2012-10

An Investigation of Distributed Schema Free Tabular Data Storage Technologies on Google App Engine and Microsoft Azure

Conor McGrath
Technological University Dublin

Gary Clynch
Technological University Dublin

An investigation of distributed schema free tabular data storage technologies on Google App Engine and Microsoft Azure

Conor McGrath [1], Gary Clynch [2]

[1] ITT Dublin, Institute of Technology Tallaght,Tallaght, Dublin 24, Ireland
cmcgrath@itnet.ie

[2] ITT Dublin, Institute of Technology Tallaght,Tallaght, Dublin 24, Ireland
gary.clynch@ittdublin.ie

Abstract

This paper examines schema free non-relational tabular storage for two important emerging Platform as a Service (PaaS) environments, where Google App Engine applications persist data to the Google Datastore, and Microsoft Azure applications store data in Azure Tables. A simple mobile web application was initially developed for both platforms, to understand how an application could be developed and deployed. Java Data Objects (JDO) was selected for Google App Engine and Windows Communication Foundation (WCF) in C# for Microsoft Azure. Many applications have a requirement to store complex data that is organised in relations in the order of *One to One*, *One to Many* and *Many to Many* that are often implemented in a relational database. Azure SQL and Google Cloud SQL are RDBMS technologies implemented on these Cloud platforms but are more costly to use. A testbed was developed to investigate implementation of these relations using automatic management features and developer managed techniques on each platform using JDO and WCF. An evaluation on primitive operations was carried out on both environments indicating each provided adequate operations to create, update, delete and display data. Due to design differences App Engine JDO scored better. An evaluation of how relations could be implemented was also conducted. Google App Engine JDO provides automatic management of *One to One* and *One to Many* relations while Microsoft Azure does not provide any relations management features. Google App Engine JDO permits storage of non-primitive types such as *java.util.List* objects which allows for tracking of child entity keys with ease, but this is not available in Azure Tables. Attempting to develop *Join* semantics in application code proved to be error prone and difficult with Azure Tables. A number of alternative proposals are made to implement relations on Azure Tables.

Keywords: Cloud Storage, PaaS, App Engine, Azure, Table Storage

1 Introduction

In these current uncertain economic times, businesses need to keep costs down, by investing in IT applications and systems that can clearly be shown to contribute to business results [1]. Cloud Computing promises to supply computing resources to match application workload demand, helping to reduce costs where workload is low and satisfy customer demand during peak seasonal periods [2].

The purpose of this work was to examine new emerging storage technologies that could be used instead of relational database technology solutions, on Google App Engine and Microsoft Azure platforms. These new services are generally classified as NoSQL type services [3].

Microsoft Azure is an operating system for the cloud that runs within Microsoft datacenters spread across multiple servers and networks. It provides compute, storage, automated service management

facilities, a relational database called SQL Azure, distributed access control and the service bus providing a mechanism to integrate customer onsite data with the cloud [4].

Google App Engine provides an operating environment running reliably under heavy workloads, can serve dynamic web pages content, implement data storage with sorting, provide transactions and query features, load balancing and scaling running JVMs and storage scaling that is automatic and a local development environment providing facilities to test solutions before deployment. Additionally, task queues allow for work to be executed outside of web requests and time based scheduled tasks can be run [5].

The remainder of the paper continues with details on experiments carried out in Section 2. A short evaluation of the development environment is conducted in Section 3. An evaluation of primitive operations is carried out in Section 4, and an evaluation of data relation experiments is examined in Section 5. Finally, in Section 6 conclusions are made with future work outlined.

2 Implementation of Experiments

This study aimed to examine both Azure Tables and App Engine JDO by porting a simple web application to each environment, providing an understanding of implementation semantics and working environment. Each testbed created data relationships in the orders of 1-1, 1-m and m-n. Further examination of primitive tabular storage APIs, features and limitations were examined via these test projects. Each component examined was reviewed and scored according to Table 1. The scale was selected to distinguish between the quality of implemented features and allows for direct comparison between features that are deemed equivalent. No emerging standard has been identified to determine correct implementation criteria of schema free storage or PaaS environments. The expectation is that an experienced software developer or system administrator would come to the same conclusions.

Microsoft Azure Tables provide tabular structured storage of data where an entity class is mapped to table storage and is persisted to a table. Each row of data in a table is represented by an entity class that inherits from the abstract class *TableServiceEntity*. All storage entities contain *PartitionKey*, *RowKey* and *TimeStamp* data fields. The key fields must be set and dictate preferences for data partitioning. A storage operation context must be derived from super class *TableServiceContext*, which itself is derived from *DataServiceContext* defined in WCF Data Services [6]. It is recommended to develop a *DataSource* class that instantiates the context where high level business methods can be developed for consumption by application code. *Entity Group Transactions* allow for batch transactions to execute over elements in the same partition, identified by the *PartitionKey*. Azure Tables does not support *Join* operations natively where LINQ operations are not implemented to do so [7]. No transactional facility is provided across tables which is commonly implemented in RDBMS environments. Details on how replication between datacenters are limited but 3 table replicas are kept across datacenters [4].

Datanucleus provided Google with a base implementation of Java Data Objects (JDO) and Java Persistence Framework (JPA) [8]. A low level entity framework was also exposed allowing developers implement solutions directly or port new frameworks such as Objectify-App Engine [9]. JDO was selected for this examination where data members of a class are annotated to define an entity class, which is mapped to the Datastore. A *PersistenceManager* Singleton object executes query, update and delete operations on entities in the Datastore. A Data Access Object (DAO) object may be implemented in the application to provide high level data operations.

The Datastore initially was deployed on Google Bigtable which was designed to scale to petabytes of data across thousands of machines in a reliable manner [10]. In 2009 it was announced that the datastore would be migrated to MegaStore [11] to provide data redundancy across datacenters. ACID transactions execute within *Entity Groups* where stored objects are mapped to an internal *Datatore* element called a *Kind* with looser consistency for operations across data groups [12]

The testbed code implementation was developed on Google App Engine (Java) and Azure (C#) to understand how relations may be modeled on Azure Tables [13] and on Google App Engine Dataservice [5].

Score	Explanation
0 (Poor)	Does not achieve objectives
1 (Good)	Achieves most objectives
2 (Very Good)	Achieves objectives

Table 1: Scoring Rules for Examinations

2.1 Data Relations Testbed

The Google App Engine Testbed *App EngineTestBed* examines relationships in two different scenarios, where relationships are managed by the code known as an *unowned relationship*, and cases where automatic management features are used to maintain the relationship between objects known as *owned relationships*. The latter case should remove some burden from the developer. Owned relationships exist only for 1-1 and 1-m cardinality. For m-n relationships the relationship needs to be managed by code. All code was developed in JDO [14] and executed from a simple test servlet. Tests were executed on the local development environment and within the Cloud. Each test case implemented *insertData*, *updateDate*, *displayData* and *deleteData* operations.

Test cases executed covered *One to One Owned Relationship*, *One to One Unowned Relationship*, *Embedded Entity Relationship* where an internal static class may be defined in an entity class, *One to Many Owned Relationship*, *One to Many Unowned Relationship* and *Many to Many Relationship* without automatic relationship management.

The Microsoft Azure Testbed code was implemented as *TestBedLib* C# library that is used by a simple test harness provided as an Azure *webrole*. Each test case defines *Delete*, *Update*, *Insert* and *Display* operations.

One to One Relationship, *One to Many Relationship* and *Many to Many Relationship* test cases were created. Azure Tables does not provide any automatic method to manage data relations.

3 Development Environment Evaluation

Results from the examination of the operating environment are shown in Table 2. Both Azure and App Engine provided introductory material, covering concepts with accompanying examples but substantial effort was needed to understand and make example code work. During the lifetime of this project efforts were made by both organisations to improve these resources via documentation, and web resources published at Google I/O and Microsoft PDC. As these are new technologies resources were limited initially, but there has been substantial improvement in examples and documentation. A rating of 1 has been set for learning for both technologies.

Both provided facilities to test locally before deployment, where each technology deployed applications to an emulated environment on the workstation. With each build iteration released there were improvements in performance. Both environments were rated 1 for testing applications.

Deployment is possible from each development environment. In the case of Google App Engine a small application will deploy in 2 to 3 minutes. This took much longer, often taking 15 minutes for a small application to be deployed on Microsoft Azure. Additionally a command line tool *appcfg*, provided by Google, may be used to deploy an application. It may be possible to integrate this into a company's build and deploy process. Google App Engine is rated 2 and Microsoft Azure rated 1.

Maintenance tasks can be carried out on applications that are deployed. In both environments it is possible to deploy to a testing area and then activate as a production release quickly when needed. Both technologies are rated at 2.

	Azure Tables	App Engine JDO
Learning	1	1
Testing	1	1
Deployment	1	2
Maintenance	2	2
Total	5	6

Table 2: Scoring of PaaS environment review

4 Evaluation of Primitive Operations

Within each environment is an obvious requirement that data can be queried, updated and deleted. From general use, experiments and examination of these technologies a number of primitive operations and attributes were examined. In line with grading method in Table 1 these operations were examined. Results are summarised in Table 3.

In the case of both environments a class is defined which is mapped to storage. It is not possible to create a subclass of this object in Azure Tables and in App Engine JDO polymorphic relationships are not supported [15]. Both technologies are rated 1.

Both environments support the easy creation and storage of an object. With Microsoft Azure Tables the ADO.Net Entity Framework is used to create the object, track the object and save changes to storage. App Engine JDO creates an entity object which is passed to the *makePeristent* method of the *PeristenceManager* object. Both environments are scored 2.

Multiple objects may be added successfully to both environments. In the case of Microsoft Azure Tables ADO.Net objects are created one at a time, each object is tracked and data persisted via a batch save operation. App Engine JDO objects are added to a collection such as *java.util.List*, and the collection may be persisted directly to storage or the elements may be retrieved from the collection one at a time and persisted to storage. In both cases the score awarded is 2.

A single object may be deleted from storage from both environments. The Microsoft ADO.Net Entity Framework provides a limited LINQ implementation where an object may be queried using indexed table keys *PartitionKey* and *RowKey*. A strong implementation feature is that it is possible to specify one object or a collection of objects to be returned. There are no restrictions on using any other data member but a full scan of the Azure Table will occur if non-indexed data member fields are used in queries. In App Engine JDO the *PersistenceManager* has a method to return an object by key or return a single object or collection by a JDO Select query. Single Object Deletion is rated as 2 for both technologies.

For cases that require batch deletion both environments support this. Microsoft Azure Tables must mark each entity for deletion and issue batch save operation to execute the request. App Engine JDO provides a facility to delete all entities in a collection without the need to iterate over that collection. Azure Tables is awarded 1 while App Engine JDO is awarded 2.

Both environments support the updating of a single entity where the entity is retrieved, updated and persisted to storage. The retrieval method has been outlined already. An Azure Table entity is tracked, the values updated and changes are saved. Each App Engine JDO object is explicitly persisted or will be persisted when the persistence manager closes. Both features are scored 2.

In tests updating by batch was not possible on Azure Tables where *ReplaceOnUpdate* save option needs to be used to persist updates one entity at a time to storage. App Engine JDO allows for collections of updated objects to be saved to storage in their collection directly. In this instance Azure Tables are scored 0 and App Engine JDO scored 2.

When creating queries Azure Tables is limited to just 2 indexed values if efficient lookup is expected. The *PartitionKey* will group data on one storage node provided it can fit. In the case where the partition continues on another storage node, or 1000 entities, or 4mb of data is transferred, a continuation token is issued in the response header [13]. The later *CloudTableQuery* implementation will process a contin-

uation token automatically. App Engine JDO allows for indexing by primary key, allows for naming of additional indexed or indexing on all stored data members automatically. Azure Tables is scored 1 and App Engine JDO scored 2.

An interesting case is where a query must be created at runtime dynamically. App Engine JDO allows a query, very similar to simple SQL language, be constructed as a string and executed. Azure Tables did not have this feature where dynamic LINQ [7] is needed. Azure Tables was scored 1 and App Engine JDO scored 2.

Both environments provide data pagination solutions where Azure Tables may use LINQ Take and Skip operations and App Engine uses a paging cursor that provides a link to the next result in the query. Both technologies are scored 2.

Multitenancy can be achieved in each environment. Azure Tables can be achieved by the application to select the names of tables, for instance by adding a prefix for customer account. App Engine JDO provides a namespace API that simply controls the visibility of application queries by adding a namespace prefix to all primary keys. Both environments are scored 2.

Operation	Azure Tables	App Engine JDO
Object Declaration	1	1
Add Object	2	2
Add Multiple Objects	2	2
Delete Single Object	2	2
Delete by Batch	1	2
Update Single Object	2	2
Update by Batch	0	2
Query by Index	1	2
Runtime Query Creation	1	2
Pagination	2	2
Multitenancy	2	2
Total	16	21

Table 3: Scoring of primitive storage operations and attributes

5 Data Relations Evaluation

Table 4 shows evaluation scores awarded. In the case of 1-1 relationships App Engine JDO provides an automatic relationship mapping. Queries on the parent object loads the child object data automatically, which can be accessed without need to query separately. In testing it was required explicitly to return the child object from the parent and then to delete the child prior to the parent. The feature is scored 1 due to this limitation. The unmanaged mode can also be used. It was found that a string representation of the child object key could be stored as an attribute in the parent object. This allowed easy construction of a query key for the child entity. Automatic managed relations allow JDO transactions to be applied to the updates, insertions and deletions. This is not possible in the unmanaged scenario. An interesting option is the embedded class in App Engine JDO where a static class, annotated as *EmbeddedOnly* may be added to the entity to be stored. The entity stored contains the data members of both classes as one entity. This feature is scored 2.

Azure Tables does not provide a method to implement automatic relationship management. An unmanaged relationship can be implemented by storing a child entity's *Partition Key* or *Row Key*. This does mean that 2 trips are needed to obtain the parent object and then the child object. This is scored 1.

For the 1-m case App Engine JDO supports automatic management of parent and children relationships, which works well, where an index is created automatically for child objects. In code a *java.util.List* object holds the list of children in the parent object. This feature is scored 2. An alternative is the unmanaged 1-m relationship where the developer must load the parent and then children initially, and then

Relationship	Azure Tables	App Engine JDO
One to One Owned	N/A	1
One to One Unowned	1	2
One to One Embedded	N/A	2
One to Many Owned	1	2
One to Many Unowned	1	1
Many to Many	1	1
Total	4	9

Table 4: Scoring of Relations

	Chain	Composite	Range	Complex	Sparse
One to One	X	X			
One to Many	X		X	X	X
Many to Many	X		X	X	X

Table 5: Techniques to implement relation storage

handle all updates, insertions and deletions. This was scored 1.

Azure Tables requires that developers provide their own algorithm to implement 1-m relations. The simplest idea is to store attributes in a parent table, which are keys to child entities and can be used to query those separate tables. This is scored 1.

In both Azure Tables and App Engine JDO m-n relationships need to be managed by the developer. It is possible in App Engine to develop a 1-m relation in both directions. In the case where the Datastore assigned and managed keys, those values were not known until the object was stored. To provide the key of a child to a parent to persist in as a collection it required the child object be written and then read to discover it's automatic key value. This extra trip to storage is not efficient. Alternatively, each child could create it's key based on a scheme such as parent object string with an element position value, where these values could be stored in the parent. App Engine JDO is rated 1.

For Azure Tables storing a collection directly is not possible as primitive storage values, such as numerical and string types, are only permitted. Joins on entity attributes are not possible either. Developing a solution was possible but slow and error prone.

App Engine JDO transactions are not possible for unmanaged cases where entities are not in the same *Entity Group* [12]. Higher consistency is inferred on local indexes, within an *Entity Group*, but this to lesser degree on global indexes between *Entity Groups*. Two-phase commit between Entity groups is possible for for 1-1 and 1-m managed relations, referred to as owned relations. All communication between datacenters are synchronous and consistent, and that secondary indexes are available on all Entity properties, which is implemented in App Engine JDO [12]. Data is mapped and stored in BigTable under *Multiversion Concurrency Control* management [16].

Table 5 outlines a number of techniques that could be applied to the management of these relations, in Azure Tables in particular. Within Azure Tables the lack of a facility to join tables on an attribute is problematic. The simplest technique is to take a 1-1 relation and store all data members in just one class. In the case of Azure Tables it is not required that all fields be filled and where fields are empty null values are placed in the table which lowers storage requirements. This can not apply to 1-m or m-n.

A further proposal is that the common idea of a horizontal join should be replaced by a vertical join, which is introduced as a *Chain Relation*. Instead of modeling a class that has fixed attributes the *PartitionKey* and *RowKey* can be used. For example, in the case of an entity class modeling a person the *PartitionKey* identifies a person and the *RowKey* identifies the type of data that is stored, where previously this would have been modeled in another entity and table. It is acknowledged that the application code

needs to be aware of what type of data is stored in the variable storage fields. It may be possible to develop a soft schema that could be stored in a table, that is loaded when the program starts or store textual data as a tuple containing type identifier and data. This could be applied to all three cases. In the 1-m and m-n cases it is proposed to implement a *NextRecord* attribute linking to the *RowKey* and a final *End* value when the list is exhausted.

A further amendment to the previous technique, called *Range Relation*, is to implement *FirstKey* and *LastKey* fields in the entity. The key for each entity is stored in the *RowKey* except for the parent which has a data type identifier. The parent can detect and access all children.

Another derivative is the *Complex Relation*, where just one *Description* key replaces the *FirstKey* and *LastKey* fields. All new elements added as *RowKeys* are written to the *Description* delimited by a colon or some other agreed delimiter.

Sparse Relation could be implemented where a new field is dynamically added to the stored entity. The *DataServiceContext* methods *WritingEntity* and *ReadingEntity* have been suggested to add and remove these data fields [17].

6 Conclusions and Future Work

The work carried out shows that developers familiar with both the Eclipse and Visual Studio 2010 integrated development environments can be productive very quickly. Each environment provides a local test server to execute test cases prior to deployment in the Cloud, with no need to make changes within the application code. An effort has been made to ensure, where possible, that well known APIs are implemented natively or are mapped to the new infrastructure, thus improving developer productivity and encouraging early investigation by developers. Overall the review scored App Engine JDO as the better environment by a narrow margin, where deployment time was substantially slower on Azure deploying code packages, where storage had been already allocated.

Effort was made to implement adequate primitive data operations on App Engine JDO and Azure Tables. One limiting feature with Azure Tables is the restriction to just two indexed keys for queries, which means that storage needs to be arranged to support queries carefully. In the case of App Engine JDO additional indexes can be added allowing for new queries to be developed during the life of the application with greater ease. Both environments use existing APIs that are strong, but are limited only by the underlying storage architecture. Currently, greater restrictions are seen within the Azure Tables environment where a developer must be aware of both a table and entities stored in this table. This differs in App Engine JDO where entities and their access keys are important. With a simpler usage model, by Google, it could be argued that there is more flexibility for architecture to be refined further. The evaluation outcome indicates App Engine JDO is stronger.

From the examination of the relations test cases it is possible to implement solutions that represent information in 1-1, 1-m and m-n relations. App Engine JDO implements techniques that are managed automatically by the environment for 1-1 and 1-m relations. These are very much welcomed, reducing the burden on the developer, and could also help ensure correctness of solutions. For m-n relations on App Engine JDO, and also for 1-1, 1-m and m-n relations on Azure Tables, these features must be implemented by the developer. The approach taken for Azure Tables test case relations was done in a relational database design mindset. A number of issues, such as the inability to join Azure Table objects via LINQ and the lack of transactions across tables made useful implementation difficult. It was proposed to implement relations in outlined *Chain*, *Composite*, *Range*, *Complex* and *Sparse* methods. In this evaluation App Engine JDO was more successful.

Further investigation by implementing a multitenant application that uses these proposed techniques, measuring performance and accuracy under various workloads could prove enlightening. Whether a transaction log should be created by writing actions to a dedicated transaction table or Blob asynchronously, could be considered. In Azure Tables, there could be a case where queries on data that does not appear in the partition key, or partition and row key, could be added as a key to a new table providing indexes to other data.

References

[1] Gartner Inc. *Top Technology Predictions for 2011 and Beyond*. 2011. http://www.gartner.com/technology/innovation/it-predictions.jsp [May 2011].

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. *Above the Clouds: A Berkeley View of Cloud Computing*. (UCB EECS-2009-28), Feb 2009.

[3] NoSQL. *NoSQL - Your Ultimate Guide to the Non-Relational Universe*. 2011. http://nosql-database.org/ [May 2011].

[4] Microsoft Inc. *Introducing the Windows Azure Platform*. 2011. http://www.microsoft.com/windowsazure/Whitepapers [May 2011].

[5] Google Inc. *Google AppEngine*, 2011. http://code.google.com/appengine/docs/whatisgoogleappengine.html [Jan 2011].

[6] Microsoft Inc. *WCF Data Services*. 2011. http://msdn.microsoft.com/en-us/data/bb931106 [May 2011].

[7] Microsoft Inc. *Windows Azure Platform - LINQ Query Operators*. 2011. http://msdn.microsoft.com/en-us/library/windowsazure/dd135725.aspx [Nov 2011].

[8] Datanucleus. *Datanucleus Access Platform*, 2011. http://www.datanucleus.org/products/accessplatform/ [May 2011].

[9] Objectify. *objectify-appengine - The simplest convenient interface to the Google App Engine datastore*. 2011. http://code.google.com/p/objectify-appengine/ [Jan 2011].

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. *Bigtable: a distributed storage system for structured data*. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[11] Google Inc. *Migration to a Better Datastore*. 2011. http://googleappengine.blogspot.com/2009/09/migration-to-better-datastore.html [May 2011].

[12] Jason Baker, Chris Bondç, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean M. Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. *Megastore: Providing Scalable, Highly Available Storage for Interactive Services*. In *In Conference on Innovative Data Systems Research (CIDR)*, pages 223–234, Jan 2011.

[13] Jai Haridas, Niranjan Nilakantan, and Brad Calder. *Windows Azure Table - Programming Table Storage*. 2009. http://www.microsoft.com/windowsazure/whitepapers [Feb 2011].

[14] Java Community Process (JCP). *JSR-000243 Java Data Objects 2.0*. 2010. http://jcp.org/aboutJava/communityprocess/final/jsr243/index.html [Oct 2010].

[15] Google Inc. *Entity Relationships in JDO*. 2011. http://code.google.com/appengine/docs/java/datastore/jdo [May 2011].

[16] Philip A. Bernstein and Nathan Goodman. *Concurrency Control in Distributed Database Systems*. 13:185–221, June 1981.

[17] Neil MacKenzie. *Entities in Azure Tables*. 2011. http://convective.wordpress.com/2009/12/30/entities-in-azure-tables/ [May 2011].