

FoundationDB Record Layer: A Multi-Tenant Structured Datastore

Christos Chrysafis, Ben Collins, Scott Dugas, Jay Dunkelberger, Moussa Ehsan, Scott Gray, Alec Grieser, Ori Herrnstadt, Kfir Lev-Ari, Tao Lin, Mike McMahon, Nicholas Schiefer, Alexander Shraer
Apple, Inc.

ABSTRACT

The FoundationDB Record Layer is an open source library that provides a record-oriented data store with semantics similar to a relational database implemented on top of FoundationDB, an ordered, transactional key-value store. The Record Layer provides a lightweight, highly extensible way to store structured data. It offers schema management and a rich set of query and indexing facilities, some of which are not usually found in traditional relational databases, such as nested record types, indexes on commit versions, and indexes that span multiple record types. The Record Layer is stateless and built for massive multi-tenancy, encapsulating and isolating all of a tenant’s state, including indexes, into a separate logical database. We demonstrate how the Record Layer is used by CloudKit, Apple’s cloud backend service, to provide powerful abstractions to applications serving hundreds of millions of users. CloudKit uses the Record Layer to host billions of independent databases, many with a common schema. Features provided by the Record Layer enable CloudKit to provide richer APIs and stronger semantics with reduced maintenance overhead and improved scalability.

ACM Reference Format:

Christos Chrysafis, Ben Collins, Scott Dugas, Jay Dunkelberger, Moussa Ehsan, Scott Gray, Alec Grieser, Ori Herrnstadt, Kfir Lev-Ari, Tao Lin, Mike McMahon, Nicholas Schiefer, and Alexander Shraer. 2019. FoundationDB Record Layer: A Multi-Tenant Structured Datastore. In *2019 International Conference on Management of Data (SIGMOD '19)*, June 30–July 5, 2019, Amsterdam, Netherlands. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3299869.3314039>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '19, June 30–July 5, 2019, Amsterdam, Netherlands
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-5643-5/19/06.

<https://doi.org/10.1145/3299869.3314039>

1 INTRODUCTION

Many applications require a scalable and highly available backend that provides durable storage. Developing and operating such backends presents many challenges. First, the high volume of data collected by modern applications, coupled with the large number of users and high access rates, requires smart partitioning and placement solutions for storing data to achieve *horizontal scalability*. Modern storage systems must scale elastically in response to increases in user demand for both storage capacity and computation. Second, as systems become larger, formerly “rare” events such as network, disk, and machine failures become everyday occurrences. A core challenge of distributed systems is maintaining *system availability and durability* in the face of these problems. Third, many techniques for addressing scalability and availability challenges, such as eventual consistency, create immense challenges for application developers. Implementing *transactions* in a distributed setting remains one of the most challenging problems in distributed systems.

Fourth, as stateful services grow, they must support the needs of many diverse users and applications. This *multi-tenancy* brings many challenges, including isolation, resource sharing, and elasticity in the face of growing load. Many database systems intermingle data from different tenants at both the compute and storage levels. Retrofitting resource isolation to such systems is challenging. Stateful services are especially difficult to scale elastically because state cannot be partitioned arbitrarily. For example, data and indexes cannot be stored on entirely separate storage clusters without sacrificing transactional updates, performance, or both.

These problems must be addressed by any company offering stateful services. Yet despite decades of academic and industrial research, they are notoriously difficult to solve correctly and require a high level of expertise and experience. Big companies use in-house solutions, developed and evolved over many years by large teams of experts. Smaller companies often have little choice but to pay larger cloud providers or sacrifice durability.

FoundationDB [10] democratizes industrial-grade highly-available and consistent storage, making it freely available to anyone as an open source solution [11] and is currently used in production at companies such as Apple, Snowflake, and

Wavefront. While the semantics, performance, and reliability of FoundationDB make it extremely useful, FoundationDB’s data model, an ordered mapping from binary keys to binary values, is often insufficient for applications. Many of them need structured data storage, indexing capabilities, a query language, and more. Without these, application developers are forced to reimplement common functionality, slowing development and introducing bugs.

To address these challenges, we present the FoundationDB Record Layer: an open source record-oriented data store built on top of FoundationDB with semantics similar to a relational database [19, 20]. The Record Layer provides schema management, a rich set of query and indexing facilities, 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. These lightweight abstractions allow for multi-tenancy at an extremely large scale: the Record Layer allows creating isolated logical databases for each tenant—at Apple, it is used to manage billions of such databases—all while providing familiar features such as transactional index maintenance.

The Record Layer represents structured values as Protocol Buffer [4] messages called records that include typed fields and even nested records. Since an application’s schema inevitably changes over time, the Record Layer includes tools for schema management and evolution. It also includes facilities for planning and efficiently executing declarative queries using a variety of index types. The Record Layer leverages advanced features of FoundationDB; for example, many aggregate indexes are maintained using FoundationDB’s atomic mutations, allowing concurrent, conflict-free updates. Beyond its rich feature set, the Record Layer provides a large set of extension points, allowing its clients to extend its functionality even further. For example, client-defined index types can be seamlessly “plugged in” to the index maintainer and query planner. Similarly, record serialization supports client-defined encryption and compression algorithms.

The Record Layer supports multi-tenancy at scale through two key architectural choices. First, the layer is completely stateless, so scaling the compute service is as easy as launching more stateless instances. A stateless design means that load-balancers and routers need only consider where the data are located rather than which compute servers can serve them. Furthermore, a stateless server has fewer resources that need to be apportioned among isolated clients. Second, the layer achieves resource sharing and elasticity with its *record store* abstraction, which encapsulates the state of an entire logical database, including serialized records, indexes, and even operational state. Each record store is assigned a contiguous range of keys, ensuring that data belonging to different tenants is logically isolated. If needed, moving a tenant is as simple as copying the appropriate range of

data to another cluster as everything needed to interpret and operate each record store is found in its key range.

The Record Layer is used by multiple systems at Apple. We demonstrate the power of the Record Layer at scale by describing how CloudKit, Apple’s cloud backend service, uses it to provide strongly-consistent data storage for a large and diverse set of applications [43]. Using the Record Layer’s abstractions, CloudKit offers multi-tenancy at the extreme by maintaining independent record stores for each user of each application. As a result, we use the Record Layer on FoundationDB to host billions of independent databases sharing thousands of schemata. In the future, we envision that the Record Layer will be combined with other storage models, such as queues and graphs, leveraging FoundationDB as a general purpose storage engine to provide transactional consistency across all these data models. In summary, this work makes the following contributions:

- An open source layer on top of FoundationDB with semantics akin to those of a relational database.
- The record store abstraction and a suite of techniques to manipulate it, enabling billions of logical tenants to operate independent databases in a FoundationDB cluster.
- An extensible architecture allowing clients to customize core features including schema management and indexing.
- A lightweight design that provides rich features on top of the underlying key-value store.

2 BACKGROUND ON FOUNDATIONDB

FoundationDB is a distributed, ordered key-value store that runs on clusters of commodity servers and provides ACID transactions over arbitrary sets of keys using optimistic concurrency control. Its architecture draws from the *virtual synchrony* paradigm [23, 26] whereby FoundationDB is composed of two logical clusters: one that stores data and processes transactions and another coordination cluster (running Active Disk Paxos [27]) that maintains membership and configuration for the first cluster. This allows FoundationDB to achieve high availability while requiring only $F + 1$ storage replicas to tolerate F failures [23]. FoundationDB is distinguished by its deterministic simulation testing framework which can quickly simulate entire clusters under a variety of failure conditions in a single thread with complete determinism. In the past year, we have run more than 250 million simulations equivalent to more than 1870 years and 3.5 million CPU-hours. This rigorous testing in simulation makes FoundationDB extremely stable and allows its developers to introduce new features and releases in a rapid cadence, unusual among similar strongly-consistent distributed—or even centralized—databases.

Layers. Unlike most databases, which bundle together a storage engine, data model, and query language, forcing users to choose all three or none, FoundationDB takes a

modular approach: it provides a highly scalable, transactional storage engine with a minimal yet carefully chosen set of features. For example, it provides no structured semantics, and each cluster provides a single, logical key space which it automatically partitions and replicates. *Layers* can be constructed on top to provide various data models and other capabilities. Currently, the Record Layer is the most substantial layer built on FoundationDB.

Transactions and Semantics. FoundationDB provides ACID multi-key transactions with strictly-serializable isolation, implemented using multi-version concurrency control (MVCC) for reads and optimistic concurrency for writes. As a result, 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. Specifically, a client performing a transaction obtains a read version, chosen as the latest database commit version, by performing a `getReadVersion` (GRV) call and 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 cluster nodes and then acknowledged to the client. FoundationDB executes operations within a transaction in parallel, while preserving the program order of accesses to each key and guaranteeing that a read following a write to the same key within a transaction returns the written value. FoundationDB imposes a 5 second transaction time limit, which the Record Layer compensates for using techniques described in Section 4.

Besides create, read, update, and delete (CRUD) operations, FoundationDB provides atomic read-modify-write operations on single keys (e.g., addition, min/max, etc.). Atomic operations occur within a transaction like other operations, but they do not create read conflicts, so a concurrent change to that value would not cause the transaction to abort. For example, a counter incremented concurrently by many clients would be best implemented using atomic addition.

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. For example, a transaction may wish to read a value that is monotonically increasing over time just to determine if the value has reached some threshold. In this case, reading the value at a snapshot isolation level (or clearing the read conflict range for the value's key) would allow the transaction to see the state of the value, but it would not result in conflicts with other transactions that may be modifying the value concurrently.

Keys, values, and order. Keys and values in FoundationDB are opaque binary values. FoundationDB imposes limits on

key and value sizes (10kB for keys and 100kB for values) with much smaller recommended sizes (32B for keys and up to 10KB for values). Transactions are limited to 10MB, including the key and value size of all keys written and the sizes of all keys in the read and write conflict ranges of the commit. In production at Apple, with workloads generated by CloudKit through the Record Layer, the median and 99th percentile transaction sizes are approximately 7KB and 36KB, respectively. Keys are part of a single global key space, and it is up to applications to divide and manage that key space with the help of several convenience APIs such as the tuple and directory layers described next. FoundationDB supports range reads based on the binary ordering of keys. Finally, range clear operations are supported and can be used to clear all keys in a certain range or starting with a certain prefix.

Tuple and directory layers. Key ordering in FoundationDB makes tuples a convenient and simple way to model data. The tuple layer, included within the FoundationDB client library, encodes tuples into keys such that the binary ordering of those keys preserves the ordering of tuples and the natural ordering of typed tuple elements. In particular, a common prefix of the tuple is serialized as a common byte prefix and defines a key *subspace*. For example, a client may store the tuple `(state, city)` and later read using a prefix like `(state, *)`.

The directory layer provides an API for defining a logical directory structure which maps potentially long-but-meaningful binary strings to short binary strings, reducing the amount of space used by keys. For example, if all of an application's keys are prefixed with its name, the prefix might be added to the directory layer. Internally, the directory layer assigns values to its entries using a sliding window allocation algorithm that concurrently allocates unique mappings while keeping the allocated integers small.

3 DESIGN GOALS AND PRINCIPLES

The Record Layer is designed to provide powerful abstractions for operations on structured data including transaction processing, schema management, and query execution. It aims to provide this rich set of features in a massively multi-tenant environment where it must serve the needs of a great number of tenants with diverse storage and retrieval needs. For example, some tenants store and locally synchronize a small amount of data while others store large amounts of data and query it in complex ways.

To illustrate this challenge concretely, Figure 3 shows the distribution of the sizes of record stores used to store CloudKit's private databases [43]. Although the vast majority of databases are quite small, much of the data is stored in relatively large databases. Furthermore, the data in the figure excludes larger databases such as the "public" databases shared

across all users of an application, which can be terabytes in size.

In the face of such a range of database sizes, it is tempting to provision and operate separate systems to store “small data” and “big data.” However, such an approach poses both operational challenges and difficulties to application developers, who must contend with varying semantics across systems. Instead, the Record Layer offers the semantics of a structured data store with massive multi-tenancy in mind by providing features that serve the diverse needs of its tenants and mechanisms to isolate them from each other.

3.1 Design principles

Achieving these goals required a variety of careful design decisions, some of which differ from traditional relational databases. This section highlights several of these principles.

Statelessness. In many distributed databases, individual database servers maintain ephemeral, server-specific state, such as memory and disk buffers, in addition to persistent information such as data and indexes. In contrast, the Record Layer stores all of its state in FoundationDB or returns it to the client so that the layer itself is completely stateless.

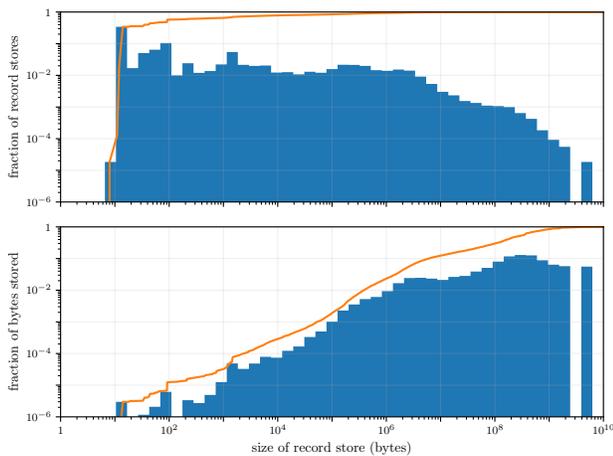


Figure 1: *Top:* The distribution of record store sizes for a 0.1% sample of CloudKit-managed record stores as a normalized histogram (blue) and cumulative density function (orange). Notice that a substantial majority of record stores contain fewer than 1 kilobytes of record data. *Bottom:* The distribution of the total size of record stores of a particular size as a normalized histogram (blue) and cumulative density function (orange). The sample represents private databases and excludes many large record stores such as public databases. All sizes include primary record data and exclude metadata and index entries.

For example, the Record Layer does not maintain any state about the position of a cursor in memory; instead, the context needed to advance a cursor in a future request is serialized and returned to the client as a continuation.

The stateless nature of the layer has three primary benefits, which we illustrate with the same example of streaming data from a cursor. First, it simplifies request routing: a request can be routed to any of the stateless servers even if it requests more results from an existing cursor since there is no buffer on a particular server that needs to be accessed. Second, it substantially simplifies the operation of the system at scale: if the server encounters a problem, it can be safely restarted without needing to transfer cursor state to another server. Lastly, storing state in FoundationDB ensures that all state has the same ACID semantics as the underlying key-value store: we do not need separate facilities for verifying the integrity of our cursor-specific metadata.

Streaming model for queries. The Record Layer controls its resource consumption by limiting its semantics to those that can be implemented on streams of records. For example, it supports ordered queries (as in SQL’s ORDER BY clause) only when there is an available index supporting the requested sort order. This approach enables supporting concurrent workloads without requiring stateful memory pools in the server. This also reflects the layer’s general design philosophy of preferring fast and predictable transaction processing over OLAP-style analytical queries.

Flexible schema. The atomic unit of data in the Record Layer is a *record*: a Protocol Buffer [4] message which is serialized and stored in the underlying key-value space. This provides fast and efficient transaction processing on individual records akin to a row-oriented relational database. Unlike record tuples in the traditional relational model, these messages can be highly structured; in addition to a variety of complex data types, these messages support nesting of record types within a field and repeated instances of the same field. As a result, list and map data structures can be implemented within a single record. Because the Record Layer is designed to support millions of independent databases with a common schema, it stores metadata separately from the underlying data so that the common metadata can be updated atomically for all stores that use it.

Efficiency. Implemented as a library rather than an independent client/server system, the Record Layer can be embedded in its client, imposing few requirements on how the actual server is implemented. Since FoundationDB is most performant at high levels of concurrency, nearly all of the Record Layer’s operations are implemented asynchronously and pipelined where possible. We also make extensive use of FoundationDB-specific features, such as controllable isolation semantics, both within the layer’s implementation and exposed to clients via its API.

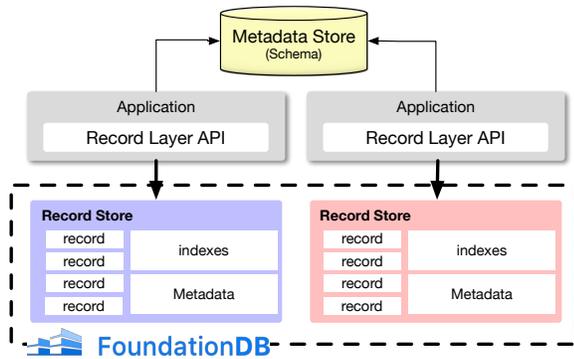


Figure 2: Architecture of the Record Layer. The core record store abstraction stores an entire logical database with a contiguous FoundationDB subspace, providing logical isolation between tenants.

Extensibility. In the spirit of the FoundationDB’s layered architecture, the Record Layer exposes a number of extension points through its API so that clients can extend its functionality. For example, clients can easily define new index types, methods for maintaining those indexes, and rules that extend its query planner to use those indexes in planning. This extensibility makes it easy to add features that are left out of the Record Layer’s core, such as memory pool management and arbitrary sorting. Our implementation of CloudKit on top of the Record Layer (discussed in Section 8) makes substantial use of this extensibility with custom index types, planner behavior, and schema management.

4 RECORD LAYER OVERVIEW

The Record Layer is primarily used as a library by stateless backend servers that need to store structured data in FoundationDB. It is used to store billions of logical databases, called *record stores*, with thousands of schemata. Records in a record store are Protocol Buffer messages and a record type is defined with a Protocol Buffer definition. A record type resembles a table in a traditional relational database in that it defines the structure of multiple records of that type. However, unlike tables in a relational database, all record types within a record store are interleaved within the same extent. The schema, also called *the metadata*, of a record store is a set of record types and index definitions on these types (see Section 6). Metadata is versioned and may be stored in FoundationDB or elsewhere (metadata management and evolution is discussed in Section 5). The record store is responsible for storing raw records, indexes defined on the record fields, and the highest version of the metadata it was accessed with. This general architecture is shown in Figure 2.

Providing isolation between record stores is key for multi-tenancy. To facilitate resource isolation, the Record Layer tracks and enforces limits on resource consumption for each

transaction, provides continuations to resume work, and can be coupled with external throttling. On the data level, the keys of each record store start with a unique binary prefix which defines a FoundationDB subspace. All the record store’s data is logically co-located within the subspace and the subspaces of different record stores do not overlap.

Primary keys and indexes are defined within the Record Layer using *key expressions*, covered in detail in Appendix A. A key expression defines a logical path through a record; applying it to a record extracts record field values and produces a tuple that becomes the primary key for the record or key of the index for which the expression is defined. Key expressions may produce multiple tuples, allowing indexes to “fan out” and generate index entries for individual elements of nested and repeated fields. Since all record types are interleaved, both queries and index definitions may span all types of records in a record store.

To avoid exposing FoundationDB’s limits on key and value sizes to clients, the Record Layer splits large records across a set of contiguous keys and splices them back together when deserializing split records. A special type of split, immediately preceding each record, holds the commit version of the record’s last modification; it is returned with the record on every read. The Record Layer supports pluggable serialization libraries, including optional compression and encryption of stored records.

The Record Layer provides APIs for storing, reading and deleting records, creating and deleting record stores and indexes in stores, scanning and querying records using the secondary indexes, updating record store metadata, managing a client application’s directory structure, and iteratively rebuilding indexes when they cannot be rebuilt as part of a single transaction.

All Record Layer operations that provide a cursor over a stream of data, such as record scans, index scans, and queries, support continuations. A continuation is an opaque binary value that represents the starting position of the next available value in a cursor stream. Results are parceled to clients along with the continuation, allowing them resume the operation by supplying the returned continuation when invoking the operation again. This gives clients a way to control the iteration without requiring the server to maintain state, and it allows scan or query operations that exceed the transaction time limit to be split across multiple transactions.

The Record Layer leverages details of FoundationDB’s implementation to improve data access speed. For example, record prefetching asynchronously preloads records into the FoundationDB client’s read-your-write cache, but does not return them to the client application. When reading batches of many records, this can potentially save a context switch and record deserialization. The Record Layer also includes mechanisms to trade off consistency for performance, such as

snapshot reads. Similarly, the layer exposes FoundationDB’s “causal-read-risky” flag, which causes `getReadVersion` to be faster at the risk of returning a slightly stale read version during the rare case of a cluster reconfiguration. This is usually an acceptable risk; for example, ZooKeeper’s “sync” operation behaves similarly [38]. Furthermore, transactions that modify state never return stale data since their reads are validated at commit time. Read version caching optimizes `getReadVersion` further by completely avoiding communication with FoundationDB if a read version was “recently” fetched from FoundationDB. Often, the client application provides an acceptable staleness and the last seen commit version as transaction parameters; the Record Layer uses a cached version as long as it is sufficiently recent and no smaller than the version previously observed by the client. This may result in reading stale data and may increase the rate of failed transactions in transactions that modify state. Version caching is most useful for read-only transactions that do not need to return the latest data and for low-concurrency workloads where the abort rate is low.

To help clients organize their record stores in FoundationDB’s key space, the Record Layer provides a `KeySpace` API which exposes the key space in a fashion similar to a filesystem directory structure. When writing data to FoundationDB, or when defining the location of a record store, a path through this logical directory tree may be traced and compiled into a tuple value that becomes a row key. The `KeySpace` API ensures that all directories within the tree are logically isolated and non-overlapping. Where appropriate, it uses the directory layer (described in Section 2) to automatically convert directory names to small integers.

5 METADATA MANAGEMENT

The Record Layer provides facilities for managing changes to a record store’s metadata. Since one of its goals is to support many databases that share a common schema, the Record Layer allows metadata to be stored in a separate key space from the data or even a separate storage system entirely. In most deployments, this metadata is aggressively cached by clients so that records can be interpreted without additional reads from the key-value store. This architecture allows low-overhead, per request, connections to a particular database.

Schema evolution. Since records are serialized into the underlying key-value store as Protocol Buffer messages (possibly after pre-processing steps, such as compression and encryption), some basic data evolution properties are inherited from Protocol Buffers: new fields can be added to a record type and appear as uninitialized in old records, and new record types can be added without interfering with old records. As a best practice, field numbers are never reused and should be deprecated rather than removed altogether.

The metadata is versioned in single-stream, non-branching, monotonically increasing fashion. Every record store tracks the highest version it has been accessed with in a small header within a single key-value pair. When a record store is opened, this header is read and the version compared with current metadata version.

Typically, the metadata will not have changed since the store was last opened, so these versions are the same. When the version in the database is newer, a client has usually used an out-of-date cache to obtain the current metadata. If the version in the database is older, changes need to be applied. New records types and fields can be added by updating the Protocol Buffer definition.

Adding indexes. An index on a new record type can be enabled immediately since there are no existing records of that type. Adding an index to an existing record type, which might already have records in the record store, is more expensive since it might require reindexing. Since records of different types may exist in the same key space, all the records need to be scanned when building a new index. If there are very few or no records, the index can be built right away within a single transaction. If there are many existing records, the index cannot be built immediately because that might exceed the 5 second transaction time limit. Instead, the index is disabled and the reindexing proceeds as a background job as described in Section 6.

Metadata versioning. Occasionally, changes need to be made to the way that the Record Layer itself encodes data. For this, the same database header that records the application metadata version also records a storage format version, which is updated at the same time. Updating may entail re-formatting small amounts of data or, in some cases, enabling a compatibility mode for old formats. We also maintain an “application version” for use by the client that can be used to track data evolution that is not captured by the metadata alone. For example, a nested record type might be promoted to a top-level record type as part of data renormalization. The application version allows checking for these changes as part of opening the record store instead of implementing checks in the application code. If a series of such changes occur, the version can also be used as a counter tracking how far along we are in applying the changes to the record store.

6 INDEX DEFINITION AND MAINTENANCE

Record Layer indexes are durable data structures that support efficient access to data, or possibly some function of the data, and can be maintained in a streaming fashion, i.e., updated incrementally when a record is inserted, updated, or deleted using only the contents of that record. Index maintenance occurs in the same transaction as the record change itself, ensuring that indexes are always consistent with the data. Our

ability to do this efficiently relies heavily on FoundationDB’s fast multi-key transactions. Efficient index scans use FoundationDB’s range reads and rely on the lexicographic ordering of stored keys. Each index is stored in a dedicated subspace within the record store so that indexes can be removed cheaply using FoundationDB’s range clear operation. Indexes may be configured with one or more *index filters*, which allow records to be conditionally excluded from index maintenance, effectively creating a “sparse” index and potentially reducing storage space and maintenance costs.

Index maintenance. Defining an index type requires implementing an *index maintainer* tasked with updating the index when records change. The Record Layer provides built-in index maintainers for a variety of index types (Section 7). The index maintainer abstraction is directly exposed to clients, allowing them to define custom index types.

When a record is saved, we first check if a record already exists with the same primary key. If so, registered index maintainers remove or update associated index entries for the old record, and the old record is deleted from the record store. A range clear to delete the old record is necessary as records can be split across multiple keys. Next, we insert the new record into the record store. Finally, registered index maintainers insert or update any associated index entries for the new record. We use a variety of optimizations during index maintenance; for example, if an existing record and a new record are of the same type and some of the indexed fields are the same, the unchanged indexes are not updated.

Online index building. The Record Layer includes an online index builder used to build or rebuild indexes in the background. To ensure that an index is not used before it is fully built, indexes begin in a *write-only* state where writes maintain the index but it cannot be used to satisfy queries. The index builder then scans the record store and instructs the index maintainer for that index to update the index for each encountered record. When the index build completes, the index is marked as *readable*, the normal state where the index is maintained by writes and usable by queries. Index building is split into multiple transactions to reduce conflicts with concurrent mutations and avoid transaction size limits.

Indexes are defined by an index type and a *key expression*, which defines a function from a record to one or more tuples consumed by the index maintainer and used to form the index key. The Record Layer includes a variety of key expressions (see Appendix A) and allows clients to define custom ones.

7 INDEX TYPES

The type of an index determines which predicates it can be used to evaluate. The Record Layer supports a variety of index types, many of which make use of specific FoundationDB features. Clients can define their own index types by implementing and registering custom key expressions and index

maintainers (see Section 6). In this Section, we outline the VALUE, Atomic Mutation and VERSION indexes. Appendix B describes the RANK and TEXT index types, used for dynamic order statistics and full-text indexing respectively.

Unlike in traditional relational systems, indexes can span multiple record types, in which case any fields referenced by the key expression must exist in all of the index’s record types. Such indexes allow for efficient searches across different record types with a common set of search criteria.

VALUE Indexes. The default VALUE index type provides a standard mapping from index entry (a single field or a combination of field values) to record primary key. Scanning the index can be used to satisfy many common predicates, e.g., to find all primary keys where an indexed field’s value is less than or equal to a given value.

Atomic mutation indexes. Atomic mutation indexes are implemented using FoundationDB’s atomic mutations, described in Section 2. These indexes are generally used to support aggregate statistics and queries. For example, the SUM index type stores the sum of a field’s value over all records in a record store, where the field is defined in the index’s key expression. In this case, the index contains a single entry, mapping the index subspace path to the sum value. The key expression could also include one or more grouping fields, in which case the index contains a sum for each value of the grouping field. While the maintenance of such an index could be implemented by reading the current index value, updating it with a new value, and writing it back to the index, such an implementation would not scale, as any two concurrent record updates would necessarily conflict. Instead, the index is updated using FoundationDB’s atomic mutations (e.g., the ADD mutation for the SUM index), which do not conflict with other mutations.

The Record Layer currently supports the following atomic mutation index types, tracking different aggregate metrics:

- COUNT - number of records
- COUNT UPDATES - num. times a field has been updated
- COUNT NON NULL - num. records where a field isn’t null
- SUM - summation of a field’s value across all records
- MAX (MIN) EVER - max (min) value ever assigned to a field, over all records, since the index has been created

Note that these indexes have a relatively low foot-print compared to VALUE indexes as they only write a single key for each grouping key, or, in its absence, a single key for each record store. However, a small number of index keys that need to be updated on each write can lead to high write traffic on those keys, causing high CPU and I/O usage for the FoundationDB storage servers that hold them. This can also result in increased read latency for clients attempting to read from these servers.

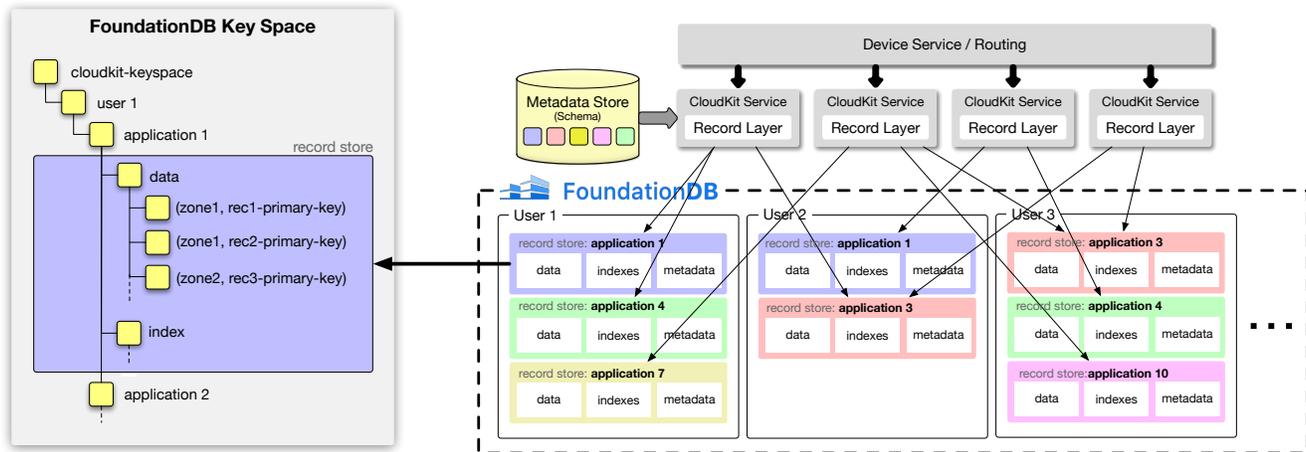


Figure 3: CloudKit architecture using the Record Layer.

VERSION. VERSION indexes are similar to VALUE indexes in that each has an index entry that maps to a primary key. However, a VERSION index allows the index’s key expression to include a special “version” field: a 12 byte value representing the commit version of the last update to the indexed record. The version is guaranteed to be unique and monotonically increasing with time within a single FoundationDB cluster. The first 10 bytes are assigned by the FoundationDB servers upon commit, and only the last 2 bytes are assigned by the Record Layer, using a counter maintained by the Record Layer per transaction. Since versions are assigned in this way for each record insert and update, each record stored in the cluster has a unique version.

Since the version is only known upon commit, it is not included within the record’s Protocol Buffer representation. Instead, the Record Layer writes a mapping from the primary key of each record to its associated version in the key space adjacent to the records so that both can be retrieved efficiently with a single range-read.

Version indexes expose the total ordering of operations within a FoundationDB cluster. For example, a client can scan a prefix of a version index and be sure that it can continue scanning from the same point and observe all the newly written data. The following section describes how CloudKit uses this index type to implement change-tracking (sync).

8 USE CASE: CLOUDKIT

CloudKit [43] is Apple’s cloud backend service and application development framework, providing much of the backbone for storage, management, and synchronization of data across devices as well as sharing and collaboration between users. We describe how CloudKit uses FoundationDB and the Record Layer, allowing it to support applications requiring more advanced features such as the transactional indexing and query capabilities described in this paper.

Within CloudKit a given application is represented by a logical *container*, defined by a schema that specifies the record types, typed fields, and indexes that are needed to facilitate efficient record access and queries. The application clients store records within named zones. Zones organize records into logical groups which can be selectively synced across client devices.

CloudKit assigns a unique FoundationDB subspace for each user, and defines a record store within that subspace for each application accessed by the user. This means that CloudKit is effectively maintaining (# users) × (# applications) logical databases, each with their own records, indexes, and other metadata. CloudKit maintains billions of such databases. When requests are received from client devices, they are routed and load balanced across a pool of available CloudKit Service processes which access the appropriate Record Layer record store and service the request.

CloudKit translates the application schema into a Record Layer metadata definition and stores it in a metadata store (depicted in Figure 3). The metadata also includes attributes added by CloudKit such as system fields tracking record creation and modification time and the zone in which the record was written. The zone name is added as a prefix to primary keys, allowing efficient per-zone access to records. In addition to user-defined indexes, CloudKit maintains a number of “system” indexes, such as an index tracking the total record size by record type that is used for quota management.

8.1 New CloudKit Capabilities

CloudKit was initially implemented using Cassandra [6] as the underlying storage engine. To support atomic multi-record operation batches within a zone, CloudKit uses Cassandra’s light-weight transactions [3]: all updates to the zone are serialized using Cassandra’s compare-and-set (CAS) operations on a dedicated per-zone update-counter. This implementation suffices for many applications using CloudKit,

	Cassandra	Record Layer
Transactions	Within Zone	Within Cluster
Concurrency	Zone level	Record level
Zone size limit	Cassandra partition size (GBs)	FoundationDB cluster size
Index Consistency	Eventual	Transactional
Indexes Stored	in Solr	in FoundationDB

Table 1: CloudKit on Cassandra and the Record Layer.

but it has two scalability limitations. First, there is no concurrency within a zone, even for operations making changes to different records. Second, multi-record atomic operations are scoped to a single Cassandra partition, which is limited in size; furthermore, Cassandra’s performance deteriorates as the size of a partition grows. The former is a concern for collaborative applications, where data is shared among many users or client devices. These limitations require application designers to carefully model their data and workload such that records updated together reside in the same zone while making sure that zones do not grow too large and that the rate of concurrent updates is minimized.

The implementation of CloudKit on FoundationDB and the Record Layer addresses both issues, as summarized in Table 1. Transactions are scoped to the entire database, allowing CloudKit zones to grow significantly larger than before and supporting concurrent updates to different records within a zone. Leveraging these new transactional capabilities, CloudKit now exposes interactive transactions to its clients, specifically to other backend services that access CloudKit through gRPC [2]. This simplifies the implementation of client applications and has enabled many new clients to use CloudKit.

Previously, only very few “system” indexes were maintained transactionally by CloudKit in Cassandra, whereas all user-defined secondary indexes were maintained in Solr. Due to high access latencies, these indexes are updated asynchronously, and queries that use them obtain an eventually consistent view of the data which requires application designers to work around perceived inconsistencies. With the Record Layer, user-defined secondary indexes are maintained transactionally with updates, so all queries return the latest data.

Personalized full-text search. Users expect instant access to data they create such as emails, text messages, and notes. Often, indexed text and other data are interleaved, so transactional semantics are important. We implemented a personalized text indexing system using the TEXT index primitives described in Appendix B that now serves millions of users. Unlike traditional search indexing systems, all updates are performed transactionally, and no background jobs are needed to perform index updates and deletes. In addition to providing a consistent view of the data, this approach also

reduces operational costs by storing all data in one system. Our system uses FoundationDB’s key order to support prefix matching with no additional overhead and n -gram searches that require only n key entries instead of the usual $O(n^2)$ keys needed to index all possible sub-strings. The system also supports proximity and phrase search.

High-concurrency zones. With Cassandra, CloudKit maintains a secondary “sync” index from the values of the per-zone update-counter to changed records [43]. Scanning this index allows CloudKit perform a *sync* operation that brings a mobile device up-to-date with the latest changes to a zone. The implementation of CloudKit using the Record Layer relies on FoundationDB’s concurrency control and no longer maintains an update-counter that creates conflicts between otherwise non-conflicting transactions. To implement a sync index, CloudKit leverages the total order on FoundationDB’s commit versions by using a VERSION index, mapping versions to record identifiers. To perform a sync, CloudKit simply scans the VERSION index.

However, versions assigned by different FoundationDB clusters are uncorrelated. This introduces a challenge when migrating data from one cluster to another, e.g., when moving users between clusters to improve load balancing and locality. The sync index must represent the order of updates across all clusters, so updates committed after the move must be sorted after updates committed before the move. CloudKit addresses this with an application-level per-user count of the number of moves, called the *incarnation*. Initially, the incarnation is 1, and it is incremented each time the user’s data is moved to a different cluster. On every record update, we write the user’s current incarnation to the record’s header; these values are not modified during a move. The VERSION sync index maps (incarnation, version) pairs to changed records, sorting the changes first by incarnation, then version.

When deploying this implementation, we needed to handle existing data with an associated update-counter value but no version. Instead of using business logic to combine the old and new sync indexes, we used the function key expression (see Section A) to make this migration operationally straightforward, transparent to the application, and free of legacy code. Specifically, the VERSION index maps a function of the incarnation, version, and update counter value to a changed record, where the function is (incarnation, version) if the record was last updated with the new method and (0, update counter value) otherwise. This maintains the order of records written using update counters and sorts all such records before records written with the new method.

8.2 Client Resource Isolation

CloudKit services a very large number of concurrent requests, and it is vital that no one client or application overwhelms the system or impacts the performance of others. A number of

Record Layer features were explicitly designed to ensure that individual requests consume a bounded amount of resources.

First, the Record Layer is designed to minimize the overhead of operations on top of the underlying key-value store. To illustrate this, we determined the median number of FoundationDB keys that were read or written while executing various common CloudKit operations. A query operation, which returns all records that match a given query, reads an average of ~ 38.3 keys, of which ~ 6.2 are not the records or index entries themselves, for a total overhead of $\sim 15\%$. Simple requests for individual records are comparatively expensive, with an average of ~ 13.3 key read of which ~ 7.7 are not for record data. This reflects the general focus of CloudKit on providing higher level services, such as queries and synchronization over individual CRUD requests.

For record save operations, it is more complicated to properly estimate the overhead. For one, write overhead is dominated by the overhead of maintaining indexes, which depends on the number of indexes defined on the record's type. Furthermore, FoundationDB's commit time for write transactions does not have a simple relationship with the number of writes. The FoundationDB client buffers writes locally until commit time, when it ships all writes to the server along with the set of conflict ranges for the transaction. In practice, the write performance depends substantially on the number of conflicts produced rather than simply the number of writes. On average, a CloudKit transaction writes ~ 8.5 records and makes ~ 34.5 key writes associated with indexes, so the total index overhead is approximately ~ 4 writes per record.

Today, the Record Layer does not provide the ability to perform in-memory query operations, such as hash joins, grouping, aggregation, or sorts. Operations such as sorting and joining must be assisted by appropriate index definitions. For example, efficient joins between records can be facilitated by defining an index across multiple record types on common field names. While this does impose some additional burden on the application developer, it ensures that the memory needed to complete a given request is limited to little more than the records accessed by the query. However, this approach may require a potentially unbounded amount of I/O to implement a given query. For this, we leverage the Record Layer's ability to enforce limits on the total number of records or bytes read while servicing a request. When one of these limits is reached, the complete state of the operation is serialized and returned to the client as a continuation. The client may re-submit the operation with the continuation to resume the operation. Because these operations are small, rate-based throttling mechanisms work more effectively. With these limits, continuations, and throttling, we ensure that all clients make some progress even when the system comes under stress.

9 RELATED WORK

Traditional relational databases offer many features including structured storage, schema management, ACID transactions, user-defined indexes, and SQL queries that make use of these indexes with the help of a query planner and execution engine. These systems typically scale for read workloads but were not designed to efficiently handle transactional workloads on distributed data [37]. For example, in shared-nothing database architectures cross-shard transactions and indexes are prohibitively expensive and careful data partitioning, a difficult task for a complex application, is required. This led to research on automatic data partitioning, e.g., [31, 40]. Shared-disk architectures are much more difficult to scale, primarily due to expensive cache coherence and database page contention protocols [33].

With the advent of Big Data, as well as to minimize costs, NoSQL datastores [13, 15, 25, 28, 32, 44] offer the other end of the spectrum—excellent scalability but minimal semantics—typically providing a key-value API with no schema, indexing, transactions, or queries. As a result, applications needed to re-implement many of the features provided by a relational database. To fill the void, middle-ground “NewSQL datastores” appeared offering scalability as well as richer feature-sets and semantics [7, 8, 10, 12, 18, 29]. For example, Google's Spanner [29] supports ACID transactions, application-defined schema, and a SQL-based query language; secondary indexes and distributed queries were also added, more recently [21]. FoundationDB [10] takes a unique approach in the NewSQL space: it is highly scalable and provides ACID transactions, but it offers a simple key-value API with no built-in data model, indexing, or queries. This choice allowed FoundationDB to build a powerful, stable and performant storage engine without attempting to implement a one-size-fits-all solution. It was designed to be the foundation while layers built on top (implemented as client-side libraries) such as the Record Layer, provide higher-level abstractions.

Multiple systems implement transactions on top of underlying NoSQL stores [9, 16, 22, 24, 34, 35, 41, 46]. The Record Layer makes use of transactions exposed by FoundationDB to implement structured storage, complete with secondary indexes, queries, and more. Like Google Percolator [41] the Record Layer is completely stateless, keeping all its metadata in the underlying FoundationDB data store.

The Record Layer has unique first-class support for multi-tenancy while most systems face the extreme challenge of retrofitting it. Salesforce's architecture [17] is similarly motivated by the need to support multi-tenancy within the database. For example, all data and metadata are sharded by application, and query optimization considers statistics collected per application and user. The Record Layer takes multi-tenancy support further through built-in resource tracking

and isolation, a completely stateless design, and the record store abstraction. For example, CloudKit faces a dual multi-tenancy challenge as it services many applications, each with a very large user-base. Each record store encapsulates all of a user’s data for one application, including indexes and metadata. This choice makes it easy to scale the system to billions of users by simply adding more clusters and moving record stores to balance the load and improve locality.

While many storage systems include support for full-text search, most provide this support using a separate system [14, 17, 43], such as Solr [1], with eventual-consistency guarantees. In our experience with CloudKit, maintaining a separate system for search is challenging; it has to be separately provisioned, maintained, and made highly-available in concert with the database (e.g., with regards to fail-over decisions). MongoDB includes built-in support for full-text search, but indexes are not guaranteed to be consistent and so queries might not return all matching documents [5].

There is a broad literature on query optimization, starting with the seminal work of Selinger et al. [42] on System R. Since then, much of the focus has been on efficient search-space exploration. Most notably, Cascades [36] introduced a clean rule-based architecture for structuring query planners and proposed operators and transformation rules that are encapsulated as self-contained components. Cascades allows logically equivalent expressions to be grouped in the so-called Memo structure to eliminate redundant work. Recently, Greenplum’s Orca query optimizer [45] was developed as a modern incarnation of Cascades’ principles. We are currently in the process of developing an optimizer that uses the proven principles of Cascades, paving the way for the development of a full cost-based optimizer (Appendix C).

10 LESSONS LEARNED

The Record Layer’s success at Apple validates the usefulness of FoundationDB’s “layer” concept, where the core distributed storage system provides a semantically simple data-store upon which complex abstractions are built. This allows systems architects to choose the parts of the database that they need without working around abstractions that they do not. Building layers, however, remains a complex engineering challenge. To our knowledge, the Record Layer is deployed at a larger scale than any other FoundationDB layer. We summarize some lessons learned building and operating the Record Layer in the hope that they can be useful for both Record Layer adopters and developers of new layers.

10.1 Building FoundationDB layers

Asynchronous processing to hide latency. FoundationDB is optimized for throughput and not individual operation latencies, meaning that effective use requires keeping as much

work outstanding as possible. Therefore, the Record Layer does much of its work asynchronously, pipelining it where possible. However, the FoundationDB client is single-threaded, with only a single network thread that talks to the cluster. Earlier versions of the Java bindings completed futures in the network thread, and the Record Layer used these for its asynchronous work, creating a bottleneck in that thread. By minimizing the amount of work done in the network thread, we were able to get substantially better performance and minimize apparent latency on complex operations by interacting with the key-value store in parallel.

Conflict ranges. In FoundationDB, a transaction conflict occurs when some keys read by one transaction are concurrently modified by another. The FoundationDB API gives full control over these potentially overlapping read- and write-conflict sets. One pattern is then to do a non-conflicting (snapshot) read of a range that potentially contains distinguished keys and adding individual conflicts for only these and not the unrelated keys found in the same range. Thus, the transaction depends only on what would invalidate its results. In the Record Layer, this technique is used for navigating the skip list used for the rank index described in Appendix B. Bugs due to incorrect manual conflict ranges are very hard to find, especially when mixed with business logic. For that reason, layers should generally define abstractions, such as indexes, for such patterns rather than relying on individual client applications to relax isolation requirements.

10.2 Using the Record Layer in practice

Metadata change safety. The Protocol Buffer compiler generates methods for manipulating, parsing and writing messages, as well as static descriptor objects containing information about the message type, such as its fields and their types. These descriptors could potentially be used to build Record Layer metadata *in code*. We do not recommend this approach over explicitly persisting the metadata in a metadata store, except for simple tests. One reason is that it is hard to atomically update the metadata code used by multiple Record Layer instances. For example, if one Record Layer instance runs a newer version of the code (with a newer descriptor), writes records to a record store, then an instance running the old version of the code attempts to read it, an authoritative metadata store (or communication between instances) is needed to interpret the data. This method also makes it harder to check that the schema evolution constraints (Section 5) are preserved. We currently use descriptor objects to generate new metadata to be stored in the metadata store.

Relational similarities. The Record Layer resembles a relational database but has slightly different semantics, which can surprise clients. For example, there is a single extent for all record types because CloudKit has untyped foreign-key

references without a “table” association. By default, selecting all records of a particular type requires a full scan that skips over records of other types or maintaining secondary indexes. For clients with a SQL-like table model, we now support emulating separate extents for each record type by adding a type-specific prefix to the primary key.

10.3 Designing for multi-tenancy

Multi-tenancy is remarkably difficult to add to an existing system. Hence, the Record Layer was built from the ground up to support massively multi-tenant use cases. We have gained substantial advantages from a natively multi-tenant design, including easier shard rebalancing between clusters and the ability to scale elastically. Our experience has led us to conclude that multi-tenancy is more pervasive than one would initially think. Put another way, many applications that do not explicitly host many different applications—as CloudKit does—can reap the benefits of a multi-tenant architecture by partitioning data according to logical “tenants” such as users, application functions, or some other entity.

11 FUTURE DIRECTIONS

The Record Layer’s current feature set grew out of CloudKit’s need to support billions of small databases, each with few users, and all with careful resource constraints. As databases grow in terms of data volume and number of concurrent users, the Record Layer may need to adapt and higher layers will be developed to support these new workloads and more complex query capabilities. However, we aim to ensure that the layer always retains its support for lightweight and efficient deployments. We highlight several future directions:

Avoiding hotspots. As the number of clients simultaneously accessing a given record store increases, checking the store header to confirm that the metadata has not changed may create a hot key. A general way to address hotspots is to replicate data at different points of the keyspace, making it likely that the copies are located on different storage nodes. For the particular case of metadata, where changes are relatively infrequent, we could also alleviate hotspots with caching. However, when the metadata changes, such caches need to be invalidated, or, alternatively, an out-of-date cache needs to be detected or tolerated.

Query operations. Some query operations are possible with less-than-perfect indexing but within the layer’s streaming model, such as a priority queue-based sort-with-small-limit or a limited-size hash join. For certain workloads, it may be necessary to fully support intensive in-memory operations with spill-over to persistent storage. Such functionality can be challenging at scale as it requires new forms of resource tracking and management and must be stateful for the duration of the query.

Materialized views. Normal indexes are a projection of record fields in a different order. COUNT, SUM, MIN, and MAX indexes maintain aggregates compactly and efficiently, avoiding conflicts by using atomic mutations. Adding materialized views, which can synthesize data from multiple records at once, is a natural evolution that would benefit join queries, among others. Adding support for materialized views to the key expressions API might also help the query planner reason about whether an index can be used to satisfy a query.

Higher layers. The Record Layer is close enough to a relational database that it could support a subset or variant of SQL, particularly once the query planner supports joins. A higher level “SQL layer” could be implemented as a separate layer on top of the Record Layer without needing to work around choices made by lower-level layers. Similarly, a higher layer could support OLAP-style analytics workloads.

12 CONCLUSIONS

The FoundationDB Record Layer offers rich features similar to those of a relational database, including structured schema, indexing, and declarative queries. Because it is built on FoundationDB, it inherits its ACID transactions, reliability, and performance. The core *record store* abstraction encapsulates a database and makes it easy to operate the Record Layer in a massively multi-tenant environment. The Record Layer offers deep extensibility, allowing clients to add features such as custom index types. At Apple, we leverage these capabilities to implement CloudKit, which uses the Record Layer to offer new features (e.g., transactional full-text indexing), speed up key operations (e.g., with high-concurrency zones), and simplify application development (e.g., with interactive transactions) across billions of databases.

In building and operating the Record Layer at scale, we have made three key observations with broader applicability. First, the Record Layer’s success validates FoundationDB’s layered architecture in a large-scale system. Second, the Record Layer’s extensible design provides common functionality while easily accommodating the customization needs of complex clients. It is easy to envision other layers that extend the Record Layer to provide rich, high-level functionality, as CloudKit does. Lastly, we find that organizing applications into logical “tenants”, which might be users, features, or some other entity, is a powerful and practically useful way to structure a system and scale it to meet demand.

ACKNOWLEDGMENTS

We thank Lyublena Antova and the anonymous reviewers for their insights and suggestions that helped us improve this paper. We are grateful to Mike Abbott, Bryan Davis, Eric Krugler, and Amol Pattekar for their guidance and support. Finally, we thank Ben Collins, Evan Tschannen and the FoundationDB team at Apple for their expertise.

REFERENCES

- [1] Apache Solr. <http://lucene.apache.org/solr/>.
- [2] gRPC: A high performance, open-source universal RPC framework. <https://grpc.io/>.
- [3] Lightweight transactions in Cassandra 2.0. <https://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0>.
- [4] Protocol Buffers. <https://developers.google.com/protocol-buffers/>.
- [5] MongoDB queries don't always return all matching documents! <https://blog.meteor.com/mongodb-queries-dont-always-return-all-matching-documents-654b6594a827>, 2016.
- [6] Apache Cassandra, 2018.
- [7] Azure Cosmos DB. <https://azure.microsoft.com/en-us/services/cosmos-db/>, 2018.
- [8] Cockroach Labs. <https://www.cockroachlabs.com/>, 2018.
- [9] CockroachDB. <https://www.cockroachlabs.com/>, 2018.
- [10] FoundationDB. <https://www.foundationdb.org>, 2018.
- [11] FoundationDB source. <https://github.com/apple/foundationdb>, 2018.
- [12] MemSQL. <https://www.memsql.com/>, 2018.
- [13] MongoDB. <https://www.mongodb.com>, 2018.
- [14] Riak: Complex Query Support. <http://basho.com/products/riak-kv/complex-query-support>, 2018.
- [15] Riak KV. <http://basho.com/products/riak-kv>, 2018.
- [16] Tephra: Transactions for Apache HBase. <https://tephra.io>, 2018.
- [17] The Force.com Multitenant Architecture. http://www.developerforce.com/media/ForcedotcomBookLibrary/Force.com_Multitenancy_WP_101508.pdf, 2018.
- [18] VoltDB. <https://www.voltdb.com/>, 2018.
- [19] Announcing the FoundationDB Record Layer. <https://www.foundationdb.org/blog/announcing-record-layer/>, 2019.
- [20] FoundationDB Record Layer on GitHub. <https://github.com/foundationdb/fdb-record-layer>, 2019.
- [21] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, A. Lloyd, S. Melnik, R. Rao, D. Shue, C. Taylor, M. van der Holst, and D. Woodford. Spanner: Becoming a sql system. In *Proc. SIGMOD 2017*, 2017.
- [22] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.
- [23] K. Birman, D. Malkhi, and R. van Renesse. Virtually Synchronous Methodology for Dynamic Service Replication. Technical report, Microsoft Research, 2010.
- [24] E. Bortnikov, E. Hillel, I. Keidar, I. Kelly, M. Morel, S. Paranjpye, F. Perez-Sorrosal, and O. Shacham. Omid, Reloaded: Scalable and Highly-Available Transaction Processing. In *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, pages 167–180, 2017.
- [25] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218, 2006.
- [26] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.
- [27] G. V. Chockler and D. Malkhi. Active Disk Paxos with infinitely many processes. *Distributed Computing*, 18(1):73–84, 2005.
- [28] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.
- [29] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaure, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, Aug. 2013.
- [30] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*, chapter 14.1. The MIT Press, 2009.
- [31] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *PVLDB*, 3(1):48–57, 2010.
- [32] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220, 2007.
- [33] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Commun. ACM*, 35(6), June 1992.
- [34] R. Escriva, B. Wong, and E. G. Sirer. Warp: Lightweight Multi-Key Transactions for Key-Value Stores. *CoRR*, abs/1509.07815, 2015.
- [35] D. G. Ferro, F. Junqueira, I. Kelly, B. Reed, and M. Yabandeh. Omid: Lock-free transactional support for distributed data stores. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 676–687, 2014.
- [36] G. Graefe. The Cascades Framework for Query Optimization. *Data Engineering Bulletin*, 18, 1995.
- [37] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996.
- [38] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, 2010.
- [39] H. Melville. *Moby Dick; or The Whale*. <http://www.gutenberg.org/files/2701>.
- [40] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *Proc. SIGMOD 2012*.
- [41] D. Peng and F. Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *In the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.
- [42] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, SIGMOD '79, 1979*.
- [43] 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.
- [44] S. Sivasubramanian. Amazon dynamoDB: a seamlessly scalable non-relational database service. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 729–730, 2012.
- [45] M. A. Soliman, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, F. Rahman, M. Petropoulos, F. Waas, S. Narayanan, K. Krikellas, and R. Baldwin. Orca: A Modular Query Optimizer Architecture for Big Data. In *Proc. SIGMOD 2014*.
- [46] Z. Wei, G. Pierre, and C.-H. Chi. Cloudtps: Scalable transactions for web applications in the cloud. *IEEE Transactions on Services Computing*, 5(4):525–539, 2012.

A KEY EXPRESSIONS

Indexes are defined by an index type and a key expression which defines a function from a record to one or more tuples consumed by the index maintainer and used to form the index key. The Record Layer includes a variety of key expressions and also allows clients to define custom ones.

The simplest key expression is `field`. When evaluated against the sample record in Figure 4, `field("id")` yields the tuple (1066). Unlike standard SQL columns, Protocol Buffer fields permit repeated fields and nested messages. Nested messages can be accessed through the `nest` key expression. For example, `field("parent").nest("a")` yields (1415). To support repeated fields, field expressions define an optional `FanType` parameter. For a repeated field, `FanType` of `Concatenate` produces a tuple with one entry containing a list of all values within the field, while `Fanout` produces a separate tuple for each value. For example, `field("elem", Concatenate)` yields (["first", "second", "third"]), and `field("elem", Fanout)` yields three tuples: ("first"), ("second"), and ("third").

To create compound indexes, multiple key expressions can be concatenated. For example, `concat(field("id"), field("parent").nest("b"))` evaluates to the single tuple (1066, "child"). If a sub-expression produce multiple values, the compound expression will produce the tuples in the Cartesian product of the sub-expressions' values.

The Record Layer also includes a variety of specialized key expressions. For example, the record type key expression produces a value that is guaranteed to be unique for each record type. When used in a primary key, this key expression allows users to treat record types like tables in a traditional relational database. It can also be used to satisfy queries about record types, such as a query for the number of records of each type that are stored within the database. Another special key expression, `version`, is described in Section 7.

In addition to allowing client-defined key expressions, the Record Layer has function key expressions which allow for the execution of arbitrary, user-defined functions against records and their constituent fields. Function key expressions are very powerful and allow users to define custom sort orders for records, for example. The layer also includes `groupBy` key expressions which can be used to divide an

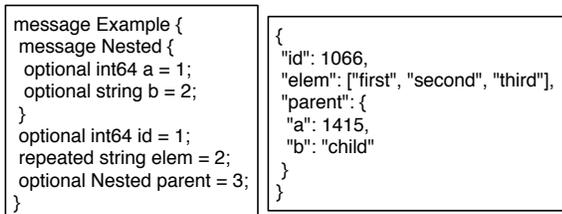
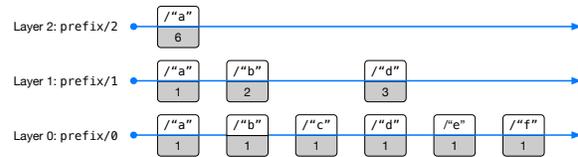
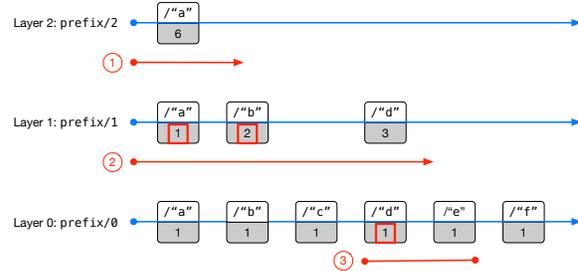


Figure 4: Example record definition and an example.



(a) Each contiguous key range used in by the skip-list is represented by a blue line and prefixed with the leading prefix of that subspace. Key-value pairs are shown as points on the keyspace line with a key and value (with a grey background).



(b) An example of finding the rank for set element "e". Scans of a range are shown with red arrows. (1) Scan the prefix/2 subspace and find only a, which comes before e. (2) Scan the prefix/1 subspace. Use the same-level fingers from (prefix, 1, "a"), (prefix, 1, "b"), contributing 1 and 2 to the sum, respectively. The last set member found is d, which also comes before e. (3) Scan the prefix/0 subspace and find e. During our scan, use the same-level finger (prefix, 0, "d") which contributes 1 to the sum, yielding a total rank of 4 (the lowest ordinal rank is 0).

Figure 5: An example RANK index with 6 elements.

index into multiple sub-indexes and can be used for indexing aggregations like SUM and COUNT. The `KeyWithValue` key expression has two sub-expressions, one of which is included in the index entry's key and the other of which is in the entry's value. Such indexes are used as covering indexes that satisfy queries without resolving the indexed record.

B RANK AND TEXT INDEX TYPES

RANK indexes. The RANK index type provides efficient access to records by their ordinal rank (according to some key expression) and conversely to determine the rank of a field's value. For example, in an application implementing a leaderboard, finding a player's position in the leaderboard could be implemented by determining their score's rank using a RANK index. Another example is an implementation of a scrollbar, where data (e.g., query results) is sorted according to some field and the user can request to skip to the middle of a long page of results, e.g., to the k -th result. Instead of linearly scanning until the k -th result is found, using continuations to restart the scan if it runs too long, the client can query for the record with rank k and begin scanning there.

Our implementation of the RANK index stores each index entry in a probabilistic augmented skip-list (Cormen et al. [30] describe a tree-based variant) persisted in FoundationDB such that each level has a distinct subspace prefix. Duplicate keys are avoided by attempting to read each key before inserting it. The lowest level of the skip-list includes every index entry, and each higher level contains a sample of entries in the level below it. For each level, each index entry contains the number of entries in the set that are greater or equal to it and less than the next entry in that level (all entries in the lowest level have the value 1). This is the number of entries skipped by following the skip-list “finger” between one entry and the next. In practice, an explicit finger is not needed: the sort order maintained by FoundationDB achieves the same purpose much more efficiently. Figure 5(a) includes a sample index representing a skip-list with six elements and three levels.

To determine the ordinal rank of an entry, a standard skip-list search is performed, starting from the highest level. Whenever the search uses a finger connecting two nodes on the same level, we accumulate the value of the first node, i.e., the number of nodes being skipped. An example of this computation is shown in Figure 5(b). The final sum represents the rank. Given a rank, a similar algorithm is used to determine the corresponding index entry. A cumulative sum is maintained and a range scan is performed at each level until following a finger would cause the sum to exceed the target rank, at which point the next level is scanned.

TEXT indexes. The TEXT index enables full-text queries on the contents of string fields. This includes simple token matching, token prefix matching, proximity search, and phrase search. The index stores tokens produced by a pluggable tokenizer using a text field as its input. Our inverted index implementation is logically equivalent to an ordered list of maps. Each map represents a postings list: it is associated with a token ($token_i$) and the keys in the map are the primary keys (pk_j) of records containing that token in the indexed text. Each value is a list of offsets in the text field containing that token expressed as the number of tokens from the beginning of the field. To determine which records contain a given token, a range scan can be performed on the index prefixed by that token to produce a list of primary keys, one for each record that contains that token. One can similarly find all records containing a token prefix. To filter by token proximity or by phrase, the scan can examine the relevant offset lists and filter out any record where the tokens are not within a given distance from each other or do not appear in the correct order.

To store the TEXT index, we use one key for each token/primary key pair, with the offset list in the value:

```
(prefix , token1 , pk1) → offsets1
(prefix , token1 , pk2) → offsets2
```

```
(prefix , token2 , pk3) → offsets3
(prefix , token3 , pk4) → offsets4
(prefix , token3 , pk5) → offsets5
(prefix , token3 , pk6) → offsets6
```

Note that the prefix is repeated in each key. While this is true for all indexes, the overhead is especially large for TEXT indexes due to the large number of entries. To address this, we reduce the number of index keys by “bunching” neighboring keys together so that for a given token, there might be multiple primary keys included in one index entry. Below is an example with a bunch size of 2, i.e., each index entry represents up to two primary keys.

```
(prefix , token1 , pk1) → [ offsets1 , pk2 , offsets2 ]
(prefix , token2 , pk3) → [ offsets3 ]
(prefix , token3 , pk4) → [ offsets4 , pk5 , offsets5 ]
(prefix , token3 , pk6) → [ offsets6 ]
```

To insert a token t and primary key pk , we perform a range scan and find the biggest key L that is less or equal to $(prefix, t, pk)$ and the smallest key R greater than $(prefix, t, pk)$. We then place a new entry in L if the insertion does not cause this entry to exceed the maximum bunch size. If it does, the biggest primary key in the bunch is removed (this might be pk and its list of offsets) and is inserted in a new index key. If the size of R 's bunch is smaller than the maximum bunch size, the bunch is merged with the newly created one. To delete the index entry for token t and primary key pk , the index maintainer performs a range scan in descending order from $(prefix, t, pk)$. The first key returned is guaranteed to contain the data for t and pk . If pk is the only key in the bunch, the entry is deleted. Otherwise, the entry is updated such that pk and its offsets are removed from the bunch, and if pk appears in the index key, the key is updated to contain the next primary key in the bunch.

Inserting an entry requires reading two key-value pairs and writing at most two (though usually only one). Deleting an entry requires reading and writing a single key-value pair. This access locality gives index updates predictable latencies and resource consumption. However, certain write patterns can result in many index entries where bunches are only partially filled. Currently, deletes do not attempt to merge smaller bunches together, although the client can request compactions. Table 2 shows a worked calculation of the possible space savings for this approach. To demonstrate space savings due to bunching, we used Melville's *Moby Dick* [39], broken up into 233 ~5 kilobyte documents. With whitespace tokenization, each document contains ~431.8 unique tokens (so the primary key can be represented using 3 bytes) each appearing an average of ~2.1 times within the document and with an average length of ~7.8 characters. For the calculation, we use 10 bytes as the prefix size (smaller than the typical size we use in production). In practice, we find that this index required ~4.9 kilobytes per document because not

	No bunch	Bunch size 20
Prefix	10 bytes	10 bytes
Token	7.8 bytes	7.8 bytes
Primary key	3 bytes	3 bytes
Encoding overhead	2 bytes	2 bytes
Key size	22.8 bytes	22.8 bytes
Offsets	3 bytes	2 bytes \times 20 = 40 bytes
Primary keys	—	3 bytes \times 19 = 57 bytes
Value size	3 bytes	97 bytes
Total size / entry	25.8 bytes	119.8 bytes
Approx. # entries	431.8	431.8/20 \approx 21.6
Total size / document	11.1 kB	2.6 kB

Table 2: Worked example illustrating the space savings provided by the bunched map.

every bunch is actually filled. In fact, the average bunch size was ~ 4.7 , significantly lower than the maximum possible because some words appear rarely in the text—some even only once. To optimize further, we could bunch across tokens. An alternative would be to implement prefix compression in FoundationDB, but even then, there is per-key overhead in both the index and FoundationDB’s internal B-tree data structure, so using fewer keys is still beneficial.

C QUERY PLANNING AND API

The Record Layer has extensive facilities for executing declarative queries on a record store. While the planning and execution of declarative queries has been studied for decades, the Record Layer makes certain unusual design decisions.

Extensible query API. The Record Layer has a fluent Java API for querying the database by specifying the types of records that should be retrieved, Boolean predicates that the retrieved records must match, and a sort order specified by a key expression (see Appendix A). Both filter and sort specifications can include “special functions” including aggregates, cardinal rank, and full-text search operations like n -gram and phrase search. This query language is akin to an abstract syntax tree for a SQL-like text-based query language exposed as an API, allowing consumers to directly interact with it in Java. Another layer on top of the Record Layer could provide translation from SQL or another query language.

Query plans. While declarative queries are convenient for clients, they need to be transformed into concrete operations such as index scans, union operations, and filters in order to execute the queries efficiently. The Record Layer’s query planner converts a declarative query specifying the records to return into an efficient combination of operations on the stream of records. The Record Layer exposes these query plans through the planner’s API, allowing its clients to cache or otherwise manipulate query plans directly. This provides functionality similar to that of a SQL PREPARE statement but

with the additional benefit of allowing the client to modify the plan if necessary [45]. Like SQL PREPARE statements, queries (and thus, query plans) may have bound static arguments (SARGS). In CloudKit we have leveraged this functionality to implement CloudKit-specific planning behavior by combining multiple plans produced by the Record Layer and binding the output of one plan to an argument of another.

Cascades-style planner. We are currently evolving the Record Layer’s planner from an ad-hoc architecture to a Cascades-style rule-based planner [36]. This new design supports deep extensibility, including custom planning logic defined completely outside the Record Layer by clients. Internally, we maintain a tree-structured intermediate representation (IR) of partially-planned queries that includes a variety of *expressions*, including both logical operations (such as a sort order needed to perform an interaction of two indexes) and physical operations (such as scans of indexes, unions of streams, and filters). We implement the planner’s functionality through a rich set of planner rules, which match to particular structures in the IR tree, optionally inspect their properties, and produce equivalent expressions.

Rules are automatically selected by the planner, but can be organized into “phases”: for example, it is better to scan part of an index than to filter all records. They are also meant to be modular; several planner behaviors are implemented by multiple rules acting in concert. While this architecture make the code base easier to understand, its key benefit is allowing complex planning behavior by combining the available rules. The rule-based architecture allows clients, who may have defined custom query functions and indexes, to plug in rules for making use of that additional functionality. For example, a client could implement a geospatial index, extend the query API with custom functions for bounding box queries, and write custom rules to plan geospatial filters as scan of the geospatial index, all while making use of built-in rules.

Future directions. The experimental planner’s IR and rule execution system were designed to anticipate future needs. For example, the data structure used by the “expression” intermediate representation currently stores only a single expression at any time: this planner repeatedly rewrites the IR each time a rule is applied. However, it could be seamlessly replaced by the compact Memo data structure [36] which succinctly represents a huge space of possible expressions. The Memo structure replaces each node in the IR tree with a group of logically equivalent expressions. Each group can then be treated as an optimization target where each member of the group represents a possible variant of the group’s logical operation. Memo allows optimization work for a small part of the query to be shared (or memoized) across many possible expressions. Adding the Memo structure paves the way to a cost-based optimizer, which uses estimates of a cost metric to choose from several possible plans.