QuiCK: A Queuing System in CloudKit

Kfir Lev-Ari, Yizuo Tian, Alexander Shraer, Chris Douglas, Hao Fu,
Andrey Andreev, Kevin Beranek, Scott Dugas, Alec Grieser, Jeremy Hemmo
Apple Inc

ABSTRACT

We present QuiCK, a queuing system built for managing asynchronous tasks in CloudKit, Apple's storage backend service. QuiCK stores queued messages along with user data in CloudKit, and supports CloudKit's tenancy model including isolation, fair resource allocation, observability, and tenant migration. QuiCK is built on the FoundationDB Record Layer, an open source transactional DBMS. It employs massive two-level sharding, with tens of billions of queues on the first level (separately storing the queued items for each user of every CloudKit app), and hundreds of queues on a second level (one per FoundationDB cluster used by CloudKit). Our evaluation demonstrates that QuiCK scales linearly with additional consumer resources, effectively avoids contention, provides fairness across CloudKit tenants, and executes deferred tasks with low latency.

CCS CONCEPTS

• **Information systems** → **Message queues**; • **Software and its engineering** → *Ultra-large-scale systems.*

KEYWORDS

Distributed queuing systems; mobile back-ends; Multi-tenant cloud services; large-scale systems

ACM Reference Format:
Kfir Lev-Ari, Yizuo Tian, Alexander Shraer, Chris Douglas, Hao Fu,, Andrey Andreev, Kevin Beranek, Scott Dugas, Alec Grieser, Jeremy Hemmo. 2021. QuiCK: A Queuing System in CloudKit. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457567

1 INTRODUCTION

To process client requests, modern cloud services perform a variety of tasks and commonly communicate with multiple backend systems. Some tasks must execute in-line with the request, to guarantee the expected semantics to clients. Other tasks must happen reliably, but not necessarily before responding to the client. Carefully identifying which tasks must happen in-line and which can be deferred reduces request latency, improves resource utilization, and optimizes interactions with dependent services. When a downstream system is unreachable, slow, or returning transient errors, postponing the call rather than failing a client request can significantly improve service availability.

In this paper, we address the management of deferred tasks in the context of CloudKit [25], Apple's strongly-consistent structured storage service and app development framework. CloudKit serves as the storage and sync engine for many of Apple's popular apps (e.g., iCloud Drive, Notes, Photos, iMessage, iWork, News, Backup and GameCenter), many 3rd party apps, and hundreds of millions of daily users. CloudKit relies on FoundationDB [9, 29], an ordered key-value store that supports strictly-serializable ACID transactions, and on the FoundationDB Record Layer [18, 19], a companion system that adds record-oriented APIs, transactional secondary indexes, a query language and more. FoundationDB and its Record Layer are the open-source bedrock of cloud infrastructure at Apple. CloudKit utilizes hundreds of FoundationDB clusters. It represents each user's app data as a logical database, assigned to one of the clusters, and frequently re-balances FoundationDB clusters by moving user data in response to changes in load, to balance capacity, or to improve locality. Sharding by user and app grants CloudKit considerable flexibility when rebalancing logical databases, as the typical database is small and accessed infrequently [19]. When scaling up CloudKit, new FoundationDB clusters are added to the fleet, and logical databases are moved to utilize this capacity, balancing their load and storage demand across clusters.

CloudKit must handle a high volume of client requests with low latency. Some tasks can be deferred and don't have to block the client. The following are examples of tasks that are too expensive to execute in-line, may be throttled by an external service, and/or have transient effects on concurrent queries:

- Create or drop indexes (in all CloudKit shards and locations, globally) when an app's schema is updated
- Update indexes in external systems (e.g., full-text indexes in Solr) when data is updated
- Schedule push-notifications to users when data is updated
- Coordinate asynchronous notifications to remote CloudKit shards when data (e.g., a Keynote document) is shared with users assigned to those shards
- Compaction of Backup snapshots and various metadata
- Cascading updates and directory deletes in iCloud Drive

Queuing and Workflow Management Systems are the canonical solutions for managing deferred tasks. Such systems provide reliability and high-availability with guaranteed (at-least-once) execution semantics through automated retries. However, using these systems to persist the tasks of a service such as CloudKit comes with significant caveats. First, there is no transactionality between CloudKit and an external queuing service; it isn't straightforward to guarantee correlated execution of deferred tasks and updates to data in CloudKit, should the queue, the task executor, or the

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457567

database fails/aborts. Second, this introduces a separate highly-available, persistent system that needs to be provisioned, operated, and tuned. Such systems manage their data differently from CloudKit and usually cannot provide per-tenant information (e.g., the number of queued items of a particular user and app), do not support per-tenant actions, and cannot guarantee isolation or fair resource allocation between the tasks of different CloudKit tenants. Naively mapping CloudKit's tenancy model to the queuing system, e.g., creating a topic per database, does not scale. Finally, CloudKit often moves user data across its shards, and their tasks must follow. Having part of a user's information reside in a different system creates both data management and security challenges. CloudKit would not only need to track tens of billions of tenant logical databases, but now also separately track the locations of their deferred tasks. For example, if a user is moved across continents, but its tasks remain behind, the tasks may become non-deterministic due to regional differences in data management policy. Neither allowing the task to operate on the remote database nor implementing a task migration system are attractive, as both need to audit the provenance of both the task and the database it affects. Collecting sufficient provenance information for every backend system the task accesses is error-prone and difficult to debug, as the task telemetry is stored in different system instances for every link in the migration chain.

To address these challenges, we've built QuiCK, a Queuing system developed for CloudKit, that is fully integrated with CloudKit itself. QuiCK stores deferred tasks with user data in CloudKit, providing transactionality with other data, and leveraging CloudKit's data model, dynamic sharding, observability, and tooling. Just like CloudKit, QuiCK relies on FoundationDB and its Record Layer.

Even though transactional DBMS systems have been previously used for queuing and workflow management, using an ACID database for queuing is still commonly considered an anti-pattern in the industry [5]. Specifically, concerns are raised with creating hot-spots in the database by using the same tables for both high-rate enqueue and dequeue operations, the added complexity and locking mechanisms needed to support concurrent consumers, and the common lack of push notification support in databases (readily available in most queuing systems). In this paper, we show how QuiCK's design overcomes these challenges. First, QuiCK avoids hot-spots by employing sharding on two levels – first and foremost, our partitioning is based on the CloudKit tenancy model, taking sharding to an extreme with potentially tens of billions of queues stored in the system (one for each user of each app). These queues are further partitioned across many FoundationDB clusters. Second, to avoid contention among consumers, QuiCK uses a fault-tolerant, coarse-grained leasing mechanism, where consumers contend on queues rather than individual queued items. We avoid contention between enqueue and dequeue operations by a careful use of the Record Layer's transactional indexing. Finally, in QuiCK a shared pool of consumers is responsible for hundreds of millions of queues. Instead of notifying consumers when new work is enqueued, QuiCK provides an efficient way for them to find non-empty queues when they're available to accept new work. This allows work scheduling to not only consider item arrival time but also to allocate computational resources fairly among CloudKit tenants. In summary, this paper makes the following contributions:

- QuiCK, a transactional queuing system built for CloudKit;
- A scalable design that avoids hot-spots and contention;
- Full support for CloudKit multi-tenancy, including isolation, fair resource allocation, observability and tenant migration;
- A detailed description of how QuiCK leverages FoundationDB and the Record Layer, recent open source-systems with growing communities. Our use-case demonstrates interesting and subtle aspects of these systems that could benefit others.

This paper is organized as follows. Section 2 includes a discussion of the design goals and requirements for managing CloudKit's asynchronous tasks. In Section 3 we review the reasons we've chosen to embed a queuing system in Cloudkit, rather than use an external system, and discuss the typical concerns with using a databases to represent queues. Section 4 provides background on FoundationDB, the Record Layer and CloudKit, as well as gives insights into their features that enabled QuiCK's design. Section 5 describes Queue Zones, a new construct implemented in CloudKit to represent individual queues. We present a detailed description of QuiCK's design in Section 6. Section 7 discusses related work. Section 8 includes evaluation. Finally, Section 9 concludes the paper.

2 DESIGN GOALS AND REQUIREMENTS

At-least-once. For all of our current use-cases, it is sufficient to ensure at-least-once semantics, where each task is guaranteed to execute but may, in some cases, execute more than once. Tasks that affect only the database could execute exactly-once using FoundationDB's support for ACID transactions. In practice, many tasks affect services outside the database. When a task is part of a larger flow, stronger functionality is often better achieved at a higher level in the stack, as suggested by the end-to-end argument [23].

High Availability. Our solution must provide the same fault-tolerance and availability for asynchronous tasks as we do for stored data. Many tasks, while not requiring low latency, are user-impacting. For example, failing to build a FoundationDB Record Layer index may cause client requests requiring the index to fail; in this case, the impact is temporary since another task will be queued and complete the build. Some types of tasks might not be re-submitted by the application, and losing them may have a prolonged effect. For example, failing to update a Solr index means clients querying that index may continue to see stale data. In another example, when data is modified in CloudKit, an asynchronous push-notification task may be created. Without this notification, other clients may not learn about this change until much later, when another change occurs or the client proactively polls the database.

Low overhead. The overhead of reliably storing a task and guaranteeing high-availability should be low enough to justify deferring the task, rather than executing it in-line. In our chosen design, the overhead of replication for tasks is negligible as it amounts to one or two additional keys in an existing FoundationDB transaction. Concurrent enqueues to the same queue cannot conflict since they update different keys in the database.

Scalability. In a cloud service deployed globally and serving hundreds of millions of users, sharding is essential for scalability. In CloudKit, since client requests often spawn tasks that operate on that user's data, the sharding of the tasks and the data should be complementary. Further, when CloudKit rebalances logical databases, any deferred work must "follow" the data, so that

they can be reliably executed at the destination. Storing and moving tasks together with data addresses this requirement.

Fairness. In a multi-tenant system different apps do not generate database load or deferred tasks uniformly, particularly over time. It is important to guarantee fairness across users and apps by allocating resources in a manner that precludes heavier users from starving smaller workloads.

Operations and monitoring. In any production system, it is important to be able to correctly provision the system, detect and handle problems, and handle unexpected load. For example, a corrupt task should not block the whole system, and permanent failures should be identified and treated differently from transient ones. It is also important to monitor and understand the workload of different users and apps. Storing tasks in CloudKit enables re-using existing tools, which were developed for other types of data in our system.

3 CHOOSING A SYSTEM FOR TASK MANAGEMENT

In this section, we connect these requirements to QuiCK's design, specifically our decision to embed tasks with the logical databases they affect. We contrast this design with one employing a canonical, dedicated external system for deferred task management. In Section 3.1 we address common, abstract concerns using ACID DBMS systems for queuing.

Queuing and Workflow Management systems frequently support at-least-once, and sometimes exactly-once semantics [4]. This is not sufficient to provide transactionality *between* CloudKit and external systems, as one system may commit while the other aborts or fails.

Consider the case where only CloudKit commits. While often recoverable, elided tasks add complexity by requiring future accesses to recognize the inconsistent state and schedule redundant tasks. Recovery code recognizing the effect of an elided task will rarely share a path with a trigger that would typically produce it, complicating maintenance. Some triggers like push notifications on updates leave no footprint; these rely on subsequent triggers or manual refresh for recovery.

Conversely, if the asynchronous task is registered before CloudKit commits, the system endures redundant execution and race conditions. Returning to the push notification example, if the CloudKit operation fails or aborts, then clients will receive spurious notifications. Since the notification is delivered in race with the database update, it is also possible (though unlikely) that the client will refresh before the update commits.

Achieving at-most-once semantics between the systems is also not trivial. If both work items and data are in the same database, a task can be executed and removed from the system transactionally.

Modern queuing and Workflow Management systems provide fault-tolerance and guarantee high-availability. Consider the effect of cross-datacenter replication on deferred task overhead. In a combined system, both a cloud service and the queuing/workflow system must replicate to multiple geographical locations before successfully completing client requests. Executing these replication tasks sequentially adds tens of milliseconds to request processing and hence isn't acceptable. Executing them concurrently complicates recovery and introduces the aforementioned race conditions. High cross-datacenter latency expands the hazard window, raising the probability of these faults affecting the request.

An external system also affects the availability of CloudKit itself. If that system is down or unreachable, or if storing a task results in an error, CloudKit may reject client requests. Executing asynchronous tasks in-line may be prohibitively expensive, significantly impact request latency, or increase the number of rejected client requests.

Systems deployed at scale calculate costs not only- and perhaps not even principally- by measuring processing overheads. Rather, the amortized cost of operating the service includes the developer and capital resources necessary to provision, monitor, and tune each system in a request pipeline. The seam between the database and task queuing system includes heuristics maximizing their combined throughput, capacity planning for growth of both systems, and health/observability metrics actively monitored by engineers. Using one highly-available storage system for both data and tasks lowers operating costs when the heterogeneous workload adds less complexity than the separate, specialized systems.

Continuing the topic of operability, recall that CloudKit operators rebalance logical databases to manage capacity and load on sharded FoundationDB clusters. Databases may also move to satisfy administrative or data policy objectives, changing the context in which the task runs. Regional differences in data-management policies may further complicate task execution and make their effect harder to predict. Storing queued tasks with user data allows us to move them together. Furthermore, with QuiCK, while CloudKit still needs to manage hundreds of millions of user databases, it neither separately tracks their asynchronous tasks, nor coordinates data moves with another system.

Contrast co-location with workarounds for an external system. CloudKit could wait for deferred tasks to drain. Client requests adding new deferred tasks would be failed, the external system would guarantee no tasks affecting the database are queued, and executing tasks (including retries) would also be drained. This would make the system unavailable to clients (but available to the task executor), adding significant coordination complexity to the back-end. Alternatively, queued tasks could be allowed to operate on logical databases in other datacenters. This entails a security policy that includes provenance for the logical database as it could be moved more than once, retry logic for a "moving" database, and undefined ordering with respect to tasks enqueued at the destination. To complicate things further, a database could be moved *back* to a context where a zombie task could address it. Since rebalancing logical databases is core to CloudKit, avoiding this complexity strongly motivated co-locating databases and tasks.

Finally, the tenant of a queuing system is usually a topic; it is the unit of resource allocation, sharding, monitoring, and ordering. In CloudKit, a tenant is a logical database, or a set of logical databases owned by a user; modern queuing systems scale to thousands of topics [6], not billions. By way of example, a topic in the external system QuiCK replaced is the type of asynchronous task generated by CloudKit. This mismatch between the tenancy models created friction that limited development in practice. For example, sharing resources fairly within any subset of CloudKit users requires an imprecise and impractical sub-partitioning for every topic. Querying outstanding work by user is inexpressible. In QuiCK, we inherit CloudKit's native tenancy model to group queued items by user and app, permitting explicit treatment of users in our scheduling

and resource allocation policy. We retain monitoring of aggregate metrics per topic, and differentiate work items of different topics for each user.

3.1 Queues in a DBMS

Next, we discuss three commonly brought-up areas of concern for representing queues in a database. The first concern is that the queue typically becomes a hot-spot because the same database table is used for both enqueuing and dequeuing tasks at a high rate. Our design circumvents the issue by having a two-level queuing system: on the bottom level there are tens of billions of independent queues embedded within logical databases in CloudKit (represented as *zones*, described in Section 5). Top-level queues, currently one per FoundationDB cluster, index non-empty queue zones. These queues can be trivially sharded further by logically partitioning the keyspace. This extreme sharding, together with a careful design of the operations in our system, helps us avoid hot-spots and contention.

The second concern is usually around management of concurrent consumers. Specifically, to avoid contention between multiple consumers attempting to execute the same task, queue implementations use or simulate locks on tasks. This creates lock management overheads and significant delays (or starvation) when a lock-holder fails. QuiCK assigns work to consumers at the granularity of a queue zone, rather than individual items, and uses a fault-tolerant, time-based leasing mechanism that automatically makes the queue zone available to other consumers if the lease holder is slow or fails. Leases on individual items further help prevent duplicate execution and improve concurrency, but they are only taken after obtaining a coarser queue-level lease.

Finally, dedicated queuing systems usually support push-based mechanisms to notify consumers of available items in the queue, while using a database often requires consumers to poll to find new items. A push-based solution, however, only makes sense if the rate of event production is relatively low [26]. In our case, we have a relatively small pool of consumers responsible for an extremely large number of queue zones, so a polling-based solution is much more appropriate. Furthermore, QuiCK doesn't react to individual work-item enqueue events, and instead chooses which queue zones to service using scheduling, fairness, and resource allocation policies.

4 FOUNDATIONDB, THE RECORD LAYER AND CLOUDKIT

This section gives a brief introduction to FoundationDB [9], the Record Layer [19], and CloudKit [25], emphasizing the particular features QuiCK relies on.

FoundationDB is an open-source, distributed, ordered key-value store that provides ACID multi-key transactions with strictly-serializable isolation, implemented using multi-version concurrency control (MVCC) for reads and optimistic concurrency for writes. Neither reads nor writes are blocked by other readers or writers; instead, conflicting transactions fail at commit time and are usually retried by the client. A client issuing a transaction first obtains a read version, chosen as the latest database commit version, by making a `getReadVersion` (GRV) call and then performs reads at that version, effectively observing an instantaneous snapshot of the database. Transactions that contain writes can be committed only if none of the values they read have been modified by another transaction since the transaction's read version. Committed transactions are written to disk on multiple servers in the FoundationDB cluster and then acknowledged to the client.

FoundationDB allows clients to customize the default concurrency control behavior by trading-off isolation semantics to reduce transaction conflicts. *Snapshot reads* do not cause an abort even if the read key was overwritten by a later transaction. One can also explicitly add or remove read or write conflict ranges. Read versions can be cached and re-used within a 5 second time window (the transaction time limit currently imposed by FoundationDB). Using cached versions may cause read-only transactions to return older data. Since read-write conflicts are checked at commit time, read-write transactions would still see the latest writes but potentially suffer from increased rate of aborts. Finally, FoundationDB exposes a "causal-read-risky" knob, which speeds up operations by avoiding certain validations during getReadVersion. The effect is similar to that of caching a read version.

FoundationDB provides basic CRUD operations on keys within a transaction. In addition, atomic read-modify-write operations on single keys (e.g., addition, min/max, etc.) are supported. These operations do not create read conflicts, such that a concurrent change to that value would not cause the transaction to abort.

QuiCK leverages all these features: it uses multi-key transactions across different parts of a cluster's keyspace and relies for correctness both on FoundationDB's ACID guarantees and on its advanced isolation APIs. QuiCK uses version caching and relaxed validation for optimization. Finally, we use atomic operations to implement efficient counters (exposed as a Record Layer count index) used to log the number of elements in each queue.

FoundationDB provides a minimal and carefully chosen set of features. *Layers* are built on top to provide various data models and other capabilities. Currently, the FoundationDB Record Layer is the most substantial layer built on top of FoundationDB. This layer is an open-source record-oriented data-store with semantics similar to a relational database. The Record Layer provides schema management, a rich set of query and indexing facilities including transactional index maintenance, and a variety of features that leverage FoundationDB's advanced capabilities. It inherits FoundationDB's strong ACID semantics, reliability, and performance in a distributed setting. The Record Layer is designed to provide multi-tenancy at an extremely large scale. It allows creating isolated logical databases for different tenants in the system, and limiting the resources consumed by each tenant and operation.

CloudKit is Apple's cloud storage service and app development framework. It provides structured strongly-consistent storage, queries, subscriptions and push-notifications, cross-device sync, and cross-user sharing capabilities. CloudKit powers many of Apple most popular apps, as well as by many externally developed apps. CloudKit allows rapid app development and at the same time helps developers by evolve their app by supporting schema evolution and enforcement. CloudKit provides apps with logical private and public databases. Each user of an app is assigned its own private database only accessible by the user. A single public database is shared and accessible by all users of an app. In addition, explicit sharing relationships can be established to allow one user to view or modify

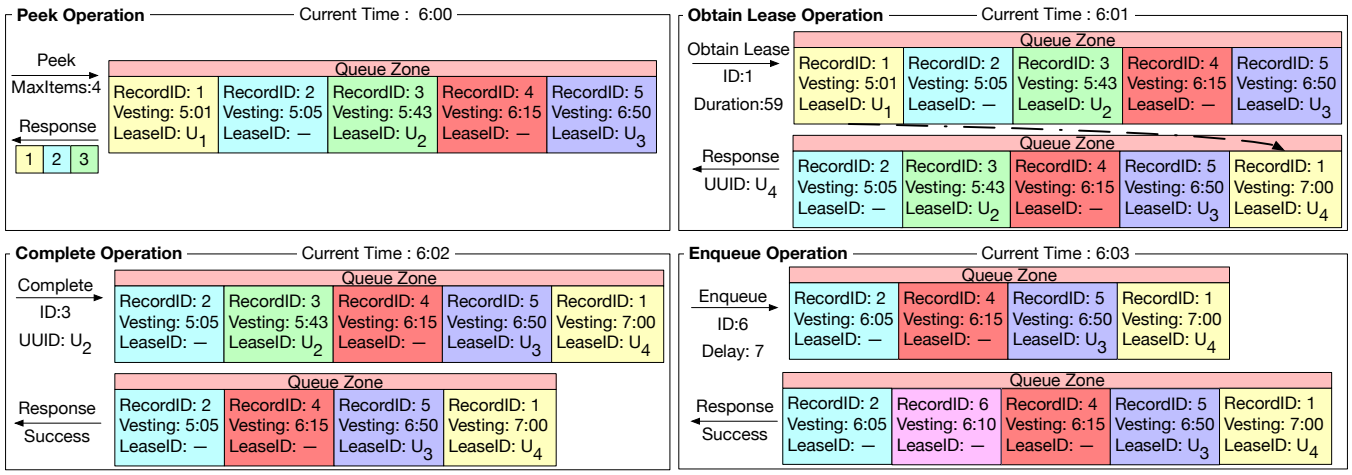

Figure 1: Peek, Obtain Lease, Complete, and Enqueue operations.

the data of another user. CloudKit achieves excellent scalability by assigning each logical database to a particular physical datastore, such as a FoundationDB cluster. Currently, CloudKit stores hundreds of billions of logical databases. Each logical database is divided into zones that store individual records. Zones can be synced across devices and shared among users; they are similar to directories (rather than database tables) in that they can store a mix of records of various types.

QuiCK leverages CloudKit's logical databases and zones, its schema management, sharding, high-availability, user-migration capabilities, and more. We have extended CloudKit to support a logical database pinned to a particular FoundationDB cluster (ClusterDB, mentioned in Section 6) and queue zones (described in Section 5). QuiCK relies on Record Layer's transactional index maintenance, query capabilities, schema support and more. We've extended CloudKit to support arbitrary transactions across multiple keys in the same FoundationDB cluster.

5 CLOUDKIT QUEUE ZONES

This section describes the design and implementation of queue APIs in CloudKit, using the FoundationDB Record Layer. Similarly to other data in CloudKit, queued items are stored in CloudKit zones, and can be created in any type of CloudKit logical database. This allows queues to take advantage of CloudKit data sharding.

Designating a zone to act as a queue is done upon its creation. CloudKit zones are different from database tables in that they can contain a mixture of record types. In particular, any type of CloudKit record can be enqueued. Queue zones maintain the following metadata about enqueued records: item priority, lease identifier, vesting time, and error count. Priority is an integer value, and lower value means higher priority. Vesting time is the wall clock-time when the item will be visible to consumers, which allows implementing delayed work-items that take effect in the future. Items in the queue are ordered by increasing priority and then vesting time, using a secondary index defined on the pair of fields.

Leases are used to avoid work duplication. When taking a lease on a queued item, the consumer is assigned a random lease identifier, used to restrict operations on the item to the lease holder, for the duration of the lease. Lease duration is determined by the consumer and should be chosen to allow sufficient time to process the item (or to extend the lease); at the same time, it should be short enough to allow other consumers to take over promptly in case a leaseholder fails. We implement fault-tolerant leases by using vesting time: when taking a lease, instead of removing the work item from the queue (and risking losing it if the consumer fails), we increase the work item's vesting time by the lease duration. This makes the item unavailable to other consumers, but only for the duration of the lease. Queue zones support the following main operations, illustrated in Figure 1:

- *enqueue(queuedItem, vesting delay, optional item id).* This operation adds a queued item to the zone. Vesting time is calculated by the CloudKit application server as current server time + vesting delay. This operation returns a randomly generated item id (by default), corresponding to the queued item. Alternatively, the item id can also be specified by the client (e.g. to enable idempotent operations).
- *peek(max_items, optional query predicate)* returns up-to *max_items* queued items in (priority, vesting time) order, that match a predicate and whose vesting time has passed (as evaluated by the CloudKit application server). Optionally can return only the record ids, rather than actual records.
- *obtain lease(item id, lease duration)* takes a lease on a queued item with the given identifier for the requested duration, by updating the vesting time to current time + lease duration. A random lease identifier is generated by the server and returned. The queued record is updated with the new lease identifier and vesting time.
- *complete(item id, optional lease id).* Used to delete an item after it has been consumed. If lease id is provided, this operation is used to indicate completion of the work item, and it succeeds only if the id matches the one in the record. If lease id isn't provided, the operation is used to cancel a queued item.

In addition, the queue API supports *dequeue*, *extend lease* and *requeue* operations. *dequeue* is a transactional combination of peek and obtain lease, *extend lease* is used to extend a lease and succeeds if a valid lease is held by the caller or if it has expired but no other consumer has leased the item. Finally, the *requeue* API updates

a queued item's vesting time and error count, and is useful for implementing re-tries to handle processing errors, as well as to release a lease.

If consuming a queued item has any side-effects in the database, the consumer may benefit from committing these side-effects and completing the item in the same transaction. This way, if another consumer has taken ownership or completed/cancelled the item, the side-effects will be aborted. If the operation fails, the response will indicate whether the lease is no longer valid or the queued item no longer exists.

Applications wishing to avoid duplicate work may choose to perform a transaction to obtain a lease before processing an item. If, on the other hand, the application partitions work between consumers in a way that duplication is unlikely, it might choose to use peek without obtaining a lease, or even to execute a single transaction to peek, process and complete the item.

It is easy to see that, despite its name, queue zones do not provide strict FIFO guarantees – vesting time is calculated based on an enqueuing server's local time, and leases might cause queued items to be re-ordered. In practice, the resulting order is sufficiently close to FIFO; strict FIFO guarantees are rarely required, and in some case are implemented by the application (e.g., using application level sequence numbers). In the future, if stronger FIFO guarantees are infact required, we can leverage FoundationDB's commit timestamps to order queued items, rather than relying on local server clocks (a similar mechanism is used to implement CloudKit sync [19]).

Push notifications. While several backend services currently use CloudKit queue zones, it isn't currently exposed directly to mobile apps. In order for mobile clients to leverage this API, it needs to be combined with CloudKit's subscriptions and push-notifications, to avoid polling. Note, however, that delayed queued items may become available for dequeue just by the virtue of time passing and without any external updates to the database that could trigger notifications. A potential way to achieve this is to notify clients when a queued item is enqueued and placed at the front of the queue, rather than when it vests. Then, CloudKit's client daemon could set a timer and notify the relevant mobile app when it's time to consume an item from the cloud database. Note that this is different from QuiCK's use of queue zones. In QuiCK, a shared pool of consumers is responsible for hundreds of thousands / millions of queue zones, which, as mentioned in Section 3, makes a push-based solution neither efficient nor desirable.

6 QUICK DESIGN

QuiCK stores the asynchronous work items of each CloudKit user in their own separate queue zone (for each application separately). Namely, the queued items for a logical CloudKit database *DB* are stored in a queue Q_{DB} under *DB*'s key-space prefix in FoundationDB. Storing the queued items in the same logical database as the data benefits locality of access and leverages the high-availability FoundationDB provides for all data. This also shards queued items into hundreds of millions of small queues, achieving scalability and helping to avoid hot-spots.

This approach, however, presents a challenge – how do consumers find logical databases with non-empty queues? Intuitively, this could potentially be achieved using a Record Layer index – an index that would automatically and transactionally track all non-empty queues in a FoundationDB cluster. Unfortunately, such an index does not align with the Record Layer abstractions – all indexes need to be contained within a logical database (more precisely, within a record store) rather than across them. Instead, QuiCK maintains an application-level index, represented as yet another queue, which we call a *top-level* queue. This queue is stored in a special type of logical database, *ClusterDB*, created in CloudKit for QuiCK, which is always pinned to a particular FoundationDB cluster, and exists in every FoudnationDB cluster in our system. The top-level queue for a FoundationDB cluser C is denoted Q_C and contains pointers to queue zones in the same cluster. While currently a single top-level queue per cluster is sufficient for our use-cases, more queues can be created for scalability by sharding the key-space. An illustration of the high-level design is presented in Figure 2.

Enqueue. When enqueueing a work item in cluster C, QuiCK transactionally adds it to Q_{DB} and enqueues to Q_C a pointer to Q_{DB} if one does not already exist for this queue zone. Usually, enqueue happens when processing a user's request, which often involves updating other data of the user in CloudKit. In that case, enqueue is done within the same transaction, achieving atomicity between data updates and deferred work. While this always ensures that every non-empty queue zone can be found by consumers through a pointer in Q_C, the opposite isn't necessarily true – dangling pointers that reference empty queue zones could exist since QuiCK garbage-collects pointers lazily. Note that transactionally updating or querying Q_C when enqueueing an item or deleting pointers (described below), requires transactions that cross the boundary of logical databases (still staying within one FoundationDB cluster). To support this, we've added such transactions to CloudKit (exploiting FoundationDB's transactions across its entire keyspace). The enqueue operation is illustrated in Figure 3; some details of enqueue are covered below where they're relevant.

High-level scheduling algorithm. A pool of consumers is assigned multiple FoundationDB clusters, each with a top-level queue. Each consumer consists of one Scanner thread, a pool of Manager threads, and a pool of Worker threads. The Scanner performs round-robin (in random order) over all top-level queues and executes the pseudocode shown in Algorithm 1. When processing a queue, the Scanner performs the following actions: (a) issues a transaction to *peek* vested (non-leased) pointers, and (b) selects some of the pointers and adds them to the local job queue. A pool of Manager threads consumes pointers from this queue, processing each pointer using Algorithm 2. The Worker threads handle work items, following Algorithm 3. Actions (a) and (b) (Algorithm 1, lines 4-11) are executed repeatedly, whenever Managers or Workers have insufficient tasks to process. In (b), the Scanner filters out pointers that are already being processed by its Manager threads. The scanner moves on to the next top-level queue if insufficient pointers were peeked or a certain number of pointers were processed.

A Manager thread (i) performs a transaction to *obtain lease* on the pointer. By taking a lease, it effectively reserves the entire queue zone rather than individual queued items, (ii) batch-dequeue up to *dequeue_max* items (the total transaction size must also stay within certain bounds imposed by FoundationDB) and passes each item separately to the Workers pool, and finally (iii) decides whether

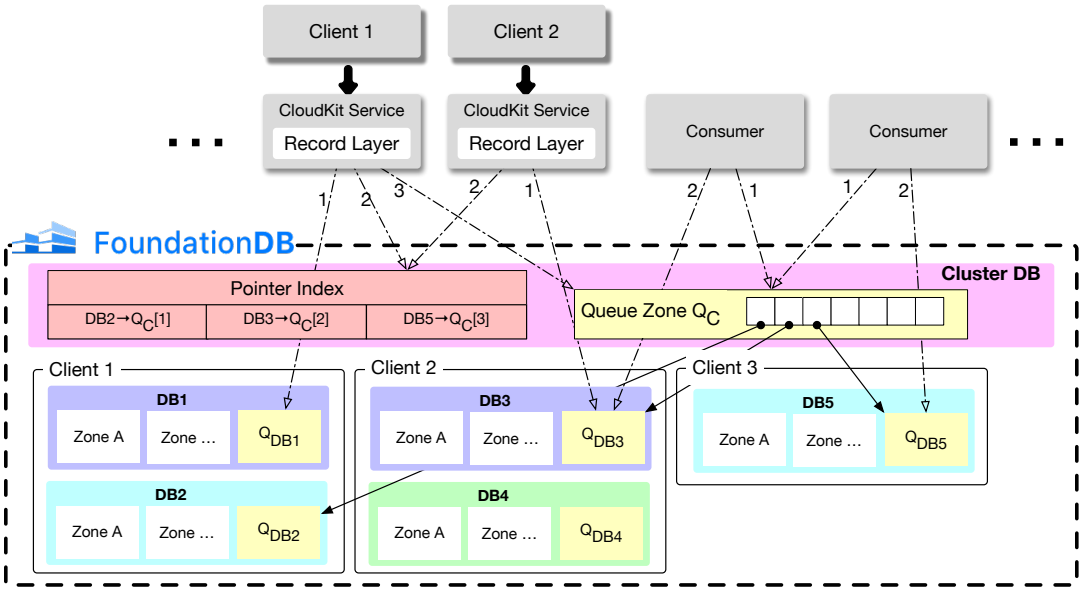

Figure 2: QuiCK architecture using FoundationDB and the Record Layer.

Algorithm 1: QuiCK's Scanner algorithm.

Input: C_{IDS} - a list of FoundationDB cluster IDs
parameter: *peek_max* - max number of pointers to peek
parameter: *selection_frac* - fraction of peeked pointers to process
parameter: *selection_max* - max peeked pointers to process
parameter: *processing_bound* - max pointers to process

1 shuffle(C_{IDS})
2 **for** $c \in C_{IDS}$ **do**
3 $processed \leftarrow 0$
4 **while** *processed* < *processing_bound* **do**
5 **wait** until at least one worker has no task to process
6 $peeked\_list \leftarrow$ **peek**(*peek_max*)
7 remove from *peeked_list* pointers already being processed by this consumer's Managers
8 *pointers* ← select `min`(*selection_max*, ⌈**size**(*peeked_list*) ∗ *selection_frac*⌉) pointers from *peeked_list*, uniformly at random
9 $processed \leftarrow processed +$ **size**(*pointers*)
10 Pass *pointers* to Managers pool, each running Alg. 2
11 **end**
12 **end**
13 **go to** 1

to delete the pointer, requeue it, or do nothing. In step (ii), recall that dequeueing an item implies taking a lease. We enforce a per-consumer limit on the number of work items of each type that can be processed concurrently; for throttled items, we may release the lease, or simply let it expire. If the pointer is requeued in step (iii), its new vesting time depends on whether or not items were found in the queue, and the earliest vesting time of any such item. For example, if the earliest vesting time of any item in the queue is 10 seconds from now, the vesting delay is set to 10 seconds. If a new item is enqueued that needs to be processed earlier, the pointer's vesting time is changed during enqueue to make it available sooner.

A Worker receives a leased item, double-checks that it is still leased, and starts processing it. In parallel, another thread periodically attempts to extend the lease, as long as the item is being processed; if this fails, processing is interrupted. Execution time is bounded, to mitigate faulty work items. If processing successfully completes, the Worker deletes the item. If an error occurs during work item processing, the Worker may retry immediately, and, if still unsuccessful, *requeue* to update the queued item's error count and vesting time using an exponential backoff algorithm (based on the error count). The same retry logic is used by Managers when dealing with pointer errors. We classify errors into transient (e.g., contention or trouble reaching the database or other sub-systems) and permanent (e.g., the user was deleted), and do not retry on permanent errors. In such cases, the item or pointer is deleted immediately. Finally, each type of queued items (i.e., a job type) can set its own retry policy, for example how many in-line retries to perform, whether to continue retrying indefinitely (this would eventually cause alerts and manual mitigation) or to discard the job after a certain period of time.

Concurrency between consumers, fairness and leases. Ideally, pointers appearing earlier in the top-level queue should be processed first, since they correspond to queues that weren't accessed by consumers for a longer time. Processing work items sequentially, however, would cause all consumers to contend on the same pointers. Hence, the choice in step (b) is done at random by most Scanners, which aims to improve concurrency and reduce contention. Specifically, Scanners select their pointers randomly out of a larger set of pointer ids (lines 6 and 8, Algorithm 1). By doing so, however, we might starve pointers that appear earlier in Q_C, or, at the very least, increase the tail latency of processing their work items. To address this problem, we occasionally select one Scanner to process

pointers in-order rather than randomly. This guarantees better tail-latency and no starvation. This consumer is selected dynamically (separately for each top-level queue Q_C), by taking a lease on a shared memory object (using memecached). In cross-datacenter deployments, only consumers in one of the datacenters attempt to take this lease (details are beyond the scope of this paper).

As an optimization, the peek on line 6 in Algorithm 1 only scans entries of the vesting time index, to get the pointer record ids, while Manager threads access the actual pointer records when needed. Scanning the index especially benefits randomized consumers since they select only a small fraction of the pointers for processing.

With pointer leases, consumers contend on queues rather than on individual work items. Usually, pointer leases are short (on the order of 1 second), which is sufficient to read work items from the database and pass them to Workers. A lease effectively moves the pointer to the end of the top-level queue, after other vested pointers. This allows other queue zones to be found by Scanners before the same queue is picked up again (of course, randomized pointer selection in step (b) means that this isn't a strict guarantee). Pointer leases, together with the upper-bound on the number of items to consume from each queue (*dequeue_max*), ensure that we don't starve smaller queues by spending too much resources on larger queues and guarantee fairness across different queue zones (and effectively, across users and apps). Work item leases, on the other hand, minimize duplicate processing by temporarily "hiding" work items already being processed, and increase concurrency by allowing other Managers to dequeue additional work items from the same queue. Dynamically extending the leases allows Workers to hold a lease for the duration of processing, without predicting the run-time in advance. If a Worker fails, a short lease duration allows Managers to promptly find and pass the item to other Workers.

Algorithm 2: Manager thread: pointer processing.

Input: p - a pointer to Q_{DB}
Input: c - a cluster ID, such that p is stored in Q_c
parameter: *dequeue_max* - max allowed items to dequeue
parameter: $p\_duration$ - pointer lease duration
parameter: *min_inactive* - min time a queue remains empty before its pointer is deleted

1 **if not** $id \leftarrow$ **obtain_lease(**$p, p\_duration$**) then**
2 **return** CONFLICT
3 is_active $\leftarrow$ **not** is_empty(Q_{DB})
4 $p_t \leftarrow p$'s last active time
5 *items* $\leftarrow$ **dequeue**(*dequeue_max*) from Q_{DB}
6 Pass *items* to Workers pool, each running Alg.3
7 min_vesting_time = read minimum vesting time in Q_{DB} // negative if Q_{DB} is empty
8 **if** $0 \leq$ *min_vesting_time or is_active* **then**
9 vesting_delay = MAX(0, min_vesting_time - now)
10 **requeue**(p, $vesting\_delay$) // also update p's last active time
11 **else if** $now - p_t \geq$ *min_inactive* **then**
12 **complete**(p, id) in Q_c
13 **return** SUCCESS

Algorithm 3: Worker thread: work item processing.

Input: *item* - a leased work item of type t in Q_{DB}
Input: *id* - a uuid of the lease on w
parameter: $execution\_bound_t$ - max allowed execution time for work item of type t

1 **do in parallel**
2 process *item* (bounded by $execution\_bound_t$)
3 **if** *failed to process* item **then**
4 $new\_delay \leftarrow$ calculate with exponential back-off, based on number of failures
5 **requeue**(*item*, $vesting\_delay = new\_delay$) in Q_{DB}
6 **else**
7 **complete**(*item*) in Q_{DB}
8 **end**
9 **end**
10 **do in parallel**
11 Periodically **extend_lease**(*item*) in Q_{DB}
12 **end**

Reducing contention between producers and consumers. QuiCK was designed to avoid unnecessary contention between clients enqueueing work items, and consumers processing them. From the algorithm above, it is easy to see that the pointers in Q_C are the only potential point of contention: enqueueing clients might read the relevant pointer record to determine whether one exists or needs to be created, while consumers update pointers when taking a lease, requeueing or deleting them. FoundationDB's optimistic concurrency control makes sure that if the pointer is updated or deleted after the enqueueing transaction's read but before its commit, the enqueueing transaction will abort. Such aborts are very undesirable since they might fail client requests. To avoid potential contention due to updates to the pointer record, enqueueing an item is divided into two parts. The first, which may be performed transactionally with the client request, doesn't actually involve reading the pointer. Instead, we maintain a Record Layer secondary index that maps each logical database identifier to a top-level queue pointer in Q_C, if one exist. This index is updated only on pointer creations or deletions, but never on updates. Hence, the only possible point of contention can happen when a pointer is deleted by a consumer.

The second part of an enqueue is a separate transaction, which reads the pointer record and updates its vesting time in case the existing vesting time is too far in the future and would therefore introduce a significant delay for the new work item. This transaction is an optimization and cannot fail a client's request.

Pointer garbage-collection. Each pointer to an empty queue stores the *last active time* we've observed work items in the corresponding queue zone. We delete the pointer only if a sufficient amount of time has passed since it was "active". This allows us to avoid repeatedly creating and deleting the same pointer by having a grace period in which the queue could be empty and work items could be enqueued cheaply without the need to create the pointer anew, which could potentially cause contention with consumers as mentioned above.

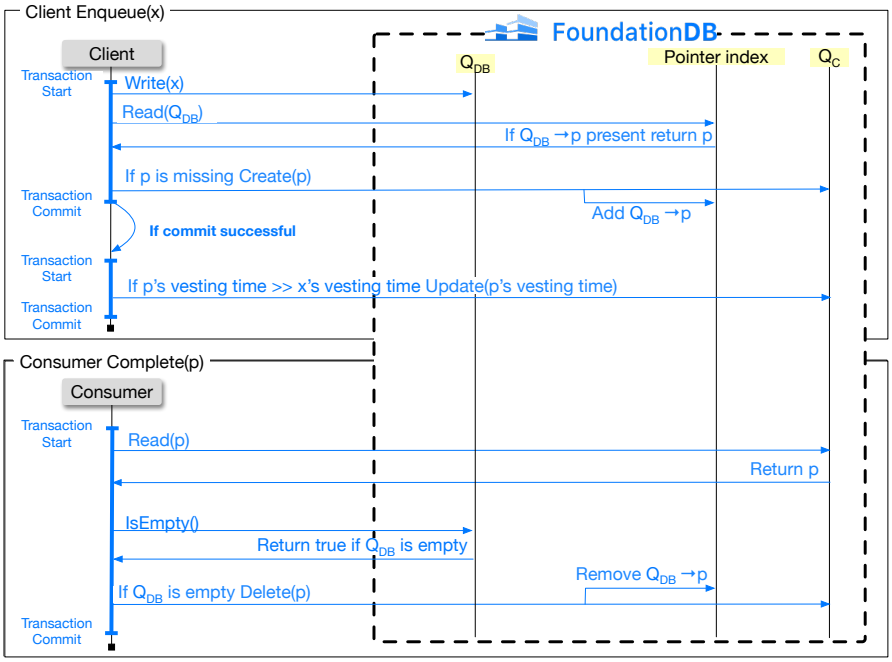

Figure 3: Client's enqueue and Consumer's complete operations.

Correctness. The key to correctness (i.e., at-least once semantics) is making sure that once a work item has been successfully enqueued, consumers can find and execute it. A consumer can miss the work item only if we delete a pointer to a non-empty queue, or if a vested pointer is never selected. The latter is prevented by having one of the consumers process Q_C sequentially. Whenever the enqueue operation requires creating a pointer, this update conflicts with any potential delete (or other creates) causing one of the transactions to abort (both transactions read and update the same key in the pointer index). If, however, the pointer already exists, the conflict is achieved in two ways: (1) the process enqueueing the work item reads an index entry corresponding to the queue zone's pointer in Q_C (see above), while any deletion of the pointer transactionally updates the index. Therefore, if the deletion commits before the enqueue, the enqueuing transaction will fail. (2) the consumer scans the key-space of the queue zone to determine that it's empty before deciding to delete the pointer (the scan and deletion are done in the same transaction), thus if the enqueue commits before the delete, successfully inserting a record into the queue zone key-space, the delete transaction will fail. Finally, we note that when the side-effects of processing a work item are confined to the same FoundationDB cluster, processing and deleting the work item in the same transaction can guarantee that the work item isn't processed multiple times, achieving exactly-once semantics.

Isolation level. FoundationDB's strict serializability semantics are essential for QuiCK's correctness. For example, a consumer must observe recent work items written by users into queue zones, and similarly, when a pointer is deleted by a consumer, a user must detect that and re-create the pointer. As an example, snapshot isolation semantics wouldn't be sufficient to guarantee that any non-empty queue zone has a pointer in Q_C (an invariant maintained by QuiCK) – a user's enqueue operation and the consumer's garbage-collection of pointers write to different locations in the database and would not conflict under snapshot isolation. At the same time, QuiCK relaxes the semantics when observing latest writes isn't important. In particular, using cached read versions reduces transaction latency, at the cost of potentially returning stale values (at serializable isolation) for read-only transactions, and potentially increasing the rate of aborts for read-write transactions (which are still strictly serializable). The "causal read risky" FoundationDB flag has a very similar effect. We use both methods to reduce transaction latency for peeks, obtain lease operations, and when reading the count of work items or pointers (for monitoring). Note that these aren't used when enqueueing work items since we do not wish to increase aborts for user-facing operations and neither for requeue transactions, which are used to record errors.

User-move and local work items. When a logical database (e.g., a user's data) is moved to a different FoundationDB cluster, the pointer index is queried to determine whether a pointer exists to this database. If so, this pointer is copied to the destination. This is done after copying the data, so that the pointer isn't prematurely deleted by consumers at the destination. Note that some work items do not have an associated user or an associated logical database, for example, asynchronous re-indexing jobs related to many users. Such work items are usually fanned out and executed on all FoundationDB clusters in our system, or, alternatively are specific to a particular cluster. We call such work items *local* since (unlike other queued items) they never have to be moved to a different FoundationDB cluster, and enqueue them directly into the top-level queue, along-side pointers to queue zones.

6.1 Supporting External Data Stores

This section describes QuiCK's support for data that isn't stored in FoundationDB. In such case, co-locating data and work items

is still important to meet the requirements outlined in Section 2. Re-implementing QuiCK's scheduling in the different data-stores, however, is often complex and/or not possible since their semantics are usually much weaker than those provided by FoundationDB (for example, Cassandra does not provide transactions across its entire key-space and does not natively support secondary indexes). Hence, Q_C and the pointer index remain in FoundationDB, while the data and work items (everything below Q_C in Figure 2) are now stored in a different data-store. Since in such cases work items are not stored in FoundationDB, we lack transactionality between pointers and work items. We next describe how the algorithm described above is modified to guarantee at-least once semantics in this case, i.e., that work items can always be found by consumers.

A consumer's algorithm remains similar. Just like before, it obtains a lease on a pointer, checks whether there are any work items (this time in the external data-store), if not it deletes the pointer, and otherwise processes and deletes work items, moves on to the next pointer, and so on. Whether or not leases can be taken on individual work items, depends on the external store. If not, a longer lease on the pointer (in FDB) can be used. Note that the above-mentioned external data-store read must be a "strong read", namely guaranteed to observe any successfully written work items (that weren't deleted) or else the pointer might be prematurely deleted.

The enqueue operation is now slightly different. First, it stores the work item in the external data-store. Then, an FDB transaction is created, reads the pointer index in FoundationDB's ClusterDB and, if the pointer doesn't exist, creates it and commits the FDB transaction. The main complexity arises when the pointer exists. In this case, the FoundationDB transaction becomes read-only, since the work item itself is not written in the same transaction (it is stored in an external data-store). In this case, the correctness argument from the previous section no longer holds, since committing a read-only transactions in FDB doesn't cause any conflicts to be checked. One alternative to resolve this could be to always update the pointer, even if it exists. This would generate unnecessary database writes and cause significant contention between producers and consumers on the pointer record. Fortunately, FoundationDB provides an API that allows us to avoid the unnecessary pointer updates – the enqueue operation declares a *write conflict* on the pointer's key in the index. This causes the transaction's conflicts to be checked at commit time, just like with a real read-write transaction. Since the write conflict is declared on the index and not on the pointer itself, an enqueue operation happening concurrently with a consumer obtaining a lease on the pointer would not cause a conflict. If the enqueue happens concurrently with a consumer's attempt to delete the pointer, and the enqueue commits earlier, the delete transaction will abort since it reads the pointer key in the index, on which the write conflict was declared.

Enqueuing the pointer in FoundationDB is done after writing the work item in an external data-store, and it's possible that the transaction writing the pointer aborts. In this case, enqueue attempts to garbage-collect the written work item. In the worst case (if garbage-collection fails), some work items might remain in the external data-store. Note, however, that in this case the client's enqueue operation fails. If the pointer is later re-created, it is possible that such old work items are resurrected and executed. Our current use-cases are idempotent and aren't sensitive to executing work items more than once, although executing work items when the client request fails might in some cases result in unnecessary work. Double execution could be further prevented by the work item processing logic (e.g., by creating a unique record whose existence would prevent redundant execution). To some extent, these issues currently exist even if the work items are stored in FoundationDB – a CloudKit server failure occurring after the enqueue commits could cause the client's request to fail, retried, and result in double execution. Ongoing work in the FoundationDB project is aiming to address this issue [11].

7 RELATED WORK

Early research on task execution in Grid Computing, the predecessor of Cloud, focused on process-centric models where jobs are submitted and managed [20, 21]. Research later evolved to data-centric models that use relational DML to submit and manage jobs, SQL for querying and analyzing job results and so on [22]. While Workflow Management systems (WMFS) use a variety of datastores, the benefits of RDBMS integration for WFMS is widely recognized [24, 27, 28], particularly transactionality, SQL support, concurrency control, scalability, durability and administrative ease. Extract-transform-load (ETL) workflows are supported by many database vendors. Even early NoSQL workflow systems relied on a relational database under the hood [8].

Queueing systems similarly use a variety of storage engines. For example, Apache Pulsar uses Apache BookKeeper and supports tiered storage. Kafka and RabbitMQ store messages in log files. Many queueing systems support pluggable connectors, through which queued messages can be exported to structured and unstructured storage systems. For example, RabbitMQ supports both relational (e.g., PostgreSQL) and time-series (e.g,. InfluxDB) databases. Database systems can also generate queued messages. For example, Slony [17] uses PostgreSQL's trigger functions to send event messages, implementing primary-backup replication and a PostgreSQL extension, pg_amqp [16], that exposed AMQP publishing via PostgreSQL functions.

Oracle Database 9i introduced fully integrated queues [12], and more recently added support for partitioned queues in 12c [14]. Oracle Database 20c added Kafka support [1], allowing Kafka producers and consumers to use the database instead of a Kafka broker. Queues are automatically partitioned into database instances, and consumers are assigned an instance to dequeue from. QuiCK partitions queues on two levels – first, according to the CloudKit tenancy model, at a granularity as fine as a queue for each user of an app, and second, across many FoundationDB clusters. Hence, the number of queues managed by QuiCK is many orders of magnitude larger. Unlike in Oracle, consumers can dequeue from multiple clusters, and within a cluster do not contend on individual messages in queues but instead take a coarser lease on an entire queue.

Integrating queues within an ACID DBMS allows leveraging its high-availability, DML, transactionality with other stored data, observability, monitoring and other existing tooling, as well as to reduce dependencies on any external queuing system. Perhaps the most salient aspect of QuiCK is its support for CloudKit's tenancy model. CloudKit's scale, security, and generality hang on the assumption of isolation of every user's application data as an independent database. QuiCK adopts CloudKit's model to allocate

resources among tenants in a fair manner, provide per-tenant monitoring of queued items, and support tenant queue migration across clusters using its native tooling.

In Dropbox's ATF [2], like in QuiCK, deferred tasks are enqueued into a database, together with other user data. A service asynchronously pulls from the database and enqueues available tasks to Amazon Simple Queue Service [7] (SQS) queues partitioned by topic and priority. Pools of consumers, each allocated for a particular topic, pull tasks from SQS, take a lease on each task in the database, execute the task, and finally remove it from the database. SQS serves as a buffer between the database and workers. ATF uses both SQS and detailed status information maintained per task in the database to coordinate task execution. QuiCK does not depend on an external queuing service; workers pull directly from the database. Just like ATF, QuiCK maintains indexes that allow workers to find work items, but these are coarse-grained and hence more space efficient and less likely to become a hot-spot. ATF is indifferent to CloudKit's user-centric tenancy model, and focuses entirely on topic independence; fairness is based on strict priorities. QuiCK schedules execution to provide fair service to all CloudKit users. ATF is a service allowing client teams to provision a pool of consumers for a particular topic. We can approximate this by creating a top-level queue per app, such that a pool of consumers could be created to work exclusively on the queues of the app's users. In QuiCK, the allocation of Workers to items is dynamic – a shared pool of Workers handle all topics, and per-topic throttling bounds the number of work items of any topic processed concurrently, i.e., the resources dedicated to the topic.

Similar to Amazon SQS [7] and Azure Storage Queues [13], CloudKit Queue Zones, described in Section 5, support delayed message delivery and do not guarantee strict FIFO semantics. All three systems rely on delayed messages to implement leases. Finally, just like Queue Zones and QuiCK, these systems do not support push-based APIs. Unlike QuiCK, however, these systems do not support transactional operations; for example, with a Queue Zone, it's possible to enqueue multiple messages as an atomic batch, or to execute multiple API calls transactionally, such as to dequeue, execute, and delete a work item all within the same transaction. Apache Kafka supports transactions, but does not support delayed messages or leases. The latter, together with a static assignment of consumers to Kafka partitions, allows Kafka to guarantee FIFO message delivery. Oracle queues provide the most comprehensive set of features, including transactions, message locking (for the duration of a dequeue transaction), delayed messages, push notifications and configurable ordering (including FIFO) [15].

8 EVALUATION

In this section we evaluate QuiCK's performance. We focus on demonstrating its scalability, latency, fairness model and its handling of potential contention with multiple consumers.

Environment and Workload Generation. We evaluated QuiCK in a test cluster – a CloudKit partition [25] mimicking our production environment. In this setting, CloudKit was using a single FoundationDB cluster (version 6.3.10) in a highly-available configuration with 2 datacenters (~13ms median round-trip-time apart), each with its own "satelite" location (~5ms apart) used for hosting

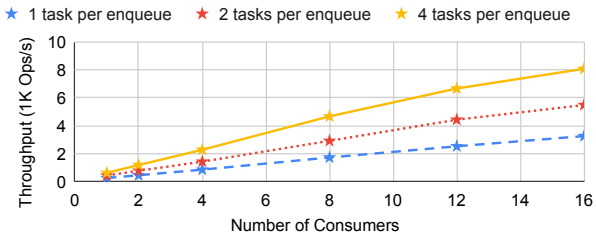

Figure 4: Saturation throughput.

FoundationDB's write-ahead logs. Commits are coordinated from a primary datacetner, applied synchronously to log replicas in the primary datacenter and satelite, and propagated asynchronous to the remote datacenter [3]. We used the latest FoundationDB Record Layer version (2.10.156.0).

QuiCK was executed on 1 to 16 servers, each with 128 system queue manager threads and 128 worker threads. Client load was generated by a separate pool of servers, according to a uniform or a skewed load distribution, simulating load from 150K distinct clients and one CloudKit app (corresponding to 150K queues in QuiCK). For uniform load, each client performed 1 enqueue operation every minute, each with 1 to 4 work items. Each work item simulated asynchronous workload of ~50ms. Skewed load was generated following Pareto distribution (with $\alpha = \log_4(5)$) to determine each client's enqueue frequency. In all experiments *peek_max* was set to 20K, and *selection_max* to 2K. Unless mentioned otherwise, we used $selection\_frac = 0.02$. In addition, we set *dequeue_max* to be equal to the number of tasks per enqueue.

Scalability. CloudKit uses hundreds of FoundationDB clusters. Since these clusters are independent, in this evaluation we've focused on QuiCK's performance with one cluster and a corresponding top-level queue. Figure 4 shows QuiCK's throughput at saturation. To reach saturation, we've gradually increased load by adding 100 new clients every second. We can see that maximum throughput scales linearly with the number of consumers (until hitting a network thread bottleneck [10]). As mentioned in Section 6, Managers take a pointer lease, dequeue up to *dequeue_max* vested work items in the zone and pass them to Workers. Throughput is hence considerably higher when multiple work items exist in the zone, since the cost of the pointer lease (and possible pointer deletion/update) is amortized.

Fairness. Figures 5 and 6 depict two test executions, one with uniform load and the other with skewed load, with a single consumer. In this case, we set *dequeue_max* to 1, meaning that 1 item was processed per queue visit, and since there is no need to avoid contention, the consumer was processing pointers sequentially. Specifically, we show (a) the time from a pointer becoming available (vested) till QuiCK starts processing the queue, and (b) the time from enqueuing of a work item till it is picked for processing by QuiCK. In Figure 5 (uniform load), both median and tail latencies are relatively low, and the time to start processing queued work items is equal to the time to pick up a pointer plus dequeuing a work item and passing it to a Worker, confirming that QuiCK gets to all queues within a short period of time and processes the same number of work items in each queue. On the other hand, with skewed

load, queue lengths ranged between 1 and 60 work items per queue. In Figure 6, we can see that even though all queues were picked up relatively quickly for processing (median ~105ms, 99.9% latency of ~305ms), and work item median latency was low (~193ms), tail latency for work items is much higher (~8.2 seconds). The reason is QuiCK's "water filling" strategy of spreading processing cycles among all queues, i.e., spending bounded time on each queue and returning to longer queues again later, rather than processing each queue to completion.

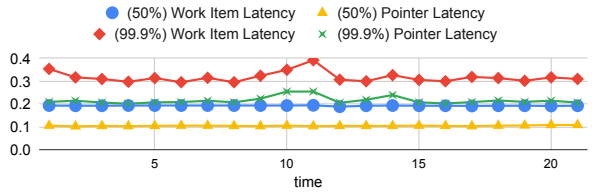

Figure 5: Latency (in seconds) - uniform load distribution.

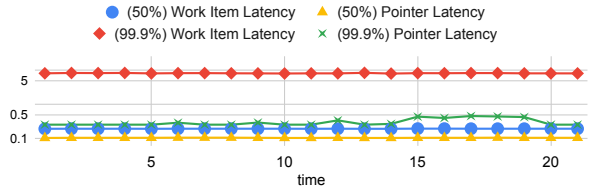

Figure 6: Latency (in seconds) - skewed load distribution.

Contention. Figures 7 shows the effect of $selection\_frac$ in Algorithm 1 on latency (top), contention (middle), and maximum throughput (bottom). In this experiment, client load was generated uniformly, with 1 item enqueued at a time, and 4 consumers were performing random selection of pointers, with $selection\_frac$ varied from 0.001 to 0.5. We can see that a smaller fraction results in less contention (since the chance of collision between consumers is lower), e.g., 0.31% of all attempts to obtain lease for $selection\_frac = 0.001$, but extreme median and tail latency (since it takes longer for a pointer to be chosen for processing). As we increase $selection\_frac$, failures to obtain lease go up until $selection\_max$ kicks in, limiting the number of chosen pointers, hence the flatter curve in the 0.1 to 0.5 range of $selection\_frac$. An extremely low $selection\_frac$ results in low maximum throughput, but with $selection\_frac$ of 0.005 and above we can see a relatively stable maximum throughput.

As the Scanner only reads the pointer index rather than the records themselves, it does not know whether pointers are leased. A Manager then tries to obtain a lease (Algorithm 2) by (a) reading the pointer record, and (b) performing a conditional update, writing a randomly chosen lease *uuid* (Section 5). If another consumer concurrently obtains the lease, a failure can happen in either step. A failure in step (a) is much cheaper, since its cost is a redundant read to FoundationDB, while a failure in (b) is generated during commit and results in extra load on FoundationDB's transaction management (namely, Resolver servers, enforcing strict-serializability with optimistic concurrency control). In our experiments, we saw that up until $selection\_frac = 0.2$, about 50% to 60% of all failures happen in step (b) and the rest in step (a). At 0.5, about 90% of all failures happen in step (a). In this case, most pointers passed to Managers are already leased, which is detected when a Manager reads the pointer in step (a). As the total percent of failures increases from 44% at $selection\_frac = 0.2$ to 64% at $selection\_frac = 0.5$, latency increases accordingly.

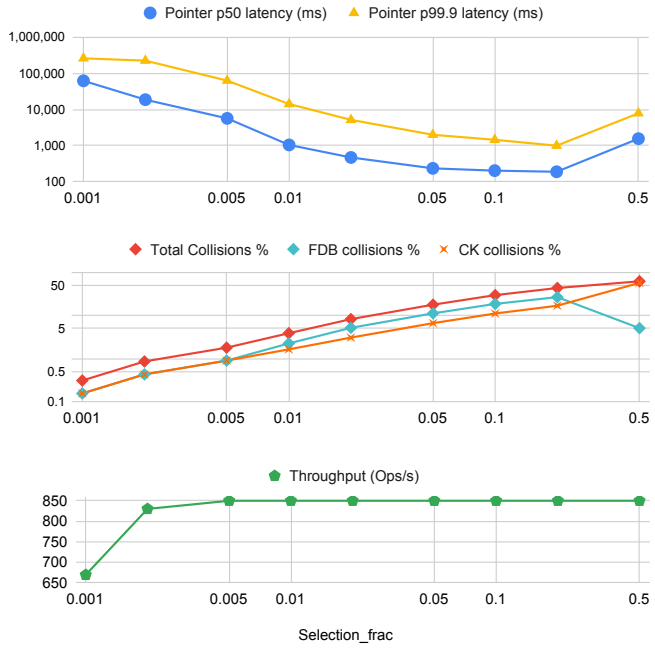

Figure 7: The affect of $selection\_frac$ on (a) median and tail latency (in milliseconds), (b) failures to obtain lease (as % of all attempts) and (c) maximum throughput.

9 CONCLUSIONS

When queues are needed to store information and tasks related to other data persisted in a database, leveraging the same database for queuing has many advantages. It can make the management of queued items simpler and their execution semantics w.r.t. stored data easier to reason about. Using the same database reduces the dependency of the system on external services, improves observability, and reduces operational complexity and overhead.

In this paper, we presented QuiCK, a queuing system built for managing CloudKit's asynchronous tasks. QuiCK stores tasks alongside data in CloudKit and fully supports its tenancy model. We demonstrated how QuiCK uses various features of FoundationDB and the FoundationDB Record Layer. Finally, we showed that QuiCK scales linearly with consumer resources, processes tasks with low latency and allocates resources fairly across CloudKit tenants.

ACKNOWLEDGMENTS

We thank the past and current contributors to QuiCK, CloudKit, FoundationDB and the Record Layer. We thank Ori Herrnstadt and Anestis Panidis for their support in the initial stages of this work. Finally, we thank the anonymous reviewers and Gabriel Kliot for their insights and suggestions that helped us improve the paper.

REFERENCES

[1] Advanced Queuing in Oracle Database 20. https://docs.oracle.com/en/database/oracle/oracle-database/20/newft/advanced-queuing.html, year=2020.

[2] Dropbox's ATF - an async task framework. https://dropbox.tech/infrastructure/asynchronous-task-scheduling-at-dropbox.

[3] FoundationDB cross-data-center configuration. https://apple.github.io/foundationdb/configuration.html#changing-the-region-configuration, year=2020.

[4] Kip-98 - exactly once delivery and transactional messaging. https://cwiki.apache.org/confluence/display/KAFKA/KIP-98+-+Exactly+Once+Delivery+and+Transactional+Messaging.

[5] Why is a database not the right tool for a queue based system. https://www.cloudamqp.com/blog/2015-11-23-why-is-a-database-not-the-right-tool-for-a-queue-based-system.html, 2015.

[6] Apache Kafka Supports 200K Partitions Per Cluster. https://blogs.apache.org/kafka/entry/apache-kafka-supports-more-partitions, 2018.

[7] Amazon SQS. https://docs.aws.amazon.com/sqs/index.html, 2020.

[8] Apache Oozie. https://oozie.apache.org/, 2020.

[9] FoundationDB. https://www.foundationdb.org, 2020.

[10] Foundationdb issue - run one network thread per database. https://github.com/apple/foundationdb/issues/3336, 2020.

[11] Getting Rid of commit_unknown_result. https://bit.ly/3flSqw5, 2020.

[12] Managing Job Queues (Oracle Database 9.2. https://docs.oracle.com/cd/B10501_01/server.920/a96521/jobq.htm, 2020.

[13] Microsoft Azure Storage Queues. https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-azure-and-service-bus-queues-compared-contrasted, 2020.

[14] Oracle Database 12c: JMS Sharded Queues. https://www.oracle.com/technetwork/database/jms-wp-2533807.pdf, 2020.

[15] Oracle Database Advanced Queuing. https://docs.oracle.com/database/121/ADQUE/, 2020.

[16] pg_amqp. https://github.com/omniti-labs/pg_amqp, 2020.

[17] Slony. https://www.slony.info/, 2020.

[18] Announcing The FoundationDB Record Layer. https://www.foundationdb.org/blog/announcing-record-layer/, 2021.

[19] C. Chrysafis, B. Collins, S. Dugas, J. Dunkelberger, M. Ehsan, S. Gray, A. Grieser, O. Herrnstadt, K. Lev-Ari, T. Lin, M. McMahon, N. Schiefer, and A. Shraer. Foundationdb record layer: A multi-tenant structured datastore. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1787–1802, New York, NY, USA, 2019. Association for Computing Machinery.

[20] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

[21] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor-a hunter of idle workstations. In *[1988] Proceedings. The 8th International Conference on Distributed*, pages 104–111, 1988.

[22] D. T. Liu and M. J. Franklin. Griddb: a data-centric overlay for scientific grids. In *Proceedings of the International Conference on Very Large Data Bases*, pages 600–611, 2004.

[23] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, Nov. 1984.

[24] S. Shankar, A. Kini, D. J. DeWitt, and J. Naughton. Integrating databases and workflow systems. *ACM SIGMOD Record*, 34(3):5–11, 2005.

[25] A. Shraer, A. Aybes, B. Davis, C. Chrysafis, D. Browning, E. Krugler, E. Stone, H. Chandler, J. Farkas, J. Quinn, J. Ruben, M. Ford, M. McMahon, N. Williams, N. Favre-Felix, N. Sharma, O. Herrnstadt, P. Seligman, R. Pisolkar, S. Dugas, S. Gray, S. Lu, S. Harkema, V. Kravtsov, V. Hong, Y. Tian, and W. L. Yih. Cloudkit: Structured Storage for Mobile Applications. *Proc. VLDB Endow.*, 11(5):540–552, Jan. 2018.

[26] A. Silberstein, J. Terrace, B. F. Cooper, and R. Ramakrishnan. Feeding frenzy: selectively materializing users' event feeds. In A. K. Elmagarmid and D. Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 831–842. ACM, 2010.

[27] M. Vrhovnik, H. Schwarz, S. Radeschutz, and B. Mitschang. An overview of sql support in workflow products. In *2008 IEEE 24th International Conference on Data Engineering*, pages 1287–1296. IEEE, 2008.

[28] M. Vrhovnik, H. Schwarz, O. Suhre, B. Mitschang, V. Markl, A. Maier, and T. Kraft. An approach to optimize data processing in business processes. In *Proceedings of the 33rd international conference on Very large data bases*, pages 615–626, 2007.

[29] J. Zhou, M. Xu, A. Shraer, B. Namasivayam, A. Miller, E. Tschannen, S. Atherton, A. J. Beamon, R. Sears, J. Leach, D. Rosenthal, X. Dong, W. Wilson, B. Collins, D. Scherer, A. Grieser, Y. Liu, A. Moore, B. Muppana, X. Su, and V. Yadav. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data, SIGMOD Conference 2021, Xian, Shanxi, China, June 03 — 05, 2021*. ACM, 2021.