*Azure SQL Database has a* [*new “serverless” mode*](https://docs.microsoft.com/en-us/azure/sql-database/sql-database-serverless?WT.mc_id=Revolutions-blog-davidsmi) *in preview that eliminates compute costs when not in use. In this post, I’ll show how you can set up a serverless database instance, and access data stored in it from R.*

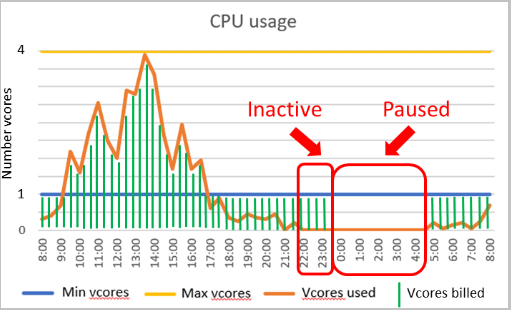
I’m working on a demo that I’ll be giving at several upcoming conferences, and for which I’ll be needing data in a database. Normally, I’d use a database installed on my local machine or in a virtual machine in the cloud, but this time I decided to go a different route: serverless.

I need the database to be around for a few months, but I’ll only be accessing it occasionally while I develop the demo and, later, present it live a few times. I need the database to be fairly large and fast (both of which rule out installing it on my laptop), and I’d prefer not to have to pay so much for the cloud resources while I’m not using it. (Yes, as a Microsoft employee I do have access to Azure services, but our department is cross-charged for our consumption and our spend is scrutinized like any other expense.) I considered using [specialized cloud-data services](https://docs.microsoft.com/azure/cosmos-db/introduction?WT.mc_id=Revolutions-blog-davidsmi), but for this talk I needed something that looked and felt like a traditional database.

I could always run a database in a virtual machine, but this time I decided to try [Azure SQL Database](https://docs.microsoft.com/azure/sql-database/?WT.mc_id=Revolutions-blog-davidsmi). Azure SQL Database runs just like an ordinary SQL Server instance on a named server, but without the need to create or manage any associated VM. Even better, I discovered that it now has a “[serverless](https://docs.microsoft.com/azure/sql-database/sql-database-serverless?WT.mc_id=Revolutions-blog-davidsmi)” pricing tier, which gets me several benefits:

* It will automatically scale its available compute resources up to the maximum number of vCores (virtual cores) I specify. The more cores in use, the more I am charged for compute. A minimum compute capacity (again, as specified) will always be provided while the database is running.
* If the database is inactive for a period I specify, it will automatically be paused. While paused, I only pay for storage, but not for any compute. When I next need to access data, the database automatically restarts on demand.

The chart below illustrates this process for an Azure SQL Database with a minimum of 1 vCores and a maximum of 4 vCores; the green lines show when and at what rate compute cycles are billed, depending on the actual compute demand (orange line) on the database.

[](https://docs.microsoft.com/azure/sql-database/sql-database-serverless?WT.mc_id=Revolutions-blog-davidsmi)

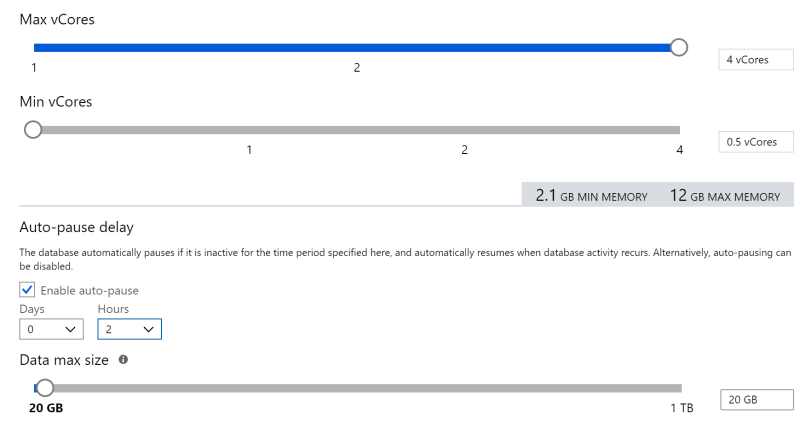
**Creating the database**

Setting up the Azure SQL Database instance is pretty simple, and [covered in detail here](https://docs.microsoft.com/azure/sql-database/sql-database-single-database-quickstart-guide?WT.mc_id=Revolutions-blog-davidsmi). From the Azure Portal, create a “SQL Database” instance (in the Databases section) and watch out for these steps during the creation process:

* You will be asked to choose a name for your new server: choose carefully here, as that will form the public URL you will use to access the server later.
* To set up serverless operation, click the “Configure database” option under “Compute + storage” and choose the **Serverless** compute tier.

[](https://revolution-computing.typepad.com/.a/6a010534b1db25970b0240a4722091200c-pi)

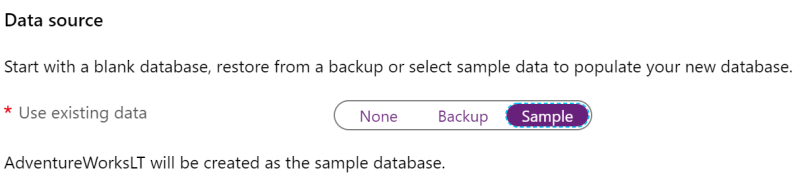
* With the serverless compute tier selected, you will then configure the capacity and performance of the database. These choices will determine the running cost.
* The maximum and minimum vCores determines the processing speed and memory allocated to the database. The default setting for minimum vCores is 0.5, and that’s the best choice if you want to minimize operating costs. You can change these settings after the database is created, too.

[](https://revolution-computing.typepad.com/.a/6a010534b1db25970b0240a4bff79b200b-pi)

* The Auto-pause delay determines how long it takes to suspend the database when it’s not being used. I chose two hours (instead of 1, the minimum) to give myself enough leeway between testing my demo and actually delivering the demo. That two hours of delay before each auto-pause will cost me $0.31, but means I can access the database during that period with no start-up delay. (This setting is reconfigurable after you create the database, and you can also switch off auto-pause later if needed.)
* Finally, you’ll need to choose the maximum capacity of your database. You pay for storage even when the database isn’t in use, so this will be your biggest factor in determining costs. You can adjust this setting up or down later, and the storage costs will adjust accordingly.

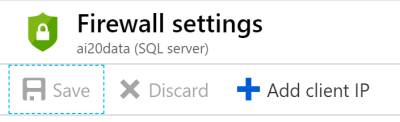
With the setup described above, this instance will cost me $3.59 per month for storage, plus at most $1.25 per active hour for compute. (In practice though I’ll be charged a tiny fraction of the compute cost, as this database will spend most of its time paused. And even when I am actively using it, it will spend most of that time consuming fewer than the maximum 4 vCores.

One last thing while you’re setting up the database: in the Additional Settings tab, you have the option of pre-populating the database with data from a SQL Server backup, or with the [AdventureWorks](https://github.com/microsoft/sql-server-samples/tree/master/samples/databases/adventure-works) data: a sample data set from a fictitious retail store. That’s the data I’ll be using in the example to follow.

[](https://revolution-computing.typepad.com/.a/6a010534b1db25970b0240a47220bc200c-pi)

**Enabling remote access to the database**

After setting up the database as described above, the first thing you’ll need to do is to open up the firewall to allow remote access. Initially, the firewall is configured to deny all remote access, so you’ll need to add the IP address of any devices that need to access the database remotely. The [details are described here](https://docs.microsoft.com/azure/sql-database/sql-database-server-level-firewall-rule?WT.mc_id=Revolutions-blog-davidsmi), but a simple way to get started is to click “Set Server Firewall” in the Overview page for the database in the Azure Portal, and click “Add Client IP”. This adds the IP address of the computer you’re currently using.

[](https://revolution-computing.typepad.com/.a/6a010534b1db25970b0240a49b54bd200d-pi)

**Installing Drivers**

You will also need to install a driver for your client machine to access the database. There are various driver options available, but I’ll be using ODBC. Windows 10 comes with a standard ODBC driver installed (it appears as “SQL Server”), but if you’re on Mac or Linux or want best results in Windows, I recommend installing the 64-bit (x64) version of [Microsoft ODBC Driver 13.1 for SQL Server](https://docs.microsoft.com/en-us/sql/connect/odbc/download-odbc-driver-for-sql-server?view=sql-server-2017#microsoft-odbc-driver-131-for-sql-server). If you use a different driver than that one, you’ll need to substitute its name in the connection string you get in next section.

**Accessing data in Azure SQL Database from R**

Now that the database is set up and the firewall open, you can access it like any other SQL Server database. I’ll be doing my demo using R, so let’s use that as an example client accessing the data remotely.

To illustrate how this works, let’s look at setting up an Azure SQL Database instance in serverless mode, configuring it with some sample data, and access the data from a client application: R.

To connect to the database with R, we’ll use the [DBI package](https://www.r-consortium.org/blog/2017/05/15/improving-dbi-a-retrospect) (an R Consortium project that provides a generalized interface to databases) and the [odbc package](https://db.rstudio.com/odbc/) from RStudio that provides the ODBC interface for SQL Server:

library(DBI)

library(odbc)

This is a good time to check you have a driver available to access the database, with:

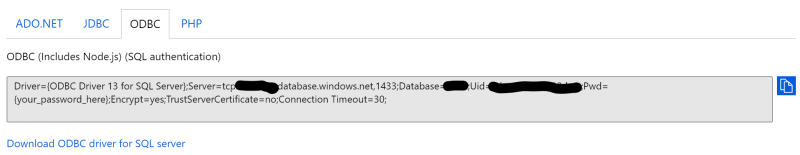
> sort(unique(odbcListDrivers()[[1]]))

[1] "ODBC Driver 13 for SQL Server" "ODBC Driver 17 for SQL Server"

[3] "SQL Server"

I recommend using “ODBC Driver 13 for SQL Server” (you can [download it here](https://docs.microsoft.com/en-us/sql/connect/odbc/download-odbc-driver-for-sql-server?view=sql-server-2017#microsoft-odbc-driver-131-for-sql-server)), but if you only have a generic ODBC driver (like “SQL Server”, which comes installed as standard on Windows 10) it will most likely work (but possibly with reduced efficiency).

With a suitable driver available, you can create a connection to the database. The easiest way to do this is to visit your database in the Azure Portal and click “show database connection strings” in the Overview pane, and choose the ODBC option. I’ve blacked out the server name, database name, and admin user ID below: you select those when you first create the database.

[](https://revolution-computing.typepad.com/.a/6a010534b1db25970b0240a4bff7ff200b-pi)

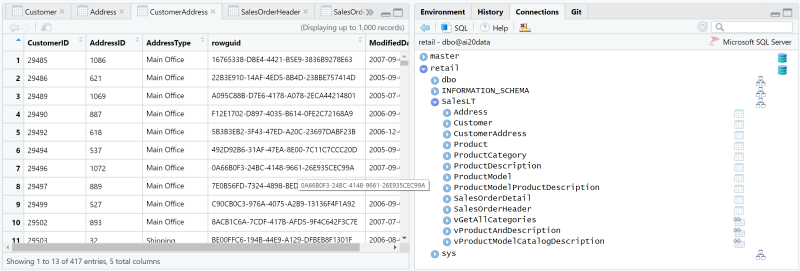
Copy that string and replace {your\_password\_here} with the password you selected during setup (without the braces). Now you can create the connection in R:

constr <- "\*insert modified connection string here\*"

con <- dbConnect(odbc(), .connection\_string=constr)

(Note: if you get the error message of the form Login timeout expired or TCP Provider: Timeout error it means the database has paused for inactivity. In that case, wait about 20 seconds and run the dbConnect call again. Sadly, the Connection Timeout setting doesn’t seem to help with this.)

Now, you have everything you need to access the data from R. If you’re using RStudio, you can use the Connections pane to explore the available tables, or even click on the View Table icon to take a look at the data in the first few rows:

[](https://revolution-computing.typepad.com/.a/6a010534b1db25970b0240a4bff80c200b-pi)

If you know the [T-SQL](https://docs.microsoft.com/sql/t-sql/language-reference?view=sql-server-2017&WT.mc_id=Revolutions-blog-davidsmi) query you want to run on the data, you can do that with dbQuery and return the results as an R data frame. For example, this query provides the product category for each product in the **Product** table:

dbGetQuery(con,'

SELECT pc.Name as CategoryName, [p.name](http://p.name) as ProductName

FROM [SalesLT].[ProductCategory] pc

JOIN [SalesLT].[Product] p

ON pc.productcategoryid = p.productcategoryid;

')

If you’re not comfortable with SQL, you can use the [dbplyr package](https://dbplyr.tidyverse.org/articles/dbplyr.html) to write the queries for you. In the example below, even though it’s using standard dplyr syntax, all of the computations are taking place in the Azure SQL Database. The results only returned to R on the collect() call, which means that you can calculate on large data in the database and return only the summarized results to R. For example, this code returns the top 5 products in the AdventureWorks sales data by total sales:

library(dplyr)

library(dbplyr)

sales <- tbl(con, in\_schema("SalesLT","SalesOrderDetail"))

sales %>%

group\_by(ProductID) %>%

summarise(Total\_Sales = sum(LineTotal, na.rm=TRUE)) %>%

arrange(desc(Total\_Sales)) %>%

head(5) %>%

collect()

**Trying it Out**

The serverless compute tier for Azure SQL Database is available in public preview now. To learn more about setting up the database, the Microsoft Learn module [Provision an Azure SQL database to store application data](https://docs.microsoft.com/learn/modules/provision-azure-sql-db/?WT.mc_id=Revolutions-blog-davidsmi) is a good place to start. All you need is an Azure subscription, and if you don’t have one already you can [use this link to sign up](https://azure.microsoft.com/free/?WT.mc_id=Revolutions-blog-davidsmi) and also get $200 in free credits to use on anything you like, including Azure SQL Database.