# Database Deployment

The document captures the best practices for database deployments and upgrades based on our learnings from VSTS.

## Deployment Maturity levels

Each application might require a different level of maturity based on it’s SLA. Here are the 4 broad classes of upgrades. Advancing up the levels require the application to take on more complexity to ensure a better user experience.

|  |  |
| --- | --- |
| Upgrade Level | Description |
| Offline | This assumes the app can tolerate stopping traffic for the duration of the entire upgrade. |
| Partially Offline Upgrade | Order of data operations are done online. The app takes a short downtime to replace tables, stored procedures etc. |
| Online Upgrade (with possible errors on inflight commands) | Upgrade is online. In flight commands might get an error / exception. Will work on retry. |
| Online Upgrade | Fully online upgrade. |

Following are the different components required to successfully get to Online Upgrade.

1. Database versioning
2. Script authoring
3. Application reaction to database versioning
4. Application locking

The below table shows the intersection between maturity level and components. Keep in mind that the sophistication required for each component might vary as you approach online upgrade. More on that later.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Level | Database versioning | Script authoring | Application reaction to database versioning | Application locking |
| Offline | Y | Y | N | N |
| Partially Offline | Y | Y | N | N |
| Online Upgrade (with possible errors on inflight commands) | Y | Y | Y | N |
| Online Upgrade | Y | Y | Y | Y |

## Sample

Going through the levels will be best illustrated with an over simplified sample. Contoso owns a product catalog service, the product catalog schema in the initial version consists of 1 table:

CREATE TABLE tbl\_Product

(

[ProductId] INT IDENTITY(1, 1) NOT NULL,

[Name] NVARCHAR(100) NOT NULL,

[Price] DECIMAL(10, 2) NOT NULL,

CONSTRAINT PK\_tbl\_Product PRIMARY KEY CLUSTERED (ProductId)

)

There are 3 stored procedures to perform the **C**reate, **R**ead and **U**pdate functionality.

|  |  |  |
| --- | --- | --- |
| CREATE PROCEDURE prc\_CreateProduct  @name NVARCHAR(100),  @price DECIMAL(10, 2)  AS  SET XACT\_ABORT ON  SET NOCOUNT ON  INSERT tbl\_Product(Name, Price)  SELECT @name, @price  SELECT @@IDENTITY AS ProductId  RETURN 0 | CREATE PROCEDURE prc\_GetProduct  @productId INT  AS  SET NOCOUNT ON  SELECT ProductId,  Name,  Price  FROM tbl\_Product  WHERE ProductId = @productId  RETURN 0 | CREATE PROCEDURE prc\_UpdateProduct  @productId INT,  @name NVARCHAR(100),  @price DECIMAL(10, 2)  AS  SET NOCOUNT ON  SET XACT\_ABORT ON  UPDATE tbl\_Product  SET Name = @name,  Price = @price  WHERE ProductId = @productId  RETURN 0 |

Contoso wants to expand to international markets. In order to do that, it requires the following changes:

1. Add a second table to store 1:N pricing info.

CREATE TABLE tbl\_ProductPrice (

ProductId INT NOT NULL,

CountryCode VARCHAR(3) NOT NULL,

[Price] DECIMAL(10, 2) NOT NULL,

CONSTRAINT PK\_tbl\_ProductPrice PRIMARY KEY CLUSTERED (ProductId, CountryCode)

)

1. Drop the column for price from tbl\_Product

CREATE TABLE tbl\_Product

(

[ProductId] INT IDENTITY(1, 1) NOT NULL,

[Name] NVARCHAR(100) NOT NULL,

~~[Price] DECIMAL(10, 2) NOT NULL,~~

CONSTRAINT PK\_tbl\_Product PRIMARY KEY CLUSTERED (ProductId)

)

1. Add a table type called typ\_ProductPricing which is used as input to the stored procedures.

CREATE TYPE typ\_ProductPrice AS TABLE (

CountryCode VARCHAR(3) PRIMARY KEY CLUSTERED,

Price DECIMAL (10, 2)

)

The stored procedures are modified as follows:

|  |  |  |
| --- | --- | --- |
| CREATE PROCEDURE prc\_CreateProduct  @name NVARCHAR(100),  @prices typ\_ProductPricing READONLY  AS  SET XACT\_ABORT ON  SET NOCOUNT ON  DECLARE @productId INT  BEGIN TRAN  INSERT tbl\_Product(Name)  SELECT @name  SELECT @productId = @@IDENTITY  INSERT tbl\_ProductPrice(ProductId, CountryCode, Price)  SELECT @productId,  CountryCode,  Price  FROM @prices  COMMIT TRAN  SELECT @productId AS ProductId  RETURN 0 | CREATE PROCEDURE prc\_GetProduct  @productId INT  AS  SET NOCOUNT ON  SELECT ProductId,  Name  FROM tbl\_Product  WHERE ProductId = @productId  SELECT CountryCode,  Price  FROM tbl\_ProductPrice  WHERE ProductId = @productId  RETURN 0 | CREATE PROCEDURE prc\_UpdateProduct  @productId INT,  @name NVARCHAR(100),  @prices typ\_ProductPricing READONLY  AS  SET NOCOUNT ON  SET XACT\_ABORT ON  BEGIN TRAN  UPDATE tbl\_Product  SET Name = @name  WHERE ProductId = @productId  MERGE tbl\_ProductPrice pp  USING @prices i  ON pp.CountryCode = i.CountryCode  AND pp.ProductId = @productId  WHEN MATCHED THEN  UPDATE SET Price = i.Price  WHEN NOT MATCHED BY TARGET THEN  INSERT (ProductId, CountryCode, Price)  VALUES (@productId, i.CountryCode, i.Price)  WHEN NOT MATCHED BY SOURCE THEN  DELETE;  COMMIT TRAN  RETURN 0 |

## Offline upgrade

Database Versioning

As a database application grows through many releases, it is important that we keep record of what version the database is. The version information should be recorded in the database itself – as an extended attribute or using a table. The version information is necessary to determine what scripts should be run to upgrade the database from a given version to the current. The version should be incremented for each planned deployment / milestone

The sample uses a simple versioning scheme. M{milestone #} e.g.

Original changes – M1

Upgrade to tbl\_ProductPricing – M2

In VSTS we use a more complex scheme since we need to deal with release branches as well as the master branch. It is of the format. M{sprint #}.{optional hotfix milestone}. E.g. M116, M117.1, M117.2

Script Authoring

There are 2 types of scripts. The requirement for scripts is that they are re-runnable and idempotent. This is because scripts might need to be re-run in case of failures.

1. Generated delta scripts – these are scripts for objects which can be auto-generated since they can be simply replaced or are new objects to the system. Examples of these objects are:
   * Stored procedures
   * User defined functions
   * User defined types
   * New Tables
   * …

See Upgrade\ToM2\ToM2Generated.sql

1. Data migration scripts – these are scripts written by the developer which migrate the data from Schema A -> Schema B. These typically contain custom business logic which cannot be generated.

See Upgrade\ToM2\Offline\upd\_ToM2OfflinePopulateProductPrice.sql

1. Script ordering – there needs to be a well defined order for running scripts. This can be achieved via a manifest file or alternatively by using a naming convention for the SQL scripts. VSTS and the sample uses a manifest.

Upgrade Orchestration

1. Take the application offline to prevent any requests.
2. Querying the current database version from the database. This would be M1.
3. Finding the manifest files (ToM2.xml) for all versions > current version. Ordering the manifest files based on their versions and executing the scripts in each file in order. Here are the scripts which will be executed in this case:

<?xml version="1.0" encoding="utf-8" ?>

<scripts>

<script>upd\_ToM2OfflinePopulateProductPrice.sql</script>

<script>ToM2Generated.sql</script>

</scripts>

1. The database version is upgraded to latest in ToM2Generated.sql
2. Bring the application online.

## Partially Online Upgrade

Database Versioning

Same as above.

Script Authoring

The key difference here is that the “Data Migration Scripts” need to be factored into online and offline scripts.

1. Online Scripts will do order of data operations while the system is live. It keeps tracks of updates and changes to existing rows via triggers and migrates all data. Characteristics of online scripts are:
   1. They are idempotent and re-runnable.
   2. They do work and commit in small transactions, to not block existing requests or fill up the transaction log.
2. Offline scripts are mainly used for metadata operations or operations that will not take very long e.g. rename and alter any objects. In this case it will drop the Price column on tbl\_Product.

Upgrade Orchestration

1. Querying the current database version from the database. This would be M1.
2. Finding the manifest file (ToM2Online.xml) where version > current version. Execute the scripts where online=true.

<?xml version="1.0" encoding="utf-8" ?>

<scripts>

<script online="true">upd\_ToM2OnlinePopulateProductPrice.sql</script>

<script online="false">upd\_ToM2OnlineDropPriceColumnAndTrigger.sql;ToM2Generated.sql</script>

</scripts>

1. Take the application offline to prevent any requests.
2. Run any scripts with online=false.
3. The database version is upgraded to latest in ToM2Generated.sql
4. Bring the application online.

## Online Upgrade with some errors.

Database Versioning

Same as above.

Script Authoring

Same as above.

Component Versioning

In order to achieve online upgrade, application code which talks to the database (typically the application tier nodes in a 3 tier app) should be able to negotiate which version of the database it’s talking to. This is so that it can formulate and pass the right parameters to stored procedures / schema and process the results correctly. Essentially you need to treat the API surface area between your application tier and data tier as a contract.

In our example, we have ProductComponent which is the main interface used to access the database. It has 2 versions:

private static SortedList<int, Type> s\_types = new SortedList<int, Type>()

{

{ 1, typeof(ProductComponent) },

{ 2, typeof(ProductComponent2) },

};

We use object inheritance to be able to have different implementations to talk to the different database versions. E.g.

GetProduct implementation on ProductComponent processes 1 result set.

public virtual Product GetProduct(int productid)

{

using (SqlCommand cmd = new SqlCommand("prc\_GetProduct"))

{

cmd.CommandType = CommandType.StoredProcedure;

cmd.Connection = m\_connection;

cmd.Parameters.AddWithValue("@productId", productid);

using (SqlDataReader reader = cmd.ExecuteReader())

{

reader.Read();

Product product = new Product();

product.ProductId = (int)reader["ProductId"];

product.Name = reader["Name"].ToString();

product.Prices = new Dictionary<string, decimal>();

product.Prices.Add("USA", (decimal)reader["Price"]);

return product;

}

}

}

GetProduct implementation on ProductComponent2 processes 2 result sets

public override Product GetProduct(int productid)

{

using (SqlCommand cmd = new SqlCommand("prc\_GetProduct"))

{

cmd.CommandType = CommandType.StoredProcedure;

cmd.Connection = m\_connection;

cmd.Parameters.AddWithValue("@productId", productid);

using (SqlDataReader reader = cmd.ExecuteReader())

{

reader.Read();

Product product = new Product();

product.ProductId = (int)reader["ProductId"];

product.Name = reader["Name"].ToString();

product.Prices = new Dictionary<string, decimal>();

reader.NextResult();

while (reader.Read())

{

product.Prices.Add(reader["CountryCode"].ToString(), (decimal)reader["Price"]);

}

return product;

}

}

}

The last piece which is required is using the Factory design pattern so that we can control which version of the component is handed out. The code for this can be seen in ProductComponentFactory.

It basically does this:

1. Walks through the list of registered components in reverse order (favoring the most recent version)
2. Finds the first version of the component which matches the database version and returns it.
3. There is a bit of caching logic which determines that once we are the latest version, we can stop querying for the database version on each component creation.

The consuming code would look like:

using (ProductComponent component = ProductComponentFactory.GetProductComponent(connectionString))

{

Product product = component.GetProduct(1);

}

Upgrade Orchestration

1. Upgrade the binaries of the application to the latest version.
2. Querying the current database version from the database. This would be M1.
3. Finding the manifest file (ToM2Online.xml) where version > current version. Execute all the scripts. The additional requirement is that all scripts where online=false will be run in 1 transaction to prevent a failure causing an inconsistent state. In this case upd\_ToM2OnlineDropPriceColumnAndTrigger.sql;ToM2Generated.sql will be executed in 1 transaction.

<?xml version="1.0" encoding="utf-8" ?>

<scripts>

<script online="true">upd\_ToM2OnlinePopulateProductPrice.sql</script>

<script online="false">upd\_ToM2OnlineDropPriceColumnAndTrigger.sql;ToM2Generated.sql</script>

</scripts>

1. The database version is upgraded to latest in ToM2Generated.sql

## Fully Online Upgrade

So what errors does this leave the application susceptible to. There are 2 types of race conditions:

1. Cases where you have opened a connection, gotten the database version and the database version changes before you execute your command will fail with a parameter mismatch exception.
2. Currently executing commands can fail with SQLServer error 2801 - The definition of object '%.\*ls' has changed since it was compiled.

In order to solve this problem we need to introduce locking. The way this works is:

1. When we go to read the current version in ProductComponent.GetComponentVersion() we end up taking a shared lock for the duration of the connection (until the connection is disposed)

using (SqlCommand cmd = new SqlCommand(@"

DECLARE @result INT

EXEC @result = sp\_getapplock @Resource = 'DatabaseVersion', @LockMode = 'Shared', @LockOwner = 'Session'

IF (@result < 0)

BEGIN

RAISERROR('Error acquiring shared lock', 16, 1)

RETURN

END

SELECT CONVERT(int, value) FROM fn\_listextendedproperty('DatabaseVersion', default, default, default, default, default, default)"))

{

cmd.Connection = m\_connection;

return (int)cmd.ExecuteScalar();

}

1. Upgrade will take an exclusive lock for all scripts with online=false. This will ensure that all commands are drained.

DECLARE @result INT

WHILE 1 = 1

BEGIN

EXEC @result = sp\_getapplock @Resource = 'DatabaseVersion', @LockMode = 'Exclusive', @LockOwner = 'Transaction', @LockTimeout = 5000

if @result >= 0

BREAK

-- Sleep 10 seconds

WAITFOR DELAY '00:00:10';

END

Other considerations:

1. All code must dispose() of connections, otherwise you end up with an orphaned session lock until the connection is closed
2. On high volume systems you will probably need to tweak the duration the exclusive lock waits for and the step up interval to prevent blocking of commands, at the same time allowing the upgrade to be successful.