CloudKit: Structured Storage for Mobile Applications

Alexander Shraer, Alexandre Aybes, Bryan Davis, Christos Chrysafis, Dave Browning, Eric Krugler, Eric Stone, Harrison Chandler, Jacob Farkas*
John Quinn, Jonathan Ruben, Michael Ford, Mike McMahon, Nathan Williams, Nicolas Favre-Felix, Nihar Sharma, Ori Herrnstadt
Paul Seligman, Raghav Pisolkar, Scott Dugas, Scott Gray, Shirley Lu, Sytze Harkema, Valentin Kravtsov, Vanessa Hong, Wan Ling Yih, Yizuo Tian

Apple, Inc.

ABSTRACT

CloudKit is Apple's cloud backend service and application development framework that provides strongly-consistent storage for structured data and makes it easy to synchronize data across user devices or share it among multiple users. Launched more than 3 years ago, CloudKit forms the foundation for more than 50 Apple apps, including many of our most important and popular applications such as Photos, iCloud Drive, Notes, Keynote, and News, as well as many third-party apps. To deliver this at large scale, CloudKit explicitly leverages multi-tenancy at the application level as well as at the user level to guide efficient data placement and distribution. By using CloudKit application developers are free to focus on delivering the application front-end and logic while relying on CloudKit for scale, consistency, durability and security. CloudKit manages petabytes of data and handles hundreds of millions of users around the world on a daily basis.

PVLDB Reference Format:
CloudKit team. CloudKit: Structured Storage for Mobile Applications. *PVLDB*, 11(5): 540 - 552, 2018.
DOI: https://doi.org/10.1145/3164135.3164138

1. INTRODUCTION

Many users have multiple devices, including phones, tablets, smart TVs, and wearable computers. Users expect to seamlessly switch from one device to another and have their latest content available. Many mobile applications (apps) support content sharing, collaboration, and always up-to-date access from any device.

Developing such apps is challenging. They must interact efficiently with backend servers, maintain a local cache for portions of the data and sync mutations between devices. Furthermore, successful mobile apps need to support many millions of users. This requires a scalable and highly-available backend. At the same time, to facilitate seamless cross-device experience, the backend has to provide strong consistency and durability. Furthermore, it has to provide adequate support for mobile clients including quickly bringing a client up-to-speed when it regains connectivity, selective content filtering, notifications (to avoid polling), and more.

*Email: shralex@apple.com

Proceedings of the VLDB Endowment, Vol. 11, No. 5

DOI: https://doi.org/10.1145/3164135.3164138

In an effort to accelerate and optimize the development of internal apps at Apple, we developed CloudKit: a platform that makes mobile app development simple and scalable. Launched more than 3 years ago, CloudKit [11] supports more than 50 Apple apps (as of Nov. 2017), including many of our most popular applications (e.g., iCloud Drive, Notes, Photos, Keynote, News, Backup, social gaming and many others), many more external apps and hundreds of millions of users on a daily basis (in Feb 2016, the number of iCloud users was 782 million [31]). Figure 1 shows the normalized number of Apple (first-party) and externally developed (third-party) apps adopting CloudKit since 2014. CloudKit allows rapidly developing the application logic, and obviates the need to solve boilerplate infrastructure problems. Many first-party apps don't have a dedicated backend team, relying completely on CloudKit.

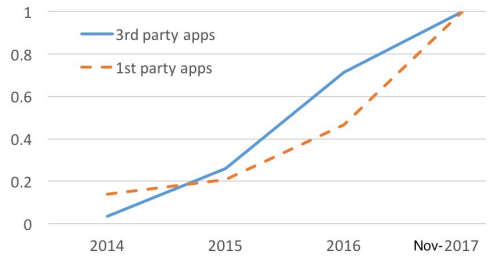

Figure 1: CloudKit adoption growth: number of apps using CloudKit normalized to their number as of November 2017.

In order to handle the combined scale of its client apps, CloudKit employs a unique approach that leverages multi-tenancy along two dimensions. First, each app operates in an isolated logical space called a CloudKit *container*. Within a container an application developer manages the app's schema, which can support a large number of record types and complex relationships. Second, within each container CloudKit uniquely divides the dataspace into many *private* databases that manage user-scoped data and one *public* database for common application data. A database instance is a self-contained unit that inherits the schema definition from the logical container and manages it's own data and indices. Each private database belongs to exactly one container and one user, and provides strong synchronization and serialization capabilities within the database. CloudKit backend is therefore free to distribute and balance the hundreds of millions of self-contained databases across the available infrastructure. Another advantage of this approach is that individual user requests target exactly one database (this user's private or the container's public database) allowing for localized execution and minimal traffic between a client device and the backend. For multi-user sharing, data is always owned by one of the users, whose private database maintains an

authoritative copy of the data. Sharing requires distributed coordination involving multiple databases.

A distinguishing feature of CloudKit is its support for schema management and evolution. CloudKit provides each container with separate sandbox and production environments, allowing developers to experiment with schema changes and deploy them to production with the push of a button using a dashboard. In the sandbox environment, CloudKit facilitates rapid app development by automatically inferring a schema when data is saved. In the production environment, CloudKit enforces the schema and allows strictly additive schema modifications, that are automatically propagated and inherited by each database instance of that container. This stands in contrast to other mobile backend solutions [17, 22, 24, 25, 28] which take a NoSQL approach: the app is free to write any data (e.g., JSON files) and schema management is left to the app developer. This is a daunting task since inevitably, schema changes need to be made, while avoiding forward and backward compatibility issues with data already stored and clients executing different versions of application code.

Applications usually cache data locally on devices and need to keep it in-sync with data stored by the backend, as it is modified by other devices and users. CloudKit provides change-tracking and a sync interface, used to keep their local state up-to-date (change-tracking can be selectively enabled for parts of the data-space). CloudKit also supports selective data filtering by exposing a flexible query interface, including full-text queries. Complementing both interfaces are subscriptions (including continuous queries) and push notifications which obviate polling and the need to configure or interact directly with push notification services.

In summary, this work makes the following contributions:

- CloudKit's unique data model and organization has proven useful for a wide range of applications, by serving as the backbone to some of the world's largest datasets.
- Through its separation of development and production environments, CloudKit manages and enforces application schema, without sacrificing rapid app development.
- CloudKit's change-tracking mechanism provides a powerful abstraction, leveraged by applications for both data and meta-data cross-device synchronization.
- To the best of our knowledge, this paper is the first to provide details of a production mobile backend system, that approaches the efficiency and synchronization problem with such explicit tools or that's operating at such scale.

This paper is organized as follows. Sections 2 and 3 describe CloudKit's data model, APIs and semantics. Section 4 describes common use-cases. Sync is described in Section 5. Section 6 explains multi-user sharing. Section 7 details the query interface, subscriptions and notifications. Section 8 describes schema management. Related work appears in Section 9. Section 10 includes production experiences and evaluation. Section 11 concludes the paper.

2. CloudKit DATA MODEL

CloudKit's novel data model, described in this section, was designed with the mobile use-case in mind. CloudKit faces a dual multi-tenancy challenge: it serves a very large number of apps, and hundreds of millions of users. CloudKit's data model represents both explicitly – each app has a dedicated container, and each user a dedicated database within the container. A key observation that motivates our design is that many apps have data which is accessible across users (e.g. news articles, maps, music), while other data is private (e.g., a user's settings, preferences, photos, docs, messages). CloudKit stores this data in designated *public* and *private* databases, respectively. The storage requirements for these two types of data are very different and hence the private and public databases have different capabilities. For example, private databases support stronger security and consistency semantics, change-tracking, sharing data with specific users, and more, while public databases are designed to be more scalable and serve many users concurrently. Unlike in traditional databases, all databases within one container share a single schema, which makes it easy to manage and evolve an app's schema. At the same time, the private databases of different users do not have to reside in the same physical location, and in fact are often moved to improve load-balancing, access locality, etc. In our experience, sharding on a natural boundary of a user's data allows for almost unlimited scalability.

The separation into private and public databases aids with security, privacy, and access control: Data stored in a user's private database is owned by the user and can only be accessed by client devices authenticated as the owner (or as another user with whom the owner shares the data through CloudKit), but not by other users of the app (or the app developer). Authentication is performed by other iCloud services, while access control is enforced by CloudKit. For many apps, a client device encrypts data prior to storing it in the private database, and only the owner's devices (or those with whom he shares the data) can decrypt it [9, 23].

Usage of the private database is counted against the user's quota; usage of the public database is billed to the app developer. Since most data resides in private databases, the vast majority of storage and processing costs are free for the app developers.

In the remainder of the section, we summarize the different constructs provided by CloudKit (see Figure 2 for illustration):

Containers. Usually, the data of one app is encapsulated in a single container (this includes information about the users – each user may expose different information to different apps). Data can be shared across applications (not a common use-case) by using shared containers. Containers are created by the developer using Xcode and managed via the CloudKit dashboard [12].

For privacy reasons, CloudKit makes sure that it is impossible for an application to correlate users across containers (of other apps), by having container-specific user identifiers. CloudKit converts such identifiers to an unique internal id for each user.

Databases. Each container is logically divided into three types of databases: a single public database, n private and n shared databases where n is the number of app users with an iCloud account. These databases are created by CloudKit automatically. Data stored in the public database is, by default, visible to all users of the app (only authenticated users can store data; security roles are used for access control). Each user has a dedicated private database, and no other user can store or access data in that database unless explicit sharing relationships are created. A shared database is a user's "window" into the private databases of other users.

Records. Records are the basic unit of storage. Each record is a dictionary of key-value pairs, called record fields. Fields can contain simple value types (e.g., strings, numbers, dates) or more complex types (e.g., locations, references to other records, and assets). A field can also contain a list of values of a certain type. Reference fields represent relationships between records while assets reference data stored externally (e.g., when a photo is uploaded by a user, only its metadata and URL are stored in CloudKit). A separate sub-system manages the creation and garbage collection of assets and generates asset references. CloudKit supports encrypted field values, available in the private database.

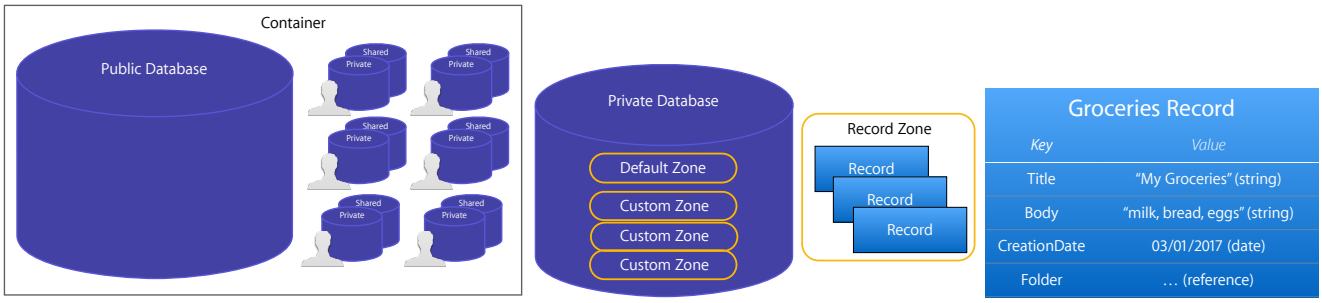

Figure 2: CloudKit data organization. On the right - an example of a record created by a groceries app.

Record Zones. Zones are a useful way to organize records into logical groups and enable an application to selectively sync subsets of data across devices. Each record belongs to one zone; zone name is a logical component of a record's full identifier. The public database has a single *default* zone. A private database contains a default zone but may in addition contain *custom* zones. Default zones provide a minimal set of features and are adequate for prototyping. Apps will generally evolve to use custom zones with additional capabilities. For example, custom zones support change-tracking, transactional multi-record batches and records may reference other records in the zone. Shared databases have custom zones, but no default zone (sharing is another feature specific to custom zones).

Environments and Schema. CloudKit provides strong schema management and enforcement, while at the same time enabling rapid app development. App developers define a schema containing record types, where each type consists of the identifiers and types of fields in the record. The schema is defined per-container – records of any type can be stored in any of the databases and zones in the container. Unlike traditional relational databases, where all rows in a table follow the same schema, the records of a zone can be of diverse record types. CloudKit provides separate development (sandbox) and production app environments. The development environment (available to app developers) is more flexible: the schema is inferred automatically when records are stored in the container. When the developer is ready to distribute the app, she uses the dashboard to migrate the development schema to production (this does not migrate the actual records). After the app has been deployed, the schema can still be modified in the development environment and migrated again to production (at this point, changes must be additive, e.g., fields and record-types can be added but not removed, though secondary indices, if defined on a field, can be removed). CloudKit propagates the app schema to all its partitions and enforces it on client operations. A detailed description of schema management appears in Section 8.

3. API, SEMANTICS, ARCHITECTURE

CloudKit APIs. CloudKit provides a rich set of CRUD APIs allowing clients to create, update, delete and fetch records and zones, list zones, upload and reference assets, get a list of changed zones in a database or changed records in a zone, listing users[1], sharing records, queries, subscriptions, notifications, and more. CloudKit exposes client-side libraries in Swift, Objective-C and JavaScript to support mobile and web developers. An exhaustive reference of server and client APIs is beyond the scope of this paper. APIs available to third-party developers can be found online [13, 14].

Dashboard. CloudKit provides a dashboard [12] where developers can view and manage app data, define secondary indices on record fields, view live server log events, visualizations, usage statistics (including errors), and more. The dashboard can also act as a web-client and issue requests, which is useful for debugging. These tools are provided for both the development and production environments, and the developer has additional options for resetting the development environment (erasing all data in that environment and making the schema match the current production schema, if any) and for deploying the schema to production (which modifies production schema to match the development schema).

[1]Each user determines what he shares with the app and other users.

Architecture overview. CloudKit supports three different interfaces, wrapping the same set of APIs (see Figure 3): a REST-like web interface (for web applications), gRPC [21] (mainly for other backend services using CloudKit) and a custom interface over TCP, used by mobile client apps through a client-side library and a daemon installed on devices. Usually, many applications on each device use CloudKit, and the daemon consolidates and manages all communication with the backend, as well as provides convenience APIs. With both gRPC and our client daemon, requests to the backend are encoded as Protocol Buffers [29].

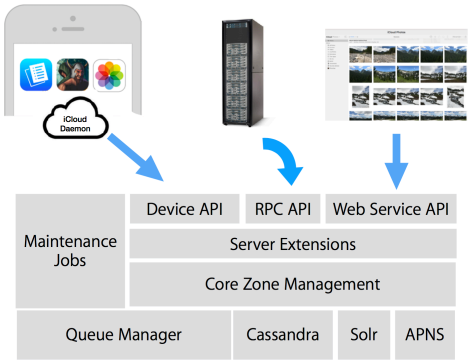

Figure 3: CloudKit architecture.

Server extensions are used for several major first-party apps, such as iCloud Drive, Photos and Backup and implement application-specific logic (e.g., conflict-resolution). Most apps using CloudKit, including many first-party apps, don't have custom server-side logic. CloudKit uses Apache Cassandra [5] as the underlying storage, and Solr [6] for indexing record fields queriable by users. Apple's Push-Notification Service [8] is used to notify users (queries, subscriptions and notifications are discussed Section 7). Asynchronous tasks are queued using a queue management system and processed by maintenance jobs. Examples include any necessary Solr index updates when a record is written, garbage-collection of expired or deleted records, and future action requests (e.g., when a user requests to delete his iCloud Photo Library, a grace period of 30 days is given, during which he can change his mind).

Data placement. Data is sharded into multiple CloudKit partitions, which constitute distinct failure domains. Each user is assigned to a single partition (partitions are replicated globally for high availability). Within each partition, there is a fixed number of stateless CloudKit JVM instances for each interface type (web, gRPC or device). Sharding by user makes this simple approach work reasonably well, though we are in the process of moving to a more dynamic deployment. We periodically migrate users from one CloudKit partition to another to improve load-balancing and access locality; currently, such migrations are infrequent and done in bulk for many users.

Each CloudKit partition has a dedicated Cassandra cluster, storing multiple *Cassandra partitions*. Within a Cassandra partition, Cassandra provides *light-weight transactions*, implemented using Paxos [38] and exposed as a compare-and-set (CAS) functionality [26]. CloudKit uses CAS to implement conditional updates, which provide a way to achieve lock-free synchronization of concurrent updates. The granularity of synchronization is different for custom and default zones, as we explain next.

Each custom zone is assigned to one Cassandra partition. This is done in order to leverage conditional and multi-key atomic updates provided by Cassandra within a partition[2]. Specifically, each custom zone has an update counter in Cassandra, and each update (to one or more records in the zone) additionally attempts to update this counter using CAS, which succeeds only if it was not concurrently updated by a different operation. This serializes all updates to the zone and allows supporting multi-record atomic operation batches. Although providing strong semantics, the assignment of a custom zone to a single Cassandra partition limits the size of the zone and does not utilize the whole Cassandra cluster. Default zones trade-off stronger semantics for scalability. They are sharded across multiple Cassandra partitions, which allows them to grow significantly larger than custom zones, but only provide single-record linearizable operations since Cassandra does not support cross-partition transactions. By default, the private database default zone is sharded into 10 Cassandra partitions while the public default zone is sharded into 10 thousand partition (some tenants, like Apple News, are sharded into many millions of partitions).

Intuitively, a private database is accessed by devices of a single user (or a small number of share participants), which are unlikely to request updates at the exact same time. Hence, zone-level synchronization where only one of the devices can make progress and the others retry is sufficient (we measure aborts due to CAS contention in Section 10.2). A public database, on the other hand, is accessed concurrently by many users and it is impractical to only allow a single client to make progress. The default zone (the only zone in the public database) allows many users to access data concurrently.

Read and update semantics. As already mentioned, all reads and updates of single records are atomic (linearizable) and custom zones further support multi-record atomic batches (including both reads and updates). Record reads optionally allow specifying the subset of record fields to retrieve. Record updates have one of three possible modes: *save-if-unchanged*, *save-changed-keys* and *save-all-keys*. With *save-if-unchanged*, an update is performed only if the record hasn't changed at the server since the last time it was fetched by this client (if the record wasn't previously fetched, the save succeeds only if the record does not exist). The check is performed using CAS on a version stored in the record, incremented by the server with every update to the record, and sent in operations and responses. The other update modes merge fields specified by the client and fields stored at the server, replacing updated fields. With *save-changed-keys*, the client sends only fields modified since the record was fetched, while *save-all-keys* sends all fields.

Note that since *save-changed-keys* updates only a subset of the fields, it is possible that the result is not what the client expected: other fields may have been updated concurrently by other clients. Even with *save-all-keys*, only fields that are sent by the client are updated. This is done to help with app versioning. It is possible that a client running a newer version of the app stores fields that are not known to old clients, and we want to prevent old clients from implicitly deleting such fields (explicit deletes are allowed).

While *save-if-unchanged* is the "safest" update mode, it provides weaker progress in the face of contention, guaranteeing that the update of one of the clients succeeds (i.e., lock-free / non-blocking progress). In cases of highly contended records it may be more desirable to merge fields in order for all clients to make progress (i.e., wait-freedom [35]). In that case, the client should make sure that the result of merging the fields is consistent (e.g., by sending some additional unchanged fields).

Reference semantics. Reference fields create a stronger relationship between records than just saving the identifier of a record as a string. Two examples are the *owning* and the *validating* references. With an owning reference, the target (referenced) record becomes the owner of the source record. Deleting the target record deletes all its source records, cascading further if these records themselves "own" other records. If a record contains two or more owning references, the record is deleted when any of its owners is deleted. A validating reference ensures that its target exists as long as the source record exists; creating the reference is only allowed if the target record exists, and deleting the target is not permitted as long as it's referenced.

Secondary indices. The dashboard allows developers to create indices on record fields, and CloudKit supports queries on indexed fields (see Section 7). Queries on record identifiers are backed by Cassandra and are therefore atomic; all other queries are backed by Apache Solr [6] and are eventually consistent. Some features leverage a secondary indexing mechanism that we developed in Cassandra, that provides filtering and ordering while guaranteeing atomicity. Such secondary indices consist of regular Cassandra records, whose key is the secondary key and value is a list of record ids with that key (in addition to other information, depending on the index). The records are split, if needed, to keep the value lists relatively short. These records are stored in the same Cassandra partition and updated transactionally with the data using Cassandra's multi-record atomic updates and CAS. Scanning the index is done using Cassandra's range scan. One example is the implementation of reference fields, where a secondary index maps each record identifier to the list of records that reference it. Another example is the implementation of short-lived records – an application can set (or change) a TTL on a record, CloudKit indexes records by expiration time and a maintenance job garbage-collects expired records by querying the index. Applications can configure whether expired records that were not yet garbage-collected are visible to users. Change-tracking, described in Section 5, is implemented using a secondary index on the value of the zone's update counter.

3.1 Conflict Resolution

A frequent use-pattern is one where a device comes online and syncs down new or changed records in a zone, bringing its local database up-to-date by incorporating the received changes. The device may also have local pending changes, not yet uploaded to CloudKit. Some of them may conflict with updates that happened while the device was offline. In this case, the app would detect and fix conflicting records when it syncs down changes. Then, pend-

[2]Cassandra's query language providing these features [15] was heavily influenced by CloudKit's requirements.

ing changes are uploaded to CloudKit. If concurrent updates are made by other devices, and the default *save-if-unchanged* mode is used, CloudKit detects the conflict based on record-version mismatch and an error is returned.

CloudKit does not offer a general conflict resolution functionality. The app is expected to handle the error, resolve any conflicts, and attempt to save the record again, if needed. CloudKit provides the app with three copies of the conflicting record to assist with comparing and merging the changes (commonly referred to as a three-way merge): (1) client record – a copy of the record the client attempted to save, (2) server record – a copy of the record as it currently exists on the server, and (3) ancestor record – a copy of the record fetched by the client, before any of the pending changes were applied. When a conflict occurs, the app should merge all changes onto the server record (this record has the up-to-date record version) and attempt to save that record.

Clients use various methods to resolve conflicts. Some use the simple "last writer wins" method, essentially forcing the update upon conflict by conditioning it on the newly received record version. Others use a deterministic method, such as "smaller device id wins". Text-editing apps using CloudKit often employ techniques such as Operational Transformation and CRDTs [42]. It is important to note that while CloudKit only detects update conflicts on individual records, app-level conflicts may span multiple records. It is up to the app to detect app-specific constraint violations and correct any illegal state by performing additional updates.

In many cases, client-side conflict resolution is preferable. For example, if record data (such as document text) is encrypted only an authorized device (and not the backend) can perform a content-based merge. In cases when the conflict cannot be resolved, end-user intervention may be required. In this case, an app would usually show an error message or pop-up a dialog UI and present the user with several options to resolve the conflict.

Nevertheless, there are several advantages for supporting server-side conflict resolution. First, resolving conflicts on the server reduces client-server communication. Second, different clients may be running different versions of the app software and arriving at a consistent client-side conflict resolution may be challenging. Finally, server-side conflict-resolution allows us to update the resolution logic without requiring a client software update.

For these reasons, CloudKit allows several first-party apps to run their conflict-resolution code in server extensions. This allows the app more flexibility to specify what other records and information it needs to resolve the error. In some cases, iterative resolution is performed by the extension. For example, in iCloud Drive when a client attempts to store a record representing a file and the file name conflicts with an existing file, the extension may attempt to rename the new file but in doing so may create a conflict with a different file, requiring additional resolution rounds. To facilitate server-side resolution, CloudKit stores conflicted records and server extensions are expected to clean-up such records after conflict resolution.

4. CloudKit USE PATTERNS

We identify five main use patterns for CloudKit:

Publish-Subscribe. In these apps, the backend or several users produce data, while others consume and query it. One example is Apple News – articles are written to CloudKit's public database and clients register query subscriptions based on their preferred topics and news outlets (in this case, the database is backed by a globally distributed Cassandra cluster). News uses the private database to save each user's preferences and sync them across its devices. Multiple apps use the public database to allow querying public information such as transportation time-tables and routes. Finally, some apps use it as a space where users can share content with each other, often encrypting it and using security roles for access control. Such apps could benefit from CloudKit's sharing features.

Cross-Device Sync. This is the prevalent use-case for CloudKit, leveraging the change-tracking capabilities of custom zones. For example, several first-party document sharing apps represent documents and folders as records in a user's private database. Document text is represented as a field value or as an asset referenced from the record. The content is kept in-sync on all user devices that can jointly edit, subscribe to change notifications and receive state updates. Another example is iCloud Photo Library, which represents photo metadata as records in a user's private database, while the photo itself (and derivatives, such as thumbnails) are assets referenced from the record. A much simpler example is an RSS-like app where users consume a feed of events, and CloudKit is used to share the feed cursor across the user's devices. Finally, a recent example is an Apple app that keeps the list of paired bluetooth devices (e.g., AirPods, mice) in-sync, so that a user can pair the device once (e.g., with his iPhone) and have it automatically pair with the other devices (such as his iPad and MacBook).

Sharing and Collaboration. A relatively recent feature, sharing was much requested by CloudKit users. It extends cross-device sync to content sharing among multiple users. It's used by most of Apple's productivity apps (e.g., Keynote, Pages, Numbers, Notes) for sharing documents, photos, presentations and other content. It is also used by our mobile gaming platform to support multi-player games. Another example is a third-party app that allows family members to jointly create and edit their genealogy tree.

Bounded Queue. These types of apps usually produce a stream of events, store a sliding-window of the most recent events in CloudKit and keep it in-sync across devices. This window is usually bounded by the number of events and/or by the age of the oldest event. Examples include apps keeping the recent call history or the most recently visited websites in Safari in-sync across devices. Other apps queue objects placed in a "trash bin", allowing the user to un-delete the object until the record expires. These apps usually set a TTL on records stored in CloudKit. Expired records are inaccessible and garbage-collected by CloudKit's maintenance jobs.

Cloud Storage. Some apps use CloudKit as a transactional key-value store, but don't (normally) sync stored data across devices. An example is Apple's mobile backup app. This is a write-heavy app, making use of two zones in each user's private database: the default zone, where file records are stored and a custom zone, where snapshot information is stored. There are usually tens of thousands of (immutable) file records per user and the default zone provides a sufficient level of consistency for such records, while massively sharding them for scalability. The snapshots consist of a multi-level directory structure, akin to i-nodes, that reference file records and can be used to re-construct a consistent snapshot of the mobile device. This information is relatively small but requires cross-record atomic transactions, provided by a custom zone.

5. CHANGE-TRACKING (SYNC)

CloudKit is most-frequently used to synchronize app data on multiple user devices (e.g., laptops, tablets or mobile phones). When one device generates new data, it is stored both on the device and in CloudKit, and propagated to all devices through CloudKit. Sync is a core CloudKit functionality enabling this synchronization.

The sync protocol must fulfill the following requirements: (1) it is initiated by a client, (2) should be efficient w.r.t. network traffic, (3) be able to sync down a part of the data at a time, where the size of each batch is determined by the client (up to certain limits), (4) due to the intermittent connectivity of mobile clients, syncing may

be interrupted at any time and should resume when the client comes back online. Finally, (5) the client should sync to a consistent state, i.e., a snapshot of the server state at some point in time.

5.1 Forward Sync

To support sync, each custom zone maintains a log of record changes. Specifically, every update to a zone is assigned a monotonically increasing version (achieved by using Cassandra's CAS) and each custom zone implements a *sync index*, mapping versions to record identifiers. When a record is modified, the index is updated transactionally, adding an entry for the newly assigned version and deleting the previous index entry for the record. A sync operation performs a forward scan of the sync index.

Deleting the previous index entry is done to minimize network traffic – we avoid returning record versions that are known to be stale. This decision directly impacts the last goal, since a consistent state is reached only when all changes in the log are synced down. This choice is acceptable for most use-cases. When it is not, a *snapshot sync* (see Section 5.3) may be more appropriate.

As part of a sync request, the client specifies a zone identifier, a maximum number of records to return, and a *continuation*. The latter is a cursor into the sync index, and allows resuming an incomplete sync. Initially, no continuation is specified which causes an index scan starting from the start of the log. An updated continuation is sent back to the client in the sync response, and returned by the client in a subsequent sync request. For the client, a continuation is an opaque sequence of bytes. This allows us to change its implementation without changing the client APIs. Finally, the sync response indicates whether or not the sync is complete.

When a record is deleted, it is replaced with a tombstone and the sync index is updated. The tombstone contains the version assigned at record creation. When a sync operation encounters a tombstone in the sync index, and the original record creation version is newer than the start point of the sync (in the continuation), the delete is not sent to the client as the server has determined the client never previously received the create. A maintenance job permanently deletes tombstones once they are seen by all recently active devices.

In some cases, CloudKit may indicate to the client that its sync continuation is no longer valid by sending "reset required" in response to a sync request. This could happen, for example, if the zone was deleted (a unique zone identifier is embedded in the continuation, thus even if a zone with the same name is re-created, continuations referring to the previously deleted zone would not be valid). Another example is a client that did not sync for a long period of time and needs to be told about a delete which was already garbage-collected (the garbage collection job marks the continuations of such devices as invalid). Reset means that a full re-sync is needed to ensure consistency and forces the client to drop its continuation and discard any local representation of server state.

A client usually invokes sync repeatedly (with updated continuations), until it is complete, at which point it has a consistent snapshot of the state. It can then pause syncing until a change notification is received (see Section 7). Our client-side library provides a method that performs these repeated sync calls under the hood, until no more changes are available, using call-backs to pass individual state changes to the app when they are available. This helps reduce latency (sync requests are submitted without waiting for the app to process previous responses) and the granularity of responses isn't exposed to the app.

5.2 Reverse Sync and Meta-Sync

Some apps need to get the newest data first. For example, when implementing a messaging app it may be important to show the last hour of messages when the user first opens the app on a new device; the complete message history doesn't need to appear immediately. Reverse sync starts by scanning the sync index backwards from the latest change committed in the zone, and then automatically continues in the forward direction (from the same starting point), similarly to a forward sync. Scan direction is encoded in the sync continuation. Notice that a consistent snapshot is achieved only after the forward part of the sync completes. This is due to the fact that multiple sync calls may be needed to completely scan the sync index in the reverse direction, and some records may be updated during that time. Such records will be encountered during the forward part of the scan.

CloudKit re-uses its sync mechanism to track various metadata changes. One note-worthy example is the tracking of changed zones in a database, exposed to users as the meta-sync operation which retrieves all zones changed from the client's last continuation. This is especially useful in the shared database (another example is the tracking or notifications, mentioned in Section 7).

5.3 Snapshot Sync

Since the sync index only contains the latest version of each record, scanning the index may skip any change that was superseded by a later change. Hence, scanning a prefix of the index is not guaranteed to provide a consistent snapshot. For some apps, this isn't acceptable. For example, a client of a hierarchical document store app may have a directory D with files F1 and F2. At this point, the sync index may include the pairs [(1, D), (2, F1), (3, F2)]. Now, suppose that D is renamed at version 20, and the index becomes [(2, F1), (3, F2), ..., (20, D)]. Syncing down only the first two entries does not provide a state of the system that ever existed. In this case, since F1 and F2 are missing a parent, the state cannot be shown to the user. This problem is exacerbated if one device is writing faster than another device is syncing – it is possible that sync state is always inconsistent at the slower device, rendering the app unusable on that device.

To address this, an app can use the snapshot sync mode. In this mode, entries are not deleted from the sync index when records are updated (they are garbage collected when no longer needed by any client). *Snapshot points* are chosen by the server (e.g., every 500 zone changes). When a client syncs, we choose a target snapshot point (latest available at the time), and return it as part of the continuation to the client. When scanning the sync index, stale record changes are filtered only when the newer record version appears before the target snapshot point. This avoids sending redundant versions while providing the client with a consistent snapshot.

A trivial implementation could store the version of every record at every snapshot point. This would be extremely wasteful, since most records remain unchanged. Instead, we use an auxiliary index to store (1) the latest version of each record, and (2) for every update, if the previous update to the record preceded the last snapshot point, we store the version of the record at the snapshot.

Suppose that, in our example above, there were snapshot points at versions 5 and 15. When (20, D) is written we update its latest version in the auxiliary index by adding an entry ((D, MAX), 20), where MAX signifies the top version of a particular record. Since the previous version of D was prior to the latest snapshot point we also add a mapping ((D, 15), 1). Suppose now that a client requests to sync and we choose 15 as the target sync point. When encountering the entry (1, D) in the sync index, we first look up (D, MAX) in the auxiliary index, and find out that the latest version is 20. Since ((D, 15), 1) is in the index, we know that 1 is the latest version up to the target snapshot, and can return that version to the client. For the entry (2, F1) in the sync index, we find ((F1, MAX),

2) in the auxiliary index and hence know immediately that this is the latest change to this record, which can be returned to the client.

Since the app must always show the client a consistent state that includes its own recent writes, if the client is writing while syncing, its target snapshot may no longer be sufficiently up-to-date. CloudKit stores the latest version written by each client, and whenever the target snapshot of a sync request falls behind, it is "forwarded" – updated to the latest snapshot point.

Snapshot sync requires a small addition to server responses, which now indicate not only whether more changes are available to sync down, but also whether the client has synced all changed up to the target snapshot point, in which case its state is consistent. Note that reverse sync only considers snapshot points when in forward mode.

Snapshot sync is more expensive than a regular sync: the auxiliary index must be updated transactionally with the sync index, scanning it may take additional time, and, we must retain older versions of records.

6. SHARING

This section describes CloudKit's support for selectively sharing records among multiple users. Examples include Apple's Notes, photo sharing, file sharing, social gaming, third-party apps for sharing shopping lists, genealogy trees and many others.

When sharing a record r for the first time, r's owner creates a *share record* s_r, a special record that facilitates sharing. The identifier of this record is called the *share identifier*. This record contains a list of *participants* – users with whom r is being shared, along with their permissions (read-write or read-only) and other information that will be made clear later in this section. For example, in Figure 4, participant $\#1$ is read-write, participant $\#2$ is read-only and anyone else has no access (*publicPermission* = *none*). When storing s_r, the share identifier is transactionally added to r using Cassandra's batch updates and CAS (both records are stored in the same custom zone). Currently, CloudKit limits the number of participants of a share to 100. Encryption adds an important layer of protection for shared data and is fully supported (encryption in CloudKit is beyond the scope of this paper).

Figure 4: Share record for a shopping list.

In order to share a record with a particular user, clients use a lookup service implemented in CloudKit. The lookup service has access to *user* records – special system records stored in the app's public database. Given an email address, phone number, or user record identifier, the lookup service returns participant information that can be stored in the share record. If the email or phone number do not correspond to a registered user, CloudKit creates a placeholder user record, and only a user that can verify that he owns the email or phone number can assume ownership of the user record and have access to information shared with that user.

Once the share record is stored (or updated with new participant information), newly added participants have to be notified and *accept* the share. To achieve this, we created the following mechanism: every share record has a server-generated unique URL which the owner can send to participants via his favorite method (e.g., email or messaging). Some of our first-party apps (e.g., Notes) show users a UI through which they share the URL, and perform the lookup and update of the list of participants under the hood.

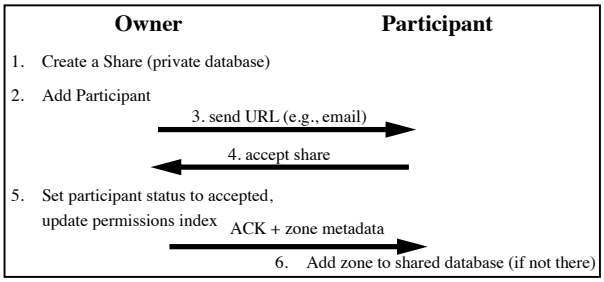

Figure 5: General flow of a new share.

Using the URL, a participant can accept the share and access shared content. This is done either via the app or via the web, depending on whether the participant's platform runs the appropriate version of CloudKit and app. If the web fall-back option is used, the user is re-directed to a web page designated by the app developer (configured via the dashboard).

When a record r in zone z is shared and accepted, zone z is automatically added to participants' shared databases, but no actual records are created. This represents a window into the remote zone they just accepted, providing access to shared records in that zone, while automatically filtering out any non-shared records. Since a zone is uniquely identified by its name and the owner identifier, two owners can share records with the same name to the same participant and two different zones will be created in its shared database.

Each participant has an *acceptStatus*, initially set to *invited*. To accept the share, a server in the participant's partition contacts a server in the owner's partition, which changes *acceptStatus* to *accepted*. A participant can leave a share, changing its *acceptStatus* to *removed*. An owner can remove participants and even delete the share record, ceasing to share the record entirely. The records continue to exist in the owner's private database.

Simple read flow. In a shopping-list app, Alice shares *groceries* with Bob. When Bob reads Alice/shopping-lists/groceries, Alice's shopping-lists zone is registered in Bob's shared database. But this is not the source of truth – Bob's database may not yet reflect permission updates made by Alice. Hence, we send an RPC to one of the backend servers in Alice's partition. It checks that Alice has read access to the zone, and otherwise returns "zone does not exist" to Bob (unlike "access denied", this does not reveal the zone's existence). If Bob has access, we read the groceries record and find the share identifier. We check the share record to make sure that Bob has read access, and if so return the groceries record to Bob. In practice, reading records for the purpose of checking permissions is avoided by maintaining a permissions index.

Hierarchies. Often, an app entity is represented by a hierarchy of records. For example, a document including images, forms, and other objects could be represented by a root record, and contained objects would be records referencing the root. When the document is shared, a share identifier is added to the root. On record access, we find the share identifier by traversing up the hierarchy.

One of the benefits of this design is that sharing is recursive and inheritable – descendants automatically inherit the sharing proper-

ties of their ancestor, even if added at a later time. Another benefit is that sharing/un-sharing a hierarchy of records requires a change to a single record. A consequence, however, is that we do not allow a descendant to be shared differently than its ancestor (though sibling records can be shared differently). Specifically, if a record includes a share identifier, no ancestor may include a share identifier (sharing an ancestor requires first un-sharing its descendants).

A read-write participant can add, modify and delete records, but they must be descendants of the root record and it cannot delete the root. Records added by a participant belong to the owner, reside in the owner's private database and counted against his quota.

Change-tracking. To support sync, the owner's custom zone maintains a share change index which is similar to a normal sync index but includes only changes to shared records in the zone. Whenever a record is updated, we find the relevant share record by traversing up the record hierarchy and update the index by adding an entry that maps the new version to both the record and the share identifiers. Participants issue a sync request to a zone in their shared database (only forward sync is currently supported), which is proxied to the owner's partition. There, the index is traversed starting from a continuation provided by the client. Changes to records in shares for which the client has read permissions are returned.

When a hierarchy of records is shared, only the root is added to the share change index. When a participant syncs, we have to return all the records and hence traverse the hierarchy (with the help of an index that maps parent records to their children). One complication is that the number of changed records that can be returned in a sync response is limited and the limit can be reached during a traversal. We can pause the traversal at any point, saving the final path from the root. When the client issues a new sync request, we resume the traversal. If the hierarchy was modified and the saved path is no longer valid, we find a valid prefix of the path, and resume the traversal from there. This method significantly simplifies the required reconciliation logic and reduces saved state, in the expense of potentially sending duplicate changes to the client. If the path is valid but changes occurred in the already-traversed portion of the hierarchy, these changes appear later in the share change index and are sent to the client after the traversal completes.

In our example, both Alice and Bob (assuming he has read-write access) can edit groceries. This is performed by a CloudKit server in Alice's partition. Once the share change index is updated, an asynchronous message is sent to all participant partitions for the relevant share. There, we notify participants that have registered database subscriptions for their shared database. Clients then follow-up with a meta-sync request to find out which zones have changed, followed by sync requests to these zones. Unlike sync, meta-sync in the shared database is processed locally in the participant's partition, avoiding multiple RPCs to the owners. Since participants are notified asynchronously, there is a small delay until a meta sync reflects new updates to shared content.

7. QUERIES, SUBSCRIPTIONS AND NOTIFICATIONS

Queries. While an app may want to sync-down some of the state it stores in the cloud, in many cases maintaining a local copy of all the data is either not desirable or not feasible. Instead of syncing down a zone's state an app can use the query interface to retrieve a subset of records stored in the zone. Queries operate on a single record type across record zones (at the database level) and support flexible filters on record field values. These include standard comparison operations for numeric values, basic string match operators for string values, and containment operators for list values. For location field values we support proximity filters, where an origin location and a proximity radius are provided. Filters can be combined using AND and OR into compound (nested) filters. For example, a user of a restaurant app may query for all Japanese restaurants within a certain radius from a location. A user of a News app may be interested in certain topics or news outlets.

Queries are implemented using Apache Solr [6]. Waiting for Solr to index new data may significantly slow-down writes, and hence is done asynchronously, without blocking client operations. This means, however, that queries are only eventually consistent and might not immediately return the most-recently written data.

Many apps use both queries and sync, either in different zones or even in the same zone. For example, queries can provide an immediate response to a specific user request, e.g., to populate a page viewed by the user, while syncing can be used to update its state in the background.

Both queries and sync are polling the data store. It is often better to register a subscription and have CloudKit notify the user when relevant data is written. Developers specify when their app needs to be notified. CloudKit supports three types of subscriptions:

Zone subscription. The client provides a zone and CloudKit sends notifications for every record change in that zone. Notifications are sent to all devices for the registered user account, except for the device which caused the zone change (e.g., saved a record). The subscription can optionally include filters based on event type (record insertion, modification or deletion) and on record type.

Database subscription. The client specifies a database (private or shared) and gets notified for changes to its zones. This includes zone creation, deletion or modifications (changes to records in the zone). The main motivation for database subscriptions (as well as meta-sync, described in Section 5.2) is the shared database, for which this is currently the only supported subscription type. In the private database, the list of zones used by an app is usually predefined and the user can subscribe for change notifications in each individual zone. In shared databases, zones appear and disappear dynamically as other users share content with the user. Hence, the list of zones is not known to the app developer a-priory.

A database subscription notifies the client that some zones have changed. The client can then use the meta-sync interface to find out which zones have changed, and finally can issue sync requests to get record changes from these zones (the client-side library provides a single method that performs these steps).

Query subscription. Similar to a zone subscription, in that it returns records, but can operate across multiple zones like the query interface. Used to register continuous queries which, just like normal queries, support flexible filters. They are triggered whenever a response to the registered query changes. The subscription includes a list of fields whose values are sent back in the notification, in addition to the record identifier.

An app developer pre-defines query subscription types in the development environment and CloudKit creates a parameterized query for each type. For example, a restaurant query could be "restaurant rating $> x$ and location within y kilometers of coordinate z". When a user sends a restaurant query subscription, CloudKit makes sure that the type was pre-defined and if so creates a CloudKit record representing the actual (non-parametarized) query subscription, including the provided values for x, y and z. The query is also indexed in Solr.

When a record is updated, CloudKit checks whether query subscriptions were registered for this record type and if so creates an asynchronous task to find all affected queries. For a limited class of query subscriptions, looking up relevant subscriptions is optimized by indexing queries similar to field values. For example, if

a user was interested in records with a numeric field value $> x$, we can index the query and when a new record with field value y is saved create a query for all subscriptions looking for values $< y$. This cannot be done for all types of query predicates and in some cases we simply read all registered queries (for the relevant record type), and evaluate a potential match with the newly saved or updated record. To bound server resource consumption, we limit the number and type of different query subscriptions for each app.

Notifications. There are two common use-cases for push notifications in CloudKit: (a) notifying a user about something important (visual notification), and (b) notifying the app that data has changed so that it can update itself (silent notification). When registering a subscription, the developer specifies various context attributes allowing a correct interpretation of the notification by the client device, as well as actions to be taken. For example, a notification can have a title and body, can count towards the app's number of missed notifications, have a specific notification sound, etc.

CloudKit uses the Apple Push Notification service (APNs) to send push notifications. When received on the device, a notification is routed to the appropriate app (launching it in the background, if needed), which in turn can choose whether and how to present it to the user, using the included context and action attributes.

Currently, notifications are best-effort and can be lost if the client is offline for a long period of time. When back online, it can get updates by using the sync, meta-sync and query interfaces. To mitigate notification loss further, we've implemented notification sync, which reuses the sync abstraction for fetching missed notifications. In the future, we may deprecate notification sync and leverage new APNs capabilities. As our service scales further, we may also support one-time subscriptions that need to be re-registered.

Examples. Consider two simple apps: a shopping app and a messaging app. In the first app users create and modify shopping lists. Each list can be a record in the *shopping-lists* custom zone in the user's private database. When one device changes or adds a list, all other devices should show the updated list immediately, without polling. This can be achieved by a zone subscription and silent notifications – whether the app is in the foreground or background, silent pushes are very useful for telling the app that some information of interest has been added or changed. The app can then sync from the zone to update its local state.

In a messaging app, suppose that users can post messages in a single chat room, subscribe to topics of interest and be notified when someone mentions such topic. Our data model could consist of two types of records: Messages and Users. Because Messages can be seen by anyone using the app, they can be stored in the public database. For this app we can use query subscriptions. One subscription would be for new records of type Message in the public database's default zone. Whenever a new message is added, a silent notification would notify the app, including the message information which the app can use to update its local state. The second subscription could capture topics of interest – the query predicate would match all Message records where message text contains such topic. The notification in this case can include a visual alert, potentially showing a banner, making a noise, and (if missed) updating the app's missed notifications count.

8. SCHEMA MANAGEMENT

For each record type, CloudKit stores a type definition in Cassandra. Table 6 shows a simplified record type definition for "Restaurant" records (e.g., in a restaurant reservation app).

A record type definition is immutable: whenever it is modified (allowed only in the development environment), a new record type is created with a different type_id (but the same type_definition_id).

Table 6: Example: Restaurant record type

type_id (primary key)	uuid1
type_name	Restaurant
type_definition_id	uuid2
Field types	(see below)
Query subscription types	(see Section 7)

field id	name	type	flags
1	"Name"	String	QSZ
2	"Cuisine"	Integer	Q

The type_definition_id signifies the "incarnation" of the type – if the "Restaurant" record type is deleted and created again, this would result in a different, incompatible record type with the same name but different type_definition_id. Record type definition stores information about record fields: each field has a name and assigned a monotonically increasing id. Fields can have up to 3 indexing flags: queryable for equality (Q), full-text searchable (S), or sortable (Z), determining whether and how field values are indexed in Solr. The type definition also stores query subscription types. These are query templates defined by the app developer (see Section 7), e.g., restaurant record inserts with cuisine c located within x kilometers of coordinate z. Each template is assigned a unique identifier, used to help index instantiations of the query in Solr, i.e., actual queries of this type submitted by users of the app.

A schema stores information about all record types for a container, including their name and type_id. A simplified schema for a restaurant reservation app is shown in Table 7.

Table 7: Example: schema with three record types

schema_id	uuid3	
parent schema_id	uuid4	
record types	Restaurant	uuid1
	Review	uuid5
	Reservation	uuid6

Like record types, schemas are immutable – each schema has a schema_id and whenever a change occurs in one of the record type definitions, a new schema is created with a different schema_id, where the updated record type name maps to its new type_id. Each schema also stores its predecessor schema_id, and various metadata (schema modification time, what device made the change, etc.). Previous versions of record types and schema definitions are retained. This allows to correctly interpret data stored using previous schema versions and to track the evolution of a container's schema. Both the development and the production environment store the latest schema_id, updated atomically using Cassandra's CAS.

Using the schema. Record store-requests contain record fields, each with a type and a value; the type is used to deserialize the value. Latest container schema is used to map the record name to its latest type definition. Each existing field in the request is validated against its type stored in the record type definition, in most cases simply checking for type equality. If previously unknown fields were provided, a new type definition is created, as well as a new schema, and the container's latest schema_id is updated. When the record is stored in Cassandra, only field_ids and field values are stored (not field names or types), in addition to the record type_id. This allows reusing field names, e.g., removing a field "Cuisine" and creating a field with the same name but a different type. Ambiguity is avoided since the two fields have different field_id numbers.

On record fetch, requests retrieve records from Cassandra, and using the type_id stored in the record we find the record type definition with which the record was stored and use it to deserialize the

record. Then, we find the latest type definition for this record type name, by using the latest container schema. If the type_definition_id has changed and is different from the one used to store the record, this record should not be returned. If only the type_id has changed, the changes are applied to the retrieved record. For example, some of the fields may have been removed from the type definition – these fields are filtered out before the record is returned to the user.

Promoting and deploying the schema. When the container schema is promoted to production, the development schema object is cloned, assigned a new schema_id, and the parent schema_id is set to the latest production schema. The latter is then atomically assigned to point to the newly created schema object. Usually, only a subset of the development schemas will be promoted.

Each container has an authoritative partition, responsible for maintaining the schema. Other partitions cache immutable schema objects, fetched from the authoritative partition on demand. Since the users of a container may be mapped to different partitions, any updates to the schema requested by a user needs to be forwarded to the authoritative partition. The user's partition may already cache a previous version of the schema with a schema_id x, in which case it creates a new schema with an id y and sends a request to the authoritative partition, which performs a CAS update on the latest container schema, conditioned on its latest version still being x. If this fails, an up-to-date schema is sent back to the user's partition and, if necessary, the process repeats.

9. RELATED WORK

Multiple commercial and open-source frameworks, e.g. [7, 10, 16, 17, 22, 24, 25, 27, 28], have emerged aiming to simplify mobile app development and provide mobile backend as-a-service. To the best of our knowledge, this is the first paper describing the implementation of such commercial framework. CloudKit is a leader in this space and has a unique data model and feature-set that was proven useful for many apps, including many of Apple's current and emerging apps. For most apps, it is the only backend needed (for others, CloudKit exposes APIs to interact with other backends). CloudKit currently scales to hundreds of millions of users.

These systems build on decades of academic research, in particular on replication protocols for mobile computing [44]. The protocols can be categorized based on communication topologies (client-server vs. peer-to-peer), partial or full replication, data model and consistency guarantees. In this design space, CloudKit follows the client-server replication model, similarly to [37, 39, 34, 46]. But, unlike [37, 39, 46], provides very limited support for disconnected operations. Instead, CloudKit APIs allow apps to re-sync and resolve potential conflicts when connectivity is restored. Similarly to other systems, like [45, 34], we chose not to offer one-fits-all automatic conflict resolution, allowing apps to define their own resolution logic. Many apps using CloudKit maintain a local cache of CloudKit records or use device storage directly to support disconnected operations and improve performance. Third-party libraries were also built to provide such functionality on top of CloudKit. CloudKit provides support for both full and partial replication but leaves this decision up to the app – an app can sync state changes and maintain a full local replica of one zone, while using queries and query subscriptions to define selective state filters (akin to systems like [33, 40, 41, 43]) in a different (or even the same) zone. Unlike systems providing weak or eventual consistency, e.g., [45, 37, 40], CloudKit supports atomic (linearizable) updates, reads and sync operations. Like [36, 4, 17], CloudKit provides condiditonal updates but goes further to provide transactional batches that can include both reads and writes. It does not, however, support full fledged transactions provided, e.g., by [1, 19, 46].

Unlike most NoSQL databases and many commercial competitors that offer generic key-value storage where the value is a blob of information (e.g., a JSON file), one of CloudKit's distinguishing attributes is its expressive schema support – a CloudKit schema consists of one or more record types that have a name, fields, and other metadata. App developers define the schema by storing data in the sandbox environment (and with the help of a dashboard) and CloudKit enforces it. For more complex data, CloudKit provides cross-record references with various semantics.

Push notification services [3, 8, 18, 30] allow application cloud backends to notify user devices. Other cloud services [32] can provide reliable notification delivery. Devices interact directly with CloudKit, store data, register various types of subscriptions and receive push notifications, without the need to configure or interact directly with a push notification or another service. Currently, CloudKit provides notification reliability by implementing notification sync, however we find that push service reliability is sufficient for the vast majority of apps. Some cloud storage services [1, 2, 20] similarly allow subscribing to data change notifications, but most don't support query subscriptions as CloudKit does.

Existing cloud services are exposed to mobile apps through REST APIs or wrapper libraries of the APIs. CloudKit supports a REST API, useful for web clients, but most mobile clients interact with it via a daemon running locally on the device. Often, multiple apps on a device use CloudKit, and the daemon maintains a single persistent connection with the cloud for all client apps, routing app requests, responses and notifications. In many cases the client-side library provided by the daemon encapsulates multiple CloudKit operations (e.g., repeated sync), or provides higher-level abstractions.

10. PRODUCTION EXPERIENCE

CloudKit has been deployed in production for more than three years. In this section we describe some of the challenges we faced, and share production measurements.

10.1 Production Challenges

Maintaining, deploying, evolving and scaling CloudKit involves many challenges. We provide several examples.

Managing dependencies. CloudKit depends on other back-end systems and services. Examples include storage systems, account management systems (e.g., for validating user accounts, or finding the CloudKit partition for a user), quota management systems, and more. These systems need to be deployed in a compatible way and provisioned to sustain CloudKit's request load. In cases when this wasn't possible, we've used caching, added more functionality to CloudKit to reduce the dependency (such as request validation) and in some cases resorted to reduce the feature set.

Backward and forward compatibility. Upgrading dependent systems is challenging since it creates compatibility issues with CloudKit. Upgrading CloudKit itself in a partition is done gradually, which may cause incompatibility with servers that weren't upgraded. For example, information added to cross-server messages or stored content in a new version of CloudKit may be ignored or misinterpreted by a server still running the old version. We're mitigating such issues by using Protocol Buffers and controlling most new features with flags, which remain turned off during a partition upgrade, and are enabled only after all hosts are updated. CloudKit is accessed from a variety of client devices and over the course of three years dozens of client software versions were released. Some clients never upgrade and CloudKit needs to support requests coming from such devices. This is a major challenge for developing and testing new features (such as sharing) and software releases.

Logging. Initially, CloudKit used two logging systems – one for a detailed account of recent events (development logs), the other for aggregate time-series and alerts. Over time, a data warehousing system was added, and recently an event aggregation system that exposes some server events to app developers through the CloudKit dashboard (this allows developers to combine client and server logs, which is tremendously helpful for debugging). These systems collect massive amounts of data and require provisioning and maintenance. For example, the log of recent events is of the order of 100TiB per day.

Multi-tenancy. The underlying Cassandra does not provide isolation between users or apps. CloudKit sets fairly restrictive limits on what a single device can do in one request (number of records, request size, size of each record, etc). Most requests to CloudKit access a small set of records. For example, sync encourages accessing only the recently updated records and in limited batch sizes. Rate-limiting requests (per client device, user and application) limits the effect a user can have on the system. Furthermore, the operations we chose to expose map linearly to storage accesses, which helps protect against malicious users and bugs. This is complemented by monitoring and alerts that prompt human intervention. Device and software heterogeneity, non-uniform app popularity across users and geographical regions, and other factors occasionally result in load imbalance. As mentioned in Section 3, we periodically rebalance users across CloudKit partitions to improve load-balancing, resource utilization and access locality.

Scale amplifies everything. When running at CloudKit's scale, every extra action or request to an external service can have a significant effect. For example, a small code change accidentally caused client device information to be updated in memcached on every request, which was both a useless increase in traffic and hurt hit rate for other cached information. In another instance, a configuration file was accidentally deleted during a release, which specified the number of connections each CloudKit JVM instance maintains with Cassandra. This caused the total number of connections to Cassandra to increase 10-15x, triggering many alerts. Finally, since CloudKit serves hundreds of millions of active users, deciding how frequently to sync or how much information to store per user may have significant costs. For example, the cost of storing an extra 1MB per user is hundreds of TiB for all users collectively, while having each device check-in with the backend every 15 min would generate hundreds of thousands requests per second.

10.2 Production Measurements

Public and private data. We identified the top apps using CloudKit, based on their number of active users in the past month, and examined their use of private and public databases. We found that 20% and 49% of the apps use only the public or the private database, respectively, and 31% of apps use both databases (20 apps use the shared database). We conclude that the segregation of data into private and public domains is useful.

Reading data. Currently, 44% of CloudKit's third-party app read traffic are record fetch operations, followed by queries (37%) and sync requests (19%). First-party apps, however, make an increasing use of the sync+notify APIs. Specifically, for 57% of first party apps sync traffic exceeds queries or record fetch operations, and for 43% of first party apps it is 90% or more of their read traffic. Figure 8 examines record read, sync and query request server-processing latency. The median record read latency is $\sim$ 18ms.; query and sync take $\sim$ 50ms.

Writing data. Figure 9 demonstrates that record save request latency depends sub-linearly on the number of records inserted or updated in the request. For third-party apps, 98% of record save

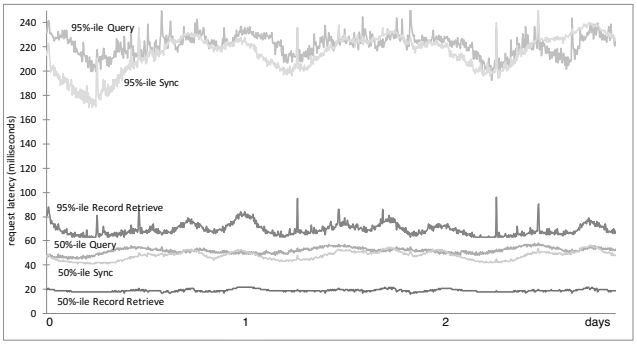

Figure 8: Operation processing latency (read, query, sync).

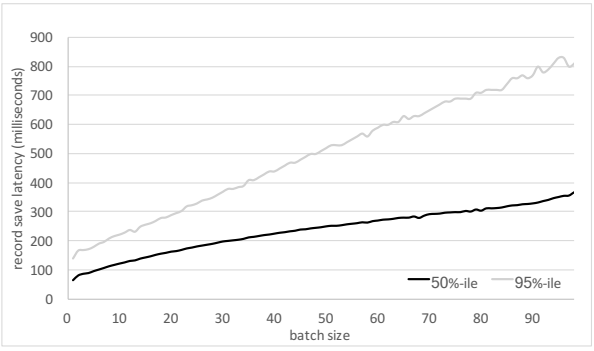

Figure 9: Record save processing latency as a function of the number of saved records.

requests do not use batching, whereas a significant fraction of first-party apps use large batches (e.g., photo library and mobile backup). Batching allows multiple records to be saved atomically (in custom zones) and saves client-server round-trips.

Exercising CloudKit's APIs. apps use CloudKit in various ways (see Section 4), resulting in diverse workload patterns. Figure 10 gives two examples. On the top we have a first-party document sharing app, subscribing to database notifications for both the private and the shared database. Upon notification, it issues a meta-sync request to find all changed zones, followed by sync requests from these zones. Meta-sync requests are also issued when the app is launched or brought to the foreground. Further increasing the rate of meta-syncs is the fact that some change notifications are sent to multiple devices and participants sharing content. Record reads are performed when a significant conflict occurs while saving a record that requires reading other records to resolve. The third-party app (bottom figure) stores and syncs multiple stream cursors, each corresponding to a different event stream (similar to Tweeter's TweetDeck). Each update generates a notification and a sync from this user's *other* devices using the app, if any, explaining why sync rate is slightly lower than the record save rate. This app uses queries to retrieve data.

Notifications. Currently, zone notifications are by far the most prevalent, accounting for 86% of notifications, followed by database notifications (13%) and finally query notifications at less than 1%. We expect the use of database notifications to increase as the adoption of sharing, a relatively new CloudKit feature, continues.

Sharing. Figure 11 examines sharing. On the top we show new share creation rate, dominated by multi-player game sessions created in our gaming platform, followed by document and activity sharing apps. On the bottom we present the update rate to shared data, and the number of participants per update (the median is a single participant (owner not counted), whereas 95% and 99% are 6 and 7 participants).

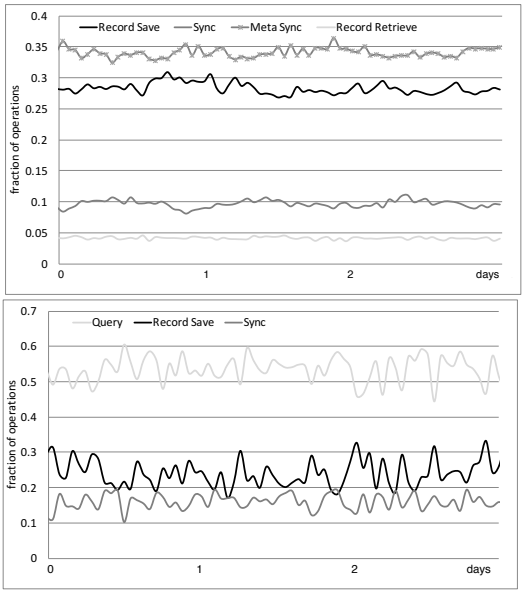

Figure 10: Workload of first-party document sharing (top) and social-stream (bottom) apps. Not all API calls are shown.

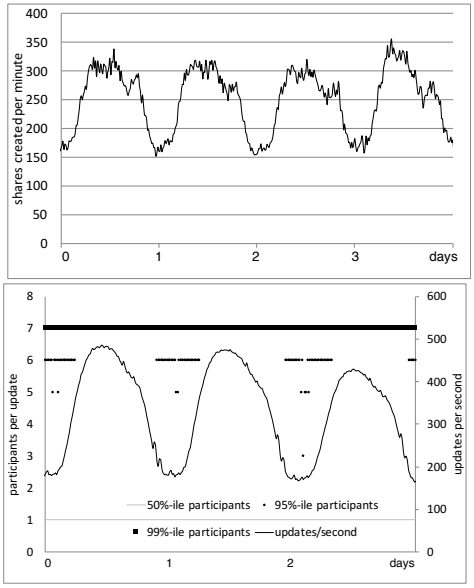

Figure 11: Sharing: num. of shares created/min (top), share updates/sec and num. of participants per update (bottom).

CAS contention. Our custom zone implementation serializes all updates to each zone using Cassandra CAS. When multiple concurrent updates are attempted in the same zone, by devices of the same user or of multiple users sharing content in the zone, only one succeeds and the other retried by the backend and may eventually fail. We measured the frequency of such errors for the busiest 150 apps (based on the volume of save requests), taken over more than 5 billion save requests from a single day. We find that 75 apps experienced less than 0.005% such request failures and 122 apps saw less then 0.1% errors. Internal apps experienced at-most 0.05% errors while 8 third-party apps experienced more than 1% errors. Our experience shows that carefully designing the app data model to partition data across multiple zones, and designing app logic to avoid correlation of data update-times across client devices as much as possible, drastically reduces zone contention.

Schema cacheability. Immutability of schema and record type definitions facilitates their cacheability. They are cached by each server instance, and on the level of individual requests. Each request may involve multiple records and record types; in order to provide a consistent view of the schema for all operations within a request, we fetch the schema once at the beginning of a request and cache it at the request level. Hence, a record type read will likely find the type definition in the request cache, or the server instance cache, before falling back to reading the definition from Cassandra. Quarter-long measurements in production across all CloudKit partitions show that cache hit rates are nearly constant: For record type definition reads the average request level cache hit rate is 0.9 while the server instance cache hit rate is 0.995. For schema definition reads both hit rates are close to 0.96. Request level hit rate is lower for record type definitions compared to schema definitions since there is a single schema but potentially multiple record types accessed by the same request. Instance-level hit rate is better for record types simply because they are configured to take much longer to expire compared to old schema definitions (old record types are useful to deserialize previously stored data). Only one out of $\sim$2800 record type reads and one out of $\sim$700 schema reads reaches Cassandra. This rate could be reduced further by using cluster-level caching (e.g., memcached).

Global use. Finally, Figure 12 presents CloudKit client requests from around the world, each region experiencing a diurnal pattern. Total usage peaks morning PST, when wake-hours overlap (a smaller peak occurs late evening PST). We run non-critical maintenance jobs when client request load is low.

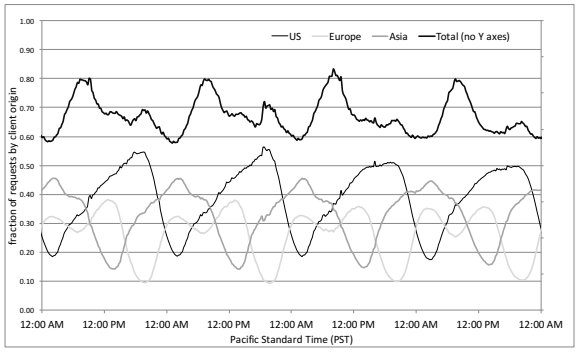

Figure 12: Relative request rate by client origin.

11. CONCLUSIONS

CloudKit is a leading mobile backend service, used extensively at Apple and by many third-party apps and hundreds of millions of active daily users.

Unlike other services which make it easy to rapidly develop an app but hard to evolve it, CloudKit prepares apps for scale from the get-go by making schema management explicit and by supporting schema evolution and deployment. CloudKit enforces the schema on client updates and rejects inconsistent schema modifications made by the developer after the app is deployed to production.

Led by an observation that most apps store user data and common app data, CloudKit explicitly partitions app data into private and public databases, with different capabilities. Private databases allow apps to further partition data into zones and guarantee strong consistency within each zone.

CloudKit provides an explicit set of tools to achieve consistency across devices and users: a sync/subscription interface (that most first-party apps are adopting) as well as queries and continuous queries for selective state replication. Zones make it easy to use these capabilities for different parts of an app's data or to use them jointly for the same data.

12. REFERENCES

[1] Amazon dynamodb. https://aws.amazon.com/dynamodb/.

[2] Amazon s3. https://aws.amazon.com/s3.

[3] Amazon simple notification service. https://aws.amazon.com/sns/.

[4] Amazon simpledb. https://aws.amazon.com/simpledb/.

[5] Apache cassandra. http://cassandra.apache.org/.

[6] Apache solr. http://lucene.apache.org/solr/.

[7] Apache usergrid. http://usergrid.apache.org/.

[8] Apple push notifications. https://developer.apple.com/notifications.

[9] Apple's approach to privacy. https://www.apple.com/privacy/approach-to-privacy/.

[10] Azure app service. https://azure.microsoft.com/en-us/services/app-service/.

[11] Cloudkit. https://developer.apple.com/icloud/cloudkit/.

[12] Cloudkit dashboard. https://developer.apple.com/videos/play/wwdc2017/226/.

[13] Cloudkit developer documentation. https://developer.apple.com/documentation/cloudkit.

[14] Cloudkit js. https://developer.apple.com/documentation/cloudkitjs.

[15] Cql features in cassandra 2.0.6. https://www.datastax.com/dev/blog/cql-in-2-0-6.

[16] deepstream. https://deepstream.io/.

[17] Firebase. https://firebase.google.com/.

[18] Google cloud messaging. https://developers.google.com/cloud-messaging/.

[19] Google cloud spanner. https://cloud.google.com/spanner/.

[20] Google cloud storage. https://cloud.google.com/storage.

[21] grpc: A high performance, open-source universal rpc framework. https://grpc.io/.

[22] Ibm cloudant. https://cloudant.com/.

[23] ios security guide. https://www.apple.com/business/docs/iOS_Security_Guide.pdf.

[24] Kinto. http://kinto.readthedocs.io/en/latest/.

[25] Kinvey. https://www.kinvey.com/.

[26] Lightweight transactions in cassandra 2.0. https://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0.

[27] Meteor. https://www.meteor.com/.

[28] Parse. https://en.wikipedia.org/wiki/Parse_(company).

[29] Protocol buffers. https://developers.google.com/protocol-buffers/.

[30] Push notifications (windows phone). https://msdn.microsoft.com/en-us/library/hh221549.aspx.

[31] Techcrunch: icloud reaches 782 million users. https://techcrunch.com/2016/02/12/apple-music-tops-11-million-subscribers-icloud-reaches-782-million/.

[32] A. Adya, G. Cooper, D. Myers, and M. Piatek. Thialfi: A client notification service for internet-scale applications. In *SOSP*, 2011.

[33] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. Practi replication. In *NSDI*, 2006.

[34] B. Chun, C. Curino, R. Sears, A. Shraer, S. Madden, and R. Ramakrishnan. Mobius: unified messaging and data serving for mobile apps. In *MobiSys*, 2012.

[35] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[36] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

[37] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst.*, 1992.

[38] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[39] E. B. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the blue file system. In *OSDI*, 2004.

[40] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *NSDI*, 2009.

[41] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger. Perspective: Semantic data management for the home. In *FAST*, 2009.

[42] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[43] J. Strauss, J. M. Paluska, C. Lesniewski-Laas, B. Ford, R. Morris, and M. F. Kaashoek. Eyo: Device-transparent personal storage. In *USENIX ATC*, 2011.

[44] D. B. Terry. *Replicated Data Management for Mobile Computing*. Synthesis Lectures on Mobile and Pervasive Computing. Morgan & Claypool Publishers, 2008.

[45] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[46] I. Zhang, N. Lebeck, P. Fonseca, B. Holt, R. Cheng, A. Norberg, A. Krishnamurthy, and H. M. Levy. Diamond: Automating data management and storage for wide-area, reactive applications. In *OSDI*, 2016.