I’m pleased to announce that the AzureTableStor package, providing a simple yet powerful interface to the Azure table storage service, is now on CRAN. This is something that many people have requested since the initial release of the AzureR packages nearly two years ago.

Azure table storage is a service that stores structured NoSQL data in the cloud, providing a key/attribute store with a schemaless design. Because table storage is schemaless, it’s easy to adapt your data as the needs of your application evolve. Access to table storage data is fast and cost-effective for many types of applications, and is typically lower in cost than traditional SQL for similar volumes of data.

You can use table storage to store flexible datasets like user data for web applications, address books, device information, or other types of metadata your service requires. You can store any number of entities in a table, and a storage account may contain any number of tables, up to the capacity limit of the storage account.

AzureTableStor builds on the functionality provided by the AzureStor package. The table storage service is available both as part of general Azure storage and via Azure Cosmos DB; AzureTableStor is able to work with either.

Tables

AzureTableStor provides a `table_endpoint` function that is the analogue of AzureStor’s `blob_endpoint`, `file_endpoint` and `adls_endpoint` functions. There are methods for retrieving, creating, listing and deleting tables within the endpoint.

```
library(AzureTableStor)

# storage account endpoint
endp <- table_endpoint(
    "https://mystorageacct.table.core.windows.net",
    key="mykey")
# Cosmos DB w/table API endpoint
endp <- table_endpoint(
    "https://mycosmosdb.table.cosmos.azure.com:443",
    key="mykey")

create_storage_table(endp, "mytable")
list_storage_tables(endp)
tab <- storage_table(endp, "mytable")
```

Entities

In table storage jargon, an *entity* is a row in a table. The columns of the table are *properties*. Note that table storage does not enforce a schema; that is, individual entities in a table can have different properties. An entity is identified by its `RowKey` and `PartitionKey` properties, which must be unique for each entity.

AzureTableStor provides the following functions to work with data in a table:

- `insert_table_entity`: inserts a row into the table.
- `update_table_entity`: updates a row with new data, or inserts a new row if it doesn’t already exist.

- `get_table_entity`: retrieves an individual row from the table.
- `delete_table_entity`: deletes a row from the table.
- `import_table_entities`: inserts multiple rows of data from a data frame into the table.

```
insert_table_entity(tab, list(
    RowKey="row1",
    PartitionKey="partition1",
    firstname="Bill",
    lastname="Gates"
))

get_table_entity(tab, "row1", "partition1")

# we can import to the same table as above:
# table storage doesn't enforce a schema
import_table_entities(tab, mtcars,
    row_key=row.names(mtcars),
    partition_key=as.character(mtcars$cyl))

list_table_entities(tab)
list_table_entities(tab, filter="firstname eq 'Satya'")
list_table_entities(tab, filter="RowKey eq 'Toyota Corolla'")
```

Batch transactions

With the exception of `import_table_entities`, all of the above entity functions work on a single row of data. Table storage provides a batch execution facility, which lets you bundle up single-row operations into a single transaction that will be executed atomically. In the jargon, this is known as an *entity group transaction*. `import_table_entities` is an example of an entity group transaction: it bundles up multiple rows of data into batch jobs, which is much more efficient than sending each row individually to the server.

The `create_table_operation`, `create_batch_transaction` and `do_batch_transaction` functions let you perform entity group transactions. Here is an example of a simple batch insert. The actual `import_table_entities` function is more complex as it can also handle multiple partition keys and more than 100 rows of data.

```
ir <- subset(iris, Species == "setosa")

# property names must be valid C# variable names
names(ir) <- sub("\\.", "_", names(ir))

# create the PartitionKey and RowKey properties
ir$PartitionKey <- ir$Species
ir$RowKey <- sprintf("%03d", seq_len(nrow(ir)))

# generate the array of insert operations: 1 per row
ops <- lapply(seq_len(nrow(ir)), function(i)
    create_table_operation(endp, "mytable", body=ir[i, ],
                           http_verb="POST")))
```

```
# create a batch transaction and send it to the endpoint
bat <- create_batch_transaction(endp, ops)
do_batch_transaction(bat)
```