

## CHAPTER 1

# Working with Azure Database Services Platform

To get the most out of the power of cloud services we need to introduce how to deal with relational data using a fully managed RDBMS service, which is, in Microsoft Azure, SQL Database.

SQL Database is the managed representation of Microsoft SQL Server in the cloud. Can be instantiated in seconds/minutes and it runs even large applications without minimal administrative effort.

## Understanding the Service

SQL Database can be viewed as a sort of Microsoft SQL Server *as-a-Service*, where those frequently-addressed topics can be entirely skipped during the adoption phase:

- License management planning: SQL Database is a pay-per-use service. There is no upfront fee and migrating between tiers of different pricing is seamless
- Installation and management: SQL Database is ready-to-use. When we create a DB, the hardware/software mix that runs it already exists and we only get the portion of a shared infrastructure we need to work. High-availability is guaranteed and managed by Microsoft and geo-replication is at a couple of clicks away.

- Maintenance: SQL Database is a PaaS, so everything is given to us as-a-Service. Updates, patches, security and Disaster Recovery are managed by the vendor. Also, databases are backup continuously to provide end-users with point-in-time restore out-of-the-box.

From the consumer perspective, SQL Database has a minimal feature misalignment with the plain SQL Server and, like every Platform-as-a-Service, those touch points can be inferred by thinking about:

- Filesystem dependencies: we cannot use features that correlates with customization of the underlying operating system, like file placement, sizes and database files which are managed by the platform.
- Domain dependencies: we cannot join a SQL Database “server” to a domain, since there is no server from the user perspective. So, we cannot authenticate with Windows authentication; instead, a growing up support of Azure Active Directory is becoming a good replacement of this missing feature.
- Server-wide commands: we cannot (we would say “fortunately”) use commands like SHUTDOWN, since everything we make is made against the *logical* representation of the Database, not to its underlying physical topology.

In addition to the perceptible restrictions, we have some more differences related to which is the direction of the service and the roadmap of advanced features. Since we cannot know **why** some of the features below are not supported, we can imagine they are related to offer a high-level service cutting down the surface area of potential issues of advanced commands/features of the plain SQL Server.

---

For a complete comparison of supported features between SQL Database and SQL Server, refer to this page: <https://docs.microsoft.com/en-us/azure/sql-database/sql-database-features>

---

At least, there are service constraints which add the last set of differences, for example:

- Database sizes: at the time of writing, SQL Database supports DB up to 1TB of size (the counterpart is 64TB)
- Performance: despite there are several performance tiers of SQL Database, with the appropriate VM set, SQL in a VM can exceed largely the highest tier of it.

---

For a good introduction of how to understand the differences between the features supported in both products, refer to this page: <https://docs.microsoft.com/en-us/azure/sql-database/sql-database-paas-vs-sql-server-iaas>.

---

## Connectivity Options

We cannot know the exact SQL Database actual implementation, outside of what Microsoft discloses in public materials. However, when we create an instance, it has the following properties:

- Public URL: in the form [myServer].database.windows.net.  
Public-faced on the Internet and accessible from everywhere.

---

Yes, there are some security issues to address with this topology, since there is no way to deploy a SQL Database in a private VNet.

---

- Connection modes:
  - from outside Azure, by default, each session between us and SQL Database passes through a Proxy, which connects to the actual ring/pool of the desired instance
  - from inside Azure, by default, clients are redirect by the proxy to the actual ring/pool after the handshake, so overhead is reduced. If we are using VMs, we must be sure the outbound port range 11000-11999 is open.

We can change the default behavior of the Proxy by changing this property: <https://msdn.microsoft.com/library/azure/mt604439.aspx>. Note that, while connecting from outside Azure, this means multiple IPs can be configured to outbound firewall rules.

---

- Authentication:
  - Server login: by default, when we create a Database, we must create a *server* before it. A server is just a logical representation of a container of database, no dedicated physical server is mapped to it. This server has an administrator credential, which have full permission on every DB created in it.
  - Database login: we can create additional credentials tied to specific databases
  - Azure AD login: we can bind Azure AD users/groups to the server instance to provide an integrated authentication experience
  - Active Directory Universal Authentication: only through a proper version of SSMS, clients can connect to SQL Database using a MFA
- Security:
  - Firewall rules: to allow just some IPs to connect to SQL Database, we can specify firewall rules. They are represented by IP ranges.
  - Certificates: by default, an encrypted connection is established. A valid certificate is provided, so it is recommended (to avoid MITM attacks) to set to “false” the option “Trust Server Certificate” while connecting to it.

Given this information above as the minimum set of knowledge to connect to a SQLDB instance, we can connect to it using the same tooling we use for SQL Server. SSMS is supported (few features won’t be enabled however), client connectivity through the SQL Client driver is seamless (as it would be a SQL Server instance) and the majority of tools/applications will continue to work by only changing the connection string.

## Libraries

In recent years Microsoft has been building an extensive support to non-Microsoft technology. This means that now we have a wide set of options to build our applications, using Azure services, even from outside the MS development stack. Regarding SQL Database, we can now connect to it through official libraries, as follows:

- C#: ADO.NET, Entity Framework (<https://docs.microsoft.com/en-us/sql/connect/ado-net/microsoft-ado-net-for-sql-server>)
- Java: Microsoft JDBC Driver (<https://docs.microsoft.com/it-it/sql/connect/jdbc/microsoft-jdbc-driver-for-sql-server>)
- PHP: Microsoft PHP SQL Driver (<https://docs.microsoft.com/it-it/sql/connect/php/microsoft-php-driver-for-sql-server>)
- Node.js: Node.js Driver (<https://docs.microsoft.com/en-us/sql/connect/node-js/node-js-driver-for-sql-server>)
- Python: Python SQL Driver (<https://docs.microsoft.com/en-us/sql/connect/python/python-driver-for-sql-server>)
- Ruby: Rudy Driver (<https://docs.microsoft.com/en-us/sql/connect/ruby/ruby-driver-for-sql-server>)
- C++: Microsoft ODBC Driver (<https://docs.microsoft.com/en-us/sql/connect/odbc/microsoft-odbc-driver-for-sql-server>)

This extended support makes SQL Database a great choice for who are adopting a RDBMS, for both new and existing solutions.

## Sizing & Tiers

The basic consumption unit of SQL Database is called DTU (Database Transaction Unit), which is defined as a blended measure of CPU, memory and I/O. We cannot “reserve” to our SQLDB instance a fixed size VM. Instead, we choose:

- Service Tier: it defines which features the DB instance has and the range of DTU between we can move it.
- Performance Level: it defines the reserved DTU for the DB instance.

Both for the official recommended approach as for the experience matured in the field, we strongly encourage to avoid too much complicated in-advance sizing activities to know exactly which tier our application needs, before testing it. We think that an order of magnitude can be of course inferred by in-advance sizing, but a more precise estimation of consumption has to be made after a measured pilot, where we can see how the new/existing application uses the database tier and, consequently, how much the DB instance is stressed by that.

---

Like in any other service offered in a PaaS fashion, we are subject to throttling, since we reach the limits of the current performance level.

For years consultants tried to explain to clients there is no way to predict exactly which is the actual performance level needed for an application since, by design, each query is different and even the most trivial KPI (i.e., queries-per-second) is useless without the proper benchmark information.

---

To understand how the benchmark behind the DTU blend is developed, see this article: <https://docs.microsoft.com/en-us/azure/sql-database/sql-database-benchmark-overview>

---

At the time of writing, SQLDB supports those Service Tiers and Performance Levels (Figure 1-1):

- Basic: it supports only a 5DTU level, with 2GB of max DB size and few simultaneous requests.
- Standard: it supports a range of 10-100DTU levels, with 250GB of max DB size and moderate simultaneous requests.
- Premium: it supports the largest levels (125-4000DTU), with 4TB of max DB size and the highest simultaneous requests.

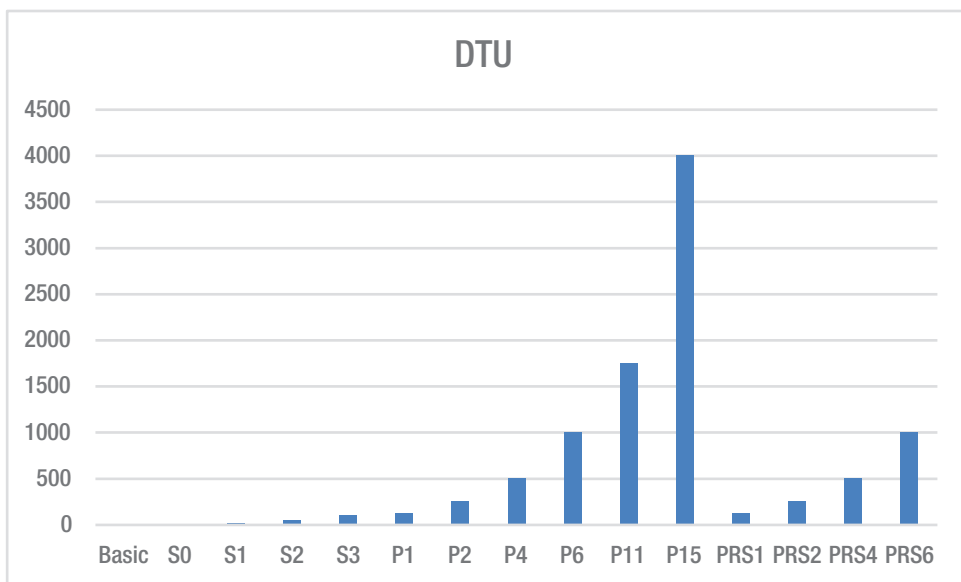
Unfortunately, service tiers and resource limits are subject to continuous change over time. We can find updated information here:

<https://docs.microsoft.com/en-us/azure/sql-database/sql-database-service-tiers>

<https://docs.microsoft.com/en-us/azure/sql-database/sql-database-resource-limits>

In addition, Premium levels offer In-Memory features, which are not available in other tiers.

- Premium RS: it supports the 125-1000DTU levels, with the same constraints of the corresponding Premium level.



**Figure 1-1.** This chart shows clearly the DTU ratio between different Tiers/Levels of SQL Database

Premium RS is a recent Tier which offers the same features as the Premium counterpart, while guaranteeing a reduced durability, which results in a sensible cost saving and more performance for I/O operations. Unfortunately, the service did not pass the preview phase and it has been scheduled for dismissal on January 31, 2019.

---

## Designing SQL Database

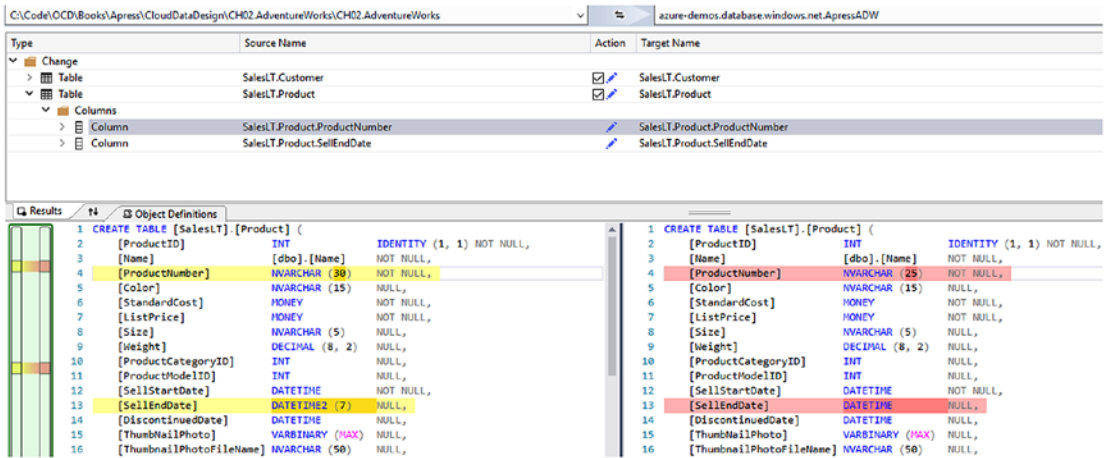
SQL Database interface is almost fully compatible with tooling used for SQL Server, so in most cases previous tooling should work with no specific issues. However, since Visual Studio offers the capability to manage the development process of a DB from inside the IDE, it is important to mention it.

Database Projects are Visual Studio artefacts which let DBA to develop every DB object inside Visual Studio, with an interesting feature set to gain productivity:

- **Compile-time checks:** VS checks the syntax of the written SQL and highlights errors during a pseudo-compilation phase. In addition, it checks references between tables, foreign keys and, more generally, gives consistence to the entire database before publishing it.
- **Integrated publishing:** VS generates the proper scripts to create (or alter) the database, based on what it finds at the target destination. It means that the target DB can even already exists and Visual Studio will run the proper change script against it without compromise the consistency.
- **Data generation:** to generate sample data in the database tables
- **Pre/Post-deployment scripts:** to execute custom logic before/after the deployment
- **Source control integration:** by using Visual Studio, it is seamless to integrate with Git or TFS to version our DB objects like code files.

Using Database Projects (or other similar tools) to create and manage the development of the database is a recommended approach (Figure 1-2), since it gives a central view of the Database Lifecycle Management. Finally, Visual Studio supports SQL Database as a target Database, so it will highlight potential issues or missing features during the Database development.





**Figure 1-2.** This image shows the Schema Compare features of Database Projects, which also targets SQL Database in order to apply changes with a lot of features (data loss prevention, single-change update, backups, etc).

We can use Database Projects even at a later stage of DB development, using the wizard “Import Database” by right-clicking the project node in Visual Studio. This wizard creates the VS objects by a reverse engineering process on the target Database.

There are other options to design databases. Official documentation follows:

- SSMS design: <https://docs.microsoft.com/en-us/azure/sql-database/sql-database-design-first-database>
- .NET design: <https://docs.microsoft.com/en-us/azure/sql-database/sql-database-design-first-database-csharp>
- Code-first design: [https://msdn.microsoft.com/en-us/library/jj193542\(v=vs.113\).aspx](https://msdn.microsoft.com/en-us/library/jj193542(v=vs.113).aspx)

## Multi-tenancy

Since many applications should be multi-tenant, where one deployment can address many customers, even the Database tier should follow this approach. This is clearly an architect choice but, since it can have consequences on performance/costs of the SQLDB instance, we analyze the various options.

## One Database for Each Tenant

This is the simplest (in terms of design) scenario, where we can also have a single-tenant architecture which we redeploy once for every client we acquire. It is pretty clear that, in case of few clients, can be a solution, while it isn't where clients are hundreds or thousands.

This approach highlights those pros/cons:

- Advantages:
  - We can retain existing database and applications and redeploy them each time we need.
  - Each customer may have a different service level and disaster recovery options
  - An update which is specific to a tenant (i.e., a customization) can be applied to just the database instance affected, leaving others untouched.
  - An optimization, which involves the specific usage of a table, can be applied to that DB only. Think about an INDEX which improves TenantX queries but worsens other tenants.
- Disadvantages:
  - We need to maintain several Databases which, in the best case are just copies with the same schema and structure. In the worst case they can be different, since they proceed in different project forks: but this is another topic, related to business, out of the scope of the chapter.
  - Every DB will need a separate configuration of features on the Azure side. Some of them can be configured at the server side (the logical server) but others are specific.
  - Every DB has a performance level and corresponding costs, which in most cases is not efficient in terms of pooling.
  - In case of Staging/Test/Other development environment, they should be made specifically for each client.

Those are just few of the pros/cons of this solution. To summarize this approach, we would say it is better for legacy applications not designed to be multi-tenant and where new implementations are very hard to achieve.

## Single Database with a Single Schema

In this scenario, we are at the opposite side, where we use just ONE database for ALL the clients now or in the future. We would probably create tables which contains a discriminant column like “TenantID” to isolate tenants.

This approach highlights those pros/cons:

- Advantages:
  - A single DB generates a far less effort to maintain and monitor it.
  - A performance tuning which is good for every client, can be applied once
  - A single DB generates just one set of features to configure and a single billing unit
- Disadvantages:
  - An update on the DB potentially affects every deployment and every customer of the solution. This results in harder rolling upgrade of the on top application.
  - If a client consumes more than others, the minor clients can be affected and the performance of each one can vary seriously. In other words, we cannot isolate a single tenant if needed.
  - This is the simplest scenario while dealing with a new solution. During the development phase we have just one database to deal with, one or more copies for other environments (Staging/UAT/Test) and a single point of monitoring and control when the solution is ready to market. However this can be just the intermediate step between a clean-and-simple solution and an elastic and tailor-made one.

## Single Database with Different Schemas

This solution is a mix between the first one and the second, since we have just one database instance, while every table is replicated once for every schema in the database itself, given that every Tenant should be mapped to a specific schema.

This approach has the union of pros/cons of the “One database for each tenant” and “Single Database with single schema” approaches.

In addition, in case we want to isolate a tenant in its own dedicated DB, we can move its schema and data without affecting others.

## Multiple Logical Pools with a Single Schema Preference

The latest approach is the one that can achieve the best pros and the less cons, compared to the previous alternatives. In this case, we think about Pools instead of Database, where a Pool can be a DB following the “Single Database with a single schema pattern” which groups a portion of the tenants.

Practically, we implement the DB as we are in the Single Database approach, with a TenantID for every table which needs to be isolated. However, falling in some circumstances, we “split” the DB into another one, keeping just a portion of tenant in the new database. Think about those steps:

1. First the DB is developed once, deployed and in production
2. New clients arrive, new TenantIDs are burned and tables now contains data of different clients (separated by a discriminant).
3. Client X needs a customization or a dedicated performance, a copy of the actual DB is made and the traffic of client X are directed to the appropriate DB instance.
4. Eventually the data of client X in the “old” DB can be cleaned up

Given the pros of that approach, we can mitigate the disadvantages as follows:

- An update on the DB potentially affects every deployment and every customer of the solution. This results in harder rolling upgrade of the on top application.
- We can migrate one tenant, perform an upgrade on it and then applying the update on every Logical Pool.

- If a client consumes more than others, the minor clients can be affected and the performance of each one can vary seriously. In other words, we cannot isolate a single tenant if needed.
- We can migrate one or more tenant to a dedicated DB (Logical Pool)

The remaining disadvantage is the effort needed to write the appropriate procedures/tooling to migrate tenants between DBs and create/delete/update different DBs with minimal manual intervention. This is a subset of the effort of the first approach with maximum degree of elasticity.

## Index Design

Indexes are standard SQL artefacts which helps to lookup data in a table. Practically speaking, for a table with millions or rows, an index can help *seeking* to the right place where the records are stored, instead of *scanning* the whole table looking for the results. A theoretical approach to index design is out of the scope of the book, so we focus on:

- Index creation
- Index evaluation
- Index management

## Index Creation

Let's consider the following table (SalesLT.Customer of the AdventureWorksLT sample Database):

```
CREATE TABLE [SalesLT].[Customer] (
    [CustomerID] INT IDENTITY (1, 1) NOT NULL,
    [NameStyle] [dbo].[NameStyle] CONSTRAINT [DF_Customer_NameStyle]
    DEFAULT ((0)) NOT NULL,
    [Title] NVARCHAR (8) NULL,
    [FirstName] [dbo].[Name] NOT NULL,
    [MiddleName] [dbo].[Name] NULL,
    [LastName] [dbo].[Name] NOT NULL,
    [Suffix] NVARCHAR (10) NULL,
    [CompanyName] NVARCHAR (128) NULL,
```

```

[SalesPerson] NVARCHAR (256) NULL,
[EmailAddress] NVARCHAR (50) NULL,
[Phone] [dbo].[Phone] NULL,
[PasswordHash] VARCHAR (128) NOT NULL,
[PasswordSalt] VARCHAR (10) NOT NULL,
[rowguid] UNIQUEIDENTIFIER CONSTRAINT [DF_Customer_rowguid]
DEFAULT (newid()) NOT NULL,
[ModifiedDate] DATETIME CONSTRAINT [DF_Customer_ModifiedDate]
DEFAULT (getdate()) NOT NULL,
CONSTRAINT [PK_Customer_CustomerID] PRIMARY KEY CLUSTERED ([CustomerID]
ASC),
CONSTRAINT [AK_Customer_rowguid] UNIQUE NONCLUSTERED ([rowguid] ASC)
);

```

---

While creating a SQL Database DB instance, we can even choose between a blank one (the common option) or a preconfigured and populated AdventureWorksLT Database

---

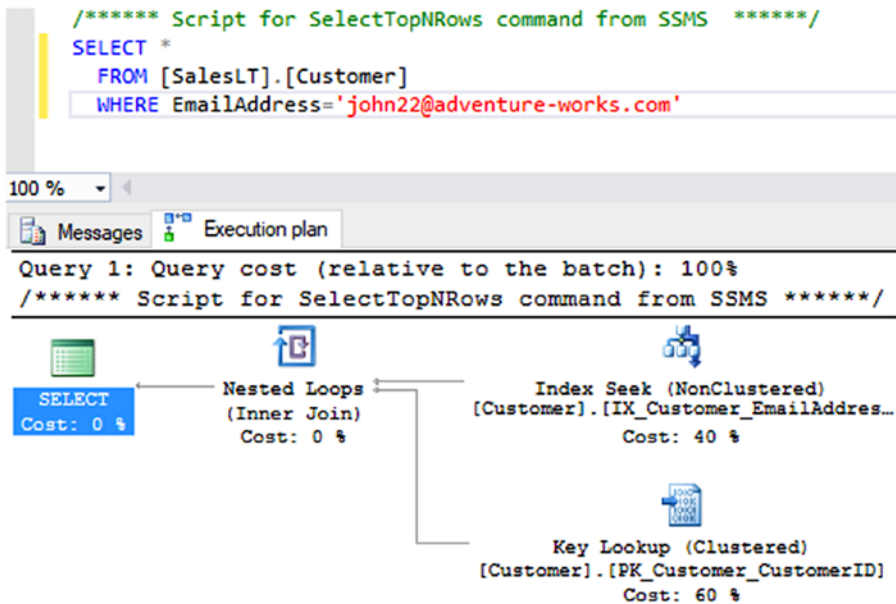
By default the following index is created:

```

CREATE NONCLUSTERED INDEX [IX_Customer_EmailAddress]
ON [SalesLT].[Customer]([EmailAddress] ASC);

```

However, despite a table definition is about requirements, an index definition is about usage. The index above will produce better performance in queries filtering the EmailAddress field. However, if the application generates the 99% of queries filtering by the CompanyName field, this index is not quite useful and it only worse the write performance (Figure 1-3).



**Figure 1-3.** This query uses the index, producing a query cost only on the seek operation (good). In SSMS, to see the query plan, right click the query pane and select “Display Estimated Execution Plan”.

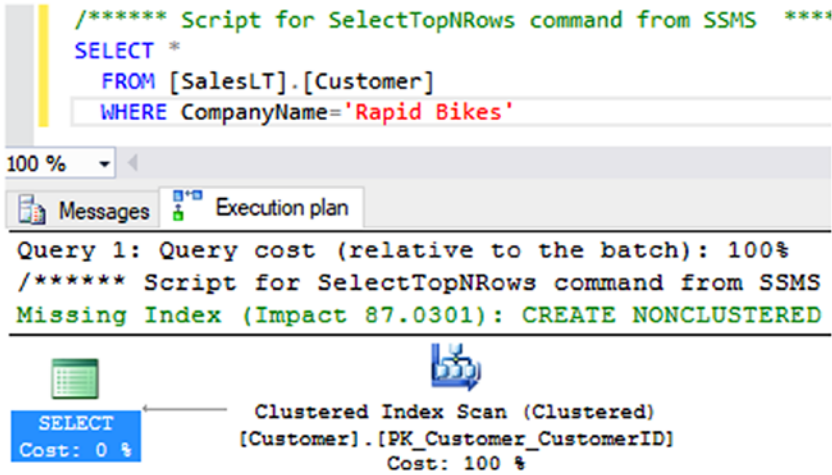
So, indexes are something related to time and usage: today we need an index and tomorrow it can be different, so despite application requirements are the same, indexes can (must) change over time.

## Index Evaluation

Which index should we create? First, we can write a Database without any indexes (while some are direct consequences of primary keys). Write performance will be the fastest while some queries will be very slow. An option can be to record each query against the database and analyze them later, by:

- Instrumenting on the application side: every application using the DB should log the actual queries.
- Instrumenting on the SQL side: the application is unaware of tracing, while SQL saves every query passing on the wire

Using the idea above, let's try to edit the query above filtering by CompanyName (Figure 1-4):



**Figure 1-4.** In this case, no seeking is performed. Instead, SSMS suggest us to create an Index, since the 100% of the query cost is on scanning

For completeness, SSMS suggest us this creation script:

```

/*
USE [AdventureWorksLT]
GO
CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]
ON [SalesLT].[Customer] ([CompanyName])
GO
*/

```

But SSMS cannot tell us if the **overall** performance impact is positive, since write queries (i.e., a high rate of updates on the CompanyName field) can be slower due to index maintenance.



## Index Management

Once an index is created and it is working, it grows and it gets fragmented. Periodically, or even manually but in a planned fashion, we need to maintain indexes by:

- **Rebuilding:** the index is, as the word suggests, rebuilt, so a fresh index is created. Sometimes rebuilding needs to take a table offline, which is to evaluate carefully in production scenarios.
- **Re-organizing:** the index is actually defragmented by moving physical pages in order to gain performance. It is the lightest (but often longest) version to maintain an index.

We can use this query to have a look to the current level of fragmentation:

```
SELECT
DB_NAME() AS DBName,
OBJECT_NAME(pstats.object_id) AS DBTableName,
idx.name AS DBIndexName,
ips.index_type_desc as DBIndexType,
ips.avg_fragmentation_in_percent as DBIndexFragmentation
FROM sys.dm_db_partition_stats pstats
INNER JOIN sys.indexes idx
ON pstats.object_id = idx.object_id
AND pstats.index_id = idx.index_id
CROSS APPLY sys.dm_db_index_physical_stats(DB_ID(),
      pstats.object_id, pstats.index_id, null, 'LIMITED') ips
ORDER BY pstats.object_id, pstats.index_id
```

While with this statement we perform Index Rebuild:

```
ALTER INDEX ALL ON [table] REBUILD with (ONLINE=ON)
```

---

**Note** that “with (ONLINE=ON)” forces the runtime to keep table online. In case this is not possible, SQL raises an error which can be caught to notify the hosting process.

---

## Automatic Tuning













SQL Database integrates the Query Store, a feature that keeps tracks of every query executed against the Database to provide useful information about performance and usage patterns. We see Query Store and Query Performance Insight later in chapter but, in the meantime, we talk about Index Tuning.

Since indexes can change over time, SQL Database can use the recent history of database usage to give an advice (Figure 1-5) of which Indexes are needed to boost the overall performance. We say “overall”, because the integrated intelligent engine reasons as follows:

1. By analyzing recent activities, comes out that an Index can be created on the table T to increase the performance
2. Using the history data collected up to now, the estimated performance increment would be P% of DTU
3. If we apply the Index proposal, the platform infrastructure takes care of everything: it creates the index in the optimal moment, not when DTU usage is too high or storage consumption is near to its maximum.
4. Then, Azure monitors the index’s impacts on performance: not only the positive (estimated) impact, but even the side effects on queries that now can perform worse due to the new Index.
5. If the overall balance is positive, the index is kept, otherwise, it will be reverted.

As a rule, if the index created is good to stay, we can include it in the Database Project, so subsequent updates will not try to remove it as consequence of re-alignment between source code and target database.

## Tuning history

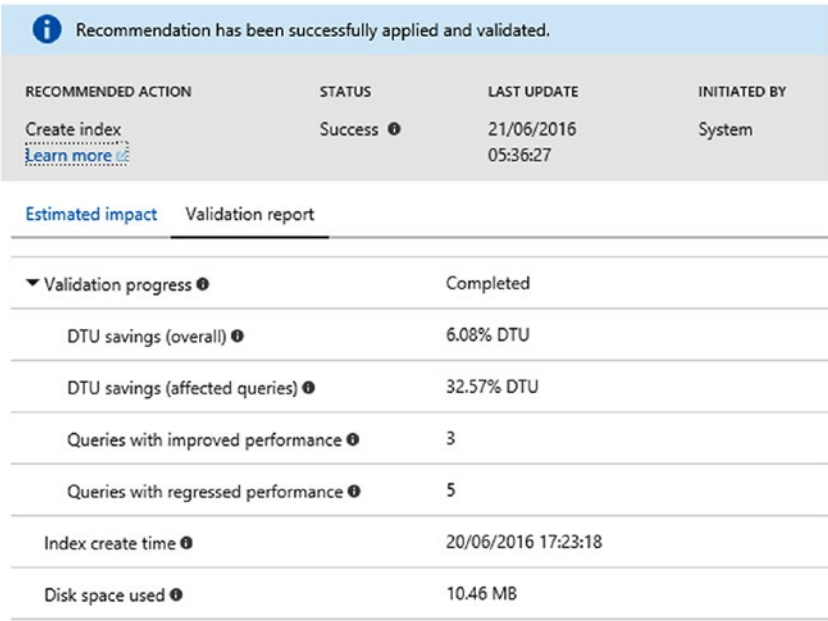
ACTION	RECOMMENDATION DESCRIPTION	STATUS	TIME
 <b>CREATE INDEX</b> Initiated by: System	Table: [redacted] Indexed columns: [redacted]	 Success	20/07/2017 18:10:02
 <b>CREATE INDEX</b> Initiated by: System	Table: [redacted] Indexed columns: [redacted]	 Reverted	04/07/2017 17:51:31
 <b>CREATE INDEX</b> Initiated by: System	Table: [redacted] Indexed columns: [redacted]	 Success	05/06/2017 22:46:21
 <b>CREATE INDEX</b> Initiated by: System	Table: [redacted] Indexed columns: [redacted]	 Reverted	30/05/2017 08:43:09
 <b>CREATE INDEX</b> Initiated by: System	Table: [redacted] Indexed columns: [redacted]	 Success	23/04/2017 14:48:55
 <b>CREATE INDEX</b> Initiated by: System	Table: [redacted] Indexed columns: [redacted]	 Success	20/04/2017 12:28:24

**Figure 1-5.** Here we have a few recommendations, where someone has been deployed successfully while others have been reverted.

---

**Note** we should keep Database Project and database aligned to avoid “drifts”, which can introduces alterations in the lifecycle of the database. An example of the “classic” drift is the quick-and-dirty update on the production Database, which is lost if not promptly propagated in the Database Projects. Another option could be to define a set of common standard indexes (“factory defaults”) and accept that automatic tuning is going to probably be better at adapting to new workload patterns (which doesn’t mean the initial effort to define “factory defaults” shouldn’t be done at all or that regular review of new indexes shouldn’t be done at all).

---



**Figure 1-6.** This is a detail of the impacts on the performance after an automatic Index has been applied to the Database.

In the image above (Figure 1-6), we see how that Automated Tuning is successfully for this Index. We see a global gain of 6% DTU (which is a huge saving) and, relatively to impacted queries, a 32% DTU savings. Since we are talking about indexes, there’s also a connected consumption of storage, which is explicated as about 10MB more.

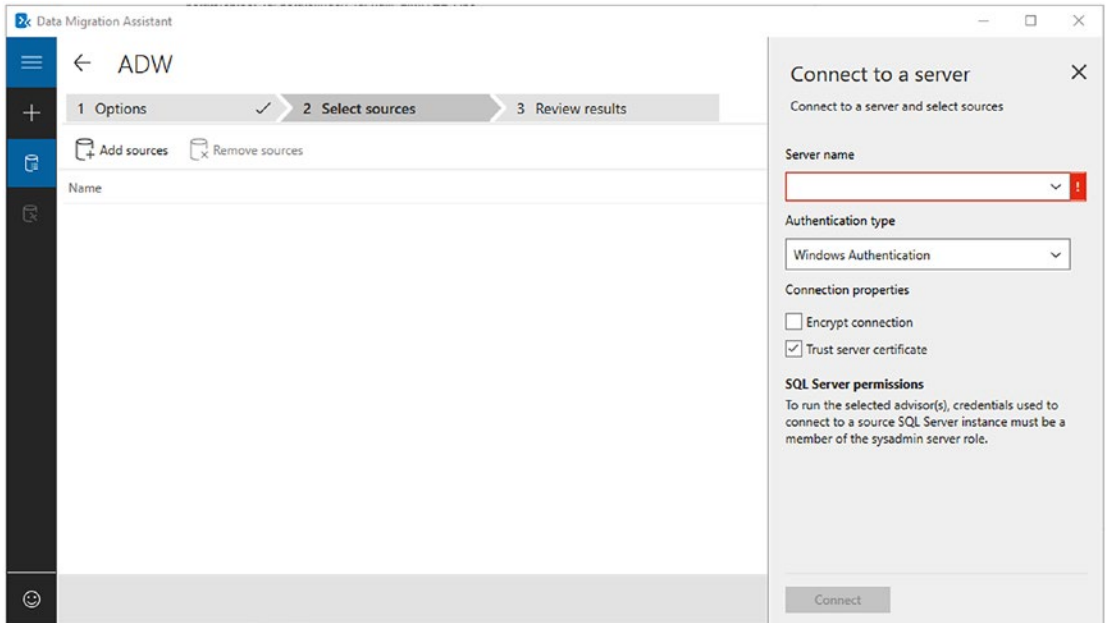
## Migrating an Existing Database

Not every situation permits to start from scratch when we are talking about RDBMS. Actually, the majority of solutions we’ve seen in the last years moving to Azure, made it by migrate existing solutions. In that scenario, Database migration is a key step to the success of the entire migration.

## Preparing the Database

To migrate an existing SQL Server database to Azure SQL Database we must check in advance if there are well-known incompatibilities. For instance, if the on-premises DB makes use of cross-database references, forbidden keywords and deprecated constructs,

the migration will probably fail. There is a list of unsupported features (discussed before) which we can check one-by-one or we can rely on an official tool called Data Migration Assistant (<https://www.microsoft.com/en-us/download/details.aspx?id=53595>).



**Figure 1-7.** DMA helps to identify in the on-premises database which features are used but not supported on the Azure side.

During the DMA assessment (Figure 1-7) we are shown with a list of potential incompatibilities we must address in order to export the on-premises Database. Of course, this process affects the existing database so, we suggest this approach:

- Identify all the critical incompatibilities
  - For the ones which can be fixed transparently to the consuming applications, fix them
  - For the other ones, requiring a rework on the applications side, create a new branch where is possible and migrate the existing applications to use the new structures one-by-one

This can be a hard process itself, even before the cloud has been involved. However, we must do this before setting up the migration process, since we must assume that applications' downtime must be minimal.

When the on-premises DB feature set is 100% compatible with Azure SQL Database V12, we can plan the moving phase.

---

Often, in documentation as well as in the public portal, we see “V12” next to the SQDB definition. V12 has been a way to differentiate two main database server engine versions, supporting different feature sets, in the past but nowadays it’s legacy.

---

## Moving the Database

Achieving a Database migration without downtime is certainly one of the most challenging activity among others. Since the Database is stateful by definition and it often acts as a backend tier for various, eterogeneous systems, we cannot replace it transparently with an in-cloud version/backup of it, as it continuously accumulates updates and new data between the backup and the new switch-on. So, there are at least two scenarios:

1. We prepare a checklist of the systems involved into the DB usage and we plan a service interruption
2. We setup a kind of replica in the cloud, to switch transparently to it in a second time

In the first case, the process can be as follows:

- We discard new connections from outside
- We let last transactions closing gracefully. If some transactions are hanging for too long, we should consider killing them
- We ensure no other clients can connect to it except maintenance processes
- We create a copy of the original Database, sure no other clients are changing its data

- We create schema and data in the Database in the cloud
- We change every applications' configuration in order to point to the new database
- We take online them one-by-one

This approach is the cleanest and simplest approach, even if facing with several concurrent applications.

---

On the other side, if we have NOT direct control over the applications connecting to the database, we must consider to introduce some ad-hoc infrastructure components that denies/throttles the requests coming from sources.

---

In the second case, the process can be harder (and in some scenarios does not guarantee full uptime):

- On the on-premises side, we setup a SQL Server Replication
- We setup a “New Publication” on the publisher side
- We setup the distributor (it can run on the same server)
- We create a Transactional publication and we select all the objects we would like to replicate
- We add a Subscription publishing to the Azure SQL Database (we need to create an empty one before)
- We run/monitor the Replication process under the SQL Service Agent service account

This approach let us continue to operate during the creation and seeding of the remote SQL Database. When the cloud DB is fully synchronized, we can plan the switch phase.

The switch phase can itself introduce downtime since, in some situations, we prefer to not span writes between the two DBs, since the replication is one way and applications pointing to the old database in the switching window, may work with stale data changed, in the meantime, on the SQL Database side.

---

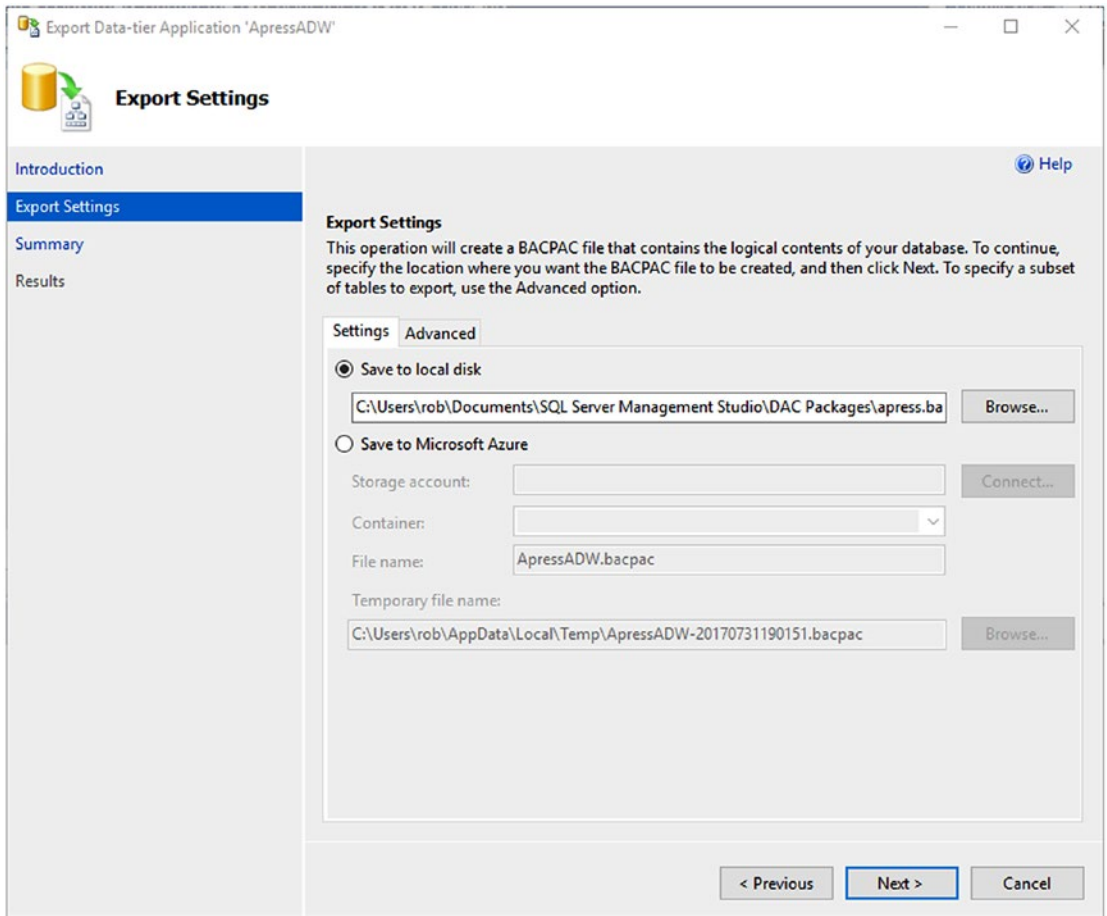
## Exporting the DB

In the previous Option 1, we generically said “copy the DB” but it can be unclear how to do that. SQL Server standard backups (the ones in .BAK format) cannot be restored into SQL Database on the cloud. So “backup” can be interpreted as follows:

- An option is to create a BACPAC on the on-premises side (Figure 1-8) and restore it on the SQL Database side (with PowerShell, the Portal or SQLPackage)
- Another option is to do it manually, by creating Scripts of the entire DB, execute them on the remote DB and then use tools like BCP to move data.

In both cases, we suggest to perform the migration phase using the most performant tier of SQL Database, to reduce the overall time and, consequently, the downtime. You can always downscale later when the migration is completed.





**Figure 1-8.** By right-clicking the database in SSMS, we can choose Tasks->Export Data-tier Application, that starts this export wizard to create a self-contained BACPAC file to use later on Azure or on-premises.

## Using SQL Database

In almost every PaaS, performance and reactions are subject to how we use the service itself. If we use it efficiently the same service can serve request better, at the opposite, if we do not follow best practices, it can generate bottlenecks.

The waste of resources in Cloud Computing and, especially, in PaaS services, is perceived of much more impact than in traditional, on-premises infrastructures. This is due to the nature of the billing mechanism, which is often provided as per-consumption. So, in the whole chapter and book, we keep in mind this sentence:

*“performance and efficiency are features”*

In the following sections we try to setup a set of topics to be addressed while we use SQLDB, to improve efficiency and get the most from the underlying service.

## Design for Failures

This is a valid concept for every single piece of code running in distributed system from the beginning of IT. It does not only fit with SQL Database. When we deal with remote dependencies, something can go wrong; in rare cases connectivity can lack for a second or more, often instead the remote service simply cannot execute the request because it is 100% busy.

With cloud-to-cloud solutions, the first is hard to reproduce, except for specific failures in the Azure region hosting our services. The second instead, it related to the bigger topic called Throttling which, to summarize, is about cutting consumer where the producer is too busy to serve the current incoming request.

Throttling is good. The alternative is a congested service which serves badly every request: we prefer a fast service which declines new operations is too much busy.

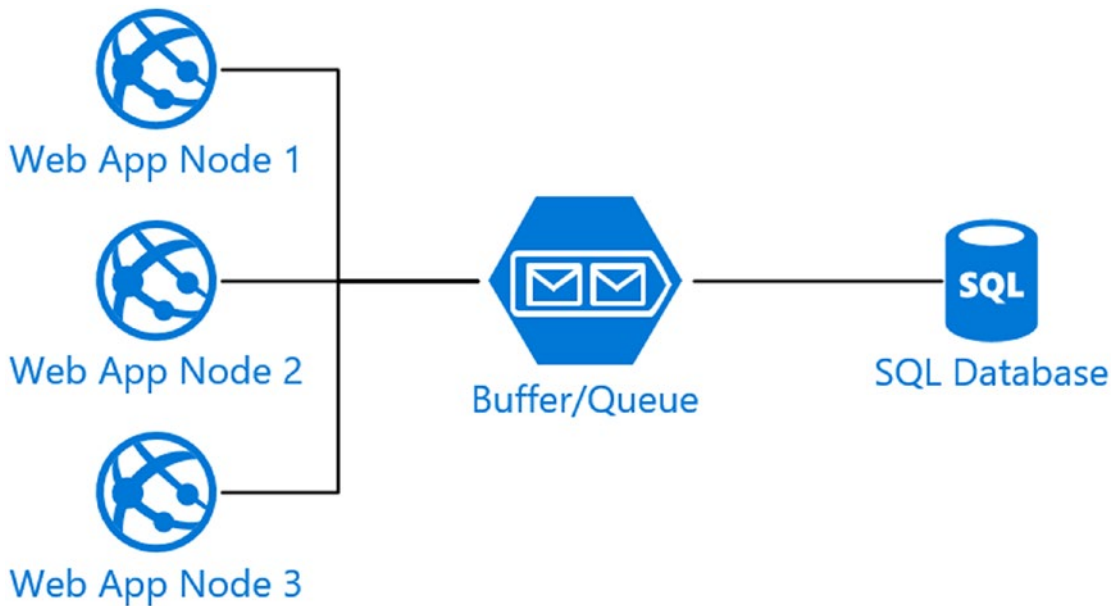
In ideal world, when a remote dependency is not available anymore, the consumer endpoint should gracefully continue to operate without it. An example can be an autonomous system which tries to write on the SQL Database and, in case of failure, it stores the record locally waiting the next availability window of the DB, to send the it eventually. In real world, this can happen too but, at least, we should write code which reacts to failing events of remote dependencies, even just retrying their operation until success.

## Buffering

If we split queries in Read queries and Write queries we easily understand that the first can even fail with few side effects. At the opposite side, a missing write can represent a damage for everyone and if the DB cannot persist the state of the application in a point of time, the application itself have the same responsibility of a DB until service is recovered.

Since this is very hard to achieve (applications which relies on a DB are often stateless and it is desirable they should remain as is), a good approach can be to decouple critical persistence operation between applications and databases, with queues.

Someone can say we are just moving away the problem, since the queue technology must be reliable and available itself. This is why we can use it totally (every write goes to the queue) or partially (only the failed writes go to the queue). In a queue-based solution, the application is the producing actor while another, new component (which takes the items from the queue and writes them in the DB) is the writing actor (Figure 1-9).



**Figure 1-9.** In this scenario we decouple applications from Database access, using a queue to perform operations. This pattern can introduce a new actor (not in the image) which consumes the items from the queue and temporize the queries to the DB to avoid bottlenecks and throttling.

If we mix this pattern with a robust and reliable distributed queue (as Azure Service Bus Queues or Azure Storage Queues) we can write robust code that is resilient in case of failures.

## Retry Policies

The approach above catches all the possible causes of SQL unavailability: either it is for a short timeframe (i.e., for throttling) either it is for a service outage. In the case we cannot or do not want to introduce a new component by modifying the overall topology,

we should at least implement, at the client side, a mechanism to retry queries that fail, to mitigate, at least, the short timeframes where the DB is not available or when it is too busy to reply.

A retry policy can be explained with this pseudo-code:

```
using (SqlConnection connection = new SqlConnection(connectionString))
{
    try
    {
        connection.Open();
        command.ExecuteNonQuery();
    }
    catch (TemporaryUnavailabilityException)
    {
        //Retry
    }
    catch (ThrottlingException)
    {
        //Retry
    }
    catch (Exception)
    {
        //Abort
    }
}
```

We should retry the query where the fault can be assigned to transient faults, faults which are by nature (temporary connectivity issues, huge load) transient and that can be restored quickly. In other cases, like failing runtime logic or too many retry attempts (indicating probably a downtime is occurring) should cause an abort of the operation.

Generally, almost each PaaS service in Microsoft Azure defines a specification where, in case of transient faults, special error codes/messages are returned to the client. This, in conjunction with code written to gracefully handle those faults, lets the applications run seamlessly.

By now, many official Microsoft libraries have native support for transient faults: for SQL Database, Entity Framework client has native support, as well as EF Core. For whom using ADO.NET directly, we suggest to investigate the project Polly

(<http://www.thepollyproject.org/>) which is a library that adds some interesting features related to retries, fallback and exception handling.

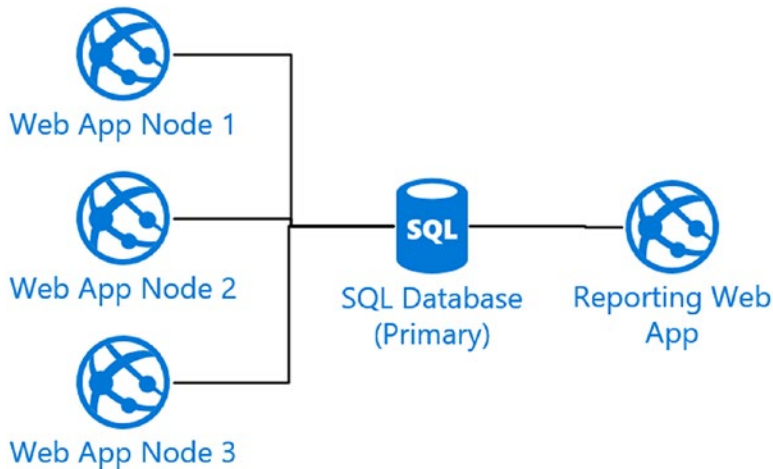
---

For a comprehensive list of transient error codes occurring in SQL Database, see this link: <https://docs.microsoft.com/en-us/azure/sql-database/sql-database-develop-error-messages>.

---

## Split between Read/Write Applications

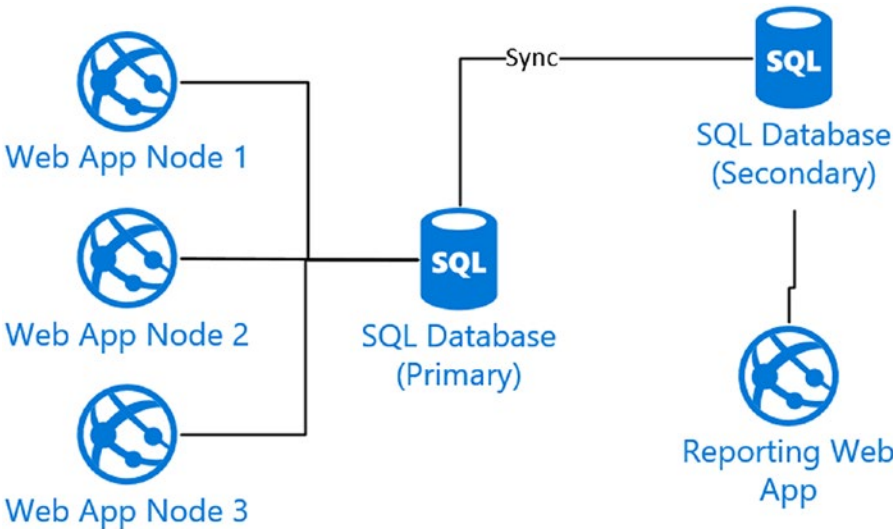
Catching the in-field experience of some companies we worked with, high-end users of SQL Server are considering, before or later, to split application logic routing request to different DB instances (even in different servers/regions), depending on the type of load and/or operation requested.



**Figure 1-10.** Multiple applications act against the primary DB. If the reporting tool is making an intensive, long-running query, Web App nodes can see a degradation of performance.

In the scenario above (Figure 1-10) our perception is that it is not a great idea to point every process on the same DB instance, because few clients with few complex analytical queries can consume the majority of the available resources, slowing down the entire

business processes. Where a Data Warehouse is not an option, we can imagine to split the scenario, at least, as follows:



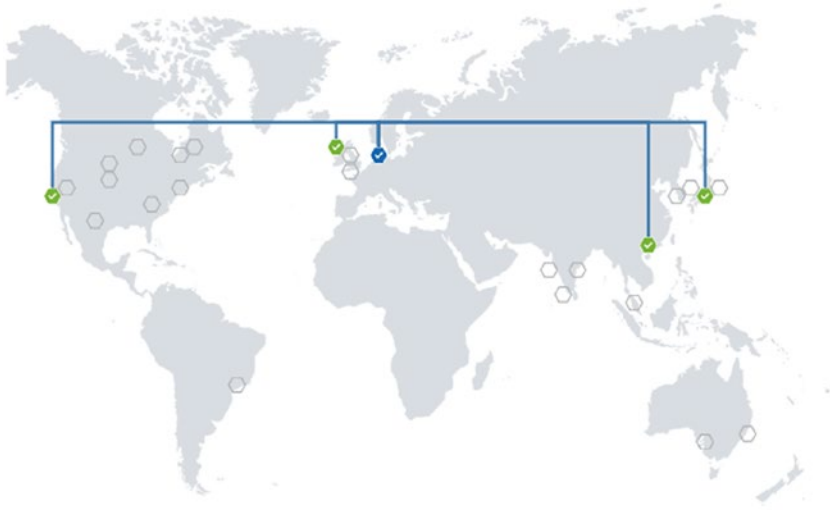
**Figure 1-11.** Since a replica relationship has been established between two SQL Databases, making one the secondary read-only replica of the primary, read-only applications (Reporting tools can fit this requirement) can point to the replica, without affecting the performance of the primary node.

In this second scenario (Figure 1-11) it is clear that analytical processing can, at maximum, consume all the resources of the secondary replica, keeping safe the critical business activities.

This approach can be extended by design in almost every scenario, since the majority of solutions (except ones based on intensive data ingestion) have a high read/write ratio.

## Using Geo-Replication

Geo-Replication (Figure 1-12) is a SQL Database feature which permits to setup, with few clicks, a secondary server in the same or different geographical region, with a synchronized (but read-only) copy of the primary DB. This is a killing feature since it can be enabled on every SQL Database DB and with few configuration steps. The time needed to perform the initial seeding (the process where data is completely aligned the secondary) can vary depending on the service tier of both DBs and the quantity of data involved.



**Figure 1-12.** An overview of a Geo-Replication scenario in place. The blue icon is the primary DB and the green ones are the secondary replicas.

---

As in the migration process, the suggestion is to upgrade the DB to a large tier before setting up the Geo-Replication, in order to reduce the seeding time. Another suggestion is to have the secondary replica at least at the same performance level of the primary. In case it is weaker, primary can experience delays while propagating updates and notifications.





---

Despite Geo-Replication is a topic often discussed as a way to improve business continuity and uptime, we suggest to implement that (if budget permits) in almost every scenario, to ensure we always have a copy to be used in several scenarios:

- Read-only APIs/Services
- Read-intensive Web Applications
- Reading production data with no much attention to the optimization of the query (i.e., from SSMS)
- Testing in production (with the guarantee no write will be accepted)

In SQL Database we can have up to 4 secondary copies (read-only) of a primary (read-write) DB (Figure 1-13). We can span the replication across different region or stay in the base region: in this last scenario we achieve a simple replication instead a real Geo-Replication. All of those combinations are valid:

- Primary DB in West Europe, Secondary 1 in North Europe, Secondary 2 in Japan West
- Primary DB in West Europe, Secondary 1 in West Europe
- Primary DB in North Europe, Secondary 1 in North Europe, Secondary 2 in East Asia
- Primary DB in West Europe, Secondary 1 in East Asia, Secondary 2 in UK West, Secondary 3 in Canada Central, Secondary 4 in West India

PRIMARY			
	West Europe	azure-demos/ApressADW	None
SECONDARIES			
	East Asia	apress-ea/ApressADW	Readable ...
	Japan West	apress-jw/ApressADW	Readable ...
	North Europe	apress-ne/ApressADW	Readable ...
	West US	apress-wu/ApressADW	Readable ...

**Figure 1-13.** In this scenario we have the primary DB in West Europe and 4 additional Readable copies in different regions around the globe.

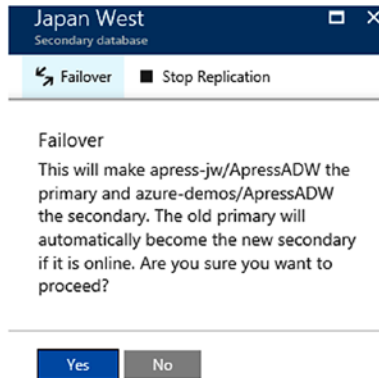
Geo-Replication is not only about splitting read/write queries but, mainly, is about Availability. In fact, each secondary database can be promoted to primary by triggering the manual Failover action, in the portal or through the APIs. To be complete, on a secondary DB we can:

1. Stop replication: the secondary DB will stop the replica relationship and becomes a valid read/write DB.

Please note that once the relationship is broken, it cannot be re-established (except by recreating a new secondary DB from scratch).



2. Failover: with failover, Azure promotes the secondary to primary and makes the “old-primary” as a read-only secondary replica. In addition, it updates the remaining secondary to synchronize with the new primary.



**Figure 1-14.** A Failover action on the Japan West replica. Azure will manage the promotion and the demotion of the respective primary/secondary DBs.

A common pitfall around Geo-Replication is how to deal with Failover (Figure 1-14), where applications that have connection strings pointing to a primary DB must be updated in order to point to the new primary. If there are multiple applications involved, this can be really hard to achieve (in a small timeframe and without downtime).

## Using Failover Groups

By using for a while Geo-Replication comes out that a common pattern is to have just one replica (to separate reads/writes or to increase availability) of a primary DB. Another common requirement is to replicate all the DBs in a server to avoid doing it one-by-one, especially where part of an elastic pool. Finally, companies really don't want to manually switch connection strings in every application during a failover action.

Failover Groups (Figure 1-15) address exactly those requirements. By using the same underlying technology of Geo-Replication, they provide:

- a way to replicate a group of DBs on a source server to a target server (that must be in a different geographical region, however)
- a way to use the same DNS name/endpoint for the primary/secondary: this ensures that applications can be untouched in case we trigger a failover action

SERVER	ROLE	READ/WRITE FAILOVER POLICY	GRACE PERIOD
 azure-demos (West Europe)	Primary	Manual	
 azure-demos-ne (North Europe)	Secondary		

Read/write listener endpoint

apress.database.windows.net

Read-only listener endpoint

apress.secondary.database.windows.net

**Figure 1-15.** In this failover relationship the azure-demos (West Europe) server is Primary and replicates to the azure-demos-ne (North Europe) secondary DBs. To have applications connecting transparently to the Primary/Secondary, we must use the generated DNS names for listeners.

In both Geo-Replication and Failover Groups solutions, we recommend to use Database-level Logins and Firewall Rules, to be sure they are propagated as database object in the various replica, avoiding potential runtime issues.

Please consider that you can't failover a single DB in the group. When the failover action is triggered, all the DBs in the group will fail to the secondary replica. The only way to failover a single DB is to remove it from the Failover Group. By removing it, it still remains in the replication relationship and can failover individually.

## Hot Features

We should avoid to talk directly of specific features of SQL Database, since they are pretty aligned with the latest version of SQL Server (that is, at the time of writing, SQL Server 2016). In fact, almost every enhancement to SQL Server is released to SQL Database on Azure before it is released in GA in the on-premises SQL Server product. This is a great chance to be early adopter of a new technology, since when the features are ready are automatically included in the PaaS with no effort or intervention by the users.

The listing of the hot features of SQL Server and SQL Database is out of the scope of this book, but we mention just a few of them, sure they are relevant to applications and useful to address performance targets.

## In-memory

With the Premium Tier of SQL Database (DBs starting with “P”, like P1, P2, etc) we have a dedicated storage for in-memory OLTP operations (in the 1GB-32GB range, depending on the performance level).

In-memory technology applies to:

- **Tables:** with in-memory tables we can address those scenario with a very fast data ingestion rate. Time for transactional processing is reduced dramatically.
- **Columnstore Indexes:** with Clustered/Non-clustered columnstore indexes we can reduce the index footprint (with great savings on the storage side) and perform analytical queries faster.

## Temporal Tables

While dealing with history data, we often see custom development pattern where, before every write, a record is copied to make the history of changes of it. However, this approach binds every touch point to be aware of this logic, where it should be better that the client applications are unaware of this, with the only purpose to write updates to the given table.

Another solution we have seen is the use of triggers. But, triggers are quite cumbersome and they should be placed for every table we would like to historicise.

Temporal tables are tables defined to integrate an history mechanism, where is the SQL Server engine which provides all the necessary stuff to save the old record values before the update occurs. During the creation of the Temporal Tables (or during the migration of an existing non-temporal one) we must specify the name of the underlying backing table which receives every update, including the schema ones. This powerful mechanism ensures that every operation made to the parent table is propagated to the history table.

In case we are altering an existing table, we should add two new fields, to keep track of temporal information, as follows:

```
ALTER TABLE [SalesLT].[Address]
ADD
```

```
    ValidFrom datetime2 (0) GENERATED ALWAYS AS ROW START HIDDEN
    constraint MT_ValidFrom DEFAULT DATEADD(SECOND, -1,
    SYSUTCDATETIME())
```

```
, ValidTo datetime2 (0) GENERATED ALWAYS AS ROW END HIDDEN
    constraint MT_ValidTo DEFAULT '9999.12.31 23:59:59.99'
, PERIOD FOR SYSTEM_TIME (ValidFrom, ValidTo);
```

And next we define the history table:

```
ALTER TABLE [SalesLT].[Address]
SET (SYSTEM_VERSIONING = ON (HISTORY_TABLE = [SalesLT].[Address_History]));
```

If we setup temporal tables while creating them, it is easier:

```
CREATE TABLE MyTable
(
    --... fields ...
    , [ValidFrom] datetime2 (0) GENERATED ALWAYS AS ROW START
    , [ValidTo] datetime2 (0) GENERATED ALWAYS AS ROW END
    , PERIOD FOR SYSTEM_TIME (ValidFrom, ValidTo)
)
WITH (SYSTEM_VERSIONING = ON (HISTORY_TABLE =
    [SalesLT].[Address_History]));
```

## JSON Support

JSON support extends the language specification of T-SQL by introducing operators and selectors useful to work with JSON data. We can:

Query a SQL table with a JSON field, applying filtering directly on the JSON nodes

Serialize a query result into JSON to let be consumed by others, or to be place into a NoSQL storage

This is a sample query with JSON support:

```
SELECT TOP (2)
    [ProductID] as [ID]
    , [SalesLT].[Product].[Name]
    , [Category].[ProductCategoryID] as [ID]
    , [Category].[Name]
FROM [SalesLT].[Product] JOIN [SalesLT].[ProductCategory] AS [Category]
ON [SalesLT].[Product].[ProductCategoryID]= [Category].
[ProductCategoryID]
FOR JSON AUTO
```

The query above produced a well-formatted JSON document as follows:

```
[{
  "ID": 680,
  "Name": "HL Road Frame - Black, 58",
  "Category": [{
    "ID": 18,
    "Name": "Road Frames"
  }]
}, {
  "ID": 706,
  "Name": "HL Road Frame - Red, 58",
  "Category": [{
    "ID": 18,
    "Name": "Road Frames"
  }]
}]
```

We can toggle the JSON array definition with the `WITHOUT_ARRAY_WRAPPER` option.

## Development Environments

In almost every context there is a practice to replicate the production environment into few isolated dev/test environments. A common topology can be the following:

- **Production:** the live environment with hot code and data. Since this chapter is about SQL Database, we see the production environment as the SQL DB instance of production.
- **Pre-production:** this is usually a pre-roll environment. While the various frontend tiers can be separated from the production environment, the Database can be either separated or the same as production. Often, the preproduction phase is just a last-minute validation of the updates related to the new deployment.

- UAT/Staging: this is usually a completely isolated end-to-end lane, with each single block of the application (including database) replicated.
- Dev/Test: those are usually one or more isolated environment where developers can have their own projection of the entire application stack, for development purposes.

With the cloud, especially with Azure, making and maintaining those environment is very easy. There is no more need to even use local development environments, since both the application and the development VM can be in the cloud.

---

We can also develop locally the code we need but pointing to the remote dependencies (i.e., SQL Database, Storage, Search) if the local-to-cloud connectivity is satisfactory.

---

## Database Copies

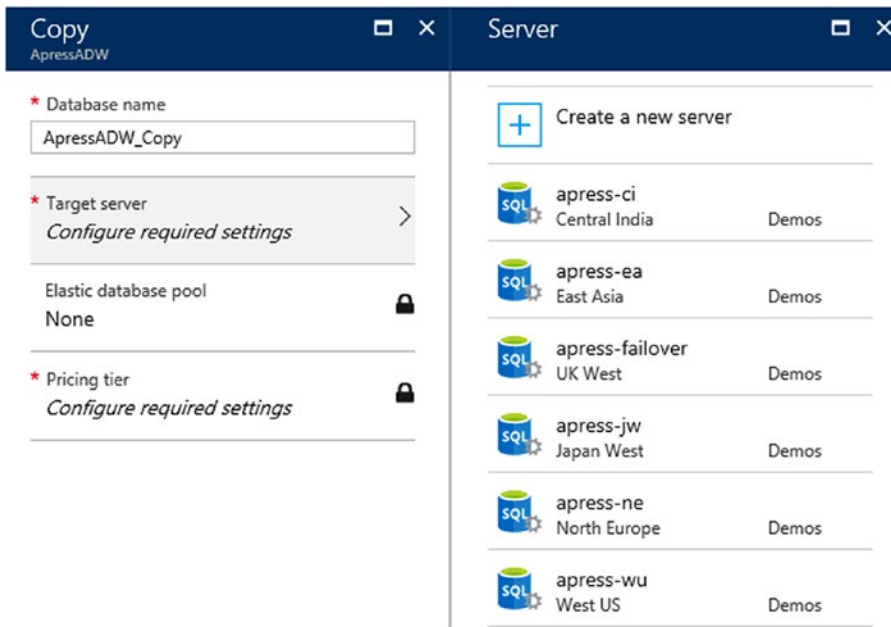
In this scenario, we can take advantage of the SQL Database Copy feature, which creates a copy of the source database instance into a fresh instance, on the same or different SQL Server (logical) container.

The copy feature creates a transactionally consistent copy of the source database into a new database and can be triggered, as well as through the Management API (Portal, PowerShell, SDK) even from inside the SQL Server (logical) container, by launching this query on the master database:

```
CREATE DATABASE AdventureWorksLT _staging AS COPY OF AdventureWorksLT;
```

The copy operation (Figure 1-16) will start and it will take some time (depending on the DB size, the performance level and the actual load of the source DB). With a query pane opened on the “master” database, we can query the status of the copy operation as follows:

```
SELECT * FROM sys.dm_database_copies
```



**Figure 1-16.** Database can be copied also versus a target server in a different datacenter. This feature is extremely useful in scenarios where regional migrations are required.

## Worst Practices

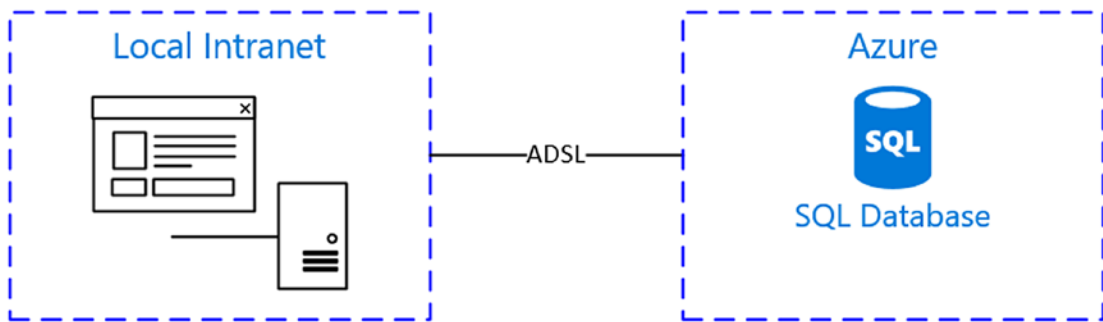
We can say SQL Database is the cloud version of SQL Server, so almost every attention to be paid to the second, can be applied to the first in order to gain performance. Every T-SQL optimization we have learned in the past, can be reused and this is definitely great.

However, SQL quality is an aspect that can be disattended by several projects, for many reasons:

- The attention paid to the software code is much more than the one paid for the SQL tier
- The skills about SQL are less developed than the ones about software development
- Nobody needs (until now) to invest so much on the DB maintenance/optimization

Let us show an example of what can happen while migrating an on-premises SQL instance to the cloud. Let's suppose we have a Windows Client intranet application (Figure 1-17), which lets users perform business activities and which relies on SQL Server hosted in a server room in the company's facility. The software is mainly a data-driven, grid-based application, where users consult data and produce reports.

The Database has only 40 tables, no fancy features involved (CLR, file-system dependencies, etc) and, to over-simplify, no stored procedures, views and other database objects than the tables mentioned. In short, let's take as an example a DB 100% compatible with the SQL Database feature surface area, which we migrate with a small amount of IT effort.



**Figure 1-17.** *In this scenario we must consider the potential bottleneck of Bandwidth between the company's facility and the cloud.*

Unfortunately, once the connection string of the application has changed, users are hanging on basic operations which before they made in seconds, and now take minutes to run. So, where is the issue? Let's make some hypotheses:

- SQL Database is “slower” than the on-premise version: this can be possible, there are many tiers and performance level to test to know if this is a performance issue. We can scale up and look for improvements, and this can tell us which initial size we should use to get acceptable performance.
- The bandwidth between “us” (the office) and the DB in the cloud is insufficient while queries are executed normally
- The software has something to investigate, since the DB/Bandwidth utilization is low

Some common pitfalls while using SQL in software development follow.



## Bad Connection Management

Some of the frameworks discussed below automatically manage all the stuff around connections. It is very common the pattern which, in response to an object `Dispose()`, the corresponding underlying DB connection is closed. Anyway, we can fall in several cases where a bad connection management keeps resource usage very high and introduces a waste of time and money:

### The Driver/Provider to Connect to the DB does not Use Connection Pools

What is a connection pool? When we connect to a DB, basic TCP operations are involved (handshake, authentication, encryption, etc). In case we are connecting from a Web App and, specifically, we have a correlation between user requests/navigation and queries, a high number of concurrent DB connections can be established and consequently, a heavy overhead on both the client machine and the RDBMS server is generated. To avoid this, many frameworks (including ADO.NET) use Connection Pools. A connection pool is a way to manage DB connections, reusing them between different contexts. The advantage is to avoid to establish a fresh connection for each command to execute against the DB; instead, applications using connection pools reuse existing connections (in most cases, transparently).

### The Connection Pools are Fragmented

What is the fragmentation of Connection Pools? Looking inside on how ADO.NET, for instance, manages the pools, we see different behaviours. Each connection pointing to a specific database originates a new connection pool, which means that if the Web App is now connecting to Database A and then to the Database B, two connection pools are allocated. In case we were using on-premise SQL Server, with Windows Authentication or Basic Authentication with Integrated Security login (fortunately not available on SQL Database), a connection pool is generated per-user and per-database.

```
//Request coming from User A on DB A
using (var connection = new SqlConnection("Integrated Security=SSPI;Initial
Catalog=DB_A"))
{
    connection.Open();
    //First connection pool for DB A is created
}
```

```
//Request coming from User B on DB A
using (var connection = new SqlConnection("Integrated Security=SSPI;Initial
Catalog=DB_A"))
{
    connection.Open();
    //Second connection pool for DB A is created
}
```

In case we are working on Sharding data, where a master connection can be established to the Shard Map and specific queries are routed to shards, each single DB is mapped to a single Connection pool. A common solution to this second scenario is to use the same connection (for example to the master DB) and then use the USE keyword to switch database inside the query. Unfortunately, even this scenario (the usage of USE statement) is not supported on SQL Database. We will discuss sharding later but keep in mind those limits and operate consequently.

```
//Usage of USE statement, NOT ALLOWED in SQL Database
var command = new SqlCommand();
command.CommandText = "USE MyShard";
using (SqlConnection connection = new SqlConnection(
    connectionString))
{
    connection.Open();
    command.ExecuteNonQuery();
}
```

As many connections lead to more overhead and latency, it is recommended to reduce fragmentation and achieve optimizations, where possible.

## The Connections are not Disposed Properly

If we do not follow correctly the recommended actions on existing connections (in ADO.NET this can be done by avoiding to call Dispose() on SqlConnection) the connection cannot be released to the Connection Pool and cannot be reused later. Relying just on garbage collector to dispose them indirectly is not a reliable and efficient solution to the problem. Depending on the framework we are using (ADO.NET directly or Entity Framework or others) we must instruct the framework to free to resources we are not using anymore.

## Client-Side Querying

There are frameworks which hide the complexity behind a database request round-trip, which let all the details (i.e., SQL Connection, Commands, Queries and Results) under the hood. Those frameworks/libraries are very useful to speed up the development process. However, if not used appropriately, they can lead to undesired behaviour.

In local-only environment, where the bandwidth is not an issue and, more generally, it isn't a strict constraint, we can fall in the wrong pattern by materializing all the data of a specific dataset to perform the query on the client (and not on the RDBMS server). This can occur with this C# snippet as follows:

```
var ta = new DataSet1TableAdapters.CustomerTableAdapter();
var customers = ta.GetData().Where(p => p.LastName.StartsWith("S"));
```

In the previous example we get ALL THE DATA of the Customer table and we perform the query on the client side. Unfortunately, this worst practice cannot be easily discovered in on-premises scenario, where a very high network bandwidth can give the developers the wrong impression of good performance. In this second case, we make the same mistake using Entity Framework:

```
using (var ctx=new ApressADWEntities())
{
    var customers = ctx.Customer.ToArray().Where(p => p.LastName.
        StartsWith("S"));
}
```

For skilled developers those misusages are clearly wrong, but it worth to remind them in order to make every effort to avoid them in production.

## Pay Attention to Entity Framework

We are estimators of Entity Framework for the great capability it had to reduce the gap between developers and SQL specialists.

In ideal world, SQL-related tasks should fall on a specialized database developer and the software part is up to the software developer. In real world, often developers are in charge to write software AND write queries. In this common situation, it can be faster (not better) to teach developers to use a middleware which translates code into the SQL queries, than teach SQL itself.

Let's take the query above:

```
var customers = ctx.Customer.ToArray()
```

Entity Framework translates this expression into the following SQL query:

```
{SELECT
    [Extent1].[CustomerID] AS [CustomerID],
    [Extent1].[NameStyle] AS [NameStyle],
    [Extent1].[Title] AS [Title],
    [Extent1].[FirstName] AS [FirstName],
    [Extent1].[MiddleName] AS [MiddleName],
    [Extent1].[LastName] AS [LastName],
    [Extent1].[Suffix] AS [Suffix],
    [Extent1].[CompanyName] AS [CompanyName],
    [Extent1].[SalesPerson] AS [SalesPerson],
    [Extent1].[EmailAddress] AS [EmailAddress],
    [Extent1].[Phone] AS [Phone],
    [Extent1].[PasswordHash] AS [PasswordHash],
    [Extent1].[PasswordSalt] AS [PasswordSalt],
    [Extent1].[rowguid] AS [rowguid],
    [Extent1].[ModifiedDate] AS [ModifiedDate]
FROM [SalesLT].[Customer] AS [Extent1]}
```

Which is okay, and it has pros/cons to consider:

As a pro, it considers exactly every table member known at the time of creation of the EF model. This means that if we add new fields without updating the EF mapping, this query continues to work and it fetches only the data we need in the application. In addition, like every query generated by EF, we do not need to use strings to pass queries to the DB, which is definitely one of the best advantages we have by using EF.

As a con, we can obtain the same result by writing the following statement:

```
SELECT * FROM SalesLT.Customer
```

Which reduce the incoming bytes to the SQL instance. However, note that, in case of fields added, they will be fetched also and maybe they are useless for the calling application.

Those examples are trivial, but think about querying complex tables with multiple joins and complex filtering logic. Entity Framework can be the best of allies, but sometimes can even generate a lot of query code which humans can definitely write better (from a performance perspective).

```
var query = ctx.SalesOrderHeader
    .Where(p => p.SalesOrderDetail.Any(q => q.Product.ProductCategory.Name.
        StartsWith("A")))
    .Where(p => p.SubTotal > 10 && p.Address.City == "Milan")
    .Select(p => new
    {
        Order=p,
        Customer=p.Customer,
        Details=p.SalesOrderDetail.Select(q=>new
        {
            Item=q.Product.Name,
            Quantity=q.OrderQty
        })
    });
```

This query hides the complexity of multiple joins, advanced filtering and multiple projection from different tables. It is clear that it can save a lot of time for non-SQL specialist, but keep in mind that the generated SQL query is something like that:

```
{SELECT
    [Project2].[AddressID] AS [AddressID],
    [Project2].[SalesOrderID] AS [SalesOrderID],
    [Project2].[RevisionNumber] AS [RevisionNumber],
    [Project2].[OrderDate] AS [OrderDate],
    ... 30 lines omitted ...
    [Project2].[PasswordHash] AS [PasswordHash],
    [Project2].[PasswordSalt] AS [PasswordSalt],
    [Project2].[rowguid1] AS [rowguid1],
    [Project2].[ModifiedDate1] AS [ModifiedDate1],
    [Project2].[C1] AS [C1],
    [Project2].[ProductID] AS [ProductID],
    [Project2].[Name] AS [Name],
```

```

[Project2].[OrderQty] AS [OrderQty]
FROM ( SELECT
    [Extent1].[SalesOrderID] AS [SalesOrderID],
    [Extent1].[RevisionNumber] AS [RevisionNumber],
    [Extent1].[OrderDate] AS [OrderDate],
... 30 lines omitted ...
    [Extent3].[Phone] AS [Phone],
    [Extent3].[PasswordHash] AS [PasswordHash],
    [Extent3].[PasswordSalt] AS [PasswordSalt],
    [Extent3].[rowguid] AS [rowguid1],
    [Extent3].[ModifiedDate] AS [ModifiedDate1],
    [Join3].[OrderQty] AS [OrderQty],
    [Join3].[ProductID1] AS [ProductID],
    [Join3].[Name] AS [Name],
    CASE WHEN ([Join3].[OrderQty] IS NULL) THEN CAST(NULL AS int) ELSE
    1 END AS [C1]
FROM    [SalesLT].[SalesOrderHeader] AS [Extent1]
INNER JOIN [SalesLT].[Address] AS [Extent2] ON [Extent1].
[BillToAddressID] = [Extent2].[AddressID]
INNER JOIN [SalesLT].[Customer] AS [Extent3] ON [Extent1].
[CustomerID] = [Extent3].[CustomerID]
LEFT OUTER JOIN (SELECT [Extent4].[SalesOrderID] AS
[SalesOrderID], [Extent4].[OrderQty] AS [OrderQty], [Extent4].
[ProductID] AS [ProductID1], [Extent5].[Name] AS [Name]
    FROM [SalesLT].[SalesOrderDetail] AS [Extent4]
    INNER JOIN [SalesLT].[Product] AS [Extent5] ON [Extent4].
    [ProductID] = [Extent5].[ProductID] ) AS [Join3] ON [Extent1].
    [SalesOrderID] = [Join3].[SalesOrderID]
WHERE ( EXISTS (SELECT
    1 AS [C1]
    FROM [SalesLT].[SalesOrderDetail] AS [Extent6]
    INNER JOIN [SalesLT].[Product] AS [Extent7] ON [Extent6].
    [ProductID] = [Extent7].[ProductID]
    INNER JOIN [SalesLT].[ProductCategory] AS [Extent8] ON
    [Extent7].[ProductCategoryID] = [Extent8].[ProductCategoryID]

```

```

WHERE ([Extent1].[SalesOrderID] = [Extent6].[SalesOrderID]) AND
      ([Extent8].[Name] LIKE N'A%')
)) AND ([Extent1].[SubTotal] > cast(10 as decimal(18))) AND
      (N'Milan' = [Extent2].[City])
) AS [Project2]
ORDER BY [Project2].[AddressID] ASC, [Project2].[SalesOrderID] ASC,
[Project2].[CustomerID1] ASC, [Project2].[C1] ASC}

```

We are not saying it is a wrong query nor a wrong approach; we just need to keep in mind that is just ONE solution to the problem and it may not be the best one.

## Batching Operations

An example of abusing Entity Framework can be its usage applied to bulk inserts. Think about this code:

```

using (var ctx = new ApressADWEntities())
{
    for (int i = 0; i < 10000; i++)
    {
        ctx.Customer.Add(new Customer()
        {
            CompanyName = $"Company {i}",
            //Missing other properties assignment
        });
    }
    ctx.SaveChanges();
}

```

On `SaveChanges`, Entity Framework spans a new INSERT statement for each record we created in the code. This is actually correct, from the EF side, but maybe it's not what we would like to have. Instead, we should focus on some sort of Bulk Insert, using the low-level API of ADO.NET or other commercial frameworks which add performance-related features on top of EF.

Some strategies to batch the operations against SQL Database can be:

- **Buffering:** when possible, decoupling the data producer from the data writer with a buffer (even a remote Queue) can avoid bottlenecks on the SQL side and it avoids the need of batching at all.
- **Transactions:** grouping several modify operations in a single transaction, as opposed to executing those same operations as distinct (implicit) transactions, results in optimized transaction-log operations improving performance.
- **Table-valued parameters:** in case we are grouping a sequence of INSERT operations, we can use user-defined table types as parameters in T-SQL statements. We can send multiple rows as a single table-valued parameter.

---

For further information about table-valued parameters follow this link: <https://docs.microsoft.com/en-us/sql/relational-databases/tables/use-table-valued-parameters-database-engine>

---

- **SQL Bulk Copy / BCP / Bulk Insert:** it is probably the best option for bulk INSERT operations.

---

Do not think that parallelize operations can always be faster while performing operations against the DB. If we are splitting 1000 operation of a single batch in 4 threads of 250 operations each, it is not guaranteed we notice a save. Indeed, we often observe a degradation of the overall performance, since there are many factors which influences the scenario.

---

## Scaling SQL Database

Scaling a RDBMS is a well-known challenging topic, since few RDBMS can scale horizontally while keeping various limits. Replicas are usually read-only and designed for high-availability and often, the only way to increase the performance of a DB, is to scale up the underlying instance.



Increasing the hardware on the underlying infrastructure is similar to increase the DTU level of the SQLDB: both approaches give to the DB engine more resources to perform queries. In this scenario, options are not on the DB side.

We encourage users to not rely only on changing the Performance Level, since they are fixed in numbers and limited in power (at the time of writing, 4000DTU is the maximum supported). What we should think about is, again, the application architecture.

Let's assuming the following as the evolving path of the DB of a SaaS solution:

1. We develop a single, multi-tenant DB
2. We keep it at Standard S0 (10 DTU) level during the development phase
3. After an hour in production with a single client, we see the DTU is always at 100%, so we increase first at S1 (20 DTU) level.
4. By monitoring the performance, we now see that the DTU is still at 100% but with few moments at 60-70%. We increase the level at S2 (50 DTU).
5. Now the DB is at 60% on average, which can be okay

Now we can realize that DTU consumed are too much for the budget. In that case we can optimize the existing application:

- By using more efficiently the DB resources
- By offload some of the SQL load to other storage types (Polyglot Persistence)
- By introduce layers of caching for frequently accessed data
- By sharding data

If the DTU consumed is aligned with the expectations (or, if not aligned, at least in budget) we can proceed with the evolution:

1. 10 new clients arrive. Since the overload of the DB is a 10% more for each new client, we double the DTU scaling to a S3 (100 DTU) level.
2. Now the DB has reached the maximum level of the Standard Tier, so we need to pay attention to the consequences of a further level increase.

- 3. A new client, which predicted consumption is of about 45 DTU by itself, subscribed for the service, we have two options:
- 4. Increase the Standard Tier to Premium. DTU would pass from 100 to 125 (in the Premium P1 plan) but price increases about 3 times.
- 5. Use the last multi-tenancy technique and create a new S2 (50 DTU) database instance, pointing the new client’s application to this DB.

**Table 1-1.** *We have two database, designed to be multi-tenant individually, with a groups of tenants each*

Database Name	# of clients	Tier	Level	Avg. Consumption
POOL001	11	Standard	S3	60% (60/100 DTU)
POOL002	1	Standard	S2	90% (45/50 DTU)

Now we have implemented the “Multiple Logical Pools with a single schema preference” which results in the best tradeoff between elasticity and governance (just one schema is applied to both DBs) but some drawbacks:

- More management effort:
  - while applying changes to the DBs
  - during backup/restore operations
  - on the entire monitoring process
- Different costs per-client to control and summarize

Please note that some of the drawbacks are have also positive impact to other aspect of the Service Management. Let’s suppose you have a client who unintentionally deletes its data from the database. If that client is in a dedicated DB, a simple backup restore can help to get the whole DB at the previous state.

Now think about the same situation in a shared DB: how can we get just the data of the given client without restoring the whole DB? In fact, the restore operation would affect even other tenants, so manual work (or scripted procedures) are needed to perform it at best.

Many SaaS providers offer different price tags for services. Can happen that the underlying infrastructure affects the price and “premium” services are often used to provide clients with dedicated resources and value-added services. One example can be the capability to restore the whole database at any given time.

---

## Managing Elasticity at Runtime

In the ideal scenario, an application would span a query between several DBs to boost performance and distribute the load. However, doing this in real world is quite hard and involves a series of hot topics, related to sharding data.

Let’s assume we are working on a multi-tenant solution where each tenant has, in each table, its own unique identifier (i.e., TenantID field). We would need, at least:

- A shard map: a dictionary where, given the key of the query (i.e., TenantID) we know which actual database to point to
- A movement tool: a middleware to organize and move (split and merge) data between sharded databases
- A query tool/library: an artifact which hides the complexity of the shards to the applications, performing the routing of the queries and, in case of queries against multiple DBs, which performs the individual queries and merge results. In this category falls optionally a transaction manager which runs the transactions to multiple DBs

As we can imagine, this can be made 99% by custom software and custom tooling, while Microsoft provides its own support with the Elastic Database Tools.

## Elastic Database Tools

Microsoft realized that it is not easy for every developer/ISV to implement by their own a fully-featured set of tools/libraries to deal with sharding. At the same time, it has been proven that sharding is the most efficient way to implement scale out on relational DBs. This spun out a set of technologies which help us to manage even complex sharding scenarios with hundreds of database involved.

The Elastic Database Tools are composed of the following:

- Elastic Database Client Library: a library which helps creating the shard map manager (a dedicated DB as the index of our shards) and the various individual shards databases.

---

An example of how to use this library is available here: <https://code.msdn.microsoft.com/windowsapps/Elastic-Scale-with-Azure-a80d8dc6>

While an overview of the library is here: <https://docs.microsoft.com/en-us/azure/sql-database/sql-database-elastic-scale-introduction>

---

- Elastic Database split/merge tool: a pre-configured Cloud Service with a Web Role and a Worker Role which presents a GUI and the routines to perform split/merge activities on the shards. We must notice that Cloud Services are actually not implemented in ARM (Azure Resource Manager) and we do not cover them in this book.
- Elastic Database Jobs: a pre-configured set of services (Cloud Service, SQL Database, Service Bus and Storage) with the necessary running software needed to run jobs against multiple databases.
- Elastic Database Query: a specific feature (in preview) of SQL Database which permits to connect/query to make cross-database queries.
- Elastic Transactions: a library which helps creating a client-coordinated transaction between different SQL Databases. At the time being, there is no server-side transaction support.

Keep in mind that tools above are just provided as individual tools and they are not full PaaS as SQLDB itself. Except the Elastic Database Query, which is a feature of SQLDB, implement the Split/Merge tool, for instance, means to take the ownership of new cloud resources, to provision, monitor and manage.

## Pooling Different DBs Under the Same Price Cap

I would suggest to design applications like mentioned in the previous section, to be elastic and resilient by design. However, let's suppose that moving tenants is too hard or too expensive and the best solution is to have "One Database for each tenant". In that case, we can easily grow up to hundreds of DBs as far as clients arrive.

Think about the situation in the table below:

Database Name	# of clients	Tier	Level	Cost	DTU	Peak DTU	Avg. Usage
DB001	1	Standard	S0	~ 15\$/m	10	8	40%
DB002	1	Standard	S1	~ 30\$/m	20	11	25%
DB003	1	Standard	S1	~ 30\$/m	20	13	40%
DB004	1	Standard	S2	~ 75\$/m	50	30	20%
DB005	1	Standard	S3	~ 150\$/m	100	65	10%
DB006	1	Standard	S3	~ 150\$/m	100	70	10%
DB007	1	Standard	S0	~ 15\$/m	10	5	20%
DB008	1	Standard	S1	~ 30\$/m	20	13	40%

We see that, with 8 clients, we have 8 DBs each one with its own Performance Level, calibrated on the peak DTU usage we need. The monthly cost will be around 495\$/month.

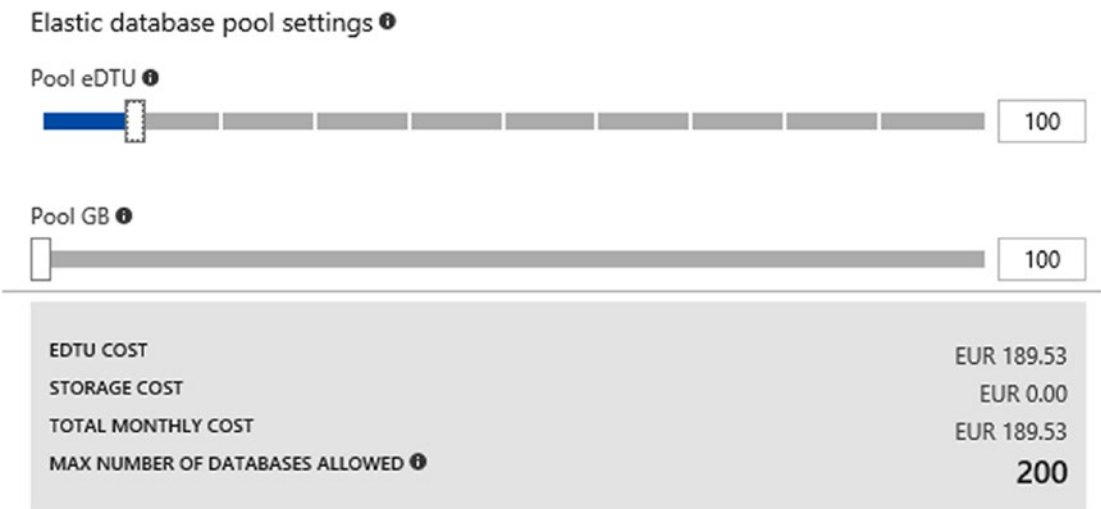
Unfortunately, it is a waste of resources: at one side, we need to size the DB based on the Peak we expect. At the other side, we see that average usage (especially for the most expensive DBs) is very low.

From the numbers above we can infer an average global DTU usage of about 57 DTU. In the optimal (and unrealistic) case tenants have peaks during different timeframes, we can even use a single DB of 100 DTU (Standard S3) containing every tenant (but this is against the requirements pointed at the beginning of the "Scaling SQL Database" section).

## SQL Database Elastic Pools

For the emerging requirement shown above, Azure has SQLDB Elastic Pools (Figure 1-18). An Elastic Pool is a logical container of a subset of the DBs of a server (which is the logical container of our DBs). We can create a Pool by specifying:

- Server: DB pools only apply to a subset of DBs of the same logical server. We cannot span between different servers.
- Pricing Tier: each DB pool has its own price, as for the standalone DBs
- Configuration: we can set how many DTU has the pool, how much capacity (in terms of GBs) and min/max DTU for each database contained



**Figure 1-18.** In this configuration we have a pool with a total of 100 DTU and 100GB for the whole set of contained DBs. The cost is approximately ~224\$/month, which is far less compared to the previous option.

We can notice, however that only 100 DTU for DBs having peaks of 65-70 DTU can be too small. At any given time, we can increase the cap of the entire pool without touching the individual DBs.

## Scaling Up

We left intentionally this option as the last of the entire section because we think it is the last resort. Don't misunderstand, scale up is good and it is part of the process, but since we cannot scale up indefinitely, we should start thinking about performance issues in time.

At the same time, we don't recommend to over-engineer a simple solution by adding shards, pools, caching layers, etc. We must know them in advance and, possibly, develop our code accordingly. Crafting the software with those ideas will reduce consumption of the resources from the beginning and solutions consuming 1000 DTU can easily be reduced to a 100 DTU impact.

Anyway, scaling up is the shortest path to gain power immediately, for example if we want to manage an unpredictable peak, or in case of planned increase of load. This is the table of most of the current levels of Tiers/DTU, they can change in time, but we strongly recommend to not design a solution which relies on the top tier, since there is no way to scale more!

Tier	Level	DTUs	Tier	Level	DTUs
Basic	B	5	Premium	P1	125
Standard	S0	10		P2	250
	S1	20		P4	500
	S2	50		P6	1000
	S3	100		P11	1750
Premium RS	PRS1	125		P15	4000
	PRS2	250			
	PRS4	500			
	PRS6	1000			

Offloading the reading operations there is a good alternative to scale up where the scenario permits it. By using the Geo-Replication feature, which creates a read-only copy (always synchronized) of the primary database in an alternative location. Applications can discriminate between reading operations and “everything else”, routing the reading operations to the secondary read-only node, keeping the primary just for the core updates. Considered that, at the time of writing, we can have up to 4 copies of a primary database, this can be very useful to distribute the read traffic between replicas, keeping the primary free from the majority (where applicable) of load.

---

## Governing SQL Database

Before letting it run in production, there are few basic actions to be performed onto a SQL Database instance. First, we should define the security boundaries and, more important, we should identify security hot spots. Second, we need to prepare the monitoring activity before the service goes live, otherwise there is a serious risk of loss of control. Third, we should plan every action related to disaster recovery and backup.

## Security Options

When a SQL Database has been created, it resides inside the logical and security boundary of the SQL Server (logical) container. Every SQLDB runs inside that container and a single container can contain several DBs.

---

There is a maximum number of DBs we can put inside a SQL Server (logical) container but, since this number can change over time, think differently. A SQL Server (logical) container, when created, shows a maximum number of DTUs which can be placed inside this. This should be a better indicator of how many DBs (depending on their size) can be placed.

---

Apart the security features which SQL Database inherits from T-SQL and SQL Server, there are some specific, value-added services of SQL Database itself.



## Authentication

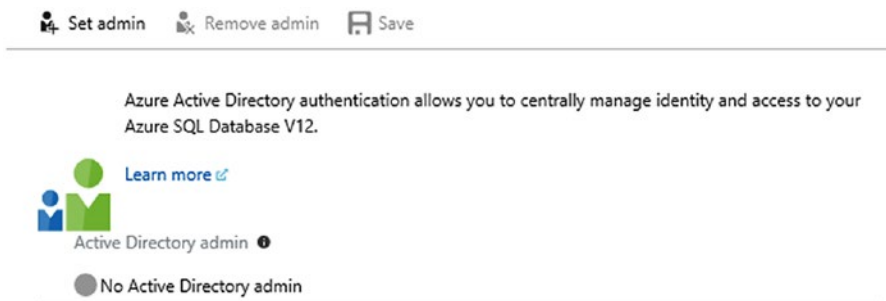
When we create a new SQL Database, we must place it into a new or existing SQL Server (logical) container. If a new container is created, we must set the administrative credentials of the entire container. Username and Password specified here will grant a full-control permission set to the entire server and the database contained in it.

So, it is not recommended to use those credentials while connecting to individual DBs. Instead, the best practice is to create specific users at the database-level (they are called “Contained Users”). This approach makes the database even more portable, since in case of a copy, the copy operation keeps all the database objects (including logins) that otherwise will be lost if defined at the server-level.

```
CREATE USER containedUser WITH PASSWORD = 'myPassword';
```

This method is known as SQL Authentication, which is very similar to the SQL Server counterpart.

However, SQL Database supports also the Azure AD Authentication (Figure 1-19), which binds Azure Active Directory to the SQL Server (logical) instance. To enable this method, we should set first the Active Directory admin on the server blade:



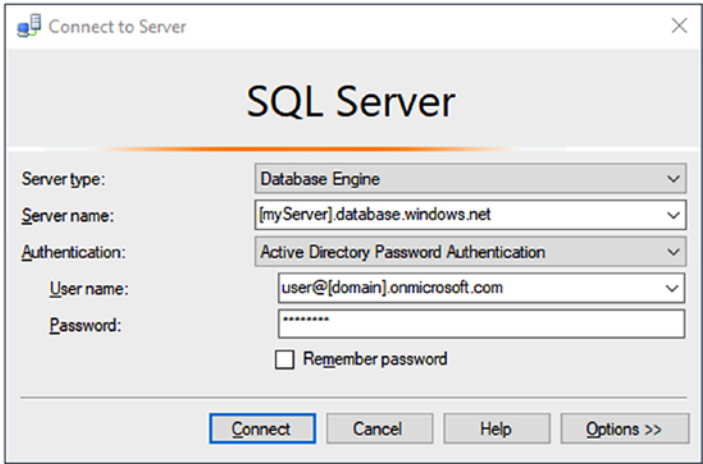
**Figure 1-19.** The page where we can setup the Azure Active Directory admin for a given SQL Server (logical) instance.

This will create a USER in the master DB with “FROM EXTERNAL PROVIDER” option. In fact, we can create additional contained users as follows:

```
CREATE USER <myUser@domain> FROM EXTERNAL PROVIDER;
```

From the client perspective, when we connect to the SQL Database using Azure AD Password authentication, the connection string should be similar as this one below:

```
Data Source; Authentication=Active Directory Password; Initial Catalog=apress; UID=user@[domain].onmicrosoft.com; PWD=[password];
```



**Figure 1-20.** This is how we connect to SQLDB using SSMS and Azure AD Password Authentication

If we scroll down the Authentication dropdown in the window above (Figure 1-20), we can notice other two options:

- Active Directory Integrated Authentication: another non-interactive authentication method to be used where the client PC is joined to a domain federated with the Azure AD tenant.
- Active Directory Universal Authentication: an interactive, token-based authentication where even complex MFA (Multi-Factor Authentication) workflows are available.

## Firewall

SQL Database, through the SQL Server (logical) container is exposed on the public internet with a public DNS name like [myServer].database.windows.net. This means everyone can potentially access the instance to (try to) login into the DB and operate remotely. Thus, it is very important to take a look as soon as possible to the firewall rules (Figure 1-21). By default, no one can access to it, but we should ensure to enable only the required IPs.

Firewall rules and in the IP Range form. This means a rule can be as follows:

RULE NAME	START IP	END IP	
<input type="text"/>	<input type="text"/>	<input type="text"/>	...
All	0.0.0.0	255.255.255.255	...

**Figure 1-21.** The list of server-level firewall rules. The rule above, for example, opens the firewall for every public IP.

Firewall rules can be set at server-level or at database-level. The order of evaluation of those rules are:

1. First the Database-level rules
2. Then the Server-level rules

Since the rules are only in the form allow (everything is not explicitly allowed is denied by default), this guarantees the server-level rules are broader and win against the database-level ones. This should suggest us to make use of database-level rules first to setup a fine-grained set of access rules.

Database-level firewall rules can be configured only using T-SQL as follows:

```
EXECUTE sp_set_database_firewall_rule N'NWRule', '0.0.0.0', '1.0.0.0';
```

A summary (Figure 1-22) of the firewall rules can be queried as follows:

- `SELECT * FROM sys.firewall_rules` - at server-level
- `SELECT * FROM sys.database_firewall_rules` - at database-level

	id	name	start_ip_address	end_ip_address
1	2	All	0.0.0.0	255.255.255.255
2	1	AllowAllWindowsAzureIps	0.0.0.0	0.0.0.0

**Figure 1-22.** This is the result of the `sys.firewall_rules` query, where the `AllowAllWindowsAzureIps` rule is a special rule allowing every Microsoft Azure IP range to enabled inter-service communication.

## Encryption

There are two ways (not mutually exclusive) to approach database encryption:

- Encryption at rest: the underlying storage is encrypted.
- Encryption in transit: data from/to the DB travels already encrypted.

Those two methods address the following scenarios:

- Someone has physical access to the storage media where the DB are stored: mitigated with Encryption at rest, the person can obtain the physical media but he/she cannot read it.
- Someone is intercepting the traffic between the client application and the DB: mitigated with Encryption in transit, the person can sniff the traffic but he/she sees only encrypted data.

## Transparent Data Encryption

SQL Database offers the TDE (Transparent Data Encryption) to address the first case. A server-generated certificate (rotated automatically and without administrative hassle at least each 90 days) is used to encrypt/decrypt the data.

To enable it on a specific DB, use this query:

```
ALTER DATABASE [myDB] SET ENCRYPTION ON;
```

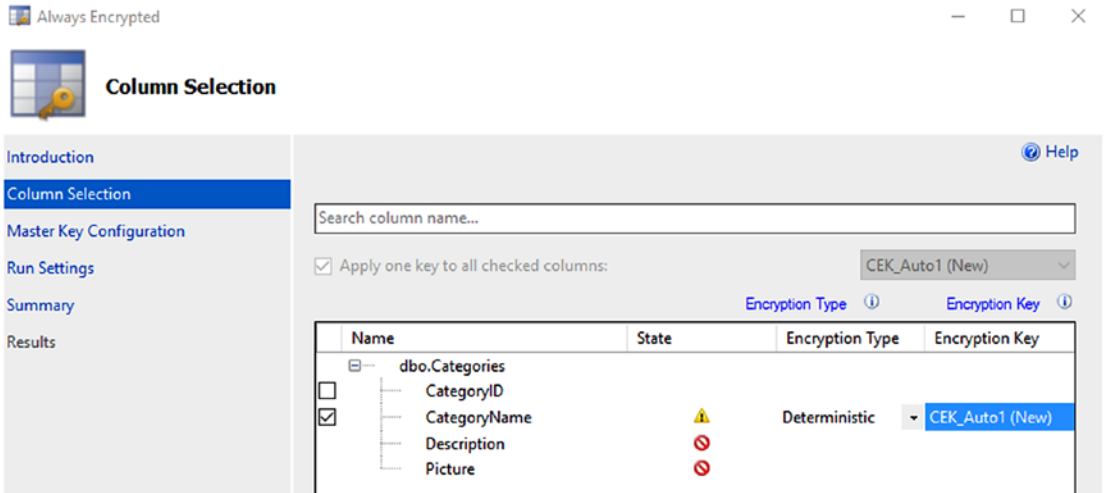
Every new SQL Database has this option enabled by default. Since we have evidences that the overhead introduced by TDE is minimal, it is recommended to enable it (or leave it enabled) a fortiori if we are subjected to compliance requirement.

## Always Encrypted

Always Encrypted is a way to encrypt SQL Database content without ever disclosing the key to SQL Database itself. This approach is the best we can achieve in terms of segregation, since the manager (Azure) cannot read the owner's data.

This approach is more complicated, since SQL Database will deal with encrypted data to be stored, indexed and queried.

Encryption is at column-level, so we can encrypt just a few columns with sensitive data, leaving untouched the rest of the DB. In the screenshot below (Figure 1-23), we are encrypting the `CategoryName` of the `Customer` table, specifying `Deterministic` as the encryption method.



**Figure 1-23.** In SSMS, we can right-click the table or the column and select *Encrypt Column(s)* to start the Wizard process

**Note** Deterministic encryption means that the same source value will generate the same encrypted value. Randomized, instead, will produce different outputs. The first is simpler, but someone can analyze patterns and discover information where data assume a small set of distinct values. The second is less predictable, but prevents SQL Database from performing searching, grouping, indexing and joining. Even with Deterministic encryption there are some missing capabilities, for example the usage of `LIKE` operator, `CASE` construct and string concatenation.

In the next step we provide the Encryption Key, which can reside in the Windows Certificate Store of the client computer (during the wizard will be auto-generated) or into Azure Key Vault. In both cases, SQL Database won't know the key content, since it is managed securely. Also, remember that the encryption process is performed of course by the client machine.

After the column has been encrypted, it has been modified as follows:

```
[CategoryName] [nvarchar](15) COLLATE Latin1_General_BIN2 ENCRYPTED WITH
(COLUMN_ENCRYPTION_KEY = [CEK_Auto1], ENCRYPTION_TYPE = Deterministic,
ALGORITHM = 'AEAD_AES_256_CBC_HMAC_SHA_256') NOT NULL
```

From this moment, every client connecting to the Database will see encrypted data (Figure 1-24), except the ones using the special parameter “column encryption setting=enabled” in the connection string (and, obviously, having the valid key to decrypt data). Since data types change, application would probably fail if not designed to accommodate those changes gracefully.

	CategoryID	CategoryName	Description
1	1	0x0190032E542220E4C220131E3B0D8FD2159092DB97B671F6...	Soft drinks, coffees, teas, beers, and ales
2	2	0x01FE377E4400B422AD75421FC08A712F8503E2FC1B081D5B...	Sweet and savory sauces, relishes, spreads, and ...
3	3	0x0101EF806BE4B158529ADF14BEF181F8C3479778DBC6AD57...	Desserts, candies, and sweet breads
4	4	0x015AB3407CCAC265AB4DB6A3BC7A99CBCE5AD3E9181D69...	Cheeses
5	5	0x017244C0D091AFC9042B1F1E180588E5275164332FD10B611...	Breads, crackers, pasta, and cereal
6	6	0x01CDBBC7EC9530EFA651EE361C052A11C63FA201DFCF752...	Prepared meats
7	7	0x0102372A930BD229CADD4169ECA86E8EE71646BC6470D7...	Dried fruit and bean curd
8	8	0x01443F71F0005E7BCD08C06CF6BD64AA5CEF094A6AAACE42...	Seaweed and fish

Figure 1-24. We see the encrypted data into the CategoryName column

## Dynamic Data Masking

If we ever think about the possibility to give access to production database to a developer to investigate a really hard issue in the application/data, we probably run into the even bigger issue of security. Can we grant a (even temporary) access to a specific user, without exposing sensitive data? And, more generally, can we setup users who can fetch the whole data and others who can fetch masked data?

Dynamic Data Masking works by setting up one or more masking rule for each column we would like to mask (Figure 1-25). The rules are simple:

- Administrators and specific users (specified in configuration) will always see unmasked data
- All the other users will see masked data

- For each column we would like to mask, we add a masking rule, specifying:
  - The table and the column
  - The masking format

Add Discard Delete

Mask name

dbo\_Customers\_ContactName

Select what to mask

Schema

dbo

Table

Customers

Column

ContactName (nvarchar)

Select how to mask

Masking field format

Default value (0, xxxx, 01-01-1900)

Default value (0, xxxx, 01-01-1900)

Credit card value (xxxx-xxxx-xxxx-1234)

Email (aXXX@XXX.com)

Number (random number range)

Custom string (prefix [padding] suffix)

**Figure 1-25.** in this example, we apply a Default value (for strings is a sequence of “x”) to the ContactName of the Customers table

## Backup options

Every SQL Database have built-in mechanism which backups the database continuously, in order to provide the Point-in-time-Restore feature. Depending on the Tier we choose we can go in the past up to 35 days to restore a copy of the DB in a specific point of time.

The restore process will restore a fully functional online database that can be used after the restore process is finished. This feature provides us application-level recovery, letting us recover a DB to copy lost data or to investigate a previous version. In the rare case we want to switch the recovered DB onto the production DB, we can rename them through SSMS:

- Rename the production DB “myDB” into something like “myDB\_old”: after that, all connection are lost and your connected systems will be down.

- Rename the recovered DB from “myDB\_[date]” to “myDB”: after that (it takes just few seconds in most cases) existing applications will find again the DB and continue to work.

For whom need to have older backups (after 35 days in the past) Azure provides some other options (Figure 1-26). We can manually export a DB into a BACPAC (by choosing “Export” in the figure below) or we can setup a Long-term backup retention policy.



**Figure 1-26.** By clicking *Export* we setup an *Export job*, creating a *BACPAC* of the database at current state

---

**Note** In the rare case e accidentally delete a DB we want to keep, we can restore it immediately through the Deleted databases blade of the SQL Server (logical) container.

Finally, the Export feature is the common way too to restore locally a database, as the last resort to DR mitigation.

---

## Long-term Retention

Long-term backup retention allows to save backups to a Recovery Services vault to extend the 35 days window of integrated point-in-time backup/restore policy.

We should use long-term as a secondary strategy to backup SQL Database where compliance requirements must be addresses. From the costs perspective, while the integrated backup/restore mechanism is included in the cost of SQL Database, long-term retention is billed through the Recovery Service vault (Figure 1-28), which billing strategy is (at the time of writing) based on storage consumption (Figure 1-27), except some free quota:



Backup Usage	
Cloud - LRS	0 B
Cloud - GRS	36.77 GB

**Figure 1-27.** This is the tile of the Azure Recovery Services vault blade where we see the GBs consumed by the long-term backup service

Restore

myDB

\* Database name

myDB\_2017-08-03T07-46Z ✓

Point-in-time

Long-term

\* Azure vault backups

Select a backup >

\* Target server

myServer West Europe 🔒

Elastic database pool

None >

\* Pricing tier

PremiumRS PRS1: 125 DTU, 500... >

Subscription

Enterprise Subscription - Prod 🔒

Azure vault backups

myDB

CREATION TIME

31/07/2017 17:19:22

28/07/2017 23:18:31

**Figure 1-28.** In this image we are restoring the myDB from a Recovery Service vault using one of the recovery points on the right

## Monitoring Options

It's proven that Platform-as-a-Service (managed) services need a fraction of the administrative effort compared to unmanaged services, where everything from the operating system upward needs to be constantly monitored, updated and fixed. However, the residual effort we need to invest on the monitoring area is crucial to have

real-time insights on the components we use, especially since managed services are presented as black boxes.

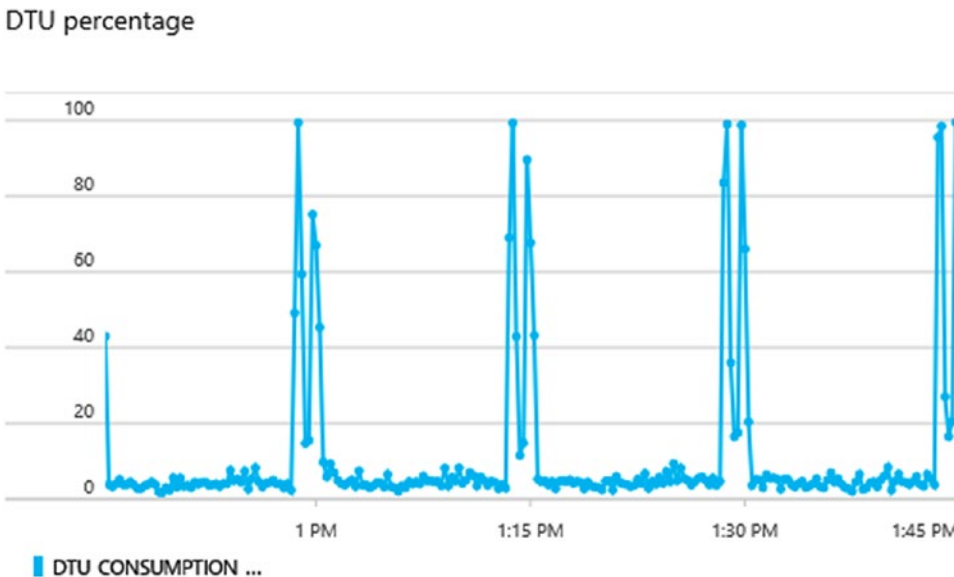
To summarize which the areas we must pay attention to with SQL Database, think about the following:

- Resources monitoring, usage and limits
- Troubleshooting features
- Anomalies/security detection

SQL Database provides most of those value-added features, but it is up to us to enable them and implement proper processes to get the most out of them.

## Resources Monitoring, Usage and Limits

One of the most important KPI of consumption is the DTU percentage (Figure 1-29). First, because if 100% is reached, new connections will be probably throttled due to service saturation. Second, because it tells us the usage pattern of our database and it can provide great value to understand the impact from the applications and, consequently, the monitoring/optimization activities to perform.

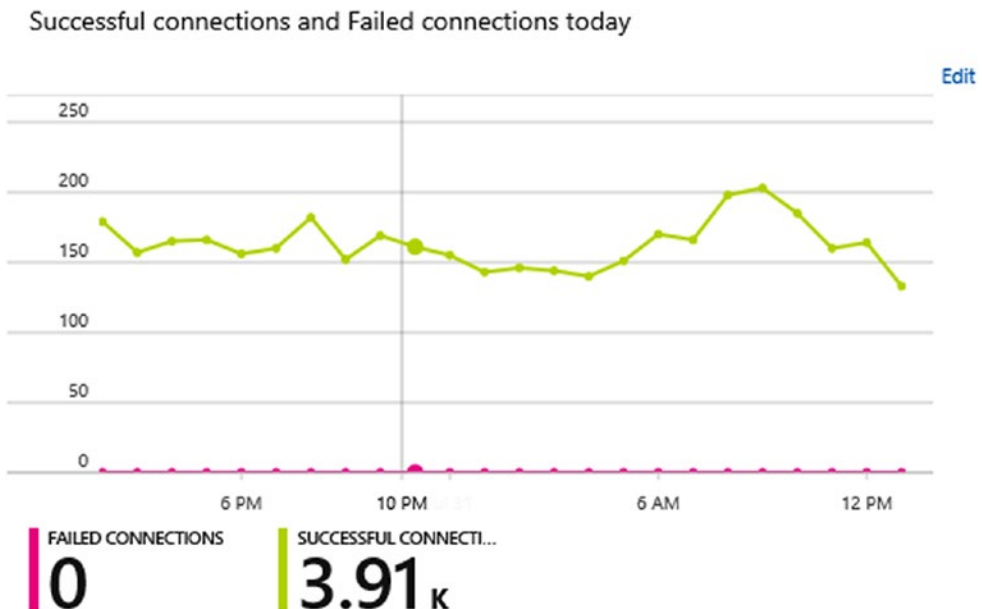


**Figure 1-29.** In this figure we can infer there is a recurrent peak in the DB DTU usage, about every 15 minutes. During this short timeframe we notice how the consumed DTU touch the 100% of usage

The image above tells us the consumption pattern of the last hour for a given Database. In this precise case, we can make some hypothesis:

- During the most of the time, DTUs are about at the 5% of consumption
- If the graph is linear and 5% is without peaks, we could even lower the Performance Level of the DB to a 1/15 tier (if it is a S3-100 DTU, we could even set it to a S0-10DTU).
- However, there are recurrent peaks about every 15 minutes, due to some scheduled/recurrent activities against it from outside (applications, scheduled queries, etc.). Since the usage in those timeframes is very high and completes relatively quickly, it risky to lower the Performance Level because the DB could take more to perform those actions.
- A good option is to investigate which is the application generating those traffic and try to optimize it in order to avoid those burst, to consequently lower the Performance Level with more confidence.

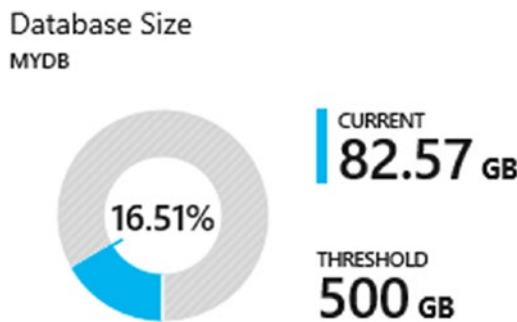
This is just an example, since every DB could have a very different usage pattern.



**Figure 1-30.** In this other image, we see connections against the DB of the last 24 hours

In the image above (Figure 1-30), instead, we can have a quick look at the actual state of connections made to the Database. This number is not particular useful itself, since SQL Database has limits on “concurrent” connections. However, we can infer from the line graph there is an average of connections for any given time of about 150-200 connections, that is enough to estimate the Performance Level we should set to avoid throttling.

At the opposite, we see there are no Failed Connections in the last 24 hours, that is good to understand how many times applications were refused to connect.



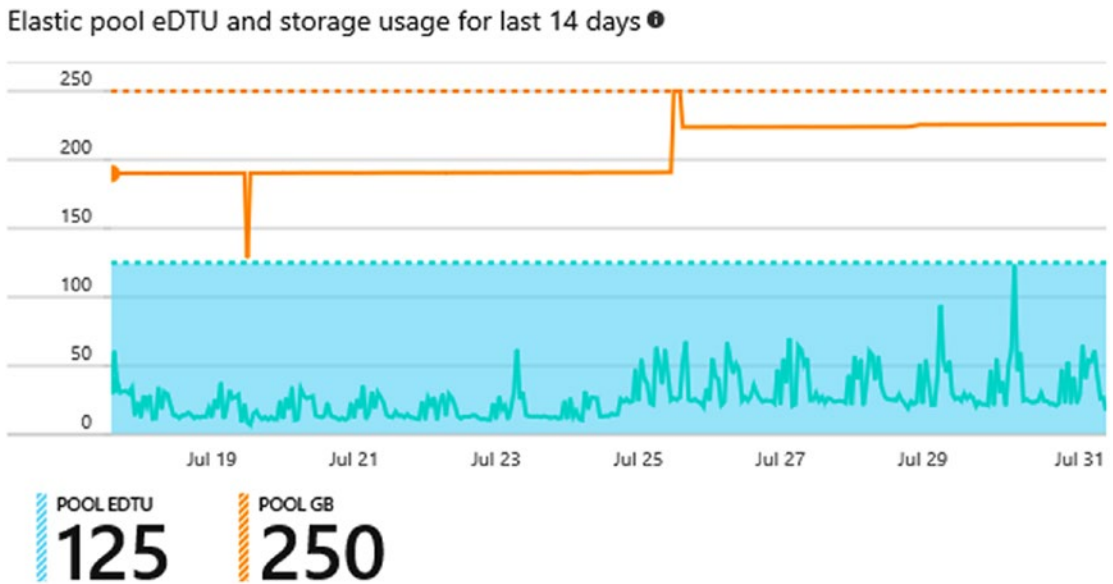
**Figure 1-31.** An indicator of the actual storage used and the threshold set for the current DB

In the image above (Figure 1-31), there is the last of the most important indicators we should monitor. We should expect storage is managed by the platform, preventing us to put effort on administrative task to extend and maintain storage, and that’s true. However, there are some hard-limits in SQL Database around storage and, in case those limits are exceeded, DB become unstable and no more writes are allowed.

Of course, in some cases we can Scale Up and provide a greater Performance Level which takes more storage with it. But there are limits too, and it must be constantly monitored.

## SQL Database Elastic Pools

An additional layer of attention must be paid with Elastic Pools, since the service type has a cap on maximum DTUs and Storage shared by all the databases inside the pool. Thus, if we place databases inside a pool, we must ensure there enough space and computation power.



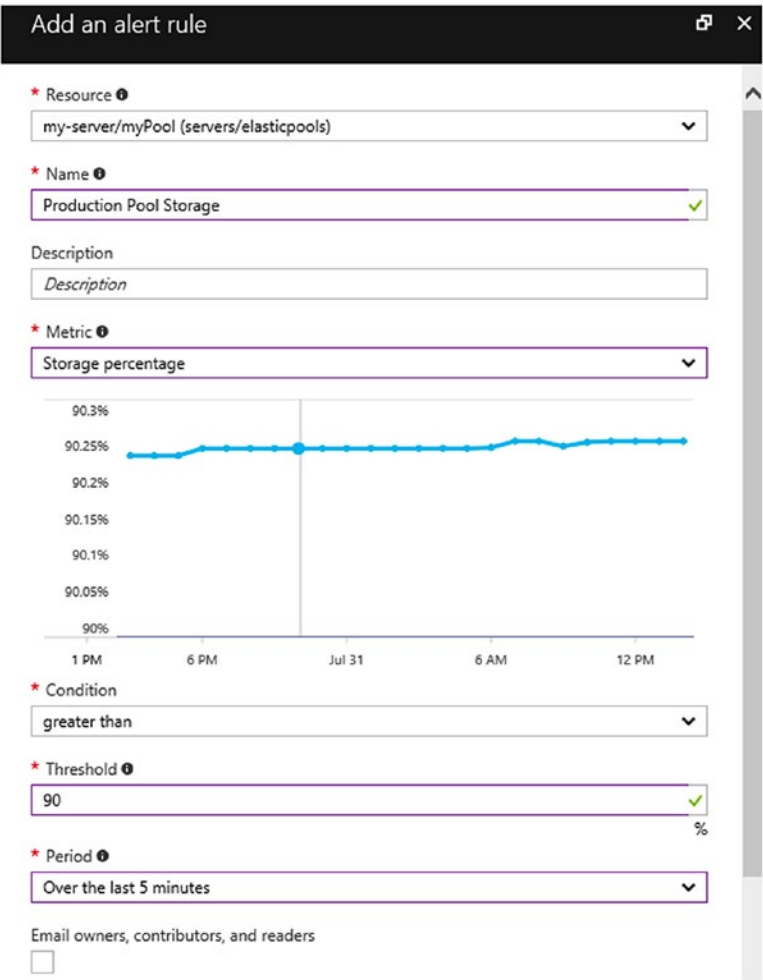
**Figure 1-32.** In this image we see a combined view of DTU and Storage consumption for an Elastic Pool

In the image above (Figure 1-32), we can notice a potential issue. We see, despite DTU consumption is always stable, there is a peak in Storage consumed in the middle of the timeframe. Under those scenario, every DB inside the Pool had certainly stopped to accept writes, with serious consequences on applications and availability.

Even in big applications, the growth rate of standalone DBs is quite predictable. What we need to pay attention to in Elastic Pools, instead, is the fact we can add/remove at runtime a 300GB database in few seconds, filling all the available space of the pool and, consequently, generating serious issues.

## Troubleshooting Features

Too many automated alerts can create false alarms but it is important to setup proper automated alerts on every critical resource. An example can be exactly the situation above, where the Storage used of an Elastic Pool reaches the maximum level: we definitely don't want to be notified by the users, instead we would like to proactively take the actions to avoid failures and availability gaps.



**Figure 1-33.** In this image we setup an Alert rule for the Elastic Pool. In case the pool Storage used percentage goes over 90%, an alert is activated and an email is sent to specified emails

The general rule of thumb is that almost every metric collected by Azure can be used to setup an Alert on it (Figure 1-33). Each service comes with its own metrics (in case of SQL Database we have DTUs, Storage, Connections, etc) and those metrics can be attached to alerts. Therefore, we should setup proper alerts for every critical building block of our infrastructure.

## Dynamic Management Views

The metrics exposed in the portal are available directly on the SQL Database instance, through plain T-SQL queries. This approach is recommended when we need to build custom tools and/or catch KPIs without passing from the Azure Portal. An example is the “sys.dm\_db\_resource\_stats” view, as follows:

```
SELECT TOP (10) [end_time]
               ,[avg_cpu_percent]
               ,[avg_data_io_percent]
               ,[avg_log_write_percent]
               ,[avg_memory_usage_percent]
               ,[xtp_storage_percent]
               ,[max_worker_percent]
               ,[max_session_percent]
               ,[dtu_limit]
               ,[avg_login_rate_percent]
FROM [sys].[dm_db_resource_stats]
```

Which produces the last 10 statistics aggregates (they are ordered using the sampling date, descending) below:

end_time	avg_cpu_percent	avg_data_io_percent	avg_log_write_percent	avg_memory_usage_percent	xtp_storage_percent	max_worker_percent	max_session_percent	dtu_limit	avg_login_rate_percent
2017-07-31 12:48:35.067	3.63	0.14	0.04	58.64	0.00	1.00	0.11	125	NULL
2017-07-31 12:48:20.033	3.89	0.33	0.09	58.57	0.00	1.50	0.11	125	NULL
2017-07-31 12:48:05.003	4.10	0.08	0.11	58.41	0.00	1.50	0.11	125	NULL
2017-07-31 12:47:49.987	4.68	0.10	0.16	58.35	0.00	1.00	0.12	125	NULL
2017-07-31 12:47:34.940	6.86	0.61	0.12	58.25	0.00	1.50	0.11	125	NULL
2017-07-31 12:47:19.907	6.82	0.47	0.14	57.83	0.00	1.50	0.10	125	NULL
2017-07-31 12:47:04.880	5.16	0.14	0.12	57.63	0.00	1.00	0.10	125	NULL
2017-07-31 12:46:49.860	7.33	0.35	0.22	57.53	0.00	1.50	0.10	125	NULL
2017-07-31 12:46:34.810	3.16	0.21	0.06	57.17	0.00	1.00	0.09	125	NULL
2017-07-31 12:46:19.797	7.24	0.07	0.15	57.06	0.00	1.50	0.09	125	NULL



Another useful DMV is about sessions, where knowing WHO is connecting to the Database is a valuable information to troubleshoot problematic queries:

```
SELECT TOP 10 * FROM [sys].[dm_exec_sessions]
```

This view produces the following output (just some columns):

session_id	host_name	program_name	login_name	cpu_time	memory_usage	
58	RD000D3A12C460	Application1	app1User	0	6	
104	RD000D3A12C460	Application1	app1User	0	6	
105	RD000D3A12C460	Application2	app2User	0	6	
107	RD0003FF71C768	Application2	app2User	0	6	
113	RD000D3A12C460	Application2	app2User	0	6	
114	RD000D3A12B52E	ExternalApp	extUser		32	7
115	DESKTOP-LOCAL	SSMS - Query	adminUser	0	3	
117	DESKTOP-LOCAL	SSMS - Query	adminUser	0	3	
119	RD000D3A12C460	ExternalApp	extUser		0	6
124	RD000D3A12C460	ExternalApp	extUser		0	6

This view is incredibly interesting from the troubleshooting perspective. We see at least:

- The Remote Machine name: 3 unique Azure-hosted machines plus the local DESKTOP machine
- The Application Name: it is strongly recommended to pass the application name in the connection string while connecting to SQL Database, in order to propagate the info here
- The Login Name: useful to know which identities are connecting to the DB

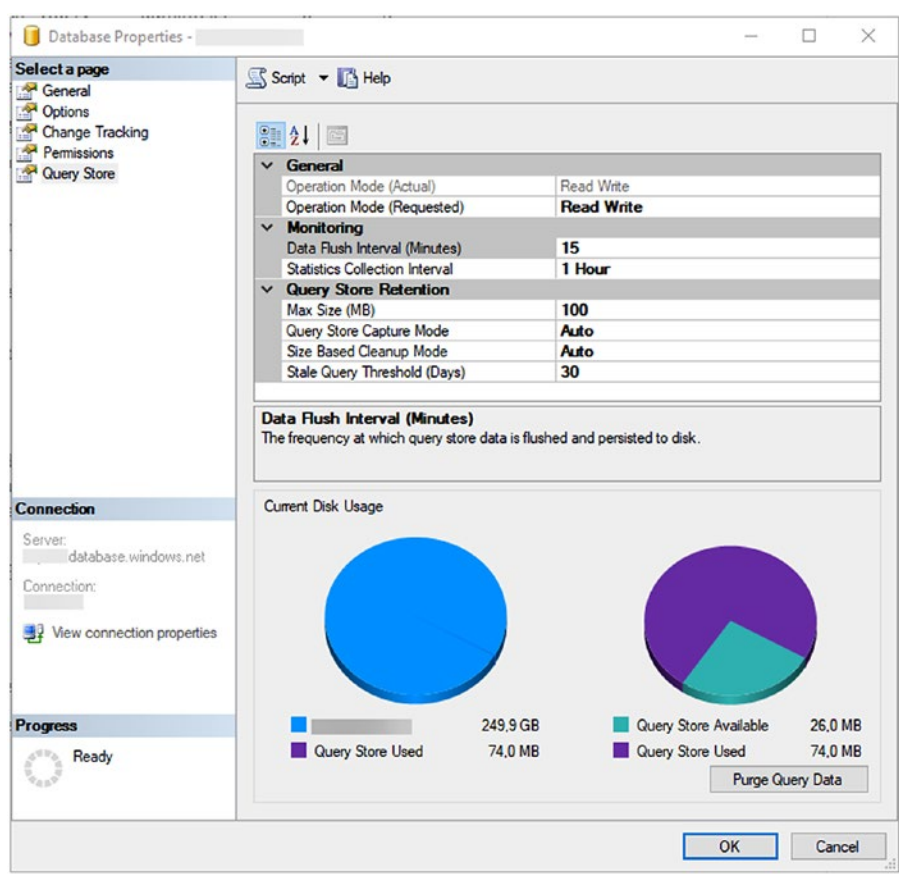
There are a lot of DMVs in SQL Database and they enable advanced monitoring scenarios. SQL Server experts can already be familiar with some of those views and it is an excessive advanced topic to be covered in this book.

## Query Performance Insight

Recently in SQL Server was added the Query Store features, that is a sort of flight data recorder of every query passing through the Database. This feature is now enabled by default as mentioned here (<https://docs.microsoft.com/en-us/azure/sql-database/sql-database-operate-query-store>) and can be enabled on existing databases through the following query:

```
ALTER DATABASE [myDB] SET QUERY_STORE = ON;
```

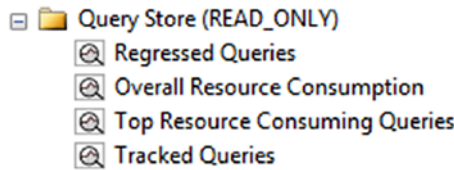
For whom already has query store enabled by default and they want to know the actual parameters of it, we can right-click the database in SSMS and select Properties:



**Figure 1-34.** This options window let us configure Query Store parameters

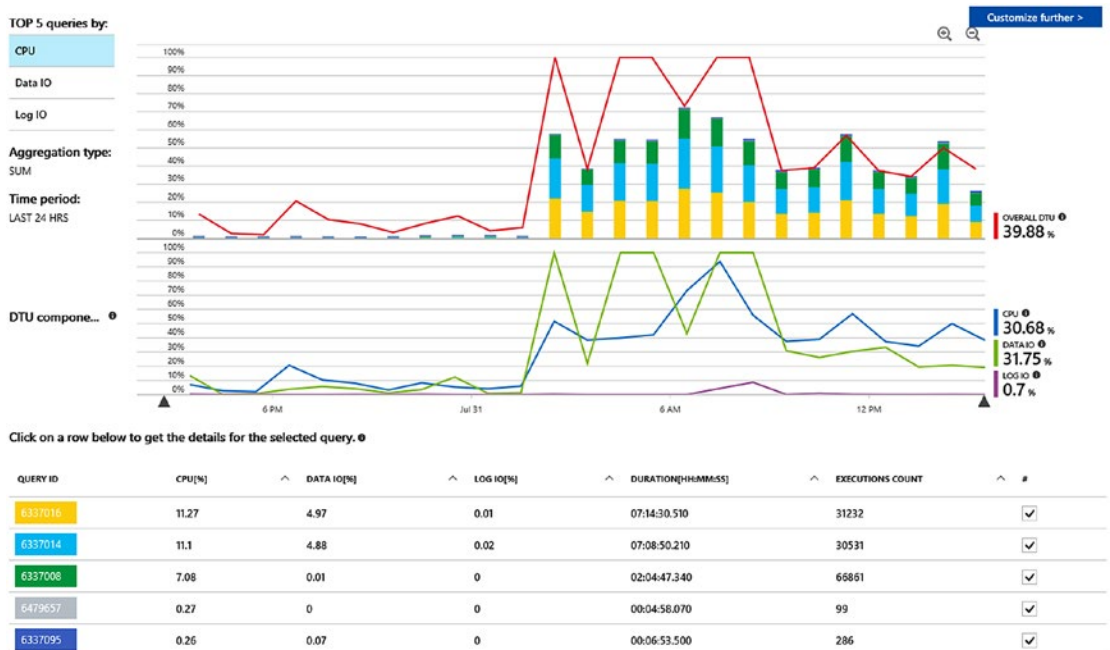
In the figure above (Figure 1-34) we can fine tune the Query Store service, specifying retention and collection options. This can focus the big picture of Query Store (Figure 1-35) before using it through SSMS or Query Performance Insight.

**Note** under certain circumstances, Query Store stops to collect data if the space is full. We can notice this state from the portal. By using the Query Store options we can either change limits or, through T-SQL, clearing the current data.



**Figure 1-35.** The Query Store node in SQL Server Management Studio

Query Performance Insight is an online tool to catch the most out of Query Store. It highlights the most consuming queries and provides relevant information to identify them to proceed with optimization:



**Figure 1-36.** Query Performance Insight showing top 5 consuming queries

In the figure above (Figure 1-36), we can drill-down in the first row to see which is the query that consumes more. Once identified the query text, we can go upward to the applications and perform further optimization.

## Anomalies/Security Detection

As part of every monitoring/management tasks, we should put in place some techniques to prevent security issues or, in case they verify, some logging to inspect and troubleshoot. SQL Database integrates an Auditing feature that collect every event coming into the SQLDB instance and ships it to a remote Storage Account for further analysis.

This feature is useful to the users to investigate problems, to re-build a complex workflow and to have a complete and detailed log of all the operations passing through the database. However, it is useful for Azure too, since Azure itself uses Auditing (if the Threat Detection feature is enabled) to perform real-time proactive detection of potential threats occurring on the DB instance (for example a brute force attack).

## Database Auditing

Database Auditing, as mentioned above, is a feature that collects Extended Events occurring on SQL Database for further analysis.

---

For a reference of what an extended event is and how they are implemented in SQL Database, compared to SQL Server, follow these links:

<https://docs.microsoft.com/en-us/sql/relational-databases/extended-events/extended-events>

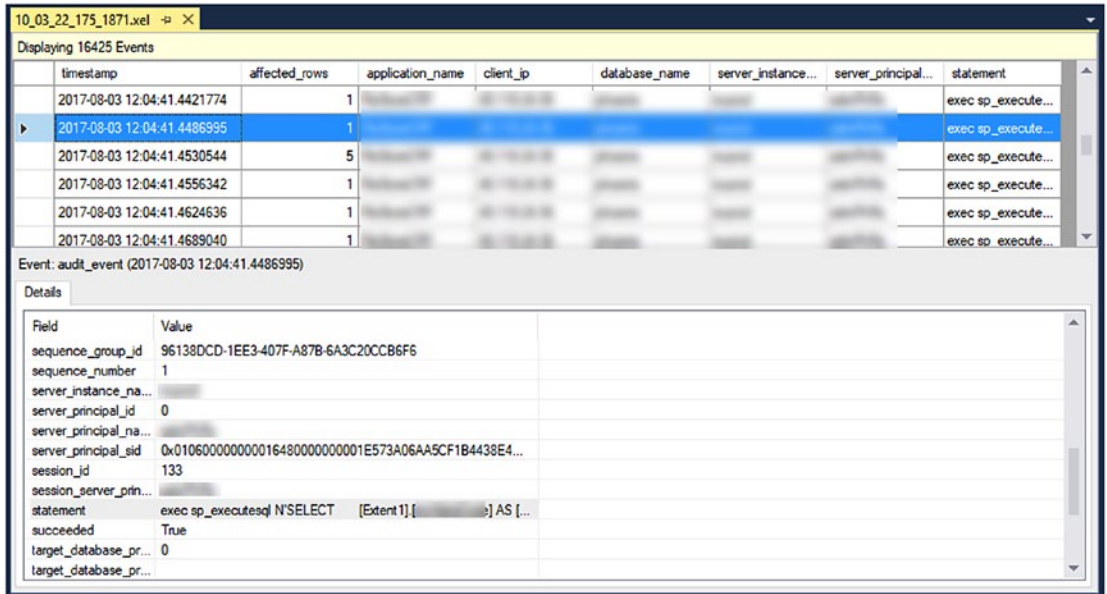
<https://docs.microsoft.com/en-us/azure/sql-database/sql-database-xevent-db-diff-from-svr>

---

After enabling the feature, SQL Database begins to collect .XEL files into the blob storage account specified using this pattern:

```
https://[account].blob.core.windows.net/sqldbauditlogs/[server]/[db]/  
SqlDbAuditing_ServerAudit/YYYY-MM-DD/hh_mm_ss_XXX_YYYY.xel
```

XEL files archived with Auditing can be downloaded from the Blob Storage and then parsed with SSMS, with the following experience:



timestamp	affected_rows	application_name	client_ip	database_name	server_instance...	server_principal...	statement
2017-08-03 12:04:41.4421774	1						exec sp_execute...
2017-08-03 12:04:41.4486995	1						exec sp_execute...
2017-08-03 12:04:41.4530544	5						exec sp_execute...
2017-08-03 12:04:41.4556342	1						exec sp_execute...
2017-08-03 12:04:41.4624636	1						exec sp_execute...
2017-08-03 12:04:41.4689040	1						exec sp_execute...

Field	Value
sequence_group_id	96138DCD-1EE3-407F-A87B-6A3C20CCB6F6
sequence_number	1
server_instance_na...	
server_principal_id	0
server_principal_na...	
server_principal_sid	0x010600000000016480000000001E573A06AA5CF1B4438E4...
session_id	133
session_server_prin...	
statement	exec sp_executesql N'SELECT [Extent1].[...] AS [...]
succeeded	True
target_database_pr...	0
target_database_pr...	

**Figure 1-37.** This is how we can read auditing data form inside SSMS

In the figure above (Figure 1-37) we can have a look of the experience of reading auditing data from within SSMS.

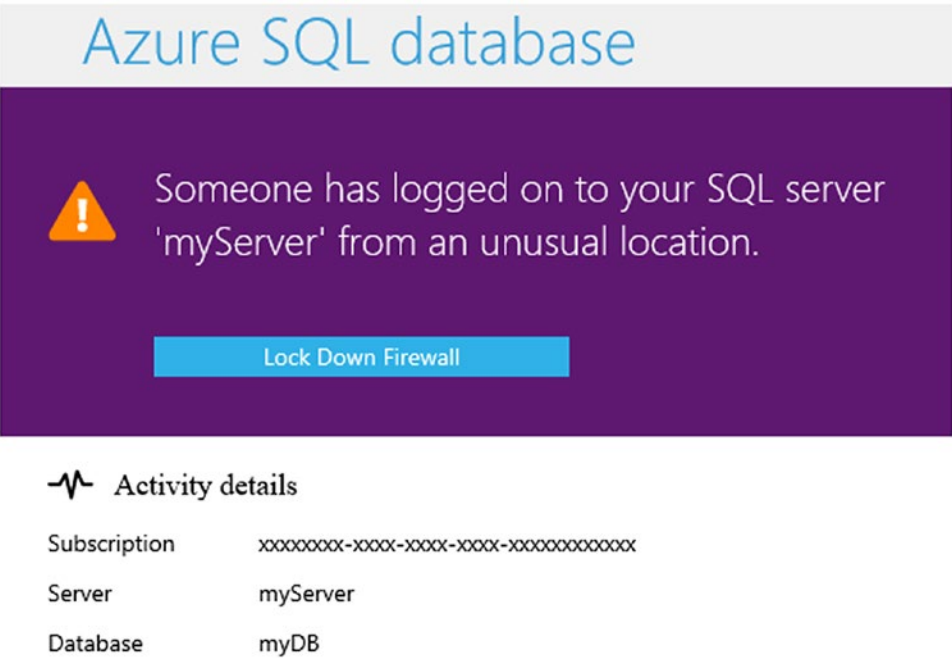
Please note we also have the complete statement executed against SQL Database, comprehensive of sensitive data. Thus, with auditing, keep in mind to protect adequately the Storage Account where the auditing is shipped, since it will contain a huge, despite it is unaggregated, of sensitive data.

Auditing can occur at server-level or at database-level. In the first case, every DB in the SQL Server (logical) instance will inherit the setting and will audit to storage. In the latter, we can fine tune this setting for a single DB.

## Threat Detection

With a simple ON/OFF toggle, we can tell Azure to use our auditing data to perform intelligent analysis and detection of issues (Figure 1-38), like:

- SQL Injections
- Brute force attacks
- Unusual outbound data flow



**Figure 1-38.** This is a sample email that has been sent from the Threat Detection service

## MySQL and PostgreSQL

The concept behind SQL Database is powerful: use a SQL Server-like Database without any effort to administer its underlying infrastructure and with a lot of value-added services to increase productivity and competition.

In the last years, if someone needed another RDBMS, like MySQL, the only choices were:

- IaaS: building your own Virtual Machine and install/configure/manage MySQL for its entire lifecycle
- Marketplace: buying an existing third-party service offering MySQL as managed service, regardless its underlying infrastructure.

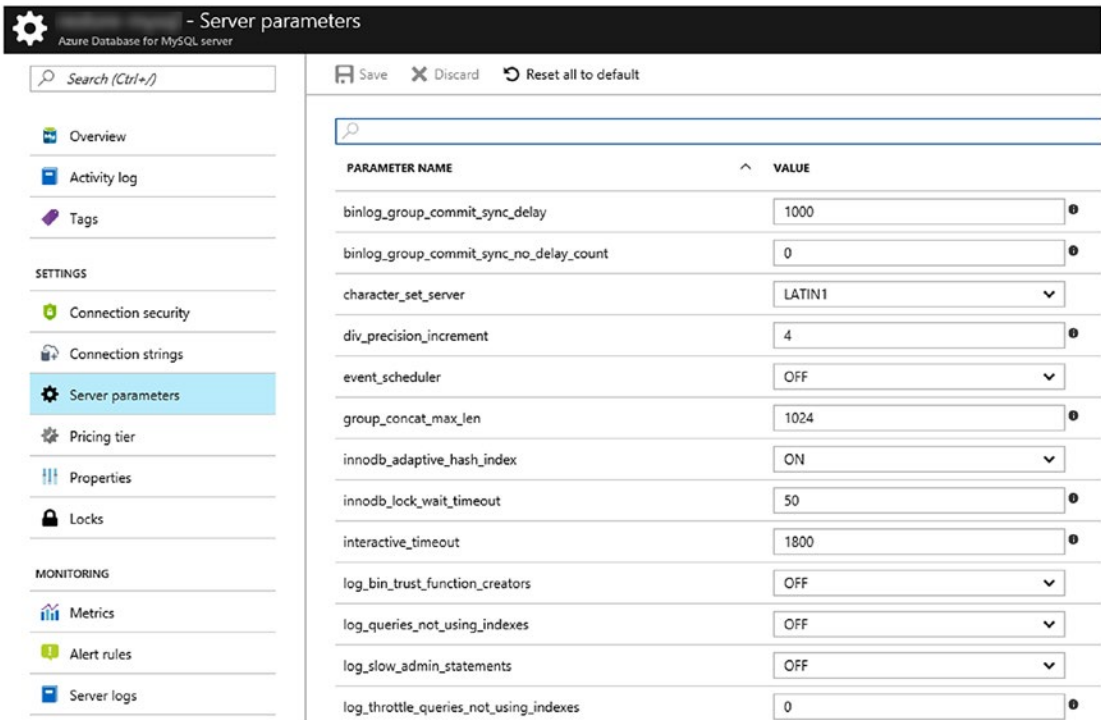
To be clear, those choices are still valid and good, but Microsoft released (now in preview) an Azure Database Service for MySQL and for PostgreSQL, offering a valid alternative to the previous options.

## MySQL

Since the vision around Database Services is to provide, regardless the underlying provider, a foundation of services and features in similar, we can expect from MySQL the same high-level features we have with SQL Database. In theory, this is true, but the service is still in preview and (at the time of writing) has limited features.

By the way, a good approach can be to highlight some similarities:

- We create a server to contain one or more database
- Server has firewall rules and encrypted security
- Only a portion of the entire MySQL engine is available, like in SQL Database there are some limitations too. In MySQL, only the InnoDB engine is supported on two versions (5.6.35 and 5.7.17, Community Edition)
- Upgrades (minor patching) is managed by the platform
- There are pricing tiers based on the performance delivered and the storage allocated
- There is the point-in-time restore feature

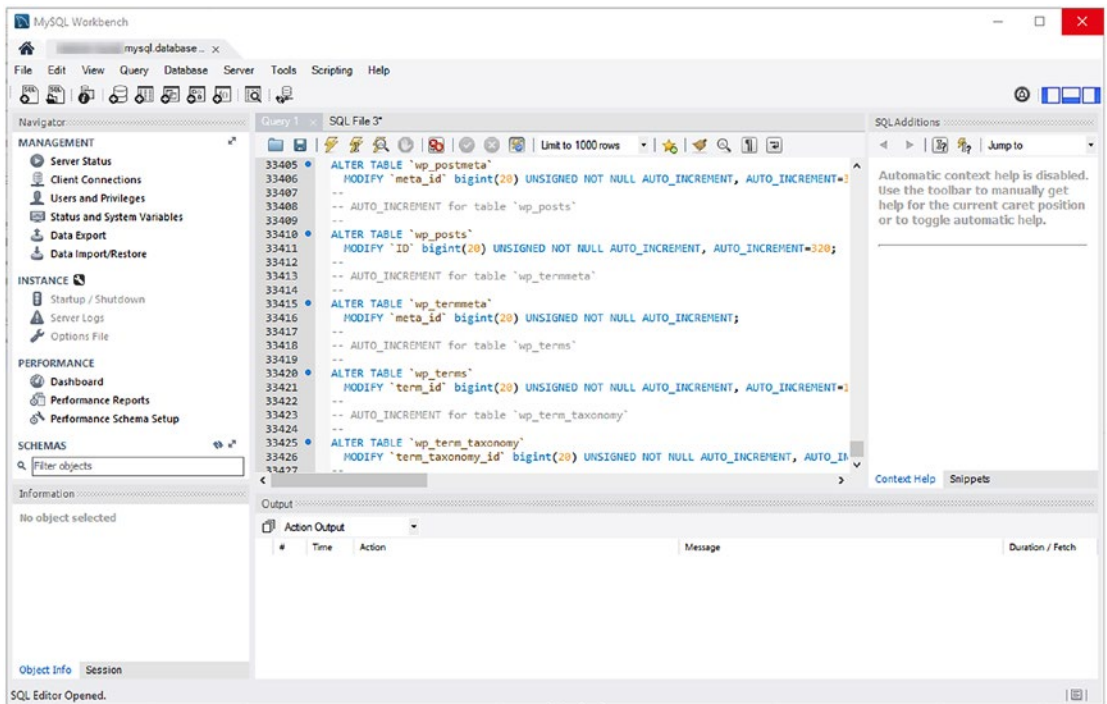


**Figure 1-39.** This blade provides the configuration or server-level parameters on the Azure Database Service for MySQL

And some differences:

- The server is not logical but it hides a real underlying dedicated resource, making it a billable resource itself. In case of MySQL, in fact, pricing tiers are per server and not per database.
- We can explicitly exclude SSL endpoint running on a dedicated port
- The concept of DTU here is called CU (Compute Units)
- The backup/restore operates at server-level
- We can set the MySQL server parameters through the Portal (Figure 1-39)





**Figure 1-40.** This is MySQL Workbench, one of the most relevant administration tool for MySQL in the market

In the figure above (Figure 1-40), we see MySQL Workbench connecting to the service. MySQL Workbench is a powerful tool, useful to administer the MySQL instance and to perform various tasks as the Import/Export feature.

## PostgreSQL

Azure Database for PostgreSQL service has been built in the same way as MySQL one. Compared to it, we can experiment the same features, the support of two versions on PostgreSQL engine, firewall and SSL support and the same pricing structure.

## Summary

In this chapter we learned how to approach SQL Database, the most advanced Database-as-a-Service of the Azure offering and one of the most advanced in the entire Cloud Ecosystem. We learned how to setup a good design process, an evolving maintenance plan and a strategy to monitor it continuously and efficiently. We also learned how to use SQL Database efficiently and how to get the most out of it with its valuable features. We focused on those features useful for a decision maker, as well as for an architect, to plan a project and know in advance the possible approaches to the service.

In the next chapter, we see how to deal with unmanaged RDBMS, with specific support to SQL Server in VMs (IaaS) and how to extend the on-premise topology with the appropriate building block offered by Azure.