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

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

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

C QUERY PLANNING AND API

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

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

Query plans. While declarative queries are convenient for clients, they need to be transformed into concrete operations such as index scans, union operations, and filters in order to execute the queries efficiently. The Record Layer's query planner converts a declarative query specifying the records to return into an efficient combination of operations on the stream of records. The Record Layer exposes these query plans through the planner's API, allowing its clients to cache or otherwise manipulate query plans directly. This provides functionality similar to that of a SQL PREPARE statement but with the additional benefit of allowing the client to modify the plan if necessary [45]. Like SQL PREPARE statements, queries (and thus, query plans) may have bound static arguments (SARGS). In CloudKit we have leveraged this functionality to implement CloudKit-specific planning behavior by combining multiple plans produced by the Record Layer and binding the output of one plan to an argument of another.

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

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

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