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 datastore 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.

1 INTRODUCTION

Many applications require a scalable and highly available backend that provides durable and consistent storage. Developing and operating such backends presents many challenges. Data loss is unacceptable and availability must be provided even in the face of network unreliability, disk errors, and a myriad of other failures. The high volume of data collected by modern applications, coupled with the large number of users and high access rates dictate smart partitioning and placement solutions for storing the data at scale. This problem must be addressed by any company offering stateful services. Yet, despite decades of academic and industrial research, it is notoriously difficult to solve correctly and requires 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 [9] democratizes industrial-grade highly-available and consistent storage, making it freely available to anyone as an open source solution [10] 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, a simple 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.

Furthermore, as stateful services grow, they must support the needs of many 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.

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 [20, 21]. 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. The Record Layer is stateless and lightweight, with minimal overhead on top of the underlying key-value store. These lightweight abstractions allow for multi-tenancy at extremely large scale: the Record Layer allows creating isolated logical databases for each tenant—at Apple, it is used to manage billions of

This is a preprint. For the latest version of this paper, please see:
https://foundationdb.org/files/record-layer-paper.pdf

一些在传统数据库中不常见的功能：
1. 嵌套类型；
2. 提交版本上的索引（后面提到的可用来做到CDC）；
3. 跨record类型的索引（相当于一个索引针对多个表）；

Record Layer：
1. 无状态；
2. 支持多租户；每个租户的状态，索引都会被保存在一个 独立的 “逻辑数据库”中。

一个应用程序中，每个用户的数据都是一个独立的“逻辑数据库”；因为它们属于同一个应用程序，所以它们的schema是相同的。

这里：“不同租户的数据”，指的应该是“不同用户的数据”。 其实，“租户”也就是一个“用户”。

FoundationDB是一个工业级的 数据库

但是FDB 仅仅提供了KV 接口，不方便用户使用。

随着“有状态”的服务增多，需要支持的“用户”和“APP”就会越来越多，而如果这些用户和APP都共用底层数据库，那就有了“多租户”的需求。

但为了实现“多租户”有很多挑战：包括
1. 隔离、2. 资源分片、3. 面对负载不断增长时如何弹性伸缩。

许多数据库系统在”计算”和”存储”级别混合来自不同租户(用户)的数据。改造这种系统，从而“资源隔离”非常难。

对“有状态”的服务进行 弹性扩展 尤其困难，因为 无法对“state”进行任意分区。
例如：如果不牺牲事务更新、性能或两者都牺牲，那么 数据和 索引 就不能存储在完全独立的存储集群上。

FDB Record Layer：
1. 基于FDB；
2. 面向Record；
3. “关系型”相似的语义；

FDB Record Layer 以最小的开销 构建在K-V引擎：
1. “无状态”；
2. 轻量；

正是因为这些轻量级抽象，Record Layer才能够支持极大规模的“多租户”：允许为每个“租户（用户）”创建独立的逻辑数据库。

在Apple，它用于管理”数十亿”这样的数据库 - 同时提供熟悉的功能，如结构化存储和事务索引 保养。

>> “Record Layer + FDB”的架构 是一个“存储计算分离”的架构，其中：
1. Record Layer：计算层；
2. FDB：存储层；

一些名字约定：
1. record： 翻译成“记录”，类似于 关系型数据库 中的一行 。
注意： record 和 document 最大的不同是：在document中，是可以动态的添加字段的，但是record的字段是固定的。
2. record-oriented：面向记录的
3. record type：PB中的一个message类型，因为除了 最顶层的record（message）之外，其余的record都会被映射 最顶层message的一个字段。所以这里的“record type”指的就是PB中不同的message类型，大部分场景下（除了最顶层的record），类似于 关系型数据库 中的一列；

名词解释：
1. stateful service：有状态的服务；如果一个应用程序时，在用户使用时需要保留下来信息的（供下次使用该APP时能够再次使用的信息，比如注册信息，订单信息等），
广义上一个需要向数据库中保留下来任何信息的 应用程序，都是有状态的；

Record Layer的贡献：
1. 语义类似于“关系数据库”的语义。
2. “record store”的抽象和一套操作它的技术，能够使 数十亿 逻辑租户能够在一个FDB集群中 “逻辑上”独立操作各自的 “逻辑数据库”。
3. 高度可扩展的体系结构：client可以自定义核心功能，包括schema管理和索引。
4. 轻量级设计，在底层键值存储之上提供丰富的功能。

Preprint. Latest version at
https://foundationdb.org/files/record-layer-paper.pdf

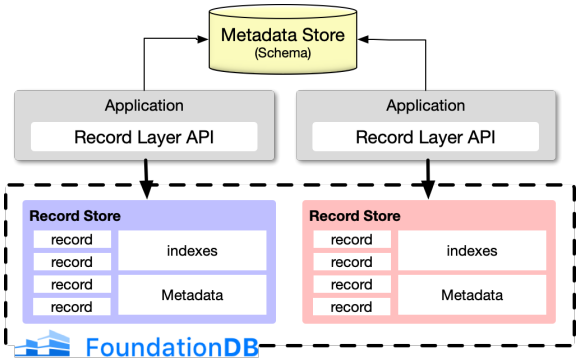


Figure 1: Architecture of the Record Layer.

such databases—all while providing familiar features such as structured storage and transactional index maintenance.

The Record Layer (Figure 1) represents structured records as Protocol Buffer [13] messages that include typed fields and even nested records. Since an application's schema inevitably (不可避免地) changes over time, the Record Layer includes tools for schema management and evolution. It also includes facilities for planning and efficiently executing declarative (陈述的, 说明的) 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 seemlessly (无缝地) "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 (大规模地) 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 (分配，分派) 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 (运行状态). 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, since everything needed to interpret (解释，翻译) and operate each record store is found in its key range.

使用 Protocol Buffer 表示数据结构：
1. 字段是有类型的；
2. 支持 嵌套类型；

（有工具来）支持 Schema Change

支持使用“描述性语言”来进行查询（使用相应类型的索引）。

利用了FDB的高级特性。
比如：原子性的更改“聚合索引”（支持“并发”、“无冲突”的更新）。

可以让 client 方便的进行扩展。比如：
1. “用户自定义的索引”可以 无缝地 插入到 索引管理器 和 查询规划器。
2. 序列化支持“用户自定义”加密和压缩算法。

能够大规模支持“多租户”的两个架构选择：
1. Record Layer是无状态的。

“无状态”的好处：
1. 可以非常容易的扩展“计算”，只需要增加 节点 就可以了。
2. 对于 负载均衡 和 路由 模块，就只需要考虑 数据的分布即可，不需要考虑 计算节点的位置（因为所有的计算节点都可以提供服务）
3. 无状态 的服务器，只需要为每个 client 分配很少的资源即可（因为无状态，即不需要维护每个client的状态）

2. 抽象成“record store”，可以做到 “资源共享”和“弹性”

1. 在每个“record store”中，都封装了“逻辑数据库”的所有状态，包括 数据（经过序列化的记录），索引，甚至 运行状态。
2. 每个Record store 都被赋予 一段连续的key区间。 也就是说，属于不同“租户”的数据，都是逻辑隔离的（因为属于不同的 key区间）

如何迁移租户： 只需要将 对应的 一个连续区间 的数据 迁移走 即可。

因为该 record store 相关的所有数据，都在这个数据区间内。

每个 “record store”，就对应一个 “逻辑数据库”

>> “record store”与“租户” 的映射关系是 “多对多”的。

- An open source layer on top of FoundationDB with semantics akin (同类的，类似的) 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.
- A highly extensible architecture, clients can customize core features including schema management and indexing.
- A lightweight design that provides rich features on top of the underlying key-value store.

为“每个应用程序”的“每个用户”维护独立的record store(逻辑数据库)来提供极端的多租户

在CloudKit中，有数十亿的“逻辑数据库”，使用 几千个 schema(表结构)。

将来，Record Layer 可能会和 其他存储模型（队列模型，图模型），共同利用 FDB作为 底层的通用存储引擎， 并且在所有存储模型中 保持 事务一致性。

>> 这里，Record Layer相当于 关系型数据库 的 行模型，和 其他存储模型是并列 的地位。

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 (极度的，极端的) 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 schemas. In the future, we envision (预想) that the Record Layer will be combined with other storage models, such as queues and graphs, leveraging FoundationDB as a general purpose storage engine and remaining transactionally consistent across all these models. In summary, this work makes the following contributions:

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 is responsible for membership and configuration of the first cluster. This allows FoundationDB to achieve high availability while requiring only $F + 1$ storage replicas to tolerate F failures [23]. One of the distinguishing (有区别的) features of FoundationDB is its deterministic (确定性的) simulation testing framework, which can simulate entire clusters under a variety of failure conditions in a single thread with complete determinism (决定论) in a short period of time. For example, in the past year, we have run more than 250 million simulations and have simulated more than 1870 years and 3.5 million CPU hours. This rigorous (严格的) testing in simulation makes FoundationDB extremely stable and allows its developers to introduce new features and releases in a rapid cadence (节奏), unusual among similar strongly-consistent distributed—or even centralized—databases.

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

两种逻辑集群：
1. 存储数据，初始事务；
2. 协调者

注意：在FDB中，能容忍的出错率，不是平常说的“小一半”。而是：只需要剩余一台机器即可。

FDB的一个重要特性：有一个 “确定性模拟测试框架”：可以在很短的时间内，用单线程 来模拟整个集群处于 各种失败的情况。

在其它大多数数据库中，经常是将 “存储引擎”，“数据模型”，“查询语言”三者捆绑起来，用户要么必须全部选择，要么全部不选择。

modular approach: it provides a highly scalable, transactional storage engine with a minimal yet carefully chosen set of features. *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 do not create read conflicts and so a concurrent change to that value would not cause the transaction to abort (they can, however, cause other transaction to abort). Instead, the operation is applied atomically at commit time. 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, reducing conflicts by trading off isolation semantics. *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. 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 as it was when it started, but 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 10 MB in size, 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. Keys are part of a single global namespace, and it is up to applications to divide and manage that namespace 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 with FoundationDB, 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 PRINCIPLES

The Record Layer is designed to provide lightweight, scalable operations on structured data, including transaction processing, schema management, and query execution. 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 so that the layer itself is completely stateless. 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

---

FDB的有一些限制
1. key和value的大小：
   key最大为10KB，value最大为100KB
   建议的大小是：key为32B，value在10KB以下
2. 事务的大小：以下内容的总大小
   所写的数据 key + value
   需要检查冲突的数据（读和写）： key

对于“需要检查冲突的数据”：
1. 在SI级别下，只检查 写 的数据
2. 在SSI级别下，读 和 写的数据都要检查

FDB 采用了模块化的方法： 提供可高度伸缩的、支持事务的存储引擎，并且具有 最少但经过仔细选择 的特性集。

模块化的

Layer 可以构建在‘存储引擎’的上层，从而提供 多种存储模型和功能。

功能

大量的；实质的

读：MVCC
写：乐观的并发控制

事务的读写操作，都不会 因为并发事务的读写操作而阻塞。

在client开启事务时，需要通过调用getReadVersion(GRV)来获取一个read_version，并且使用这个version来进行MVCC读取。

这个read_version是当前数据库最新的提交版本。

因为SSI的隔离级别，即要检测 R-W,W-R,W-W冲突。

所以，一个包含写操作的事务，只有再它所读取的数据，没有被其它事务并发修改（从事务开始 到 提交事务这段时间）的情况下，才能够成功提交。

数据被 持久化到多台节点以后，才会返回给client

瞬间的；即时的

执行顺序：
1. 在一个txn内的多个操作是并行执行的；
2. 保持 编程 的顺序；
3. 对于“先写后读”的事务，保证读能够 读取到 之前 写的数据。

FDB中，一个事务最长为 5秒

付报酬，补偿

在“单个Key”操作上，FDB提供了 原子的“read-modify-writde”操作。

1. 这种操作不会造成“读冲突”，所以该事务不会被回滚（但是注意可能会造成其它事务被回滚）。
2. 这种操作在commit时会被原子提交；

使用场景举例：被多个client并发修改的计数器。

可以定制默认的并发控制行为（隔离级别）。

“Snapshot read”（SI）级别：不会因为读写冲突而造成事务失败。

在任意隔离级别下：都可以显式地 添加或者移除 冲突的区间

使用场景举例：一个“阈值”变量，不希望因为对它有读取操作，造成一些事务失败：
1. 在SI级别下进行读取；
2. 在SSI级别下，主动移除对该key读取的冲突；

不透明的

和RocksDB类似，在FDB（K-V数据库）中的Key和Value都是 二进制值。

Keys 是一个独立的 命名空间；这个命名空间由 上层应用程序 来决定如何进行 拆分和管理；有一些辅助的API，比如 Tuple Layer 和 Directory Layer.

FDB支持 在Key的顺序上进行 range查询和 range删除

注意：range操作，必须和 key的顺序一直，如果不一致，则不支持

这两个Layer是内置在FDB中的。

Tuple Layer：把一个tuple 编码到对应的K-V中。注意在编码的过程中，是保存的。

Tuple中的元素是有类型的。

tuple的公共前缀被序列化为 公共字节前缀，这也定义了一个subspace。

directory layer：定义了一个 逻辑的目录结构，把 一些 较长的字符串 映射成 较短的。从而在编码的时候，减少key的长度；

滑动窗口

内部实现机制：使用一个 滑动窗口的分配算法，能做到可以并发分配，也能保持需要总内存较少

问题：“滑动窗口”的内存分配是什么机制？

Record Layer 是面向 “结构化”数据的

一共有 5 点

短暂的；朝生暮死的

Record Layer把所有状态都保存在FDB中，而Record Layer自身是无状态的。

比如：Record Layer不在内存中维护 当前的数据位置指针(cursor)；而是把这个 指针 返回给client，这样client在下次访问时带着这个指针进行访问。

“无状态”的3个优点：
1. 简化路由规则：1个请求可以被路由到 任何服务器（因为服务器是无状态的，所以任何服务器都可以处理）；也可以使用同一个cursor 来获取多次结果；
2. 能充分简化 大规模系统的运维工作：如果server遇到问题，它只需要重启即可，而不需要把任何状态信息传递到其它节点。
3. 把state保存在FDB中，可以做到 事务的修改这些state：所以也就不需要 额外的工具来 校验这些 state信息的完整性；

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 Protocol Buffer [13] 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, Protocol Buffer messages support nesting of record types within a field and repeated instances of the same field; these advanced features allow implementations of list and map data structures within a single record, for example. Because the Record Layer is designed to support millions of independent databases with a common schema, it stores metadata separately from the underlying data. The common metadata can be updated atomically for all stores that use it.

Efficiency. Several parts of the Record Layer's design enable it to run efficiently at scale. For example, the Record Layer is implemented as a library that can be embedded in its client rather than an independent client/server system; while the Record Layer can be (and in many cases is) embedded in a server environment, it makes no presumptions and imposes few requirements on how the 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 and versioning, both within the layer's implementation and exposed to clients via its API.

Extensibility. In the spirit of the FoundationDB's layered architecture, the Record Layer exposes a large number of extension points though 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 for clients to add features that are left out of the Record Layer's core, such as memory pool management and arbitrary sorting, in a way that makes sense for their use case. 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 schemas. Records in a record store are Protocol Buffer messages and a record's type is defined with a Protocol Buffer definition. 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.

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, defining 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.

Unlike traditional relational databases, all record types within a record store are interleaved within the same extent, and hence both queries and index definitions may span all types of records within the store. From a relational database perspective, this is akin to having the ability to create a single index across all tables that include the same column.

Primary keys and indexes are defined within the Record Layer using *key expressions*, covered in detail in Section 6.1. A key expression defines a logical path through records; 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.

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 allows scan or query operations that exceed the transaction time limit to be split across multiple transactions.

The Record Layer exposes many of FoundationDB's transaction settings to allow faster data access. For example, record prefetching asynchronously preloads records into FoundationDB client's read-your-write cache, but does not return it 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 in the rare case that the FoundationDB master is disconnected from the rest of the system but is not yet aware of this fact. 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 they perform validations at commit stage. 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 keyspace 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 show up 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 keeps track of the highest version it has been accessed with by storing it 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 records of that type yet. 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

切分的类型：保存在每个record的开头，并且包含最后提交的版本号。在每次读取时，都会返回该版本号。

拼接

支持可插拔的 序列化库：可以对数据进行压缩 和 加密。

可插拔的

所有的Record Layer操作，都是在 流式数据上 提供了一个cursor，并支持 “重新开始该操作”

“continuation” 是一个二进制值，用来表示： 在当前数据流中，下一次的开始位置。

支持延续

不透明的

在结果中，同时包含 结果数据 和 continuation信息；从而client在下一次请求时，只要带上continuation信息，就可以继续上次操作。

通过在交互信息中传递continuation，在迭代数据的过程中，1. server端就不需要维护任何 状态信息；2. 将包含较长时间的scan或者query操作的事务，切分为多个事务

为了效率，Record Layer暴露了很多 事务相关的设置。

分配

1. 数据可以 异步地预取，并且保存在客户端的 read-your-write 缓存中。

因为数据已经在cache中，所以在真正读取的时候，就可以减少一次“上下文切换”，和“反序列化”。

2. 可以控制事务的“一致性级别”。

3. 提供“causal-read-risky”标记

这个标记 可以加快getReadVersion的速度，但是可能会获取一个过时的 read version(即读取不到最新的数据)

>> 说明：就是原来的master被网络分区了，集群出现了 多个leader的情况。 这个问题，说明FDB中，没有像Raft那样有Leader Lease的约束。

注意：有写操作的事务，不会返回一个 过时的数据，因为它会在commit阶段进行冲突检查

4. 可以在Record Layer中 缓存 read version。

注意：
1. 有可能读取到 过时的数据；
2. 如果该事务有写操作，会增大 该事物失败的可能性（因为时间戳比较早，即相当于事务的跨度变大了）

预加载

陈旧的

推荐使用 缓存read version的方式 的场景：
1. 对于“只读事务”非常有用：因为这种事务不需要读取到最新的数据；
2. 低并发的场景：因为这种场景，因为冲突而失败的概率很小；

为了帮助用户在FDB的key space的形式下组织他们的record store，FDB提供了KeySpace API：把key space映射成 类似文件系统路径的方式。

KeySpace API保证所有的路径 是 “逻辑隔离”和“没有重叠”的。

在适当的时候，会使用 “directory layer”把 路径 映射为 较小的数字（能够减少key的长度）

露面，显露

在大部分部署的场景，metadata信息都是被尽量缓存下来的。（这样在每次解析数据时，就不需要再额外的去获取）

注意：这里主要讲的“加减列”，但是不包含“修改列类型”

从PB继承的特性：
1. 可以添加字段（在旧的记录中表现为 未初始化）；
2. 可以添加新的record 类型，并且不影响已有record类型；

最佳实践：
1. 不要重用 PB结构中的 field number；
2. 在移除字段时，使用 deprecated，而不是直接移除；

metadata的版本：单调递增。

在每个record store中，都记录着它所见过的 元信息的 最大版本号。

保存在该record store的 header中（以K-V的形式）；在record store 被“打开”的时候，会读取这个header，并与当前client所记录的 相比较。

版本不相等有两种情况：
1. 数据库中记录的版本更新：说明当前client中缓存的元数据已经过期了；
2. 数据库几种的版本较老：说明这个schema已经有新版本了，可能需要进行一些 操作，比如添加一些新的record类型，或者一些字段。

重建索引

新建索引的两种情况：
1. 一个新的record type：直接生效，因为没有历史数据，不需要重建；
2. 一个已经存在的record type：如果已经有数据，需要 重建索引(backfill)

在重建索引时：
1. 如果数据量很少，那么很快就可以重建完成；
2. 如果数据量很大，那么一个事务无法重建完成（FDB中，每个事务最长为5秒），那么这个事务就转化为 一个“后台任务”进行，直到回填完成，索引才对外可见。

元信息的几个版本号：
1. schema version: 也称为 metadata version，即元数据的version；
2. storage format version: 数据编码格式的版本号，在修改了数据的编码方式时（压缩 或者 加密方式）使用；
3. application version: client用来跟踪 数据的格式（即数据的格式不仅仅 只在元信息中体现）

因为 所有的record type都在同一个 key space中，
所以在重建索引时，所有的记录都需要被 扫描，用来生成index的数据

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.

问题：
解决数据的“编码”方式：比如“压缩”格式和“加密”格式。

解决方法：
保存“存储格式”的版本号，也保存在“record store的header中”。

如果变化了“存储格式版本号”，可能需要：
1. 重新对数据进行格式化；
2. 在某些情况下，需要对老格式的数据开启“兼容模式”。

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 必需 reformatting small amounts of data, or, in some cases, enabling a compatibility 兼容性 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 再归一化. 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.

比如：在一个record type中，把 嵌套类型 提升到顶级。

校验“application version”的时机：
1. 在打开 record store时；
2. 在 业务代码中检查；

如果这种变化“较多”的时候，可以用“application version”来知道 与 最新版本的差距；

6 INDEX DEFINITION AND MAINTENANCE

Record Layer的 索引 是持久化的，以“数据流”的形式来维护。

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 字典序的 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.

index和data是在 同一个事务内 更新的，也就是说，两者一直是一致的。

高效的保证条件：
1. 索引的更新：强依赖 FDB的多key事务 的高性能；
2. 高效的scan index：依赖FDB的range查询，依赖在FDB中key是有序存储的；

每个index的数据，在record store中都是保存在一个独立的subspace下，所以是连续存储的，所以删除索引很容易（range clear操作）。

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 filter的作用：
1. 让一些记录不需要建立索引；
2. 能支持“稀疏”索引；
3. 减少索引占用的存储空间，以及 维护索引的开销；

Unlike indexes in classic relational databases, indexes are first-class citizens 一等公民 in the Record Layer. That is, they can be scanned directly (with or without retrieving the record to which the index points) rather than only being used internally to optimize queries. As a result, the Record Layer supports the ability to define and utilize indexes for which a query syntax may be difficult to express.

在Record Layer中，索引是“一等公民”。
意思是：可以直接单独使用索引。

可以 定义和利用索引来实现一些 高级查询。
>> 意思是说：Record Layer中的索引，本质上和 record是相同的；只不过是它包含一个和record的映射关系；但是因为“映射关系”是任意的，所以可以对这个映射关系进行定制，从而实现高级功能

对比：
在传统数据库中，索引 不能 单独使用，而是只能用来 优化查询。

问题：什么样的高级查询？举个例子？

在插入（更新）一条数据，“索引管理器”的工作步骤：
1. 先（根据主键）检查是否已经存在；
   如果已经存在，那么“索引管理器”先 删除或更新 索引条目，并删除 旧值条目；
2. 将 当前记录 插入到 record store.
3. “索引管理器” 插入或更新 新值 对应的索引条目。

每定义一个index，都要实现一个 index maintainer（索引管理器）：
用来在更新data的时候，进行索引的更新。

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

已经内置了很多的“索引管理器”。

“索引管理器”的抽象是暴露给用户的，允许用户 自定义 索引类型，并且进行“事务地”管理更新。

When a record is saved, we first check if a record already exists with the new record's primary key. If so, registered index maintainers remove or update associated index entries for the old record and delete the old record 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.

“索引管理器”做了很多优化。 比如：如果该记录的旧值 和 新值的类型相同，并且 某些索引对应的列值 没有被修改，那么就不需要修改 相关的索引条目。
注意：因为 数据本身的条目都是要修改的，所以不影响 事务冲突的检测。

Online index building. The Record Layer includes an online index builder used to build or rebuild indexes in the background. To ensure that the index is not used before it is fully built, indexes begin in a *write-only* state, where writes maintain the index but queries do not use it yet. 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. Online index building is broken into multiple transactions to reduce conflicts with concurrent mutations.

使用“在线索引构建器”来在“后台”进行索引的 创建 和重建。

在创建完成之前，index不能被读取。
回填时，开始的状态是write-only，即可以写入，但是不能读取。

在创建完成后，index的状态变为readable，才是可以读取的。

为了减少和正常写请求的冲突，索引的回填 被拆分为 多个事务。

6.1 Key Expressions

索引的定义包含两部分：1. 索引类型； 2. key expression

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.

key expression：是一个函数，把一条记录 映射成 一条或多条tuple。然后“索引管理器”会使用这些tuple去 生成 索引数据；

```
message Example {
 message Nested {
  optional int64 a = 1;
  optional string b = 2;
 }
 optional int64 id = 1;
 repeated string elem = 2;
 optional Nested parent = 3;
}
```

```
{
 "id": 1066,
 "elem": ["first", "second", "third"],
 "parent": {
  "a": 1415,
  "b": "child"
 }
}
```

Figure 2: Example Protocol Buffer message definition and a sample message instance.

Field key expressions. The simplest key expression is `field`. When evaluated against the sample record in Figure 2, `field`

key expression的分类：
1. field key expression：针对基本类型；
2. nest key expression：针对嵌套类型；
3. 可以使用 concat() 定义组合索引：结果是tuple的 笛卡尔积；
4. 高级的key表达式
   4.1 Record Type Key Expr：把原始record按照type进行拆分，每个type都有唯一的值；
   4.2 Function Key Expr：用户自定义函数（处理record和它的字段）；
   4.3 Group By Key Expr：定义一组聚合方式；
   4.4 KeyWithValue Key Expr：一个组合表达式，定义一个列子集（类似于storing columns）
5. 用户自定义 key表达式

index type:
1. VALUE：提供从一个 或 多个 字段 到 主键的映射；

对于数组类型，有两种FanType:
1. Concatenate;
2. Fanout

对比：
1. field("elem", Concatenate) -- (["first", "second", "third"])
2. field("elem", Fanout) -- ("first"), ("second"), and ("third").

("id") yields the tuple (1066). Unlike standard SQL columns, Protocol Buffer fields permit repeated fields and nested messages. We support nested messages 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").

对于 数组类型（repeated字段），
1. Concatenate:会将数组的所有值作为一个tuple返回；
2. Fanout：会将数组的每个元素作为一个tuple返回。

连结 使连续

展开，分别

FanType determines the type of queries supported by an index. By default, each tuple (along with the record's primary key) becomes an index entry. With Concatenate, the index can efficiently find records that have values matching a certain sequence or beginning with a certain sequence, since Protocol Buffers preserve the order of repeated values [14] (e.g., all records where elem begins with "first"). Such an index can also be used to lexicographically sort repeated values. Fanout can be used to find records containing a particular value anywhere in the list of repeated values.

FanType决定了 索引的功能；（注意只有数组类型，即repeated类型，才有FanType属性）

key expression生成的每个tuple，都会和该记录的主键一起，生成一条 索引记录

1. 使用Concatenate：
1.1 可以快速查找 以指定值开头（指定前缀）的 记录；
1.2 可以用来对数组中元素进行排序

因为在PB中，repeated类型是有序的，所以可以找特定的前缀。

2.使用 Fanout类型，可以快速的找到 数组中的任何成员。

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 any of the sub-expressions produce multiple values, the compound expression will produce the tuples in the Cartesian product of the sub-expressions' values.

如何定义 组合索引：通过 'concat' 联合多个 tuple。

如果子表达式 会生成多个tuple，那么整体就生成一个将所有的子表达式 "笛卡尔积" 的结果（对 使用concat连接的子表达式 之间）。

笛卡尔的

Advanced key expressions. In addition to its core key expressions, the Record Layer includes a variety of key expressions to support particular features. For example, the record type key expression produces a value that is guaranteed to be unique for each record type. This key expression is useful for defining primary keys that divide the primary record subspace by record type, allowing users to treat the record types more 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.

"高级的key表达式"：为了支持一些特定的功能。

record type key expr：针对每个record类型，只生成唯一的key。

这个key expr，根据record type把record分成 多个主键。

用途：
1.将原始记录，使用record type进行拆分，从而用户可以像传统数据库中的table一样去使用。
2. 满足那些关于record type的查询；比如：查询每种类型的record的数量

In addition to allowing clients to define their own 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. For example, one might define a "negation" function that is defined on some integer field of a record and returns a single tuple with that field negated. Function key expressions are very powerful and can allow, for example, users to define custom sort orders based on arbitrary functions of the records.

用户可以定义自己的 key expr.

function key expr：可以针对record和它的成员 执行任意的函数。

例如：可以针对某个integer类型的字段定义一个negation()，它的返回值是一个tuple，值是这个整数"取非"运算的结果。

构成的

否认，拒绝，取非运算

不定的，取非运算

Function Key Expr：可以做到 让用户提供对record的任意函数，从而自定义排序结果。

Another special key expression is groupBy, which defines "split points" used to divide an index into multiple sub-indexes. For example, a SUM index using the key expression field("parent").nest("a").groupBy(field("parent").nest("b")) enables efficiently finding the sum of the parent field's a field over all records where the parent field's b field is equal to "child".

这里的"SUM"是指索引的类型（下一节会讲到）

The KeyWithValue key expression has two sub-expressions, where the first is included in the index's key while the second is included in the value. This defines a covering index satisfying queries that need a subset of the fields included in the index value without an additional record lookup.

使用这种索引，会在，会在索引条目的value中保存一些列；这样在查询的时候，如果只需要这些列，就可以减少一次查询（因为可以直接从index条目的value中读取，而不需要"先找到PK，然后再用PK去查询"）

7 INDEX TYPES

索引的类型，决定了可以执行哪些"过滤"操作

The type of an index determines which predicates it can help 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 A describes the RANK and TEXT index types, used for dynamic order statistics and full-text indexing, respectively.

自定义索引类型的方法：实现和注册 自定义的 key expr 和 "索引管理器"；

Index可以跨越多个record type

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.

要求：在Key Expr中引用的字段，必须在（所对应的多个）record type中都存在。

用途：使用一组条件，在所有 record type中进行搜索

标准 条件

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.

最基本的索引类型：提供从（1个或多个）列到 主键的映射。

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

依赖 FDB的 原子更新 特性；

用途：对 统计信息的聚合和查询。

例如：使用SUM 对所有record的某个字段进行 求和。

在对所有记录求SUM时，index中只会有一行 数据，即 sum的结果。

可以包含 1个或多个group字段，这时结果会有多条，即每个group字段的 sum结果。

维护类似SUM索引的常用步骤：
1. 读取index的值；
2. 更新index的值；
3. 写回到index；

即这是一个典型的 "读-改-写" 的事务流程；但是这种实现方式，很容易造成事务因为冲突和失败（并发执行这个操作的时候）。

对于 Atomic mutation indexes，为了避免事务因为冲突而失败，采用了FDB的atomic mutation特性（事务不会因为冲突而失败）。

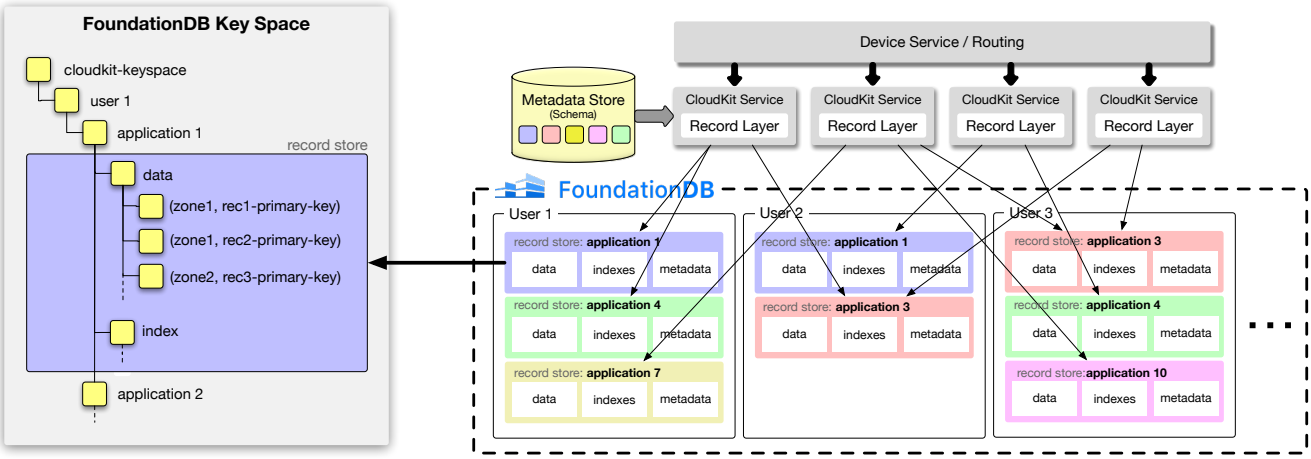


Figure 3: CloudKit architecture using the Record Layer.

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.

VERSION. VERSION indexes are very similar to VALUE indexes in that they define an index entry and a mapping from each entry to the associated primary key. The main difference between them is that a VERSION index allows the index's key expression to include a special "version" field: a 12 byte monotonically increasing value representing the commit version of the last update to the indexed record. Specifically, FoundationDB commits transactions in batches, where each transaction can include multiple operations on individual records. The first 8 bytes of the version are the FoundationDB commit version and identify the transaction batch. The following 2 bytes identify a particular transaction within the batch, and finally the last 2 bytes identify an operation within the transaction. The version is guaranteed to be unique and monotonically increasing with time, within a 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. However, the Record Layer must be able to determine the version of a record in order to perform index maintenance, e.g., to find the relevant index entries when deleting a record. To this end, the Record Layer writes a mapping from the primary key of each record to its associated version. This mapping is stored next to the key-value pair representing the Protocol Buffer value for this key, 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 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 at which point the appropriate Record Layer record store is accessed and the request is serviced.

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. 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, used for quota management.

8.1 New CloudKit Capabilities

CloudKit was initially implemented using Cassandra [4] 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, but 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 are 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. 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 applications on top of 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, requiring 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 A that now serves millions of users. Unlike traditional search indexing systems, all updates are done tranactionally 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 by creating only *n* key entries instead of the usual $O(n^2)$ keys needed to create all possible sub-strings for supporting *n*-gram searches. 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, commit versions assigned by different FoundationDB clusters are uncorrelated. This introduces a challenge when migrating data from one cluster to another; CloudKit periodically moves users to improve load balance 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 CloudKit increments it each time the user's data is moved to a different cluster. On every

1. 每个用户会对应一个唯一的subspace;
2. 每个application对应一个record store;

在使用gRPC来和CloudKit通信时，也是支持 交互事务的。

之前，系统中只有很少量的"system"索引，用户自定义的二级索引都是保存在 Solr中。

因为访问的延迟较高，这些索引是"异步更新"的，查询也只能做到"最终一致"（用户在觉察到不一致的时候，需要有特殊处理）。

在使用FDB和Reocrd Layer以后，索引 和数据是事务更新的，所以不再有上面的问题。

好处：
1. 所有更新都是 事务地（能够保证一致性的视图）；
2. 没有后台线程 去进行 索引的更新和删除；
3. 因为在同一个系统中（不依赖外部系统），所以可以减少运维成本；

功能：
1. 支持 前缀匹配；
2. 支持 n-gram搜索（而且是O(n)的，传统的方法是O(n^2)的。
3. 支持 近似度搜索；
4. 支持短语搜索；

跨集群迁移数据的问题：version是不相关的。

为了更好的负载均衡和数据分布，CloudKit定期的迁移用户数据

问题： 这种跨集群的数据迁移，能够对 "灰度上线"带来好处?

"逻辑数据库"总数：用户数 * 应用数；

问题：metadata store的内容？zone name的格式？

zone是CloudKit中的概念，代表一个可以（在多台设备）同步的单位（对应一个Reocrd Store中的一个PB结构）

最早的时候，CloudKit使用Cassandra作为底层的存储引擎；

有2个扩展性问题：
1. 同一个zone内的修改是串行的。即使修改的是不同的记录；
2. 原子修改多行记录，会映射到Cassandra的一个分区，容量是有限的；

随着容量增大会变差，Cassandra的性能变差。

对于需要协作的app，即用户需要可以在多个设备上同时修改同一个app的数据。因为在同一个zone内不允许并发，所在在开发这种程序时，就需要非常多的额外工作。

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 by version.

When deploying this implementation, we needed to handle previously stored data with an associated update-counter value but no version. Instead of adding business logic to combine the old and new sync indexes, we used the function key expression (see Section 6.1) 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 of them before records written with the new method.

8.2 Client Resource Isolation

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 requirements to service a given request are strictly fixed to little more than a single record ("row") 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, such as total records or bytes read while servicing a request. When one of these limits has been reached, the current state of the operation is captured and returned to the client in the form of a Record Layer continuation. The client may then re-submit the operation with the continuation to resume the operation. If even these operations become too frequent and burdensome on the system, other CloudKit throttling mechanisms kick in, slowing the rate at which the clients make requests to the server. With these limits, continuations, and throttling, we ensure that all clients make some progress even when the system comes under stress. CloudKit uses the same throttling mechanism when reacting to stress indicators coming from FoundationDB.

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 [12, 16, 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 a richer feature-set and semantics [5, 7, 9, 11, 19, 29]. FoundationDB [9] takes a unique approach in the NewSQL space: it is highly scalable and provides ACID transactions, but 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, such as the Record Layer, provide higher-level abstractions.

Multiple systems implement transactions on top of underlying NoSQL stores [8, 17, 22, 24, 34, 35, 41]. The Record Layer makes use of transactions exposed by FoundationDB to implement structured storage, complete with secondary indexes, queries, and other functionality.

The Record Layer has a unique first-class support for multi-tenancy. Without this support, most systems have to retrofit it, which is extremely challenging. Salesforce's architecture [18] is similarly motivated by the need to support multi-tenancy within the database. For example, all data and metadata is 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 facilitating scalability, and its key record store abstraction. For example, CloudKit faces a dual multi-tenancy challenge as it needs to service 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 database nodes and moving record stores to balance the load and improve locality.

目前，不支持 需要内存计算的操作：比如 join, group by, 排序

这些操作，必须借助于定义相应的索引来完成。

比如：如果希望进行 join，那么可以通过在相应的record type 都有的字段上，建立一个索引（这样，只需要 range方式迭代索引，就是join的结果）

好处：
这种方式，把查询所需要的内存限制在 仅仅比普通的单条查询 多一点点。

坏处：
1. 因为使用索引，在更新时需要事务的修改多条记录，所以会有一些负担。
2. 为了实现一个查询，可能会需要无限数量的IO。

解决方法（可能的无限量的IO）：
1. 利用Record Layer的能力去加以限制（比如：限制结果的总数量或总大小），当超过限制时，就保存下当前的状态，并返回一个"可重启点"，后面client在发送请求时，带着这个"可重启点"，就可以继续上次的操作；
2. 如果这种"重启"操作非常频繁 造成系统负担，那么CloudKit会启动限流机制：限制client发送请求的速度；

通过"可重启点"和"阈值"限制，能够保证在非常大的压力下，client依然可以做一些工作（性能受限，但仍可工作）

对于其它情况引起的 FDB压力较大情况，CloudKit使用同样的 "熔断"策略；

传统数据库的扩展性：
1. 可扩展读；
2. 但在设计上就不能高效的处理 分布式事务；

在shard-nothing架构的数据库中，跨分片的事务和索引 都是非常昂贵的；并且需要仔细的数据分区，这对于复杂的应用程序来说，是非常复杂的。
这导致了有 很多对于"自动分区"的研究；

对于shard-disk架构，非常难扩展。原因是：
1. 保证cache一致性代价很高；
2. page的竞争协议

NoSQL的特点：
1. 非常好的扩展性；
2. 语义较少（大部分都是只提供K-V API，没有schema，没有索引，没有事务，没有查询引擎（可能是指除了点查询之外的查询方式））。

FDB 不是一个"one-size-fits-all"的解决方案；

FDB被设计为：为上层的Layer提供基础设施；

其它很多系统在NoSQL上自己实现事务（因为普通的NoSQL不支持事务），但是Record Layer可以直接使用FDB的事务。

Salesforce的架构，同样是受到了"需要在数据库中实现多租户"的推动。
注意：Record Layer中多租户的实现方式，和Salesforce并不相同。

1. 所有的数据 和元数据 都按照"应用程序"进行分片；
2. "查询优化器"要考虑到 '每个程序和用户' 级别的统计数据；

CloudKit中"多租户"的双重挑战：
1. 支持 很多 APP；
2. 每个APP有很多用户；

>> 在Salesforce中，用户数量不多；

Record Layer在"多租户"功能上走的更深：
1. 通过 内置的 资源跟踪 和 隔离；
2. 完全无状态的设计（可促进 扩展性）；
3. 抽象出来 key record store；

每个Record Store都封装了 一个用户 在一个APP中 的所有数据（包括索引和元数据）。

通过这种组织形式，可以很容易的扩展用户（支持 十亿 级别的用户数）：只需要简单的增加机器，并（根据 负载均衡 和 数据分布位置）迁移 record store。

While many storage systems include support for full-text indexing and search, most provide this support using a separate system [15, 18, 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 queries (of any kind of index) are not guaranteed to be consistent, i.e., they are not guaranteed to return all matching documents [6].

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 separation of logical and physical query plans, 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 B).

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 scalable, robust, but semantically simple datastore upon which complex abstractions are built. This allows systems architects to pick and 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 currently 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 developers of new FoundationDB layers and Record Layer adopters.

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 FoundationDB Java binding completed Java Futures in this 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 minimimze 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 were concurrently modified by another. The FoundationDB API gives full control over these potentially overlapping read- and write-conflict sets. In particular, it allows for manually adding read conflicts. 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. This way, the transaction depends only on what would invalidate its results. This is done, for instance, in navigating the skip list used for rank / select indexing, described in Appendix A. Bugs due to incorrect manual conflict ranges are naturally hard to find and made even harder to find when mixed-in with business logic. For that reason, it is important to define Record Layer abstractions, such as indexes, for such patterns, rather than relying on indiviudal 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 sightly 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 who do not need this shared extent, we

全文索引和全文检索的支持：
1. 大部分系统都是通过外部系统（比如Solr）来实现，只能做到"最终一致性"；

MongoDB内置了"全文检索"，但是因为它不支持分布式事务，所以当使用索引进行检索的时候，只能保证"最终一致"。

正在开发 基于 瀑布模型 的 查询优化器。

FDB是为"吞吐量"而优化的，不是为了 单个操作的延迟；

FDB的client是单线程的，有一个专门的网络线程；

注意：要尽量减少 "网络线程"的工作量；

在之前版本的FDB java client lib中，在网络线程中执行 Java Future对象的结束操作（可能是要发一个网络包），造成了这个"网络线程"成为瓶颈。

1. FDB的API可以充分控制各种"读-"和"写-"冲突；
2. 设置可以 手动增加"读冲突"；

一种使用"手动增加读冲突"的模式：
1. 先增加一些读冲突；
2. 做一个'无冲突读（快照读）'去读一个range'：这个range可能会包含"之前增加读冲突的key"。

这样，这个事务，只有在 其它事务也同时更新了"手动增加读冲突"的key时，才会失败；而在这个range中，没有增加读冲突的key，是不会影响事务成功提交的。

事务通过 搜索skiplist来判断是否有冲突；

追查 由于 "用户错误的 手动控制 冲突区间"造成的问题是非常难的，尤其是 和业务逻辑结合在一起的场景下；
所以，"定义一种 记录层 的抽象"是非常重要的（这样，在追查问题时，就可以剥离用户的业务逻辑）。

PB提供了 获取 响应结构的 "静态描述对象"的方法。

可以 在代码中 使用这些方法去 创建Record Layer的元信息，但不推荐这样做，而是建议将metatadata保存在metadata store中。
原因：
1. 无法原子的更新多台 Record Layer实例所使用的元信息；
2. 很难去检查schema的修改是否违反了约束条件；

在一个record store中，所以 类型 的record是交叉存储的；

所以：读取某一个类型的record，需要：
1. 全表扫描这个record store(包含其他 type的record)；
2. 维护一个 二级索引；

如果不希望这种共享，目前支持为每个类型的record添加一个不同的前缀，从而模拟出是各自独立的区域；

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

将“多租户”功能，添加到一个 已有的系统，是非常难的。

所以，在Record Layer中，在设计时就考虑了对于“多租户”的支持

收益：
1. 更容易的 集群间分片均衡；
2. 弹性伸缩；

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, groups of users, different application functions, or some other entity.

11 FUTURE DIRECTIONS

The current state of the Record Layer stems greatly from the immediate needs of CloudKit: that is, the ability to support billions of small databases, each database having few users, and all within a carefully controlled set of resources. As databases grow, both in terms of data volume and in terms of number of concurrent users, the Record Layer may need to adapt and layers will be developed on top expanding its functionality to support these new workloads and more complex query capabilities. It is our goal, however, that the layer will always retain its ability to support lightweight and efficient deployments. We highlight several future directions:

随着用户的增多，每次访问时都去检查metadata是否已经修改，如果这些metadata都在一台机器上，那么它会成为热点。

在keyspace的不同点进行replicate，从而增大 副本在不同存储节点上的可能性。

在一些场景下，更新不频繁，也可以使用cache来缓解热点。

但是这种情况下，对应的cache
1. 需要被设置为无效；
2. 要么能够检测出来是否无效，或者能够容忍。

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 if all these requests go to the same storage node. 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.

即使使用“不太完美”的索引，但是在“流模型”中，依然是可行的。比如：基于优先级队列的小限制排序，或 有限大小的hash join；

>> 目前，只要索引不是完美的，都不能用来进行查询；

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.

但是，一些查询确实需要 完全支持密集型内存操作，甚至需要将中间结果落盘。

这种类型的操作，对 可扩展性 有非常大的挑战，因为它需要新的“资源跟踪”和“管理形式”，并且 在查询期间必须是有状态的。

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 is a record-oriented data store with rich features similar to those of a relational database, including structured schema, indexing, and declarative queries. Because it is built on FounationDB, 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 seamlessly add features outside of the core library, including custom index types, schema managers, and record serializers. At Apple, we leverage these capabilities to implement CloudKit, which hosts billions of databases. CloudKit 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).

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 a complex system like CloudKit. It is easy to envision other layers that extend the Record Layer to provide richer and higher-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.

3个关键的观察：
1. Record Layer的成功验证了FDB在大型系统中的分层体系结构；
2. Record Layer的可扩展设计提供了公共功能，同时很容易适应CloudKit等复杂系统的定制需求；将来很可能会有其他Layer来继续Record Layer，以提供更丰富和更高级别的功能。
3. 将应用程序组织成逻辑“租户”（可能是用户、特性或其他实体），是构造系统和扩展系统以满足需求的一种强大且实用的方法。

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] Apache Cassandra, 2018.

[5] Azure Cosmos DB. https://azure.microsoft.com/en-us/services/cosmos-db/, 2018.

[6] Blog: MongoDB queries don't always return all matching documents! https://blog.meteor.com/mongodb-queries-dont-always-return-all-matching-documents-654b6594a827, 2018.

[7] Cockroach Labs. https://www.cockroachlabs.com/, 2018.

[8] CockroachDB. https://www.cockroachlabs.com/, 2018.

[9] FoundationDB. https://www.foundationdb.org, 2018.

[10] FoundationDB on GitHub. https://github.com/apple/foundationdb, 2018.

[11] MemSQL. https://www.memsql.com/, 2018.

[12] MongoDB. https://www.mongodb.com, 2018.

[13] Protocol Buffers. https://developers.google.com/protocol-buffers/, 2018.

[14] Protocol Buffers: Specifying Field Rules. https://developers.google.com/protocol-buffers/docs/proto#specifying-field-rules, 2018.

[15] Riak: Complex Query Support. http://basho.com/products/riak-kv/complex-query-support, 2018.

[16] Riak KV. http://basho.com/products/riak-kv, 2018.

[17] Tephra: Transactions for Apache HBase. https://tephra.io, 2018.

[18] The Force.com Multitenant Architecture. http://www.developerforce.com/media/ForcedotcomBookLibrary/Force.com_Multitenancy_WP_101508.pdf, 2018.

[19] VoltDB. https://www.voltdb.com/, 2018.

[20] Announcing the FoundationDB Record Layer. https://www.foundationdb.org/blog/announcing-record-layer/, 2019.

[21] FoundationDB Record Layer on GitHub. https://github.com/foundationdb/fdb-record-layer, 2019.

[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. Mwaura, 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/2701/2701-0.txt.

[40] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, 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 *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 337–348, New York, NY, USA, 2014. ACM.

RANK index作用：
1. 根据 排名（根据一些key expr） 找到 对应的记录；
2. 反过来，给定记录，确定它的排名；

例子1：排行榜程序中：给定分数，快速确定他的排名；

注意：之所以每个节点上记录的数量值，是该节点在全部集合中大于该节点的数（另一个条件是在当前层级中与下一个节点的距离），而不是小于该节点的数，是因为这样：在插入节点的时候，只需要把每个查找路线上的节点的 计数都加1即可；如果是记录“小于该其值的节点数”，那么在插入节点时，就需要将在所有层级上、所有的大于该其值的节点中 所记录的 节点数都修改，这样代价就高了。

例子2：滚动条（翻页）：记录是排过序的，用户希望跳过某些结果，直接到达指定的某个序号处，然后开始顺序扫描。
方法一：
使用VALUE index，先顺序扫描到第K处，然后将它作为一个 cursor，从而如果中断的时候可以重新扫描；

方法二：
使用 RANK index，更高效；

Preprint. Latest version at
https://foundationdb.org/files/record-layer-paper.pdf

A RANK AND TEXT INDEX TYPES

RANK *indexes*. The RANK index type allows clients to efficiently find records by their ordinal (顺序的，依次的) 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 (排行榜), finding a player's position in the leaderboard could be implemented by looking-up their score's rank using a RANK index. Another example is an implementation of a scrollbar (滚动条), 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. One way to implement this could be to linearly scan a VALUE index until we get to the k-th result, and use Record Layer's cursors to efficiently restart the scan if it is interrupted. An implementation using a RANK index can be much more efficient and query for the record that has rank k, then begin scanning from that position in the index.

RANK index是基于“排名集(ranked set)”实现的，底层数据结构是skip-list，持久化存储在FDB中。

Our implementation of the RANK index stores each index entry in a durable data structure that we call a *ranked set*: a probabilistic (概率性的) 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, in the same transaction. 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 4(a) includes a sample index representing a skip-list with six elements and three levels.

1. 每个层级都有一个不同的前缀；
2. 在同一个事务中，在插入前会先尝试读取，从而避免插入重复的key；
3. 最底层 是所有index条目；每个更高层级 都包含下一层的条目样本。
4. 每个节点上都包含一个数量：a. 在整个集合中大于或等于它的节点数量；b. 在当前level上与下一个节点的距离；（最底层的节点，该数量值为1）
5. 注意：不需要显式的“指针”，因为利用 FDB保存数据的有序性，更高效。

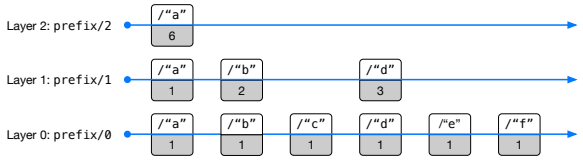
节点上保存的数量值：就是在搜索过程中，从当前节点到下一个节点 需要跳过的节点数量。

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 4(b). The final sum represents the rank. Given a rank, a similar algorithm is used to determine the corresponding index entry. A cumulative (累积的) sum is maintained and a range scan is performed at each level until following a finger would cause the sum to exceeds the target rank, at which point the next level is scanned.

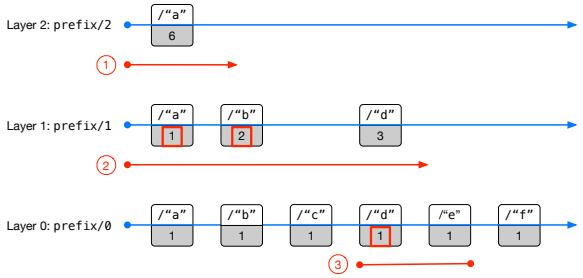
查找1：给定记录去查找它的排名（最低的排名是0）：
1. 从最顶层开始查找；
2. 当 需要在同一层级上从前往后 跳转时，累加前一个节点上记录的数值；
3. 最底层的节点，累加1；
4. 最后累加的数值，就是 该记录的排名。

查找2：给定排名找相应的记录：
1. 也是维持一个累加和，并在每层进行range scan；
2. 如果累加一个节点上记录的值，会超过当前的“目标排名”，那么就开始下一层的扫描；

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



(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 particular range of the keyspace are shown with red arrows. (1) Scan the prefix/2 subspace and find only a, which comes before e. (2) Scan 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 4: Example of index entries representing a skip-list with 6 elements and 3 levels

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:

$$(\text{prefix}, \text{token}_1, \text{pk}_1) \rightarrow \text{offsets}_1$$
$$(\text{prefix}, \text{token}_1, \text{pk}_2) \rightarrow \text{offsets}_2$$
$$(\text{prefix}, \text{token}_2\ \ \text{pk}_3) \rightarrow \text{offsets}_3$$

$(prefix, token_3, pk_4) \rightarrow offsets_4$
$(prefix, token_3, pk_5) \rightarrow offsets_5$
$(prefix, token_3, pk_6) \rightarrow offsets_6$

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.

$(prefix, token_1, pk_1) \rightarrow [offsets_1, pk_2, offsets_2]$
$(prefix, token_2, pk_3) \rightarrow [offsets_3]$
$(prefix, token_3, pk_4) \rightarrow [offsets_4, pk_5, offsets_5]$
$(prefix, token_3, pk_6) \rightarrow [offsets_6]$

To insert a token t and primary key pk the index maintainer performs a range scan and finds the biggest key L that is less or equal to `(prefix,t,pk)` and the smallest key R bigger than `(prefix,t,pk)`. It then places a new entry in the bunch corresponding to L, but only 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 the index maintainer makes it 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. In order 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 index entry is simply deleted. Otherwise, the entry is updated such that `pk` and its list of 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 FoundationDB key-value pairs and writing at most two, though usually only one, key-value pair. Deleting an entry always requires reading and writing a single FoundationDB key-value pair. This access locality makes index updates use predictable resources and have predictable latencies. However, it is easy to see that there are certain write patterns that 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 execute bunch compactions.

Numerical Example. To demonstrate the benefit of this optimization, we used Herman Melville's Moby Dick [39], broken up by line into 233 roughly equal 5 kilobytes documents of raw text. With whitespace tokenization, each document contains ~431.8 unique tokens (so the primary key can be represented using 2 bytes) with the average length of ~7.8 characters (encoded with 1 byte per character), appearing an average of ~2.1 times within the document. There are approximately 2 bytes of overhead within the key to encode each of the token and primary key. For the calculation, we use 10 bytes as the prefix size (much smaller than the typical size we use in production). The value is encoded with 1 byte per offset plus 1 byte for each list. In total, the index requires 21.8 key bytes and 3 value bytes per token, or ~10.7 kilobytes per document. For each byte in the original text, we write 2.14 bytes in the index.

With a bunch size of 20, each key still includes 19.8 bytes for the prefix (10 bytes), token (7.8 bytes) and encoding overhead (2 bytes), as well as up to 20 primary keys and values (representing individual documents containing the token), each of size 2 and 3 bytes, respectively (100 bytes total), i.e., ~6 amortized bytes per token per document (~1 for key and 5 for the value). Multiplied by the number of tokens, the index size is 2.6 kilobytes per document – roughly half of the original document size. When actually measured, the index required ~4.9 kilobytes per document, almost as much as the document text itself. The reason for the discrepancy is that not every bunch is actually filled. In fact, the average bunch size was ~4.7, significantly lower than the maximum possible; in fact, some words appear only once in the entirety of Moby Dick and therefore will be necessarily be given their own bunch. To optimize further, we are considering bunching across tokens, and implementing prefix compression in FoundationDB. Note that even then, there is per-key overhead in the index as well as in FoundationDB's internal B-tree data structure, so reducing the number of keys on the Record Layer level is still beneficial.

B QUERY PLANNING AND API

Often, clients of the Record Layer want to search, sort, and filter the records in a record store in order to retrieve specific information. Instead of forcing clients to manually inspect indexes, 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 in order to meet its goals. In this section, we present the query interface exposed by the Record Layer, the architecture of its extensible query planner, and the reasoning behind key design decisions.

Extensible query API. The Record Layer has a fluent, declarative Java API for querying the database. This query API can be used to specify 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 Section 6.1). Both the filter and sort expressions can include "special functions" on the record set, including aggregates,

cardinal rank, and several full-text search operations such as *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 a first class citizen, allowing consumers to directly interact with it in Java. Another layer on top of the Record Layer could provide translation from SQL or a related query language.

Query plans. While declarative queries are convenient for clients, they need to be transformed into concrete operations on the record store, such as index scans, union operations, and filters in order to execute the queries efficiently. For example, the query plan in Figure 5 implements the query as a union of two index scans that produce streams of records in the appropriate order. The Record Layer's query planner is responsible for converting a declarative query—which specifies what records are to be returned but not how they are to be retrieved—into an efficient combination of operations that map directly to manipulations of 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]. In the same fashion as SQL PREPARE statements, Record Layer queries (and thus, query plans) may have bound static arguments (SARGS). In CloudKit we have leveraged this functionality to implement certain CloudKit-specific planning behavior by combining multiple plans produced by the Record Layer and binding the output of an "outer" plan as input values to an "inner" plan.

We are currently evolving the Record Layer's planner from an ad-hoc architecture to a Cascades-style rule-based planner. Our new planner design supports deep extensibility, including custom planning logic defined completely outside the Record Layer by clients, and has an architecture that provides a path to cost-based optimization.

```
RecordQuery.builder().setSort(field("age"))
  .setRecordType("Employee").setFilter(
    Query.or(field("age").greaterThan("30"),
             field("address").nest(
               field("country").equals("US"))))
```

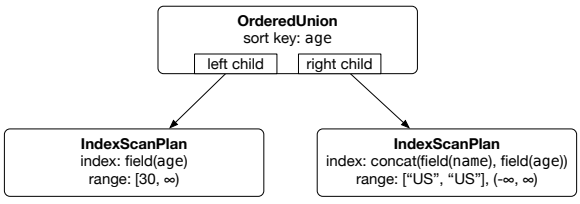


Figure 5: An example query and the corresponding plan structure. The two index scans produce streams of records ordered by the field "age" so they can be combined with an `OrderedUnion` plan.

Cascades-style planner. The new planner architecture uses the top-down optimization framework presented in Cascades [36]. Internally, we maintain a tree-structured intermediate representation called an expression of partially-planned queries that includes 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 expression tree, optionally inspect their properties, and then produce equivalent expressions. An example of a simple rule that converts a logical filter into a scan over the appropriate index range is shown in Figure 5.

Rules are automatically selected by the planner, but can be organized into "phases" based on their utility; for example, we prefer to scan a relevant index rather than scan all records and filter them after deserialization. They are also meant to be modular; several planner behaviors are implemented by multiple rules acting in concert. While this modular architecture make the code base easier to understand, its primary benefit is allowing more complicated planning behavior by mixing-and-matching the available rules.

The rule-based architecture of the planner allows clients, who may have defined custom query functions and indexes, to plug in rules for implementing that custom functionality. For example, a client could implement an index that supports geospatial queries and extend the query API with custom functions for issuing bounding box queries. By writing rules that can turn geospatial queries into operations on a geospatial index, the client could have the existing planner plan those queries while making use of all of the existing rules.

Future directions. In designing the intermediate representation and rule execution system in the Record Layer's experimental planner, we have tried to anticipate future needs. For example, the data structure used by the "expression" intermediate representation currently stores only a single expression at any time: in effect, this planner currently operates by repeatedly rewriting the intermediate representation each time a rule is applied. However, it is designed to be seamlessly replaced by the compact Memo data structure [36] which allows the planner to succinctly represent a huge space of possible expressions. At its core, the Memo structure replaces each node in the expression tree with a group of logically equivalent expressions. Each group can then be treated as an optimization target, with each expression in the group representing possible implementations of that group's logical operation. With this data structure, optimization work for a small part of the query can be shared (or memoized, as the name suggests) across any possible expressions. Adding the Memo structure paves the way to a cost-based optimizer, which uses estimates of the cost of different possibilities to choose from several possible plans.