Improving Data Services of Mobile Cloud Storage with Support for Large Data Objects using OpenStack Swift

Aslam B Nandyal[1], Mohammed Rafi[2], M Siddappa[3], Babu B. Sathish[4]
Dept. of Computer Science & Engg, University BDT College of Engineering, Davanagere, Karnataka, India[1,2]
Dept. of Computer Science & Engg, Sri Siddhartha Institute of Technology Maralur Tumkur, Karnataka, India[3]
Department of Computer Science & Engg, R.V. College of Engineering Bengaluru, 560059, India[4]

***Abstract*—Providing data services support for large file upload and download is increasingly vital for mobile cloud storage. There is an increase in mobile users whose data access trends show more access and large file sharing. It is a challenging task for Mobile Application Developers to handle upload and retrieve large files to/from a mobile app because of difficulties with latency, bandwidth, speed, errors, and disruptions to service in a wireless mobile environment. Some scenarios require these large files to be used offline, sometimes to be updated by a single user, and sometimes be shared among all other users. The Wireless mobile environment must consider mobile user's constraints, such as frequent disconnections and low bandwidth, which affect the ability to handle data and transactions management. The primary objective of this study is to propose a cloud-based Mobile Sync service (sometimes referred as Mobile backend as a Service) with OpenStack Swift object storage to manage large objects efficiently using two main techniques of segmentation and object chunking with compression in a mobile cloud environment. This work further contributed to a prototype implementation of the proposed framework and provides Application Programming Interface (API) consisting of Create, Read, and Delete queries and chunking operations and a lightweight sync protocol that can manage large file synchronization and access. The experimental findings with object-chunking tested size settings show that the proposed Mobile Sync framework can accommodate large files ranging from 100MB to 1GB and provides a decrease in upload/download synchronization times of 63.203% / 92.987% percent as compared to other frameworks.**

Keywords*—Mobile cloud computing; mobile backend as a service; large files; distributed systems

I. Introduction

The development of enterprise mobile services can use variety of cloud computing models. The predominant service models for cloud computing are mainly classified as Infrastructure as a Service (IaaS), Software as a Service (SaaS) and Platform as a Service (PaaS) [1]. With the introduction of a new service model known as Mobile Backend as a Service (MBaaS), making it easy to integrate cloud-based applications with mobile platforms has become possible.

The model of Sync framework offers a cloud server infrastructure to store application data and facilitate easy configuration. The developer needs to do significant work for the application to remain responsive during interruptions in communication due to poor or no network. The Sync framework offers a solution for the unreliable connection problem with customized synchronization and replication processes and helps synchronize with multiple clients. An intelligent Sync framework allows enterprise data to take offline and facilitate sync operation by syncing data across multiple mobile devices with the backend systems, detect and resolve the conflicts with configurable, standards-based rules, setting precedence based on policies [2] [3]. Ideally, the Sync framework should provide a consistent state at all times (strong consistency). However, the CAP theorem [4] for the distributed systems enforces the Sync framework to guarantee immediate availability and tolerate network partitions to provide a weak form of consistency, commonly known as eventual consistency [5].

Data Services is one of the critical capabilities of an MBaaS platform and provide the following features:

- **Data Management**: A quality MBaaS framework will provide the services to create, save, manage and sync application data and files within the framework itself, in addition to a mechanism for connecting into both public and private systems of record and abstraction layers [3].
- **Online/Offline Workflow**: TThe mobile apps can operate in online and offline modes, and hence MBaaS can support offline/online database synchronization [6].
- **Sync**: An intelligent MBaaS framework allows enterprise data to take offline, syncing data across multiple devices with the backend systems, detect and resolve the conflicts with configurable, standards-based rules, setting precedence based on policies [2].
- **Caching**: Various caching methods are offered as a part of the MBaaS platforms to reduce latency and boost app performance. The caching can be provided as an inbuilt feature in the framework or at a connection point to systems of record or even cross-platform on-device caching strategies (via client SDKs) [7].

Due to the rising numbers of people making their files shareware accessible via mobile devices, the reliability and efficiency of service for large files is a more crucial feature in mobile file-sharing than previously thought [8] [9]. Developers have to deal with complex issues of latency, speed, timeouts, and interruptions during the uploading and retrieving from mobile apps [10] [11] [12].

Recently, some efforts are focused on finding out the trend

of mobile users to access data on both public and private cloud storage services and several different personal cloud storage systems. One such study in [9] aimed to analyze data access trends in large-scale mobile cloud storage [9].

This research aimed to analyze the database of 350 million HTTP transaction log files from mobile applications to find the trend in how quickly cloud storage is being used for collaborative and large file storage. This study concluded that for retrieving one file in multiple sessions, the average volume was about 70 MB.

An empirical study was conducted by a cloud storage vendor [8], which involved the services of media file uploading, transformations, and storage in the cloud. This research studied a data set of one million mobile applications rendering these services. Their observation examined the statistical information regarding the number of files, including different file sizes and formats, which have been uploaded (from the year 2015 to 2016). Their research showed that the amount of growth files with size 100MB and above is 170%, while files of other sizes increased by 50% year over year. According to this study, it appears that the file sizes are increasing as mobile users make more frequent use of the files or share files and have larger storage requirements (above 100MB).

Some of the mobile operating systems limit the size of the file over which over-the-air (OTA) or app-store downloads are not allowed [13]. For example, Apple's iOS platform [14] limits the Cellular Data downloads to a file size of 100 MB. Android OS limits the size of downloads via cellular data to 150MB [15]. Based on the above studies and mobile operating system guidelines, it can be concluded that a file with a size greater than 100MB is considered a large file.

The rest of this paper is structured in the following manner: Next Section II provides a brief background and problem formulation. Section III analyzes some of the Mobile Sync frameworks in the literature along with support for large objects. Section IV provides the background information of the OpenStack cloud platform and support for large file storage in the Swift module that handles object storage. Section V describes the details of the proposed framework for data services to handle large files. The data management at both mobile Client and Cloud server-side is discussed. Section VI discusses the performance of the proposed Mobile Sync framework, followed by a conclusion and future work in Section VII.

II. Background and Problem Formulation

More recently, there has been an exponential growth in mobile devices with requirements for seamless personal data synchronization and availability across devices for mobile users. Different methods, such as Chunking, Deduplication, Segmentation, and Delta-encoding, have been used by cloud storage providers to maximize storage space and reduce transmission time [10]. In addition to custom features, various Mobile Cloud Storage providers have developed and implemented services to incorporate techniques of Chunking, Deduplication, Segmentation, and Delta-encoding. Despite all the efforts, there is still much space for improvement in handling syncing data in the mobile cloud storage, as the sync time is much longer than expected under some circumstances. Executing data synchronization is a challenging task in a mobile/wireless environment with frequent disconnections.

As commercial storage systems are primarily closed source with encrypted data, the public remains unclear regarding their designs and operating processes. Exhaustive research of the sync protocol of specific frameworks can be time-consuming to determine the cause of sync difficulty and maybe inefficient [16].

Furthermore, while some existing services attempt to integrate multiple capabilities to increase sync speed and efficiency in mobile/wireless environments, whether these strategies are viable is still unclear [11].

Ultimately, as a mobile cloud storage system will need storage and network technologies, storage techniques must be flexible and function effectively in a mobile/wireless environment. Communications in such environments are vulnerable to high delay or interruption due to mobility and changing channel conditions [17].

Although several mobile sync frameworks support mobile customer data replication and management systems, they lack support for large artifacts (more than 10MB to Gigabytes) [11] [17] [18] [19] [20] [21]. The main observation from the literature study in the papers [22] [23] [24] revealed that out of 19, only nine frameworks (47.36 percent) support large objects, including commercial frameworks; additionally, few have limitations (in terms of maximum file upload size, chunking support option, and handling techniques for better performance of large objects). In the case of local storage and updates on the cloud and on other client mobile devices, managing large data and maintaining consistency becomes difficult.

Deduplication techniques, in particular, do not always lead to sync efficiency by reducing redundant data transfers. Reasonable attempts to implement delta encoding algorithms are hampered by the distributed nature of storage infrastructure and may lead to high overhead traffic due to the lack of incremental sync. When synchronizing files across a slow network is necessary, the iterative synch scheme suffers from low throughput [16].

To tackle the above challenges, Chunking, Segmentation, and Compression techniques are suggested to improve the sync performance for large objects in modern mobile cloud storage systems, focusing on Data management for large objects.

III. Related Work and Large Object Handling Techniques in Frameworks

Cloud storage providers use different techniques to optimize the storage space and speed up data transmissions [10]. The main techniques used in different synchronization frameworks are Chunking, Bundling, Segmentation, Compression, Deduplication, and Delta-encoding.

1) Chunking: For each piece of the large file uploaded to the Server, some frameworks will break up the upload into multiple pieces and upload the parts one at a time. The process of dividing the file into several smaller files or sections is called file chunking.

2) Bundling: During uploading multiple files together, some frameworks of cloud providers combine them into larger bundles for the sake of efficiency before storing them in the cloud. In file bundling, the transmission latency is reduced because cloud servers have fewer connections to mobile clients.
3) Segmentation: This method consists of creating a file that takes in a large object and divides it into smaller, self-contained portions. Because Segmentation allows for virtually unlimited segment uploads in a single object with faster and parallel segment uploads, it is a favorite technique used by Internet service providers and application developers.
4) Compression: Prior to transmitting to the cloud, the file data can be compressed. With a bit of overhead of processing, data compression can minimize the traffic and reduce storage requirements [11].
5) Deduplication: In the case of an identical copy of a file being uploaded by the user or another user in the cloud storage service, that file can be deduplicated. In such cases, instead of sending the file over the network again, it maintains only a unique link to avoid network traffic and reduce storage requirements. Some frameworks support chunk level deduplication if the provider's data storage unit is a chunk object rather than a file.
6) Delta-encoding: This process transmits only the modified portion of a file with Compression. If the previous version of the file already exists on the server, the transmission will contain only the modified parts of the file compared previous version.

A. Analysis of Large Object Support in Frameworks

For mobile cloud services, supporting large file uploading and retrieval is critical as data sizes of sharing content are increasing, and mobile users are accessing or sharing enormous size files [9] [8]. However, only Desktop customers and not mobile apps can use valuable large object services. Open-source and commercial cloud storage services for mobile devices are analyzed for large file object support. The studies are mainly classified into three categories: open source frameworks (Parse Server [25] , BaasBox [21], Simba's [11] and Open Data Kit 2.0 [26]), academic research reference frameworks (SwiftCloud [18], QuickSync [16]) and commercial mobile cloud frameworks (Dropbox [27], Google Drive [28], Amazon Dynamo [29], CloudKit [30]).

Table II in the Appendix summarizes the large object support and techniques used to optimize the storage space and speed up data transmissions in the various reference frameworks. Many commercial cloud-based frameworks support large objects, but not every framework addresses large files, unfortunately.

Further investigation on frameworks revealed the various techniques of handling large objects and the maximum files supported with options of resuming uploads if interrupted due to network disconnections. A brief explanation of each framework with respect to handling large files is given below.

In the category of open-source reference frameworks, the Parse Server [25] service platform uses MongoDB as the backend datastore. It can only handle files up to 10MB in size. The *'ParseFile'* is a particular data type that makes it easier for Developers to store application files in the cloud. Parse provides another option with a data type known as *'PareObject'* to upload an array of up to 10MB in bytes or as a series of Streams. Implementing the *'SaveAsync'* method, the users can save the file to the Parse framework.

Another MBaaS open-source framework based n the Play framework but does not support large files by default is BaasBox [21]. However, based on the component design architecture, some custom implementations are needed to support uploading and downloading data up to 300MB. It is built on the Play web application framework, which is a lightweight, stateless, open architecture. BassBox requires configuration modifications for the maximum payload size in POST operations. By default, the value of POST request size is 100KB, which can be changed based on the server configuration. Special HTTP requests known as Body parsers in the Play framework are used for POST or PUT operations.

Simba [11] is a recent framework aimed to expedite the development and deployment of data-centric mobile apps and enable them to store data into the cloud storage. Simba extended the table interface of Izzy [17] but the sync protocol of Simba [11] does not support streaming APIs that can handle large files like Media or Video.

SwiftCloud [18] is another middleware framework that is based on the technique of Conflict-Free Replicated Data Types (CRDTs) with the Riak [31] key store and does not support storing objects over 50MB for performance reasons.

Mobius [7] is an application platform that enables real-time cloud-based data replication and messaging for mobile devices. Mobius is focused on addressing the development challenges of data management and messaging for data-centric mobile apps and does not deal with large files. Also, large size handling is not considered in the case of special CRDT cloud types in libraries like TouchDevelop [32] [33]. On the other hand, high-level ideas of sets and maps enable support for large files, but the storage providers do not support the storage of large objects (greater than 50 MB) for performance reasons.

Open Data Kit 2.0 [26] supports the Android operating system and enables the data to be stored in the cloud and handle offline data management. The default size limit on remote procedure calls in Android service is 1MB. To overcome the limitation of 1MB, the ODK Kit exposes higher-level features using a transport-level interface to developers. Using a client-side proxy, the ODK Kit implements a chunking interface.

Dropbox [27] is a commercial sync service provider, and files uploaded via the dedicated REST APIs can be up to 150MB in size. There is a maximum file size of 150 MB only when uploading with the files put API. Dropbox exposes chunked upload API to upload large chunks of data. A chunk of any size between 150 MB and 4 MB can be chosen. Dropbox has a built-in support to resume uploads if the upload is affected because of network disconnections.

QuickSync [16] is a framework that focuses on addressing the sync performance issues considering the network conditions. The framework integrate Seafile/Dropbox APIs and allows large data up to 180MB. It allows the big files to

be uploaded using the "chunk" API support. The APIs also supports automatic presumable data upload. A chunk of any size between 150 MB and 4 MB can be chosen.

Google Drive [28] is another file sync service provider that provides SDK to upload/retrieve data to/from cloud storage. To achieve resumable uploads, more than 5MB of data can be uploaded by making one request or via multiple requests with Google Drive SDK. In order to make the upload as fast as possible, larger chuck sizes are recommended. The upload request in Google Drive recommends chunk size to be in multiples of 256 KB.

Apple has extended the service of iCloud [32] with Cloud Kit [30], a new way of storing and accessing data stored in iCloud storage. The iCloud storage allowance is dependent on user type (paid or premium subscriptions allow larger storage). There is no explicit description document size limitation and Core Data (iOS local) storage limit. However, uploads depend on the device's or iCloud user storage limit. The operating system (iOS) starts and manages the upload and download of data from devices on the iCloud account. The iCloud app only needs to adopt the lifecycle of document management and need not communicate directly with iCloud servers. There is no need to invoke data upload or download operations in most cases.

Amazon uses DynamoDB [29] as the backend data store for cloud storage, focusing on high availability. Mobile SDKs from Amazon provide the way to interact with cloud services via REST APIs for DynamoDB. DynamoDB allows safe update operation of data even in the face of network partitions or server failures. In the DynamoDB, the total maximum item size is 400 KB, which includes both the name attribute's binary length and value. Suppose the application requires more storage space than that allowed by the Amazon DynamoDB limit. In that case, the developer may try compressing large attributes, or the app may store data in AWS Simple Cloud Storage (S3) [33] and associate the Amazon S3 ID with the Amazon DynamoDB entity using the Amazon object identifier.

The key observation from the study is that out of 19, only nine frameworks (47.36%) support large objects, and a few also has limitations. Because current Mobile Sync frameworks do not support large objects and have restrictions (in terms of maximum file upload size, chunking support, configuration, and large object handling strategies for better performance), this article provides an enhanced cloud-based Mobile Sync framework.

IV. OpenStack Swift and Support for Large Objects

OpenStack platform (developed by NASA and Rackspace) offers a combination of open-source tools for the management of the core cloud computing services in the areas of computing, identity, storage, networking, and image services. This platform can be customized and integrated with additional packages based on the requirements. The framework proposed in this paper is based on Swift, the object storage of OpenStack. The primary design hierarchy of Swift is based on a *tenant/container/object* structure to efficiently, safely, and cheaply store files.

The flow of request processing in Swift is shown in Fig. 1. When a client submits a request to retrieve an object A, an intermediary proxy server retrieves the object. The proxy is regarded as a stateless single entry point to the storage cluster and allows it to scale to arbitrary clients. The proxy is also responsible for determining the appropriate object server for the client's requested object and eventually returning the response object to the client. Apart from the object catalog, the cluster has a running container catalog that stores data about objects grouped within containers. Alternatively, hash functions are used to locate containers. Swift is based on the Web Server Gateway Interface (WSGI), which allows frameworks to define a pipeline. This pipeline comprises one or more middlewares that can pre-process requests before they reach the main web server component.

With regard to the enterprise architecture, Objects correspond to files and are arranged in Containers, i.e.,directories. Tenants form the highest level of hierarchy to set up an organization assigning a set of containers. Swift defines two types of ACLs: tenant-level and container-level. A tenant-level ACL allows administrative access to the tenants. Container level ACLs define the permissions on the container for reading, writing, and listing. OpenStack Swift limits the association of any ACL for objects, and the ACL of a container applies to all objects in it.

This study recommended the techniques of Segmentation and chunking to deal with large object files. The principal goal is to attain faster reading and writing speeds using a low object-to-node ratio with a lesser number of objects having large chunk sizes. In addition to effective data reduction, the method uses effective bandwidth reduction techniques [11] [16].

Using the Chunking Mechanism, large objects can be split into smaller parts of a certain data unit when uploading data to cloud storage services without raising the resulting file size issue (in the user interface of memory-constrained devices). Hence when a user attempts to upload a large file, some frameworks divide the input file into smaller parts and then upload these smaller chunks asynchronously later. The proposed chunking mechanism can handle large object data management and end-to-to-end consistency in a mobile cloud environment [16].

It is also a common technique to decrease network traffic by using chunks [34]. The proposed Mobile sync framework in this article provides data structures that can accommodate both table and object data. A single Table ($Table_{DS}$) can contain several rows (Row*DS*). In the proposed cloud synchronology design, in the case of a *RowDS* with one or more objects, the changeset for the sync will only consider the modified chunks. Individual chunks are not versioned.

The technique of Segmentation consists of creating a file that takes in a large object and divides it into smaller, self-contained portions. These smaller segments are then transmitted as one object together. Segmentation allows a nearly infinite object size, with more segments that can be uploaded nearly simultaneously in parallel for quick transfer. The Open Stack Swift [35] supports large object uploads by utilizing this segmentation technique. Such a framework would provide cloud-based services with support for large files of any size, from megabytes to gigabytes, and allow the Developers to

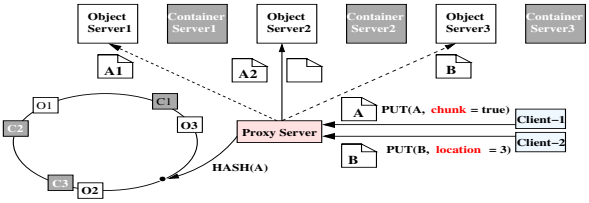

Fig. 1. Request Processing in OpenStack Swift.

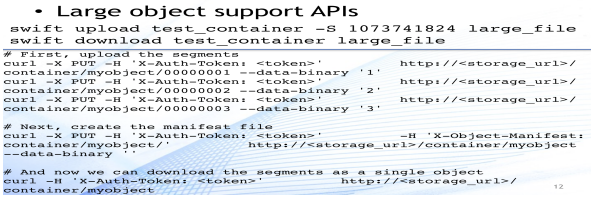

Fig. 2. Open Stack Swift APIs for Large Object Support.

arbitrarily complex, synchronized large objects to be built and maintained in the cloud.

Fig. 1 and 2 shows the sequence of request in segmentation process of OpenStack Object Storage. While the data is being uploaded, the mobile client can specify how much of it to chunk, and the proxy server will divide it into smaller sections. The different blocks are given internal names according to their position in the cluster. Creation of order lists all object file names also creates a manifest file (see Fig. 2). The proxy reads the manifest file to fetch the parts from the GET request of client. With the integration of OpenStack Object Storage, the proposed Mobile Sync framework can support files that are as large as 5 GB in size.

In addition to the techniques of Chunking and Segmentation, this research also incorporates Compression so that the data will be transmitted to the cloud in a smaller, more compressed form. Google's SPDY [36] with Google Protocol buffers [37] are employed, which uses multiplex to HTTP extensions to save on network overhead and perform better Compression for multiplexing requests over a single connection. Google Protocol buffers provide an expandable serialization mechanism for structured data, which are language and platform-neutral.

V. Proposed Framework for Data Services to Handle Large Files

A. Prototype Implementation

There are two main modules to the proposed architecture of the Mobile Sync framework: one for executing on the handheld device (Data Service) and the other for data in the cloud. Together, these two modules assist enables mobile application development with the Software Development Kit (SDK) provided by the framework. Fig. 3 shows the basic block diagram of the proposed Mobile Sync framework.

The *'Framework Data Service'* (FDS)is a module that connects the mobile apps and the Cloud Data Server by using a custom sync protocol and works as a mediator for transferring data and messages. Each client application is built with the

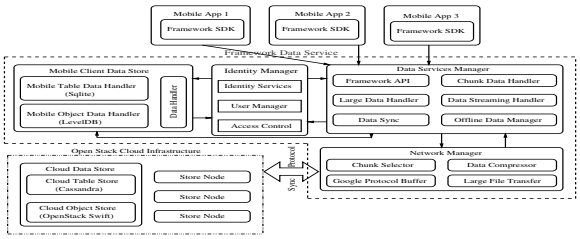

Fig. 3. Block Diagram of Proposed Mobile Sync Framework.

Framework DataService API provided by the SDK and talks to the system-wide data service (FDS) via streaming and CRUD APIs. Mobile Local database store consisting of Sqlite and LevelDB, enable FDS to save all the application data and metadata in the local store. For the Mobile client apps that run on mobile devices, the local database is not accessible directly since FDS manages it.

The primary modules used to manage large objects are *'Chuck Data Handler'* and handler modules like *'Large Data and Streaming Data'*, which are included within the framework SDK. These modules use a simplified data storage model for apps to use the chunking process proposed in this research article. *'Mobile Client Data Store'* is a local storage module to store tabular data and large app objects in the memory of a mobile device (typically the internal flash memory or the external SD card). On the client-side, the framework integrates SQLite for table and LevelDB for object data (in addition to handling large files).

The data model of the proposed Mobile Sync framework facilitates the storage of data for all client applications and hides the complex details of storing and synchronizing data. A custom Data Model consists of chuck layout support with additional support for larger objects.

The *'Cloud Data Store'* is the other main server-side module responsible for data management and interacts with the *'Framework Data Service'* (FDS) via the custom Sync Protocol. The primary responsibility of this module is to manage data across multiple mobile clients of the framework and implement the chunking & segmentation method at the server-side. For supporting large files in the cloud, through Object Storage, the framework uses the Infrastructure of OpenStack Swift, which has built-in support for the Segmentation process. The framework API must handle the request and responses from the OpenStack Swift server.

B. Framework APIs

The proposed Mobile Sync framework API is designed in the same way as the well-known CRUD interface, allowing apps to set Table/Object properties, access their data, push new data, and resolve conflicts. The framework provides a streaming API abstraction that allows objects to be written to or read from, making it ideal for dealing with large object. It also allows to read or write only a portion of a huge object locally. Table I lists only the APIs that are specific to chunk processing.

TABLE I. CHUNK PROCESSING APIS IN THE PROPOSED MOBILE SYNC FRAMEWORK.

API	Purpose
UploadObjectChunk[]	Creating chunk
GetChunkObjectRange[]	Retrieving chunk
RemoveChunkObject[]	Remove a chunk
RemoveChunks(table, chunkref)	Removing multiple chunks
UploadManifest[dataptr]	Creating large objects using Chunk
DownloadManifest[]	Retrieving large objects using Chunk
RemoveObjects[objptr]	Deleting large objects

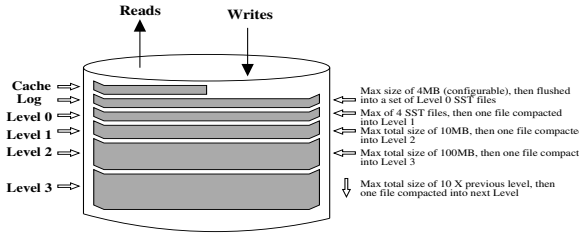

Fig. 4. Architecture of LevelDB with SSTable and MemTable.

C. Mobile-side Large Objects Handling Method through LevelDB with LSM

This proposed Mobile Sync framework integrates SQLite for the table data structure and LevelDB for the file object storage on the client-side. LevelDB is integrated into the SDK of the proposed Mobile Sync framework for handling large files. LevelDB uses an advanced data structure known as Sorted String Table [38] and Log-Structured Merge (LSM) [39] to handle large workloads with gigabytes of data. LevelDB's append and update performance meet the throughput criteria for the mobile client-side layer. LevelDB also has atomic snapshot in-like functionality, which is used for synchronization.

Sorted String Table (SSTable) [38] is a valuable and practical data structure for storing key-value pairs in large numbers and high throughput sequential access. SSTable offers flexibility for sequential read/write requests with workloads consisting of data sets that are Gigabytes in size.

The Log-Structured Merge (LSM) [39] architecture adds various new behaviors to the SSTable. Write operations are always fast no matter the size of the data set (append-only) because the LSM permits all write requests directly to the MemTable index. In addition, random reads can be obtained quickly or quickly served from memory (Initially search MemTable and then the indexes in SSTable). SSTable periodically flushes the MEMTables to the disk.

LevelDB architecture uses the SSTable and MemTable processing schemes to form an efficient database engine with powerful algorithms. Many other related products follow the same architecture include Apache Cassandra, Google BigTable, and Hadoop's HBase.

Fast write operations are allowed in LevelDB regardless of the data-set size, as all write operations are directly executed to the log and the MemTable. The logs of up to 2MB are periodically written to disk assorted string table files (SST) into a database. Each piece of SST data is compressed into single-writable 4K sections. Entries are positioned so that an end-marker block designates the beginning of each data set, and the most recently processed section of the list points to the start of the next. Bloom filters perform lookups more quickly and enable fast search of indexed blocks.

For an improved reading speed, LevelDB breaks SST into sets or levels (see Fig. 4). Each level in LevelDB has ten times the size of the previous one.

D. Server-side Large Object Support with Segmentation

This proposed Mobile Sync framework integrates Cassandra [40] for the table data structure and OpenStack Swift [35] for the file object storage on the cloud server-side.

OpenStack Swift [35] employs the process of Segmentation to enable large file uploads. The goal of the Segmentation process is to create a single file that divides the object into segments. With Segmentation, it is possible to upload a single object of virtually any size while taking advantage of the double uploads and the ability to upload multiple segments in parallel.

Unless the size of an object exceeds the maximum value (5 GB) set for the Swift cluster, each object is considered as a single file and stored in the disk. This restriction of the maximum file size of 5 GB avoids one object taking up all of the storage while half of the disk is empty. If the item to be stored is enormous, it is typically stored in several segments to allow future reassembly.

VI. RESULTS AND DISCUSSION

The proposed Mobile Sync framework is designed for use in large file sync scenarios, from a couple of hundred megabytes up to several gigabytes. Multiple modules serve both the clients and the servers from in same architecture.

The evaluation process discusses the performance of the proposed Mobile Sync framework for the following factors:

1) The performance of Application Programming Interface (API) consisting of Create, Read and Delete queries and chunking operations [11] [41] [42]
2) The efficiency of Sync Protocol [43] [44] [45].

A. Efficiency of Chunking and Data Access APIs

The Application Programming Interface (API) of the proposed Mobile Sync framework for cloud storage is designed to handle requests from thousands of mobile clients. For evaluating the performance of the cloud storage interface, a Linux Test client is implemented. The prototype included a test application to issue requests of subscriptions for reading or writing to a table data structure exposed by the SDK by generating a configurable number of threads. The test application will then generate required read/write (**I/O**) requests. Both file object and tabular data sizes can be configured as per the requirement during the API testing. The chunk size for objects and consistency scheme is also configurable.

A series of tests are conducted with an individual API call. A combination API invokes the Create-Read-Delete to carry out the performance testing of exposed APIs from the proposed Mobile Sync framework. Test files ranging in varying sizes from 1MB up to 1 GB are generated for testing. A test

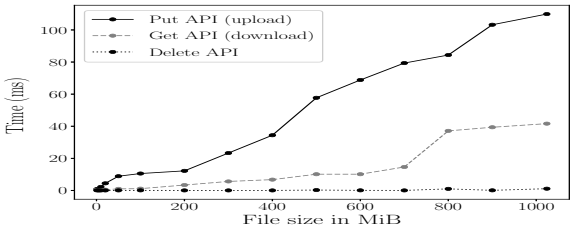

Fig. 5. Latency for Put, Get and Delete APIs in Proposed Mobile Sync Framework.

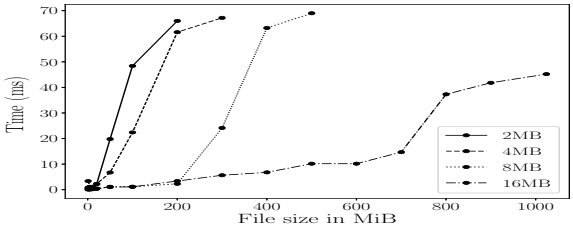

Fig. 6. Latency for Upload Operation with Different Chunk Sizes in Proposed Mobile Sync Framework.

suite is created for large file reading to analyze if the Mobile Sync framework can handle 1MB to 1GB of data. In addition, testing is also done to delete individual files. The Create-Read test is done with 8MB of the default chunk size. A graphical representation in Fig. 5 shows the latency for Put, Get and Delete APIs in the proposed Mobile Sync framework.

As illustrated in Fig. 5, latency increases with the file size. Upload APIs also took longer than download APIs because the upload operation includes data acknowledgments and processing time. It is observed that the time variation of upload data for a size greater than 500MB is less since the framework sync protocol employs data compression during the optimized transfer of data in the network.

The Delete operation is quicker Mobile Sync framework, as the operation completes instantly by marking objects as deleted, instead of removing them. The objects are permanently removed in the scheduled deletion cycle through configuration. The experimental evaluations show that the proposed Mobile Sync framework APIs can handle reading, and removing large files with excellent efficiency.

The third measurement evaluated the performance of object chunking. A method for modifying the object-to-to-node ratio is achieved by altering the testing file size with varying chunk size [11] [41] [42]. A large file size with a larger chunk size results in a low object-to-node ratio, enabling fast reads and writes. Fig. 7 illustrate the impact of configuring different object chuck size (2MB, 4MB, 8Mb and 16MB) on Upload or Put query for proposed Mobile Sync framework. Having a large chunk size demonstrated to be more effective since the large chunks only require a few partitions but can transmit more data efficiently and quickly. For the proposed Mobile Sync framework chunk size of 16MB is recommended subjected to the

B. Sync Protocol Efficiency

The main goal of the proposed Mobile Sync framework is the efficient synchronization of mobile device data through high-level abstractions. It is , therefore, essential to ensure that the synchronization process is not significantly affected by overhead by the proposed framework. So, it has to be shown that the Sync protocol of the Mobile Sync framework is lightweight. To do so the overhead of synchronization of rows with 1 byte of table data with six scenarios is calculated as follows: First with no data, second with one-byte data, and four other cases with 64 KB, 200 KB, 100 MB, and 300 MB objects. In order to minimize compressibility, random bytes are produced for the payload.

Fig. 7 and Fig. 8 show the overhead of sync protocol for a single message with 1 row and ten rows, respectively, for different payloads.

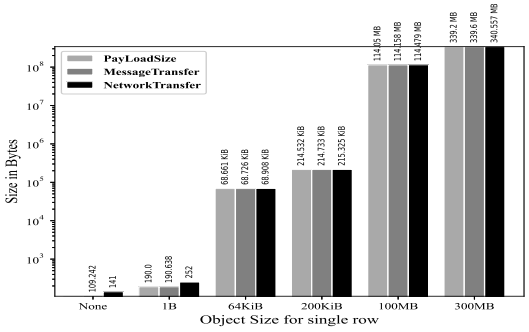

Fig. 7. Overhead of Sync Protocol for a Single Message with 1 Row with Different Payloads.

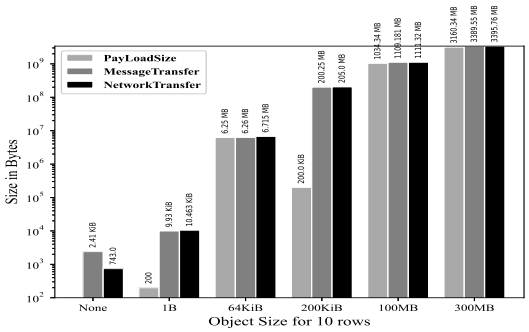

Fig. 8. Overhead of Sync Protocol for a Single Message with 10 Rows with Different Payloads.

The test results indicated that the Sync Protocol of the Mobile Sync framework produces a total message overhead of approximately 109 bytes. There is no object in this request but a single row with 1 byte of tabular data. There will be a reduced overhead for per-row baseline requests with the integration of data compression and batch operations of 10 rows turning into one sync request. Thus, with an increase in the payload (table or object) size, the data transfer overhead ultimately becomes negligible.

To sum up, the network overhead is reduced for the batched row or multiple rows operations because of data compression used in sync protocol. By working with single rows instead of

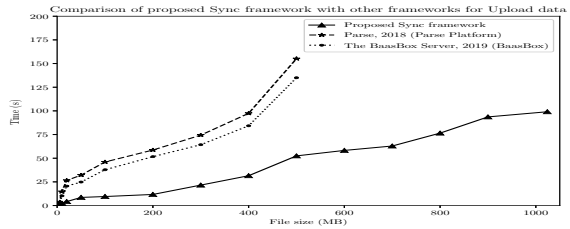

Fig. 9. Comparison of Proposed Mobile Sync Framework with other Frameworks for Upload Data.

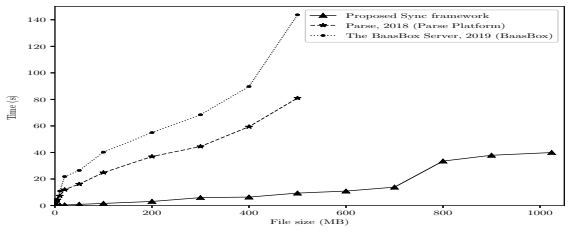

Fig. 10. Comparison of Proposed Sync Framework with Other Frameworks for Download Data.

batches, the Mobile Sync framework incurs a little overhead. Hence the sync protocol is lightweight and efficient with batching or group operations. More or less, empirical testing supports the claim that synchronization protocol is lightweight.

C. Performance Comparison with Other Sync Frameworks

The proposed Mobile Sync framework local performance is compared with two other open source mobile frameworks namely ParseServer [25] and BaasBox [21]. A file size of up to 500MB is tested in both Parse Server and BaasBox with the prebuilt virtual machine setup available. Some custom changes are required in Parse Server and BaasBox to support files upto 500MB .

The proposed Mobile Sync framework local performance is compared with two other open-source mobile frameworks, namely ParseServer and BaasBox, with a chunk size of 16MB. Fig. 9 shows latency comparison of Upload API (Put query) for the proposed Mobile Sync framework. The BaasBox upload process is handled by the REST APIs in the Play framework and takes more time than the proposed Mobile Sync framework due to the time taken for HTTP buffer processing and acknowledgment. Also, BaasBox does not support dedicated APIs through the Play framework to handle large files. BaasBox seems to be better in efficiency than the Parse Server framework. Overall the proposed framework reduces synchronization time with object chunking by 65.4% for upload on average when compared ParseServer and BaasBox.

Fig. 10 shows latency comparison of Get API (download query) for the proposed Mobile Sync framework, Parse Server, and BaasBox. The performance of Parse Server is comparatively better than BaasBox for files up to 500MB, and it should be noted that the experiment for download run on mobile with only a single file and no other application running on the device. The latency, which is the measure of response time between the device and a service's Server, must be considered. Since the BaasBox, Parse Server, and proposed Mobile Sync framework run on the virtual network setup in the testing network, the performance is better. The proposed Mobile Sync framework runs better than Parse Server and BaasBox since while downloading, the tests are configured with a chunking feature of 16MB to retrieve data in the device. The download tests are also dependent on the memory available for downloading and processing in the client device, depending on the RAM size. Overall the proposed framework reduces synchronization time with object chunking by 93.7% for download on average when compared to ParseServer and BaasBox.

Delete API performance in the proposed Mobile Sync framework and other two frameworks Parse Server, and BaasBox is almost identical. These frameworks follow a lazy deletion policy wherein objects are marked deleted and physically removed after the completion of the sync operation.

VII. Conclusion and Future Work

It is essential to develop Mobile applications to store and access data from backend enterprise systems. Certain usage patterns require storing and access data in large files. Uploading or downloading large files is a complex and time-consuming process for developers because of difficulties with latency, bandwidth, speed, errors, and disruptions to service in wireless mobile environment. In this article different techniques (Chunking, Bundling, Segmentation, Compression, Deduplication and Delta-encoding) for large objects (ranging from hundreds of megabytes to 5GB) in different mobile cloud storage solutions are analyzed. A Mobile Sync framework is proposed with both flexible chunking support and high throughput feature with segmentation technique to store and access large objects for a mobile cloud storage framework. The prototype implementation of the framework supported upload or download larger objects from the cloud storage, with support for tunable chunking configuration at the mobile side and local caching or data transfer only for a part of large objects. The extensive evaluations under the representative data-set show that the Mobile Sync framework works can quickly and effectively store large files and effortlessly keep minimal traffic burden on large workloads with reduced synchronization time. As a future work it is desired to extend the Mobile Sync framework with support for consistency schemes (like Eventual, Strong, Sequential consistency and others).

References

[1] P. Mell and T. Grance, "The nist definition of cloud computing recommendations of the national institute of standards and technology," *Nist Special Publication*, vol. 145, p. 7, 2011.

[2] M. Satyanarayanan, "Fundamental challenges in mobile computing," *Annual ACM Symposium on Principles of Distributed Computing*, pp. 1–7, 1996.

[3] A. Gheith, R. Rajamony, P. Bohrer, K. Agarwal, M. Kistler, B. W. Eagle, C. Hambridge, J. Carter, and T. Kaplinger, "Ibm bluemix mobile cloud services," *IBM Journal of Research and Development*, vol. 60, no. 2-3, pp. 7–1, 2016, doi:https://doi.org/10.1147/JRD.2016.2515422.

[4] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, p. 51, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=564585.564601

[5] N. Agrawal, A. Aranya, and C. Ungureanu, "Mobile data sync in a blink," in *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, 2013.

[6] H. Wu, L. Hamdi, and N. Mahe, "Tango: a flexible mobility-enabled architecture for online and offline mobile enterprise applications," in *Mobile Data Management (MDM), 2010 Eleventh International Conference on*. IEEE, 2010, pp. 230–238.

[7] B.-G. Chun, C. Curino, R. Sears, A. Shraer, S. Madden, and R. Ramakrishnan, "Mobius: unified messaging and data serving for mobile apps," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012, pp. 141–154, doi:https://doi.org/10.1145/2307636.2307650.

[8] F. Shanon Montelongo, "How to upload large files," https://blog.filestack.com/thoughts-and-knowledge/how-to-upload-large-files/.

[9] Z. Li, X. Wang, N. Huang, M. A. Kaafar, Z. Li, J. Zhou, G. Xie, and P. Steenkiste, "An empirical analysis of a large-scale mobile cloud storage service," in *Proceedings of the 2016 Internet Measurement Conference*. ACM, 2016, pp. 287–301.

[10] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside dropbox: understanding personal cloud storage services," in *Proceedings of the 2012 ACM conference on Internet measurement conference*. ACM, 2012, pp. 481–494, doi:https://doi.org/10.1145/2398776.2398827.

[11] D. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H. V. Madhyastha, and C. Ungureanu, "Simba: Tunable end-to-end data consistency for mobile apps," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 7, doi:https://doi.org/10.1145/2741948.2741974. [Online]. Available: https://github.com/SimbaService/Simba

[12] Y. Bai and Y. Zhang, "Stoarranger: Enabling efficient usage of cloud storage services on mobile devices," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 1476–1487.

[13] T. Ketola, "Quantifying software development: Applying mobile monetization techniques to your software development process," in *2014 Computer Games: AI, Animation, Mobile, Multimedia, Educational and Serious Games (CGAMES)*. IEEE, 2014, pp. 1–4.

[14] A. Inc, "ios app ota limit in cellular network," https://github.com/baasbox/baasbox/, accessed: 2021-05-01.

[15] Google, "Reduce your app size," https://developer.android.com/topic/performance/reduce-apk-size, accessed: 2021-05-01.

[16] Y. Cui, Z. Lai, X. Wang, and N. Dai, "Quicksync: Improving synchronization efficiency for mobile cloud storage services," *IEEE Transactions on Mobile Computing*, vol. 16, no. 12, pp. 3513–3526, 2017, doi:https://doi.org/10.1109/TMC.2017.2693370.

[17] S. Hao, N. Agrawal, A. Aranya, and C. Ungureanu, "Building a delay-tolerant cloud for mobile data," in *2013 IEEE 14th International Conference on Mobile Data Management*, vol. 1. IEEE, 2013, pp. 293–300, doi:https://doi.org/10.1109/MDM.2013.43.

[18] N. Preguiça, M. Zawirski, A. Bieniusa, S. Duarte, V. Balegas, C. Baquero, and M. Shapiro, "Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine," in *2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops (SRDSW)*. IEEE, 2014, pp. 30–33.

[19] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro, "Putting consistency back into eventual consistency," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 6, doi:https://doi.org/10.1145/2741948.2741972.

[20] Parse, "Parse," 2016, http://parse.com.

[21] "The baasbox server," https://github.com/baasbox/baasbox/, accessed: 2019-01-26.

[22] Y. P. Faniband, I. Ishak, F. Sidi, and M. A. Jabar, "A review of data synchronization and consistency frameworks for mobile cloud applications," *INTERNATIONAL JOURNAL OF ADVANCED COMPUTER SCIENCE AND APPLICATIONS*, vol. 9, no. 12, pp. 601–611, 2018.

[23] ——, "Netmob: A mobile application development framework with enhanced large objects access for mobile cloud storage service," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 7, 2019. [Online]. Available: http://dx.doi.org/10.14569/IJACSA.2019.0100784

[24] ——, "Enhancing mobile backend as a service framework to support synchronization of large object," in *Proceedings of the 2017 International Conference on Information Technology*, ser. ICIT 2017. New York, NY, USA: ACM, 2017, pp. 383–387, doi:http://doi.acm.org/10.1145/3176653.3176719.

[25] P. Platform, "Parse platform," 2016, https://parseplatform.github.io/.

[26] W. Brunette, S. Sudar, M. Sundt, C. Larson, J. Beorse, and R. Anderson, "Open data kit 2.0: A services-based application framework for disconnected data management," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2017, pp. 440–452, doi:https://doi.org/10.1145/3081333.3081365.

[27] Dropbox, "Build your app on the dropbox platform," 2016, https://www.dropbox.com/developers.

[28] G. Drive, "Google drive," 2016, https://developers.google.com/drive/.

[29] "Amazon dynamodb - best practices for storing large items and attributes," https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-use-s3-too.html, accessed: 2019-01-26.

[30] A. Shraer, A. Aybes, B. Davis, C. Chrysafis, D. Browning, E. Krugler, E. Stone, H. Chandler, J. Farkas, J. Quinn *et al.*, "Cloudkit: structured storage for mobile applications," *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 540–552, 2018.

[31] R. Klophaus, "Riak core: Building distributed applications without shared state," in *ACM SIGPLAN Commercial Users of Functional Programming*. ACM, 2010, p. 14, doi:https://doi.org/10.1145/1900160.1900176.

[32] A. Inc, "icloud for developers," 2016, "http://developer.apple.com/icloud".

[33] A. W. S. Mobile, "Aws sdk," 2016, https://aws.amazon.com/mobile/.

[34] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, "Ivy: A read/write peer-to-peer file system," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 31–44, 2002, doi:https://doi.org/10.1145/844128.844132.

[35] "Openstack swift object storage service," 2018, http://swift.openstack.org.

[36] Google, "Research & Drafts — SPDY — Google Developers." [Online]. Available: https://developers.google.com/speed/protocols

[37] "Protocol buffers," https://developers.google.com/protocol-buffers/, accessed: 2019-01-26.

[38] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[39] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[40] A. Lakshman and P. Malik, "Cassandra: structured storage system on a p2p network," in *Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM, 2009, pp. 5–5, doi:https://doi.org/10.1145/1582716.1582722.

[41] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective." *Cidr*, pp. 134–143, 2011. [Online]. Available: http://www.cidrdb.org/cidr2011/Papers/CIDR11{_}Paper15.pdf

[42] L. Rupprecht, R. Zhang, B. Owen, P. Pietzuch, and D. Hildebrand, "Swiftanalytics: Optimizing object storage for big data analytics," in *2017 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2017, pp. 245–251.

[43] D. Bermbach, E. Wittern, and S. Tai, *Cloud service benchmarking*. Springer, 2017.

[44] M. Klems, D. Bermbach, and R. Weinert, "A runtime quality measurement framework for cloud database service systems," in *Quality of Information and Communications Technology (QUATIC), 2012 Eighth International Conference on the*. IEEE, 2012, pp. 38–46.

[45] D. Bermbach and S. Tai, "Benchmarking eventual consistency: Lessons learned from long-term experimental studies," in *2014 IEEE International Conference on Cloud Engineering*. IEEE, 2014, pp. 47–56.

[46] S. Burckhardt, "Bringing touchdevelop to the cloud," 2013, https://www.microsoft.com/en-us/research/blog/bringing-touchdevelop-to-the-cloud/.

[47] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood, "Cloud types for eventual consistency," in *European Conference on Object-Oriented Programming*. Springer, 2012, pp. 283–307, doi:https://doi.org/10.1007/978-3-642-31057-7_14.

[48] D. Bermbach, J. Kuhlenkamp, B. Derre, M. Klems, and S. Tai, "A middleware guaranteeing client-centric consistency on top of eventually consistent datastores." in *IC2E*, 2013, pp. 114–123, doi:https://doi.org/10.1109/IC2E.2013.32.

[49] "Kony mobilefabric," http://docs.kony.com/7_0_PDFs/sync/kony_sync_orm_api_guide.pdf, accessed: 2019-01-26.

[50] "Evernote system limits," https://help.evernote.com/hc/en-us/articles/209005247, accessed: 2019-01-26.

[51] Kinvey, "Kinvey baas," 2016, https://www.kinvey.com/.

APPENDIX

TABLE II. SUMMARY OF VARIOUS TECHNIQUES EMPLOYED TO SUPPORT LARGE OBJECTS IN DIFFERENT REFERENCE DATA SYNCHRONIZATION FRAMEWORKS.

Framework	Large Object Support	Chunking	Bundling	Segmentation	Compression	Deduplication	Delta Encoding	Open-Source
Parse Server [25]	✗	✗	✗	✗	✗	✗	✗	✓
BaasBox [21]	✗	✗	✗	✗	✗	✗	✗	✓
Open Data Kit 2.0 [26] *	✗	✗	✗	✗	-	✗	✗	✓
Simba [11]	✗	-	✗	✗	✓	✗	✗	✓
SwiftCloud [18]	✗	-	✓	✗	✗	✗	✗	✓
Indigo [19]	✗	-	✓	✗	✗	✗	✗	✗
Izzy [17]	✗	-	✗	✗	✗	✗	✗	✗
Mobius [7]	✗	-	✗	✗.	✗	✗	✗	✗
TouchDevelop [46] [47]	✗	-	✓	✗	-	✗	✗	✗
Middleware for client-centric consistency [48]	✗	-	✗	✗	-	-	-	✗
QuickSync [16]	✓	✓	✓	✗	✗	✗	✗	✗
NetMob [23]	✓	✓	✓	✓	✓	✗	✗	✗
Bluemix Mobile Cloud Service [3]	✓	✓	✓	✓	-	-	-	✗
Dropbox [27]	✓	✓	✓	✗	✓	✗	✗	✗
Amazon DynamoDB [29]	✓	✗	✗	✗	-	-	-	✗
Google Drive [28]	✓	✓	✗	✓	-	✓	✓	✗
iCloud with CloudKit [32]	✓	✓	-	-	-	-	✓	✗
Kony [49]	✓	✓	✓	-	-	-	-	✗
Evernote [50]	✓	✓	-	-	-	✓	✗	✗
Kinvey [51]	✓	-	-	-	-	-	-	✗