineage Key])
SELECT [City Key], [Customer Key], [Stock Item Key], [Order Date Key],
  [Picked Date Key], [Salesperson Key], [Picker Key], [WWI Order ID],
  [WWI Backorder ID], Description, Package, Quantity, [Unit Price],
  [Tax Rate], [Total Excluding Tax], [Tax Amount], [Total Including Tax],
  [Lineage Key]
FROM [WideWorldImportersDW].[Fact].[Order];
GO
```

［Message］に「Total execution time」が出力されたことを確認後、クエリシートは閉じずに置いておき次の手順に進みます。

3. さらに別の新しいクエリシート（「クエリウィンドウ３」）を表示して「DB01」のログバックアップによるログの切り捨てを試みます。

Accelerated_Data_Recovery.sql

```
-- 以下はクエリウィンドウ 3 で実行
USE DB01;
GO
-- ログの切り捨て前のトランザクションログのサイズを確認
SELECT 'Before Truncate' AS memo, db_name(database_id) AS database_name,
       used_log_space_in_bytes, used_log_space_in_percent
FROM sys.dm_db_log_space_usage;
GO
-- チェックポイント発行
CHECKPOINT;
GO
-- トランザクションログを切り捨て
BACKUP LOG DB01 TO DISK = N'NUL';
GO
-- ログの切り捨て後のトランザクションログのサイズを確認
SELECT 'After Truncate' AS memo, db_name(database_id) AS database_name,
       used_log_space_in_bytes, used_log_space_in_percent
FROM sys.dm_db_log_space_usage;
GO
-- ログの切り捨てを妨げる原因を確認
SELECT name, log_reuse_wait_desc FROM sys.databases WHERE name = 'DB01'
```

```
GO
```

Results　Messages

	memo	database_name	used_log_space_in_bytes	used_log_space_in_percent
1	Before Truncate	DB01	1636712448	46.7894

	memo	database_name	used_log_space_in_bytes	used_log_space_in_percent
1	After Truncate	DB01	1636737024	46.7901

「used_log_space_in_bytes」で示されるトランザクションログのサイズが減少しておらず、ログの切り捨てができていない事がわかります。

Results　Messages

	name	log_reuse_wait_desc
1	DB01	ACTIVE_TRANSACTION

また、上図の「log_reuse_wait_desc」で示されるログの切り捨てを妨げる原因が「ACTIVE_TRANSACTION」であることから、アクティブトランザクションの存在が原因であることがわかります。今回のケースでは「クエリウィンドウ２」のアクティブトランザクションが犯人です。

DB01（ADR 無効）：アクティブトランザクションをコミット後にログを切り捨てる

1. 「クエリウィンドウ２」のトランザクションをコミットで完了させます。

Accelerated_Data_Recovery.sql

```
-- 以下はクエリウィンドウ２で実行
-- トランザクションをコミット
COMMIT;
GO
```

［Message］に「Total execution time」が出力されたことを確認後次の手順に進みます。

Messages

```
20:59:45    Started executing query at Line 1
            Commands completed successfully.
            Total execution time: 00:00:00.007
```

2. 「クエリウィンドウ３」であらためてログバックアップを実行し、ログの切り捨てを試みます。

Accelerated_Data_Recovery.sql

```
-- 以下はクエリウィンドウ 3 で実行
USE DB01;
GO
-- ログの切り捨て前のトランザクションログのサイズを確認
SELECT 'Before Truncate' AS memo, db_name(database_id) database_name,
       used_log_space_in_bytes, used_log_space_in_percent
FROM sys.dm_db_log_space_usage;
GO
-- チェックポイント発行
CHECKPOINT;
GO
-- トランザクションログを切り捨て
BACKUP LOG DB01 TO DISK = N'NUL';
GO
-- ログの切り捨て後のトランザクションログのサイズを確認
SELECT 'After Truncate' AS memo, db_name(database_id) database_name,
       used_log_space_in_bytes, used_log_space_in_percent
FROM sys.dm_db_log_space_usage;
GO
-- ログの切り捨てを妨げる原因を確認
SELECT name, log_reuse_wait_desc FROM sys.databases WHERE name = 'DB01'
GO
```

Results　Messages

	memo	database_name	used_log_space_in_bytes	used_log_space_in_percent
1	Before Truncate	DB01	587653120	16.79949

	memo	database_name	used_log_space_in_bytes	used_log_space_in_percent
1	After Truncate	DB01	13975552	0.3995251

「used_log_space_in_bytes」で示されるトランザクションログのサイズが大きく減少しており、ログの切り捨てができた事がわかります。

Results　Messages

	name	log_reuse_wait_desc
1	DB01	NOTHING

また、上図の「log_reuse_wait_desc」で示されるログの切り捨てを妨げる原因が「NOTHING」であることから、ログの切り捨てに関する妨げが存在しなくなったことがわかります。

DB02（ADR 有効）：アクティブトランザクションが存在する状態でログを切り捨てる

1. 次に「DB02（ADR 有効）」で動作を確認します。まず新しいクエリシート（「クエリウィンドウ４」）を表示して事前準備を行います。

Accelerated_Data_Recovery.sql

```
-- 以下はクエリウィンドウ 4 で実行
USE DB02;
GO
-- チェックポイント発行
CHECKPOINT;
GO
-- 完全バックアップを取得するまで単純復旧モデル相当の動作となるため、
-- 完全バックアップを取得(ファイルは捨てる)
BACKUP DATABASE DB02 TO DISK = N'NUL';
GO
-- 検証前に一度トランザクションログを切り捨て
BACKUP LOG DB02 TO DISK = N'NUL';
GO
```

2. 上記とは別の新しいクエリシート（「クエリウィンドウ５」）を表示して「DB02」でトランザクションを開始します。単一のトランザクション内でデータ更新を行ったのち該当トランザクションを完了させずに放置します。

Accelerated_Data_Recovery.sql

```
-- 以下はクエリウィンドウ 5 で実行
SET NOCOUNT ON;
GO
USE DB02;
GO
-- トランザクションを開始
BEGIN TRANSACTION;

-- 約 100 万件 DELETE
DELETE FROM dbo.OrderHistoryADR;

-- 約 20 万件 INSERT
INSERT dbo.OrderHistoryADR (
  [City Key], [Customer Key], [Stock Item Key],
  [Order Date Key], [Picked Date Key], [Salesperson Key], [Picker Key],
  [WWI Order ID], [WWI Backorder ID], Description, Package, Quantity,
  [Unit Price], [Tax Rate], [Total Excluding Tax], [Tax Amount],
  [Total Including Tax], [Lineage Key])
SELECT [City Key], [Customer Key], [Stock Item Key], [Order Date Key],
  [Picked Date Key], [Salesperson Key], [Picker Key], [WWI Order ID],
  [WWI Backorder ID], Description, Package, Quantity, [Unit Price],
  [Tax Rate], [Total Excluding Tax], [Tax Amount], [Total Including Tax],
  [Lineage Key]
FROM [WideWorldImportersDW].[Fact].[Order];
GO
```

［Message］に「Total execution time」が出力されたことを確認後クエリシートは閉じずに置いておき、次の手順に進みます。

```
Messages

21:44:41        Started executing query at Line 6
                Commands completed successfully.
                Total execution time: 00:00:50.119
```

3. さらに別の新しいクエリシート（「クエリウィンドウ６」）を表示して「DB02」のログバックアップによるログの切り捨てを試みます。

Accelerated_Data_Recovery.sql

```sql
-- 以下はクエリウィンドウ６で実行
USE DB02;
GO
-- ログの切り捨て前のトランザクションログのサイズを確認
SELECT 'Before Truncate' AS memo, db_name(database_id) AS database_name,
       used_log_space_in_bytes, used_log_space_in_percent
FROM sys.dm_db_log_space_usage;
GO
-- チェックポイント発行
CHECKPOINT;
GO
-- トランザクションログを切り捨て
BACKUP LOG DB02 TO DISK = N'NUL';
GO
-- ログの切り捨て後のトランザクションログのサイズを確認
SELECT 'After Truncate' AS memo, db_name(database_id) AS database_name,
       used_log_space_in_bytes, used_log_space_in_percent
FROM sys.dm_db_log_space_usage;
GO
-- ログの切り捨てを妨げる原因を確認
SELECT name, log_reuse_wait_desc FROM sys.databases WHERE name = 'DB02'
GO
```

Results　Messages

	memo	database_name	used_log_space_in_bytes	used_log_space_in_percent
1	Before Truncate	DB02	1111416832	96.7096

	memo	database_name	used_log_space_in_bytes	used_log_space_in_percent
1	After Truncate	DB02	306941952	26.70846

「used_log_space_in_bytes」で示されるトランザクションログのサイズが大きく減少しており、「クエリウィンドウ５」でのアクティブトランザクションが存在している状態でもログの切り捨てができた事がわかります。

Results　Messages

	name	log_reuse_wait_desc
1	DB02	NOTHING

また、上図の「log_reuse_wait_desc」で示されるログの切り捨てを妨げる原因が「NOTHING」であることから、アクティブトランザクションの存在がログの切り捨ての妨げになっていないことがわかります。

➢ Index Operation Enhancements

1. 概要

インデックス処理に関して 2 つの機能強化があります。

1. 「インデックス作成と再構築のオンライン処理」は従来のインデックス（＝B ツリーインデックス）に加え非クラスター化列ストアインデックスに対応していましたが、SQL Server 2019 でクラスター化列ストアインデックスにも対応します。
2. 「オンラインインデックス処理の再開と中断」はインデックス（＝B ツリーインデックス）の再構築処理に対応していましたが、SQL Server 2019 ではインデックス（＝B ツリーインデックス）の作成処理にも対応します。

2. 利用シーン

SQL Server 2019 での強化ポイントをそれぞれ解説します。

- **クラスター化列ストアインデックスの作成と再構築のオンライン実行（＝同時実行）の強化**

  SQL Server 2017 以前の「インデックス作成と再構築のオンライン処理」は従来のインデックス（＝B ツリーインデックス）に加え非クラスター化列ストアインデックスに対応していましたが、SQL Server 2019 でクラスター化列ストアインデックスにも対応します。オンライン処理はシステムのメンテナンスウィンドウに制約のあるケースや、常時稼動が求められるサービス環境で重要な機能です。非オンライン処理の場合はサービス利用ユーザーによるデータアクセスは待機させられます。これは基になるデータや関連付けられたインデックスに他のロックとの互換性の低いロック（SCH-M）を必要とするためです（非クラスター化インデックスの作成については SCH-M を獲得しないため、例えば別セッションからの SELECT は同時実行が可能です）。一方オンライン処理の場合はサービス利用ユーザーによるデータ変更は待機させられることなく同時実行が許可されます。オンライン処理のトレードオフは空きディスク容量をより多く必要とする点と、追加の制御に伴いインデックス構築処理時間が伸びる可能性がある点です。

- **オンラインインデックス作成の中断・再開**

  オンラインインデックス（＝B ツリーインデックス）再構築の中断・再開が SQL Server 2017 で導入され、この中断・再開が SQL Server 2019 ではインデックス（＝B ツリーインデックス）初期構築（作成）にも対応します。システムの開発後の運用に入ってからも新たな要件に対応するために追加のインデックス作成を行うケースは多々あり、またクエリ性能を維持するための定期的なインデックス再構築はとても重要な運用タスクです。一方でインデックス構築はシステム負荷を要すため、その必要性とサービス影響とのトレードオフを十分に検討する必要がありました。この中断・再開によりスケジューリングの柔軟性が向上したことで、従来と比較してより積極的にインデックス構築を行いやすくなります。なお、「中断」とは、ユーザーによる明示的な中断に限らず、エラー（ディスク不足

によるエラーやインスタンス障害など）にも対応するため、耐障害性が向上している点にも注目です。

これまでの解説を下記にマトリクスで整理します。

オンライン インデックス処理	作成	作成の中断と再開	再構築	再構築の中断と再開
クラスター化インデックス	○	○ （2019～）	○	○ （2017～）
非クラスター化インデックス	○	○ （2019～）	○	○ （2017～）
クラスター化 列ストアインデックス	○ （2019～）	–	○ （2019～）	–
非クラスター化 列ストアインデックス	○ （2017～）	–	○ （2017～）	–

3. やってみよう

SQL Server 2019 で機能強化されたインデックス処理を試してみましょう。

✱ クラスター化列ストアインデックスをオンライン処理で作成

ここではオンライン処理でのクラスター化列ストアインデックス作成中に、別のセッションのクエリが待機せず同時実行が可能なことを確認します。

クラスター化列ストアインデックスをオンライン処理で作成する

1. オンライン処理で列ストアインデックスを作成します。
   CREATE CLUSTERED COLUMNSTORE INDEX コマンドのオプションとして、「WITH (ONLINE = ON)」を指定することにより、列ストアインデックスの作成がオンライン処理されます。

Index_Operation_Enhancements.sql

```
USE WideWorldImportersDW;
GO
CREATE CLUSTERED COLUMNSTORE INDEX CCX_Fact_OrderHistoryExtended
  ON Fact.OrderHistoryExtended
WITH (ONLINE = ON);
GO
```

なお、SQL Server 2017 以前のバージョンでは、クラスター化列ストアインデックスの作成時にこのオプションを指定すると以下のエラーとなります。

```
Msg 35318, Level 15, State 1
```

列ストア インデックスを作成するときに ONLINE オプションを指定することはできないため、ステートメントが失敗しました。ONLINE オプションを指定せずに列ストア インデックスを作成してください。

2. 別の新しいクエリシートを表示して列ストアインデックス作成中のテーブルにアクセスします。

Index_Operation_Enhancements.sql

```
USE WideWorldImportersDW;
GO
SELECT TOP(10) * FROM Fact.OrderHistoryExtended;
GO
```

クエリは待機することなく結果が返ってきます。

Results　Messages

	Order Key	City Key	Customer Key	Stock Item Key	Order Date Key	Picked Date Key	Salesper
1	2526883	88100	358	66	2013-12-27	2013-12-27	83
2	2526884	45084	373	217	2013-12-30	2013-12-30	83
3	2526885	45084	373	11	2013-12-30	2013-12-30	83
4	2526886	40273	296	145	2013-12-31	2013-12-31	83
5	2526887	56140	12	57	2013-12-31	2013-12-31	83
6	2526888	50767	5	42	2013-04-19	2013-04-19	62
7	2526889	51350	316	146	2013-04-19	2013-04-19	62
8	2526890	51350	316	66	2013-04-19	2013-04-19	62
9	2526891	64929	97	105	2013-04-19	2013-04-19	62
10	2526892	72294	193	133	2013-06-19	2013-06-19	62

非オンラインモードで実行した場合は、インデックス作成に伴い獲得されるロック（SCH-M ロック）と SELECT に伴い獲得されるロック（IS ロック）が競合するため、SELECT はインデックス作成が完了するまで待機させられます。

以上の動きは再構築についても同様です。

Index_Operation_Enhancements.sql

```
USE WideWorldImportersDW;
GO
ALTER INDEX CCX_Fact_OrderHistoryExtended ON Fact.OrderHistoryExtended REBUILD
WITH (ONLINE  = ON);
GO
```

＊ オンラインインデックス作成の中断と再開

ここではオンラインインデックス作成を擬似的なエラーで自動中断させ、それが正常に再開することを確認してみましょう。

オンラインインデックスの作成を実行する

1. インデックスの作成を疑似的なエラーで中断させるため、今回は WideWorldImportersDW データベースのデータファイルのサイズを縮小し、またファイル自動拡張を無効化します。こうすることでオンラインインデックス作成が領域不足でエラーとなり、その結果中断状態となることを誘発させます。

Index_Operation_Enhancements.sql

```
USE WideWorldImportersDW;
GO
-- インデックスが存在する場合は削除する
IF EXISTS (SELECT * FROM sys.indexes WHERE name = N'CCX_Fact_OrderHistoryExtended')
BEGIN
  DROP INDEX CCX_Fact_OrderHistoryExtended ON Fact.OrderHistoryExtended;
END;
GO
-- データファイルのサイズを可能な限り縮小
DBCC SHRINKFILE (N'WWI_UserData', 0);
GO
-- データファイルのサイズを 200MB 分拡張
DECLARE @size int = (
  SELECT ((size * 8) / 1024) + 200 FROM sys.database_files
  WHERE name = N'WWI_UserData');
DECLARE @sqlstmt nvarchar(max);

SELECT @sqlstmt = concat(
  'ALTER DATABASE WideWorldImportersDW MODIFY FILE (',
  'NAME = N''WWI_UserData'', SIZE = ',
  @size,
  'MB)'
);
EXEC sp_executesql @sqlstmt;
GO

USE master;
GO
-- データファイルの自動拡張を無効化
ALTER DATABASE WideWorldImportersDW MODIFY FILE (
  NAME = N'WWI_UserData', FILEGROWTH = 0
);
GO
```

2. オンラインインデックス作成を開始します。

Index_Operation_Enhancements.sql

```
USE WideWorldImportersDW;
GO
CREATE NONCLUSTERED INDEX IX_Customer_Key ON Fact.OrderHistoryExtended (
```

```
  [City Key] ASC,
  [Customer Key] ASC
) WITH (ONLINE = ON, RESUMABLE = ON);
GO
```

「Resumable online index creation」は、オンラインインデックス作成時にのみ使用できるものであるため、「RESUMABLE = ON」に加えて「ONLINE = ON」の指定も必要です。コマンドに「ONLINE = ON」を指定せずに「RESUMABLE = ON」のみを指定して CREATE INDEX コマンドを実行すると、以下のエラーが返されます。

```
Msg 11438, Level 15, State 1
ONLINE オプションが 'OFF' に設定されている場合は、再開可能 オプションを 'ON' に設定することはできません。
```

3. インデックス作成処理が領域不足により中断となることを確認します。

```
Messages
2:27:57     Started executing query at Line 1
            Commands completed successfully.
2:27:57     Started executing query at Line 3
            ステートメントは終了されました。
            Msg 1105, Level 17, State 2, Line 3
            データベース 'WideWorldImportersDW' にオブジェクト 'dbo.SORT temporary run
            Total execution time: 00:00:06.984
```

```
Msg 1105, Level 17, State 2, Line 3
データベース 'WideWorldImportersDW' にオブジェクト 'dbo.SORT temporary run storage: 72057594084196352' の領域を割り当てられませんでした。'USERDATA' ファイル グループがいっぱいです。不要なファイルの削除、ファイル グループ内のオブジェクトの削除、ファイル グループへの新しいファイルの追加、またはファイル グループの既存のファイルの自動拡張の設定のいずれかを行ってディスク領域を作成してください。
```

タイミングによっては以下のエラーメッセージとなる、もしくは両方のエラーが出力される場合となることもありますが、次の手順へ進んで問題ありません。

```
Msg 1101, Level 17, State 1
データベース 'WideWorldImportersDW' の新しいページを割り当てられませんでした。ファイル グループ 'USERDATA' のディスク領域が不足しています。ファイル グループ内のオブジェクトの削除、ファイル グループへの新しいファイルの追加、またはファイル グループの既存のファイルの自動拡張の設定のいずれかを行って、必要な領域を作成してください。
```

インデックス作成処理の状態を確認する

カタログビューからインデックス作成処理の状態を確認します。

```
Index_Operation_Enhancements.sql
USE WideWorldImportersDW;
GO
SELECT name, percent_complete, state_desc, last_pause_time, page_count
FROM sys.index_resumable_operations;
GO
```

「IX_Customer_Key」の「state_desc」列の値が「PAUSED」となっており、インデックスの作成処理が中断状態にあることが確認できます。また「percent_complete」の値からインデックスの作成処理がどの程度まで進んでいるのかを確認することもできます。

Results　Messages

	name	percent_complete	state_desc	last_pause_time	page_count
1	IX_Customer_Key	13.5086244987295	PAUSED	2020-03-06 02:28:03.130	0

なおオンラインインデックス作成処理の「中断」には以下のような種類があります。

中断の種類		説明
手動中断		ALTER INDEX コマンドに「PAUSE」オプションを指定してインデックスの作成を行っているセッションとは別のセッションから明示的に中断コマンドを発行することを示します。
自動中断		
	エラー検出による自動中断	今回実行したようにインデックスの作成処理が何らかの要因でエラーとなったことによる自動中断を示します。
	明示的自動中断	CREATE INDEX コマンドの「MAX_DURATION」オプションに指定した時間に達してもインデックスの作成が終了していない場合の自動中断を示します。「MAX_DURATION」オプションを指定することで、インデックスの作成が自動で中断されるまでの時間を分単位で指定することができます。

エラーの原因を解消する

WideWorldImportersDW データベースのファイル自動拡張を有効化してエラーの原因を解消させます。

```
Index_Operation_Enhancements.sql
USE master;
GO
ALTER DATABASE WideWorldImportersDW MODIFY FILE (
  NAME = N'WWI_UserData', FILEGROWTH = 65536KB
);
GO
```

インデックス作成処理を再開して状態を確認する

1. インデックス作成処理を再開します。

```
Index_Operation_Enhancements.sql
USE WideWorldImportersDW;
GO
ALTER INDEX IX_Customer_Key ON Fact.OrderHistoryExtended RESUME;
GO
```

2. インデックス作成処理中に状況を確認します。インデックス作成処理とは別のセッションでクエリを実行します。

```
Index_Operation_Enhancements.sql
USE WideWorldImportersDW;
GO
SELECT name, percent_complete, state_desc, last_pause_time, page_count
FROM sys.index_resumable_operations;
GO
```

「IX_Customer_Key」の「state_desc」列の値が「RUNNING」となっており、インデックスの作成処理が再開していることが確認できます。また、上記クエリを複数回実行すると、「percent_complete」の値が徐々に増えていくことが確認でき、インデックスの作成処理が進行していることがわかります。

Results　Messages

	name	percent_complete	state_desc	last_pause_time	page_count
1	IX_Customer_Key	59.5100405337666	RUNNING	2020-03-06 02:28:03.130	16850

➢ Always On Availability Group Enhancements

1. 概要

Always On Availability Group に関して 3 つの機能強化があります。

1. 同期レプリカの最大数が従来の 3 台から 5 台（プライマリレプリカ 1 台、同期セカンダリレプリカ 4 台）に拡張されます。
2. 「クラスターレス読み取りスケール可用性グループ」や「クロスプラットフォームクラスターによる分散型可用性グループ」の構成は高可用性を目的とした構成ではないため、自動フェールオーバーやプライマリレプリカへの透過接続は提供されません。そのため手動フェールオーバーを行う場合はそのたびにクライアントからの接続先をフェールオーバー先の新しいプライマリレプリカへ変更する必要がありました。この課題に対して SQL Server 2019 では Secondary-to-primary replica connection redirection 機能により、クライアントで指定された接続先レプリカに関係なく、接続を常にプライマリレプリカへリダイレクトすることができます。
3. SQL Server 内部のタスクは関連付けられたワーカーないしスケジューラーを移動することはないため、そのタスクの重みによってはスケジューラーに関連付けられた論理 CPU 間で一時的に負荷の不均一が生じるケースがありました。この課題に対して SQL Server 2019 は Scheduler worker migration のメカニズムを導入し、アイドル状態のスケジューラーへタスクを移動させて負荷の不均一を解消し、システム全体の CPU 利用効率を上げることでパフォーマンス改善を図っています。執筆時点では「REDO 処理のパラレル化」が Scheduler worker migration に対応しています。

2. 利用シーン

Always On Availability Group は SQL Server 2012 での登場以来、継続的に機能強化が続けられています。ここではスケーラビリティ、パフォーマンス、フォールトトレランスの観点ごとに強化ポイントをいくつかピックアップして解説します。

Always On Availability Group のスケーラビリティ強化

❖ 同期レプリカ数の上限

SQL Server 2012 の登場初期は 2 台、SQL Server 2016 で 3 台、そして SQL Server 2019 では 5 台に拡張されます。

❖ ロードバランスされた読み取り自動ルーティングの制御（SQL Server 2016 で導入）

読み取り自動ルーティングは読み取りを目的としたクライアントからの接続要求に対して自動的にセカンダリレプリカへルーティングします。これによりレポーティングやバックアップなどの読み取りを目的としたセカンダリレプリカへの明示的な接続についても、それをホストする SQL Server インスタンスを意識することなく透過的に接続できるようになります。従来はあらかじめ設定された優先順に基づいて最初に使用可能なレプリカへルーティングされていましたが、SQL Server 2016 ではロードバランスされたルーティングを設定できるようになりました。設定は可

用性グループのパラメーター（READ_ONLY_ROUTING_URL と READ_ONLY_URL_LIST）によって設定します。詳細は次の Docs を確認ください。

◇ Always On 可用性グループの読み取り専用ルーティングの構成
…読み取り専用レプリカ間の負荷分散の構成
https://docs.microsoft.com/ja-jp/sql/database-engine/availability-groups/windows/configure-read-only-routing-for-an-availability-group-sql-server?view=sql-server-ver15#loadbalancing

❖ 分散トランザクションサポート（SQL Server 2016 で導入）

SQL Server 2016 にて「異なるインスタンス＆異なる可用性グループのデータベース」をまたぐ分散トランザクションがサポートされました。さらに、SQL Server 2016 SP2 では「同一インスタンス＆異なる可用性グループのデータベース」をまたぐ分散トランザクション、ならびに「同一インスタンス上の同一可用性グループのデータベース」をまたぐ分散トランザクションがサポートされました。詳細は次の Docs を確認ください。

◇ Always On 可用性グループ用に分散トランザクションを構成する
https://docs.microsoft.com/ja-jp/sql/database-engine/availability-groups/windows/configure-availability-group-for-distributed-transactions?view=sql-server-ver15

❖ 自動シード処理によるセカンダリレプリカの初期化（SQL Server 2016 で導入）

セカンダリレプリカを初期化するためには「プライマリでバックアップ取得→セカンダリでリストア」というステップを踏む必要がありました。SQL Server 2016 で実装された自動シード処理を利用することで、セカンダリレプリカのデータベースを自動的に作成することができるようになりました。詳細は次の Docs を確認ください。

◇ 自動シード処理を使用して、Always On 可用性グループのセカンダリ レプリカを初期化する
https://docs.microsoft.com/ja-jp/sql/database-engine/availability-groups/windows/automatic-seeding-secondary-replicas?view=sql-server-ver15

❖ プライマリレプリカへの自動リダイレクト制御（SQL Server 2019 で導入）

SQL Server 2017 で導入された「クラスターレス読み取りスケール可用性グループ」や「クロスプラットフォームクラスターによる分散型可用性グループ」については、高可用性を目的とした構成ではないため自動フェールオーバーやプライマリレプリカへの透過接続は提供されていませんでした。そのため各レプリカの現在のロールに応じてクライアントの接続設定を明示的に変更する必要がありました。この課題に対して SQL Server 2019 ではクライアントで指定された接続先レプ

リカに関係なく接続を常にプライマリレプリカにリダイレクトすることができるようになります。この機能は可用性グループのパラメーター（READ_WRITE_ROUTING_URL）によって設定します。

Always On Availability Group のパフォーマンス強化

❖ ログ転送に関連する処理の最適化（SQL Server 2016 で導入）

HW の進化に伴うボトルネックの遷移に対してログ転送に関連する処理を最適化することで性能改善が図られました。これには「Log Writer のマルチスレッド化」、「REDO 処理のパラレル化」、「ログブロックの圧縮アルゴリズム改善と並列化」、「暗号化のアルゴリズム改善」、「コンテキストスイッチの削減」、「スピンロックの最適化」といった改善が含まれています。

❖ レプリカの役割に応じたデータベース設定の制御（SQL Server 2016 で導入）

クエリ実行の最適化に影響する MAXDOP、LEGACY_CARDINALITY_ESTIMATION、PARAMETER_SNIFFING、QUERY_OPTIMIZER_HOTFIXES のパラメーターについて、プライマリとセカンダリの役割に応じた設定をもつことができるようになりました。例えば、オンライン処理を行うプライマリレプリカについては MAXDOP=1、レポートティング処理を行うセカンダリレプリカについては MAXDOP=0 といった切り替えが可能になります。詳細は次の Docs を確認ください。

- ◇ ALTER DATABASE SCOPED CONFIGURATION (Transact-SQL)
  https://docs.microsoft.com/ja-jp/sql/t-sql/statements/alter-database-scoped-configuration-transact-sql?view=sql-server-ver15

❖ コミットに必要な同期セカンダリ最小数の制御（SQL Server 2017 で導入）

すべての同期セカンダリへのログ転送完了を待たずに設定した最小数のログ転送完了によってプライマリレプリカ上のトランザクションのコミットを完了させることができるようになり、パフォーマンスとフォールトトレランスのトレードオフを選択できるようになりました。最小数の設定は可用性グループの REQUIRED_SYNCHRONIZED_SECONDARIES_TO_COMMIT パラメーターによって設定します。詳細は次の Docs を確認ください。

- ◇ CREATE AVAILABILITY GROUP (Transact-SQL)
  https://docs.microsoft.com/ja-jp/sql/t-sql/statements/create-availability-group-transact-sql?view=sql-server-ver15

❖ CPU スケジューリングの最適化（SQL Server 2019 で導入）

SQL Server 内部のタスクは関連付けられたワーカーないしスケジューラーを移動することはないため、そのタスクの重みによってはスケジューラーに関連付けられた論理 CPU 間で一時的に負荷

の不均一が生じるケースがありました。この課題に対して SQL Server 2019 は Scheduler worker migration のメカニズムを導入し、アイドル状態のスケジューラーへタスクを移動させ負荷の不均一を解消し、システム全体の CPU 利用効率を上げることでパフォーマンス改善を図ります。執筆時点では「REDO 処理のパラレル化」が Scheduler worker migration に対応しています。Scheduler worker migration の詳細は、SQL Server 2019 自習書「1. Intelligent Performance」-「Scheduler worker migration」の項目を参照ください。

Always On Availability Group のフォールトトレランス強化

❖ データベースレベルの正常性検出制御（SQL Server 2016 で導入）

従来はインスタンスの正常性が自動フェールオーバーのトリガーでしたが、SQL Server 2016 ではデータベースの正常性も自動フェールオーバーのトリガーになりました。データベースレベルの正常性検出の ON/OFF は可用性グループのパラメーター（DB_FAILOVER）によって設定します。ON に設定すると、可用性グループ内のデータベースがオンライン以外のすべての状態で自動フェールオーバーのトリガーとなります。この動作はインスタンスの正常性検出（FAILURE_CONDITION_LEVEL）とは独立して機能します。詳細は次の Docs を確認ください。

◇ 可用性グループのデータベース レベルの正常性検出フェールオーバー オプション
https://docs.microsoft.com/ja-jp/sql/database-engine/availability-groups/windows/sql-server-always-on-database-health-detection-failover-option?view=sql-server-ver15

Always On Availability Group のクラスター構成のバリエーション拡張

❖ 単一ドメインクラスターによる可用性グループ（SQL Server 2012 で導入）

Always On が導入された当初にサポートされたクラスター構成です。

詳細は次の Docs を確認ください。

◇ AlwaysOn 可用性グループの概要（SQL Server）
https://docs.microsoft.com/ja-jp/sql/database-engine/availability-groups/windows/overview-of-always-on-availability-groups-sql-server?view=sql-server-ver15

❖ ワークグループクラスターによるドメインに依存しない可用性グループ（SQL Server 2016 で導入）

ドメインに属さないワークグループサーバーによるクラスター構成です。

詳細は次の Docs を確認ください。

◇ ドメインに依存しない可用性グループを作成する
https://docs.microsoft.com/ja-jp/sql/database-engine/availability-groups/windows/domain-independent-availability-groups?view=sql-server-ver15

❖ マルチドメインクラスターによる分散型可用性グループ（SQL Server 2016 で導入）

ドメインをまたぐ分散型のクラスター構成です。

詳細は次の Docs を確認ください。

◇ 分散型可用性グループ
https://docs.microsoft.com/ja-jp/sql/database-engine/availability-groups/windows/distributed-availability-groups?view=sql-server-ver15

❖ 基本的な可用性グループ（SQL Server 2016 で導入）

Standard Edition 向けに提供される機能制限のあるクラスター構成です。

詳細は次の Docs を確認ください。

◇ 単一データベース用の基本的な Always On 可用性グループ
https://docs.microsoft.com/ja-jp/sql/database-engine/availability-groups/windows/basic-availability-groups-always-on-availability-groups?view=sql-server-ver15

❖ Pacemaker/Corosync による Linux 版可用性グループ（SQL Server 2017 で導入）

SQL Server on Linux におけるクラスター構成です。

詳細は次の Docs を確認ください。

◇ Linux デプロイでの SQL Server 可用性の基本
https://docs.microsoft.com/ja-jp/sql/linux/sql-server-linux-ha-basics?view=sql-server-ver15

❖ クロスプラットフォームクラスターによる分散型可用性グループ（SQL Server 2017 で導入）

プラットフォームをまたぐ分散型のクラスター構成です。

詳細は次の Docs を確認ください。

◇ Windows 上と Linux 上で SQL Server の Always On 可用性グループを構成する (クロスプラットフォーム)
https://docs.microsoft.com/ja-jp/sql/linux/sql-server-linux-availability-group-cross-platform?view=sql-server-ver15

❖ クラスターレス読み取りスケール可用性グループ（SQL Server 2017 で導入）

クラスターウェアを利用しないクラスター構成です。

詳細は次の Docs を確認ください。

◇ Always On 可用性グループの読み取りスケールを構成する
https://docs.microsoft.com/ja-jp/sql/database-engine/availability-groups/windows/configure-read-scale-availability-groups?view=sql-server-ver15

3. やってみよう

ここでは SQL Server 2019 で導入された「プライマリレプリカへの自動リダイレクト制御」を試してみましょう。今回は「クラスターレス読み取りスケール可用性グループ」構成で動作確認を実施するため、まずは事前準備で「クラスターレス読み取りスケール可用性グループ」の構築手順を解説し、その後に「プライマリレプリカへの自動リダイレクト制御」の動作を確認します。

✱ 事前準備

「クラスターレス読み取りスケール可用性グループ」を構成します。作成する環境の完成イメージは以下の通りです。

ここでは Windows Server 2019 をインストールしたサーバーが 2 台とそれぞれのサーバー上に SQL Server 2019 がインストール済みであることを前提にそれ以降の手順を解説します。
なお、それぞれのサーバー名には「SERVER01」と「SERVER02」を設定しているので、環境に応じて読み替えてください。

Always On 可用性グループ機能を有効化する

@プライマリ（SERVER01） @セカンダリ（SERVER02）

1. SERVER01 の SQL Server サービスで Always On 可用性グループ機能の有効化とサービス再起動を行います。

Always_On_Availability_Group_Enhancements.ps.txt

```
Enable-SqlAlwaysOn -ServerInstance SERVER01 -Force
Restart-Service -Name 'MSSQLSERVER'
```

2. SERVER01 と同様に、SERVER02 でも機能の有効化とサービス再起動を行います。

Always_On_Availability_Group_Enhancements.ps.txt

```
Enable-SqlAlwaysOn -ServerInstance SERVER02 -Force
Restart-Service -Name 'MSSQLSERVER'
```

Always On 可用性グループの通信で使用するポートを開く

@プライマリ（SERVER01） @セカンダリ（SERVER02）

SERVER01、SERVER02 それぞれで TCP ポート 1433 と 5022 を開きます。

Always_On_Availability_Group_Enhancements.ps.txt

```
New-NetFirewallRule -DisplayName "TCP 1433" -Direction Inbound -LocalPort 1433 -Protoc
ol TCP -Action Allow
New-NetFirewallRule -DisplayName "TCP 5022" -Direction Inbound -LocalPort 5022 -Protoc
ol TCP -Action Allow
```

なお、「TCP ポート 1433」が SQL Server サービスのポート、「TCP ポート 5022」が Always On 可用性グループの通信で使用するポートです。それぞれのポートは環境に応じて読み替えてください。

エンドポイント間通信用の SQL Server 認証ログインを構成する

@プライマリ（SERVER01） @セカンダリ（SERVER02）

SERVER01、SERVER02 それぞれでエンドポイント間通信用の SQL Server 認証ログインとデータベースユーザーを作成します。

Always_On_Availability_Group_Enhancements.sql

```
USE master;
GO
-- エンドポイントアクセス用ログインの作成
CREATE LOGIN DBM_Endpoint_Login WITH PASSWORD = 'Passw0rd1!';
```

```
GO
-- master 上に DB ユーザーを作成し、DBM_Endpoint_Login ログインとマッピング
CREATE USER DBM_Endpoint_User FOR LOGIN DBM_Endpoint_Login;
GO
```

証明書を作成してエクスポートする

@プライマリ（SERVER01）

エンドポイント間通信を SQL Server 認証で行う場合には証明書を使用したエンドポイント間認証が必要です。SERVER01 で証明書の作成とエクスポートを行います。

Always_On_Availability_Group_Enhancements.sql

```
USE master;
GO
-- マスターキーの作成（既に存在する場合は作成をスキップする）
IF NOT EXISTS (
  SELECT 1 FROM sys.symmetric_keys WHERE name = '##MS_DatabaseMasterKey##'
)
BEGIN
  CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'Passw0rd1!';
END
GO
-- 証明書の作成
CREATE CERTIFICATE DBM_Certificate WITH SUBJECT = 'DBM_Certificate';
GO
-- 証明書のエクスポート
BACKUP CERTIFICATE DBM_Certificate
  TO FILE = 'C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLSERVER¥MSSQL¥DATA¥DBM_
Certificate.cer'
  WITH PRIVATE KEY (
    FILE = 'C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLSERVER¥MSSQL¥DATA¥DBM_C
ertificate.pvk',
    ENCRYPTION BY PASSWORD = 'Passw0rd1!!'
  );
```

「C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLSERVER¥MSSQL¥DATA」配下に作成された「DBM_Certificate.cer」と「DBM_Certificate.pvk」の 2 つのファイルをセカンダリサーバーの「C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLSERVER¥MSSQL¥DATA」配下にコピーします。

証明書をインポートする

@セカンダリ（SERVER02）

エンドポイント間通信用に SERVER01 からコピーした証明書を SERVER02 へインポートします。

Always_On_Availability_Group_Enhancements.sql

```
USE master;
GO
-- マスターキーの作成（既に存在する場合は作成をスキップする）
IF NOT EXISTS (
  SELECT 1 FROM sys.symmetric_keys WHERE name = '##MS_DatabaseMasterKey##'
)
BEGIN
```

```
  CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'Passw0rd1!';
END
GO
-- 証明書のインポート
CREATE CERTIFICATE DBM_Certificate
  AUTHORIZATION DBM_Endpoint_User
  FROM FILE = 'C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLSERVER¥MSSQL¥DATA¥DB
M_Certificate.cer'
  WITH PRIVATE KEY (
    FILE = 'C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLSERVER¥MSSQL¥DATA¥DBM_C
ertificate.pvk',
    DECRYPTION BY PASSWORD = 'Passw0rd1!!'
  );
GO
```

エンドポイントを作成する

@プライマリ（SERVER01） @セカンダリ（SERVER02）

SERVER01、SERVER02 それぞれでエンドポイントを作成します。

Always_On_Availability_Group_Enhancements.sql

```
USE master;
GO
-- エンドポイントの作成
CREATE ENDPOINT Hadr_Endpoint
  AS TCP (LISTENER_PORT = 5022)
  FOR DATA_MIRRORING (
    ROLE = ALL,
    AUTHENTICATION = CERTIFICATE DBM_Certificate,
    ENCRYPTION = REQUIRED ALGORITHM AES
  );
GO
-- エンドポイントを開始
ALTER ENDPOINT Hadr_Endpoint STATE = STARTED;
GO
-- エンドポイント間通信用のログインにエンドポイントへの接続権限を付与
GRANT CONNECT ON ENDPOINT::Hadr_Endpoint TO DBM_Endpoint_Login;
GO
```

読み取りスケール可用性グループを作成する

@プライマリ（SERVER01）

SERVER01 で読み取りスケールの可用性グループを作成します。

Always_On_Availability_Group_Enhancements.sql

```
USE master;
GO
-- 読み取りスケール可用性グループの作成
CREATE AVAILABILITY GROUP TestAG WITH (CLUSTER_TYPE = NONE) FOR REPLICA ON
  N'SERVER01' WITH (
    ENDPOINT_URL = N'tcp://SERVER01:5022',
    AVAILABILITY_MODE = SYNCHRONOUS_COMMIT,
    FAILOVER_MODE = MANUAL,
    SEEDING_MODE = AUTOMATIC
  ),
  N'SERVER02' WITH (
```

```
    ENDPOINT_URL = N'tcp://SERVER02:5022',
    AVAILABILITY_MODE = SYNCHRONOUS_COMMIT,
    FAILOVER_MODE = MANUAL,
    SEEDING_MODE = AUTOMATIC
  );
GO
-- 可用性グループ TestAG にデータベースの作成権限を付与
-- (自動 SEEDING モードで同期させる場合、可用性グループに DB 作成権限が必要なため)
ALTER AVAILABILITY GROUP TestAG GRANT CREATE ANY DATABASE;
GO
```

メンバーサーバーを可用性グループへ参加させる

@セカンダリ (SERVER02)

SERVER02 を可用性グループへ参加させます。

Always_On_Availability_Group_Enhancements.sql

```
USE master;
GO
-- 可用性グループ TestAG へ参加
ALTER AVAILABILITY GROUP TestAG JOIN WITH (CLUSTER_TYPE = NONE);
GO
-- 可用性グループ TestAG にデータベースの作成権限を付与
-- (自動 SEEDING モードで同期させる場合、可用性グループに DB 作成権限が必要なため)
ALTER AVAILABILITY GROUP TestAG GRANT CREATE ANY DATABASE;
GO
```

可用性データベースを作成する

@プライマリ (SERVER01)

可用性グループにデータベースを追加します。

Always_On_Availability_Group_Enhancements.sql

```
USE master;
GO
-- 新しいデータベース TestDB を作成
CREATE DATABASE TestDB;
GO
-- TestDB の復旧モデルを「完全 (FULL)」に設定
ALTER DATABASE TestDB SET RECOVERY FULL;
GO
-- TestDB 上にテスト用のテーブルを作成
CREATE TABLE TestDB..test (c1 int NOT NULL PRIMARY KEY CLUSTERED, c2 varchar(30));
GO
-- 前提条件を満たすため、TestDB の完全バックアップを実行
BACKUP DATABASE TestDB TO DISK = N'C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQL
SERVER¥MSSQL¥Backup¥ TestDB.bak';
GO
-- 可用性グループ TestAG へデータベース TestDB を追加
ALTER AVAILABILITY GROUP TestAG ADD DATABASE TestDB;
GO
```

可用性グループによる同期を確認する

@セカンダリ (SERVER02)

SERVER01 で作成したデータベースが SERVER02 に同期されていることを確認します。

```
Always_On_Availability_Group_Enhancements.sql
USE master;
GO
-- データベース TestDB が作成されていることを確認
SELECT name, create_date, state_desc, recovery_model_desc FROM sys.databases
  WHERE name = 'TestDB';
GO
-- データベース TestDB の同期状態を確認
SELECT DB_NAME(database_id) AS 'database', synchronization_state_desc
  FROM sys.dm_hadr_database_replica_states;
GO
```

次の結果が返されれば、可用性グループが正常に機能しています。

Results　Messages

	name	create_date	state_desc	recovery_model_desc
1	TestDB	2020-03-06 13:32:33.943	ONLINE	FULL

	database	synchronization_state_desc
1	TestDB	SYNCHRONIZED

✱ プライマリレプリカへの自動リダイレクト制御の動作確認

プライマリレプリカへの自動リダイレクト制御を有効化するためには新たに追加されたパラメーター「READ_WRITE_ROUTING_URL」を設定しますが、この設定に加えて次の２つの設定がすべて満たされていることで自動リダイレクトが機能する点に注意してください。

- 可用性グループの「SECONDARY_ROLE」の「ALLOW_CONNECTION」に「ALL」が指定されていること。
- SQL Server の接続設定の「Application Intent」で「ReadWrite（既定）」が指定されていること。

設定パターンに応じてセカンダリレプリカへの接続がどのように制御されるかを次の表に整理します。

		SECONDARY_ROLE の ALLOW_CONNECTION の設定値		
		NO（既定）	READ_ONLY	ALL
接続設定 Application Intent	ReadWrite（既定）	接続失敗	接続失敗	ReadWrite 接続としてプライマリへ自動リダイレクト
	ReadOnly	接続失敗	Read 専用接続としてセカンダリへ接続	Read 専用接続としてセカンダリへ接続

新たに追加された「READ_WRITE_ROUTING_URL」を設定する

@プライマリ（SERVER01）

事前準備で作成した可用性グループに対して「READ_WRITE_ROUTING_URL」設定を行います。

Always_On_Availability_Group_Enhancements.sql

```
USE master;
GO
ALTER AVAILABILITY GROUP TestAG MODIFY REPLICA ON
  N'SERVER01' WITH (PRIMARY_ROLE (READ_WRITE_ROUTING_URL = 'TCP://SERVER01:1433'));
GO
ALTER AVAILABILITY GROUP TestAG MODIFY REPLICA ON
  N'SERVER02' WITH (PRIMARY_ROLE (READ_WRITE_ROUTING_URL = 'TCP://SERVER02:1433'));
GO
```

「SECONDARY_ROLE」の「ALLOW_CONNECTION」に ALL を指定する

@プライマリ（SERVER01）

続けて、可用性グループがセカンダリロールである場合に許可される接続の種類を「ALL」に設定します。

Always_On_Availability_Group_Enhancements.sql

```
USE master;
GO
ALTER AVAILABILITY GROUP TestAG MODIFY REPLICA ON
  N'SERVER01' WITH (SECONDARY_ROLE (ALLOW_CONNECTIONS = ALL));
GO
ALTER AVAILABILITY GROUP TestAG MODIFY REPLICA ON
  N'SERVER02' WITH (SECONDARY_ROLE (ALLOW_CONNECTIONS = ALL));
GO
```

セカンダリレプリカに対して Application Intent = ReadWrite で接続する

セカンダリレプリカに対して「Application Intent = ReadWrite」で接続し、該当の接続がプライマリレプリカへ自動リダイレクトされることを確認します。

コマンドプロンプトから次のコマンドを実行します。なお sqlcmd は既定で「Application Intent = ReadWrite」の動作になります。-U と-P は環境に応じて読み替えてください。

Always_On_Availability_Group_Enhancements.bat.txt

```
sqlcmd -S SERVER02 -U sa -P Passw0rd -d TestDB -Q "SELECT @@SERVERNAME" -W
```

「SELECT @@SERVERNAME」は、接続先の SQL Server が稼働するサーバー名を返します。現在のセカンダリレプリカである「SERVER02」に接続しましたが、クエリの結果では現在のプライマリレプリカである「SERVER01」が返ってきています。これにより、プライマリレプリカへ自動リダイレクトされたことが確認できます。

＊ フェールオーバー後も自動リダイレクトが正しく機能することの動作確認

+αとしてフェールオーバー後も自動リダイレクトが正しく機能することを確認してみます。

可用性グループを手動フェールオーバーする

@セカンダリ（SERVER02）

「クラスターレス読み取りスケール可用性グループ」のフェールオーバーは次のような「強制フェールオーバー（FORCE_FAILOVER_ALLOW_DATA_LOSS）」で行う必要があります。これを使って、SERVER02 へフェールオーバーさせます。

Always_On_Availability_Group_Enhancements.sql

```
USE master;
GO
-- 強制フェールオーバー
ALTER AVAILABILITY GROUP TestAG FORCE_FAILOVER_ALLOW_DATA_LOSS;
GO
-- フェールオーバー後の状態を確認
SELECT primary_replica, primary_recovery_health_desc
  FROM sys.dm_hadr_availability_group_states;
GO
```

クエリ結果で「primary_replica」の値が "SERVER02"、「primary_recovery_health_desc」の値が "ONLINE" となっていれば、正常にフェールオーバーが完了しています。

Results　Messages

	primary_replica	primary_recovery_health_desc
1	SERVER02	ONLINE

可用性データベースの同期を再開する

@新セカンダリ
（SERVER01）

強制フェールオーバー後はデータ同期が中断状態となるため、手動でデータ同期を再開させる必要があります。

Always_On_Availability_Group_Enhancements.sql

```
USE master;
GO
-- データ同期の再開
ALTER DATABASE TestDB SET HADR RESUME;
GO
-- 同期の再開が反映されるまで必要に応じて待機
WAITFOR DELAY '00:00:15'
GO
-- 可用性データベースの同期状態を確認
SELECT DB_NAME(database_id) AS 'database', synchronization_state_desc
  FROM sys.dm_hadr_database_replica_states;
GO
```

クエリ結果で可用性データベースの「synchronization_state_desc」の値が "SYNCHRONIZED" と出力されていれば、正常に同期が再開できています。

Results　Messages

	database	synchronization_state_desc
1	TestDB	SYNCHRONIZED

新しいセカンダリレプリカに対して Application Intent = ReadWrite で接続する

先ほどと同様に、コマンドプロンプトから次のコマンドを実行します。-U と-P は環境に応じて読み替えてください。

Always_On_Availability_Group_Enhancements.bat.txt

```
sqlcmd -S SERVER01 -U sa -P Passw0rd -d TestDB -Q "SELECT @@SERVERNAME" -W
```

管理者: C:¥Windows¥system32¥cmd.exe

```
C:¥Users¥Administrator>sqlcmd -S SERVER01 -U sa -P Passw0rd -d TestDB
-Q "SELECT @@SERVERNAME" -W

SERVER02

(1 行処理されました)

C:¥Users¥Administrator>
```

現在のセカンダリレプリカである「SERVER01」に接続しましたが、クエリの結果では現在のプライマリレプリカである「SERVER02」が返ってきています。これにより、新しいプライマリレプリカへ自動リダイレクトされたことが確認できます。

➢ Database Snapshots Enhancements

1. 概要

In-Memory OLTP 機能によるインメモリ化されたテーブルに対する Database Snapshots がサポートされました。レポートのためのデータの履歴保持、オペレーションミスへの備え、テスト用途でのコピーなど、インメモリ化されたテーブルデータのある時点のコピー（スナップショット）をコピー元の大きさに依存することなく高速に作成し、格納効率よく保持することができるようになります。

2. 利用シーン

データベーススナップショットは既存のデータベースに対して作成できる読み取り専用の静的ビューです。データベーススナップショットを使用することで随時更新されていくデータとは別に、スナップショットを作成した時点のデータへアクセスすることができるようになります。これはレポートのためのデータの履歴保持、オペレーションミスへの備え、テスト用途でのコピーなどに活用することできます。
スナップショットはスパースファイルを使用しており、スナップショット作成直後は本質的に空のスパースファイルファイルが作成され、ソースデータベースで更新されるページが増えるにつれてファイルのサイズが大きくなります。こうした動きによりスナップショットはソースデータベースの大きさに依存せず高速に作成され、格納効率よく元データを保持することができるようになります。

SQL Server 2017 以前は In-Memory OLTP 機能によるメモリ最適化ファイルグループを含むデータベースに対してデータベーススナップショットを作成することができませんでしたが、SQL Server 2019 ではこの制限がなくなりました。

4. User experience enhancement for DBA

このセクションは、主にデータベース管理者にとっての使いやすさが強化された機能を取り上げます。

Query Profiling	軽量クエリプロファイラのフットプリントがさらに軽くなり既定で有効な機能となります。
Plan cache	ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE で plan_handle を指定できるようなり、クエリごとにプロシージャキャッシュをクリアすることができるようになります。
Columnstore indexes	データ圧縮機能による圧縮効果を推定するための sp_estimate_data_compression_savings ストアドプロシージャが列ストアインデックスにも対応し、これから作成する列ストアインデックスの推定サイズや、作成済みの列ストアインデックスをアーカイブ圧縮した際の推定サイズを事前に知ることができるようになります。 また、クエリチューニングに活用する DBCC CLONEDATABASE 機能が列ストアインデックスにも対応し、列ストアに関するトラブルシューティングの手段が強化されます。
Error Messages	列のサイズを超過するデータ挿入または更新が要求されたことを示すエラーメッセージが強化され、対象となったオブジェクトや切り捨てられたデータを確認できるようになります。
Page Information	データページの詳細情報を取得するための新しい動的管理関数 sys.dm_db_page_info が追加されます。これによりデータページへのロックやラッチの競合、もしくはページ破損など、より複雑な問題のトラブルシューティングやデバッグのための作業プロセスが簡素化されます。
Query Store	QUERY_CAPTURE_POLICY オプションによりクエリストアでキャプチャする情報を細かく指定できるようになり、キャプチャ処理のオーバーヘッドや蓄積データの量を制御できるようになります。
Statistics	sys.dm_os_wait_stats 動的管理ビューに追加された新しい待機の種類 WAIT_ON_SYNC_STATISTICS_REFRESH によって統計情報の更新に伴うユーザークエリの待機時間をインスタンスレベルで把握することができるようになります。
Resource Governor	リソースガバナーの REQUEST_MAX_MEMORY_GRANT_PERCENT に指定できる値範囲を float の精度で指定できるようになり、よりきめ細かいメモリリソース制御が可能になります。
Setup options	SQL Server の主要なチューニングパラメーターである max degree of parallelism、max server memory (KB)、min server memory (KB)の設定が Windows 版についてはセットアップウィザードから設定することができるようになります。

4.1. Query Profiling

➢ Lightweight Query Profiling

1. 概要

軽量クエリプロファイラのフットプリントがさらに軽くなり、既定で有効な機能となります。例えば、クエリチューニングの職人が現場に到着した時点で既に問題の足跡（実際のクエリ実行プラン）が収集済みである状況を想像してください。従来のようなクエリの性能問題を検出した後の情報採取であったり、それ以前に問題の再現待ちであったりと、後手に回りがちな対処でしたが、軽量プロファイリングによって将来起こり得るクエリの性能問題に対する軽量な"網"を事前に仕掛けておくことができるのは大きな進歩です。

2. 利用シーン

SQL Server には事後のトラブルシュートのために各種ログに加えて、default trace、system health、telemetry events、動的管理ビューなど既定で様々な情報（≒SQL Server のフライトレコード）が採取されています。こうした情報と SQL Server のメカニズムを熟知した技術者の経験によって、多くの問題は紐解かれ対処されます。

クエリ性能問題も同様です。典型的には動的管理ビューを通じて「クエリごとのフライトレコード（実行時間や回数、CPU や I/O のリソース使用など）」を確認することで、問題のクエリを特定します。特定したクエリそのものに明確な問題が見られる場合には、クエリを書き換えることで完了するケースもありますが、クエリそのものに限らず実行時のいくつかの条件が重なり合っているであろうと考えられるケースもあります。このようなケースでは、該当クエリの「クエリ実行プラン」を確認しながら、それを「より適切なクエリ実行プラン」へと矯正するために、クエリそのものや SQL Server インフラ（パラメーター、インデックス、統計情報など）に手を入れてチューニングを行います。

しかし、上記のプロセスだけでは原因を明確に特定できないケースがあります。その理由は主に 2 つあり、ひとつは上記プロセスで得られる「クエリごとのフライトレコード」や「クエリ実行プラン」がキャッシュ上に存在する情報であることです。つまり、タイミングによっては欲しい情報がキャッシュアウトによって消失していることがあります。もうひとつは、ここで得られる「クエリ実行プラン」が、正確には「推定のクエリ実行プラン」であることです。つまり、実際に実行されたクエリ実行プラン（「実際のクエリ実行プラン」）とは異なっていることがあります。

このように、性能問題の原因究明に重要な「実際のクエリ実行プラン」を確認する手段として、SET STATISTICS XML、SET STATISTICS PROFILE、ライブクエリ統計（sys.dm_exec_query_profiles）といったものが従来提供されていましたが、問題のクエリを手元で実行する必要があったり、そもそも稼働中のデータベースに対して問題のクエリを実行すること自体が容易ではないケースもあります。かといって、本番と同等のテスト環境を準備することも難しいといったような困難がつきまといます。また別のアプローチとして SQL Server プロファイラー（Showplan XML）や拡張イベント

（query_post_execution_showplan）を使用した不特定のクエリに対して網を張ることもできますが、フットプリントが大きいためこれも本番のデータベースに対しての適用は容易ではありません。

こうした困難を背景に、SQL Server 2014 SP2 で Lightweight Query Profiling（軽量クエリプロファイリング）が導入され「実際のクエリ実行プラン」をより軽量に採取できるようになり、SQL Server 2019 でさらに磨きがかかりました。

軽量クエリプロファイリング インフラストラクチャ V1

導入当初の V1 では、主に採取情報を絞る(CPU 情報をカットし行数情報と I/O 情報を収集する)ことで軽量化が図られました。機能は既定で無効となっており、拡張イベント（query_thread_profile）を通じて有効化します。

軽量クエリプロファイリング インフラストラクチャ V2

次に SQL Server 2016 SP1 のタイミングで V2 となり、作りの改善によってさらに軽量化されました。V1 と同様に機能は既定で無効となっており、トレースフラグ 7412 を通じて有効化します。またライブクエリプランを採取するための動的管理関数（sys.dm_exec_query_statistics_xml）の導入や、SQL Server 2016 SP2 CU3 のタイミングでは特定のクエリに絞って軽量クエリプロファイリングを有効化するためのクエリヒント（QUERY_PLAN_PROFILE）と拡張イベント（query_plan_profile）が導入されました。

軽量クエリプロファイリング インフラストラクチャ v3

そして SQL Server 2019 のタイミングで V3 となり、採取情報をさらに絞る（CPU 情報と I/O 情報をカットし行数情報を収集する)ことで軽量化が図られ、ついに既定で有効の機能となりました。またデータベーススコープ設定（LIGHTWEIGHT_QUERY_PROFILING）が導入され、データベースごとに有効・無効を制御することができるようになりました。さらに、動的管理関数（sys.dm_exec_query_plan_stats）とデータベーススコープ設定（LAST_QUERY_PLAN_STATS）の導入により、“最後の”「実際のクエリ実行プラン」が採取することができるようになったり、拡張イベント（query_post_execution_plan_profile）の導入により、不特定のクエリに対してより軽量な網を張ることができるようになります。

軽量クエリプロファイリングの詳細については、次の Docs を確認ください。

◇ クエリ プロファイリング インフラストラクチャ
https://docs.microsoft.com/ja-jp/sql/relational-databases/performance/query-profiling-infrastructure?view=sql-server-ver15

3. やってみよう

ここでは新たに追加された動的管理関数（sys.dm_exec_query_plan_stats）で「実際のクエリ実行プラン」を確認することによる性能チューニングの疑似体験と、新たに追加された拡張イベント（query_post_execution_plan_profile）を利用した「実際のクエリ実行プラン」のプロファイリングを行います。

＊ 「実際のクエリ実行プラン」を利用した性能チューニング

機能を有効化する

「実際のクエリ実行プラン」を利用するために、データベーススコープ設定（LAST_QUERY_PLAN_STATS）を有効化します。

Lightweight_Query_Profiling.sql

```
-- このハンズオンは WideWorldImportersDW を利用
USE WideWorldImportersDW;
GO
-- LAST_QUERY_PLAN_STATS を有効化
ALTER DATABASE SCOPED CONFIGURATION SET LAST_QUERY_PLAN_STATS = ON;
GO
-- 設定の確認
SELECT * FROM sys.database_scoped_configurations
WHERE name IN (N'LAST_QUERY_PLAN_STATS', N'LIGHTWEIGHT_QUERY_PROFILING');
GO
```

結果　メッセージ

	configuration_id	name	value	value_for_secondary	is_value_default
1	22	LIGHTWEIGHT_QUERY_PROFILING	1	NULL	1
2	24	LAST_QUERY_PLAN_STATS	1	NULL	0

なお LAST_QUERY_PLAN_STATS の機能は軽量クエリプロファイリングを利用するため、「LIGHTWEIGHT_QUERY_PROFILING」も有効化されている必要があります。

性能問題を擬似的に再現させるために統計情報を実体と乖離するよう更新する

クエリを実行する前に実際と乖離した行数でメタデータを更新しています。こうすることで不適切な実行プランが作成されることを誘導し、性能劣化を疑似的に発生させます。

Lightweight_Query_Profiling.sql

```
USE WideWorldImportersDW;
GO
-- インデックスの作成
CREATE NONCLUSTERED INDEX IX_Lineage_Key ON Fact.OrderHistory ([Lineage Key])
  INCLUDE ([Order Key], [Stock Item Key], [Description]);
GO
```

```
-- 疑似的に実際と乖離した行数でメタデータを更新
UPDATE STATISTICS Fact.OrderHistory WITH ROWCOUNT = 1;
GO
--データとクエリの実行プランのキャッシュをクリア（クリーンな状態での検証のため）
DBCC DROPCLEANBUFFERS;
GO
DBCC FREEPROCCACHE;
GO
```

性能問題となるクエリを実行する

性能劣化を疑似的に発生させます。

Lightweight_Query_Profiling.sql

```
USE WideWorldImportersDW;
GO
-- クエリ実行のパフォーマンス情報の表示を有効化
SET STATISTICS TIME ON;
GO
-- 性能問題となるクエリを実行
SELECT fo.[Order Key], fo.Description, si.[Lead Time Days]
FROM Fact.OrderHistory AS fo
  INNER HASH JOIN Dimension.[Stock Item] AS si
    ON fo.[Stock Item Key] = si.[Stock Item Key]
WHERE fo.[Lineage Key] = 9 AND si.[Lead Time Days] > 19;
GO
```

今回はクエリ実行時間が「46,701ms」でした。ここでは期待より大幅に時間を要したものと解釈し、性能チューニングの対象とするものとします。

「実際のクエリ実行プラン」を採取し性能劣化原因を分析する

新しく追加された動的管理関数（sys.dm_exec_query_plan_stats）を通じて、先程の性能劣化したクエリの「実際のクエリ実行プラン」を採取します。

Lightweight_Query_Profiling.sql

```
USE master;
GO
SELECT cp.cacheobjtype,cp.objtype, st.dbid, st.text, qps.query_plan
FROM sys.dm_exec_cached_plans AS cp
  CROSS APPLY sys.dm_exec_sql_text(plan_handle) AS st
  CROSS APPLY sys.dm_exec_query_plan_stats(plan_handle) AS qps
```

```
WHERE st.text LIKE '%SELECT%' AND st.dbid = DB_ID('WideWorldImportersDW');
GO
```

結果 メッセージ

	cacheobjtype	objtype	dbid	text	query_plan
1	Compiled Plan	Adhoc	6	-- 性能問題となるクエリを実行 SELECT fo.[Order Key], fo.Desc...	<ShowPlanXML xmlns="http://schemas.microsoft.com/...

確認対象クエリの「query_plan」列のリンクをクリックします。

［Index Seek］と［Hash Match］の間の矢印にカーソルを合わせて、実行プランのプロパティを表示してみます。

実際の行数	3702592
予測行数	1
行の推定サイズ	123 B
データの推定サイズ	123 B

統計情報を元に推定されている予測行数「1」と実際の行数「3,702,592」は大きく乖離していることがわかり、これが性能劣化の原因である可能性が高いと分析できます。

「推定のクエリ実行プラン」では実際の行数を確認することができないため、このように「実際のクエリ実行プラン」によって原因を特定できるケースは多々あります。

性能チューニングを行う

「実際のクエリ実行プラン」を確認した結果、今回の性能劣化はクエリ自体の問題ではなくメタデータの精度が悪いことが原因であると考えられます。よって今回の性能チューニングではメタデータが最新の状態となるよう更新します。

```
Lightweight_Query_Profiling.sql
```

```
USE WideWorldImportersDW;
GO
DBCC UPDATEUSAGE (WideWorldImportersDW,'Fact.OrderHistory') WITH COUNT_ROWS;
GO
```

チューニング結果を確認する

最初と同じ SELECT クエリを実行し効果を確認します。

Lightweight_Query_Profiling.sql

```
-- 古いクエリ実行プランが再利用されないようにクリア
DBCC DROPCLEANBUFFERS;
GO
DBCC FREEPROCCACHE;
GO
-- クエリ実行のパフォーマンス情報の表示を有効化
SET STATISTICS TIME ON;
GO
--クエリを再実行
USE WideWorldImportersDW;
GO
SELECT fo.[Order Key], fo.Description, si.[Lead Time Days]
FROM Fact.OrderHistory AS fo
  INNER HASH JOIN Dimension.[Stock Item] AS si
    ON fo.[Stock Item Key] = si.[Stock Item Key]
WHERE fo.[Lineage Key] = 9 AND si.[Lead Time Days] > 19;
GO
```

今回は「2,132ms」となり劣化時の「46,701ms」と比較し大幅に改善しました。

「実際のクエリ実行プラン」の情報をあらためて採取する

あらためて「実際のクエリ実行プラン」を採取します。

Lightweight_Query_Profiling.sql

```
USE master;
GO
SELECT cp.cacheobjtype,cp.objtype, st.dbid, st.text, qps.query_plan
FROM sys.dm_exec_cached_plans AS cp
  CROSS APPLY sys.dm_exec_sql_text(plan_handle) AS st
  CROSS APPLY sys.dm_exec_query_plan_stats(plan_handle) AS qps
WHERE st.text LIKE 'SELECT%' AND st.dbid = DB_ID('WideWorldImportersDW');
GO
```

結果 | メッセージ

	cacheobjtype	objtype	dbid	text	query_plan
1	Compiled Plan	Adhoc	6	SELECT fo.[Order Key], fo.Description, si.[Lead ...	<ShowPlanXML xmlns="http://schemas.microsoft.com/...

劣化時のクエリ実行プランと比較し大きく変わっていることがわかります。

なお、「実際のクエリ実行プラン」は“最後の“「実際のクエリ実行プラン」になるため、過去にさかのぼって”以前の”「実際のクエリ実行プラン」を確認することはできないことに留意してください。過去にさかのぼって「実際のクエリ実行プラン」を履歴で確認するためには、次のハンズオンで実施する拡張イベント（query_post_execution_plan_profile）を利用します。

✱ 「実際のクエリ実行プラン」のプロファイリング

拡張イベントを定義する

新たに追加された拡張イベント（query_post_execution_plan_profile）を定義し開始します。

Lightweight_Query_Profiling.sql

```
USE master;
GO

-- 拡張イベントの定義（database_name = N'WideWorldImportersDW'で収集対象をフィルタ）
CREATE EVENT SESSION QueryPlanLWP ON SERVER
ADD EVENT sqlserver.query_post_execution_plan_profile (
  SET collect_database_name = (1)
  ACTION (
    sqlos.task_time, sqlserver.database_id, sqlserver.database_name,
    sqlserver.query_hash_signed, sqlserver.query_plan_hash_signed, sqlserver.sql_text
  ) WHERE (database_name = N'WideWorldImportersDW')
) ADD TARGET package0.event_file (SET FILENAME = N'C:¥work¥QueryPlanLWP.xel')
WITH (
  MAX_MEMORY = 4096 KB, EVENT_RETENTION_MODE = ALLOW_SINGLE_EVENT_LOSS,
  MAX_DISPATCH_LATENCY = 30 SECONDS, MAX_EVENT_SIZE = 0 KB,
  MEMORY_PARTITION_MODE = NONE, TRACK_CAUSALITY = OFF, STARTUP_STATE = OFF
);
GO
```

```
-- 定義した拡張イベントの開始
ALTER EVENT SESSION QueryPlanLWP ON SERVER STATE = START;
GO
```

拡張イベントセッションのログは「FILENAME」に指定した「C:¥work」フォルダに出力するよう構成しています。環境に応じて適宜読み替えてください。

前の手順で実行したクエリをあらためて実行する

今回は先に解説した"「実際のクエリ実行プラン」を利用した性能チューニング"で実行したクエリをあらためて実行します。

Lightweight_Query_Profiling.sql

```
USE WideWorldImportersDW;
GO
-- インデックスの作成
IF NOT EXISTS (SELECT * FROM sys.indexes WHERE name = N'IX_Lineage_Key')
BEGIN
  CREATE NONCLUSTERED INDEX IX_Lineage_Key ON Fact.OrderHistory ([Lineage Key])
    INCLUDE ([Order Key], [Stock Item Key], [Description]);
END;
GO

--疑似的に実際と乖離した行数でメタデータを更新
UPDATE STATISTICS Fact.OrderHistory WITH ROWCOUNT = 1;
GO

--データとクエリの実行プランのキャッシュをクリア（クリーンな状態での検証のため）
DBCC DROPCLEANBUFFERS;
GO
DBCC FREEPROCCACHE;
GO

-- 性能問題となるクエリを実行
SELECT fo.[Order Key], fo.Description, si.[Lead Time Days]
FROM Fact.OrderHistory AS fo
  INNER HASH JOIN Dimension.[Stock Item] AS si
    ON fo.[Stock Item Key] = si.[Stock Item Key]
WHERE fo.[Lineage Key] = 9 AND si.[Lead Time Days] > 19;
GO

-- 「実際のクエリ実行プラン」を確認
USE master;
GO
SELECT cp.cacheobjtype,cp.objtype, st.dbid, st.text, qps.query_plan
FROM sys.dm_exec_cached_plans AS cp
  CROSS APPLY sys.dm_exec_sql_text(plan_handle) AS st
  CROSS APPLY sys.dm_exec_query_plan_stats(plan_handle) AS qps
WHERE st.text LIKE '%SELECT%' AND st.dbid = DB_ID('WideWorldImportersDW');
GO

-- 性能チューニング（メタデータが最新の状態となるよう更新）
USE WideWorldImportersDW;
```

```
GO
DBCC UPDATEUSAGE (WideWorldImportersDW,'Fact.OrderHistory') WITH COUNT_ROWS;
GO

--クエリをあらためて実行
SELECT fo.[Order Key], fo.Description, si.[Lead Time Days]
FROM Fact.OrderHistory AS fo
  INNER HASH JOIN Dimension.[Stock Item] AS si
    ON fo.[Stock Item Key] = si.[Stock Item Key]
WHERE fo.[Lineage Key] = 9 AND si.[Lead Time Days] > 19;
GO

-- 「実際のクエリ実行プラン」を確認
USE master;
GO
SELECT cp.cacheobjtype,cp.objtype, st.dbid, st.text, qps.query_plan
FROM sys.dm_exec_cached_plans AS cp
  CROSS APPLY sys.dm_exec_sql_text(plan_handle) AS st
  CROSS APPLY sys.dm_exec_query_plan_stats(plan_handle) AS qps
WHERE st.text LIKE '%SELECT%' AND st.dbid = DB_ID('WideWorldImportersDW');
GO
```

拡張イベントセッションのログから「実際のクエリ実行プラン」を確認する

SSMS のオブジェクトエクスプローラーから［管理］-［拡張イベント］-［セッション］-［QueryPlanLWP］を展開し、［package0.event_file］を右クリックして［ターゲットデータの監表示］を選択すると、拡張イベントセッションのログを表示することができます。

上記図のように、拡張イベントセッションが開始された後に実行されたクエリの実際の実行プランの情報が query_post_execution_plan_profile イベントによって収集され、拡張イベントログファイルに蓄積されていることがわかります。sys.dm_exec_query_plan_stats は“最後の“「実際のクエリ実行プラン」が確認できる一方で、query_post_execution_plan_profile は「実際のクエリ実行プラン」を履歴としてログに残るため、過去にさかのぼって「実際のクエリ実行プラン」を確認することができます。

4.2. Plan cache

➢ Plan cache granular control

1. 概要

ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE で plan_handle を指定できるようなり、クエリごとにプロシージャキャッシュをクリアすることができるようになります。

2. 利用シーン

従来の ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE は plan_handle を指定できず、プロシージャキャッシュ全体をクリアすることしかできませんでした。
もしくは DBCC FREEPROCCACHE コマンドによって plan_handle、sql_handle、pool_name を指定した個別のプロシージャキャッシュクリアを行うこともできますが、ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE に比較し強い権限が必要であったり、Azure SQL Database では DBCC が利用できないといった留意が必要でした。
SQL Server 2019 では ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE で plan_handle を指定できるようなり、クエリごとにプロシージャキャッシュをクリアすることができるようになります。

3. やってみよう

それでは、実際に ALTER DATABASE SCOPED CONFIGURATION コマンドを使って特定の実行プランの情報のみをクリアする動作を見てみましょう。

ハンズオン用の環境を準備する

ハンズオンで使用するデータベースとオブジェクトを準備します。

Plan_cache_granular_control.sql

```
-- サンプルデータベース DB01 を作成
CREATE DATABASE DB01;
GO
USE DB01;
GO
-- DB01 にテーブルを作成しデータを挿入
CREATE TABLE Table11 (c1 int, c2 varchar(10));
INSERT INTO Table11 VALUES (1101, 'test1101'), (1102, 'test1102');
GO
CREATE TABLE Table12 (c1 int, c2 varchar(10));
INSERT INTO Table12 VALUES (1201, 'test1201'), (1202, 'test1202');
GO
-- DB01 にストアドプロシージャを作成
CREATE PROCEDURE SelectTable11 AS
  SELECT * FROM Table11;
GO
CREATE PROCEDURE SelectTable12 AS
```

```
  SELECT * FROM Table12;
GO
CREATE PROCEDURE SelectTable12c2 AS
  SELECT c2 FROM Table12;
GO

-- サンプルデータベース DB02 を作成
CREATE DATABASE DB02;
GO
USE DB02;
GO
-- DB02 にテーブルを作成しデータを挿入
CREATE TABLE Table21 (c1 int, c2 varchar(10));
INSERT INTO Table21 VALUES (2101, 'test100'), (2102, 'test101');
GO
-- DB02 にストアドプロシージャを作成
CREATE PROCEDURE SelectTable21 AS
  SELECT * FROM Table21;
GO
```

クエリを実行しプロシージャキャッシュを確認する

前の手順で作成したプロシージャを実行します。

Plan_cache_granular_control.sql

```
EXEC DB01..SelectTable11;
EXEC DB01..SelectTable12;
EXEC DB01..SelectTable12c2;
EXEC DB02..SelectTable21;
GO
```

プロシージャキャッシュを確認します。

Plan_cache_granular_control.sql

```
SELECT
  CONVERT(text, SUBSTRING(dest.text, (deqs.statement_start_offset/2) + 1,
    ((CASE WHEN deqs.statement_end_offset = -1
        THEN LEN(CONVERT(nvarchar(MAX), dest.text)) * 2
        ELSE deqs.statement_end_offset end - deqs.statement_start_offset)/2) + 1)
  ) AS query_text, deqs.plan_handle
FROM sys.dm_exec_query_stats AS deqs
  CROSS APPLY sys.dm_exec_sql_text (deqs.sql_handle) AS dest
WHERE dest.dbid = DB_ID(N'DB01') or dest.dbid = DB_ID(N'DB02')
OPTION (RECOMPILE);
GO
```

Results　Messages

	query_text	plan_handle
1	SELECT * FROM Table12	0x0500070041968625A097ADC05A…
2	SELECT * FROM Table21	0x05000800CF4D9E2320474CDF5A…
3	SELECT c2 FROM Table12	0x050007007ABA7A26109FADC05A…
4	SELECT * FROM Table11	0x0500070008729224309 0ADC05A…

それぞれのクエリがプロシージャキャッシュ内に確認できます。

特定のプロシージャキャッシュのみ削除する

今回は「SELECT * FROM Table12」のプロシージャキャッシュを削除します。該当クエリの「plan_handle」の値をコピーします。

Results　Messages

	query_text	plan_handle
1	SELECT * FROM Table12	0x0500070041968625A097ADC05A…
2	SELECT * FROM Table21	0x05000800CF4D9E2320474CDF5A…
3	SELECT c2 FROM Table12	0x050007007ABA7A26109FADC05A…
4	SELECT * FROM Table11	0x0500070008729224309 0ADC05A…

ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE の引数にコピーした「plan_handle」の値を指定します。

Plan_cache_granular_control.sql

```
USE DB01;
GO
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE 0x0500070041968625A097ADC05A
020000010000000000000000000000000000000000000000000000000000;
GO
```

プロシージャキャッシュから削除されたことを確認します。

Plan_cache_granular_control.sql

```
SELECT
  CONVERT(text, SUBSTRING(dest.text, (deqs.statement_start_offset/2) + 1,
    ((CASE WHEN deqs.statement_end_offset = -1
        THEN LEN(CONVERT(nvarchar(MAX), dest.text)) * 2
        ELSE deqs.statement_end_offset end - deqs.statement_start_offset)/2) + 1)
  ) AS query_text, deqs.plan_handle
FROM sys.dm_exec_query_stats AS deqs
  CROSS APPLY sys.dm_exec_sql_text (deqs.sql_handle) AS dest
WHERE dest.dbid = DB_ID(N'DB01') or dest.dbid = DB_ID(N'DB02')
OPTION (RECOMPILE);
GO
```

Results　Messages

	query_text	plan_handle
1	SELECT * FROM Table21	0x05000800CF4D9E2320474CDF5A…
2	SELECT c2 FROM Table12	0x050007007ABA7A26109FADC05A…
3	SELECT * FROM Table11	0x0500070008729224309 0ADC05A…

「SELECT * FROM Table12」のプロシージャキャッシュが削除されています。

特定のデータベースに関連するすべてのプロシージャキャッシュを削除する

ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE の引数に「plan_handle」を指定しない場合は該当データベースに関連するすべてのプロシージャキャッシュが削除されます。

Plan_cache_granular_control.sql

```
USE DB01;
GO
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
```

プロシージャキャッシュから削除されたことを確認します。

Plan_cache_granular_control.sql

```
SELECT
  CONVERT(text, SUBSTRING(dest.text, (deqs.statement_start_offset/2) + 1,
    ((CASE WHEN deqs.statement_end_offset = -1
        THEN LEN(CONVERT(nvarchar(MAX), dest.text)) * 2
        ELSE deqs.statement_end_offset end - deqs.statement_start_offset)/2) + 1)
  ) AS query_text, deqs.plan_handle
FROM sys.dm_exec_query_stats AS deqs
  CROSS APPLY sys.dm_exec_sql_text (deqs.sql_handle) AS dest
WHERE dest.dbid = DB_ID(N'DB01') or dest.dbid = DB_ID(N'DB02')
OPTION (RECOMPILE);
GO
```

Results　Messages

	query_text	plan_handle
1	SELECT * FROM Table21	0x05000800CF4D9E2320474CDF5A…

「DB01」データベースのプロシージャキャッシュはすべて削除されていることが確認できます（「SELECT * FROM Table12」は「DB02」データベースのプロシージャキャッシュです）。

4.3. Columnstore indexes

SQL Server 2019 での列ストアインデックス関連の強化は次の 2 点です。

Estimate compression for Columnstore indexes	データ圧縮機能による圧縮効果を推定するための sp_estimate_data_compression_savings ストアドプロシージャが列ストアインデックスにも対応し、これから作成する列ストアインデックスの推定サイズや、作成済みの列ストアインデックスをアーカイブ圧縮した際の推定サイズを事前に知ることができるようになります。
Columnstore stats in DBCC CLONEDATABASE	クエリチューニングに活用する DBCC CLONEDATABASE 機能が列ストアインデックスにも対応し、列ストアに関するトラブルシューティングの手段が強化されます。

➢ Estimate compression for Columnstore indexes

1. 概要

データ圧縮機能による圧縮効果を推定するための sp_estimate_data_compression_savings ストアドプロシージャが列ストアインデックスにも対応し、これから作成する列ストアインデックスの推定サイズや、作成済みの列ストアインデックスをアーカイブ圧縮した際の推定サイズを事前に知ることができるようになります。

2. 利用シーン

従来の sp_estimate_data_compression_savings ストアドプロシージャは行ストアのテーブルやインデックスを対象にデータ圧縮（行圧縮もしくはページ圧縮）の効果を確認するための機能でしたが、SQL Server 2019 で機能が拡張され列ストアにも対応します。
これにより、列ストアインデックスを作成した場合の推定サイズや、作成した列ストアインデックスを削除した場合の推定サイズを事前に知ることができるようになります。

なお sp_estimate_data_compression_savings ストアドプロシージャの引数に指定したソースオブジェクトの種類によって、推定サイズの算出に使用される列ストアインデックスの種類が決まります。
以下は@data_compression パラメーターに COLUMNSTORE または COLUMNSTORE_ARCHIVE を指定した場合の対応表です。例えば sp_estimate_data_compression_savings にヒープを指定した場合はクラスター化列ストアインデックスを作成した場合の推定サイズが返されます。

ソースオブジェクト	推定オブジェクト
ヒープ	クラスター化列ストアインデックス
クラスター化インデックス	クラスター化列ストアインデックス
非クラスター化インデックス	非クラスター化列ストアインデックスおよびテーブルのパーティション列
非クラスター化列ストアインデックス	非クラスター化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	クラスター化列ストアインデックス

逆に、作成したクラスター化列ストアインデックスを削除しヒープに戻す、もしくは作成した非クラスター化列ストアインデックスを非クラスター化インデックスとして再作成する場合についてもサイズを推定することができます。
以下は@data_compression パラメーターに NONE または ROW、PAGE を指定した場合の対応表です。例えば sp_estimate_data_compression_savings にクラスター化列ストアインデックスを指定した場合はヒープに戻った場合のテーブルの推定サイズが返されます。

ソースオブジェクト	推定オブジェクト
クラスター化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
非クラスター化列ストアインデックス	非クラスター化インデックス

sp_estimate_data_compression_savings ストアドプロシージャの詳細仕様については、次の Docs を確認ください。

◇ sp_estimate_data_compression_savings (Transact-sql)
… 列ストア インデックスに関する考慮事項
https://docs.microsoft.com/ja-jp/sql/relational-databases/system-stored-procedures/sp-estimate-data-compression-savings-transact-sql?view=sql-server-ver15#considerations-for-columnstore-indexes

3. やってみよう

それでは列ストアインデックスを作成した場合の推定サイズや、作成した列ストアインデックスを削除した場合の推定サイズを確認してみましょう。

ハンズオン用の環境を準備する

ハンズオンで使用するデータベースとオブジェクトを準備します。

Estimate_compression_for_Columnstore_indexes.sql

```
--サンプルデータベース DB01 を作成
CREATE DATABASE DB01;
GO

USE DB01;
GO
--WideWorldImportersDW データベースからデータをコピー
SELECT * INTO TABLE01 FROM WideWorldImportersDW.Fact.OrderHistoryExtended;
SELECT * INTO TABLE02 FROM WideWorldImportersDW.Fact.OrderHistoryExtended;
GO
--ヒープに非クラスター化インデックスを作成
CREATE NONCLUSTERED INDEX IX_Stock_Item_Key ON TABLE02 ([Stock Item Key]);
```

```
GO
```

「COLUMNSTORE」圧縮における推定サイズを確認する

sp_estimate_data_compression_savings を利用し、「COLUMNSTORE」圧縮を行った場合の推定サイズを確認します。

Estimate_compression_for_Columnstore_indexes.sql

```
USE DB01;
GO
-- ヒープ（index_id=0）にクラスター化列ストアインデックスを作成した場合の推定サイズを確認
EXEC sp_estimate_data_compression_savings 'dbo', 'TABLE01', 0, NULL, 'COLUMNSTORE';
GO
-- 非クラスター化インデックス(index_id=2)の列に非クラスター化列ストアインデックスを作成した場合の推定サイズを確認
EXEC sp_estimate_data_compression_savings 'dbo', 'TABLE02', 2, NULL, 'COLUMNSTORE';
GO
```

Results　Messages

	object_name	schema_name	index_id	partition_number	size_with_current_compression_setting(KB)	size_with_requested_compression_setting(KB)	sa
1	TABLE01	dbo	0	1	6043640	595712	4

	object_name	schema_name	index_id	partition_number	size_with_current_compression_setting(KB)	size_with_requested_compression_setting(KB)	sa
1	TABLE02	dbo	2	1	529232	97080	3

「size_with_current_compression_setting(KB)」はヒープおよび非クラスター化インデックスの現在のサイズを示し、「size_with_requested_compression_setting(KB)」はヒープにクラスター化列ストアインデックスを作成した場合の推定サイズ、および非クラスター化インデックスの列に非クラスター化列ストアインデックスを作成した場合の推定サイズを示します。

	size_with_current_compression_setting(KB)	size_with_requested_compression_setting(KB)
ヒープ	6,043,640	595,712
非クラスター化インデックス列	529,232	97,080

「COLUMNSTORE_ARCHIVE」圧縮における推定サイズを確認する

同様に sp_estimate_data_compression_savings を利用して「COLUMNSTORE_ARCHIVE」圧縮を行った場合の推定サイズも確認してみます。

Estimate_compression_for_Columnstore_indexes.sql

```
USE DB01;
GO
-- ヒープ（index_id=0）に COLUMNSTORE_ARCHIVE でクラスター化列ストアインデックスを作成した場合の推定サイズを確認
```

```
EXEC sp_estimate_data_compression_savings
                    'dbo', 'TABLE01', 0, NULL, 'COLUMNSTORE_ARCHIVE';
GO
-- 非クラスター化インデックス(index_id=2)の列に COLUMNSTORE_ARCHIVE で非クラスター化列ストアインデックスを作成した場合の推定サイズを確認
EXEC sp_estimate_data_compression_savings
                    'dbo', 'TABLE02', 2, NULL, 'COLUMNSTORE_ARCHIVE';
GO
```

Results　Messages

	object_name	schema_name	index_id	partition_number	size_with_current_compression_setting(KB)	size_with_requested_compression_setting(KB)	sa
1	TABLE01	dbo	0	1	6043640	448456	3

	object_name	schema_name	index_id	partition_number	size_with_current_compression_setting(KB)	size_with_requested_compression_setting(KB)	sa
1	TABLE02	dbo	2	1	529232	94504	3

先に確認した列ストアインデックス作成後の推定サイズと同様に、アーカイブ圧縮による列ストアインデックス作成後の推定サイズが確認できます。

	size_with_current_compression_setting(KB)	size_with_requested_compression_setting(KB)
ヒープ	6,043,640	448,456
非クラスター化インデックス列	529,232	94,504

列ストアインデックスを作成しサイズを確認する

ヒープにクラスター化列ストアインデックス、非クラスター化インデックスの列に非クラスター化列ストアインデックスを作成し、作成された列ストアインデックスのサイズを確認します。

Estimate_compression_for_Columnstore_indexes.sql

```
USE DB01;
GO
-- ヒープ（index_id=0）にクラスター化列ストアインデックスを作成
CREATE CLUSTERED COLUMNSTORE INDEX CCI_TABLE01 ON TABLE01;
GO
-- 非クラスター化インデックス(index_id=2)の列に非クラスター化列ストアインデックスを作成
CREATE NONCLUSTERED COLUMNSTORE INDEX NCI_TABLE02 ON TABLE02 ([Stock Item Key]);
GO

-- 作成したクラスター化列ストアインデックスのサイズを確認
EXEC sp_estimate_data_compression_savings 'dbo', 'TABLE01', 1, NULL, 'COLUMNSTORE';
GO
-- 作成した非クラスター化列ストアインデックスのサイズを確認
EXEC sp_estimate_data_compression_savings 'dbo', 'TABLE02', 3, NULL, 'COLUMNSTORE';
GO
```

Results　Messages

	object_name	schema_name	index_id	partition_number	size_with_current_compression_setting(KB)	size_with_requested_compression_setting(KB)	sa
1	TABLE01	dbo	1	1	485056	483624	4

	object_name	schema_name	index_id	partition_number	size_with_current_compression_setting(KB)	size_with_requested_compression_setting(KB)	sa
1	TABLE02	dbo	3	1	102120	102120	6

先に確認した推定サイズと実際に作成された実サイズを比較すると次のとおり概ね近似値となっていることが確認できます。

	列ストアインデックス 推定サイズ（KB）	列ストアインデックス 実サイズ（KB）
ヒープ	595,712	485,056
非クラスター化インデックスの列	97,080	102,120

作成した列ストアインデックスを削除した場合の推定サイズを確認する

作成したクラスター化列ストアインデックスを削除しヒープに戻す場合と作成した非クラスター化列ストアインデックスを非クラスター化インデックスとして再作成する場合のサイズを推定します。

Estimate_compression_for_Columnstore_indexes.sql

```
USE DB01;
GO
-- クラスター化列ストアインデックス（index_id=1）を削除しヒープに戻した場合の推定サイズを確認
EXEC sp_estimate_data_compression_savings 'dbo', 'TABLE01', 1, NULL, 'NONE';
GO
-- 非クラスター化列ストアインデックス（index_id=3）を非クラスター化インデックスとして再作成する場合の推定サイズを確認
EXEC sp_estimate_data_compression_savings 'dbo', 'TABLE02', 3, NULL, 'NONE';
GO
```

Results　Messages

	object_name	schema_name	index_id	partition_number	size_with_current_compression_setting(KB)	size_with_requested_compression_setting(KB)	sa
1	TABLE01	dbo	1	1	485056	5916384	4

	object_name	schema_name	index_id	partition_number	size_with_current_compression_setting(KB)	size_with_requested_compression_setting(KB)	sa
1	TABLE02	dbo	3	1	102120	518720	7

この推定サイズと先に確認したヒープと非クラスターインデックスの実サイズを比較すると次のとおり概ね近似値となっていることが確認できます。

	ヒープ 推定サイズ（KB）	ヒープ 実サイズ（KB）
クラスター化列ストアインデックス	5,916,384	6,043,640

	非クラスター化インデックス推定サイズ（KB）	非クラスター化インデックス実サイズ（KB）
非クラスター化インデックス列	518,720	529,232

➢ Columnstore stats in DBCC CLONEDATABASE

1. 概要

クエリチューニングに活用する DBCC CLONEDATABASE 機能が列ストアインデックスにも対応し、列ストアに関するトラブルシューティングの手段が強化されます。

2. 利用シーン

DBCC CLONEDATABASE はクエリパフォーマンスの問題をトラブルシューティングするために必要な要素（スキーマや統計情報）のみを含むデータベースを軽量かつ高速にコピーします。ユーザーはこのデータベースのコピーを利用して「推定のクエリ実行プラン」を得ることでクエリチューニングを行います。

ただし、列ストアインデックスを含むデータベースの DBCC CLONEDATABASE には留意が必要でした。それは列ストアインデックスの統計情報が作成されるタイミングに起因するもので、行ストアインデックス場合はインデックスが作成されたタイミングで統計情報が更新される一方で列ストアインデックスについてはインデックスが作成されたタイミングでは統計情報が更新されない（≒永続化されない）ため、DBCC CLONEDATABASE を実行する場合はその事前にユーザーが手動で列ストアインデックスの統計情報を更新する（≒永続化する）必要がありました。

◇ Considerations when tuning your queries with columnstore indexes on clone databases
https://techcommunity.microsoft.com/t5/sql-server/considerations-when-tuning-your-queries-with-columnstore-indexes/ba-p/385294

SQL Server 2019 では DBCC CLONEDATABASE 機能が列ストアインデックスにも対応し、従来必要であった事前の列ストアインデックスの統計情報の更新（≒永続化）が不要になります。

3. やってみよう

ここではソースデータベースとクローンデータベースそれぞれで列ストアインデックスに関する「推定のクエリ実行プラン」を確認し、それらが同一であることを確認することで従来必要であった事前の列ストアインデックスの統計情報の更新（≒永続化）が不要になったことを確認します。

ハンズオン用環境を準備する

ハンズオンで使用するデータベースとオブジェクトを準備します。

Columnstore_stats_in_DBCC_CLONEDATABASE.sql

```
-- サンプルデータベースを作成
CREATE DATABASE DB01;
GO
USE DB01;
GO
```

```
-- WideWorldImportersDW データベースからデータをコピー
SELECT [City Key], [Customer Key], [Stock Item Key], [Order Date Key],
       [Quantity], [Unit Price], [Tax Rate], [Lineage Key]
INTO OrderHistoryExtended FROM WideWorldImportersDW.Fact.OrderHistoryExtended;
GO

-- クラスター化列ストアインデックスを作成
CREATE CLUSTERED COLUMNSTORE INDEX CCI_OrderHistoryExtended ON OrderHistoryExtended;
GO

-- 統計情報が作成されていることを確認
SELECT * FROM sys.stats WHERE object_id = OBJECT_ID(N'OrderHistoryExtended');
GO
```

結果 | メッセージ

	object_id	name	stats_id	auto_created	user_created	no_recompute	has_filt
1	581577110	CCI_OrderHistoryExtended	1	0	0	0	0

列ストアインデックスを作成する際に指定した名前の統計情報オブジェクトが作成されていることが確認できます。

ソースデータベースで「推定のクエリ実行プラン」を確認する

SSMS のメニューから［クエリ］-［推定実行プランの表示］を選択して、次のクエリの「推定のクエリ実行プラン」を表示します。

Columnstore_stats_in_DBCC_CLONEDATABASE.sql

```
SELECT [Tax Rate], [Lineage Key], SUM(Quantity) AS SUM_QTY,
       SUM([Unit Price]) AS SUM_BASE_PRICE, COUNT(*) AS COUNT_ORDER
FROM OrderHistoryExtended
WHERE [Order Date Key] <= DATEADD(dd, -73, '2015-11-13')
GROUP BY [Tax Rate], [Lineage Key]
ORDER BY [Tax Rate], [Lineage Key];
GO
```

ここでは［実行プラン］タブの一番右端の「列ストア インデックス スキャン（Clustered)」オペレーターに対して、マウスカーソルを合わせてプロパティ情報を確認します。

列ストア インデックス スキャン (Clustered)
列ストア インデックスを全体的に、または特定の範囲だけスキャンします。

物理操作	列ストア インデックス スキャン
論理操作	Clustered Index Scan
推定実行モード	Batch
ストレージ	ColumnStore
操作の推定コスト	3.07746 (27%)
I/O の推定コスト	1.44831
サブツリーの推定コスト	3.07746
CPU の推定コスト	1.62915
予測実行回数	1
予測行数	22716900
読み取り対象の予測行数	29620700
行の推定サイズ	27 B
順序付け	False
ノード ID	4

述語
[DB01].[dbo].[OrderHistoryExtended].[Order Date Key] <='2015-09-01 00:00:00.000'
オブジェクト
[DB01].[dbo].[OrderHistoryExtended].[CCI_OrderHistoryExtended]
出力一覧
[DB01].[dbo].[OrderHistoryExtended].Quantity,
[DB01].[dbo].[OrderHistoryExtended].Tax Rate,
[DB01].[dbo].[OrderHistoryExtended].Lineage Key

このプロパティ情報を以降の手順で実行するクローンデータベースにおける「推定のクエリ実行プラン」のプロパティ情報と比較してみます。

なお、「推定のクエリ実行プラン」の元となった統計情報を確認してみます。

Columnstore_stats_in_DBCC_CLONEDATABASE.sql

```
USE DB01;
GO
--統計の状態確認
DBCC SHOW_STATISTICS('OrderHistoryExtended','CCI_OrderHistoryExtended')
WITH STATS_STREAM;
GO
SELECT sp.stats_id, name, last_updated FROM sys.stats AS stat
CROSS APPLY sys.dm_db_stats_properties(stat.object_id,stat.stats_id) AS sp
WHERE stat.object_id = object_id('OrderHistoryExtended');
GO
```

次のように列ストアインデックスの統計情報はこの時点で永続化されていないため、システムビューから確認できる統計情報の最終更新日時が NULL となっています。

結果 メッセージ

	Stats_Stream	Rows	Data Pages
1	0x01000000080000000000000000000005DAAD0550000000...	29620736	14394

	stats_id	name	last_updated
1	1	CCI_OrderHistoryExtended	NULL
2	2	_WA_Sys_00000004_22AA2996	2020-03-31 12:07:41.1866667
3	3	_WA_Sys_00000007_22AA2996	2020-03-31 12:07:41.6266667
4	4	_WA_Sys_00000008_22AA2996	2020-03-31 12:07:41.9733333

従来はこの状態でクローンデータベースを作成すると列ストアインデックスの統計情報は正しくクローンされないため、事前に手動で統計情報を更新する（≒永続化する）必要がありました。

クローンデータベースを作成する

DB01 データベースのクローンデータベースを作成します。

Columnstore_stats_in_DBCC_CLONEDATABASE.sql

```
-- クローンデータベースを作成
DBCC CLONEDATABASE ('DB01', 'DB01_CLONE');
GO
-- ソースデータベースとクローンデータベースのサイズを確認
SELECT DB_NAME(database_id) AS DatabaseName, name, size * 8 / 1024 AS [FileSize(MB)]
FROM sys.master_files WHERE database_id IN (DB_ID(N'DB01'), DB_ID(N'DB01_CLONE'))
GO
```

結果 メッセージ

	DatabaseName	name	FileSize(MB)
1	DB01	DB01	1608
2	DB01	DB01_log	1608
3	DB01_CLONE	DB01	8
4	DB01_CLONE	DB01_log	8

15.0 RTM) | SERVER03¥Administrator... | DB01 | 00:00:01 | 4 行

クローンデータベースは軽量かつ高速にコピーされます。

クローンデータベースの統計情報の状況を確認する

クローンデータベースの上の統計情報を確認します。

Columnstore_stats_in_DBCC_CLONEDATABASE.sql

```
USE DB01_CLONE;
GO
--統計の状態確認
DBCC SHOW_STATISTICS('OrderHistoryExtended','CCI_OrderHistoryExtended')
```

```
WITH STATS_STREAM
GO
SELECT sp.stats_id, name, last_updated FROM sys.stats AS stat
CROSS APPLY sys.dm_db_stats_properties(stat.object_id,stat.stats_id) AS sp
WHERE stat.object_id = object_id('OrderHistoryExtended');
GO
```

結果 メッセージ

	Stats_Stream	Rows	Data Pages
1	0x01000000080000000000000000000000C67DC5860000000...	29620736	14451

	stats_id	name	last_updated
1	1	CCI_OrderHistoryExtended	2020-03-31 12:10:13.6666667
2	2	_WA_Sys_00000004_22AA2996	2020-03-31 12:07:41.1866667
3	3	_WA_Sys_00000007_22AA2996	2020-03-31 12:07:41.6266667
4	4	_WA_Sys_00000008_22AA2996	2020-03-31 12:07:41.9733333

統計情報が正しくクローンされていることが確認できます。

なお、SQL Server 2017 以前で同じ手順を実施すると次のようになります。

結果 メッセージ

	Stats_Stream	Rows	Data Pages
1	0x01000000080000000000000000000000009893DAEA0000000...	29620736	0

	stats_id	name	last_updated
1	1	CCI_OrderHistoryExtended	NULL
2	2	_WA_Sys_00000004_35BCFE0A	2020-03-31 12:17:00.9100000
3	3	_WA_Sys_00000007_35BCFE0A	2020-03-31 12:17:01.7566667
4	4	_WA_Sys_00000008_35BCFE0A	2020-03-31 12:17:02.4433333

SQL Server 2017 以前では統計情報が正しくクローンされていないことが確認できます。

クローンデータベースで「推定のクエリ実行プラン」を確認する

ソースデータベースと同様にクローンデータベースに対して、SSMS のメニューから［クエリ］-［推定実行プランの表示］を選択して、次のクエリの「推定のクエリ実行プラン」を表示します。

Columnstore_stats_in_DBCC_CLONEDATABASE.sql

```
SELECT [Tax Rate], [Lineage Key], SUM(Quantity) AS SUM_QTY,
       SUM([Unit Price]) AS SUM_BASE_PRICE, COUNT(*) AS COUNT_ORDER
FROM OrderHistoryExtended
WHERE [Order Date Key] <= DATEADD(dd, -73, '2015-11-13')
GROUP BY [Tax Rate], [Lineage Key]
ORDER BY [Tax Rate], [Lineage Key];
```

```
GO
```

［実行プラン］タブの一番右端の「列ストア インデックス スキャン（Clustered）」オペレーターに対して、マウスカーソルを合わせてプロパティ情報を確認します。

次のとおりソースデータベースで確認したプロパティ情報と同じ内容がクローンデータベースでも確認することができます。

■SQL Server 2019 ソースデータベース

列ストア インデックス スキャン (Clustered)

列ストア インデックスを全体的に、または特定の範囲だけスキャンします。

物理操作	列ストア インデックス スキャン
論理操作	Clustered Index Scan
推定実行モード	Batch
ストレージ	ColumnStore
操作の推定コスト	3.07746 (27%)
I/O の推定コスト	1.44831
サブツリーの推定コスト	3.07746
CPU の推定コスト	1.62915
予測実行回数	1
予測行数	22716900
読み取り対象の予測行数	29620700
行の推定サイズ	27 B
順序付け	False
ノード ID	4

述語
[DB01].[dbo].[OrderHistoryExtended].[Order Date Key] <='2015-09-01 00:00:00.000'

オブジェクト
[DB01].[dbo].[OrderHistoryExtended].[CCI_OrderHistoryExtended]

出力一覧
[DB01].[dbo].[OrderHistoryExtended].Quantity, [DB01].[dbo].[OrderHistoryExtended].Tax Rate, [DB01].[dbo].[OrderHistoryExtended].Lineage Key

■SQL Server 2019 クローンデータベース

列ストア インデックス スキャン (Clustered)

列ストア インデックスを全体的に、または特定の範囲だけスキャンします。

物理操作	列ストア インデックス スキャン
論理操作	Clustered Index Scan
推定実行モード	Batch
ストレージ	ColumnStore
操作の推定コスト	3.07746 (27%)
I/O の推定コスト	1.44831
サブツリーの推定コスト	3.07746
CPU の推定コスト	1.62915
予測実行回数	1
予測行数	22716900
読み取り対象の予測行数	29620700
行の推定サイズ	27 B
順序付け	False
ノード ID	4

述語
[DB01_CLONE].[dbo].[OrderHistoryExtended].[Order Date Key] <='2015-09-01 00:00:00.000'

オブジェクト
[DB01_CLONE].[dbo].[OrderHistoryExtended].[CCI_OrderHistoryExtended]

出力一覧
[DB01_CLONE].[dbo].[OrderHistoryExtended].Quantity, [DB01_CLONE].[dbo].[OrderHistoryExtended].Tax Rate, [DB01_CLONE].[dbo].[OrderHistoryExtended].Lineage Key

なお、SQL Server 2017 以前で同じ手順を実施すると次のようにソースデータベースで確認できる内容とは異なるため、クローンデータベースで確認できる内容はチューニングに活用することはできません。

■SQL Server 2017 ソースデータベース

列ストア インデックス スキャン (Clustered)
列ストア インデックスを全体的に、または特定の範囲だけスキャンします。

物理操作	列ストア インデックス スキャン
論理操作	Clustered Index Scan
推定実行モード	Batch
ストレージ	ColumnStore
操作の推定コスト	3.07894 (27%)
I/O の推定コスト	1.44979
サブツリーの推定コスト	3.07894
CPU の推定コスト	1.62915
予測実行回数	1
予測行数	22704000
読み取り対象の予測行数	29620700
行の推定サイズ	27 B
順序付け	False
ノード ID	4

述語
[DB01].[dbo].[OrderHistoryExtended].[Order Date Key] <='2015-09-01 00:00:00.000'
オブジェクト
[DB01].[dbo].[OrderHistoryExtended].[CCI_OrderHistoryExtended]
出力一覧
[DB01].[dbo].[OrderHistoryExtended].Quantity, [DB01].[dbo].[OrderHistoryExtended].Tax Rate, [DB01].[dbo].[OrderHistoryExtended].Lineage Key

■SQL Server 2017 クローンデータベース

列ストア インデックス スキャン (Clustered)
列ストア インデックスを全体的に、または特定の範囲だけスキャンします。

物理操作	列ストア インデックス スキャン
論理操作	Clustered Index Scan
推定実行モード	Batch
ストレージ	ColumnStore
操作の推定コスト	1.63227 (16%)
I/O の推定コスト	0.003125
サブツリーの推定コスト	1.63227
CPU の推定コスト	1.62915
予測実行回数	1
予測行数	22704000
読み取り対象の予測行数	29620700
行の推定サイズ	36 B
順序付け	False
ノード ID	4

述語
[DB01_CLONE].[dbo].[OrderHistoryExtended].[Order Date Key] <='2015-09-01 00:00:00.000'
オブジェクト
[DB01_CLONE].[dbo].[OrderHistoryExtended].[CCI_OrderHistoryExtended]
出力一覧
[DB01_CLONE].[dbo].[OrderHistoryExtended].Quantity, [DB01_CLONE].[dbo].[OrderHistoryExtended].Unit Price, [DB01_CLONE].[dbo].[OrderHistoryExtended].Tax Rate, [DB01_CLONE].[dbo].[OrderHistoryExtended].Lineage Key

4.4. Error messages

➢ Verbose truncation warnings

1. 概要

列のサイズを超過するデータ挿入、または更新が要求されたことを示すエラーメッセージが強化され、対象となったオブジェクトや切り捨てられたデータを確認できるようになります。

2. 利用シーン

データを挿入または更新する際、該当のデータがターゲットとなる列のサイズを超過している場合には Msg 8152「文字列データまたはバイナリ データが切り捨てられます。」が出力されますが、このメッセージからは対象となったオブジェクト（テーブル名、カラム名）や切り捨てられたデータを知ることができませんでした。

SQL Server 2019 ではこのメッセージが強化され、対象のオブジェクトや切り捨てられたデータが Msg 2628「テーブル'%.*ls'、列'%.*ls'で文字列またはバイナリデータが切り捨てられます。切り捨てられた値：'%.*ls'」として出力されるようになります。この変更はデータベースオプション VERBOSE_TRUNCATION_WARNINGS（既定値：ON）を OFF にすることで旧来の Msg 8152 に戻すこともできます。

なお、該当のメッセージ強化は SQL Server 2016 や SQL Server 2017 にもバックポートされており、トレースフラグ 460 によって有効化できます（既定値：OFF）。

3. やってみよう

それでは、実際に強化された新しいメッセージの出力を確認してみましょう。

ハンズオン用環境を準備する

ハンズオンで使用するデータベースとオブジェクトを準備します。

Verbose_truncation_warnings.sql

```
USE master;
GO
-- サンプルデータベース DB01 を作成
CREATE DATABASE DB01;
GO

USE DB01;
GO
-- VERBOSE_TRUNCATION_WARNINGS の設定値を確認
SELECT name, value FROM sys.database_scoped_configurations
WHERE name = N'VERBOSE_TRUNCATION_WARNINGS'
GO
-- DB01 にサンプルテーブル Table11 を作成
CREATE TABLE Table1 (c1 int, c2 varchar(10));
GO
```

Results　Messages

	name	value
1	VERBOSE_TRUNCATION_WARNINGS	1

データベースオプション「VERBOSE_TRUNCATION_WARNINGS」が「1」（＝有効）となっていることが確認できます。

新しいエラーメッセージの出力内容を確認する

テーブルに定義した列のバイト数を超過するデータ挿入を試行してクエリがエラーとなることを確認します。

Verbose_truncation_warnings.sql

```
USE DB01;
GO
-- 10 バイトで定義されている列に 13 バイトのデータを挿入
INSERT INTO Table1 VALUES (1101, '1234567890ABC');
GO
-- データを確認
SELECT * FROM Table1;
GO
```

Results　Messages

```
16:26:33    Started executing query at Line 1
            Commands completed successfully.
16:26:33    Started executing query at Line 3
            Msg 2628, Level 16, State 1, Line 4
            テーブル 'DB01.dbo.Table1'、列 'c2' で文字列またはバイナリ データが切り捨てられます。切り捨てられた値: '1234567890'。
            ステートメントは終了されました。
16:26:33    Started executing query at Line 6
            (0 rows affected)
            Total execution time: 00:00:00.007
```

Results　Messages

c1	c2

データの切り捨てが発生することが Msg 2628 として返され、該当のエラーメッセージから対象となったオブジェクト（テーブル名、カラム名）や切り捨てられたデータを確認できるようになっています。該当クエリはエラーになっているため切り捨てられたデータが登録されることはありません。

なお、次のように「ANSI_WARNINGS」オプションを OFF にすることで、データの切り捨てをエラーとせずに処理を継続させる制御も可能です。

Verbose_truncation_warnings.sql

```
SET ANSI_WARNINGS OFF
GO
USE DB01;
GO
-- 10 バイトで定義されている列に 13 バイトのデータを挿入
INSERT INTO Table1 VALUES (1101, '1234567890ABC');
GO
-- データを確認
SELECT * FROM Table1;
GO
```

```
Results    Messages
16:28:03    Started executing query at Line 1
            Commands completed successfully.
16:28:03    Started executing query at Line 3
            Commands completed successfully.
16:28:03    Started executing query at Line 5
            (1 row affected)
16:28:03    Started executing query at Line 8
            (1 row affected)
            Total execution time: 00:00:00.007
```

Results　Messages

	c1	c2
1	1101	1234567890

データの切り捨てはエラーにならず INSERT は正常終了し、切り捨てられたデータが登録されていることが確認できます。

新しいエラーメッセージの出力を無効化する

無効化するには、データベースオプション「VERBOSE_TRUNCATION_WARNINGS」を OFF に設定します。

Verbose_truncation_warnings.sql

```
USE DB01;
GO
-- VERBOSE_TRUNCATION_WARNINGS の設定を無効化
ALTER DATABASE SCOPED CONFIGURATION SET VERBOSE_TRUNCATION_WARNINGS = OFF;
```

```
GO
-- VERBOSE_TRUNCATION_WARNINGS の設定値を確認
SELECT name, value FROM sys.database_scoped_configurations
WHERE name = N'VERBOSE_TRUNCATION_WARNINGS'
GO
```

Results　Messages

	name	value
1	VERBOSE_TRUNCATION_WARNINGS	0

先ほどと同じクエリを実行します。

Verbose_truncation_warnings.sql

```
USE DB01;
GO
-- 10 バイトで定義されている列に 13 バイトのデータを挿入
INSERT INTO Table1 VALUES (1101, '1234567890ABC');
GO
```

Messages

```
12:09:13    Started executing query at Line 1
            Commands completed successfully.
12:09:13    Started executing query at Line 3
            Msg 8152, Level 16, State 30, Line 4
            文字列データまたはバイナリ データが切り捨てられます。
            ステートメントは終了されました。
            Total execution time: 00:00:00.001
```

従来どおりデータの切り捨てが発生することが Msg 8152 として返されます。この場合、該当のエラーメッセージから対象となったオブジェクト（テーブル名、カラム名）や切り捨てられたデータを確認することはできません。

4.5. Page Information

➢ sys.dm_db_page_info

1. 概要

データページの詳細情報を取得するための新しい動的管理関数 sys.dm_db_page_info が追加されます。これにより、データページへのロックやラッチの競合もしくはページ破損など、より複雑な問題のトラブルシューティングやデバッグのための作業プロセスが簡素化されます。

2. 利用シーン

データページへのロックやラッチの競合もしくはページ破損など、より複雑な問題のトラブルシューティングやデバッグのためにはデータページの詳細情報を知る必要があります。こうしたケースに対して、従来は DBCC PAGE というデータページの詳細情報を確認するためのコマンドを利用していました。しかし、DBCC PAGE の利用には以下のような課題がありました。

- 該当コマンドは"Undocumented"である
- DBCC コマンドゆえに情報採取のための他のクエリとの結合ができず使い勝手が悪い

SQL Server 2019 では、データページの詳細情報を取得するための新しい動的管理関数 sys.dm_db_page_info が追加されます。重要なポイントはこの機能が動的管理関数として提供されるため、sys.dm_exec_requests や sys.sysprocesses などの他の情報と関連付けた情報採取が可能になることです。sys.dm_exec_requests や sys.sysprocesses には、新たに page_resource という列が追加されています。page_resource は新たに追加される関数 page_resource を通じて、データページの所在（database_id + file_id + page_id）を得ることができます。この所在を sys.dm_db_page_info に渡すことで、データページの詳細情報を取得することができます。詳細はこの先のハンズオンで紹介するクエリを確認してください。

なお、sys.dm_db_page_info は DBCC PAGE の完全な代替ではありません。DBCC PAGE はデータページ全体のさまざまな情報を取得できますが、sys.dm_db_page_info で採取できる情報はページのヘッダ情報に限られます。ただし、データページへのロックやラッチの競合もしくはページ破損などのトラブルシューティングやデバッグのための貴重な情報を得ることができ、より詳細な情報が必要なケースは DBCC PAGE を活用するというように初動調査を簡素化できます。

3. やってみよう

ここでは擬似的に再現させたロック競合やラッチ競合に対して、新しい動的管理関数 sys.dm_db_page_info を利用した競合対象の特定を実践してみます。

ハンズオン用の環境を準備する

ハンズオンに必要なデータベース、テーブル、ストアドプロシージャを作成します。

sys_dm_db_page_info_and_fn_PageResCracker.sql

```
CREATE DATABASE DB01;
GO

USE DB01;
GO
CREATE TABLE ProductTable (
  product_id int IDENTITY(1,1) NOT NULL PRIMARY KEY CLUSTERED,
  product_name sysname NOT NULL
);
GO
INSERT INTO ProductTable VALUES (N'Pencil'), (N'Note');
GO

CREATE TABLE StockTable (
  rec_id     int IDENTITY(1,1) NOT NULL PRIMARY KEY CLUSTERED,
  product_id int NOT NULL,
  stock_num  int NOT NULL
);
GO
INSERT INTO StockTable VALUES (1, 100), (2, 50);
GO

CREATE PROCEDURE CreateTmpTable AS
  CREATE TABLE #tmpTable (c1 int);
GO
```

ロック競合となった対象ページの詳細情報を取得する

1. ロック競合を擬似的に再現させます。

新しいセッションを開き次のトランザクション A を実行します。

sys_dm_db_page_info_and_fn_PageResCracker.sql

```
-- トランザクション A（ここでは明示的にページロックを要求しています）
USE DB01;
GO
BEGIN TRAN
  UPDATE StockTable WITH (PAGLOCK) SET stock_num = 80 WHERE product_id = 1;
GO
```

別の新しいセッションを開き次のトランザクション B を実行します。

sys_dm_db_page_info_and_fn_PageResCracker.sql

```
-- トランザクション B（ここでは明示的にページロックを要求しています）
USE DB01;
GO
BEGIN TRAN
  UPDATE StockTable WITH (PAGLOCK) SET stock_num = 50 WHERE product_id = 1;
GO
```

このトランザクション B は先発のトランザクション A と処理対象のページが競合するため、トランザクション A が完了するまで待機状態となります。

2. ロック競合となった対象ページの詳細情報を確認します。

このハンズオンでは、トランザクション B のクエリは条件が即値で与えられているため競合の対象は自明ですが、副問合せや結合など複雑な条件が与えられた場合などクエリからは競合対象を特定することが困難であるケースであると見立て、新しい動的管理関数 sys.dm_db_page_info を利用して競合ページを特定してみます。

トランザクション A、B とは異なる別の新しいセッションを開き次のクエリを実行します。

sys_dm_db_page_info_and_fn_PageResCracker.sql

```
USE DB01;
GO
SELECT OBJECT_NAME(ddpi.object_id) AS object_name, der.wait_resource,
       der.wait_type, der.total_elapsed_time,
       SUBSTRING(dest.text, (deqs.statement_start_offset / 2) + 1,
        ((CASE deqs.statement_end_offset
          WHEN -1 THEN DATALENGTH(dest.text)
         ELSE deqs.statement_end_offset
         END - deqs.statement_start_offset) / 2) + 1) AS query_text,
       ddpi.*
FROM sys.dm_exec_requests AS der
  LEFT OUTER JOIN sys.dm_exec_query_stats AS deqs
    ON der.sql_handle = deqs.sql_handle
  CROSS APPLY sys.fn_PageResCracker(der.page_resource) AS fp
  CROSS APPLY sys.dm_db_page_info(fp.db_id, fp.file_id, fp.page_id, N'DETAILED')
    AS ddpi
  CROSS APPLY sys.dm_exec_sql_text(der.sql_handle) AS dest
GO
```

Results　Messages

	object_name	wait_resource	wait_type	total_elapsed_time	query_text	database_id	file_id	page_id	pag
1	StockTable	PAGE: 7:1:240	LCK_M_U	155254	UPDATE [StockTable] WITH(pag…	7	1	240	1

このようにロック競合となったセッションのリクエスト情報や競合対象のページ情報が確認でき、トラブルシューティングに役立たせることができます。

次の手順に進む前に、トランザクション A および B のセッションを終了させます。

ラッチ競合となった対象ページの詳細情報を取得する

1. ラッチ競合を擬似的に再現させます。

ページラッチ待機を疑似的に発生させるため、ostress ツールを使用します。SQL Server 2019 自習書「Introduction」-「クライアントツールのインストール」に従って、以降の手順を実施する環境へあらかじめインストールしておく必要があります。
ostress ツールを準備した後、コマンドプロンプトから次のコマンドを実行します。

sys_dm_db_page_info_and_fn_PageResCracker.bat.txt

```
"C:¥Program Files¥Microsoft Corporation¥RMLUtils¥ostress.exe" -n100 -r30000 -Slocal
host -Usa -PPassw0rd -dDB01 -Q"exec CreateTmpTable"
```

サーバー名（-Slocalhost）、ユーザー名（-Usa）、パスワード（-PPassw0rd）は環境に合わせて適宜変更してください。

コマンドの実行後、コンソールに出力されたメッセージの末尾に「Worker threads created, beginning execution...」というメッセージが表示されれば、正常に実行開始されています。

2. ラッチ競合となった対象ページの詳細情報を確認します。

ここでは、システム全体でスループットが大幅に劣化している状況で、またラッチが疑わしいケースであると見立て、新しい動的管理関数 sys.dm_db_page_info を利用して競合ページを特定してみます。

sys_dm_db_page_info_and_fn_PageResCracker.sql

```
USE tempdb;
GO
SELECT OBJECT_NAME(ddpi.object_id) AS object_name, der.wait_resource,
       der.wait_type, der.total_elapsed_time, ddpi.*
FROM sys.dm_exec_requests AS der
  LEFT OUTER JOIN sys.dm_exec_query_stats AS deqs
    ON der.sql_handle = deqs.sql_handle
  CROSS APPLY sys.fn_PageResCracker(der.page_resource) AS fp
  CROSS APPLY sys.dm_db_page_info(fp.db_id, fp.file_id, fp.page_id, N'DETAILED')
    AS ddpi
  CROSS APPLY sys.dm_exec_sql_text(der.sql_handle) AS dest
GO
```

Results　Messages

	object_name	wait_resource	wait_type	total_elapsed_time	database_id	file_id	page_id	page_header_version	page_type	page_type_desc	pa
1	sysschobjs	2:1:328	PAGELATCH_EX	39	2	1	328	1	2	INDEX_PAGE	0
2	sysschobjs	2:1:328	PAGELATCH_EX	26	2	1	328	1	2	INDEX_PAGE	0
3	sysschobjs	2:1:328	PAGELATCH_EX	35	2	1	328	1	2	INDEX_PAGE	0
4	sysschobjs	2:1:328	PAGELATCH_EX	57	2	1	328	1	2	INDEX_PAGE	0
5	sysschobjs	2:1:328	PAGELATCH_EX	35	2	1	328	1	2	INDEX_PAGE	0
6	sysschobjs	2:1:328	PAGELATCH_EX	34	2	1	328	1	2	INDEX_PAGE	0
7	sysschobjs	2:1:328	PAGELATCH_EX	36	2	1	328	1	2	INDEX_PAGE	0
8	sysschobjs	2:1:328	PAGELATCH_EX	34	2	1	328	1	2	INDEX_PAGE	0
9	sysschobjs	2:1:328	PAGELATCH_EX	41	2	1	328	1	2	INDEX_PAGE	0

得られた情報から、tempdb の sysschobjs に対して PAGELATCH_EX の待機が多数発生していることが特定できます。今回のハンズオンクエリでは、tempdb に対して多数のオブジェクトを多重セッションで作成する処理を実行しており、該当クエリがラッチ競合の原因であると考えることができます。

SQL Server 2019 自習書「1. Intelligent Performance」の以下項目のハンズオンにおいても、ラッチ競合のトラブルシューティングで sys.dm_db_page_info を活用していますので、併せてご確認ください。

- ▶ Memory-Optimized TempDB Metadata
- ▶ Concurrent PFS updates

4.6. Query Store

クエリストアは性能チューニングツールとして SQL Server 2016 で実装されました。クエリストアはクエリ実行プランや稼働統計情報などクエリのフライトレコードを履歴で蓄積します。ユーザーは蓄積された履歴からクエリ実行プランの変化のタイミングや、その前後のクエリ実行プランや処理時間の相違を把握することでトラブルシュートに活用します。また、専用のストアドプロシージャ（sp_query_store_force_plan）を通じて、履歴内のクエリ実行プランを強制的に適用することもできるようになり、従来あったクエリヒント（USE PLAN）によるクエリ実行プラン強制に比較し格段に使い勝手が向上しました。
さらに SQL Server 2017 ではクエリごとの待機統計情報もキャプチャできるようになり、前後の相違をより詳細に把握することができるようになりました。
そして SQL Server 2019 では次の 2 つの機能強化が入ります。 1 つは STATIC カーソルと FAST_FORWARD カーソルがクエリ実行プラン強制に対応することです。もうひとつは新たに追加されたオプション（QUERY_CAPTURE_POLICY）によってクエリストアによる情報収集動作をより詳細に制御することができるようになることです。

クエリストアの詳細については、次の Docs を確認ください。

◇ クエリのストアを使用した、パフォーマンスの監視
https://docs.microsoft.com/ja-jp/sql/relational-databases/performance/monitoring-performance-by-using-the-query-store?view=sql-server-ver15

以降は新たに追加されたオプション（QUERY_CAPTURE_POLICY）を解説します。

➢ Custom capture policy for Query Store（QUERY_CAPTURE_POLICY）

1. 概要

QUERY_CAPTURE_POLICY オプションによりクエリストアでキャプチャする情報を細かく指定できるようになり、キャプチャ処理のオーバーヘッドや蓄積データの量を制御できるようになります。

2. 利用シーン

SQL Server 2017 以前は QUERY_CAPTURE_MODE オプションの ALL、AUTO、NONE でキャプチャを制御していましたが、SQL Server 2019 では新たに CUSTOM を設定できるようになります。

ALL	すべてのクエリをキャプチャします。SQL Server 2017 以前の既定値です。
AUTO	実行の数とリソースの消費量に基づいて関連するクエリをキャプチャします。SQL Server 2019 の既定値です。
NONE	新たなクエリのキャプチャは停止し、既にキャプチャ済みのクエリのみキャプチャを継続し

	ます。
CUSTOM	QUERY_CAPTURE_POLICY オプションに従ってキャプチャします。

QUERY_CAPTURE_MODE オプションで CUSTOM を指定する場合、併せて QUERY_CAPTURE_POLICY オプションで各パラメーターを設定します。STALE_CAPTURE_POLICY_THRESHOLD を除き、各パラメーターで設定する閾値のいずれかを超過したクエリがキャプチャの対象となります。

STALE_CAPTURE_POLICY_THRESHOLD	下記のイベント発生による累積値をリセットする期間を定義します。既定値は 24 HOURS です。
EXECUTION_COUNT	クエリ実行回数を定義します。 既定値は 30 です。
TOTAL_COMPILE_CPU_TIME_MS	クエリコンパイルによる CPU 合計経過時間を ms 単位で定義します。 既定値は 1000 です。
TOTAL_EXECUTION_CPU_TIME_MS	クエリ実行による CPU 合計経過時間を ms 単位で定義します。 既定値は 100 です。

このように、重要ではないクエリをキャプチャ対象から除外することで、キャプチャ処理のオーバーヘッドや蓄積データの量を制御します。
これらのオプションを指定してクエリストアを有効化するクエリのサンプルは以下のようになります。

```
ALTER DATABASE DB01 SET QUERY_STORE = ON (
  OPERATION_MODE = READ_WRITE,
  QUERY_CAPTURE_MODE = CUSTOM,
  QUERY_CAPTURE_POLICY = (
    STALE_CAPTURE_POLICY_THRESHOLD = 24 HOURS,
    EXECUTION_COUNT = 30,
    TOTAL_COMPILE_CPU_TIME_MS = 1000,
    TOTAL_EXECUTION_CPU_TIME_MS = 100
  )
);
GO
```

4.7. Statistics

➢ Diagnostics on statistics （WAIT_ON_SYNC_STATISTICS_REFRESH, SELECT（STATMAN））

1．概要

sys.dm_os_wait_stats 動的管理ビューに追加された新しい待機の種類 WAIT_ON_SYNC_STATISTICS_REFRESH によって統計情報の更新に伴うユーザークエリの待機時間をインスタンスレベルで把握することができるようになります。

2．利用シーン

クエリの性能を決定づけるクエリ実行プランは統計情報を元に作成されるため、「統計情報の鮮度（と精度）」を保つことはクエリの性能に対して非常に重要です。
「統計情報の鮮度」はユーザーのクエリ実行の延長で SQL Server が独自のロジックによって判定しており、もし古いと判断した場合には「統計情報の更新」を行います。
「統計情報の更新」は既定ではユーザーのクエリ実行と同期されるため、ユーザーのクエリ実行は「統計情報更新の更新」を待ち合わせる必要があります（データベースオプション AUTO_UPDATE_STATISTICS_ASYNC を有効化することで、ユーザーのクエリ実行を待機させずに非同期で統計情報を更新させることもできますが、統計情報が古いために望ましいクエリ実行プランが生成されないケースが生じうることとのトレードオフが必要です）。

SQL Server 2019 では「統計情報更新を同期するための待機」に費やされた時間をインスタンスレベルで把握することができるようになります。「統計情報更新を同期するための待機」はユーザークエリにとっては予期せぬ性能劣化として表面化するため望ましくありません。データベース管理者はこの待機時間が目立つ場合には統計情報の手動メンテナンスを計画することでユーザークエリの予期せぬ性能劣化を回避することに役立たせることができます。
また、sys.dm_exec_requests 動的管理ビューの command 列値に“SELECT（STATMAN）“が出力されることを監視することで今現在「統計情報更新を同期するための待機」を要しているクエリが特定できるようになります。

4.8. Resource Governor

➢ Memory grant percent in Resource Governor

1. 概要

リソースガバナーの REQUEST_MAX_MEMORY_GRANT_PERCENT に指定できる値範囲を float の精度で指定できるようになり、よりきめ細かいメモリリソース制御が可能になります。

2. 利用シーン

リソースガバナーの REQUEST_MAX_MEMORY_GRANT_PERCENT（単一の要求がプールから取得できるメモリの最大量を指定するオプション）に指定できる値範囲は従来 1～100 でしたが、SQL Server 2019 では float の精度で指定することができるようになり、よりきめ細かいメモリリソース制御が可能になります。

4.9. Setup options

➢ MAXDOP and Memory Config during Setup

1. 概要

SQL Server の主要なチューニングパラメーターである max degree of parallelism、max server memory (KB)、min server memory (KB)の設定が Windows 版についてはセットアップウィザードから設定することができるようになります。

2. 利用シーン

従来、これらのサーバー構成オプションはセットアップ後に手動で設定を変更する必要がありましたが、SQL Server 2019 では、Windows 版についてのみセットアップウィザードから設定することができるようになります。また、単にセットアップウィザードから設定することができるようになっただけではなく、推奨値を示してくれるように改善されています。

- **max degree of parallelism**

  クエリオプティマイザーが並列実行可能と判断したクエリにおいて、並列実行の上限（使用する論理プロセッサの上限）を制限するパラメーターで、同時実行性や並列化された処理の同期オーバーヘッドを踏まえてチューニングします。

  SQL Server 2019 では Windows 版セットアップウィザードの「MaxDOP」タブから設定を行います。

既定ではサーバーの論理 CPU 数に基づいて「並列処理の最大次数（MaxDOP）」欄に推奨設定値が表示されますが、これを独自の値に変更することもできます。

- **max server memory および min server memory**

max server memory は SQL Server の Memory Manager によって管理されるメモリの上限を指定します。SQL Server は定期的に OS に対して物理メモリの使用状況を確認していますが、OS が物理メモリの空きが少ないと報告しない限りはキャッシュできるデータを多くするために必要に応じてメモリを確保し続けます（既定の動作の場合）。一方で OS からの報告で物理メモリの不足を検知しても使用中である場合には即座に解放できないため、このような場合にはページングが発生し SQL Server プロセスのワーキングセットが縮小されパフォーマンス悪化を招くことがあります。max server memory はこれを極力回避することを目標にチューニングします。

min server memory は最低限必要なワーキングセットを保証したい場合に設定します。

SQL Server 2019 では Windows 版セットアップウィザードの「メモリ」タブから max server memory と min server memory の設定を行います。

既定では、「既定」ラジオボタンがチェックされた状態です。推奨の値に変更するには「推奨」ラジオボタンにチェックします。また推奨値以外の独自の値に変更することもできます。

5. Enhanced user experience for Developers

このセクションは、主にデータベース開発者向けの機能強化として次の話題を取り上げます。

機能名	概要
Graph	いくつかの制限事項が解消され使い勝手が向上しています。
Unicode support	Unicode 符号化文字集合に対する文字符号化方式として従来の UTF-16 に加えて、新たに UTF-8 を選択することができるようになります。
Language extensions	言語拡張機能が新たに追加され Java コード実行と Transact-SQL クエリ実行が統合されます。
Spatial	GDA2020 に対応した新しい SRID（Spatial Reference Identifier）が追加されます。

5.1. Graph

グラフデータベースは SQL Server 2017 で実装された機能です。SQL Server に統合されているため、SQL Server が既に有しているパフォーマンス、セキュリティ、ハイアベイラビリティなどのリッチな機能の上でグラフデータベースを展開できるようになりました。

グラフデータベースはエンティティ間の複雑な関係をモデル化するもので、ソーシャルグラフ、リコメンデーション、経路検索、不正検知といったエンティティ間の関係が主な関心事となるシナリオに最適化された、「データ構造」「データ操作」「制約」を提供します。

従来のリレーショナルデータベースはエンティティ間の関係を外部キーによって実現し、結合によって答えを導きます。これは単純な関係を導くには十分に機能しますが、関連付ける関係が複数ある場合や、例えば友人の友人さらにその友人といったように、階層関係を辿る必要がある場合にはクエリが複雑化し処理時間も大きく長期化します。こうしたケースに対してグラフデータベースの場合はエンティティ間の「関係」を明示的に格納するため、単にその関係を辿ることで答えを導くことができます。

SQL Server のグラフデータベースではエンティティを「ノード」と表現し、エンティティの関係を「エッジ」と表現します。上の例ではノードとエッジは以下のようになります。

ノード		
	「人物」ノード	「A さん」「B さん」「C さん」「D さん」
	「お店」ノード	「Z カフェ」
エッジ		
	「友達」エッジ	「A さん->B さん」「A さん->C さん」「C さん->D さん」
	「お気に入り」エッジ	「A さん->Z カフェ」
	「所有」エッジ	「D さん->Z カフェ」

SQL Server のグラフデータベースは T-SQL 拡張を使用してグラフモデル内のノードとエッジを表すためにテーブルを使用しています。またここでの「データベース」という用語は論理的であり、ノードテーブルとエッジテーブル、およびそれらを結合するデータとメタデータの集合を指しています。

グラフデータベースの機能や使い方については、SQL Server 2017 の自習書でも解説されているので必要に応じて確認ください。

◇ SQL Server 2017 自習書 No.1「SQL Server 2017 新機能の概要」
http://download.microsoft.com/download/7/2/9/729DF466-7E91-4351-AD33-029B4232A9DF/SQL_2017_SelfLearning_Ver1.pdf

このグラフデータベースに対して、SQL Server 2019 では次のような機能強化が行われています。これらの機能について以降のページで詳細を解説します。

機能名	概要
Edge constraint in Graph Database	エッジに対して制約を定義できるよう強化が行われ、エッジに接続されるノードの制限や、ノード削除にともなうエッジへの CASCADE 処理を制御できるようになります。
New graph function - SHORTEST_PATH	SHORTES_PATH 関数によってノード間の最短パスを照会することができるようになります。
Use derived table or view aliases in graph match query	ノードやエッジを結合するビューや派生テーブルを定義できるようになり、グラフデータベースへのクエリを直感的シンプルに記述することができるようになります。
Using the T-SQL MERGE statement to support scenarios like upsert	ノードテーブルのみ対応していた MERGE 文がエッジテーブルにも対応します。

➢ Edge constraint in Graph Database

1. 概要

エッジに対して制約を定義できるよう強化が行われ、エッジに接続されるノードの制限やノードの削除にともなうエッジへの CASCADE 処理を制御できるようになります。

2. 利用シーン

SQL Server 2017 では任意のノード間を任意のエッジで接続することができたため、例えば「レストラン」から「人物」に向けて「お気に入り」というような、意味としてふさわしくない定義が可能でした。またエッジによる関係が存在する場合にも、それに関連するノードを削除することが可能でした。このように従来は操作の妥当性や整合性を開発者が意識的に制御する必要がありましたが、SQL Server 2019 でエッジ制約が導入されたことでエッジに接続されるノードの制限や、ノードの削除にともなうエッジへの CASCADE 処理を制御することができるようになります。

3. やってみよう

それでは、エッジに制約を作成したときの動作を実際に確認してみましょう。

ノードとエッジを定義する

3 つのノード「Person」「Location」「Product」と 1 つのエッジ「LiveIn」を定義します。

Edge_constraint_in_Graph_Database.sql

```
USE WideWorldImporters;
GO
-- ノードテーブルの作成
CREATE TABLE Person   (ID int PRIMARY KEY, name varchar(100)) AS NODE;
CREATE TABLE Location (ID int PRIMARY KEY, name varchar(100)) AS NODE;
CREATE TABLE Product  (ID int PRIMARY KEY, ProductName varchar(100)) AS NODE;
-- エッジテーブルの作成
CREATE TABLE LiveIn AS EDGE;
GO
```

エッジ制約を定義する

「LiveIn」エッジに対して「Person」ノードから「Location」ノードに向けての関係定義のみを許可します。またノード削除にともなう CASCADE 削除処理を有効にします。

Edge_constraint_in_Graph_Database.sql

```
USE WideWorldImporters;
GO
-- エッジ制約を作成
ALTER TABLE LiveIn ADD CONSTRAINT EC_LIVEIN1 CONNECTION (Person TO Location)
  ON DELETE CASCADE;
GO
```

コマンド指定内容	説明
CONNECTION (Person TO Location)	エッジテーブルで関連付けられるノードテーブルを指定しています。今回の場合、「Person」ノードから「Location」ノードに向かってのみ「LiveIn」エッジを作成できるということを示しています。
ON DELETE CASCADE	ノードを削除した時の動作を指定しています。「CASCADE」はノードが削除された場合に、関連するエッジも伝播して削除することを示します。なお、「CASCADE」の代わりに「NO ACTION」を指定することで、エッジと関連付けられたノードを削除しようとするとエラーにすることもできます。

定義したノードとエッジの意味は次の通りです。

データを登録する

ノードとエッジにデータを登録します。

Edge_constraint_in_Graph_Database.sql

```
USE WideWorldImporters;
GO
-- ノードテーブルへのデータ挿入
INSERT INTO Person VALUES (1, 'Taro');
INSERT INTO Person VALUES (2, 'Jiro');
INSERT INTO Person VALUES (3, 'Hanako');
INSERT INTO Location VALUES (1, 'Tokyo');
INSERT INTO Location VALUES (2, 'Osaka');
INSERT INTO Location VALUES (3, 'Nagoya');
INSERT INTO Product VALUES (1, 'Item1');
-- エッジテーブルへのデータ挿入
INSERT INTO LiveIn VALUES ((SELECT $node_id FROM Person WHERE ID = 1),
  (SELECT $node_id FROM Location WHERE ID = 1));
INSERT INTO LiveIn VALUES ((SELECT $node_id FROM Person WHERE ID = 2),
  (SELECT $node_id FROM Location WHERE ID = 2));
INSERT INTO LiveIn VALUES ((SELECT $node_id FROM Person WHERE ID = 3),
  (SELECT $node_id FROM Location WHERE ID = 3));
GO
```

```
-- データ挿入結果を確認
SELECT Person.name, Location.name FROM Person, LiveIn, Location
WHERE MATCH (Person-(LiveIn)->Location)
GO
```

Results　Messages

	name	name
1	Taro	Tokyo
2	Jiro	Osaka
3	Hanako	Nagoya

エッジ制約の動作を確認する

1. 「Person」-「Product」間に「LiveIn」エッジを定義します。

Edge_constraint_in_Graph_Database.sql

```
USE WideWorldImporters;
GO
-- 「Person」-「Product」間に「LiveIn」エッジを作成
INSERT INTO LiveIn VALUES ((SELECT $node_id FROM Person WHERE ID = 1),
  (SELECT $node_id FROM Product WHERE ID = 1));
GO
```

Messages

```
18:25:01    Started executing query at Line 1
            Commands completed successfully.
18:25:01    Started executing query at Line 3
            Msg 547, Level 16, State 0, Line 3
            INSERT ステートメントは EDGE 制約 "EC_LIVEIN1" と競合しています。競合が発生した
            ステートメントは終了されました。
            Total execution time: 00:00:00.028
```

クエリの実行がエラーとなったことがわかります。エラーメッセージの全文は以下の通りです。

```
Msg 547, Level 16, State 0, Line 3
INSERT ステートメントは EDGE 制約 "EC_LIVEIN1" と競合しています。競合が発生したの
は、データベース "WideWorldImporters"、テーブル "dbo.LiveIn" です。
```

制約が機能し意味としてふさわしくないエッジ定義を制御することができています。

2. 「Person」ノードテーブルからデータを削除します。

Edge_constraint_in_Graph_Database.sql

```
USE WideWorldImporters;
GO
-- 「Person」ノードテーブルの「Taro」のデータを削除
DELETE Person where name = 'Taro';
GO
-- データ削除後のデータの状態を確認
SELECT Person.name, Location.name FROM Person, LiveIn, Location
WHERE MATCH (Person-(LiveIn)->Location)
GO
```

Results　Messages

	name	name
1	Jiro	Osaka
2	Hanako	Nagoya

制約が機能し Taro ノードに関連する LiveIn エッジが削除されていることが確認できます。

➢ New graph function - SHORTEST_PATH

1. 概要

SHORTES_PATH 関数によってノード間の最短パスを照会することができるようになります。

2. 利用シーン

ノード間の最短パスを照会することはグラフデータベースへの典型的なクエリですが、SQL Server 2017 では開発者が手動で求める必要がありました。SQL Server 2019 で実装された「SHORTEST_PATH」関数は該当の照会機能を提供します。

3. やってみよう

複数の駅間の最短経路を求めるケースを例に「SHORTEST_PATH」関数の使用方法を確認してみましょう。

ノードとエッジを定義する

ノード「Station」とエッジ「NextTo」を定義します。

New_graph_function-SHORTEST_PATH.sql

```
USE WideWorldImporters;
GO
-- ノードテーブルの作成
CREATE TABLE Station (ID int PRIMARY KEY, name varchar(100)) AS NODE;
-- エッジテーブルの作成
CREATE TABLE NextTo AS EDGE;
GO
```

データを登録する

ノードとエッジにデータを登録します。

New_graph_function-SHORTEST_PATH.sql

```
USE WideWorldImporters;
GO
-- ノードテーブルへのデータ挿入
INSERT INTO Station VALUES (1, 'Station1');
INSERT INTO Station VALUES (2, 'Station2');
INSERT INTO Station VALUES (3, 'Station3');
INSERT INTO Station VALUES (4, 'Station4');
INSERT INTO Station VALUES (5, 'Station5');
INSERT INTO Station VALUES (6, 'Station6');
GO
-- エッジテーブルへのデータ挿入
INSERT INTO NextTo VALUES ((SELECT $node_id FROM Station WHERE ID = 1),
  (SELECT $node_id FROM Station WHERE ID = 2));
INSERT INTO NextTo VALUES ((SELECT $node_id FROM Station WHERE ID = 2),
  (SELECT $node_id FROM Station WHERE ID = 3));
```

```
INSERT INTO NextTo VALUES ((SELECT $node_id FROM Station WHERE ID = 3),
  (SELECT $node_id FROM Station WHERE ID = 4));
INSERT INTO NextTo VALUES ((SELECT $node_id FROM Station WHERE ID = 4),
  (SELECT $node_id FROM Station WHERE ID = 5));
INSERT INTO NextTo VALUES ((SELECT $node_id FROM Station WHERE ID = 5),
  (SELECT $node_id FROM Station WHERE ID = 6));
INSERT INTO NextTo VALUES ((SELECT $node_id FROM Station WHERE ID = 2),
  (SELECT $node_id FROM Station WHERE ID = 5));
INSERT INTO NextTo VALUES ((SELECT $node_id FROM Station WHERE ID = 6),
  (SELECT $node_id FROM Station WHERE ID = 1));
GO
-- データ挿入結果を確認
SELECT st1.name AS st1_name, st2.name AS st2_name
FROM Station AS st1, NextTo, Station AS st2
WHERE MATCH (st1-(NextTo)->st2);
GO
```

Results　Messages

	st1_name	st2_name
1	Station1	Station2
2	Station2	Station3
3	Station3	Station4
4	Station4	Station5
5	Station5	Station6
6	Station2	Station5
7	Station6	Station1

登録したデータは次の通りです。

最短経路を照会する

準備したデータを使用して「Station 1」から各駅への最短経路を検索します。

New_graph_function-SHORTEST_PATH.sql

```
USE WideWorldImporters;
GO
SELECT
  st1.name AS FromST,
  LAST_VALUE(st2.name) WITHIN GROUP (GRAPH PATH) AS ToST,
  STRING_AGG(st2.name, '->') WITHIN GROUP (GRAPH PATH) AS ShortestRoot
FROM
  Station AS st1,
  NextTo FOR PATH AS nt,
  Station FOR PATH AS st2
WHERE MATCH(SHORTEST_PATH(st1(-(nt)->st2)+)) AND st1.name = 'Station1'
GO
```

Results　Messages

	FromST	ToST	ShortestRoot
1	Station1	Station2	Station2
2	Station1	Station3	Station2->Station3
3	Station1	Station5	Station2->Station5
4	Station1	Station4	Station2->Station3->Station4
5	Station1	Station6	Station2->Station5->Station6
6	Station1	Station1	Station2->Station5->Station6->Station1

例えば「Station1」から「Station5」への経路は順に辿るのではなく、「Station2」から「Station5」への路線を利用することが最短であることが示されています。

SHORTEST_PATH 関数の詳細については、以下の Docs を確認ください。

◇ SHORTEST_PATH (Transact-sql)
https://docs.microsoft.com/ja-jp/sql/relational-databases/graphs/sql-graph-shortest-path?view=sql-server-ver15

➢ Use derived table or view aliases in graph match query

1. 概要

ノードやエッジを結合するビューや派生テーブルを定義できるようになり、グラフデータベースへのクエリを直感的シンプルに記述することができるようになります。

2. 利用シーン

例えば、次のような人物を表す「人物」ノードが「友達」エッジと「親戚」エッジの異なる二つのエッジで接続されるケースを考えます。

このとき、「A さんの先輩もしくは後輩である人を検索したい」という要求があった場合、「先輩」と「後輩」は別のエッジとなるため、SQL Server 2017 ではそれぞれのエッジに対してクエリを発行する必要がありました。

SQL Server 2019 では、「先輩」と「後輩」のエッジを結合するビューの作成、またはクエリ中での派生テーブルの作成を行うことにより、シンプルなクエリで実現することができるようになります。

同様に次のようなケースでノードを結合するビューや派生テーブルを作成することもできます。

3. やってみよう

それではグラフデータベースでビューや派生テーブルを使用してみましょう。

ノードとエッジを定義する

ノード「Person」「Company」「City」とエッジ「LacateAt」「BeFrom」を定義します。

Use_derived_table_or_view_aliases_in_graph_match_query.sql

```
USE WideWorldImporters;
GO
-- ノードテーブルの作成
CREATE TABLE Person  (ID int PRIMARY KEY, name varchar(100)) AS NODE;
CREATE TABLE Company (ID int PRIMARY KEY, name varchar(100)) AS NODE;
CREATE TABLE City    (ID int PRIMARY KEY, name varchar(100)) AS NODE;
GO
-- エッジテーブルの作成
CREATE TABLE LocateAt AS EDGE;
CREATE TABLE BeFrom AS EDGE;
GO
```

定義したノードとエッジの意味は次の通りです。

データを登録する

ノードとエッジにデータを登録します。

Use_derived_table_or_view_aliases_in_graph_match_query.sql

```
USE WideWorldImporters;
GO
-- ノードテーブルへデータを挿入
INSERT INTO Person VALUES (1,'Taro');
INSERT INTO Person VALUES (2,'Jiro');
INSERT INTO Person VALUES (3,'Hanako');
INSERT INTO City VALUES (1,'Tokyo');
INSERT INTO City VALUES (2,'Kanagawa');
```

```
INSERT INTO City VALUES (3,'Chiba');
INSERT INTO Company VALUES (1,'Company1');
GO
-- エッジテーブルへデータを挿入
INSERT INTO LocateAt VALUES ((SELECT $node_id FROM Person WHERE ID = 3),
  (SELECT $node_id FROM City WHERE ID = 1));
INSERT INTO LocateAt VALUES ((SELECT $node_id FROM Person WHERE ID = 1),
  (SELECT $node_id FROM City WHERE ID = 2));
INSERT INTO LocateAt VALUES ((SELECT $node_id FROM Person WHERE ID = 2),
  (SELECT $node_id FROM City WHERE ID = 3));
INSERT INTO BeFrom VALUES ((SELECT $node_id FROM Person WHERE ID = 1),
  (SELECT $node_id FROM City WHERE ID = 1));
INSERT INTO BeFrom VALUES ((SELECT $node_id FROM Person WHERE ID = 2),
  (SELECT $node_id FROM City WHERE ID = 2));
INSERT INTO BeFrom VALUES ((SELECT $node_id FROM Person WHERE ID = 3),
  (SELECT $node_id FROM City WHERE ID = 3));
INSERT INTO LocateAt VALUES ((SELECT $node_id FROM Company WHERE ID = 1),
  (SELECT $node_id FROM City WHERE ID = 1));
GO
-- データ挿入結果を確認
SELECT 'LocateAt', Person.name AS Person_name, City.name AS City_name
FROM Person, LocateAt, City
WHERE MATCH (Person-(LocateAt)->City)
GO
SELECT 'BeFrom', Person.name AS Person_name, City.name AS City_name
FROM Person, BeFrom, City
WHERE MATCH (Person-(BeFrom)->City)
GO
SELECT 'LocateAt', Company.name AS Company_name, City.name AS City_name
FROM Company, LocateAt, City
WHERE MATCH (Company-(LocateAt)->City)
GO
```

Results　Messages

	(No column name)	Person_name	City_name
1	LocateAt	Hanako	Tokyo
2	LocateAt	Taro	Kanagawa
3	LocateAt	Jiro	Chiba

	(No column name)	Person_name	City_name
1	BeFrom	Taro	Tokyo
2	BeFrom	Jiro	Kanagawa
3	BeFrom	Hanako	Chiba

	(No column name)	Company_name	City_name
1	LocateAt	Company1	Tokyo

エッジを結合するビューを定義し検索する

ここでは「LocateAt」（所在地）または「BeFrom」（出身地）が「Tokyo」である「Person」を検索するケースをビューによってシンプルなクエリにします。

1. ビューを定義します。

```
Use_derived_table_or_view_aliases_in_graph_match_query.sql
USE WideWorldImporters;
GO
CREATE VIEW Live_AND_BeFrom AS
  SELECT *, 'LocateAt' AS Category FROM LocateAt
  UNION ALL
  SELECT *, 'BeFrom' FROM BeFrom;
GO
```

2. ビューを介した検索を行います。

```
Use_derived_table_or_view_aliases_in_graph_match_query.sql
USE WideWorldImporters;
GO
SELECT Person.name AS PersonName, City.name AS CityName,Category
FROM Person, City, Live_AND_BeFrom
WHERE MATCH(Person-(Live_AND_BeFrom)->City) AND City.name = 'Tokyo';
GO
```

Results　Messages

	PersonName	CityName	Category
1	Hanako	Tokyo	LocateAt
2	Taro	Tokyo	BeFrom

エッジを結合する派生テーブルを定義し検索する

同じく「LocateAt」（所在地）または「BeFrom」（出身地）が「Tokyo」である「Person」を検索するケースを派生テーブルによってシンプルなクエリにします。

```
Use_derived_table_or_view_aliases_in_graph_match_query.sql
USE WideWorldImporters;
GO
SELECT Person.name AS PersonName, City.name AS CityName,Category
FROM (
  SELECT *, 'LocateAt' AS Category FROM LocateAt
  UNION ALL
  SELECT *, 'BeFrom' FROM Befrom
) AS T,Person, City
WHERE MATCH(Person-(T)->City) AND City.name = 'Tokyo';
GO
```

Results　Messages

	PersonName	CityName	Category
1	Hanako	Tokyo	LocateAt
2	Taro	Tokyo	BeFrom

ノードを結合するビューを定義し検索する

ここでは「LocateAt」（所在地）が「Tokyo」である「Person」または「Company」を検索するケースをビューによってシンプルなクエリにします。

1. ビューを定義します。

Use_derived_table_or_view_aliases_in_graph_match_query.sql

```
USE WideWorldImporters;
GO
CREATE VIEW Person_AND_Company AS
  SELECT name, 'Person' AS Category FROM Person
  UNION ALL
  SELECT name, 'Company' FROM Company;
GO
```

2. ビューを介した検索を行います。

Use_derived_table_or_view_aliases_in_graph_match_query.sql

```
USE WideWorldImporters;
GO
SELECT Person_AND_Company.name, City.name AS CityName, Category
FROM Person_AND_Company, City, LocateAt
WHERE MATCH(Person_AND_Company-(LocateAt)->City) AND City.name = 'Tokyo';
GO
```

Results　Messages

	name	CityName	Category
1	Hanako	Tokyo	Person
2	Company1	Tokyo	Company

ノードを結合する派生テーブルを定義し検索する

同じく「LocateAt」（所在地）が「Tokyo」である「Person」または「Company」を検索するケースを派生テーブルによってシンプルなクエリにします。

Use_derived_table_or_view_aliases_in_graph_match_query.sql

```
USE WideWorldImporters;
GO
SELECT Person_AND_Company.name, City.name AS CityName, Category
FROM (
 SELECT name,'Person' AS Category  FROM Person
 UNION ALL
 SELECT name,'Company' FROM Company
) AS Person_AND_Company, LocateAt, City
WHERE MATCH(Person_AND_Company-(LocateAt)->City) AND City.name = 'Tokyo';
GO
```

Results Messages

	name	CityName	Category
1	Hanako	Tokyo	Person
2	Company1	Tokyo	Company

➢ Using the T-SQL MERGE statement to support scenarios like upsert

1. 概要

ノードテーブルのみ対応していた MERGE 文がエッジテーブルにも対応します。

2. 利用シーン

例えば、「人物」ノードが「商品」ノードと「累計購入数」という属性値を持つ「購入」エッジで接続されているケースを考えます。

このケースにおいて「A さんが商品 C を新たに 5 つ購入した」という事象が発生した場合に行うべき操作は次のようになります。

- 「A さん」がすでに「商品 C」を購入したことがあった場合、「購入」エッジの累計購入数を加算する。
- 「A さん」が「商品 C」を初めて購入する場合、新しく「購入エッジ」を追加する。

こうした操作に MERGE 文が便利です。
SQL Server 2017 はノードテーブルのみの MERGE 文が対応していましたが、SQL Server 2019 でエッジテーブルにも対応します。

3. やってみよう

それではエッジテーブルへ MERGE 文をクエリしてみましょう。

ノードとエッジを定義する

ノード「Person」「Product」とエッジ「Bought」を定義します。

Using_the_T-SQL_MERGE_statement_to_support_scenarios_like_upsert.sql

```
USE WideWorldImporters;
GO
-- ノードテーブルの作成
CREATE TABLE Person  (ID int PRIMARY KEY, name varchar(100)) AS NODE;
CREATE TABLE Product (ID int PRIMARY KEY, name varchar(100)) AS NODE;
```

```
-- エッジテーブルの作成
CREATE TABLE Bought(bought_count int) AS EDGE;
GO
```

定義したノードとエッジの意味は次の通りです。

データを登録する

ノードとエッジにデータを登録します。

Using_the_T-SQL_MERGE_statement_to_support_scenarios_like_upsert.sql

```
USE WideWorldImporters;
GO
-- ノードテーブルへのデータ挿入
INSERT INTO Person VALUES (1,'Taro');
INSERT INTO Person VALUES (2,'Jiro');
INSERT INTO Product VALUES (1,'PC');
INSERT INTO Product VALUES (2,'LAN cable');
INSERT INTO Product VALUES (3,'Keyboard');
GO
-- エッジテーブルへのデータ挿入
INSERT INTO Bought VALUES ((SELECT $node_id FROM Person WHERE ID = 1),
  (SELECT $node_id FROM Product WHERE ID = 3),1);
INSERT INTO Bought VALUES ((SELECT $node_id FROM Person WHERE ID = 2),
  (SELECT $node_id FROM Product WHERE ID = 2),3);
GO
-- データ挿入結果を確認
SELECT Person.name AS PersonName, Product.name AS ProductName, Bought.bought_count
FROM Person, Product, Bought
WHERE MATCH(Person-(Bought)->Product);
GO
```

Results　Messages

	PersonName	ProductName	bought_count
1	Taro	Keyboard	1
2	Jiro	LAN cable	3

初期状態では以下 2 点の情報が存在しています。

- 「Taro」が「Keyboard」を累計 1 つ購入した
- 「Jiro」が「LAN cable」を累計 3 つ購入した

MERGE を使用して「Bought」エッジテーブルへ UPSERT を行う

例として次のシナリオを想定した MERGE を実行します。

- 「Taro」が「LAN cable」を新たに 1 つ購入した
- 「Taro」が「Keyboard」を新たに 3 つ購入した

Using_the_T-SQL_MERGE_statement_to_support_scenarios_like_upsert.sql

```
USE WideWorldImporters;
GO
-- UPSERT の実行（「Taro」が「LAN cable」を新たに 1 つ購入した）
DECLARE @PersonID varchar(100) = 1  -- 'Taro'
DECLARE @ProductID varchar(100) = 2 -- 'LAN cable'
DECLARE @add_count int = 1
MERGE Bought
  USING (
   (SELECT @PersonID, @ProductID, @add_count) AS T (PersonID, ProductID, add_count)
      JOIN Person ON T.PersonID = Person.ID
      JOIN Product ON T.ProductID = Product.ID)
   ON MATCH (Person-(Bought)->Product)
WHEN MATCHED THEN
  UPDATE SET bought_count = bought_count + add_count
WHEN NOT MATCHED THEN
  INSERT ($from_id, $to_id, bought_count)
    VALUES (Person.$node_id, Product.$node_id, add_count);
GO

-- UPSERT の実行（「Taro」が「Keyboard」を新たに 3 つ購入した）
DECLARE @PersonID varchar(100) = 1  -- 'Taro'
DECLARE @ProductID varchar(100) = 3 -- 'Keyboard'
DECLARE @add_count int = 3
MERGE Bought
  USING (
   (SELECT @PersonID, @ProductID, @add_count) AS T (PersonID, ProductID, add_count)
      JOIN Person ON T.PersonID = Person.ID
      JOIN Product ON T.ProductID = Product.ID)
   ON MATCH (Person-(Bought)->Product)
WHEN MATCHED THEN
  UPDATE SET bought_count = bought_count + add_count
WHEN NOT MATCHED THEN
  INSERT ($from_id, $to_id, bought_count)
    VALUES (Person.$node_id, Product.$node_id, add_count);
GO

-- UPSERT の実行結果を確認
SELECT Person.name AS PersonName, Product.name AS ProductName, Bought.bought_count
FROM Person, Product, Bought
WHERE MATCH(Person-(Bought)->Product);
GO
```

1 つ目の MERGE は「Taro」にとって「LAN cable」は初めての購入であるため INSERT とし、続いて 2 つ目の MERGE は「Taro」にとって「Keyboard」はリピート購入であるため UPDATE としています。

Results　Messages

	PersonName	ProductName	bought_count
1	Taro	Keyboard	4
2	Jiro	LAN cable	3
3	Taro	LAN cable	1

5.2. Unicode support

➢ Support for UTF-8 character encoding

1. 概要

Unicode 符号化文字集合に対する文字符号化方式として従来の UTF-16 に加えて、新たに UTF-8 を選択することができるようになります。

2. 利用シーン

SQL Server はこれまで長らく Unicode データ型（nchar, nvarchar）によって Unicode 符号化文字集合によるデータベースの多言語対応をサポートしてきました。この Unicode データ型の文字符号化方式には UTF-16 が利用されています。そして SQL Server 2019 では文字符号化方式として新たに UTF-8 を選択することができるようになります。

これら 2 つの文字符号化方式の違いにおいて重要なポイントは文字が対応するコードポイントによって必要なストレージサイズが異なるということです。そしてストレージサイズが異なるということは性能にも直結する話題であるということです。

次の表はコードポイントの範囲と必要なストレージサイズの対応表です。

コード範囲（16 進数）	コード範囲（10 進数）	UTF-8 の ストレージバイト	UTF-16 の ストレージバイト
000000 – 00007F	0～127	1	2
000080 – 00009F 0000A0 – 0003FF 000400 – 0007FF	128～159 160～1,023 1,024～2,047	2	2
000800 – 003FFF 004000 – 00FFFF	2,048～16,383 16,384～65,535	3	2
010000 – 03FFFF 040000 – 10FFFF	65,536～262,143 262,144～1,114,111	4	4

例えばコード範囲 000000-00007F に位置づく ASCII 文字であれば、新しく対応した UTF-8 を選択することでストレージサイズが半分に圧縮されるため、性能向上が期待できます。一方で 000800-00FFFF に位置づく日本語、中国語、韓国語などは UTF-8 を選択することでストレージサイズが 1.5 倍必要になります。

このように、扱う文字に応じた相違を踏まえて最適な文字符号化方式を選択する必要があります。

また、既存データベースの UTF-16 データを新しく対応した UTF-8 に以降する際は、UTF-8 で必要となるストレージサイズを踏まえてデータ型のサイズを調整する必要があることにも留意が必要です。

3. やってみよう

ここでは、Unicode 未対応のサンプルデータベース「DB01」を文字符号化方式 UTF-16 で対応する場合と、文字符号化方式 UTF-8 で対応する場合のそれぞれでデータ移行を実施して結果を確認してみます。

サンプルデータベースを作成する

動作確認用に以下 3 つのサンプルデータベースを作成します。

- DB01：移行元となる Unicode 未対応のデータベース
- DB02：文字符号化方式 UTF-16 により Unicode 対応したデータベース
- DB03：文字符号化方式 UTF-8 により Unicode 対応したデータベース

Support_for_UTF-8_character_encoding.sql

```
USE master;
GO
-- テスト用の 3 つのサンプルデータベースを作成
CREATE DATABASE DB01 COLLATE Japanese_XJIS_140_CS_AS_KS_WS;
GO
CREATE DATABASE DB02 COLLATE Japanese_XJIS_140_CS_AS_KS_WS;
GO
CREATE DATABASE DB03 COLLATE Japanese_XJIS_140_CS_AS_KS_WS_UTF8;
GO
```

UTF-16 と異なり UTF-8 に対応させる場合には UTF-8 に対応した照合順序を選択する必要があります。UTF-8 に対応した照合順序のリストは以下クエリで確認できます。

Support_for_UTF-8_character_encoding.sql

```
USE master;
GO
-- UTF-8 に対応する照合順序のリスト
SELECT name, description FROM fn_helpcollations() WHERE name LIKE '%UTF8';
GO
-- 日本語で UTF-8 に対応する照合順序のリスト
SELECT name, description FROM fn_helpcollations() WHERE name LIKE 'Japanese%UTF8';
GO
```

非 Unicode データベースにデータを登録する

「DB01」データベースにテーブルを作成しデータを登録します。

Support_for_UTF-8_character_encoding.sql

```
USE DB01;
GO
-- ループ処理のパフォーマンス向上のためカウント処理を OFF に設定
SET NOCOUNT ON;
-- Unicode 未対応のサンプルテーブルを作成
CREATE TABLE dbo.non_unicode_table (
  id int IDENTITY(1,1) NOT NULL PRIMARY KEY,
  col1 varchar(50) NOT NULL
);
GO

-- ランダムなアルファベット文字列を 100 万件登録
```

```
BEGIN TRANSACTION;
DECLARE @i int = 1;
WHILE @i <= 10000
BEGIN
  INSERT INTO dbo.non_unicode_table (col1)
  SELECT REPLICATE(CONCAT(
    CHAR(FLOOR(65 + (RAND() * 25))),
    CHAR(FLOOR(65 + (RAND() * 25))),
    CHAR(FLOOR(65 + (RAND() * 25))),
    CHAR(FLOOR(65 + (RAND() * 25))),
    CHAR(FLOOR(65 + (RAND() * 25))),
    CHAR(FLOOR(65 + (RAND() * 25))),
    CHAR(FLOOR(65 + (RAND() * 25))),
    CHAR(FLOOR(65 + (RAND() * 25))),
    CHAR(FLOOR(65 + (RAND() * 25))),
    CHAR(FLOOR(65 + (RAND() * 25)))
    ), 5);
  SET @i += 1;
END;
COMMIT TRANSACTION;
GO
-- 登録されたデータ件数を確認
SELECT COUNT(*) FROM dbo.non_unicode_table;
GO
```

Results　Messages

	(No column name)
1	10000

非 Unicode データベースを Unicode 対応データベースに移行する

1. UTF-16 の「DB02」データベースに移行します。

Support_for_UTF-8_character_encoding.sql

```
USE DB02;
GO
-- varchar から nvarchar に列定義を変換したテーブルを作成
CREATE TABLE dbo.utf16_table(
  id int IDENTITY(1,1) NOT NULL PRIMARY KEY,
  col1 nvarchar(50) NOT NULL
);
GO
-- 移行元の「non_unicode_table」テーブルからデータを移行
BEGIN TRANSACTION;
  INSERT INTO dbo.utf16_table(col1) SELECT col1 FROM DB01.dbo.non_unicode_table;
COMMIT TRANSACTION;
```

UTF-16 のデータを格納するために Unicode データ型（nvarchar）を指定しています。

2. UTF-8 の「DB03」データベースに移行します。

```
Support_for_UTF-8_character_encoding.sql
USE DB03;
GO
-- 移行元と同一定義でテーブルを作成
CREATE TABLE dbo.utf8_table(
  id int IDENTITY(1,1) NOT NULL PRIMARY KEY,
  col1 varchar(50) NOT NULL
);
GO
-- 移行元の「non_unicode_table」からデータを登録
BEGIN TRANSACTION;
  INSERT INTO dbo.utf8_table(col1) SELECT col1 FROM db01.dbo.non_unicode_table;
COMMIT TRANSACTION;
```

UTF-16 とは異なり UTF-8 に対応した照合順序を設定することで UTF-8 への対応になるため、データ型には char か varchar を指定します。

3. ストレージサイズを確認します。

```
Support_for_UTF-8_character_encoding.sql
USE DB01;
GO
-- 移行元の「non_unicode_table」テーブルの容量を確認
EXEC sp_spaceused N'dbo.non_unicode_table';
GO

USE DB02;
GO
-- UTF-16 エンコードの「utf16_table」テーブルの容量を確認
EXEC sp_spaceused N'dbo.utf16_table';
GO

USE DB03;
GO
-- UTF-8 エンコードの「utf8_table」テーブルの容量を確認
EXEC sp_spaceused N'dbo.utf8_table';
GO
```

Results　Messages

	name	rows	reserved	data	index_size	unused
1	non_unicode_table	10000	712 KB	672 KB	16 KB	24 KB

	name	rows	reserved	data	index_size	unused
1	utf16_table	10000	1288 KB	1160 KB	16 KB	112 KB

	name	rows	reserved	data	index_size	unused
1	utf8_table	10000	840 KB	672 KB	16 KB	152 KB

今回のデータはコード範囲 000000-00007F に位置づく ASCII 文字としたため、UTF-16 に比較し UTF-8 はおよそ半分のサイズに収まっていることが確認できます。
先に解説した通り、例えば 000800-00FFFF に位置づく日本語、中国語、韓国語を格納する場合には UTF-16 に比較し UTF-8 はストレージサイズが 1.5 倍必要とするため、こうした扱う文字に応じた相違を踏まえて最適な文字符号化方式を選択する必要があります。

5.3. Language extensions

➢ Extensibility Framework SQL Server Language Extensions

1. 概要

言語拡張機能が新たに追加され、Java コード実行と Transact-SQL クエリ実行が統合されます。

2. 利用シーン

言語拡張機能は SQL Server 2019 で実装された機能で、Java コード実行と Transact-SQL クエリ実行を統合します。これにより SQL Server が既に有しているパフォーマンス、セキュリティ、ハイアベイラビリティなどのリッチな機能の上で Java コードを実行できるようになります。

3. やってみよう

それでは Transact-SQL からの Java コードの呼び出しを実践してみましょう。

✱ 環境設定

言語拡張機能を利用するための環境設定を行います。

言語拡張機能をインストールする

以下のマニュアルを参照して、言語拡張機能をインストールします。

◇ SQL Server の言語拡張を Windows にインストールする
https://docs.microsoft.com/ja-jp/sql/language-extensions/install/install-sql-server-language-extensions-on-windows?view=sql-server-ver15

◇ SQL Server の言語拡張を Linux にインストールする
https://docs.microsoft.com/ja-jp/sql/linux/sql-server-linux-setup-language-extensions?view=sql-server-ver15

以降の手順は Windows 版を前提に解説します。また、Java ランタイムは SQL Server のインストーラーに含まれている既定の JRE である Zulu Open JRE 11.0.3 を使用する、ものとします。

システム環境変数を設定する

インストールした Java ランタイムのためのシステム環境変数「JRE_HOME」を設定します。

変数名	JRE_HOME

変数値	C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLSERVER¥AZUL-OpenJDK-JRE¥

環境変数の設定を反映するために SQL Server Launchpad サービスを再起動します。

Extensibility_Framework_SQL_Server_Language_Extensions.bat.txt

```
net stop "SQL Server Launchpad (MSSQLSERVER)"
net start "SQL Server Launchpad (MSSQLSERVER)"
```

SQL Server で外部スクリプト機能を有効化する

インスタンスで外部スクリプト機能を有効化します。

Extensibility_Framework_SQL_Server_Language_Extensions.sql

```
EXEC sp_configure 'external scripts enabled', 1
RECONFIGURE WITH OVERRIDE
GO
```

設定を反映するために SQL Server サービスを再起動します。

Extensibility_Framework_SQL_Server_Language_Extensions.bat.txt

```
net stop "SQL Server (MSSQLSERVER)" /y
net start "SQL Server (MSSQLSERVER)"
net start "SQL Server Launchpad (MSSQLSERVER)"
```

ハンズオン用データベースを作成する

ハンズオン用のデータベースとテーブルを作成し、データを登録します。

Extensibility_Framework_SQL_Server_Language_Extensions.sql

```
-- サンプルデータベース DB01 を作成
CREATE DATABASE DB01;
GO

USE DB01;
GO
-- DB01 にテーブル Table1 を作成
CREATE TABLE Table1 (id int NOT NULL, text nvarchar(100) NOT NULL);
```

```
GO
-- Table1 へパターンマッチングに使用するテストデータを挿入
INSERT INTO Table1 VALUES (1, N'This sentence contains java');
INSERT INTO Table1 VALUES (2, N'This sentence does not');
INSERT INTO Table1 VALUES (3, N'I love Java!');
GO
```

データベースに Java ランタイムと SDK 用ライブラリを登録する

Java ランタイムを外部言語オブジェクトとして、SDK 用ライブラリを外部ライブラリとして、それぞれデータベースに登録します。

Extensibility_Framework_SQL_Server_Language_Extensions.sql

```
USE DB01;
GO
-- 外部言語 Java を作成
CREATE EXTERNAL LANGUAGE Java FROM (
  CONTENT = N'C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLSERVER¥MSSQL¥Binn¥jav
a-lang-extension.zip',
  FILE_NAME = 'javaextension.dll'
);
GO

-- 外部言語 Java が作成されたことを確認
SELECT * FROM sys.external_language_files WHERE file_name = 'javaextension.dll';
GO

-- SDK 用の外部ライブラリの作成
CREATE EXTERNAL LIBRARY sdk
FROM (CONTENT = 'C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLSERVER¥MSSQL¥Binn
¥mssql-java-lang-extension.jar')
WITH (LANGUAGE = 'Java');
GO

-- 外部ライブラリが作成されたことを確認
SELECT * FROM sys.external_libraries;
GO
```

Results　Messages

	external_language_id	content	file_name	platform	platform_desc	parameters	environment_variables
1	65536	0x504B03041400000080002ACF8…	javaextension.dll	1	WINDOWS	NULL	NULL

	external_library_id	name	principal_id	language	scope	scope_desc
1	65536	sdk	1	Java	0	PUBLIC

✱ Java アプリケーションのビルド

Java アプリケーションをビルドします。

Zulu OpenJDK を準備する

1. Zulu OpenJDK を以下から入手してインストールしてください。

◇ Download Zulu Community
https://jp.azul.com/downloads/zulu-community/?package=jdk

なお、本検証ではバージョン 11 を利用するものとします。

2. インストール完了後、システム環境変数「JAVA_HOME」を設定します。

変数名	JAVA_HOME
変数値	C:¥Program Files¥Zulu¥zulu-11

「zulu-11」の部分はインストールする Zulu OpenJDK のバージョンによって異なるためインストールしたバージョンに合わせて適宜読み替えます。

3. 続けて、システム環境変数「Path」に次のパスが登録されていることを確認してください。登録されていない場合は新規登録を行います。先ほどと同様に、「zulu-11」の部分は適宜読み替えてください。

設定値	C:¥Program Files¥Zulu¥zulu-11¥bin

Java ソースコードを準備する

テキストエディタを開き、次のコードをペースト後、「RegexSample.java」という名前で、「C:¥work¥pkg」フォルダに保存します。

RegexSample.java

```
package pkg;

import com.microsoft.sqlserver.javalangextension.PrimitiveDataset;
import com.microsoft.sqlserver.javalangextension.AbstractSqlServerExtensionExecutor;
import java.util.LinkedHashMap;
import java.util.LinkedList;
import java.util.ListIterator;
import java.util.regex.*;

public class RegexSample extends AbstractSqlServerExtensionExecutor {
    private Pattern expr;

    public RegexSample() {
        // Setup the expected extension version, and class to use for input and output
 dataset
        executorExtensionVersion = SQLSERVER_JAVA_LANG_EXTENSION_V1;
        executorInputDatasetClassName = PrimitiveDataset.class.getName();
        executorOutputDatasetClassName = PrimitiveDataset.class.getName();
    }

    public PrimitiveDataset execute(PrimitiveDataset input, LinkedHashMap<String, Obje
ct> params) {
        // Validate the input parameters and input column schema
        validateInput(input, params);

        int[] inIds = input.getIntColumn(0);
        String[] inValues = input.getStringColumn(1);
        int rowCount = inValues.length;

        String regexExpr = (String)params.get("regexExpr");
        expr = Pattern.compile(regexExpr);

        System.out.println("regex expression: " + regexExpr);

        // Lists to store the output data
        LinkedList<Integer> outIds = new LinkedList<Integer>();
        LinkedList<String> outValues = new LinkedList<String>();

        // Evaluate each row
        for(int i = 0; i < rowCount; i++) {
            if (check(inValues[i])) {
                outIds.add(inIds[i]);
                outValues.add(inValues[i]);
            }
        }

        int outputRowCount = outValues.size();

        int[] idOutputCol = new int[outputRowCount];
```

```
        String[] valueOutputCol = new String[outputRowCount];

        // Convert the list of output columns to arrays
        outValues.toArray(valueOutputCol);

        ListIterator<Integer> it = outIds.listIterator(0);
        int rowId = 0;

        System.out.println("Output data:");
        while (it.hasNext()) {
            idOutputCol[rowId] = it.next().intValue();

            System.out.println("ID: " + idOutputCol[rowId] + " Value: " + valueOutputC
ol[rowId]);
            rowId++;
        }

        // Construct the output dataset
        PrimitiveDataset output = new PrimitiveDataset();

        output.addColumnMetadata(0, "ID", java.sql.Types.INTEGER, 0, 0);
        output.addColumnMetadata(1, "Text", java.sql.Types.NVARCHAR, 0, 0);

        output.addIntColumn(0, idOutputCol, null);
        output.addStringColumn(1, valueOutputCol);

        return output;
    }

    private void validateInput(PrimitiveDataset input, LinkedHashMap<String, Object> p
arams) {
        // Check for the regex expression input parameter
        if (params.get("regexExpr") == null) {
            throw new IllegalArgumentException("Input parameter 'regexExpr' is not fou
nd");
        }

        // The expected input schema should be at least 2 columns, (INTEGER, STRING)
        if (input.getColumnCount() < 2) {
            throw new IllegalArgumentException("Unexpected input schema, schema should
 be an (INTEGER, NVARCHAR or VARCHAR)");
        }

        // Check that the input column types are expected
        if (input.getColumnType(0) != java.sql.Types.INTEGER &&
                (input.getColumnType(1) != java.sql.Types.VARCHAR && input.getColumnTy
pe(1) == java.sql.Types.NVARCHAR )) {
            throw new IllegalArgumentException("Unexpected input schema, schema should
 be an (INTEGER, NVARCHAR or VARCHAR)");
        }
    }

    private boolean check(String text) {
        Matcher m = expr.matcher(text);

        return m.find();
    }
}
```

jar ファイルを作成する

1. Java コードをコンパイルします。

```
Extensibility_Framework_SQL_Server_Language_Extensions.bat.txt
cd C:¥work
javac -cp "C:¥Program Files¥Microsoft SQL Server¥MSSQL15.MSSQLSERVER¥MSSQL¥Binn¥mssql-java-lang-extension.jar" pkg¥RegexSample.java
```

「C:¥work¥pkg」フォルダ配下に「RegexSample.class」ファイルが生成されていることを確認します。

2. jar ファイルを生成します。

```
Extensibility_Framework_SQL_Server_Language_Extensions.bat.txt
jar cvf regex.jar pkg
```

「C:¥work」フォルダ配下に「regex.jar」ファイルが生成されていることを確認します。

✱ アプリケーションの登録と呼び出し

Java アプリケーションを SQL Server に統合します。

アプリケーションを登録する

作成したアプリケーションを外部ライブラリとして登録します。

Extensibility_Framework_SQL_Server_Language_Extensions.sql

```
USE DB01;
GO
-- RegEx 用の外部ライブラリの作成
CREATE EXTERNAL LIBRARY regex
FROM (CONTENT = 'C:¥work¥regex.jar')
WITH (LANGUAGE = 'Java');
GO
-- 外部ライブラリが作成されたことを確認
SELECT * FROM sys.external_libraries;
GO
```

Results　Messages

	external_library_id	name	principal_id	language	scope	scope_desc
1	65536	sdk	1	Java	0	PUBLIC
2	65537	regex	1	Java	0	PUBLIC

アプリケーションを呼び出す

ここではアプリケーション呼び出しをラップするプロシージャを作成し、プロシージャを介してアプリケーションを利用します。

1. アプリケーション呼び出しをラップするプロシージャを作成します。

Extensibility_Framework_SQL_Server_Language_Extensions.sql

```
USE DB01;
GO
-- Java クラスを呼び出すストアドプロシージャを作成
CREATE PROCEDURE java_regex @expr nvarchar(200), @query nvarchar(400)
AS
BEGIN
  EXEC sp_execute_external_script
    @language     = N'Java',
    @script       = N'pkg.RegexSample',
    @input_data_1 = @query,
    @params       = N'@regexExpr nvarchar(200)',
    @regexExpr    = @expr
  WITH RESULT SETS ((ID int, text nvarchar(100)));
END;
GO
```

このアプリケーションは「@input_data_1」で与えられるデータセットから、変数「@regexExpr」で与えられる正規表現パターンにマッチするレコードを抽出する機能を持ちます。

2. プロシージャを介してアプリケーションを利用します。

Extensibility_Framework_SQL_Server_Language_Extensions.sql

```
USE DB01;
GO

-- パターンマッチングにかけるデータセットを確認
SELECT id, text FROM Table1;
GO

-- 正規表現パターン「[Jj]ava」でデータセットの評価を実行
EXECUTE java_regex N'[Jj]ava', N'SELECT id, text FROM Table1';
GO
```

Results　Messages

	id	text
1	1	This sentence contains java
2	2	This sentence does not
3	3	I love Java!

	ID	text
1	1	This sentence contains java
2	3	I love Java!

正規表現パターン「[Jj]ava」にマッチする２行が抽出されたことが確認できます。

5.4. Spatial

➢ New spatial reference identifiers (SRIDs)

1. 概要

GDA2020 に対応した新しい SRID（Spatial Reference Identifier）が追加されます。

2. 利用シーン

オーストラリアにおける測地基準が GDA94 から GDA2020 へ更新されました。これにより、従来の GDA94 では GPS などの衛星測位システムから返される座標との間で見られた乖離がなくなり、より正確な座標データが提供されるようになりました。これに伴い、SQL Server 2019 にも GDA2020 に対応した新しい SRID（Spatial Reference Identifier）が追加されています。この新しい SRID は SQL Server の空間データ メソッドを使用してデータの操作を実行する際に使用され、例えば Transact-SQL から特定の地点の緯度経度間の距離を計算するといったことに使われます。

空間データや SRID の詳細については、以下の Docs を確認ください。

◇ SRID (Spatial Reference Identifier)
https://docs.microsoft.com/ja-jp/sql/relational-databases/spatial/spatial-reference-identifiers-srids?view=sql-server-ver15

GDA2020 の詳細については、以下を確認ください。

◇ Geocentric Datum of Australia 2020 (GDA2020)
http://www.ga.gov.au/scientific-topics/positioning-navigation/geodesy/datums-projections/gda2020

6. Platform choice

このセクションは、SQL Server が動作するプラットフォームの拡充として、Linux 対応とコンテナー対応の強化を取り上げます。

- **SQL Server 2019 on Linux**
  SQL Server 2019 on Linux では Linux ディストリビューションのより新しいリリースへの対応のほか、SQL Server on Windows との機能差を埋める機能拡充がなされています。

- **SQL Server 2019 on Linux Container with Docker**
  SQL Server 2019 on Linux Container with Docker では、その配布に関して 2 つのトピックがあり、ひとつは RedHat 用の SQL Server コンテナーイメージが新しく公開されるようになること、もうひとつは SQL Server 2019 コンテナーイメージが非ルートコンテナーとして公開されるようになることです。

6.1. SQL Server 2019 on Linux

SQL Server の Linux 版への対応表明は、2016 年 3 月 7 日に公式アナウンスがありました。その後の SQL Server 2017 においてそれが現実のものとしてリリースされ、今現在 SQL Server は Linux でも稼働しています。
ここではその仕組の詳細は割愛しますが、キーとなるコンポーネントが SQL Platform Abstraction Layer（SQLPAL）です。

SQLPAL はプラットフォーム固有のコードを統合して抽象化します。これにより SQL Server コードはプラットフォーム非依存となります。つまり SQLPAL を介すことで、Linux 向けにリライトやポーティングすることなく、長年の実績ある従来の SQL Server コードが実行されていることがポイントです。

次の図は Linux でのプロセスモデルです。

SQLPAL のより詳細については技術記事を確認ください。

◇ SQL Server on Linux: How? Introduction
https://blogs.technet.microsoft.com/dataplatforminsider/2016/12/16/sql-server-on-linux-how-introduction/

➢ SQL Server 2019 on Linux の強化ポイント

- Linux ディストリビューションリリースへの対応

  SQL Server 2019 は Linux ディストリビューションのより新しいリリースに対してサポートが拡充されています。2020 年 3 月時点の SQL Server 2019 がサポートする Linux ディストリビューションリリースは以下の通りです。

Red Hat Enterprise Linux	7.3、7.4、7.5、7.6、8
SUSE Enterprise Linux Server	12 SP2、12 SP3、12 SP4
Ubuntu	16.04LTS

  なお、最新のサポート状況は Docs で確認してください。

  ◇ Linux 上の SQL Server 2019 のリリース ノート
  …サポートされているプラットフォーム
  https://docs.microsoft.com/ja-jp/sql/linux/sql-server-linux-release-notes-2019?view=sql-server-ver15#supported-platforms

- Persistent Memory 対応の強化

  SQL Server 2019 の Linux 版は SQL Server 2017 の Linux 版で未サポートだった Persisted Log Buffer 機能に加え、SQL Server 2019 から導入された Hybrid Buffer Pool 機能、そして Enlightenment 機能がサポートされます。Enlightenment 機能はクリーンページの読み取りに特化した Hybrid Buffer Pool 機能とは異なり、PMEM 上のデータファイルおよびログファイルに対し Linux カーネルの I/O スタックをバイパスした Read、Write のアクセスを提供します。詳細は、SQL Server 2019 自習書「1. Intelligent Performance」-「Enhanced PMEM support（Hybrid Buffer Pool & Enlightenment）」の項目に記載していますので、そちらをご参照ください。

- Windows 版 SQL Server との機能差を埋める機能拡充

  従来から Windows 版でサポートされていた以下の機能が Linux 版にも対応しました。

  ◇ Replication
  https://docs.microsoft.com/ja-jp/sql/linux/sql-server-linux-replication?view=sql-server-ver15

  ◇ Change Data Capture（CDC）
  https://docs.microsoft.com/ja-jp/sql/relational-databases/track-changes/about-change-data-capture-sql-server?view=sql-server-ver15

◇ Microsoft Distributed Transaction Coordinator (MSDTC)
https://docs.microsoft.com/ja-jp/sql/linux/sql-server-linux-configure-msdtc?view=sql-server-ver15

◇ Machine Learning Services
https://docs.microsoft.com/ja-jp/sql/advanced-analytics/?view=sql-server-ver15

◇ PolyBase
https://docs.microsoft.com/ja-jp/sql/relational-databases/polybase/polybase-guide?view=sql-server-ver15

次の表に、SQL Server 2019 における Windows 版と Linux 版の機能対応を整理します。まだ Linux 版に対応していない機能がいくつか存在しますが、かなり機能差が埋まってきています。

Performance

Feature	Windows	Linux
Columnstore	●	●
Large object binaries in clustered columnstore indexes	●	●
Online nonclustered columnstore index rebuild	●	●
In-Memory Database: In-Memory OLTP	●	●
In-Memory Database: hybrid buffer pool	●	●
In-Memory Database: memory-optimized tempdb metadata	●	●
Enlightenment	–	●
Multi-instance support	●	–
Table and index partitioning	●	●
Data compression	●	●
Resource Governor	●	●
Partitioned Table Parallelism	●	●
Buffer pool extension	●	–
IO Resource Governance	●	●
Read-ahead	●	●
Advanced scanning(aka merrygoround scan)	●	●
Delayed Durability	●	●
Intelligent Database: automatic tuning	●	●
Intelligent Database: batch mode for row store	●	●
Intelligent Database: row mode memory grant feedback	●	●
Intelligent Database: approximate count distinct	●	●

Intelligent Database: table variable deferred compilation	●	●
Intelligent Database: scalar UDF inlining	●	●
Intelligent Database: Batch Mode Adaptive Joins	●	●
Intelligent Database: Batch Mode Memory Grant Feedback	●	●
Intelligent Database: Interleaved Execution for Multi-Statement Table Valued Functions	●	●
Bulk insert improvements	●	●

Scalability

Feature	Windows	Linux
NUMA Aware and Large Page Memory and Buffer Array Allocation	●	●
Change data capture	●	●
Snapshot replication	●	●
Transactional replication	●	●
Stretch database	●	–
Heterogeneous subscribers	●	–
Merge replication	●	–
Oracle publishing	●	–
Peer to peer transactional replication	●	–

Scalability

Feature	Windows	Linux
Log shipping	●	●
Database mirroring	●	–
Backup compression	●	●
Database snapshot	●	●
Always On failover cluster instance	●	●
Always On availability groups	●	●
Basic availability groups	●	–
Automatic read write connection re-routing	●	●
Minimum replica commit availability group	●	●
Clusterless availability group	●	●
Online page and file restore	●	●
Online index create and rebuild	●	●
Resumable online index rebuilds	●	●
Online schema change	●	●

Fast recovery	●	●
Accelerated database recovery	●	●
Mirrored backups	●	●
Hot add memory and CPU	●	●
Hybrid backup to Azure (backup to URL)	●	●

Scalability

Feature	Windows	Linux
Row-level security	●	●
Always Encrypted	●	●
Always Encrypted with Secure Enclaves	●	–
Dynamic data masking	●	●
Basic auditing	●	●
Fine grained auditing	●	●
Transparent database encryption	●	●
Extensible key management	●	–
User-defined roles	●	●
Contained databases	●	●
Encryption for backups	●	●

Manageability

Feature	Windows	Linux
Dedicated admin connection	●	●
PowerShell scripting support	●	●
Support for data-tier application component operations - extract, deploy, upgrade, delete	●	●
Policy automation (check on schedule and change)	●	●
Performance data collector	●	●
Standard performance reports	●	●
Plan guides and plan freezing for plan guides	●	●
Direct query of indexed views (using NOEXPAND hint)	●	●
Automatic indexed views maintenance	●	●
Distributed partitioned views	●	●
Parallel indexed operations	●	●
Automatic use of indexed view by query optimizer	●	●
Parallel consistency check	●	●

SQL Server Utility Control Point	●	●

Programmability

Feature	Windows	Linux
JSON	●	●
Query Store	●	●
Temporal	●	●
Common Language Runtime (CLR) Integration	●	●
Java Language Runtime Integration	●	●
Native XML support	●	●
XML indexing	●	●
MERGE & UPSERT capabilities	●	●
FILESTREAM support	●	–
FileTable	●	–
Date and Time datatypes	●	●
Internationalization support	●	●
Full-text and semantic search	●	●
Specification of language in query	●	●
Service Broker (messaging)	●	●
Transact-SQL endpoints	●	●
Graph	●	●
UTF-8 support	●	●
Spatial indexes	●	●
Planar and geodetic datatypes	●	●
Advanced spatial libraries	●	●
Import/export of industry-standard spatial data formats	●	●
Database mail	●	●

Management tools

Feature	Windows	Linux
SQL CMD (Command Prompt tool)	●	●
bcp Utility (Command Prompt tool)	●	●
SQL Server Management Studio	●	●
Azure Data Studio	●	●

SQL Server Related Services

Feature	Windows	Linux
SQL Server Agent	●	●
Integration Services	●	●
Master Data Services	●	–
Data Quality Services	●	–
Analysis Services	●	–
Reporting Services	●	–
SQL Server Browser	●	–
SQL Server R services	●	–

なお、対応機能の最新情報は次の Docs を確認ください。

◇ Windows 対応機能
https://docs.microsoft.com/ja-jp/sql/sql-server/editions-and-components-of-sql-server-version-15?view=sql-server-ver15

◇ Linux 対応機能
https://docs.microsoft.com/ja-jp/sql/linux/sql-server-linux-editions-and-components-2019?view=sql-server-ver15

以降は新たにサポートされた Machine Learning Services をピックアップして、サンプルコードを解説します。

➢ Machine Learning Services on Linux

1. 概要

SQL Server 2017 で Windows 版にのみ導入されていた Machine Learning Services が Linux 版にも対応しました。

2. 利用シーン

Machine Learning Services は「Bringing intelligence to where your data lives」というコンセプトのもと、SQL Server 2017 で導入された機能です。

これは、データサイエンス言語の代表である R/Python のランタイムを、SQL Server と統合させることで実現されます。

その目標は可用性、セキュリティ、性能などのリッチな機能を持つ SQL Server にデータサイエンスを相乗りさせることで、本番環境に耐えうるデータサイエンス実行基盤となることです。データや学習モデルがデータベースの外に出ないという安全性や応答性だけではなく、それらへのアクセスは SQL Server の持つセキュリティ機能（アクセス制御、暗号化、監査など）で守ることができます。また、リソースガバナーによるデータサイエンス処理のリソース制御や Always On による高可用構成など、本番環境に必要とされる要件を SQL Server の機能で実現することができます。

下図は SQL Server と R/Python 言語ランタイムの統合アーキテクチャです。

データサイエンスコードは SQL Server が提供する「sp_execute_external_script」システムストアドプロシージャを介して SQL Server に渡されます。通常は sp_execute_external_script とデータサイエンスコードをラップしたユーザー定義プロシージャを作成し、そのユーザー定義プロシージャを呼び出すことでデータサイエンスコードを実行させます。

3. やってみよう

それでは実際に Linux 上で Machine Learning Services を使った機械学習を実施してみましょう。

＊ 事前準備

事前準備として Machine Learning Services のインストールと検証用のデータベースを作成します。

Machine Learning Services をインストールする

Machine Learning Services を使用するためには、追加のパッケージをインストールする必要があります。インストール手順は Linux ディストリビューションごとに異なります。使用する環境に合わせて次の Docs を参考にインストールしてください。

◇ Linux に SQL Server Machine Learning Services (Python と R) をインストールする
https://docs.microsoft.com/ja-jp/sql/linux/sql-server-linux-setup-machine-learning?toc=%2Fsql%2Fadvanced-analytics%2Ftoc.json&view=sql-server-ver15

検証用のデータベースを作成する

検証用のデータベースとテーブルを作成します。

Machine_Learning_Services_on_Linux.sql

```
USE master;
GO
-- テスト用のデータベースを作成
```

```
CREATE DATABASE MLDB;
GO
USE MLDB
GO
-- 学習データ格納用のテーブルを作成
DROP TABLE IF EXISTS iris_data;
CREATE TABLE iris_data (
  [Sepal.Length] float,
  [Sepal.Width] float,
  [Petal.Length] float,
  [Petal.Width] float,
  Species nvarchar(100)
);
GO
-- モデル格納用のテーブルを作成
DROP TABLE IF EXISTS iris_model;
CREATE TABLE iris_model(
  model varbinary(max)
);
GO
-- 予測結果を格納するテーブルを作成
DROP TABLE IF EXISTS iris_predict;
CREATE TABLE iris_predict(
  setosa_prob float,
  versicolor_prob float,
  virginica_prob float,
  Species_Pred nvarchar(max),
  [Sepal.Length] float,
  [Sepal.Width] float,
  [Petal.Length] float,
  [Petal.Width] float,
  Species nvarchar(100)
);
GO
```

動作テストを行う

続けて下記のスクリプトを実行し「Hello Python !」が返ってくることを確認します。

Machine_Learning_Services_on_Linux.sql

```
EXECUTE sp_execute_external_script
  @language = N'Python',
  @script = N'print (''Hello Python !'')';
GO
```

環境によっては以下のエラーが発生する場合があります。

```
メッセージ 39012、レベル 16、状態 14、行 3
'Python' スクリプトのランタイムと通信できませんでした。要求 ID: C92496D7-F156-4919-B0D
D-B76B169C91E5。'Python' ランタイムの要件を確認してください。
外部スクリプトからの STDERR メッセージ:

Failed to load library /opt/mssql-extensibility/lib/sqlsatellite.so with error libc++.
so.1: cannot open shared object file: No such file or directory.
......
```

この場合は root でログインして以下のコマンドを実行してください。

Machine_Learning_Services_on_Linux.bash.txt

```
cp /opt/mssql/lib/libc++* /lib64
```

この問題についての詳細は次の Docs を確認ください。

◇ Linux で libc++.so を使用せずに sp_execute_external_script を使用するときのエラー
https://docs.microsoft.com/ja-jp/sql/advanced-analytics/known-issues-for-sql-server-machine-learning-services?view=sql-server-ver15#14-error-when-using-sp_execute_external_script-without-libcso-on-linux

✱ Machine Learning Services の動作を確認する

準備が完了したら、Machine Learning Services で R および Python のスクリプトが実行できることを確認してみましょう。

単純な R スクリプトを実行する

sp_execute_external_script プロシージャで R スクリプトを実行します。

Machine_Learning_Services_on_Linux.sql

```
-- R スクリプトで文字列を表示
EXECUTE sp_execute_external_script
  @language = N'R',
  @script = N'MyOutput<-MyInput',
  @input_data_1 = N'SELECT 1',
  @input_data_1_name = N'MyInput',
  @output_data_1_name = N'MyOutput'
WITH RESULT SETS(([Hello World] int));
GO
```

Results　Messages

	Hello World
1	1

sp_execute_external_script プロシージャの各引数の意味は以下の通りです。

引数	説明
@language	使用するスクリプト言語を示します。
@script	@language で指定した言語で記述されたスクリプトです。
@input_data_1	スクリプトにわたすデータセットの定義です。このデータセットは DataFrame として扱います。
@input_data_1_name	スクリプトにわたすデータセットの変数名です（既定値は InputDataSet）。
@output_data_1_name	スクリプトから返されるデータセットの変数名です（既定値は OutputDataSet）。

「WITH RESULT SETS」はスクリプトから返されるデータセットの列バインド定義です。今回は出力する結果セットの列名を「Hellow World」、データ型を「int」型で定義しています。

単純な Python スクリプトを実行する

続けて、下記のスクリプトを実行します。

Machine_Learning_Services_on_Linux.sql

```
-- Python スクリプトで文字列を表示
EXECUTE sp_execute_external_script
  @language = N'Python',
  @script = N'MyOutput = MyInput',
  @input_data_1 = N'SELECT 1',
  @input_data_1_name = N'MyInput',
  @output_data_1_name = N'MyOutput'
WITH RESULT SETS(([Hello World] int));
GO
```

Results　Messages

	Hello World
1	1

✱ R スクリプトを使った機械学習を体験してみる

続いて、R スクリプトを使った簡単な機械学習を実施してみましょう。

今回は iris（アヤメ）の品種を予測するモデルを作成とモデルを利用した予測を行います。

機械学習は以下の流れで実施します。

- アヤメ品種予測モデルを作成するための学習データを用意し SQL Server に登録する
- 学習データを利用してアヤメ品種予測モデルを作成し SQL Server に登録する
- 作成したアヤメ品種予測モデルを利用して新たなデータに対する品種を予測する

アヤメ品種予測モデルを作成するための学習データを用意し SQL Server に登録する

今回の学習データは R に組み込まれている iris データセットを流用するため、iris データセットを「iris_data」テーブルに格納します。

Machine_Learning_Services_on_Linux.sql

```
USE MLDB
GO
-- iris データセットをテーブルに格納
INSERT INTO iris_data
  EXEC sp_execute_external_script
    @language = N'R',
    @script = N'OutputDataSet <- iris';
GO
-- データの内容を確認
SELECT * FROM iris_data;
GO
-- データの件数を確認
SELECT COUNT(*) FROM iris_data;
GO
```

以下のように「iris_data」テーブルに格納されたデータの内容と格納した件数（150 件）が表示されることを確認します。

Results　Messages

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa

	(No column name)
1	150

この iris データセットはアヤメの品種（setosa、Versicolour、Virginica）毎の特徴を記録したデータで、次のような情報を持っています。

列名	説明
Sepal.Length	がく片（Sepal）の長さ
Sepal.Width	がく片（Sepal）の幅
Petal.Length	花びら（Petal）の長さ
Petal.Width	花びら（Petal）の幅
Species	アヤメの品種（setosa、Versicolour、Virginica）

学習データを利用してアヤメ品種予測モデルを作成し SQL Server に登録する

今回はランダムフォレストの rxDForest 関数によってモデルを作成します。また、作成したモデルは rxSerializeModel 関数でシリアライズ化し、「iris_model」テーブルに格納します。指定する引数の意味は以下の通りです。

❖ rxDForest(目的変数 ~ 説明変数１ ＋ 説明変数２ ＋ …, データ)

引数	説明
目的変数	予測の結果として求める値です。品種（Species）を指定します。
説明変数	予測するためのもとになる情報です。今回はがく片の長さ（Sepal.Length）、花びらの長さ（Petal.Length）、花びらの幅（Petal.Width）を指定します。
データ	学習に利用するデータセットです。今回は SQL Server 上の「iris_data」テーブルのデータを渡します。

Machine_Learning_Services_on_Linux.sql

```
USE MLDB
GO
-- クリーンアップ
TRUNCATE TABLE iris_model
GO
-- モデルを作成して iris_model テーブルに格納
INSERT INTO iris_model
  EXEC sp_execute_external_script
    @language = N'R',
    @script = N'
# モデルを作成し model1 に格納
model1 <- rxDForest (Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Widt
h, data = InputDataSet)
# rxSerializeModel でシリアライズ化したものを data.frame でデータフレームに変換し、
# OutputDataSet に格納して sp_execute_external_script の実行結果として返却
OutputDataSet <- data.frame(rxSerializeModel(model1, realtimeScoringOnly = TRUE))
',
    @input_data_1 = N'SELECT * FROM iris_data';
GO
-- 保存したモデルを確認
SELECT * FROM iris_model;
GO
```

「model」テーブルにバイナリ形式のモデルが登録されていることを確認します。

Results　Messages

	model
1	0x626C6F621A51B2C2C050E0CDA3…

アヤメ品種予測モデルを利用して新たなデータに対する品種を予測する

「iris_model」テーブルに格納したモデルを使って、「iris_data」テーブルのがく片、花びらの長さと幅のデータから品種を予測してみましょう。

予測には T-SQL の PREDICT 関数を使います。PREDICT 関数に指定する引数は以下の通りです。

引数	説明
MODEL	予測に使用するモデルを指定します。今回は「iris_model」テーブルに格納したモデルを取得して使用します。
DATA	予測に使用するためのデータを格納したテーブル、ビュー、またはデータを返すストアドプロシージャを指定します。今回はモデルの作成に使用したものと同じ「iris_data」テーブルを使用します。

PREDICT 関数の結果は、品種への該当確率が「setosa_Pred」「versicolor_Pred」「virginica_Pred」の各列に表示されます。そのままだと確認しづらいため、以下では最後のクエリで最も該当確率が高いものを判定して出力した上で、実際のデータ上の品種と比較して一致（o）、不一致（x）を出力します。

Machine_Learning_Services_on_Linux.sql

```
USE MLDB;
GO
-- クリーンアップ
TRUNCATE TABLE iris_predict;
GO
-- モデルを格納する変数を定義
DECLARE @model varbinary(max);

-- モデルを取得して変数に格納
SELECT @model = model FROM iris_model;

-- 取得したモデルと iris_data テーブル格納されたデータを使って予測（PREDICT）を実施
INSERT INTO iris_predict
  SELECT * FROM PREDICT ( MODEL = @model, DATA = iris_data )
    WITH (setosa_prob float,
          versicolor_prob float,
          virginica_prob float,
          Species_Pred nvarchar(max)) AS p;
GO
-- 予測精度を計算して出力
```

```
SELECT [○], [×], ([○] / 150.00 * 100) AS [予測精度]
FROM (
  SELECT
    SUM(CASE WHEN Species_Pred = species THEN 1 ELSE 0 END) AS [○],
    SUM(CASE WHEN Species_Pred != species THEN 1 ELSE 0 END) AS [×]
  FROM iris_predict
) AS iris_result;
GO
-- 予測結果を出力
SELECT
  Species_Pred AS [予測結果],
  Species AS [実際の品種],
  [Sepal.Length],
  [Sepal.Width],
  [Petal.Length],
  [Petal.Width]
FROM iris_predict;
GO
```

予測精度と実際の予測結果が以下のように出力されます。

Results　Messages

	○	×	予測精度
1	144	6	96.000000

	予測結果	実際の品種	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
1	setosa	setosa	5.1	3.5	1.4	0.2
2	setosa	setosa	4.9	3	1.4	0.2
3	setosa	setosa	4.7	3.2	1.3	0.2
4	setosa	setosa	4.6	3.1	1.5	0.2
5	setosa	setosa	5	3.6	1.4	0.2
6	setosa	setosa	5.4	3.9	1.7	0.4
7	setosa	setosa	4.6	3.4	1.4	0.3
8	setosa	setosa	5	3.4	1.5	0.2
9	setosa	setosa	4.4	2.9	1.4	0.2

いくつか予測間違いも検出されますが、概ね正しい予測結果が得られることが確認できます。

＊ Python スクリプトを使った機械学習を体験してみる

続いて、Python スクリプトを使った簡単な機械学習を実施してみましょう。

R スクリプトと同様に、iris（アヤメ）の品種を予測するモデルの作成と、モデルを利用した予測を行います。

機械学習は以下の流れで実施します。

- アヤメ品種予測モデルを作成するための学習データを用意し SQL Server に登録する
- 学習データを利用してアヤメ品種予測モデルを作成し SQL Server に登録する
- アヤメ品種予測モデルを利用して新たなデータに対する品種を予測する

アヤメ品種予測モデルを作成するための学習データを用意し SQL Server に登録する

先の R の手順で作成した「iris_data」テーブルを利用します。

学習データを利用してアヤメ品種予測モデルを作成し SQL Server に登録する

今回はランダムフォレストの rx_dforest 関数によってモデルを作成します。また、作成したモデルは rxSerializeModel 関数でシリアライズ化し、「iris_model」テーブルに格納します。

Machine_Learning_Services_on_Linux.sql

```
USE MLDB;
GO
-- クリーンアップ
TRUNCATE TABLE iris_model;
GO
-- モデルを作成して model テーブルに格納
INSERT INTO iris_model
  EXEC sp_execute_external_script
    @language = N'Python',
    @script = N'
import pandas as pd
from revoscalepy import rx_dforest, rx_serialize_model
InputDataSet["Species"] = InputDataSet["Species"].astype("category")

# モデルを作成し model1 に格納
model1 = rx_dforest ("Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Widt
h", data = InputDataSet)

# rx_serialize_model でシリアライズ化したものを iris_model に格納
iris_model = rx_serialize_model(model1, realtime_scoring_only = True)
# OutputDataSet に格納して sp_execute_external_script の実行結果として返却
OutputDataSet = pd.DataFrame(data={"model": [iris_model]})
',
    @input_data_1 = N'SELECT * FROM iris_data';
GO
-- 保存したモデルを確認
SELECT * FROM iris_model;
GO
```

「iris_model」テーブルにバイナリ形式のモデルが登録されていることを確認します。

Results　Messages

	model
1	0x626C6F62B58CAB9A7F5D5EAADB…

アヤメ品種予測モデルを利用して新たなデータに対する品種を予測する

R と同様に、T-SQL の PREDICT 関数を使って予測を実施してみましょう。

使用するクエリは R のものと同じ下記スクリプトです。

Machine_Learning_Services_on_Linux.sql

```
USE MLDB;
GO
-- クリーンアップ
TRUNCATE TABLE iris_predict;
GO
-- モデルを格納する変数を定義
DECLARE @model VARBINARY(MAX);

-- モデルを取得して変数に格納
SELECT @model = model FROM iris_model;

-- 取得したモデルと iris_data テーブル格納されたデータを使って予測（PREDICT）を実施
INSERT INTO iris_predict
  SELECT * FROM PREDICT ( MODEL = @model, DATA = iris_data )
    WITH ( setosa_prob float,
           versicolor_prob float,
           virginica_prob float ,
           Species_Pred nvarchar(max)) AS p;
GO
-- 予測精度を計算して出力
SELECT [○], [×], ([○] / 150.00 * 100) AS [予測精度]
FROM (
  SELECT
    SUM(CASE WHEN Species_Pred = species THEN 1 ELSE 0 END) AS [○],
    SUM(CASE WHEN Species_Pred != species THEN 1 ELSE 0 END) AS [×]
  FROM iris_predict
) AS iris_result;
GO
-- 予測結果を出力
SELECT
  Species_Pred AS [予測結果],
  Species AS [実際の品種],
  [Sepal.Length],
  [Sepal.Width],
  [Petal.Length],
  [Petal.Width]
FROM iris_predict;
GO
```

予測精度と実際の予測結果が以下のように出力されます。

Results　Messages

	○	×	予測精度
1	144	6	96.000000

	予測結果	実際の品種	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
1	setosa	setosa	5.1	3.5	1.4	0.2
2	setosa	setosa	4.9	3	1.4	0.2
3	setosa	setosa	4.7	3.2	1.3	0.2
4	setosa	setosa	4.6	3.1	1.5	0.2
5	setosa	setosa	5	3.6	1.4	0.2
6	setosa	setosa	5.4	3.9	1.7	0.4
7	setosa	setosa	4.6	3.4	1.4	0.3
8	setosa	setosa	5	3.4	1.5	0.2
9	setosa	setosa	4.4	2.9	1.4	0.2

いくつか予測間違いも検出されますが、概ね正しい予測結果が得られることが確認できます。

6.2. SQL Server 2019 on Linux Container with Docker

ここでは SQL Server コンテナーの基本の解説に交えて、SQL Server 2019 on Linux Container with Docker の強化ポイントを解説します。

➢ Why SQL Server Containers?

SQL Server コンテナーの主な活用ポイントは４つです。

- Portable

  コンテナーをサポートするプラットフォーム（Windows、Linux、macOS、Kubernetes）であればどこでも稼働させることができるため、高い移植性を維持でき、状況に応じてより軽快にプラットフォームを選択できるようになります。

- Lightweight

  コンテナーは単にホストコンピューター上のプロセスです。そのため、OS のイメージを完全に内包する仮想マシンと比較して遥かに軽量です。また、コンテナープロセスの元となるコンテナーイメージは、イメージレイヤと呼ばれる層の集合体で構成されており、共通するイメージレイヤはコンテナーイメージ間で共有されるため、同一ホスト上で複数のコンテナーを実行する際のフットプリントとリソースが削減されます。

- Consistent

  Portable で Lightweight になったことで、よりタイムリーかつブロードな配布が容易になります。例えば、開発やテスト用のデータベースはその多くが開発者間で共有利用されるため、共有で問題を起

こさないようにするためのルールが生産性を下げているケースはよくありました。しかしコンテナーであれば Portable で Lightweight であるため、開発者個々にデータベースを配布することが容易になります。データベース管理者は配布する SQL Server コンテナーイメージを作成し、開発者は個々にそれを入手し使用します。また、Windows と Linux のそれぞれのコンテナーイメージを用意しておくことで、開発者はプラットフォームに縛られることなくデータベースを自分の環境の中に構成できるようになります。こうしたシナリオは DevOps において非常に重要です。

- Efficient

  コンテナーによる SQL Server は極めて短時間で起動ができます。また、パッチについても極めて迅速に適用でき、さらに万が一のロールバックも容易であるため、ダウンタイムを削減します。この点については以降のサンプルコードを試してぜひ実感してください。

➢ コンテナーのライフサイクル

コンテナーの起点はコンテナーイメージです。コンテナーイメージはコンテナーレジストリから入手（pull）するか、Dockerfile という定義ファイルをビルド（build）することで、独自のコンテナーイメージを作成することもできます。このコンテナーイメージを起動（run）することでコンテナーが作成され、同時にコンテナープロセスが起動します。コンテナーの OS 環境内でコマンドを実行（exec）することもできます。起動したコンテナープロセスは停止（stop）や再開（start）を行うことができ、作成したコンテナーが不要になれば破棄（rm）することもできます。また、作成したコンテナーを元にしたイメージ化（commit）することも可能で、新たにイメージ化したコンテナーイメージは、コンテナーレジストリに配信（push）することで共有されます。

docker ...	解説
build	コンテナイメージ作成
pull	コンテナイメージ取得
run	コンテナ作成＆起動
stop	コンテナ停止
start	コンテナ起動
exec	指定したコマンドを コンテナOS上で実行
rm	コンテナ破棄
commit	コンテナのイメージ化
push	コンテナイメージ配信
rmi	コンテナイメージ破棄

➢ SQL Server コンテナーの基本操作

いくつかの具体的なコマンドイメージ例を紹介しながら、SQL Server コンテナーのライフサイクルを解説します。

SQL Server コンテナーイメージの入手（push）

Microsoft Container Registry から入手します。SQL Server 2017 コンテナーの公開初期は、Docker 社が提供するコンテナーレジストリである「Docker Hub Registry」によって公開されていましたが、現在はマイクロソフト社が提供するコンテナーレジストリである「Microsoft Container Registry」で公開されています。なお、「Docker Hub Registry」でも引き続き SQL Server コンテナーの検索が可能です。

「Microsoft Container Registry」での SQL Server コンテナーイメージのアドレスは次の通りです。

◇ SQL Server for Red Hat Enterprise Linux コンテナーイメージ
mcr.microsoft.com/mssql/rhel/server:<tag>

◇ SQL Server for Ubuntu コンテナーイメージ
mcr.microsoft.com/mssql/server:<tag>

◇ SQL Server for SUSE Enterprise Linux Server コンテナーイメージ
現時点では SUSE 用の SQL Server コンテナーイメージは公開されていないため、独自に作成する必要があります。SUSE 用の SQL Server コンテナーイメージを作成するには、次のドキュメントが参考になります。
https://github.com/microsoft/mssql-docker/tree/master/linux/preview/SLES

<tag>は SQL Server のバージョンを指定します。指定できる<tag>は次のアドレスで入手できます。

◇ SQL Server for Red Hat Enterprise Linux コンテナーイメージ
https://mcr.microsoft.com/v2/mssql/rhel/server/tags/list
RedHat 用の SQL Server コンテナーイメージは、SQL Server 2019 から新しく公開されるようになっています。

◇ SQL Server for Ubuntu コンテナーイメージ
https://mcr.microsoft.com/v2/mssql/server/tags/list

Ubuntu 用の SQL Server コンテナーイメージは従来の SQL Server 2017 に加えて SQL Server 2019 も新しく公開されています。

例えば、次の「docker pull」コマンドで SQL Server 2017 CU19 for Ubuntu コンテナーイメージを入手します。

Container_Basic_Operation.bash.txt

```
# docker pull mcr.microsoft.com/mssql/server:2017-CU19-ubuntu-16.04
```

```
~ % sudo docker pull mcr.microsoft.com/mssql/server:2017-CU19-ubuntu-16.04
2017-CU19-ubuntu-16.04: Pulling from mssql/server
59ab41dd721a: Pull complete
57da90bec92c: Pull complete
06fe57530625: Pull complete
5a6315cba1ff: Pull complete
739f58768b3f: Pull complete
3a58fde0fc61: Pull complete
89b44069090d: Pull complete
3f6b360deb9e: Pull complete
b8d0242f03c2: Pull complete
Digest: sha256:f551ff159479f402bc3a120d9bb509c78bbcb8d1fe401552e102f28bc654db0e
Status: Downloaded newer image for mcr.microsoft.com/mssql/server:2017-CU19-ubuntu-16.04
mcr.microsoft.com/mssql/server:2017-CU19-ubuntu-16.04
```

SQL Server コンテナーの作成と起動（run）

入手した SQL Server 2017 CU19 for Ubuntu コンテナーイメージからコンテナーを作成し起動します。

Container_Basic_Operation.bash.txt

```
# docker run -e ACCEPT_EULA=Y -e MSSQL_SA_PASSWORD=P@ssw0rd -p 11433:1433 -v ~/DataVol
ume:/var/opt/mssql --name sql17cu19 -d mcr.microsoft.com/mssql/server:2017-CU19-ubuntu
-16.04
```

パラメーターの意味は次の通りです。

 # docker run

-e 'ACCEPT_EULA=Y'	SQL Server用の環境変数
-e 'MSSQL_SA_PASSWORD=P@ssw0rd'	同上
-p 11433:1433	ホストポートとコンテナポートのマッピング
-v ~/DataVolume:/var/opt/mssql	ホストボリュームのマッピング（データ永続化）
--name sql17cu19	コンテナ名
-d	コンテナをバックグラウンド起動
mcr.microsoft.com/mssql/server:2017-CU19-ubuntu-16.04	コンテナの元となるイメージ

```
~ % sudo docker run -e ACCEPT_EULA=Y -e MSSQL_SA_PASSWORD=P@ssw0rd -p 11433:1433
 -v ~/DataVolume:/var/opt/mssql --name sql17cu19 -d mcr.microsoft.com/mssql/server:2017-CU19-ubu
ntu-16.04
f87996482b537f4fd3b9a81fe913fe04da9afa4f756a73b3647965ef469c2e33
~ %
```

ここまでのわずかの操作で SQL Server が起動します。

次の「docker ps」コマンドは SQL Server コンテナープロセスが起動していることを Docker エンジンに照会しています。

Container_Basic_Operation.bash.txt

```
# docker ps -a --filter name=sql17cu19
```

STATUS が「Up」となっていることでコンテナープロセスが起動していることが確認できます。

SQL Server コンテナーへの接続

起動した SQL Server に sqlcmd で接続し、「SELECT @@VERSION」を実行します。
なお、この SQL Server コンテナーは起動時に「-p 11433:1433」を指定しました。すなわちコンテナー上の 1433 ポートで待ち受けている SQL Server に接続するためにはホストの 11433 ポートに接続する必要があります。
また、ここでの localhost とはコンテナーをホストしているサーバーであることに注意してください。

Container_Basic_Operation.bash.txt

```
# sqlcmd -S localhost,11433 -U sa -P P@ssw0rd -Q 'select @@version'
```

もう一つの接続例を示します。次の「docker exec」コマンドは指定したコマンドをコンテナーの OS 環境内で実行します。

Container_Basic_Operation.bash.txt

```
# docker exec sql17cu19 /opt/mssql-tools/bin/sqlcmd -S localhost,1433 -U sa -P P@ssw0r
d -Q 'select @@version'
```

このコマンドの意味は、SQL Server コンテナーの OS 環境内にインストールされた sqlcmd を、SQL Server コンテナーの OS 環境内で実行します。すなわち、ここでの localhost とはコンテナー自身を指しており、SQL Server へ接続するポートは 1433 となります。

ここでは sqlcmd を利用していますが、SSMS や ADS など他の SQL Server クライアントからも同様に、従来どおりの SQL Server として接続が可能です。

SQL Server コンテナーの停止と再開（stop & start）

起動した SQL Server コンテナーを停止するには「docker stop」コマンドです。

Container_Basic_Operation.bash.txt

```
# docker stop sql17cu19
# docker ps -a --filter name=sql17cu19
```

STATUS が「Exited」となっていることで、コンテナープロセスが停止していることが確認できます。なお、「停止(stop)」が、作成したコンテナーを削除することはありません。そのため、コンテナー上で生成されたデータも残り、コンテナーの再開によって再度アクセスが可能です。

停止した SQL Server コンテナーを再開するには「docker start」コマンドです。

Container_Basic_Operation.bash.txt

```
# docker start sql17cu19
# docker ps -a --filter name=sql17cu19
```

```
~ % sudo docker start sql17cu19
sql17cu19
~ % sudo docker ps -a --filter name=sql17cu19
CONTAINER ID   IMAGE                                          COMMAND                  CREATED          STATUS          PORTS                    NAMES
39b6d2ccdebf   mcr.microsoft.com/mssql/server:2017-CU19-ubuntu-16.04   "/opt/mssql/bin/nonr…"   2 minutes ago    Up 4 seconds    0.0.0.0:11433->1433/tcp  sql17cu19
~ %
```

STATUS が「Up」に遷移していることが確認できます。

SQL Server コンテナーの OS 環境操作（exec）

先の手順で「docker exec」コマンドを利用し、指定したコマンドをコンテナーの OS 環境内で実行しましたが、「docker exec」コマンドにはもう一つの使い方はコンテナーの OS 環境を操作することです。

次のコマンドで OS 環境につなぎます。

Container_Basic_Operation.bash.txt

```
# docker exec -it sql17cu19 /bin/bash
```

先の手順とは異なり、「-it」オプションを設定しています。「-i」は標準入力を開き続けること、「-t」は疑似 TTY を割り当てることを意味しており、この結果、コンテナーOS 環境へのターミナル接続相当になります。

```
~ % sudo docker exec -it sql17cu19 /bin/bash
root@39b6d2ccdebf:/#
```

例えば、ホスト上の「uname -n」で返るノード名と、「docker exec -it」後のコンテナーOS 環境上の「uname -n」で返るノード名が異なることが確認できます。

```
@ubuntu ~ % uname -n
ubuntu
@ubuntu ~ % sudo docker exec -it sql17cu19 /bin/bash
root@39b6d2ccdebf:/# uname -n
39b6d2ccdebf
root@39b6d2ccdebf:/#
```

コンテナーOS 環境上で「ps」コマンド打つと SQL Server プロセスが起動していることも確認できます。

```
root@39b6d2ccdebf:/# ps -auxf
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root       9149  2.8  0.0  18232  3296 pts/0    Ss   12:27   0:00 /bin/bash
root       9165  0.0  0.0  34420  2892 pts/0    R+   12:27   0:00  \_ ps -auxf
root          1  0.0  0.1 130160 25976 ?        Ssl  Mar02   0:02 /opt/mssql/bin/sqlservr
root          9  1.1  5.6 9188592 921808 ?      Sl   Mar02  13:35 /opt/mssql/bin/sqlservr
root@39b6d2ccdebf:/#
```

このコンテナーOS 環境はホスト OS から独立したものであるため、例えばコンテナーOS 環境でパッケージをインストールすると、あくまでもコンテナーOS 環境でのインストールになります。

例として、コンテナーOS 環境にターミナル接続し次のコマンドで mssql-cli ツールをインストールしてみます。

Container_Basic_Operation.bash.txt

```
host # docker exec -it sql17cu19 /bin/bash
container # apt-get update
container # apt-get install mssql-cli
container # which mssql-cli
container # mssql-cli -S localhost,1433 -U sa -P P@sw0rd
```

```
Get:13 http://archive.ubuntu.com/ubuntu xenial/main amd64 Packages [1558 kB]
Get:14 http://archive.ubuntu.com/ubuntu xenial/restricted amd64 Packages [14.1 kB]
Get:15 http://archive.ubuntu.com/ubuntu xenial/universe amd64 Packages [9827 kB]
Get:16 http://archive.ubuntu.com/ubuntu xenial/multiverse amd64 Packages [176 kB]
Get:17 http://archive.ubuntu.com/ubuntu xenial-updates/universe Sources [337 kB]
Get:18 http://archive.ubuntu.com/ubuntu xenial-updates/main amd64 Packages [1433 kB]
Get:19 http://archive.ubuntu.com/ubuntu xenial-updates/restricted amd64 Packages [13.1 kB]
Get:20 http://archive.ubuntu.com/ubuntu xenial-updates/universe amd64 Packages [1022 kB]
Get:21 http://archive.ubuntu.com/ubuntu xenial-updates/multiverse amd64 Packages [19.3 kB]
Get:22 http://archive.ubuntu.com/ubuntu xenial-backports/main amd64 Packages [7942 B]
Get:23 http://archive.ubuntu.com/ubuntu xenial-backports/universe amd64 Packages [8807 B]
Fetched 26.8 MB in 14s (1798 kB/s)
Reading package lists... Done
root@39b6d2ccdebf:/# apt-get install mssql-cli
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  mssql-cli
0 upgraded, 1 newly installed, 0 to remove and 16 not upgraded.
Need to get 54.3 MB of archives.
After this operation, 253 MB of additional disk space will be used.
Get:1 https://packages.microsoft.com/ubuntu/16.04/prod xenial/main amd64 mssql-cli all 0.15.0-1 [54.3 MB]
Fetched 54.3 MB in 7s (7596 kB/s)
debconf: delaying package configuration, since apt-utils is not installed
Selecting previously unselected package mssql-cli.
(Reading database ... 8879 files and directories currently installed.)
Preparing to unpack .../mssql-cli_0.15.0-1_all.deb ...
Unpacking mssql-cli (0.15.0-1) ...
Setting up mssql-cli (0.15.0-1) ...
root@39b6d2ccdebf:/# which mssql-cli
/usr/bin/mssql-cli
root@39b6d2ccdebf:/#
```

コンテナーOS 環境の 「/usr/bin/mssql-cli 」にインストールされたことが確認できます。

次にコンテナーから抜け、ホスト上で「/usr/bin/mssql-cli」を検索してみます。

Container_Basic_Operation.bash.txt

```
host # ls -la /usr/bin/mssql-cli
```

ホスト上の「/usr/bin/mssql-cli」には何も存在しないことが確認できます。

SQL Server コンテナーの DB 作成と永続化

この SQL Server コンテナーは起動時に「-d ~/DataVolume:/var/opt/mssql」を指定しました。これは、コンテナーOS 環境内の「/var/opt/mssql（データファイルが作成されるディレクトリ）」へホスト側の「~/DataVolume」をマウントすることを指定しています。
ここでは、この指定がどのような意味なのかを確認してみます。

まず、SQL Server コンテナーにデータベースを作成しデータを投入します。

Container_Basic_Operation.bash.txt

```
# sqlcmd -S localhost,11433 -U sa -P P@ssw0rd -Q 'create database DevDB'
# sqlcmd -S localhost,11433 -U sa -P P@ssw0rd -Q 'use DevDB; create table tab01(id in
t, name varchar(max)); insert into tab01 values (1, "SQL Server runs on Docker!")'
# sqlcmd -S localhost,11433 -U sa -P P@ssw0rd -Q 'select name from sys.databases; use
DevDB; select * from tab01'
```

次に SQL Server コンテナーを削除「docker rm」します。

Container_Basic_Operation.bash.txt

```
# docker rm -f sql17cu19
# docker ps -a --filter name=sql17cu19
```

```
~ % sudo docker rm -f sql17cu19
sql17cu19
~ % sudo docker ps -a --filter name=sql17cu19
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
~ %
```

コンテナーを削除したため、それ伴ってコンテナー内のデータはすべて無くなります。
ただし、このコンテナーは起動時に「-d ~/DataVolume:/var/opt/mssql」を指定したため、コンテナーOS 環境内の「/var/opt/mssql（データファイルが作成されるディレクトリ）」はホスト側の「~/DataVolume」に永続化されています。

次のコマンドでホスト側の「~/DataVolume」を確認します。

Container_Basic_Operation.bash.txt

```
# ls -la ~/DataVolume/data
```

```
~ % ls -la ~/DataVolume/data
total 69512
drwxr-xr-x 2 root root     4096 Mar  2 15:58 .
drwxrwxr-x 6 o9o9 o9o9     4096 Mar  2 15:18 ..
-rw-r----- 1 root root  8388608 Mar  2 16:02 DevDB.mdf
-rw-r----- 1 root root  8388608 Mar  2 16:02 DevDB_log.ldf
-rw-r----- 1 root root  4194304 Mar  2 16:02 master.mdf
-rw-r----- 1 root root  2097152 Mar  2 16:02 mastlog.ldf
-rw-r----- 1 root root  8388608 Mar  2 16:02 model.mdf
-rw-r----- 1 root root  8388608 Mar  2 16:02 modellog.ldf
-rw-r----- 1 root root 14024704 Mar  2 16:02 msdbdata.mdf
-rw-r----- 1 root root   524288 Mar  2 16:02 msdblog.ldf
-rw-r----- 1 root root  8388608 Mar  2 15:26 tempdb.mdf
-rw-r----- 1 root root  8388608 Mar  2 15:31 templog.ldf
~ %
```

先に作成したデータベースのデータファイルを含めてデータが永続化されていることが確認できます。もし、ホストボリュームをマウントしない場合には、コンテナーの削除によってコンテナー上で生成されたデータもすべて無くなります。

SQL Server コンテナーのアップグレード

前の手順で作成した SQL Server 2017 コンテナーを、SQL Server の最新バージョンである SQL Server 2019 にアップグレードしてみます。

まず、SQL Server 2019 コンテナーイメージを入手します。

Container_Basic_Operation.bash.txt

```
# docker pull mcr.microsoft.com/mssql/server:2019-GA-ubuntu-16.04
# docker image list
```

```
~ % sudo docker pull mcr.microsoft.com/mssql/server:2019-GA-ubuntu-16.04
2019-GA-ubuntu-16.04: Pulling from mssql/server
59ab41dd721a: Already exists
57da90bec92c: Already exists
06fe57530625: Already exists
5a6315cba1ff: Already exists
739f58768b3f: Already exists
fd449e8d7345: Pull complete
51d0933375e5: Pull complete
64f21ba81504: Pull complete
55b6919c0cc6: Pull complete
Digest: sha256:c8fa22553ce421b0482febcafa712b29cbb933b0d97a8671686797b31cf157a9
Status: Downloaded newer image for mcr.microsoft.com/mssql/server:2019-GA-ubuntu-16.04
mcr.microsoft.com/mssql/server:2019-GA-ubuntu-16.04
~ % sudo docker image list
REPOSITORY                        TAG                    IMAGE ID       CREATED        SIZE
mcr.microsoft.com/mssql/server    2017-CU19-ubuntu-16.04 a8343d3ce21c   4 weeks ago    1.39GB
mcr.microsoft.com/mssql/server    2019-GA-ubuntu-16.04   76c7c66bff02   4 months ago   1.57GB
~ %
```

次に現在稼働している SQL Server 2017 を停止します。

Container_Basic_Operation.bash.txt

```
# docker stop sql17cu19
# docker ps -a --filter name=sql17cu19
```

```
~ % sudo docker stop sql17cu19
sql17cu19
~ % sudo docker ps -a --filter name=sql17cu19
CONTAINER ID   IMAGE                                      COMMAND                  CREATED          STATUS                     PORTS   NAMES
f87996482b53   mcr.microsoft.com/mssql/server:2017-CU19-ubuntu-16.04   "/opt/mssql/bin/nonr…"   37 minutes ago   Exited (0) 40 seconds ago          sql17cu19
~ %
```

新たに SQL Server 2019 コンテナーを作成し起動します。

Container_Basic_Operation.bash.txt

```
# docker run -e ACCEPT_EULA=Y -e MSSQL_SA_PASSWORD=P@ssw0rd -u 0:0 -p 11433:1433 -v ~/
DataVolume:/var/opt/mssql --name sql19ga -d mcr.microsoft.com/mssql/server:2019-GA-ubu
ntu-16.04
```

```
~ % sudo docker run -e ACCEPT_EULA=Y -e MSSQL_SA_PASSWORD=P@ssw0rd -u 0:0 -p 11433:1433 -v ~/DataVolume:/var/opt/mssql --name sql19ga -d mcr.microsoft.com/mssql/server:
2019-GA-ubuntu-16.04
ea2a6409fbf4c1d0220a814df3e958158ddedcc90712f0bdeab91b6d4e979e10
~ % sudo docker ps -a
CONTAINER ID   IMAGE                                      COMMAND                  CREATED          STATUS                     PORTS                    NAMES
ea2a6409fbf4   mcr.microsoft.com/mssql/server:2019-GA-ubuntu-16.04   "/opt/mssql/bin/perm…"   11 seconds ago   Up 4 seconds               0.0.0.0:11433->1433/tcp  sql19ga
39b6d2ccdebf   mcr.microsoft.com/mssql/server:2017-CU19-ubuntu-16.04   "/opt/mssql/bin/nonr…"   21 hours ago     Exited (0) 9 minutes ago                            sql17cu19
~ %
```

ポイントは、SQL Server 2017 のデータベースの永続化のために使用したホストボリュームを、そのまま指定している点です。これは SQL Server 2017 で作成したデータベースを、SQL Server 2019 にアタッチすることに相当します。その他のオプションも SQL Server 2017 で指定したものと同様ですが、一点だけ異なるのは「-u 0:0」オプションの追加です。これについては後半の「Non-root Container」で詳細を解説するのでここでは気にせず手順を進めてください。

起動した SQL Server 2019 に接続します。

Container_Basic_Operation.bash.txt

```
# sqlcmd -S localhost,11433 -U sa -P P@ssw0rd -Q 'select @@version'
```

```
~ % sqlcmd -S localhost,11433 -U sa -P P@ssw0rd -Q 'select @@version'
Sqlcmd: Error: Microsoft ODBC Driver 17 for SQL Server : Login failed for user 'sa'. Reason:
Server is in script upgrade mode. Only administrator can connect at this time..
~ %
```

SQL Server に古いフォーマットのデータベースをアタッチすると、その起動直後にアップグレードプロセスに入ります。この振る舞いは SQL Server コンテナーも同様です。上の手順で SQL Server 2019 起動直後に接続するとエラーとなり、その理由として「Server is in script upgrade mode.」が返されます。

アップグレードプロセスが完了するのをしばらく待った後にあらためて接続します。

Container_Basic_Operation.bash.txt

```
# sqlcmd -S localhost,11433 -U sa -P P@ssw0rd -Q 'select name from sys.databases; use
DevDB; select * from tab01'
```

```
~ % sqlcmd -S localhost,11433 -U sa -P P@ssw0rd -Q 'select @@version'sqlcmd -S localhost,11433 -U sa -P P@ssw0rd
 -Q 'select name from sys.databases; use DevDB; select * from tab01'
name
--------------------------------------------------------------------------------------------------------------------------------
master
tempdb
model
msdb
DevDB

(5 rows affected)
Changed database context to 'DevDB'.
id          name


----------- ----------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------
------------
          1 SQL Server runs on Docker!
```

SQL Server 2017 のデータベースが SQL Server 2019 のデータベースとしてアップグレードされていることが確認できます。

SQL Server コンテナーのカスタムイメージの作成（build）

冒頭で解説した通り、Portable で Lightweight なコンテナーによって開発者個々にデータベースを配布することが容易になりました。データベース担当者は、独自にカスタムした SQL Server コンテナーイメージを作成し、コンテナーレジストリを介して開発者個々に配布します。

コンテナーイメージは Dockerfile という定義ファイルを作成し、それを「docker build」コマンドに渡すことで作成します。

コンテナイメージを自分で作る

Dockerfile＋docker build

Dockerfileはコンテナイメージの定義ファイル（YAML形式）。
docker buildはDockerfile内の命令に従いコンテナイメージを作成。

命令（一部）	解説
COPY	指定したソースから、指定した対象のコンテナーにファイルがコピーされます。
CMD	コンテナーの既定の実行可能ファイルが指定されます。 各 Dockerfile には、CMD 命令が 1 つのみ存在します。
ENV	コンテナー内の環境変数の値が設定されます。
EXPOSE	コンテナーが特定の TCP ポートまたは UDP ポートでリッスンすることが Docker に通知されます。
RUN	現在のイメージの新しいレイヤーでコマンドが実行され、結果が新しいイメージにコミットされます。
WORKDIR	CMD、RUN、およびコマンドを実行するその他の命令のコンテナー内に作業ディレクトリが設定されます。

オリジナルの SQL Server コンテナーイメージの Dockerfile イメージは次の通りです。

SQL Server オリジナル Dockerfile

FROM <OS base image>	例：ubuntu:16.04
LABEL < Microsoft label info >	例：vendor="Microsoft"
EXPOSE 1433	コンテナ内のSQL Serverがリッスンするポート
COPY < SQL Server bin and lib >	/opt/mssql/bin、/opt/mssql/lib
RUN ./install.sh	依存パッケージやツールのインストール
CMD ["/opt/mssql/bin/sqlservr"]	SQL Serverの起動

例えば、独自に SQL Server カスタムする Dockerfile の例は次の通りです。

Sample1 はバックアップファイルを SQL Server コンテナーイメージに内包させ、利用者にバックアップファイルをリストアさせることで、共通のデータベースを配布するシンプルな方法の例です。Sample2 はさらに踏み込んだ発展型で、docker build の中でデータベース作成とスキーマ定義を行うスクリプトを実行させることで、共通のデータベースを配布する方法の例です。

以降は Sample1 の方法で独自の SQL Server コンテナーイメージを作成する手順を解説します。
まず、コンテナーイメージに内包させたいデータベースバックアップファイルを準備します。ここでは前の手順で作成したデータベースのバックアップを行います。

Container_Basic_Operation.bash.txt

```
# sqlcmd -S localhost,11433 -U sa -P P@ssw0rd -Q 'backup database DevDB to disk="/tmp/DevDB.bak" with format'
```

コンテナー内に作成したバックアップファイルを「docker cp」コマンドによりホスト側にコピーします。

Container_Basic_Operation.bash.txt

```
# docker cp sql19rc1:/tmp/DevDB.bak ~/SQL_Server_Container/DevDB.bak
```

次に Dockerfile を作成します。

Container_Basic_Operation.bash.txt

```
# touch Dockerfile
# echo 'FROM mcr.microsoft.com/mssql/server:2019-CU2-ubuntu-16.04'    >> Dockerfile
# echo 'COPY ./DevDB.bak /tmp/DevDB.bak'
                                >> Dockerfile
# echo 'CMD ["/opt/mssql/bin/sqlservr"]'
                                >> Dockerfile
# cat Dockerfile
```

作成した Dockerfile で、「docker build」コマンドによるコンテナーイメージを作成します。

Container_Basic_Operation.bash.txt

```
# docker build -t sql19cu2/devdb:v1.0 .
```

少々時間を要すためコマンドが完了するのをしばらく待ちます。

作成したコンテナーイメージは、コンテナーレジストリを介して共有します。Docker 社が提供するコンテナーレジストリを利用したり、Azure Container Registry などのクラウドサービスを利用する方法などがありますが、ここではローカルの環境にコンテナーレジストリを構築してみます。コンテナーレジストリ自体もコンテナーイメージとして Apache License 2.0 で公開されているので、これを利用します。具体的には次のコマンドでコンテナーレジストリのコンテナーを起動するだけです。なお、コンテナーイメージが事前に pull されていない場合は run の中で pull してくれます。

Container_Basic_Operation.bash.txt

```
# mkdir ~/RegistryVolume
# docker run -d -p 5000:5000 --name containers_reg -v ~/RegistryVolume:/var/lib/regist
ry registry
# docker ps -a
```

```
~ % mkdir ~/RegistryVolume
~ % sudo docker run -d -p 5000:5000 --name containers_reg -v ~/RegistryVolume:/var/lib/registry registry
72d3434a0dca2523e790c78daf09b628631b17d4ef9b738ab2364eb386919216
~ % sudo docker ps -a
CONTAINER ID   IMAGE                                          COMMAND                  CREATED          STATUS          PORTS                    NAMES
72d3434a0dca   registry                                       "/entrypoint.sh /etc…"   21 seconds ago   Up 6 seconds    0.0.0.0:5000->5000/tcp   containers_reg
858ab6706ac0   mcr.microsoft.com/mssql/server:2019-CU2-ubuntu-16.04   "/opt/mssql/bin/perm…"   2 hours ago      Up About an hour   0.0.0.0:11433->1433/tcp   sql19cu2
~ %
```

「docker push」コマンドにより、ローカルコンテナーレジストリに独自に作成した SQL Server コンテナーイメージをアップロードします。

Container_Basic_Operation.bash.txt

```
# docker tag sql19cu2/devdb:v1.0 localhost:5000/sql19cu2/devdb:v1.0
# docker push localhost:5000/sql19cu2/devdb:v1.0
```

```
~ % sudo docker tag sql19cu2/devdb:v1.0 localhost:5000/sql19cu2/devdb:v1.0
~ % sudo docker push localhost:5000/sql19cu2/devdb:v1.0
The push refers to repository [localhost:5000/sql19cu2/devdb]
3eb85ff182d4: Pushed
df9e3c1a2afe: Pushed
5d12250f4cfc: Pushed
92d30847d049: Pushed
7cd35e62d6fb: Pushed
2de391e51d73: Pushed
d73dd9e65295: Pushed
686245e78935: Pushed
d7ff1dc646ba: Pushed
644879075e24: Pushed
v1.0: digest: sha256:ae79e61d82be548366352fc7b9ad6119843989dcd022a80034ff9620b1b5616d size: 2411
~ %
```

開発者に見立てた別のクライアント環境から構築したローカルコンテナーレジストリにアクセスし、独自に作成した SQL Server コンテナーイメージを入手し起動します。

Container_Basic_Operation.bash.txt

```
# mkdir ~/datavol
# docker run -e ACCEPT_EULA=Y -e MSSQL_SA_PASSWORD=P@ssw0rd -u 0:0 -p 11433:1433 -v ~/
datavol:/var/opt/mssql --name sql19cu2 -d localhost:5000/sql19cu2/devdb:v1.0
# docker ps -a --filter name=sql19cu2
```

```
~ % mkdir ~/datavol
~ % sudo docker run -e ACCEPT_EULA=Y -e MSSQL_SA_PASSWORD=P@ssw0rd -u 0:0 -p 11433:1433 -v ~/datavol:/var/opt/mssql --name sql19cu2 -d localh
ost:5000/sql19cu2/devdb:v1.0
23597faafe519e3dc158e3e464c38a80c83babaac15bdfcfa336b1a9e699ac8a
~ % sudo docker ps -a --filter name=sql19cu2
CONTAINER ID   IMAGE                                  COMMAND                  CREATED          STATUS          PORTS                    NAMES
23597faafe51   localhost:5000/sql19cu2/devdb:v1.0     "/opt/mssql/bin/perm…"   11 seconds ago   Up 6 seconds    0.0.0.0:11433->1433/tcp   sql19cu2
~ %
```

起動した独自 SQL Server コンテナーに内包されたバックアップファイルをリストアすることで、この開発者専用のデータベースを準備します。

```
Container_Basic_Operation.bash.txt
# docker exec sql19cu2 ls -l /tmp
# sqlcmd -S localhost,11433 -U sa -P P@ssw0rd -Q 'RESTORE DATABASE [DevDB] FROM DISK =
 "/tmp/DevDB.bak"'
# sqlcmd -S localhost,11433 -U sa -P P@ssw0rd -Q 'use DevDB; select * from tab01'
```

バックアップした当時のデータが復元されているのが確認できます。

SQL Server コンテナーへの CU 適用とロールバック

今度は、SQL Server に最新の Cumulative Update（CU）適用してみます。手順は前のアップグレードと同様です。このプロセスが従来と比較していかに容易で迅速であるかを確認してみましょう。

まず、SQL Server 2019 CU2 コンテナーイメージを入手します。

```
Container_Basic_Operation.bash.txt
# docker pull mcr.microsoft.com/mssql/server:2019-CU2-ubuntu-16.04
# docker image list
```

次に現在稼働している SQL Server 2019 GA を停止します。

```
Container_Basic_Operation.bash.txt
```

```
# docker stop sql19ga
# docker ps -a --filter name=sql19ga
```

```
~ % sudo docker stop sql19ga
sql19ga
~ % sudo docker ps -a --filter name=sql19ga
CONTAINER ID        IMAGE                                          COMMAND                  CREATED             STATUS                      PORTS               NAMES
ea2a6409fbf4        mcr.microsoft.com/mssql/server:2019-GA-ubuntu-16.04   "/opt/mssql/bin/perm…"   31 minutes ago      Exited (0) 6 seconds ago                        sql19ga
~ %
```

新たに SQL Server 2019 CU2 コンテナーを作成し起動します。

Container_Basic_Operation.bash.txt

```
# docker run -e ACCEPT_EULA=Y -e MSSQL_SA_PASSWORD=P@ssw0rd -u 0:0 -p 11433:1433 -v ~/DataVolume:/var/opt/mssql --name sql19cu2 -d mcr.microsoft.com/mssql/server:2019-CU2-ubuntu-16.04
```

```
~ % sudo docker run -e ACCEPT_EULA=Y -e MSSQL_SA_PASSWORD=P@ssw0rd -u 0:0 -p 11433:1433 -v ~/DataVolume:/var/opt/mssql --name sql19ga -d mcr.microsoft.com/mssql/server:
2019-GA-ubuntu-16.04
ea2a6409fbf4c1d0220a814df3e958158ddedcc90712f0bdeab91b6d4e979e10
~ % sudo docker ps -a
CONTAINER ID        IMAGE                                          COMMAND                  CREATED             STATUS              PORTS                    NAMES
ea2a6409fbf4        mcr.microsoft.com/mssql/server:2019-GA-ubuntu-16.04     "/opt/mssql/bin/perm…"   11 seconds ago      Up 4 seconds        0.0.0.0:11433->1433/tcp  sql19ga
39b6d2ccdebf        mcr.microsoft.com/mssql/server:2017-CU19-ubuntu-16.04   "/opt/mssql/bin/nonr…"   21 hours ago        Exited (0) 9 minutes ago                     sql17cu19
~ %
```

アップグレード同様に初回起動時にアップグレードプロセスが走ります。アップグレードプロセスが完了するのをしばらく待った後にあらためて接続し、SQL Server 2017 で作成したデータベースにアクセスしてみます。

Container_Basic_Operation.bash.txt

```
# sqlcmd -S localhost,11433 -U sa -P P@ssw0rd -Q 'select @@version'
```

```
~ % sqlcmd -S localhost,11433 -U sa -P P@ssw0rd -Q 'select @@version'

-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------
Microsoft SQL Server 2019 (RTM-CU2) (KB4536075) - 15.0.4013.40 (X64)
        Feb  3 2020 16:40:57
        Copyright (C) 2019 Microsoft Corporation
        Developer Edition (64-bit) on Linux (Ubuntu 16.04.6 LTS) <X64>


(1 rows affected)
~ %
```

CU2 が適用された SQL Server 2019 コンテナーに差し替わっていることが確認できます。

もうひとつ追加の検証をします。この CU 適用によりアプリケーションの動作になんからの不都合が生じたためロールバックしなければならないケースに直面したとします。
その場合にやるべきことは SQL Server 2019 CU2 コンテナーを停止し、SQL Server 2019 GA コンテナーにロールバックすることです。

Container_Basic_Operation.bash.txt

```
# docker stop sql19cu2
# docker start sql19ga
```

```
~ % sudo docker stop sql19cu2
sql19cu2
~ % sudo docker start sql19ga
sql19ga
~ % sqlcmd -S localhost,11433 -U sa -P P@ssw0rd -Q 'select @@version'

-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
Microsoft SQL Server 2019 (RTM) - 15.0.2000.5 (X64)
        Sep 24 2019 13:48:23
        Copyright (C) 2019 Microsoft Corporation
        Developer Edition (64-bit) on Linux (Ubuntu 16.04.6 LTS) <X64>


(1 rows affected)
~ %
```

従来のように既存の SQL Server に CU 適用するのではなく、CU 適用済みの新しい SQL Server コンテナーを作成して差し替えるだけで CU 適用は完了です。また、古い SQL Server コンテナーは残るため、もし何らかの不都合で元に戻す必要が出た場合も同じく差し替えるだけで済み、迅速というだけではなく安全性も高いと言えます。なお、これが成し得るのは CU が互いに互換性をもつことが保証されているためです。

SQL Server コンテナーの破棄（rm）とコンテナーイメージの破棄（rmi）

アプリケーションのテストによって問題がないことが確認できたら、不要なコンテナーは「rm」コマンドで破棄します。

Container_Basic_Operation.bash.txt

```
# docker rm sql19ga
```

```
~ % sudo docker ps -a
CONTAINER ID   IMAGE                                                  COMMAND                  CREATED             STATUS                      PORTS                    NAMES
858ab6706ac0   mcr.microsoft.com/mssql/server:2019-CU2-ubuntu-16.04   "/opt/mssql/bin/perm…"   28 minutes ago      Up About a minute           0.0.0.0:11433->1433/tcp  sql19cu2
ea2a6409fbf4   mcr.microsoft.com/mssql/server:2019-GA-ubuntu-16.04    "/opt/mssql/bin/perm…"   About an hour ago   Exited (0) 9 minutes ago                             sql19ga
39b6d2ccdebf   mcr.microsoft.com/mssql/server:2017-CU19-ubuntu-16.04  "/opt/mssql/bin/nonr…"   22 hours ago        Exited (255) 3 minutes ago                           sql17cu19
~ % sudo docker rm sql19ga
sql19ga
~ % sudo docker rm sql17cu19
sql17cu19
~ % sudo docker ps -a
CONTAINER ID   IMAGE                                                  COMMAND                  CREATED             STATUS              PORTS                    NAMES
858ab6706ac0   mcr.microsoft.com/mssql/server:2019-CU2-ubuntu-16.04   "/opt/mssql/bin/perm…"   29 minutes ago      Up 2 minutes        0.0.0.0:11433->1433/tcp  sql19cu2
~ %
```

また、不要なコンテナーイメージは「rmi」コマンドで破棄します。

Container_Basic_Operation.bash.txt

```
# docker rmi mcr.microsoft.com/mssql/server:2019-GA-ubuntu-16.04
```

```
~ % sudo docker image list
REPOSITORY                        TAG                      IMAGE ID            CREATED             SIZE
mcr.microsoft.com/mssql/server    2019-CU2-ubuntu-16.04    56655b462301        2 weeks ago         1.54GB
mcr.microsoft.com/mssql/server    2017-CU19-ubuntu-16.04   a8343d3ce21c        4 weeks ago         1.39GB
mcr.microsoft.com/mssql/server    2019-GA-ubuntu-16.04     76c7c66bff02        4 months ago        1.57GB
~ % sudo docker rmi mcr.microsoft.com/mssql/server:2019-GA-ubuntu-16.04
Untagged: mcr.microsoft.com/mssql/server:2019-GA-ubuntu-16.04
Untagged: mcr.microsoft.com/mssql/server@sha256:c8fa22553ce421b0482febcafa712b29cbb933b0d97a8671686797b31cf157a9
Deleted: sha256:76c7c66bff021dbe2ff1c974bce60f490bb7978db6f3363b76ae5a9ff6d31e21
Deleted: sha256:548240776c3a798babbdc6297b4e27c2948192e669310fbb9961e345c72f8b45
Deleted: sha256:24105cb694d7733b63b89ba7c6396f7051c5b36241d226caca62bad987cf4df1
Deleted: sha256:c97cb18e630f42dcc91db9014a8a5b41de0721962fb70e605cbec744e54bd29c
Deleted: sha256:df0ecbbc4bb6765aa4fe27ca096ee1c1c692e1637ab1e604d5f387ac67a3b27c
~ % sudo docker rmi mcr.microsoft.com/mssql/server:2017-CU19-ubuntu-16.04
Untagged: mcr.microsoft.com/mssql/server:2017-CU19-ubuntu-16.04
Untagged: mcr.microsoft.com/mssql/server@sha256:f551ff159479f402bc3a120d9bb509c78bbcb8d1fe401552e102f28bc654db0e
Deleted: sha256:a8343d3ce21c1a659f3d3148a6c90a05b08abce53101806867fab2754f84ac8c
Deleted: sha256:cd8e736b209e91d17b8a6c4204d6382004ee413f0f3803b45da8342917396cdb
Deleted: sha256:9bfa71b27e75e6e80cf0112d99bbd41836cd0c47061447adb12f937e223f7c7a
~ % sudo docker image list
REPOSITORY                        TAG                      IMAGE ID            CREATED             SIZE
mcr.microsoft.com/mssql/server    2019-CU2-ubuntu-16.04    56655b462301        2 weeks ago         1.54GB
~ %
```

➢ Non-root SQL Server Container

1. 概要

SQL Server 2017 のコンテナーイメージはルートコンテナーとして公開されていましたが、SQL Server 2019 コンテナーイメージは非ルートコンテナーとして公開されるようになります。

2. 利用シーン

コンテナーのブレイクアウトを防止する観点で、非ルートコンテナーでの実行がより望ましいとされます。例えば、Red Hat OpenShift はルートコンテナーの実行を既定で許可していません。SQL Server 2017 のコンテナーイメージはルートコンテナーとして公開されていましたが、SQL Server 2019 コンテナーイメージについては非ルートコンテナーとして公開されるようになります。なお、SQL Server 2017 については今後リリースされる新しい CU が適用された SQL Server 2017 を含めたすべてのリリースがルートコンテナーとして公開されるようです。

ここではルートコンテナーの脆弱性の具体例を示します。

「docker」コマンドの実行が許可された次のような非ルートユーザー「user01」がいるとします。

Non-root_SQL_Server_Container.bash.txt

```
root@host# id user01
root@host# cat /etc/sudoers | grep user01
```

例えば「user01」ユーザーによる「useradd」コマンドの実行は許可されずエラーとなります。

Non-root_SQL_Server_Container.bash.txt

```
root@host# su - user01
user01@host$ useradd user02
user01@host$ sudo useradd user02
```

```
root@ubuntu[redacted]:~# su - user01
No directory, logging in with HOME=/
user01@ubuntu[redacted]:/$ useradd user02
useradd: Permission denied.
useradd: cannot lock /etc/passwd; try again later.
user01@ubuntu[redacted]:/$ sudo useradd user02
[sudo] password for user01:
Sorry, user user01 is not allowed to execute '/usr/sbin/useradd user02' as
root on ubuntu[redacted].510lxoxyhvlu5poty4n5dltdpf.lx.internal.cloudapp.net.
user01@ubuntu[redacted]:/$
```

ただし、「user01」ユーザーは「docker」コマンドのみ sudo が許可されており、コンテナーを起動することはできます。ここでは SQL Server 2017 コンテナー（ルートコンテナー）を起動します。
ここでのポイントは「-v /etc:/etc」オプションです。コンテナー内の「/etc」にホストの「/etc」をマウントしています。

Non-root_SQL_Server_Container.bash.txt

```
user01@host$ sudo docker run -e ACCEPT_EULA=Y -e MSSQL_SA_PASSWORD=P@ssw0rd -p 1433:1433 -
v /etc:/etc --name sql17cu19 -d mcr.microsoft.com/mssql/server:2017-CU19-ubuntu-16.04
user01@host$ sudo docker ps -a --filter name=sql17cu19
```

```
user01@ubuntu[redacted]:/$ sudo docker run -e ACCEPT_EULA=Y -e MSSQL_SA_PASSWORD=P@ssw0rd -p 1433:1433
 -v /etc:/etc --name sql17cu19 -d mcr.microsoft.com/mssql/server:2017-CU19-ubuntu-16.04
e7e9805d04aa24f99f622a4dbf34e4d2382a096ba5a93783e51dd54c546ac12a
user01@ubuntu[redacted]:/$ sudo docker ps -a --filter name=sql17cu19
CONTAINER ID        IMAGE                                                   COMMAND
     CREATED             STATUS              PORTS                     NAMES
e7e9805d04aa        mcr.microsoft.com/mssql/server:2017-CU19-ubuntu-16.04   "/opt/mssql/bin/nonr
…"   18 seconds ago      Up 11 seconds       0.0.0.0:1433->1433/tcp    sql17cu19
user01@ubuntu[redacted]:/$
```

起動した SQL Server 2017 コンテナーの OS 環境に「user01」ユーザーから「docker exec」コマンドでターミナル接続します。接続したコンテナーOS 環境でのユーザーはルートユーザーになっていることに注目してください。

Non-root_SQL_Server_Container.bash.txt

```
user01@host$ sudo docker exec -it sql17cu19 /bin/bash
root@container # whoami
```

```
user01@ubuntu:/$ sudo docker exec -it sql17cu19 /bin/bash
root@fd6c31c5dbd3:/# whoami
root
root@fd6c31c5dbd3:/#
```

ルートユーザーとして接続されたコンテナーOS 環境で「user02」ユーザーを作成し、加えて「user02」ユーザーを「sudo」グループに追加したのちに、ターミナル接続をクローズしてください。

Non-root_SQL_Server_Container.bash.txt

```
root@container # useradd user02
root@container # passwd user02
root@container # adduser user02 sudo
root@container # id user02
root@container # exit
```

```
root@fd6c31c5dbd3:/# useradd user02
root@fd6c31c5dbd3:/# passwd user02
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
root@fd6c31c5dbd3:/# adduser user02 sudo
Adding user `user02' to group `sudo' ...
Adding user user02 to group sudo
Done.
root@fd6c31c5dbd3:/# id user02
uid=1000(user02) gid=1000(user02) groups=1000(user02),27(sudo)
root@fd6c31c5dbd3:/# exit
exit
user01@ubuntu:/$
```

ホストに戻った「user01」ユーザーから次のコマンドを実行してみてください。

Non-root_SQL_Server_Container.bash.txt

```
user01@host$ id user02
user01@host$ su - user02
```

コンテナー内で作成した「user02」ユーザーが、ホスト上にも作成されてしまっていることが確認できます。そのまま続けて「user02」ユーザーにログインします。

```
user01@ubuntu:/$ id user02
uid=10002(user02) gid=10002(user02) groups=10002(user02),27(sudo)
user01@ubuntu:/$ su - user02
Password:
No directory, logging in with HOME=/
$
```

前の手順で、「user02」ユーザーを「sudo」グループに追加していることを思い出してください。

Non-root_SQL_Server_Container.bash.txt

```
user01@host$ su - user02
user02@host$ sudo cat /etc/sudoers | grep %sudo
```

```
user01@ubuntu:/$ su - user02
Password:
No directory, logging in with HOME=/
$ sudo cat /etc/sudoers | grep %sudo
[sudo] password for user02:
%sudo   ALL=(ALL:ALL) ALL
$
```

すなわち、「docker」コマンドのみ「sudo」が許可されていた「user01」ユーザーが、すべてのコマンドで「sudo」が許可される「user02」ユーザーを作成できてしまっていることを示します。

Non-root_SQL_Server_Container.bash.txt

```
user01@host$ id user02
user01@host$ sudo cat /etc/sudoers | grep %sudo
user01@host$ id user01
user01@host$ sudo cat /etc/sudoers | grep user01
```

```
$ id user02
uid=10002(user02) gid=10002(user02) groups=10002(user02),27(sudo)
$ sudo cat /etc/sudoers | grep %sudo
%sudo   ALL=(ALL:ALL) ALL
$ id user01
uid=1001(user01) gid=1001(user01) groups=1001(user01)
$ sudo cat /etc/sudoers | grep user01
user01 ALL=(ALL) /usr/bin/docker
$
```

これにより、例えば手順冒頭の「user01」ユーザーでエラーとなった「useradd」コマンドは「user02」ユーザーを介して実行ができるようになってしまいます。

```
Non-root_SQL_Server_Container.bash.txt
user01@host$ sudo useradd user03
user01@host$ sudo - user02
user02@host$ sudo useradd user03
user02@host$ sudo passwd user03
user02@host$ exit
user01@host$ id user03
```

```
user01@ubuntu[redacted]:/$ sudo useradd user03
[sudo] password for user01:
Sorry, user user01 is not allowed to execute '/usr/sbin/useradd user03' as
root on ubuntu[redacted].5l0lxoxyhvlu5poty4n5dltdpf.lx.internal.cloudapp.net.
user01@ubuntu[redacted]:/$ su - user02
Password:
No directory, logging in with HOME=/
$ sudo useradd user03
$ sudo passwd user03
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
$ exit
user01@ubuntu[redacted]:/$ id user03
uid=10003(user03) gid=10003(user03) groups=10003(user03)
user01@ubuntu[redacted]:/$
```

3. やってみよう

SQL Server 2017 のルートコンテナーと比較し、SQL Server 2019 の非ルートコンテナーの振る舞いを確認しましょう。

```
Non-root_SQL_Server_Container.bash.txt
host# docker run -e ACCEPT_EULA=Y -e MSSQL_SA_PASSWORD=P@ssw0rd -p 1433:1433 -v ~/DataVolu
me:/var/opt/mssql --name sql19ga -d mcr.microsoft.com/mssql/server:2019-GA-ubuntu-16.04
host# docker ps -a --filter name=sql19ga
```

```
[redacted] ~ % sudo docker run -e ACCEPT_EULA=Y -e MSSQL_SA_PASSWORD=P@ssw0rd -p 1433:1433 -v ~/DataVolume:/var/opt/mssql
--name sql19ga -d mcr.microsoft.com/mssql/server:2019-GA-ubuntu-16.04
df2087778a703b6d6f1e7fa6d29828345ffa222dee20817e9f2ec9c3c73853fa
[redacted] ~ % sudo docker ps -a --filter name=sql19ga
CONTAINER ID        IMAGE                                          COMMAND                  CREATED             STATUS
                    PORTS               NAMES
df2087778a70        mcr.microsoft.com/mssql/server:2019-GA-ubuntu-16.04   "/opt/mssql/bin/perm…"   11 seconds ago      Exited (
1) 3 seconds ago                        sql19ga
[redacted] ~ % sudo docker logs sql19ga
SQL Server 2019 will run as non-root by default.
This container is running as user mssql.
Your master database file is owned by root.
To learn more visit https://go.microsoft.com/fwlink/?linkid=2099216.
sqlservr: Unable to open /var/opt/mssql/.system/instance_id: Permission denied (13)
/opt/mssql/bin/sqlservr: Unable to open /var/opt/mssql/.system//instance_id: Permission denied (13)

[redacted] ~ %
```

「docker ps」コマンドの結果の STATUS が「Exited」となっており、起動直後にコンテナーが停止していることが確認できます。「docker logs」コマンドで確認できるログによると「/var/opt/mssql」に対して「Permission denied」とあります。これは同じくログ内の「This container is running as user mssql.」

が示す通り、SQL Server コンテナーは「mssql」ユーザーで実行されており、そのユーザーに対してコンテナー内の「/var/opt/mssql」へのアクセス権が無いことが原因で、コンテナー起動がエラーになっていることがわかります。

この非ルートコンテナーを起動するには正しいアクセス権を設定する必要があります。
具体的には、コンテナー内の「/var/opt/mssql」にはホストの「~/DataVolume」をマウントしているので、ホストの「~/DataVolume」に対してアクセス権を設定する必要があります。設定方法は以下のいずれかです。

- root グループにデータディレクトリへのアクセス許可を設定する

Non-root_SQL_Server_Container.bash.txt

```
host# chgrp -R root ~/DataVolume
host# chmod -R g+rw ~/DataVolume
```

- 非ルートユーザーをファイルの所有者として設定
  UID の「10001」はコンテナー内に作成された「mssql」ユーザーの UID です。

Non-root_SQL_Server_Container.bash.txt

```
host# chown -R 10001:0 ~/DataVolume
```

上記の通り、「~/DataVolume」に対してアクセス権を設定して非ルートコンテナーを起動してみます。

Non-root_SQL_Server_Container.bash.txt

```
host# ls -la ~/DataVolume
host# chown -R 10001:0 ~/DataVolume
host# ls -la ~/DataVolume
host# docker run -e ACCEPT_EULA=Y -e MSSQL_SA_PASSWORD=P@ssw0rd -p 1433:1433 -v ~/DataVolu
me:/var/opt/mssql --name sql19ga -d mcr.microsoft.com/mssql/server:2019-GA-ubuntu-16.04
host# docker ps -a --filter name=sql19ga
```

```
~ % ls -la ~/DataVolume
total 12
drwxr-xr-x  3 root root 4096 Mar  5 07:16 .
drwxr-xr-x 22 o9o9 o9o9 4096 Mar  5 07:19 ..
drwxr-xr-x  5 root root 4096 Mar  2 13:55 .system
~ % sudo chown -R 10001:0 ~/DataVolume
~ % ls -la ~/DataVolume
total 12
drwxr-xr-x  3 10001 root 4096 Mar  5 07:16 .
drwxr-xr-x 22 o9o9  o9o9 4096 Mar  5 07:19 ..
drwxr-xr-x  5 10001 root 4096 Mar  2 13:55 .system
~ % sudo docker run -e ACCEPT_EULA=Y -e MSSQL_SA_PASSWORD=P@ssw0rd -p 1433:1433 -v
~/DataVolume:/var/opt/mssql --name sql19ga -d mcr.microsoft.com/mssql/server:2019-GA-ubuntu-16.04
b40bceddf99e13b3a3afe8669e3d95efe6e4cb9680ea9ecb38e07614f463bbf1
~ % sudo docker ps -a --filter name=sql19ga
CONTAINER ID        IMAGE                                                   COMMAND
CREATED             STATUS              PORTS                   NAMES
b40bceddf99e        mcr.microsoft.com/mssql/server:2019-GA-ubuntu-16.04     "/opt/mssql/bin/perm…"
About a minute ago  Up About a minute   0.0.0.0:1433->1433/tcp   sql19ga
```

「docker ps」コマンドの結果で STATUS が「Up」となっており正しく起動しているようです。
実際にアクセスしてみます。

Non-root_SQL_Server_Container.bash.txt

```
host# sqlcmd -S localhost,1433 -U sa -P P@ssw0rd -Q 'select @@version'
```

```
~ % sqlcmd -S localhost,1433 -U sa -P P@ssw0rd -Q 'select @@version'

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------
Microsoft SQL Server 2019 (RTM) - 15.0.2000.5 (X64)
        Sep 24 2019 13:48:23
        Copyright (C) 2019 Microsoft Corporation
        Developer Edition (64-bit) on Linux (Ubuntu 16.04.6 LTS) <X64>

(1 rows affected)
~ %
```

確かに、SQL Server 2019 コンテナーが起動していることが確認できます。
さらに、このコンテナーが非ルートコンテナーで実行されていることを確認します。

Non-root_SQL_Server_Container.bash.txt

```
host# docker exec -it sql19ga /bin/bash
container$ whoami
container$ id mssql
container$ ls -la /var/opt/mssql
```

```
~ % sudo docker exec -it sql19ga /bin/bash
mssql@b40bceddf99e:/$ whoami
mssql
mssql@b40bceddf99e:/$ id mssql
uid=10001(mssql) gid=0(root) groups=0(root)
mssql@b40bceddf99e:/$ ls -la /var/opt/mssql
total 24
drwxr-xr-x 6 mssql root 4096 Mar  5 07:20 .
drwxr-xr-x 1 root  root 4096 Oct  9 18:58 ..
drwxr-xr-x 5 mssql root 4096 Mar  2 13:55 .system
drwxr-xr-x 2 mssql root 4096 Mar  5 07:20 data
drwxr-xr-x 2 mssql root 4096 Mar  5 07:30 log
drwxr-xr-x 2 mssql root 4096 Mar  5 07:20 secrets
mssql@b40bceddf99e:/$
```

SQL Server 2017 のルートコンテナーでは「whoami」コマンドの結果が「root」ユーザーでしたが、SQL Server 2019 の非ルートコンテナーでは「mssql」ユーザーになっていることが確認できます。また、「id」コマンドの結果から「mssql」ユーザーの UID が「10001」であり root グループに属していること、「ls」コマンドの結果からデータディレクトリの所有者が「mssql」ユーザーとなっていることがそれぞれ確認できます。

もし、既に起動している SQL Server 2017 のルートコンテナーを、SQL Server 2019 の非ルートコンテナーにアップグレードする場合は、SQL Server 2017 のルートコンテナーが作成したデータファイルに対して、「mssql」ユーザーがアクセスできるよう権限を付与する必要があります。

こうして、非ルートコンテナーでの実行が確認できました。非ルートコンテナーはコンテナーのブレイクアウトを防止する観点でより望ましい実行手段です。

なお、SQL Server 2019 自習書「6. Platform choice」-「SQL Server コンテナーの基本操作」のパートで、SQL Server 2019 コンテナーを起動する際に指定した「-u 0:0」オプションにより、SQL Server 2019 コンテナーをルートコンテナーとして起動することも可能です。
また、これとは逆に SQL Server 2017 を非ルートコンテナーとして起動したい場合には、Dockerfile による独自の SQL Server コンテナーイメージを作成します。手順の詳細については次の Docs を確認ください。

◇ 非ルート SQL Server 2017 コンテナーを作成して実行する
https://docs.microsoft.com/ja-jp/sql/linux/sql-server-linux-configure-docker?view=sql-server-ver15#buildnonrootcontainer

7. Scalable big data solution

Big Data Clusters（BDC）はデータに基づいた意思決定や新たな価値創造といった、ますます需要が高まるシナリオに対して、ワンストップのデータ統合管理と分析のためのデータプラットフォームを提供する新機能です。

BDC は「データ仮想化」によってサイロ化された外部データソースを統合したり、BDC 内部の「データレイク」にビッグデータを蓄えたりすることができます。こうして統合されたビッグデータは「機械学習」処理にかけたり、分析用に切り出したデータセットとして「データマート」を構成したりすることができます。このようにしてデータから得られるインサイトは REST API を通じて BDC 内外へ公開することができます。

従来はこうしたコンポーネントを個々に用意しそれらを組み合わせてデータプラットフォームとして構築する必要がありましたが、BDC はこれを単一パッケージとして提供します。BDC によるデータプラットフォームの実体は Kubernetes 上に展開されており、SQL Server だけでなく Spark や HDFS といったビッグデータ処理に関連するコンポーネントも含まれこれらが協調するよう構成されています。

7.1. BDC の機能

➢ データ仮想化

BDC では Polybase を使用してデータを仮想化します。データ仮想化は、元のデータを移動することなく、外部のデータソースへ直接リアルタイムにクエリします。分析のリアルタイム性が得られるだけでなく、重複するストレージのコストやデータパイプライン構築維持のコストを圧縮します。Polybase は外部のデータソースとして、SQL Server、HDFS、Azure BLOB Storage、Azure SQL Data Warehouse、Azure SQL Database、Oracle、Teradata、MongoDB、ODBC 互換のデータソースをサポートしています。

➢ データレイク

BDC はスケーラブルな HDFS を提供し非構造化データや半構造化データを蓄えることができます。また HDFS の階層化機能により、外部の HDFS と互換性のあるファイルシステムをマウントすることができます。この機能により、アプリケーションは内外のデータレイクに対して透過的にアクセスすることができるようになります。

➢ データマート

BDC はスケーラブルなコンピューティングと、スケーラブルなストレージで構成されたデータマートを提供します。様々なデータソースから分析用に切り出したデータセットとして、データマートを構成することができます。データレイクからデータマートへデータ連携する際は、BDC に内蔵された Spark を利用することもできます。データマートのデータは分散配置され、それらデータに対するクエリは並列で効率的に処理されます。

➢ ビッグデータ処理と機械学習

BDC はデータレイクやデータマートに蓄えたデータに対するビッグデータ処理や、機械学習タスクの実行環境を提供します。例えば、BDC 内の Apache Knox エンドポイントを介した Spark ジョブサブミットや、Apache Livy による REST API での Spark ジョブサブミットが可能です。また Jupyter ノートブックによるインタラクティブなビッグデータ処理を行うことも可能です。

➢ アプリケーション展開

BDC は、ユーザーのアプリケーションを BDC 内のコンテナーとして展開、管理、実行する機能を提供します。現在サポートされるアプリケーションランタイムは R、Python、SSIS、MLeap です。これによって BDC は、End to End の機械学習プラットフォームにもなります。

➢ 運用管理

これまで紹介した機能は次に示す各エンドポイントを通じて利用します。

また BDC 内には運用管理のための機能があらかじめ備わっており、こうした機能は OSS を活用して構成されています。例えば、パフォーマンスデータは「collectd」で収集し、「InfluxDB」に蓄積、「Grafana」で可視化されており、ログは「fluentd」で収集し、「ElasticSearch」に蓄積、「Kibana」で可視化されています。

7.2. BDC のアーキテクチャ

➢ コントローラー

コントローラーは、BDC の展開と管理のためのコアロジックをホストします。

azdata によって展開されるコントローラーは、その後に他のコンポーネントの展開を引き継ぎます。
コントローラーの最も重要なコンポーネントの 1 つにコントローラーサービスがあり、これは BDC の操作のための REST API を公開しています。
コントローラーには他にも各サービスへの接続をサポートするプロキシ、ログ監視に使用される Kibana と Elasticsearch、メトリック監視に使用される Grafana と InfluxDB、および BDC のメタデータを格納する SQL Server などのポッドが含まれます。

```
~ % kubectl get pods -n sqlbigdata -lplane=control
NAME              READY   STATUS    RESTARTS   AGE
control-6fhcd     3/3     Running   0          26d
controldb-0       2/2     Running   0          26d
controlwd-fzvms   1/1     Running   0          26d
gateway-0         2/2     Running   0          26d
logsdb-0          1/1     Running   0          26d
logsui-56kv6      1/1     Running   0          26d
metricsdb-0       1/1     Running   0          26d
metricsdc-7px86   1/1     Running   0          26d
metricsdc-kj4cn   1/1     Running   0          26d
metricsdc-z6zsc   1/1     Running   0          26d
metricsui-dtd6r   1/1     Running   0          26d
mgmtproxy-w6h74   2/2     Running   0          26d
~ %
```

詳細は次の Docs を確認ください。

◇ SQL Server ビッグ データ クラスターのコントローラーとは
https://docs.microsoft.com/ja-jp/sql/big-data-cluster/concept-controller?view=sql-server-ver15

➢ コンピュートプール

コンピュートプールは 1 つ以上のポッドで構成されており、それぞれに SQL Server コンテナーが配置されます。BDC を介したすべての外部テーブルクエリはコンピュートプールを使用しており、コンピュートプールが Polybase スケールアウトグループとして機能します。

```
~ % kubectl get pods -n sqlbigdata | grep compute
compute-0-0        3/3     Running   0          26d
~ %
```

詳細は次の Docs を確認ください。

◇ SQL Server ビッグ データ クラスターのコンピューティング プールとは
https://docs.microsoft.com/ja-jp/sql/big-data-cluster/concept-compute-pool?view=sql-server-ver15

➢ SQL Server マスターインスタンス

マスターインスタンスは、ユーザーデータベースの配置など従来の SQL Server 機能を提供する以外にも、コンピュートプール内の SQL Server インスタンス間でクエリを分散処理するために使用するスケールアウトクエリエンジンや、BDC 内部の各種メタ情報を管理しています。また、Machine Learning Service が構成済みであるため、Java、R、Python のランタイムが SQL Server に統合されています。

```
~ % kubectl get pods -n sqlbigdata | grep master
master-0        3/3     Running   0          26d
~ %
```

詳細は次の Docs を確認ください。

◇ SQL Server ビッグ データ クラスターのマスター インスタンスとは
https://docs.microsoft.com/ja-jp/sql/big-data-cluster/concept-master-instance?view=sql-server-ver15

➢ データプール

データプールはデータマートとしての役割を担います。 1 つ以上のポッドで構成されており、それぞれに SQL Server コンテナーが配置されます。データプールの SQL Server へは直接アクセスすることなく、SQL Server マスターインスタンス上の外部テーブルを介しアクセスします。

SQL Server マスターインスタンス上に外部テーブルを作成すると、データプール上の各 SQL Server に同名のテーブルが作成されます。またテーブルへのデータ登録は、シャーディングによってデータプール上の各 SQL Server に分散配置され、コンピュートプールを介してスケーラブルなデータアクセスが行われま

す。なおデータプールは INSERT、SELECT、TRUNCATE のみがサポートされます。データプールをリフレッシュするには TRUNCATE 後にデータを再投入します。

```
~ % kubectl get pods -n sqlbigdata | grep data
data-0-0        3/3     Running   0          26d
data-0-1        3/3     Running   0          26d
~ %
```

詳細は次の Docs を確認ください。

◇ SQL Server ビッグ データ クラスターのデータ プールとは
https://docs.microsoft.com/ja-jp/sql/big-data-cluster/concept-data-pool?view=sql-server-ver15

➢ ストレージプール

ストレージプールはデータレイクとしての役割を担います。SQL Server、Spark、HDFS を含む 1 つ以上のポッドで構成されており、HDFS によるデータストレージや Spark を使用したデータ処理を提供します。

詳細は次の Docs を確認ください。

◇ 記憶域プールとは (SQL Server ビッグ データ クラスター)
https://docs.microsoft.com/ja-jp/sql/big-data-cluster/concept-storage-pool?view=sql-server-ver15

➢ アプリケーションプール

アプリケーションプールは、BDC 上でのアプリケーションの作成に基づいて展開されるポッドの集まりです。現在サポートされているアプリケーションランタイムは、R、Python、MLeap、SSIS の 4 種類です。BDC 上にアプリケーションを展開するとコントローラーが spec.yaml ファイル内で指定されたラインタイムを検査し、対応するランタイムハンドラーを呼び出します。このランタイムハンドラーによってアプリケーションの作成が行われ、Kubernetes ReplicaSet が作成されます。この ReplicaSet には、それぞれに展開するアプリケーションが含まれた 1 つ以上のポッドが含まれています。ポッドの数は spec.yaml ファイルで設定された replica パラメーターで定義され、ポッドの中には poolsize パラメーターセットで定義された数のプールを含めることができます。

詳細は次の Docs を確認ください。

◇ ビッグ データ クラスターへのアプリケーション展開とは
https://docs.microsoft.com/ja-jp/sql/big-data-cluster/concept-application-deployment?view=sql-server-ver15

7.3. やってみよう

ここでは BDC を作成することに加えていくつかのユースケースを試してみます。

- BDC の展開と管理
- データ仮想化（PolyBase）によるサイロ化されたデータの統合
- データレイク（Storage Pool）の展開と外部 HDFS との統合
- Jupyter と Spark MLlib による Python 機械学習
- スケールアウト可能なデータマート（Data Pool）の展開と PowerBI によるデータ可視化
- アプリケーションのデプロイと REST API によるアプリケーションの呼び出し

➢ BDC の展開と管理

まずは BDC を展開します。加えて BDC ダッシュボードに接続し、正常性監視、メトリック監視、ログ監視を見てみましょう。

BDC の展開には、次に示すいくつかの手段があります。

- 展開済みの Kubernetes に対して「azdata bdc create」コマンドによって BDC を展開する。
- 公開された展開スクリプトによって Kubernetes と BDC を一括で展開する（展開するクリプトが対応する Kubernetes は、Azure Kubernetes Service もしくは kubeadm による単一ノード Kubernetes です）。
- Azure Data Studio によって BDC を展開する。

今回は、Azure Kubernetes Service（AKS）への展開スクリプトによって BDC を展開します。なおこの展開スクリプトは、指定した Azure サブスクリプション上に AKS を始めとする様々なリソースを作成するため、検証後不要であれば削除してください。

✱ 事前準備

BDC の展開に先立ち、いくつかのツールが必要になります。次の Docs を確認して、ツールをインストールしてください。

◇ SQL Server 2019 のビッグ データ ツールをインストールする
https://docs.microsoft.com/ja-jp/sql/big-data-cluster/deploy-big-data-tools?view=sql-server-ver15

以降の手順では、次のツールがインストール済みであることを前提とします。

- curl
- python
- azdata
- kubectl
- Azure CLI
- Azure Data Studio（ADS）

✱ BDC の展開

1. 「az login」コマンドで、AKS デプロイ先とする Azure アカウントにログインします。

BDC の展開と管理.bash.txt

```
az login
```

2. 展開スクリプトをダウンロードします。

BDC の展開と管理.bash.txt

```
curl -o deploy-sql-big-data-aks.py "https://raw.githubusercontent.com/Microsoft/sql-server-samples/master/samples/features/sql-big-data-cluster/deployment/aks/deploy-sql-big-data-aks.py"
```

なお、BDC 展開後はスケール (レプリカ数) やストレージ設定を変更することができないため、こうしたパラメーターをカスタムすることも可能です。展開のカスタムについては、次の Docs を確認ください。

◇ Kubernetes 上に SQL Server ビッグ データ クラスターを展開する方法
…カスタムの構成
https://docs.microsoft.com/ja-jp/sql/big-data-cluster/deployment-guidance?view=sql-server-ver15#customconfig

3. 展開スクリプトを実行し、AKS と BDC を展開します。

BDC の展開と管理.bash.txt

```
python deploy-sql-big-data-aks.py
```

スクリプトを実行すると、次の項目に関する入力を求められます。

項目	説明	本手順での設定値
Provide your Azure subscription ID	使用する Azure Subscription ID を指定します。	使用する Azure Subscription ID を指定してください。
Provide Azure resource group name to be created	AKS をデプロイする際に新規作成するリソースグループ名を指定します。	sql19bdc
Provide Azure region	AKS をデプロイするリージョンを指定します。	westus（米国西部）
Provide VM size for the AKS cluster	AKS を構成する Azure VM のサイズとノード数を指定します。この Azure VM のスペックは次の条件を満たす必要があります ● すべてのノード合計で 8 個以上の vCPU ● VM ごとに 64GB のメモリ ● すべてのノード合計で 24 個以上のディスク接続がサポートされた VM サイズ	Standard_E8s_v3
Provide number of worker nodes for AKS cluster		3
Provide name of AKS	作成する AKS クラスター及び BDC の名前を	sqlbigdata

cluster and SQL big data cluster	指定します。	
Provide username to be used for Controller and SQL Server master accounts	BDC の管理者アカウント名を指定します。	admin
Provide password to be used for Controller user, Knox user (root) and SQL Server Master accounts	BDC の管理者アカウントのパスワードを指定します。	MySQLBigData2019

Big Data Cluster を AKS 上に展開する際の前提条件の詳細は次の Docs を確認ください。

◇ SQL Server ビッグ データ クラスターの展開のために Azure Kubernetes Service を構成する
…前提条件
https://docs.microsoft.com/ja-jp/sql/big-data-cluster/deploy-on-aks?view=sql-server-ver15#prerequisites

しばらく待ち、「Cluster deployed successfully.」のメッセージの後、BDC が提供するサービスへのエンドポイント一覧が表示されれば BDC の展開は正常終了です。

```
bin — -zsh — 144×40
Cluster deployment documentation can be viewed at:
https://aka.ms/bdc-deploy

NOTE: Cluster creation can take a significant amount of time depending on
configuration, network speed, and the number of nodes in the cluster.

Starting cluster deployment.
Waiting for cluster controller to start.
Waiting for cluster controller to start.
Waiting for cluster controller to start.
Waiting for cluster controller to start.
Waiting for cluster controller to start.
Waiting for cluster controller to start.
Waiting for cluster controller to start.
Waiting for cluster controller to start.
Cluster controller endpoint is available at [redacted]:30080.
Cluster control plane is ready.
Storage pool is ready.
Data pool is ready.
Compute pool is ready.
Master pool is ready.
Cluster deployed successfully.
Logged in successfully to `https://[redacted]:30080` in namespace `sql19bdc`. Setting active context to `sql19bdc`.

SQL Server big data cluster endpoints:
Description                                          Endpoint                                                Name               Protocol
---------------------------------------------------  ------------------------------------------------------  -----------------  ----------
Gateway to access HDFS files, Spark                  https://[redacted]:30443                                gateway            https
Spark Jobs Management and Monitoring Dashboard       https://[redacted]:30443/gateway/default/sparkhistory   spark-history      https
Spark Diagnostics and Monitoring Dashboard           https://[redacted]:30443/gateway/default/yarn           yarn-ui            https
Application Proxy                                    https://[redacted]:30778                                app-proxy          https
Management Proxy                                     https://[redacted]:30777                                mgmtproxy          https
Log Search Dashboard                                 https://[redacted]:30777/kibana                         logsui             https
Metrics Dashboard                                    https://[redacted]:30777/grafana                        metricsui          https
Cluster Management Service                           https://[redacted]:30080                                controller         https
SQL Server Master Instance Front-End                 [redacted],31433                                        sql-server-master  tds
HDFS File System Proxy                               https://[redacted]:30443/gateway/default/webhdfs/v1     webhdfs            https
Proxy for running Spark statements, jobs, applications https://[redacted]:30443/gateway/default/livy/v1      livy               https
[redacted]
%
```

kubectl get pod コマンドで Pod の一覧を取得すると次のようになります。

```
~ % kubectl get pod -n sql19bdc
NAME                READY   STATUS    RESTARTS   AGE
appproxy-9zdhm      2/2     Running   0          17m
compute-0-0         3/3     Running   0          17m
control-4mnhf       3/3     Running   0          29m
controldb-0         2/2     Running   0          29m
controlwd-h6lsx     1/1     Running   0          22m
data-0-0            3/3     Running   0          17m
data-0-1            3/3     Running   0          17m
gateway-0           2/2     Running   0          17m
logsdb-0            1/1     Running   0          22m
logsui-r26h2        1/1     Running   0          22m
master-0            3/3     Running   0          17m
metricsdb-0         1/1     Running   0          22m
metricsdc-4nkrp     1/1     Running   0          22m
metricsdc-h8zpz     1/1     Running   0          22m
metricsdc-xhdg9     1/1     Running   0          22m
metricsui-grpdt     1/1     Running   0          22m
mgmtproxy-lhpzx     2/2     Running   0          22m
nmnode-0-0          2/2     Running   0          17m
sparkhead-0         4/4     Running   0          16m
storage-0-0         4/4     Running   0          16m
storage-0-1         4/4     Running   0          16m
~ %
```

なお、状況によっては、次のようなエラーが発生する場合があります。

```
Operation failed with status: 'Bad Request'. Details: The credentials in ServicePrinci
palProfile were invalid. Please see https://aka.ms/aks-sp-help for more details. (Deta
ils: adal: Refresh request failed. Status Code = '400'. Response body: {"error":"unaut
horized_client","error_description":"AADSTS700016: Application with identifier 'XXXXXX
XX-XXXX-XXXX-XXXX-XXXXXXXXXXXX' was not found in the directory 'XXXXXXXX-XXXX-XXXX-XXX
X-XXXXXXXXXXXX'. This can happen if the application has not been installed by the admi
nistrator of the tenant or consented to by any user in the tenant. You may have sent y
our authentication request to the wrong tenant
```

これは、展開スクリプト内で作成している「Azure AD サービスプリンシパル」を後続の処理で読み取れないことを意味しています。原因は Azure AD サービスプリンシパル作成が非同期であるため、その処理の遅延によって後続処理がエラーになっていると考えられます。

このエラーが発生した場合は、初回の展開スクリプト実行で作成した Azure AD サービスプリンシパルを使用するように展開スクリプトを書き換えた後、再度スクリプトを実行します。以下に対応手順の例を示します。エラーとなっていない場合は以下の手順をスキップしてください。

① エラーメッセージ内の「Application with identifier」に表示されているアプリケーション ID をメモします。

```
Operation failed with status: 'Bad Request'. Details: The credentials in ServicePri
ncipalProfile were invalid. Please see https://aka.ms/aks-sp-help for more details.
 (Details: adal: Refresh request failed. Status Code = '400'. Response body: {"erro
r":"unauthorized_client","error_description":"AADSTS700016: Application with identi
fier 'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX' was not found in the directory 'XXXXXXX
X-XXXX-XXXX-XXXX-XXXXXXXXXXXX'. This can happen if the application has not been ins
talled by the administrator of the tenant or consented to by any user in the tenan
t. You may have sent your authentication request to the wrong tenant
```

② サービスプリンシパルのパスワード情報は作成時にしか確認できないため、一度資格情報をリセットします。コマンドプロンプトで以下のコマンドを実行します。
<Application ID>にはメモしたアプリケーション ID を張り付けます。

BDC の展開と管理.bash.txt

```
az ad sp credential reset --name <Application ID>
```

実行結果の中から「appId」と「password」に表示される値をメモします。

```
az ad sp credential reset --name XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
{
  "appId": "XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX",
  "name": "XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX",
  "password": "XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX",
  "tenant": "XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX"
}
```

③ 「deploy-sql-big-data-aks.py」をエディターで開き、「az aks create」コマンドを実行している部分を見つけます。

```
command = "az aks create --name "+CLUSTER_NAME+" --resource-group "+GROUP_NAME+" --
generate-ssh-keys --node-vm-size "+VM_SIZE+" --node-count "+AKS_NODE_COUNT
```

コマンドの末尾に以下のオプションを追加します。
<Application ID>および<Password>にはメモした「appId」と「password」を張り付けます。

```
+" --service-principal <Application ID> --client-secret <Password>"
```

「deploy-sql-big-data-aks.py」を上書き保存し、再度展開スクリプトを実行します。

BDC の展開と管理.bash.txt

```
python deploy-sql-big-data-aks.py
```

✱ BDC ダッシュボードへの接続と管理（正常性監視、メトリクス監視、ログ監視）

1. SQL Server マスターインスタンスへの接続エンドポイントを確認します。
   まず azdata コマンドで BDC にログインします。

BDC の展開と管理.bash.txt

```
azdata login
```

コマンドを実行すると「Namespace」「Username」「Password」の入力を求められます。次の情報を入力して Enter キーを押下します。

項目	入力値
Namespace	sqlbigdata
Username	admin
Password	MySQLBigData2019

以下のメッセージが表示されれば正常にログインできています。

2. 次に、SQL Server マスターインスタンスのエンドポイントを照会します。

BDC の展開と管理.bash.txt

```
azdata bdc endpoint list -e sql-server-master
```

返される情報の中から「endpoint」に設定された値をメモします。

3. 今回は ADS から BDC に接続します。

新しい接続画面で下記表の内容を入力して「Connect」をクリックします。

項目	入力値
Server	メモした「endpoint」の値
Authentication type	SQL Login

User name	admin
Password	MySQLBigData2019

接続が成功すると、以下のような画面が表示されます。

4. BDC ダッシュボードを介して正常性監視、メトリクス監視、ログ監視を行います。
   先の画面の「SQL Server Big Data Cluster」タブ -> 「Cluster Dashboard」タイルをクリックします。

この手順により、次のダッシュボードが表示され、各コンポーネントの正常性と各サービスへのエンドポイント一覧を確認することできます。

任意のコンポーネントをクリックすると、画面上部に該当コンポーネントを構成する Pod の一覧が表示されるので、その一覧から各 Pod の正常性を確認できます。画面下部には、該当 Pod のメトリクスやログを参照するためのリンクが表示されます。

メトリクスへの任意のリンクをクリックすると、認証が求められます。

項目	入力値
User name	admin
Password	MySQLBigData2019

上記を入力して認証を通過すると、次のような「Grafana」によるリアルタイムなメトリクス監視のダッシュボードが確認できます。

ログへの任意のリンクをクリックすると、同じく認証が求められます。

項目	入力値
User name	admin
Password	MySQLBigData2019

上記を入力して認証を通過すると、次のような「Kibana」によるログ監視のダッシュボードが確認できます。

このように、BDC には管理のための機能がクラスター内にあらかじめ備わっており、こうした機能は OSS を活用して構成されています。パフォーマンスデータは「collectd」で収集し、「InfluxDB」に蓄積、「Grafana」で可視化されており、ログは「fluentd」で収集し、「ElasticSearch」に蓄積、「Kibana」で可視化されています。

➢ データ仮想化（PolyBase）によるサイロ化されたデータの統合

ここでは、PolyBase によるデータ仮想化によってサイロ化されたデータを統合します。
具体的には「Oracle」「Azure Blob Storage」「SQL Server（Azure SQL Database）」「HDFS」をソースとする外部テーブルを作成し、その外部テーブルを介した統合データをクエリします。

✱ 事前準備

このハンズオンで使用するデータとデータソース、ならびにオブジェクト名は次の通りです。

データ	データソース	オブジェクト名
EC サイトを利用するユーザーのマスターデータ	Oracle	USERS テーブル
EC サイトで販売している商品のマスターデータ	Azure Blob Storage	Products.csv
ユーザーがある商品に対してつけた「評価点」（5 点満点）と「コメント」のデータ	Azure SQL Database	ProductReviews テーブル
各商品の月ごとの販売数データ	HDFS	ProductSales.csv

これらのオブジェクトには次の関係があるとします。

ここでは外部のデータソースに対して、ハンズオンで使用するデータを登録します。

Oracle（Oracle Database はセットアップ済であるものとします）

次のユーザーとテーブルを定義します。

データ仮想化によるサイロ化されたデータの統合.sql

```
CREATE USER ORA_USER
IDENTIFIED BY "OraPassw0rd"
TEMPORARY TABLESPACE temp;
GRANT DBA TO ORA_USER;
CREATE TABLE ORA_USER.USERS(
    USERID INT NOT NULL,
    GENDER CHAR(1) NOT NULL,
    AGE INT NOT NULL
);
INSERT INTO ORA_USER.USERS (USERID, GENDER, AGE)
VALUES (1, 'F', 19);
INSERT INTO ORA_USER.USERS (USERID, GENDER, AGE)
VALUES (2, 'M', 18);
INSERT INTO ORA_USER.USERS (USERID, GENDER, AGE)
VALUES (3, 'F', 40);
INSERT INTO ORA_USER.USERS (USERID, GENDER, AGE)
VALUES (4, 'M', 45);
COMMIT;
```

Azure Blob Storage

1. 次のストレージアカウントを作成し、該当ストレージアカウント内にコンテナーを作成します。

ストレージ アカウント名	任意
アカウントの種類	BlobStorage
コンテナー名	任意

▶ ストレージアカウント名は任意、アカウントの種類は「BlobStorage」を選択します。

▶ コンテナー名は任意です。

2. 作成したストレージアカウントのアクセスキーをメモします。

3. 作成したコンテナーに対して、サンプルスクリプト「7_Big Data Clusters」フォルダに含めている「Products.csv」をアップロードします。

SQL Server（SQL Server もしくは Azure SQL Database がセットアップ済であるものとします）

次のユーザーとテーブルを定義します。

データ仮想化によるサイロ化されたデータの統合.sql

```
CREATE USER SQLDB_USER WITH PASSWORD=N'SQLDBPassw0rd', DEFAULT_SCHEMA=dbo;
GO
ALTER ROLE db_owner ADD MEMBER SQLDB_USER;
GO
CREATE TABLE ProductReviews (
    ID INT NOT NULL,
    UserID INT NOT NULL,
    ProductID INT NOT NULL,
    Evaluation DECIMAL(3, 2) NOT NULL,
    Comment NVARCHAR(100) NULL
)
GO
INSERT ProductReviews (ID, UserID, ProductID, Evaluation, Comment) VALUES (1, 1, 1, CA
ST(2.00 AS Decimal(3, 2)), N'写真より小ぶりで量が少ない')
GO
INSERT ProductReviews (ID, UserID, ProductID, Evaluation, Comment) VALUES (2, 1, 2, CA
ST(4.00 AS Decimal(3, 2)), N'大容量で良い')
GO
INSERT ProductReviews (ID, UserID, ProductID, Evaluation, Comment) VALUES (3, 1, 3, CA
ST(1.00 AS Decimal(3, 2)), N'すぐに壊れた')
GO
INSERT ProductReviews (ID, UserID, ProductID, Evaluation, Comment) VALUES (4, 2, 1, CA
ST(5.00 AS Decimal(3, 2)), N'安いところがよい。')
GO
INSERT ProductReviews (ID, UserID, ProductID, Evaluation, Comment) VALUES (5, 2, 2, CA
ST(5.00 AS Decimal(3, 2)), N'セールで安くなっていた。')
GO
INSERT ProductReviews (ID, UserID, ProductID, Evaluation, Comment) VALUES (6, 2, 3, CA
ST(4.00 AS Decimal(3, 2)), N'セールで安くなっていた。')
GO
```

```
INSERT ProductReviews (ID, UserID, ProductID, Evaluation, Comment) VALUES (7, 3, 1, CA
ST(1.00 AS Decimal(3, 2)), N'酸味が強くあまりおいしくありませんでした。')
GO
INSERT ProductReviews (ID, UserID, ProductID, Evaluation, Comment) VALUES (8, 3, 2, CA
ST(5.00 AS Decimal(3, 2)), N'値段の割に容量が多く、省エネ機能も付いており満足です。')
GO
INSERT ProductReviews (ID, UserID, ProductID, Evaluation, Comment) VALUES (9, 3, 3, CA
ST(3.00 AS Decimal(3, 2)), N'値段相応の機能がついており、不満はありません。')
GO
INSERT ProductReviews (ID, UserID, ProductID, Evaluation, Comment) VALUES (10, 4, 1, C
AST(3.00 AS Decimal(3, 2)), N'(コメントなし)')
GO
INSERT ProductReviews (ID, UserID, ProductID, Evaluation, Comment) VALUES (11, 4, 2, C
AST(3.00 AS Decimal(3, 2)), N'(コメントなし)')
GO
INSERT ProductReviews (ID, UserID, ProductID, Evaluation, Comment) VALUES (12, 4, 3, C
AST(3.00 AS Decimal(3, 2)), N'(コメントなし)')
GO
```

HDFS（Hadoop クラスターがセットアップ済であるものとします）

Hadoop クラスターに対して、サンプルスクリプト「7_Big Data Clusters」フォルダに含めている「ProductSales.csv」をアップロードし、HDFS に配置します。

データ仮想化によるサイロ化されたデータの統合.bash.txt

```
hadoop fs -put /tmp/ProductSales.csv /ProductSales.csv
```

✱ 外部テーブルの作成

ここでは BDC 内の SQL Server マスターインスタンスへ外部テーブルを作成します。

SQL Server マスターインスタンスにデータベースを作成

1. SQL Server マスターインスタンスにハンズオンに使用するデータベースを作成します。

データ仮想化によるサイロ化されたデータの統合.sql

```
USE master;
GO
CREATE DATABASE Demo;
GO
```

2. 続けて SQL Server マスターインスタンスにマスターキーを作成します。

データ仮想化によるサイロ化されたデータの統合.sql

```
USE [Demo];
GO
CREATE MASTER KEY ENCRYPTION BY PASSWORD ='MasterKeyPassw0rd';
GO
```

このマスターキーは外部データソースへの接続の資格情報を保護するために使用します。

外部データソースに接続するための資格情報を作成

次のコマンドを実行します。

データ仮想化によるサイロ化されたデータの統合.sql

```
-- Oracle 資格情報
USE [Demo];
GO
CREATE DATABASE SCOPED CREDENTIAL Oracle_Credential
WITH IDENTITY = 'ORA_USER', Secret = 'OraPassw0rd';
GO

-- Azure Blob Storage 資格情報
USE [Demo];
GO
CREATE DATABASE SCOPED CREDENTIAL AzBlob_Credential
WITH IDENTITY = '<任意の ID>', Secret = '<ストレージアカウントのアクセスキーを指定>';
GO

-- SQL Server 資格情報
USE [Demo];
GO
CREATE DATABASE SCOPED CREDENTIAL SQLDB_Credential
WITH IDENTITY = 'SQLDB_USER', Secret = 'SQLDBPassw0rd';
GO

-- HDFS 資格情報
USE [Demo];
GO
CREATE DATABASE SCOPED CREDENTIAL HDFS_Credential
WITH IDENTITY = '<HDFS ユーザー>', Secret = '<HDFS ユーザーのパスワード>';
GO
```

外部データソースを作成

次のコマンドを実行します。

データ仮想化によるサイロ化されたデータの統合.sql

```
-- Oracle 外部データソース
USE [Demo];
GO
CREATE EXTERNAL DATA SOURCE Oracle_Source
WITH (
    LOCATION = 'oracle://<Oracle サーバー>:<ポート>',
    CREDENTIAL = Oracle_Credential
);
GO

-- Azure Blob Storage 外部データソース
USE [Demo];
GO
CREATE EXTERNAL DATA SOURCE AzBlob_Source
```

```
WITH (
    TYPE = HADOOP,
    LOCATION ='wasbs://<コンテナー名>@<ストレージアカウント名>.blob.core.windows.net',
    CREDENTIAL = AzBlob_Credential
);
GO

-- SQL Server 外部データソース
USE [Demo];
GO
CREATE EXTERNAL DATA SOURCE SQLDB_Source
WITH (
    LOCATION = 'sqlserver://<SQL Server サーバー>:<ポート>',
    CREDENTIAL = SQLDB_Credential
);
GO

-- HDFS 外部データソース
USE [Demo];
GO
CREATE EXTERNAL DATA SOURCE HDFS_Source
WITH (
    TYPE = HADOOP,
    LOCATION = 'hdfs://<Hadoop Namenode>:<ポート>',
    CREDENTIAL = HDFS_Credential
);
GO
```

外部テーブルを作成

次のコマンドを実行します。

データ仮想化によるサイロ化されたデータの統合.sql

```
-- Oracle 外部データソース
USE [Demo];
GO
CREATE EXTERNAL TABLE Ora_Users (
    USERID INT NOT NULL,
    GENDER  CHAR(1) COLLATE Latin1_General_100_BIN2_UTF8  NOT NULL,
    AGE INT NOT NULL
)
WITH (
    LOCATION = '[<データベース名>].[ORA_USER].[USERS]',
    DATA_SOURCE = Oracle_Source
);
GO

-- Azure Blob Storage 外部データソース
USE [Demo];
GO
CREATE EXTERNAL FILE FORMAT AzBlob_Source_Format
WITH (
    FORMAT_TYPE = DELIMITEDTEXT,
    FORMAT_OPTIONS (FIRST_ROW = 2,FIELD_TERMINATOR =',')
);
GO
```

```
CREATE EXTERNAL TABLE AzBlob_Products (
    ProductID INT NOT NULL,
    ProductName nvarchar(20) NOT NULL,
    ProductCategory nvarchar(20) NULL
)
WITH (
    LOCATION='/Products.csv',
    DATA_SOURCE = AzBlob_Source,
    FILE_FORMAT = AzBlob_Source_Format
);
GO

-- SQL Server 外部データソース
USE [Demo];
GO
CREATE EXTERNAL TABLE SQLDB_ProductReviews (
    ID INT NOT NULL,
    UserID INT NOT NULL,
    ProductID INT NOT NULL,
    Evaluation DECIMAL(3, 2) NOT NULL,
    Comment NVARCHAR(100) NULL
)
WITH (
    LOCATION='[<データベース名>].[dbo].[ProductReviews]',
    DATA_SOURCE=SQLDB_Source
);
GO

-- HDFS 外部データソース
USE [Demo];
GO
CREATE EXTERNAL FILE FORMAT HDFS_Source_Format
WITH (
    FORMAT_TYPE = DELIMITEDTEXT,
    FORMAT_OPTIONS (FIRST_ROW = 2,FIELD_TERMINATOR =',')
);
GO
CREATE EXTERNAL TABLE HDFS_ProductSales (
    ProductID INT NOT NULL,
    YearAndMonth varchar(7) NOT NULL,
    SalesVolume  INT NOT NULL
)
WITH (
    LOCATION='/ProductSales.csv',
    DATA_SOURCE = HDFS_Source,
    FILE_FORMAT = HDFS_Source_Format
);
GO
```

もし接続先が Oracle 12.2 以降の場合、外部テーブルの作成時に ORA-28040 エラーが発生する場合があります。その場合、Oracle 側で以下の対処を行ってみてください。なお、この問題は SQL Server 2019 CU2 で改善されています。

- ▶ sqlnet.ora ファイルに「SQLNET.ALLOWD_LOGON_VERSION_SERVEE=11」を追加する。
- ▶ SEC_CASE_SENSITIVE_LOGON 初期化パラメーターを「false」にする。

✱ 外部データソースへのクエリ

前の手順で作成した外部テーブルを介して外部データソースへクエリします。

外部データソースへのシンプルクエリ

次のコマンドを実行します。

データ仮想化によるサイロ化されたデータの統合.sql

```
USE [Demo];
GO

-- Oracle
SELECT * FROM Ora_Users;
GO

-- Azure Blob Storage
SELECT * FROM AzBlob_Products;
GO

-- SQL Server
SELECT * from SQLDB_ProductReviews;
GO

-- HDFS
SELECT * FROM HDFS_ProductSales;
GO
```

商品カテゴリごとの平均評価点を求めるクエリ（SQL Server と Azure Blob Storage の結合クエリ）

次のコマンドを実行します。

データ仮想化によるサイロ化されたデータの統合.sql

```
USE [Demo];
GO
SELECT
        products.ProductCategory,
        AVG(reviews.evaluation) AS AVG_Evaluation
FROM SQLDB_ProductReviews reviews
INNER JOIN AzBlob_Products products
        ON reviews.ProductID = products.ProductID
GROUP BY products.ProductCategory;
GO
```

女性ユーザーが各商品につけたコメント一覧を求めるクエリ（SQL Server と Azure Blob Storage と Oracle の結合クエリ）

次のコマンドを実行します。

データ仮想化によるサイロ化されたデータの統合.sql

```
USE [Demo];
GO
SELECT
```

```
        products.ProductName,
        users.USERID,
        users.GENDER,
        reviews.Comment
FROM SQLDB_ProductReviews reviews
INNER JOIN AzBlob_Products products
        ON reviews.ProductID = products.ProductID
INNER JOIN Ora_Users users
        ON reviews.UserID = users.USERID
WHERE users.GENDER = 'F';
GO
```

"20 歳以下の平均評価が 3 未満の商品"の"2019 年 1 月または 2019 年 2 月の販売数"を求めるクエリ（SQL Server と Azure Blob Storage と Oracle と HDFS の結合クエリ）

次のコマンドを実行します。

データ仮想化によるサイロ化されたデータの統合.sql

```
USE [Demo];
GO
SELECT rev.ProductName,sales.YearAndMonth,sales.SalesVolume,rev.AVG_Evaluation
FROM HDFS_ProductSales sales
INNER JOIN (
```

```
        SELECT products.ProductID,products.ProductName, AVG(reviews.evaluation) AS AVG
_Evaluation
        FROM SQLDB_ProductReviews reviews
        INNER JOIN Ora_Users users
            ON reviews.UserID = users.USERID
        INNER JOIN AzBlob_Products products
            ON reviews.ProductID = products.ProductID
        WHERE users.AGE <= 20
        GROUP BY products.ProductID,products.ProductName
    )rev
ON sales.ProductID = rev.ProductID
WHERE (sales.YearAndMonth = '2019/1' OR sales.YearAndMonth = '2019/2')
AND rev.AVG_Evaluation < 3.00;
GO
```

➢ データレイク（Storage Pool）の展開と外部 HDFS との統合

ここでは BDC 内部のデータレイクへのデータの登録と参照に加え、外部のデータレイクを BDC のデータレイクとしてマウントし統合します。
具体的には「Azure Data Lake Storage Gen2（ADLS）」と「Amazon S3」を BDC のデータレイクとしてマウントし、それをデータソースとする外部テーブルを作成した後、その外部テーブルを介した統合データをクエリします。

✱ 事前準備

このハンズオンで使用するデータとデータソース、ならびにオブジェクト名は次の通りです。

データ	データソース	オブジェクト名
人事部で管理する社員のマスターデータ	BDC 上の HDFS	Person.cs
各社員が所持している携帯電話の電話番号データ	Azure Data Lake Storage Gen2	Phone.csv
各社員のメールアドレスデータ	Amazon S3	EmailAddress.csv

これらのオブジェクトには次の関係があるとします。

ここでは外部のデータソースに対してハンズオンで使用するデータを登録します。

ADLS

1. 次のストレージアカウントを作成し、該当ストレージアカウント内にコンテナーを作成します。

ストレージ アカウント名	任意
アカウントの種類	StorageV2（汎用 v2）
Data Lake Storage Gen2	有効
コンテナー名	任意

▶ ストレージアカウント名は任意、アカウントの種類は「StorageV2（汎用 v2）」を選択します。

▶ Data Lake Storage Gen2 は「有効」を選択します。

▶ コンテナー名は任意です。

2. 作成したストレージアカウントのアクセスキーをメモします。

3. 作成したコンテナーに対して、サンプルスクリプト「7_Big Data Clusters」フォルダに含めている「Phone.csv」をアップロードします。

Amazon S3

1. Amazon S3 のバケットを作成します。（バケット名は任意です。）

2. 作成したバケットに対して、サンプルスクリプト「7_Big Data Clusters」フォルダに含めている「EmailAddress.csv」をアップロードします。

▶ ［アクセス許可を設定する］画面では、［アカウントの追加］ボタンから使用するアカウントに［読み込み］権限を付与します。

▶ アップロードの完了を確認します。

3. 使用するアカウントの発行済アクセスキーをメモします。アクセスキーを発行していない場合は、次の Docs の内容を参考にアクセスキーとシークレットを取得してください。

◇ IAM ユーザーのアクセスキーの管理
https://docs.aws.amazon.com/ja_jp/IAM/latest/UserGuide/id_credentials_access-keys.html

✱ データレイクへのデータの登録と参照

ここでは、この先のハンズオンに使うデータをデータレイクへ登録してみます。

BDC 内のデータレイクへデータ登録

1. HDFS へのアクセスに使用するエンドポイントを確認します。
BDC の HDFS へのアクセスには、Apache Hadoop に対するアクセスのインターフェイスを提供する Apache Knox を使用しており、Knox エンドポイントは BDC 上の「gateway-svc-external」という kubernetes サービスとして展開されています。
次のコマンドを実行して、「gateway-svc-external」サービスの外部 IP アドレスを取得します。

データレイクの展開と外部 HDFS との統合.bash.txt
```
kubectl get service gateway-svc-external -n sqlbigdata
```

「EXTERNAL-IP」が示す IP アドレスが Knox エンドポイントになります。

2. Knox エンドポイントに対して、curl コマンドで HDFS を操作します。

まずディレクトリ「HumanResources」を作成します。
なお、Knox の認証ユーザーは「root」固定です。

データレイクの展開と外部 HDFS との統合.bash.txt
```
curl -i -L -k -u root:MySQLBigData2019 -X PUT "https://<gateway-svc-external service external IP>:30443/gateway/default/webhdfs/v1/HumanResources?op=MKDIRS"
```

```
root@39b6d2ccdebf: / — -zsh — 105×19
~ % curl -i -L -k -u root:MySQLBigData2019 -X PUT "https://<IP Address>:30443/gateway/default/w
ebhdfs/v1/HumanResources?op=MKDIRS"
HTTP/1.1 200 OK
Date: Tue, 10 Mar 2020 11:28:34 GMT
Set-Cookie: KNOXSESSIONID=node013ua8vwl7hugxzu79jx1s19oq2910.node0;Path=/gateway/default;Secure;HttpOnly
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Set-Cookie: rememberMe=deleteMe; Path=/gateway/default; Max-Age=0; Expires=Mon, 09-Mar-2020 11:28:34 GMT
Date: Tue, 10 Mar 2020 11:28:34 GMT
Cache-Control: no-cache
Expires: Tue, 10 Mar 2020 11:28:34 GMT
Date: Tue, 10 Mar 2020 11:28:34 GMT
Pragma: no-cache
X-FRAME-OPTIONS: SAMEORIGIN
Content-Type: application/json;charset=utf-8
Transfer-Encoding: chunked
Server: Jetty(9.4.12.v20180830)

{"boolean":true}%
~ %
```

次に、サンプルスクリプト「7_Big Data Clusters」フォルダに含めている「Person.csv」ファイルを登録します。

データレイクの展開と外部 HDFS との統合.bash.txt

```
curl -i -L -k -u root:MySQLBigData2019 -X PUT "https://<gateway-svc-external servic
e external IP>:30443/gateway/default/webhdfs/v1/HumanResources/Person.csv?op=create
" -H "Content-Type: application/octet-stream" -T Person.csv
```

```
root@39b6d2ccdebf: / — -zsh — 105×36
~ % curl -i -L -k -u root:MySQLBigData2019 -X PUT "https://<IP Address>:30443/gateway/default/w
ebhdfs/v1/HumanResources/Person.csv?op=create" -H "Content-Type: application/octet-stream" -T Person.csv
HTTP/1.1 307 Temporary Redirect
Date: Tue, 10 Mar 2020 11:25:12 GMT
Set-Cookie: KNOXSESSIONID=node011cnekoxal1mx3h6lktku8w592904.node0;Path=/gateway/default;Secure;HttpOnly
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Set-Cookie: rememberMe=deleteMe; Path=/gateway/default; Max-Age=0; Expires=Mon, 09-Mar-2020 11:25:12 GMT
Date: Tue, 10 Mar 2020 11:25:12 GMT
Cache-Control: no-cache
Expires: Tue, 10 Mar 2020 11:25:12 GMT
Date: Tue, 10 Mar 2020 11:25:12 GMT
Pragma: no-cache
X-FRAME-OPTIONS: SAMEORIGIN
Location: https://[redacted]:30443/gateway/default/webhdfs/data/v1/webhdfs/v1/HumanResources/Person.csv?
_=AAAACAAAABAAAADQqoHcIwMCv8UGNYk2jgcEnKl7kMlkVS5nW6GKNIpCYh-wDc5CvKwLQTTrx0oWtpRgZ16cYUBpDBk0-mee1597OCG
mSGq9KKrgSbbQyncrsc_7r-4f56jyHQV66UnYAEhi1wFwmmPwoUIhSXCI8Azp4sKm3VrziOVYaKUiCgfEbA6PLlbjspfcfBtlW4ynQlRb
v6QP1TdGBW-MGy4wzsLHcWTbT0lZY2AIHkP-irJ8rvKE1qp3BBXX7Dhp1dZNgqjYfa29z_5S57vjGLBPjKos1Fa7XeLpq6980Po0RhLRd
K0adpaIAZzV6Q
Content-Type: application/octet-stream
Content-Length: 0
Connection: close
Server: Jetty(9.4.12.v20180830)

HTTP/1.1 100 Continue

HTTP/1.1 201 Created
Date: Tue, 10 Mar 2020 11:25:13 GMT
Set-Cookie: KNOXSESSIONID=node0a71io6nplwc61ij656hloh9nh2905.node0;Path=/gateway/default;Secure;HttpOnly
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Set-Cookie: rememberMe=deleteMe; Path=/gateway/default; Max-Age=0; Expires=Mon, 09-Mar-2020 11:25:13 GMT
Location: https://[redacted]:30443/gateway/default/webhdfs/v1/HumanResources/Person.csv
Access-Control-Allow-Origin: *
Connection: close
Server: Jetty(9.4.12.v20180830)

~ %
```

登録されたファイルを確認してみます。

データレイクの展開と外部 HDFS との統合.bash.txt

```
curl -k -u root:MySQLBigData2019 -X GET "https://<gateway-svc-external service external IP>:30443/gateway/default/webhdfs/v1/HumanResources/?op=liststatus" | python -mjson.tool
```

```
— root@39b6d2ccdebf: / — -zsh — 81×27
~ % curl -k -u root:MySQLBigData2019 -X GET "https://<IP Address>:30443
/gateway/default/webhdfs/v1/HumanResources/?op=liststatus" | python -mjson.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   292    0   292    0     0    140      0 --:--:--  0:00:02 --:--:--   140
{
    "FileStatuses": {
        "FileStatus": [
            {
                "accessTime": 1583839760089,
                "blockSize": 134217728,
                "childrenNum": 0,
                "fileId": 16924,
                "group": "supergroup",
                "length": 49525,
                "modificationTime": 1583839760838,
                "owner": "root",
                "pathSuffix": "Person.csv",
                "permission": "644",
                "replication": 2,
                "storagePolicy": 0,
                "type": "FILE"
            }
        ]
    }
}
~ %
```

3. ADS からデータを参照してみます。

   ADS で BDC の SQL Server マスターインスタンスに接続し、［Connections］画面を開き、SQL Server マスターインスタンス配下の［HDFS］-［HumanResources］を展開し［Person.csv］を右クリックし［Preview］をクリックします。

✱ 外部データレイクのマウントと統合クエリ

ここでは外部のデータレイクを BDC のデータレイクとしてマウントし、さらに外部テーブルを介して T-SQL からのクエリで統合してみます。

「controller-svc-external」サービスにログイン

1. 「controller-svc-external」サービスへのアクセスに使用するエンドポイントを確認します。

データレイクの展開と外部 HDFS との統合.bash.txt

```
kubectl get svc controller-svc-external -n sqlbigdata
```

2. 「azdata」コマンドで「controller-svc-external」サービスにログインします。

データレイクの展開と外部 HDFS との統合.bash.txt

```
azdata login -e https://<controller-svc-external service external IP>:30080/
```

ユーザー名とパスワードには次の値を入力します。

項目	入力値
Username	admin

Password	MySQLBigData2019

```
[redacted] — root@39b6d2ccdebf: / — -zsh — 67×6
[redacted] ~ % azdata login -e https://<IP Address>:30080/
Username: admin
Password:
Logged in successfully to `https://<IP Address>:30080/` in namespace
 `sqlbigdata`. Setting active context to `sqlbigdata`.
[redacted] ~ %
```

Azure Data Lake Storage Gen2 コンテナーのマウント

1. 資格情報を設定します。

   Azure Data Lake Storage Gen2 への接続の資格情報として、今回はアクセスキーを利用します。次のコマンドで「MOUNT_CREDENTIALS」環境変数に資格情報を設定します。

データレイクの展開と外部 HDFS との統合.bash.txt

```
MOUNT_CREDENTIALS=fs.azure.abfs.account.name=<storage-account-name>.dfs.core.windows.net,fs.azure.account.key.<storage-account-name>.dfs.core.windows.net=<storage-account-access-key>
```

   <storage-account-name>には Azure Data Lake Storage Gen2 のストレージアカウント名を、<storage-account-access-key>には事前準備でメモしたストレージアカウントのアクセスキーを設定してください。

2. Azure Data Lake Storage Gen2 コンテナーをマウントします。

データレイクの展開と外部 HDFS との統合.bash.txt

```
azdata bdc hdfs mount create --remote-uri abfs://<blob-container-name>@<storage-account-name>.dfs.core.windows.net/ --mount-path /mounts/ADLS
```

   <blob-container-name>にはコンテナー名を、<storage-account-name>にはストレージアカウント名を記載してください。「--mount-path」は BDC の HDFS 側のマウント先パスを指定します。今回は「/mounts/ADLS」ディレクトリにマウントしています。

```
[redacted] — root@39b6d2ccdebf: / — -zsh — 99×9
[redacted] ~ % MOUNT_CREDENTIALS=fs.azure.abfs.account.name=sql19azuredatalakegen2.dfs.core.windows
.net,fs.azure.account.key.sql19azuredatalakegen2.dfs.core.windows.net=[redacted]
[redacted] ~ % azdata bdc hdfs mount create --remote-uri abfs://adlsgen2@sql19azuredatalakegen2.dfs
.core.windows.net/ --mount-path /mounts/ADLS
Warning: Attempting to mount abfs://adlsgen2@sql19azuredatalakegen2.dfs.core.windows.net/ without u
sing any credentials.
Create mount for /mounts/ADLS submitted successfully. Check mount status for progress
[redacted] ~ %
```

コマンドを実行すると「Create mount for /mounts/ADLS submitted successfully.」のメッセージが表示されますが、処理は非同期に行われるためこの時点では正常に処理が行われたかは分かりません。次のコマンドでマウントが正常に行われたかを確認します。

データレイクの展開と外部 HDFS との統合.bash.txt

```
azdata bdc hdfs mount status
```

state が「Ready」となっていればマウントは正常に行われています。

3. ADS からデータを参照してみます。

Amazon S3 バケットのマウント

1. 資格情報を設定します。

   Azure Data Lake Storage Gen2 と同様に「MOUNT_CREDENTIALS」環境変数に資格情報を設定します。

データレイクの展開と外部 HDFS との統合.bash.txt

```
MOUNT_CREDENTIALS=fs.s3a.access.key=<AWS Access Key>,fs.s3a.secret.key=<AWS Secret>
```

   <AWS Access Key>には S3 に接続する際に使用するアクセスキー、<AWS Secret>にはシークレットを入力してください。

2. Amazon S3 バケットをマウントします。

データレイクの展開と外部 HDFS との統合.bash.txt

```
azdata bdc hdfs mount create --remote-uri s3a://<S3 Bucket Name>/ --mount-path /mounts/S3
```

   <S3 Bucket Name>には Amazon S3 のバケット名を指定してください。「--mount-path」は BDC の HDFS 側のマウント先パスを指定します。今回は「/mounts/S3」ディレクトリにマウントしています。

   コマンドを実行すると「Create mount for /mounts/ADLS submitted successfully.」のメッセージが表示されますが、処理は非同期に行われるためこの時点では正常に処理が行われたかは分かりません。次のコマンドでマウントが正常に行われたかを確認します。

データレイクの展開と外部 HDFS との統合.bash.txt

```
azdata bdc hdfs mount status
```

state が「Ready」となっていればマウントは正常に行われています。

3. ADS からデータを参照してみます。

外部テーブルを介したデータレイクへの統合クエリ

1. データレイクを外部データソースとして定義します。

データレイクの展開と外部 HDFS との統合.sql

```
USE [Demo];
GO
CREATE EXTERNAL DATA SOURCE SqlStoragePool
WITH (LOCATION = 'sqlhdfs://controller-svc/default');
GO
```

2. データレイク上のファイルのフォーマットを定義します。

データレイクの展開と外部 HDFS との統合.sql

```
USE [Demo];
GO
CREATE EXTERNAL FILE FORMAT My_CSV_Format
WITH (
```

```
    FORMAT_TYPE = DELIMITEDTEXT,
    FORMAT_OPTIONS (FIRST_ROW = 2,FIELD_TERMINATOR =','));
GO
```

3. 外部テーブルを定義します。

データレイクの展開と外部 HDFS との統合.sql

```
USE [Demo];
GO

-- BDC 内のデータレイク
CREATE EXTERNAL TABLE StoragePool_Person (
    BusinessEntityID INT,
    Title VARCHAR(5),
    FirstName VARCHAR(50),
    MiddleName VARCHAR(50),
    LastName VARCHAR(50),
    Suffix VARCHAR(5),
    ModifiedDate DATETIME
)
WITH (
        LOCATION='/HumanResources/Person.csv',
        DATA_SOURCE = SqlStoragePool,
        FILE_FORMAT = My_CSV_Format
);
GO

-- BDC にマウントした外部データレイク (Azure Data Lake Storage Gen2)
CREATE EXTERNAL TABLE StoragePool_ADLS_Phone (
    BusinessEntityID INT,
    PhoneNumber VARCHAR(20),
    ModifiedDate DATETIME
)
WITH (
        LOCATION='/mounts/ADLS/Phone.csv',
        DATA_SOURCE = SqlStoragePool,
        FILE_FORMAT = My_CSV_Format
);
GO

-- BDC にマウントした外部データレイク (Amazon S3)
CREATE EXTERNAL TABLE StoragePool_S3_EmailAddress (
    BusinessEntityID INT,
    EmailAddressID INT,
    EmailAddress VARCHAR(50),
    ModifiedDate DATETIME
)
WITH (
        LOCATION='/mounts/S3/EmailAddress.csv',
        DATA_SOURCE = SqlStoragePool,
        FILE_FORMAT = My_CSV_Format
);
GO
```

4. T-SQL で外部テーブルの統合クエリを介してデータレイクにアクセスします。

データレイクの展開と外部 HDFS との統合.sql

```
USE [Demo];
GO
SELECT
        person.BusinessEntityID,
        person.FirstName,
        person.MiddleName,
        person.Lastname,
        phone.PhoneNumber,
        email.EmailAddress
FROM StoragePool_Person person
INNER JOIN  StoragePool_ADLS_Phone phone
ON person.BusinessEntityID = phone.BusinessEntityID
INNER JOIN StoragePool_S3_EmailAddress email
ON person.BusinessEntityID = email.BusinessEntityID
ORDER BY person.BusinessEntityID;
GO
```

➢ Jupyter と Spark MLlib による Python 機械学習

ここでは Jupyter ノートブックを利用し、インタラクティブな Python 機械学習を行います。機械学習のための学習データは、BDC のデータレイク上に登録します。また、機械学習のライブラリには Spark MLlib を利用します。こういったコンポーネントはすべて BDC に内蔵されており、個別に準備する必要はありません。

✱ 事前準備

機械学習で作成する認識モデルの解説と、学習データの準備手順を解説します。

このハンズオンで作成する認識モデル

このハンズオンでは 0～9 の手書き数字画像を認識させるモデルを作成します。
学習データには MNIST(Mixed National Institute of Standards and Technology database)を利用します。MNIST とは 0～9 の手書き数字画像のデータセットで機械学習の分野で評価に広く用いられているサンプルデータです。

MNIST のデータサンプル

今回使用するデータは上図のような画像形式ではなく、libsvm 形式のテキストデータを利用します。

```
7 203:84 204:185 205:159 206:151 207:6
```

一番左の数字がラベル（この画像が表現する数字）を表し、「XXX：YYY」となっている箇所が「XXX ピクセル目の画素値が YYY」であることを表すデータです。

なお、以降の手順で「画像形式の学習データのダウンロードと libsvm 形式への変換」を解説していますが、libsvm 形式をインターネットから入手し、それを代替としても問題ありません。その場合は「画像形式の学習データのダウンロードと libsvm 形式への変換」を読み飛ばしてください。

画像形式の学習データのダウンロードと libsvm 形式への変換

1. 学習データをダウンロードします。

Jupyter と SparkMLlib による Python 機械学習.bash.txt

```
ubuntu# curl -o train-images-idx3-ubyte.gz http://yann.lecun.com/exdb/mnist/train-i
mages-idx3-ubyte.gz
ubuntu# curl -o train-labels-idx1-ubyte.gz "http://yann.lecun.com/exdb/mnist/train-
labels-idx1-ubyte.gz"
ubuntu# curl -o t10k-images-idx3-ubyte.gz "http://yann.lecun.com/exdb/mnist/t10k-im
ages-idx3-ubyte.gz"
ubuntu# curl -o t10k-labels-idx1-ubyte.gz "http://yann.lecun.com/exdb/mnist/t10k-la
bels-idx1-ubyte.gz"
```

2. ダウンロードした圧縮ファイルを解凍します。

Big_Data_Clusters.bash.txt

```
ubuntu# gzip -dc train-images-idx3-ubyte.gz >train-images-idx3-ubyte
ubuntu# gzip -dc train-labels-idx1-ubyte.gz >train-labels-idx1-ubyte
ubuntu# gzip -dc t10k-images-idx3-ubyte.gz >t10k-images-idx3-ubyte
ubuntu# gzip -dc t10k-labels-idx1-ubyte.gz >t10k-labels-idx1-ubyte
```

3. 画像ファイルをバイナリからテキスト形式に変換します。

Big_Data_Clusters.bash.txt

```
ubuntu# od -An -v -tu1 -j16 -w784 train-images-idx3-ubyte | sed 's/^ *//' | tr -s '
 ' >./train-images.txt
ubuntu# od -An -v -tu1 -j16 -w784 t10k-images-idx3-ubyte | sed 's/^ *//' | tr -s '
' >./test-images.txt
```

4. ラベルファイルをテキスト形式に変換します。

Big_Data_Clusters.bash.txt

```
ubuntu# od -An -v -tu1 -j8 -w1 train-labels-idx1-ubyte | tr -d ' ' >./train-labels.
txt
ubuntu# od -An -v -tu1 -j8 -w1 t10k-labels-idx1-ubyte | tr -d ' ' >./test-labels.tx
t
```

5. テキスト形式に変換した画像ファイルとラベルファイルを使用し LIBSVM 形式ファイルを作成します。

Big_Data_Clusters.bash.txt

```
ubuntu# cat train-images.txt |¥
  awk '{
    for (i = 1; i <= NF; i++) if ($i > 0) printf("%d:%d%s", i, $i, i < NF ? " " : "
");
    printf("¥n");
  }' |¥
```

```
  paste -d' ' train-labels.txt - > mnist_training_0.txt
ubuntu# cat test-images.txt |¥
  awk '{
    for (i = 1; i <= NF; i++) if ($i > 0) printf("%d:%d%s", i, $i, i < NF ? " " : "
");
    printf("¥n");
  }' |¥
  paste -d' ' test-labels.txt - > mnist_test_0.txt
```

6. 出力されたファイルの行末尾の半角スペースを取り除きます。

Big_Data_Clusters.bash.txt

```
ubuntu# sed "s/ $//" mnist_test_0.txt > mnist_test.txt
ubuntu# sed "s/ $//" mnist_training_0.txt > mnist_training.txt
```

最終的に「mnist_training.txt」を学習用データ、「mnist_test.txt」をモデル評価用データとして以降のハンズオンで使用します。

BDC のデータレイク上に学習データを登録

今回は ADS を利用して BDC のデータレイク上に学習データをアップロードします。

1. 学習データ用のディレクトリを作成します。
   ［HDFS］を右クリックし［New directory］を選択すると画面上部にテキストボックスがでてくるので「MNIST」と入力して Enter キーを押します。

2. 学習データを登録します。

［MNIST］を右クリックし［Upload files］を選択するとファイルブラウザが立ち上がるので、「mnist_test.txt」と「mnist_training.txt」の 2 つのファイルを選択肢して［Upload］をクリックします。

3. 学習データが登録できたか確認してみます。

✱ Jupyter と Spark MLlib による Python 機械学習

ADS から Jupyter ノートブックを実行し Spark MLlib による Python 機械学習を行います。

ノートブックを開く

1. ADS の接続一覧から SQL Server マスターインスタンスを右クリックして［Manage］をクリックし、開いた画面で［SQL Server Big Data Cluster］タブを選択、続けて［Open Notebook］タイルをクリックします。

2. 立ち上がるファイルブラウザから、サンプルスクリプト「7_Big Data Clusters」フォルダに含めている「MNIST_dtree.ipynb」ファイルを選択し開きます。

開いたノートブックで［Kernel］の表示が「Spark|Scala」、［Attach To］の表示が「SQL Server マスターインスタンス」となっていることを確認します。

手書き数字認識モデルの作成と評価

1. 1 つ目のセルの実行ボタンをクリックし、データレイク上の「mnist_training.txt」と「mnist_test.txt」のデータをそれぞれ「training」と「test」のデータフレームに取り込みます。

「We have 60000 training images and 10000 test images.」と表示されている通り、60,000 件の学習用データと 10,000 件の評価用データが準備されました。

2. 2 つ目のセルの実行ボタンをクリックし、Spark MLlib をインポートします。

3. 3 つ目のセルの実行ボタンをクリックし、ラベルの文字列カラムをラベルインデックスのカラムにエンコードを行う定義をします。

この処理は例えば「画像に映っている動物が犬、猫、鳥のいずれであるかを分類する」というような問題の場合に、「犬」「猫」「鳥」というラベルに対して「0」「1」「2」のように数値のインデックス用ラベルを割り当てる処理です。

4. ４つ目のセルの実行ボタンをクリックし、分類器として「決定木」を定義します。

5. この処理は学習に使用する分類器（Classifier）の定義をしています。Spark の機械学習ライブラリでは、分類器クラスによって様々な分類アルゴリズムの学習を行う機能が提供されています。その中から今回は分類アルゴリズムとして「決定木」を使用しています。

6. ５つ目のセルの実行ボタンをクリックし、ML パイプラインを定義します。

7. この処理は ML パイプラインを定義しています。Spark の機械学習には、データフレームに対する連続した変換操作を一つにまとめる ML パイプラインという仕組みがあります。ML パイプラインを使用することで、機械学習を行う一連の操作をワークフローとしてまとめることができます。こ

こでは、先ほど定義した 2 つの処理（インデックスラベルの割り当てと決定木分類器の作成）をパイプラインにセットしています。

8. 6 つ目のセルの実行ボタンをクリックし、手書き数字認識モデルを作成します。

この処理は先ほど作成した ML パイプラインに渡される「training」データフレームを学習することで決定木による手書き数字認識モデルを作成しています。

9. 8 つ目のセルの実行ボタンをクリックし、手書き数字認識モデルの予測を行います。

この処理は作成した手書き数字認識モデルを利用して「test」データフレーム内のテスト用データの予測を行っています。「IndexedLabel」はテスト用データの答えを示し、「prediction」は作成

した手書き数字認識モデルが予測した答えです。すなわち両方の数字が一致しているものは予測が正解、不一致のものは不正解を意味します。

10. ８つ目のセルの実行ボタンをクリックし、手書き数字認識モデルの評価を行います。

Evaluator は予測結果を評価するためのクラスです。今回は「test」データフレームの予測を行った「predictions」変数の結果に対して、「accuracy（正答率）」を算出しています。
結果画面の赤枠部分の通り今回の正答率は「0.8174」でした。

➢ スケールアウト可能なデータマート（Data Pool）の展開と PowerBI によるデータ可視化

ここではデータレイクに蓄積した Big Data をとある分析用に切り出したデータセットとしてデータマートに連携し Power BI によって可視化してみます。

✱ 事前準備

ハンズオンで使用するデータレイクへのデータ登録と、データレイクに対する外部テーブルを定義します。

データレイクへのデータ登録

1. ここでは ADS を使用した GUI 操作で解説します。まず［HDFS］を右クリックし、［New directory］をクリックし、画面上部のテキストボックスに「RetailAnalysis」と入力して Enter キーを押します。

2. ［RetailAnalysis］を右クリックし、［Upload files］をクリックします。ファイルブラウザが立ち上がるので、サンプルスクリプト「7_Big Data Clusters」フォルダに含めている「Sales.csv」

「Store.csv」「Time.csv」「District.csv」の４つのファイルを選択肢して［Upload］をクリックします。

外部テーブル定義

1. はじめにデータレイク上のファイルのフォーマットを定義します。

スケールアウト可能なデータマートの展開と PowerBI によるデータ可視化.sql

```
USE [Demo];
GO
CREATE EXTERNAL FILE FORMAT My_CSV_Format
WITH (
    FORMAT_TYPE = DELIMITEDTEXT,
    FORMAT_OPTIONS (FIRST_ROW = 2,FIELD_TERMINATOR =','));
GO
```

2. 次に、データレイク上のファイルに対する外部テーブルを作成します。

スケールアウト可能なデータマートの展開と PowerBI によるデータ可視化.sql

```
USE [Demo];
GO
CREATE EXTERNAL TABLE StoragePool_Sales (
    MonthID Int,
    ItemID Int,
    LocationID Int,
    Sum_GrossMarginAmount decimal(10,2),
    Sum_Regular_Sales_Dollars decimal(10,2),
    Sum_Markdown_Sales_Dollars decimal(10,2),
    ScenarioID Int,
    ReportingPeriodID Int,
    Sum_Regular_Sales_Units Int,
    Sum_Markdown_Sales_Units Int
)
WITH (
        LOCATION='/RetailAnalysis/Sales.csv',
        DATA_SOURCE = SqlStoragePool,
        FILE_FORMAT = My_CSV_Format
);
GO
CREATE EXTERNAL TABLE StoragePool_Store (
    LocationID Int,
    [City Name] varchar(50),
    Territory char(2),
    PostalCode Int,
    OpenDate Datetime,
    SellingAreaSize Int,
    DistrictName varchar(50),
    Name varchar(50),
    StoreNumberName varchar(50),
    StoreNumber Int,
    City varchar(50),
    Chain varchar(50),
    DM varchar(50),
    DM_Pic varchar(100),
    DistrictID Int,
    [Open Month No] Int,
```

```
    [Open Month] char(3),
    [Open Year] Int,
    [Store Type] varchar(50)
)
WITH (
        LOCATION='/RetailAnalysis/Store.csv',
        DATA_SOURCE = SqlStoragePool,
        FILE_FORMAT = My_CSV_Format
);
GO
CREATE EXTERNAL TABLE StoragePool_Time (
    ReportingPeriodID Int,
    Period Int,
    FiscalYear Int,
    FiscalMonth char(3),
    Month Date
)
WITH (
        LOCATION='/RetailAnalysis/Time.csv',
        DATA_SOURCE = SqlStoragePool,
        FILE_FORMAT = My_CSV_Format
);
GO
CREATE EXTERNAL TABLE StoragePool_District (
    DistrictID Int,
    District varchar(50),
    DM varchar(50),
    DM_Pic_fl varchar(100),
    DM_Pic varchar(100),
    BusinessUnitID Int,
    DMImage varchar(50)
)
WITH (
        LOCATION='/RetailAnalysis/District.csv',
        DATA_SOURCE = SqlStoragePool,
        FILE_FORMAT = My_CSV_Format
);
GO
```

3. 最後に、外部テーブルを介してデータレイクにアクセスできることを確認します。

スケールアウト可能なデータマートの展開と PowerBI によるデータ可視化.sql

```
USE [Demo];
GO
SELECT count(*) AS ROW_COUNT FROM StoragePool_Sales;
GO
SELECT count(*) AS ROW_COUNT FROM StoragePool_Store;
GO
SELECT count(*) AS ROW_COUNT FROM StoragePool_Time;
GO
SELECT count(*) AS ROW_COUNT FROM StoragePool_District;
GO
```

それぞれの外部テーブルは、次の件数になるはずです。

テーブル	レコード件数
StoragePool_Sales	923,371
StoragePool_Store	104
StoragePool_Time	734
StoragePool_District	9

✱ データマート（Data Pool）へのテーブル定義

ここではデータマートとなるテーブルを定義します。

外部データソースの定義

データマートへのテーブル定義の前にデータマートを外部データソースとして定義する必要があります。LOCATION は「'sqldatapool://controller-svc/default'」を指定します。

スケールアウト可能なデータマートの展開と PowerBI によるデータ可視化.sql

```
USE [Demo];
GO
CREATE EXTERNAL DATA SOURCE SqlDataPool
WITH (LOCATION = 'sqldatapool://controller-svc/default');
GO
```

データマートへのテーブル定義

SQL Server マスターインスタンスに対して次の外部テーブルを定義すると、データマートへのテーブル定義として伝播します。

スケールアウト可能なデータマートの展開と PowerBI によるデータ可視化.sql

```
USE [Demo];
GO
CREATE EXTERNAL TABLE DataPool_Sales (
    MonthID Int,
    ItemID Int,
    LocationID Int,
    Sum_GrossMarginAmount decimal(10,2),
    Sum_Regular_Sales_Dollars decimal(10,2),
    Sum_Markdown_Sales_Dollars decimal(10,2),
    ScenarioID Int,
    ReportingPeriodID Int,
    Sum_Regular_Sales_Units Int,
    Sum_Markdown_Sales_Units Int
)
WITH (
        DATA_SOURCE = SqlDataPool,
        DISTRIBUTION = ROUND_ROBIN
);
GO
CREATE EXTERNAL TABLE DataPool_Store (
```

```
    LocationID Int,
    [City Name] varchar(50),
    Territory char(2),
    PostalCode Int,
    OpenDate Datetime,
    SellingAreaSize Int,
    DistrictName varchar(50),
    Name varchar(50),
    StoreNumberName varchar(50),
    StoreNumber Int,
    City varchar(50),
    Chain varchar(50),
    DM varchar(50),
    DM_Pic varchar(100),
    DistrictID Int,
    [Open Month No] Int,
    [Open Month] char(3),
    [Open Year] Int,
    [Store Type] varchar(50)
)
WITH (
        DATA_SOURCE = SqlDataPool,
        DISTRIBUTION = ROUND_ROBIN
);
GO
CREATE EXTERNAL TABLE DataPool_Time (
    ReportingPeriodID Int,
    Period Int,
    FiscalYear Int,
    FiscalMonth char(3),
    Month Date
)
WITH (
        DATA_SOURCE = SqlDataPool,
        DISTRIBUTION = ROUND_ROBIN
);
GO
CREATE EXTERNAL TABLE DataPool_District (
    DistrictID Int,
    District varchar(50),
    DM varchar(50),
    DM_Pic_fl varchar(100),
    DM_Pic varchar(100),
    BusinessUnitID Int,
    DMImage varchar(50)
)
WITH (
        DATA_SOURCE = SqlDataPool,
        DISTRIBUTION = ROUND_ROBIN
);
GO
```

＊ データレイクからデータマートへのデータ連携

ここでは Big Data と見立てたデータレイク上のデータを、分析用に切り出したデータセットとして、データマートに連携します。

データレイクからデータマートへのデータ連携

データレイクとデータマート、それぞれに定義した外部テーブルを介してデータを連携します。なお、今回は簡易ハンズオンであるためデータレイクとデータマートがすべて１：１（単純な INSERT INTO データプール SELECT * FROM データレイク）で対応していますが、実運用では分析の目的に応じたデータの ETL 処理を行うことになります。

スケールアウト可能なデータマートの展開と PowerBI によるデータ可視化.sql

```
USE [Demo];
GO
INSERT INTO DataPool_Sales
SELECT * FROM StoragePool_Sales;
GO
INSERT INTO DataPool_Store
SELECT * FROM StoragePool_Store;
GO
INSERT INTO DataPool_Time
SELECT * FROM StoragePool_Time;
GO
INSERT INTO DataPool_District
SELECT * FROM StoragePool_District;
GO
```

次のコマンドでデータマート上のそれぞれのテーブルにデータが登録されていることを確認します。

スケールアウト可能なデータマートの展開と PowerBI によるデータ可視化.sql

```
USE [Demo];
GO
SELECT count(*) AS ROW_COUNT FROM DataPool_Sales;
GO
SELECT count(*) AS ROW_COUNT FROM DataPool_Store;
GO
SELECT count(*) AS ROW_COUNT FROM DataPool_Time;
GO
SELECT count(*) AS ROW_COUNT FROM DataPool_District;
GO
```

+α：データマートを覗いてみよう

データマートは Data Pool というコンポーネントによってホストされており、Data Pool はスケールアウト可能な複数のポッドで構成されています。

例えば、今回 BDC の展開に利用した Azure Kubernetes Service（AKS）への展開スクリプトの場合は、既定では Data Pool を 2 つのポッドで構成しています。

通常の運用では、SQL Server なスターインスタンスからコンピュートプールを介して SQL Server データプールインスタンスにアクセスしますが、「AT DATA_SOURCE」オプションによって SQL Server データプールインスタンスそれぞれに対するクエリを実行することができます。
ここでは、これを利用してデータが複数のポッドにスケールアウトされているかを確認してみます。

次のコマンドはサーバーの名前を返す「@@SERVERNAME」を SQL Server データプールインスタンスそれぞれに実行します。

スケールアウト可能なデータマートの展開と PowerBI によるデータ可視化.sql

```
USE [Demo];
GO
EXECUTE('
    Use Demo;
    SELECT @@SERVERNAME AS SERVER,name AS TABLE_NAME
    FROM sys.tables WHERE name = ''DataPool_Sales'';'
) AT  DATA_SOURCE SqlDataPool;
GO
```

この結果からわかることは Data Pool は 2 つのポッドで構成されていることと、それぞれのポッド名は「data-0-0」と「data-0-1」である、さらに「DataPool_Sales」テーブルの実体がそれぞれのポッド（＝SQL Server データプールインスタンス）に作成されていることです。

BDC ダッシュボードからも、たしかに「data-0-0」と「data-0-1」の２つのポッドで構成されていることがわかります。

次に、テーブルへのデータアクセスは SQL Server マスターインスタンス上の外部テーブルを介して、透過的に分散処理されていることを確認します。

マスターインスタンスから参照できる 923,371 件のデータが、データプールの 2 つのポッドに 461,340 件と 462,031 行に分散して配置されていることが分かります。

✱ Power BI によるデータマートの可視化

あらかじめデザイン済みの Power BI レポート「PowerBI_Report.pbit」を介してデータマード上のデータを可視化します。

Power BI Desktop の準備

Power BI レポートは Power BI Desktop で開きます。Power BI Desktop が未インストールであれば、次のページからダウンロードしインストールを行ってください。

◇ Power BI Desktop―対話型レポート
https://powerbi.microsoft.com/ja-jp/desktop/

データマートへの接続

サンプルスクリプト「7_Big Data Clusters」フォルダに含めている「PowerBI_Report.pbit」ファイルをダブルクリックして起動します。
Power BI Desktop が起動すると、次の警告メッセージが表示されるため［OK］をクリックします。

続いて接続情報の入力画面が表示されるので ServerName と DatabaseName を入力し、▼をクリックし［編集］を選択します。

項目	入力値
ServerName	SQL Sevrer マスターインスタンスのエンドポイント
DatabaseName	Demo

続いて資格情報の入力画面が表示されるので、ユーザー名とパスワードを入力して［接続］をクリックします。

項目	入力値
ユーザー名	admin
パスワード	MySQLBigData2019

［接続］をクリックすると以下のような警告メッセージが表示されるため、［OK］をクリックします。

［OK］をクリックするとデータソースからのデータの読み込みが開始されます。また、Power BI Desktop 画面上部に「適用されていない保留中の変更がクエリにあります。」のメッセージが表示されるため、［変更の適用］をクリックします。

レポート参照

読み込みが完了すると以下のような画面が表示されます。

画面右の［フィールド］に表示されているように、このレポートは先ほど作成したデータプール上のテーブルのデータを可視化しています。

例えば、レポート右下のマップは、「DataPool_Store」テーブルの「PostalCode（郵便番号）」列と「DataPool_Sales」テーブルの「TotalSales」計算列を使用して「店舗ごとの売り上げ」を円の大きさで可視化しています。

➢ アプリケーションのデプロイと REST API によるアプリケーションの呼び出し

BDC ではアプリケーションを BDC 上のコンテナーとしてデプロイすることができます。これらのアプリケーションは Swagger 準拠の Web サービスとして公開され、BDC の内外で使用できます。現在 BDC にデプロイすることができるアプリケーションは R、Python、MLeap、SSIS の 4 種類です。ここでは BDC 上にアプリケーションをデプロイし REST API を通じてアプリケーションを呼び出す手順を解説します。

✱ 事前準備

本ハンズオンの REST API によるテストには「Postman」を利用しています。必要に応じて次のサイトからダウンロードしてインストールしてください。

◇ POSTMAN
https://www.postman.com/

なお、「Postman」の「SETTINGS」で［SSL certificate verification］の設定は［OFF］としています。

✱ アプリケーションのデプロイ

独自に作成したアプリケーションをデプロイします。

「controller-svc-external」サービスにログイン

1. 「controller-svc-external」サービスへのアクセスに使用するエンドポイントを確認します。

アプリケーションのデプロイと REST_API によるアプリケーションの呼び出し.bash.txt

```
kubectl get svc controller-svc-external -n sqlbigdata
```

```
~ % kubectl get svc controller-svc-external -n sqlbigdata
NAME                      TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)          AGE
controller-svc-external   LoadBalancer   <IP Address>   <IP Address>   30080:31426/TCP  22d
~ %
```

2. 「azdata」コマンドで「controller-svc-external」サービスにログインします。

アプリケーションのデプロイと REST_API によるアプリケーションの呼び出し.bash.txt

```
azdata login -e https://<controller-svc-external service external IP>:30080/
```

ユーザー名とパスワードには次の値を入力します。

項目	入力値
Username	admin
Password	MySQLBigData2019

```
~ % azdata login -e https://<IP Address>:30080/
Username: admin
Password:
Logged in successfully to `https://<IP Address>:30080/` in namespace
 `sqlbigdata`. Setting active context to `sqlbigdata`.
~ %
```

アプリケーションのデプロイ

1. 次のコマンドでアプリケーションをデプロイします。

   「--spec」パラメーターには YAML 仕様ファイル「spec.yaml」が格納されたディレクトリを指定します。今回は、サンプルスクリプト「7_Big Data Clusters」フォルダに含めている「spec.yaml」を使用するよう指定します。

アプリケーションのデプロイと REST_API によるアプリケーションの呼び出し.bash.txt

```
azdata app create --spec .
```

```
~/addpy % ls
add.py          spec.yaml
~/addpy % azdata app create --spec .
Application `add-app/v1` successfully created.
~/addpy %
```

なお、今回使用する YAML 仕様ファイル「spec.yaml」は以下のようになっています。

```
#spec.yaml
name: add-app #name of your python script
version: v1  #version of the app
runtime: Python #the language this app uses (R or Python)
src: ./add.py #full path to the location of the app
entrypoint: add #the function that will be called upon execution
replicas: 1  #number of replicas needed
poolsize: 1  #the pool size that you need your app to scale
inputs:  #input parameters that the app expects and the type
   x: int
   y: int
output: #output parameter the app expects and the type
   result: int
```

アプリケーションをデプロイするために、コントローラーに認識させる必要のある情報が記載されています。例えば、「runtime」でアプリケーションの種類を指定していたり、「src」でデプロイするアプリケーションのソース「add.py」を指定したり、「inputs」「output」でアプリケーションの入出力を定義したりしています。

デプロイしたアプリケーションである、Python アプリケーションのソースコード「add.py」（サンプルスクリプト「7_Big Data Clusters」フォルダに含む）は、以下のような 2 つの数値を受け取り、その合計を返すシンプルなアプリケーションになっています。

```
#add.py
def add(x, y):
  result = x+y
  return result
result=add(x,y)
```

2. アプリケーションデプロイの状態を確認します。
   アプリケーションのデプロイは非同期に処理されるため、現在の状態を次のコマンドで確認します。

アプリケーションのデプロイと REST_API によるアプリケーションの呼び出し.bash.txt

```
azdata app list
```

デプロイを開始した直後は State が「WaitingForCreate」となっています。その場合はしばらく待ってから再度確認してください。State が「Ready」に遷移するとデプロイしたアプリケーションが使用可能になります。

```
addpy — root@39b6d2ccdebf: / — -zsh — 94×11
~/addpy % azdata app create --spec .
Application `add-app/v1` successfully created.
~/addpy % azdata app list
App                                          Name     State            Version
-------------------------------------------- -------- ---------------- ---------
https://          :30080/api/v1/app/add-app/v1  add-app  WaitingForCreate v1
~/addpy % azdata app list
App                                          Name     State    Version
-------------------------------------------- -------- -------- ---------
https://          :30080/api/v1/app/add-app/v1  add-app  Ready    v1
~/addpy %
```

3. アプリケーションの動作を確認します。

   azdata コマンドを使用してアプリケーションを実行します。

アプリケーションのデプロイと REST_API によるアプリケーションの呼び出し.bash.txt

```
azdata app run --name add-app --version v1 --inputs x=1,y=2
```

```
addpy — root@39b6d2ccdebf: / — -zsh — 83×12
~/addpy % azdata app run --name add-app --version v1 --inputs x=1,y=2
{
  "changedFiles": [],
  "consoleOutput": "",
  "errorMessage": "",
  "outputFiles": {},
  "outputParameters": {
    "result": 3
  },
  "success": true
}
~/addpy %
```

Input に x = 1 、 y = 2 を受け取り、それらの足し算の結果を result = 3 として返却しており正常に実行できていることが分かります。

✱ REST API によるアプリケーションの呼び出し

BDC にデプロイしたアプリケーションは、REST API を使用して外部のアプリケーションから実行することができます。ここでは Postman を使用して REST API を呼び出してみます。

アクセストークンの入手

1. アクセストークンの入手は「controller-svc-external」サービスを利用するため、まず「controller-svc-external」サービスのエンドポイントを確認します。

アプリケーションのデプロイと REST_API によるアプリケーションの呼び出し.bash.txt

```
kubectl get svc controller-svc-external -n sqlbigdata
```

```
o9o9 — root@39b6d2ccdebf: / — -zsh — 92×5
~ % kubectl get svc controller-svc-external -n sqlbigdata
NAME                     TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)          AGE
controller-svc-external  LoadBalancer   <IP Address>   <IP Address>   30080:31426/TCP  22d
~ %
```

2. Postman を起動し、次のパラメーターで POST リクエストを「controller-svc-external」サービスのエンドポイントへ送信します。

HTTP メソッド	POST
URL	https://<controller-svc-external service external IP>:30080/api/v1/token
Authorization TYPE	Basic Auth
Username	admin
Password	MySQLBigData2019

3. レスポンスの Body の中の［access_token］をメモします。

REST API によるアプリケーションの呼び出し

1. REST API の発行先となる URL を取得します。

アプリケーションのデプロイと REST_API によるアプリケーションの呼び出し.bash.txt

```
azdata bdc endpoint list -e app-proxy
```

Application Proxy の endpoint の値をメモします。

2. Postman を起動し、次のパラメーターで POST リクエストを「controller-svc-external」サービスのエンドポイントへ送信します。

HTTP メソッド	POST
URL	<application proxy endpoint>/api/app/add-app/v1/run 構文は次の通りです。 <application proxy URL>/api/app/<appname>/<version>/run
Authorization TYPE	Bearer Token
Token	前の手順でメモしたアクセストークン
Body タイプ	raw / JSON

Body 値	{
	"x":4,
	"y":40
	}

3. レスポンスを確認します。

Input に x ＝4、 y ＝40 を受け取り、それらの足し算の結果を result＝44 として返却しており正常に実行できていることが分かります。

あとがき

本書は私が信頼するチームのメンバーの三浦 和、平岡 迪子、菱田 敏宏、安田 充、奥 良太とともに執筆しました。執筆にあたり、わたしたちが一番に大事にしようと決めたことは現場の目線です。SQL Server 2019 が現場の目線で見たときにどのようにうつるのか、現場が抱えるどのような課題を解決するものなのか、普段わたしたちが現場のお役に立たせていただく中で得てきた経験と知見を結集しようと、そう決めて今日の完成に至りました。限られた時間の中で今のわたしたちのベストを尽くせたと考えています。日々現場で戦うエンジニアの皆様に対し、本書が少しでもお役にたてることがあればと願っています。

最後になりましたが、執筆機会を提供いただいた日本マイクロソフト株式会社の担当者様、検証機を提供いただいたインテル株式会社の担当者様、そして私たちの執筆活動を支えてくれていた関係者の皆様に深く感謝申し上げます。

2020 年 3 月 31 日

NEC ソリューションイノベータ株式会社

太田 智行

SQL Server 2019 自習書

2020 年 3 月

日本電気株式会社

NEC ソリューションイノベータ株式会社