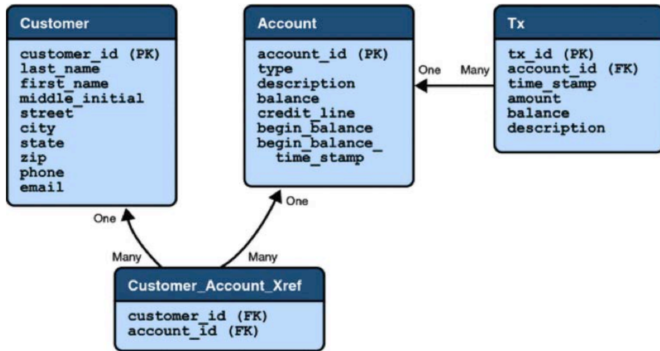
WHITE PAPER

NimbusBase: Technical Overview

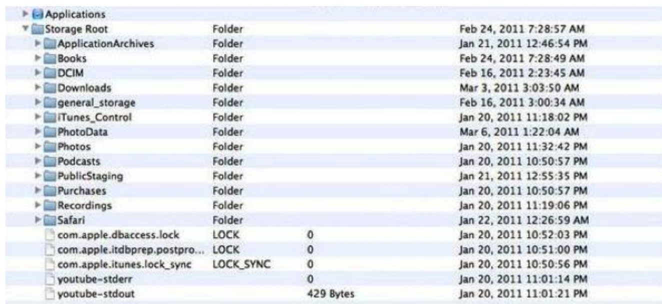
Understand the technology behind
NimbusBase

I. Storage of structured data on top of personal clouds

The main problem with storing structured data on personal cloud is that the data are in different formats. Mobile structured data is stored in the lines in a table format with linking relationships:



And personal clouds stores data like documents in a file system:



Name	Kind	Size	Date
Applications			
Storage Root	Folder		Feb 24, 2011 7:28:57 AM
ApplicationArchives	Folder		Jan 21, 2011 12:46:54 PM
Books	Folder		Feb 24, 2011 7:28:49 AM
DCIM	Folder		Feb 16, 2011 2:23:45 AM
Downloads	Folder		Mar 3, 2011 3:03:50 AM
general_storage	Folder		Feb 16, 2011 3:00:34 AM
iTunes_Control	Folder		Jan 20, 2011 11:18:02 PM
PhotoData	Folder		Mar 6, 2011 1:22:04 AM
Photos	Folder		Jan 20, 2011 11:32:42 PM
Podcasts	Folder		Jan 20, 2011 10:50:57 PM
PublicStaging	Folder		Jan 21, 2011 12:55:35 PM
Purchases	Folder		Jan 20, 2011 10:50:57 PM
Recordings	Folder		Jan 20, 2011 11:19:06 PM
Safari	Folder		Jan 22, 2011 12:26:59 AM
com.apple.dbaccess.lock	LOCK	0	Jan 20, 2011 10:52:03 PM
com.apple.itdbprep.postpro...	LOCK	0	Jan 20, 2011 10:51:00 PM
com.apple.itunes.lock_sync	LOCK_SYNC	0	Jan 20, 2011 10:50:56 PM
youtube-stderr		0	Jan 20, 2011 11:01:14 PM
youtube-stdout		429 Bytes	Jan 20, 2011 11:01:21 PM

The simplest way to store the structured data would be to serialize the entire data file for the database you are trying to back up into a single file. However, this means that each time you make a change in the data, you have to upload a large file with a small change inside. This is unacceptable as mobile networks are slow as bandwidth is an issue.

So the correct solution is to divide the data in some way so that each update to the data means a small upload or change to the cloud. At NimbusBase, we decided to take the smallest unit possible, a line of data in a table (Or a single instance of a data for a data model) and store that as a single file in the personal cloud. We did this by converting each line from its table format to a JSON file with the name being its id.

So a simplistic view of how NimbusBase work would be this: let's take a sample table of data:

Id	Name	Age	Gender
1	Ray Wang	30	M
2	Alex Volodarsky	26	M

The first line would be converted to a JSON file called "1.txt", with the content:

{uuid: "1", name: "Ray Wang", Age: "30", gender: "M"}

(Note, the real NimbusBase data model and conversion algorithm is much more complex.)

Now given a set of CoreData model or an sqlite database, the job of NimbusBase is to convert every single line of user generated database data, and also mirrors that perfectly as JSON files on a user's personal cloud.

II. Syncing data: keeping the mobile database and the personal cloud data files in sync

Now that we have a format to convert the data, the problem keeping the cloud and the device database in sync. If there were only one personal cloud account file system and one device, this would be easy. However, each user often has multiple devices, so the problem is syncing multiple devices with one cloud file system.

Some consideration we though of when syncing mobile device data:

1. We have to assume that the user has more than one device, all their devices will write to the cloud
2. The user is most likely only on one device at a time.
3. The user's device will be offline sometimes, when the device is offline, data will be created on it that will be synced when it's online

With these in mind, we create a sync algorithm that goes as follows.

A. There is two copies of the data, one local to the device, and one from the cloud.

B. Latest timestamp on a piece of data is always the most right data. We can know the "freshness" of the data on the cloud by it's last changed date on the file metadata, and the "freshness" of the data locally by the last changed timestamp in the database table.

C. Sync has two parts, we call them instant sync, and all sync.

a. Instant sync happens when the user is on their device and online, and they change a piece of data on their phone. That change is instantly pushed to the cloud. This will always be the latest data since the user is at most on one device at a time and current usage will mean the most recent data.

b. All sync happens when you have just entered the app or when you come back from being offline. It compares every single line of local data to every line of cloud data. We look at three timestamps to determine which copy of the data to move.
•Local_date: timestamp record stored in local database showing when it was last modified
•Cloud_date: timestamp file stored on cloud showing when it was last modified
•Sync_date: last date a local database record synced with the cloud records

With these three dates, we used the following algorithm:

If a record is only on the local database, and not on the cloud:
If sync_date <= local_date
//this means it's a new record created on
//this device
↑ Upload and create this record on the cloud
If local_date <= sync_date
//this means it's a record that was deleted by
//another device that is not deleted locally yet
↓ Delete the local record

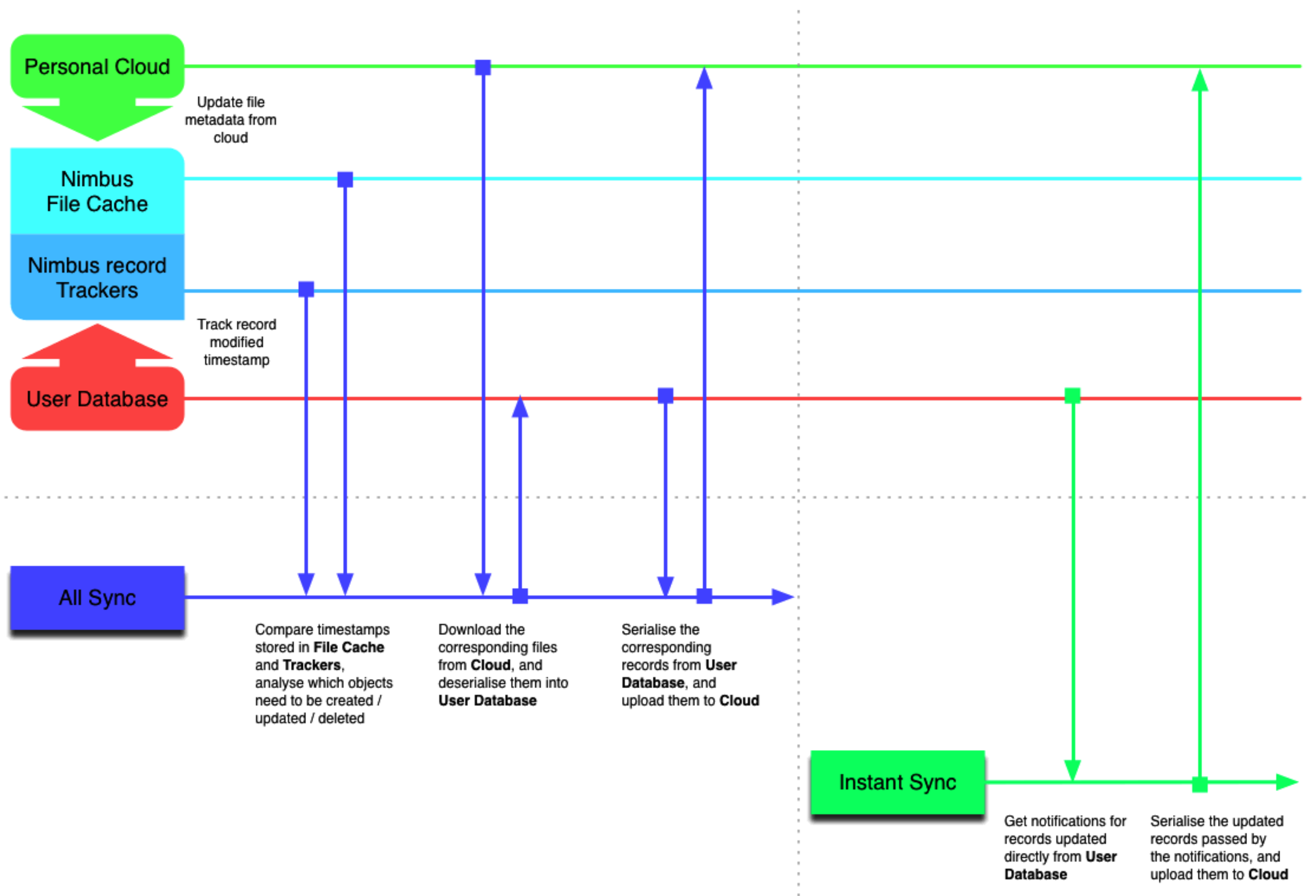
If a record is only on the cloud, and not in the local database
If sync_date <= cloud_date
//this means it's a new record created by
//another device
↓ Create the record locally
If cloud_date <= sync_date
//this means it's a record that is deleted
//on this local device that hasn't been
//deleted on the cloud yet
↑ Delete the record on the cloud

If the record is both on the cloud and the local database
If (local_date <= sync_date) and (cloud_date <= sync_date)
//since both dates are less than when the
//data has last synced, nothing should
//happen
Skip
If cloud_date <= local_date
//The local record has changed since it was
//last synced
↑ Update the cloud version with the local version
if local_date <= cloud_date
//The cloud record has changed since it was
//last synced
↓ Update the local version with the cloud version

As you can see, the last changed data can be determined by the combination of the three timestamps, and after determining the latest copy, we just have to copy that version to the other location.

I. Adapting the algorithm to mobile

So how did we adapt the algorithms described in I and II into an actual product? On the next we see a diagram of how the code actually runs:



All Sync:

On an actual mobile device, NimbusBase store two sets of records. The Nimbus File Cache mirrors the set of files that we have on the personal cloud. The Nimbus record tracker is our own metadata for the data in the user's database. Cloud_date comes from the first, and local_date and sync_date from the second. Thus we're able to make the comparisons we need to implement the algorithm for sync.

Once the correct copy of the data is determined, it is either serialized into JSON and uploaded to the cloud or deserialized and pulled down from the cloud to the user's database.

Instant Sync:

For instant sync, we already know that our data is the latest, so we just push it up to the cloud after serializing it.

Adapting the algorithm to iOS

For iOS, CoreData is the database. We read the data models that the developer has created and attach callback functions so that each time that the app changes, we call instant sync.

Adapting the algorithm to Android

Adapting the algorithm in Android is different. iOS uses CoreData, which is an ORM on top of SQLite. Android uses an SQLite database directly. This means we are accessing a more raw level of data and have to:

1. Read the structure of each of the SQL tables that a user creates
2. Use database trigger creation to get notified when a piece of data changes

IV. Migrations

One of the challenges that we face is migration. As you know, developers don't just create a single set of data objects and then never change it again in future versions. When we talked to developers after we released our iOS product, this was their first concern.

Given that a user can completely change the structure of their data models, we must account for things like:

A. Attribute changes; a user changes the name of a data model attribute or adds an attribute
B. Data model name changes. We need to match the old model with the new model so we don't lose all the data.
C. App changes on the database and the primary key of a record itself

There are two parts in dealing with the above changes. The first is detecting that the changes happened and the second is dealing with the change by changing the appropriate data records.

For detection, we can keep track of the current model version for a data model, and then when it changes, we can map the previous version to the new version.

Making the actual migration changes is more varied. For A and B above, we can change the current records to match the new records by updating the appropriate local or cloud records. For something on the scale of C, the only solution is to delete all local data files and repopulate it from the cloud.

So A and B becomes cases we have to really deal with in the case of migrations. Our strategy is this. To prevent mass change in cloud records and needless call of the APIs, we keep a dictionary of old column names for an object. When we encounter an old version of a data, we first translate the data column name to the current version, and then compare it to our current version. When we make changes, we update the columns name, but if not we do not have do a massive data change on cases of A and B.

About NimbusBase

NimbusBase provides tools to both application developers and personal cloud storage providers to make it as simple as possible to build apps on top of personal cloud. Our software allows developers to build more powerful, ubiquitous apps whose users will be able to control their data and access it anytime, anywhere. We believe that people today should have access to all their data, not just pictures or files but also app data, regardless of device or platform. NimbusBase is based in NYC and was founded in 2012.

Contact us at admin@nimbusbase.com or @nimbusbase
www.nimbusbase.com