Progressive Partitioning for Parallelized Query Execution in Google's Napa

Junichi Tatemura
Google Inc

Tao Zou
Google Inc

Jagan Sankaranarayanan
Google Inc

Yanlai Huang
Google Inc

Jim Chen
Google Inc

Yupu Zhang
Google Inc

Kevin Lai
Google Inc

Hao Zhang
Google Inc

Gokul Nath Babu Manoharan
Google Inc

Goetz Graefe
Google Inc

Divyakant Agrawal
Google Inc

Brad Adelberg
Google Inc

Shilpa Kolhar
Google Inc

Indrajit Roy
Google Inc

napa-paper@google.com

ABSTRACT

Napa holds Google's critical data warehouses in log-structured merge trees for real-time data ingestion and sub-second response for billions of queries per day. These queries are often multi-key look-ups in highly skewed tables and indexes.

In our production experience, only progressive query-specific partitioning can achieve Napa's strict query latency SLOs. Here we advocate good-enough partitioning that keeps the per-query partitioning time low without risking uneven work distribution. Our design combines pragmatic system choices and algorithmic innovations. For instance, B-trees are augmented with statistics of key distributions, thus serving the dual purpose of aiding lookups and partitioning. Furthermore, progressive partitioning is designed to be "good enough" thereby balancing partitioning time with performance. The resulting system is robust and successfully serves day-in-day-out billions of queries with very high quality of service forming a core infrastructure at Google.

PVLDB Reference Format:
Junichi Tatemura, Tao Zou, Jagan Sankaranarayanan, Yanlai Huang, Jim Chen, Yupu Zhang, Kevin Lai, Hao Zhang, Gokul Nath Babu Manoharan, Goetz Graefe, Divyakant Agrawal, Brad Adelberg, Shilpa Kolhar, and Indrajit Roy. Progressive Partitioning for Parallelized Query Execution in Google's Napa. PVLDB, 16(12): 3475-3487, 2023.
doi:10.14778/3611540.3611541

1 INTRODUCTION

Napa powers Google-wide data warehouse needs [9]. It ingests petabytes of data per-day and serves billions of queries with sub-second latency. Napa has hundreds of databases with thousands of tables and materialized views; many of those are multi-petabytes in size. Napa tables are continuously updated by a massive planetary-scale stream of writes modifying existing rows or adding new rows. Users require the query results to be consistent and fresh, and demand continuous availability in the presence of data center failures or network partitions.

Napa serves business critical clients who expect strict service level objectives (SLOs) on their sub-second query response time. To support the competing demands of high throughput ingestion and low-latency querying, Napa implements a fairly standard distributed table and view maintenance framework that is based on the LSM-tree (Log-Structured Merge Tree) paradigm [18]. LSM is widely used in the current generation of data warehouses to efficiently integrate and incorporate streaming updates to existing data. Napa scales LSM to meet the challenges of Google's operating environment, which includes ingesting trillions of rows daily.

```
SELECT K1, K2, SUM(Val)
    ↪ FROM t(K1, K2, K3)
    ↪ WHERE K1 in (1, 7, ..., x )
    ↪ AND K2 in (10, 20, ..., y)
    ↪ GROUP BY K1, K2;
```

Figure 1: An example many-key look up query that is representative of the Napa query workloads.

Napa's diverse query workload consists of large scans and many-key lookups. The analytical queries with many-key lookups have strict QoS requirements and is the main focus of this paper. Figure 1 is an example representative of a many-key lookup query. The Napa table *t* has K1, K2, and K3 as primary keys and furthermore is also sorted and indexed by their primary keys. The query specifies a lookup on two of the prefix keys K1 and K2. The goal here is to break up the key space of, possibly, a petabyte table into

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611541

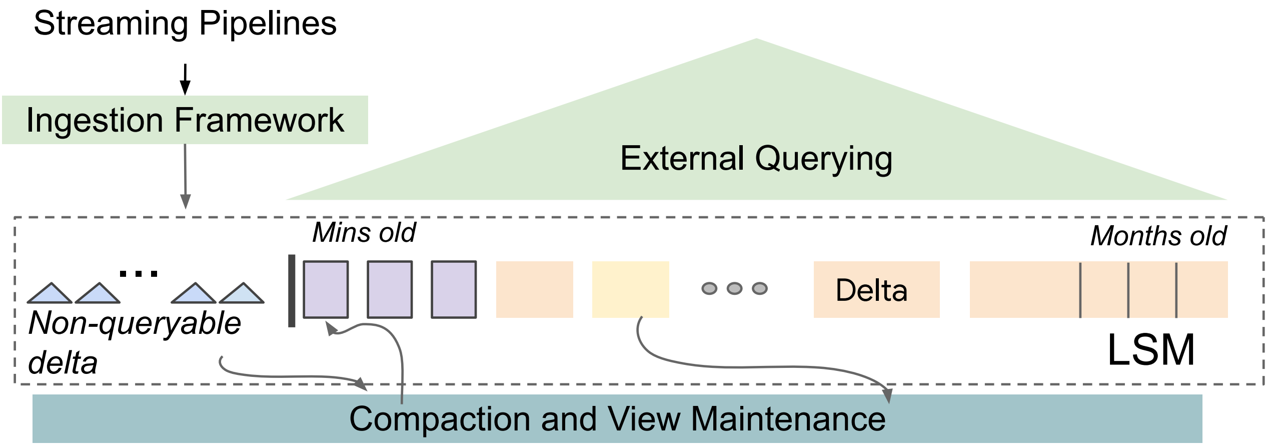

Figure 2: The Napa architecture

fairly evenly sized partitions such that the query can be parallelized and has low query latency. From our experience in running production services, we found the following three requirements important:

↪ **[Query-specific partitioning]** Any approach that we take should meet the latency SLOs across a diverse set of query workloads. Parallelizing the query execution via partitioning is often required to meet the strict latency SLO. In our experience, the partitioning granularity needs to be adjusted on a per-query basis to meet the latency and resource budget requirements. The expectation from the partitioning step is that it is able to produce number of partitions denoted by the parallelization requirement within bounded amount of error.

↪ **[Evenness]** Execution involves the partitioning of the tables into key ranges such that the partitioning results in even partitions. Partitioning should operate on tables with extreme skews where some keys span terabytes in disk. As an example, in Figure 1, a key range like $< K1 = 1, K2 = 20 >$ may correspond to, say, a hundred GB portion of the table while another key range may be considerably smaller at a few MB.

↪ **[Progressiveness]** Query partitioning must balance overall execution time against the time and effort to produce query-specific partitions. For instance, one can produce perfectly even partitions while still ending up missing the QoS requirement. It is imperative that the partitioning method has the notion of "good enough" in the sense that it stops when the partitioning is of sufficient quality. Our proposed technique in the paper is progressive such that 1) the longer the algorithm runs the better the quality of the resultant partitioning; 2) the algorithm stops once the desired error bound has been met.

Numerous techniques have been proposed in the literature that leverage the performance improvements resulting from database partitioning [1, 6, 17, 19–21, 25]. Much of the published literature focuses on write-time organization of the database and tables as a way to speed up query performance. Here good performance is a fortuitous alignment of write-time partitioning granularity and query parallelization requirements. A table is split at write time into many units using hash or range partitioning on keys. If the unit is large, it is not conducive for queries that may require fine grained partitioning to achieve low latency. If the unit is small, the metadata book-keeping overhead is high, and processing the metadata alone imposes challenges to low latency; also, scan query's performance suffers due to per-unit overhead. Our experimental results show that the use of statistics that is too fine-grained can often result in queries spending too much time generating partitioning at the expense of overall query execution time.

As noted, by using fine-grained statistical distribution of the data at the storage level, one can produce even query-specific partitioning albeit at a high cost. We address this dilemma by leveraging B-trees to optimize access to the statistical information on the tables. Each LSM run has an associated B-tree index which are enhanced so that index nodes maintain size information for the associated key ranges. With these enhancements, we can estimate the input data size of the query starting with the root of the B-tree. If this estimation is not accurate enough or if the statistics point to areas of skews, we can descend to the next level in the index structure to obtain a finer level of statistical distributions for the key ranges overlapping with the query.

Our proposed algorithm traverses the B-tree to produce even and query-specific partitioning. It is progressive in the sense that it descends and accesses additional index information only if it does not satisfy the stipulated error bounds. In addition, the refinement is selective that it only descends to the lower levels of trees for those partitions that do not have tight enough bounds on the error. Effectively, this means that if we want to create 3 partitions (i.e., 2 split points) for a query range, the first split point may be found at the root itself but for the second split point may require us to descend to the level below the root. This is the essence of our overall approach for progressive query-specific partitioning. The complexities that need to be addressed is that we have to identify these split points across many LSM runs or equivalently by combining information from multiple B-trees of varying sizes. These are described later in the paper.

The rest of the paper is organized as follows. Section 2 provides related work. Section 3 provides a high-level overview of the Napa system while the data and query models are explained in Section 4.

Our progressive algorithm is provided in Section 5 and experiment results are presented in Section 6. Concluding remarks are drawn in Section 7.

2 RELATED WORK

Parallel Databases and parallel execution of SQL operators has had a long history in database research [3, 11]. Due to the set-oriented semantics of SQL execution, the key idea to parallelize the execution of SQL queries is to partition the input data and use SIMD (single instruction multiple data) model of computation for parallel query execution. The early work on parallel databases and database machines [4] based on dataflow architectures were all predicated on this paradigm. With the emergence of large data centers and cloud computing as well as the massive scale of data in most contemporary applications there is an emerging need to leverage this classical paradigm of data partitioning (also referred to as data sharding) in the context of modern data infrastructures. In the early years of scalable analysis of internet application data companies such as Google and others discarded database technologies and instead developed large-scale analysis infrastructures that were based on the map-reduce paradigm [8]. The key principle underlying the map-reduce paradigm was to partition the input datasets over a large number of machines and process them in parallel to reduce the latency of analysis queries significantly.

As database research and practice started to embrace cloud computing, the prevalent approach that is used is to partition the data or to split a table on some key, using hash or range partitioning. Thus queries that have selection predicates involving the key can be processed by only accessing the relevant portions of data. In the same vein, an ideal data partitioning mechanism can enable efficient collocated join, and minimize the number of distributed transactions. Due to these performance improvements resulting from database partitioning, numerous techniques have been proposed in the literature [1, 6, 17, 19–21, 25]. Optimizing data partitioning based on query workloads has been explored in commercial databases such as IBM DB2 [21, 25] and Microsoft SQL Server [1, 17]. For OLTP workloads, graph-based [6, 20] and skew-aware [19] partitioning approaches have been developed to minimize the number of distributed transactions. We note however that all of these approaches are focused on finding the ideal data partitioning solutions at the write time based on given query workloads. That is, most of the focus is to ensure that the data is partitioned appropriately when it is stored in the database with the hope that a large percentage of the query workload will be aligned with the data partitioning mechanism and granularity determined at write-time.

To the best of our knowledge, there are no efforts to partition the data in a way that is optimized for individual query's latency and resource budget requirements. The closest work is what is referred to as *database cracking* [12, 14, 15, 22, 23]. In database cracking, the data layout is dynamically rearranged based on the input query workload. The idea of adaptively improving partitioning online has been extended to distributed database settings [24]. These approaches, although can adjust to aggregated workload changes over time, are not designed to optimally handle a wide spectrum of concurrent query workloads. Recently, there are research initiatives to use machine learning to better guide the process of data partitioning for databases stored in the cloud [13]. In this work, the authors introduce a new learned partitioning advisor based on Deep Reinforcement Learning (DRL) for OLAP-style workloads. However, as is the case with prior work in data partitioning, the focus of this work is also in the context of write-time partitioning.

3 NAPA PRELIMINARIES

Napa [9] stores and manages many petabytes of historical data that is continuously ingested from a diverse set of internet-scale applications. Napa was designed to be used Google-wide and to serve the diverse needs of many analytical applications. Many of our front-end clients issue complex analytical queries over petabyte scale tables and expect answers in sub-seconds. The following key aspects of Napa are the bedrock principles of its design and are aligned with our client requirements: (i) Robust Query Performance: Napa clients expect low query latency, typically sub-seconds, as well as low variance in latency under a wide spectrum of query and data ingestion load; (ii) System Flexibility: While performance is important, our clients also require the flexibility to change system configurations to their dynamic requirements such as trading freshness or recency of ingested data for better performance; (iii) High-throughput Data Ingestion: Napa's ingestion, storage and query serving functions perform under a massive update load.

Napa's system [9] diagram shown in Figure 2 consists of an ingestion infrastructure, a table and materialized view maintenance framework and a query serving infrastructure. There is also a controller component that is responsible for the overall management of the entire system. Napa's tables and materialized views are organized as LSMs and the goal of the system is to ensure that the LSM is maintained against the latency, cost and freshness demands specified by the database owners.

A Napa table consists of multiple data sets called deltas (corresponding to sorted runs of an LSM-tree) and we have one B-tree index for each delta. Note that each delta corresponds to updates on a table during a time window (e.g., last 1 minute, 1 day etc.). Napa tables have multi-part keys and aggregate values. Queries often run lookups on some subset of keys. Note that while the deltas are non-overlapping temporally based on ingestion time, they are overlapping in the key space. In other words, a query specifying a lookup key may find qualifying rows in all the deltas that make up the table span.

↪ **Progressive partitioning using B-trees:** The B-trees on the deltas are fairly generic except that it hierarchically stores statistics of the underlying data. In particular, we store the number of rows that is indexed by each sub-tree and aggregate that statistics up the level. The partitioning algorithm we propose in the paper takes queried keys as input, retrieves and merges relevant keys taken from these B-tree indices, to generate an approximate histogram. Starting with the highest level histograms (i.e., the B-tree root index blocks), it tries to find accurate enough partitions. If it needs more detailed information, the algorithm will retrieve required index blocks from the next highest level, repeating this trial and retrieve until it reaches the desired accuracy.

4 DATA AND QUERY MODEL

4.1 LSM Data Model

To support efficient snapshot reads over large scale data sets under high-throughput ingestion, we employ log-structured merge (LSM) trees [16, 18]. An LSM tree is an append-only data structure where new data ingestion is stored separately from the old data. To maintain read efficiency and reduce storage costs, an operation called compaction is periodically applied to merge new and old data.

LSM consists of immutable files stored in a distributed file system, each containing data updates for a specific timestamp range. The data in each file is sorted and indexed by a hierarchical B-tree index. Such an immutable file along with its associated indexes is called a *delta*. For compaction, we employ the tiered merging policy [16]: Ingested data is committed as a base level delta with a singleton time range (e.g., [T1, T1]), and a compaction reads multiple deltas at one level (e.g., [T1, T2], [T3, T4], [T5, T6]) and writes a new delta (e.g., [T1, T6]) at the next level. We leverage this tiered delta structure to support multiversion snapshot reads: For a query over a snapshot at time T, the system finds the smallest set of deltas that contain all the updates in [0, T] without any duplicates.

When the table has a primary (unique) key, we need to identify the value of the row at time T while the updates of the same key can appear in multiple deltas in [0, T]. These rows are *versions* or *updates* of the same key, and should be merged into a single final row during read time. The merging can be done in different ways and the solution in this paper does not assume a particular merging method. One frequently used way of merging is to take the most recent update among the occurrences.

↪ **Progressive partitioning & LSM:** The LSM data model complicates the task of query-specific partitioning in the following ways. First, the keys specified in the query may be present in many (possibly all) of the deltas. Note that the same query worker must process all deltas where the keys are present in order to reconcile all relevant versions. This effectively means that the work per worker is the sum of the matched rows across the various deltas. Second, this proves to be a serious challenge for the evenness of the partitioning since some deltas may contribute many more matched rows than others. Third, a query may involve seeking across multiple deltas and not all deltas equally contribute to the query. Thus, the partitioning effort may not be uniform across the deltas. For some deltas, it may be sufficient to perform coarser partitioning while others require significantly more effort in producing equal splits.

4.2 Query Model

```
SELECT * FROM R
    ↪ WHERE CustomerId IN (c1, c2, c3) AND
    ↪ Date BETWEEN d1 and d2;
```

Figure 3: Sample query with keys specified.

While we support parallel execution of general SQL queries, the focus of this paper is parallelizing scans accessing the table data stored as deltas. A scan worker involves part of the distributed query plan that is executable locally in a single machine. Operations done by a scan worker include selection, projection and partial (local) aggregation. Typically, the output of scan workers will be consumed by other distributed downstream operations such as full aggregation, sort, and join. Parallelization of other operations are not directly associated with the stored data (deltas) and not the focus of this paper.

For selection, the scan worker seeks over sorted rows in the deltas using prefix keys referenced by the query predicates. As an example, consider the following query shown in Figure 3 with different selection conditions than shown in Figure 1.

Suppose table R holds measurements in multiple dimensions including CustomerId and Date. If the primary key (i.e. compound of dimensions) has sort order $\langle \text{CustomerId, Date}, \cdots \rangle$, we can use a prefix key $\langle \text{CustomerId, Date} \rangle = \{\langle c1, d1\rangle, \langle c1, d2\rangle, \cdots, \langle c3, d2\rangle\}$ to seek the relevant blocks in the deltas. In general, the scan worker takes a set of prefix key ranges. A prefix key range is a prefix of the table key where the last column of the prefix can be a range condition. The query example has such a range condition "Date BETWEEN $d1$ AND $d2$", so we will have 3 prefix key ranges $\{\langle c1, [d1, d2]\rangle, \langle c2, [d1, d2]\rangle, \langle c3, [d1, d2]\rangle\}$. Each prefix key range corresponds to a (potentially empty) contiguous row range in a delta sorted by the primary key. In the following example, we argue that the partitioning should be dynamically generated on a per query basis to meet the latency requirements across a spectrum of query workloads.

↪ **Why rely on progressive query-specific partitioning?** Consider a 1PB table. Typical queries have selective predicates, but with high variance in their selectivity (e.g., from 0.0001% to 1%). They also come with different resource budget which translates into different limits on the number of scan workers. Thus, the ideal partitioning unit size can vary from 10MB (e.g., when reading 1GB data using 100 scan workers) to 1GB (e.g., when reading 1TB data using 1000 scan workers) on a *per-query* basis. The next complexity here is that it is possible that there is one particular "Date = $d3$", which accounts for most of the data to be scanned in a query. So, that means that we need to partition the range $\langle c1, d3\rangle$, $\langle c2, d3\rangle$ and $\langle c3, d3\rangle$ much more finely than the other key ranges. This means the partitioning algorithm needs to be *progressive* such that it can stop early on other dates and focus on generating finer grained partitions for "Date = $d3$."

Standard write-time partitioning mechanisms are not able to address the requirements discussed above. Consider these examples:

↪ Fine-grained range partitions (e.g., 10MB per partition). This means at partitioning time of reading 1TB data with 1000 scan workers, at least 100K write-time partitions need to be considered, resulting in unnecessarily high partitioning cost.

↪ Coarse-grained range partitions (e.g., 1GB per partition). While this reduces partitioning cost for the case mentioned above, it can only produce a single partition when reading 1GB data even with 100 scan workers, and therefore is unable to meet the parallelism requirements.

↪ Hash partition on $\langle \text{CustomerId, Date} \rangle$. Given the skewness in data (e.g., one particular "Date = $d3$" can have a lot more data than the other dates), such partitioning does not have good control on the partitioning granularity, and can potentially lead to both high partitioning cost and insufficient parallelism depending on query workloads.

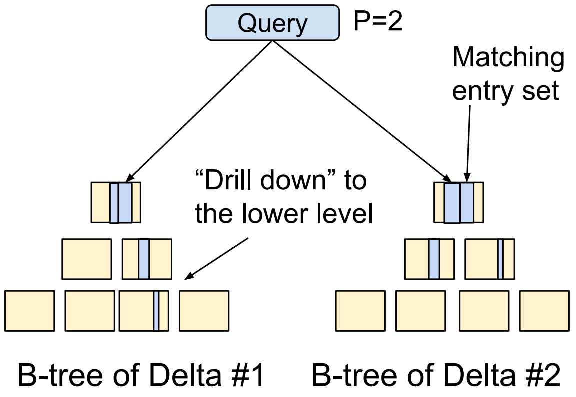

Figure 4: An example query requesting 2 partitions over 2 deltas

Note by this example, we have established that partitioning is highly specific to query under consideration and existing write-time partitioning is inadequate for workloads with a wide spectrum of query workloads. We have also established that one needs a way of producing both fine and coarse grained partitions even on the same key range, based on the query predicates, latency target and resource budget.

This leads to the central idea in this paper to leverage an indexing structure that is used to serve lookup queries. If the index structure can also serve in aiding the partitioning and satisfying the requirements mentioned above, it can serve the dual purpose of fast querying and fast, even and "good enough" partitioning. We discuss the nature of the progressive query-specific partitioning algorithm that Napa uses in the following section.

5 PROGRESSIVE PARTITIONING

In this section, we present the progressive query-specific partitioning algorithm using *size-enhanced* B-trees. Our solution is to enhance typical B-trees with the statistics on data size in a hierarchical manner. For each delta, we maintain a B-tree pointing to data blocks, e.g. a block in the PAX layout [2]. The index not only helps a query to efficiently seek on data using prefix keys but also provides statistical information for partitioning. Both querying and query-specific partitioning traverse the B-tree. The querying traverses the B-tree through the leaf level to visit data blocks. The query-specific partitioning does not visit data blocks and visits the index node at the leaf level only when required for finding "good enough" partitions as we describe below.

To tie the algorithm with the use-case described thus far, recall that the query latency QoS and resource budget considerations are reduced to a limit on the number of scan workers. Given a particular query, this limit is equal to the number of target partitions P. Next, depending on the QoS requirements and prior workload analysis, we also specify an error margin ratio θ which is a parameter to control the precision of partitioning results. The error margin ratio is defined as the ratio between the estimated size of the partition splits and the ideal size of the partition if one could partition perfectly equally. We illustrate this with an example later in this section. An external tuning module automatically adjusts P and θ in a feedback loop but the details of these are beyond the scope of this paper. Note that the proposed algorithm is progressive in terms of θ in the sense that it terminates as soon as the the required error margin ratio is achieved.

5.1 Algorithm Overview

For a given query Q, the algorithm divides D deltas into P key range partitions that are nearly even with an error margin of θ. Each delta has a size-enhanced B-tree, which carries keys and the size of data between these keys. We analyze the estimated size of matching data by combining keys and sizes from D indices. To get the most precise matching estimate from B-trees, we may need to visit the leaf index entries matching with the query. However, reading all the B-trees at the leaf index entries level can be prohibitively expensive for a latency sensitive query. On the other extreme, a lightweight solution would try to infer the sizes of the P partitions by just accessing the root nodes of the B-trees of each delta. Clearly, the ideal solution lies somewhere between the two extremes. The progressive algorithm starts the analysis at the highest level (i.e., root) of the B-trees and accesses lower levels of the tree index selectively and strategically only if more precise information is required.

Figures 4–5 provide an illustration of how the algorithm works. Figure 4 shows a query which is on a table with two deltas (i.e., $D = 2$). The number of partitions requested is 2 (i.e., $P = 2$). Without loss of generality, the table's primary key consists of a single column Key, and the query includes a predicate Key IN $(K1, K2, ..., K8)$.

In Figure 4, we can see the traversal of the algorithm as it seeks positions to cut the matching data half, precisely enough with the error margin ratio θ which will be introduced later in the example. As one can see from the figure, the algorithm traverses deeper (referred to as drill down) in the first B-tree but not all index entries that match with the query (called *matching entries*) need to be drilled down equally. In some sense, the algorithm forms a "wave" or "front" that is just sufficient to partition the data at the required error margin ratio.

Now, we can look at the mechanics of the actual decision process that decides if deeper visits of the trees are needed. Figure 5 shows matched prefix key ranges of query Q with the root node of B-trees corresponding to deltas #1 and #2, resulting in two ordered sets of matching entries: $\{[K1, K3), [K4, K8)\}$ from one B-tree and $\{[K2, K5)\}, [K6, K7)\}$ from the other. The index also includes the size such as the number of rows or bytes of each entry. For example, the second matching entry of the first ordered set indicates that the data size between $K4$ and $K8$ is 20 in delta #1.

To find where to partition (or split), the algorithm merges these keys in the matching entries to distinct keys in the sort order, which serve as split point candidates. In this example, the total size is estimated as 15 + 20 + 20 + 15 = 70 from the sum of all matching entry sizes, and the algorithm selects a split point between K4 and K5. Note that this split will cut entries $[K4, K8)$ and $[K2, K5)$ somewhere in the middle. Since we do not know the precise data distribution within these entries, we can only estimate the partition

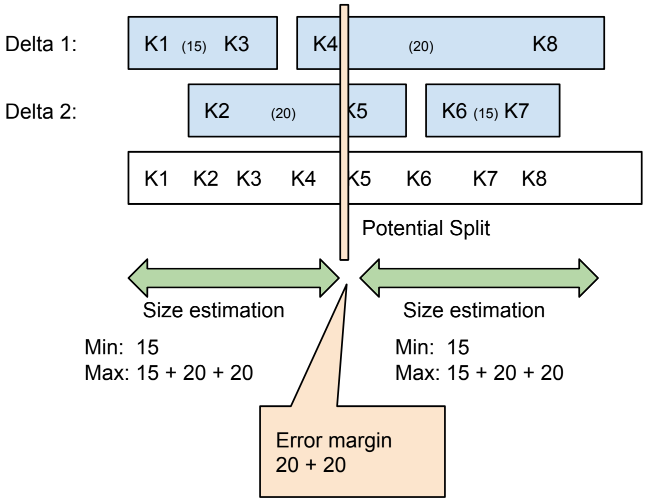

Figure 5: Size estimation for a potential split

sizes. For the first partition, the smallest possible size is 15 (when the data in the cut entries is skewed to the right side of the partition) and the largest possible size is 55 (when the data in the cut partition is skewed to the left side of the partition). We estimate the size as $(15+55)/2 = 35$ with the size margin $55 - 15 = 40$. Similarly, for the second partition the smallest possible size is also 15 (when all the data is skewed towards the left in the split nodes) and the largest possible size is 55 (when all the data is skewed towards the right) resulting in the size margin 40. If the ratio of the size margin to the estimate size (40/35 for the first partition and 40/35 for the second partition) is within the error margin ratio θ, this split is considered to be accurate enough. On the other hand, if the error margin is too large, the algorithm will retrieve matching entries from the lower level of the B-trees for more fine-grained size information. In this example, we replace entries [K4, K8) and [K2, K5) with their matching child entries. We call this operation of replacing an entry with lower-level entries a drill down. Note that we only need to drill down entries contributing to the error margin. The other entries are kept in the working set (called the *matching entry set*) without drill down.

The progressive algorithm repeats the analysis and drill-down until it finds a "good enough" partitioning result. In the following, we describe the details of the algorithm.

5.2 Algorithm Details

In this section, we describe in detail the progressive partitioning algorithm. We start with the basic building blocks of the algorithm followed by a complete description.

During partitioning, we maintain information acquired from index entries for each delta d_i s.t., $i = 1, \cdots, D$ matching with query Q. We call this information a *matching entry set* E_i for delta d_i with query Q. Given D deltas, we maintain the matching entry sets $\mathcal{E} = \{E_1, E_2, \cdots, E_i, \cdots, E_D\}$.

In a B-tree node, we embed the size information between two consecutive entries e_i and e_{i+1}. The size corresponds to the number of rows or number of bytes between $[key_i, key_{i+1})$ where key_i is the boundary key corresponding to entry e_i. For simplicity of exposition, we assume that the size is the number of rows corresponding to the data block. To represent approximate estimation of matching data for a query, we utilize this size information during the match between the index entries and the queried keys.

We represent this size-embedded index entry in a B-tree index node as $e = \langle \text{start}, \text{end}, \text{size}, \text{level}, \text{block} \rangle$, which corresponds to the key range $[e.\text{start}, e.\text{end})$ having the estimated size $e.\text{size}$. The entry e is in an index block at tree level $e.\text{level}$ (the value of 0 for $e.\text{level}$ corresponds to the leaf level). Finally, $e.\text{block}$ is a pointer to the child index block having range $[e.\text{start}, e.\text{end})$ (if $e.\text{level} > 0$).

Initially, we start with a (virtual) root entry e_0 that represents the entire delta. Its start (end) key is the minimum (maximum) key of the input and the size is the total delta size. This gives the initial size estimate of the query: $e_0.\text{size}$ if $[e_0.\text{start}, e_0.\text{end}]$ overlaps with the query Q (0 otherwise). To refine this size estimation, we can visit the root index block and match it with the queried keys. Entry e is matching if its range $[e.\text{start}, e.\text{end})$ overlaps with Q. From the root index block, we can identify the set of matching entries.

The *matching entry set* E provides approximate distribution of matching data: sampled keys and sizes between them. If the entry range $[e.\text{start}, e.\text{end})$ is included in Q, the corresponding size is the "precise" estimate whereas if it only partially overlaps with Q, it provides the maximum (or minimum) size estimate. If we want a more precise estimate of matching data, we can select the corresponding entry from E and visit the corresponding block at the lower level of the B-tree. We call this operation DrillDown, which is defined formally as follows as Algorithm 1.

Algorithm 1 DrillDown(E, e, Q)

```
return $E \setminus \{e\} \cup \{c | c \in \text{GetBlockEntries}(e.\text{block})$ AND $[c.\text{start}, c.\text{end}) \cap Q \neq \varnothing\}$;
```

In DrillDown, entry e is removed from the matching entry set E, and replaced with a new set of entries retrieved from e's child index block that also overlap with the query Q. GetBlockEntries is the process to acquire entries from e's child index block via pointer $e.\text{block}$. This process involves I/O operations. In our actual implementation, we drill down multiple entries $DD = \{e_1, \cdots, e_k\}$ over the matching entries $\mathcal{E}$ across all deltas in a batch: DrillDown($\mathcal{E}, DD, Q$) to exploit I/O parallelism.

5.3 Split point candidates

Given the entry sets $\mathcal{E} = \{E_1, \cdots, E_D\}$ acquired from the indices of D input deltas, we next find keys to split them into P partitions. We do this in two stages: (1) generate a set of split point candidates and (2) find split points from the candidates. The split point candidates are the keys in each E_i that have an overlap with the query. Let there be m such candidates represented by distinct keys $K = \{k_1, k_2, \cdots, k_m\}$ and *cumulative size estimates* $C = \{c_1, c_2, \cdots, c_m\}$. Note that $K = \{k_1, k_2, \cdots, k_m\}$ is a sorted list of the distinct keys that appear either as the start or end key of an entry in $\mathcal{E}$ (i.e., combined from all the deltas). Given the target partition size based on the number of partitions P and cumulative size identified in C, we find split points in K such that the difference between the actual partition size and the target partition size is minimized (the greedy heuristic will be described in detail below). As shown in the example (Figure 5), we consider splitting the input at a location between two consecutive keys. We call the location between k_i and k_{i+1} a *split point candidate* at i. We consider $m - 1$ split point candidates between m keys. The i-th point is to cut somewhere between k_i and k_{i+1} using a prefix key p where p in $(k_i, k_{i+1}]$ and is a minimum distinguishable prefix of k_{i+1}. The reason for using a prefix p instead of key k_i is to prioritize a shorter key length in a partition boundary. When we read rows from the data, we will need to compare key columns between the boundary and a row. When the data is laid out in a columnar-oriented way (e.g. PAX [2]), a longer key length involves more columnar access.

To find a place to split the input, we introduce a cumulative size estimate c_i for each key k_i as the size of the matching data within a range $(-\infty, k_i]$. Instead of having one number for c_i, we maintain the minimum and maximum estimation bounds (i.e., $c_i.min$ and $c_i.max$). We define the minimum and maximum cumulative sizes as follows. $c_i.min$, the minimum cumulative size at split point candidate i, is the sum of the entry sizes where the entry range [e.start, e.end] is contained in $(-\infty, p)$ for any prefix key $p \in (k_i, k_{i+1}]$. $c_i.max$, the maximum cumulative size at split point candidate i, is the sum of the entry sizes where the entry range [e.start, e.end] overlaps with $(-\infty, p)$ for some prefix key $p \in (k_i, k_{i+1}]$. For example, in Figure 5, the split point candidate at K4 has the minimum cumulative size as the size of entry $[K1, K3]$ and the maximum cumulative size as the total size of entries $[K1, K3]$, $[K2, K5]$, and $[K4, K8]$.

When evaluating a partition between two consecutive split points i and j $(i < j)$, we use the following two metrics:

(1) the *estimated partition size* between i and j:
$size(i, j) = (|c_j.min - c_i.max| + |c_j.max - c_i.min|)/2$
(2) the *size margin* at i:
margin(i) = $c_i.max - c_i.min$

$size(i, j)$ is the average of minimum and maximum partition sizes, and margin(i) and margin(j) provide the estimation precision for $size(i, j)$ at both ends. If $margin(i)/size(i, j) > \theta$, we consider the margin at i is too large for the partition between i and j.

To create P partitions, we need to choose $P - 1$ points from the m split point candidates to split the input evenly. Notice that such selection may not always be possible. First, if $m < P$, the number of candidates is too few for P partitions. Even if there are many candidates $m > P$, we may not have points that split the data accurately enough (the size margin is too large). In both cases, we need to find where to drill down to get more granular information. In the following, we describe how to select points (split point selection), how to check if they are good enough (partition quality), and how to select where to drill down if the result does not meet the quality threshold.

5.4 Split point selection

Given cumulative size estimates $C = \{c_1, \cdots, c_m\}$ and the target number of partitions P, we find split points $S = \{s_1, \cdots, s_{P-1}\}$ where $s_i \in [1, m)$. We employ a simple greedy selection of split points.

We use target cumulative sizes $T = \{t_1, \cdots, t_{P-1}\} : t_k = \textsc{TotalSize}(\mathcal{E}) * k/P$ for each $k = 1, \cdots, P - 1$. For example, if $\textsc{TotalSize}(\mathcal{E})$ = 1000, and P = 10, then T = $\{100, 200, \cdots, 900\}$. Note that $\textsc{TotalSize}(\mathcal{E})$ is the total size of the matching set $\sum_{e \in \mathcal{E}} e.size$, which is already computed in C as the cumulative size of the last entry ($c_m.max = c_m.min$).

We search for a split point closest to each target t_k such that

$$s_k = argmin_{i=1,\cdots,m-1} max(|t_k - c_i.min|, |t_k - c_i.max|).$$

Given that the target sizes are monotonically increasing, we can find points in $O(min(P \log m, P + m))$ where P is the number of partitions and m is the number of distinct keys in $\mathcal{E}$ that overlap with the query.

5.5 Partition Qualification

We next evaluate whether the partitioning results with given splits $S = \{s_1, \cdots, s_{P-1}\}$ is good enough. Specifically, we identify unqualified points in S, which will contribute to the degraded partition quality. We employ two criteria for qualification and identify unqualified points U as:

$$\begin{aligned} U = \{s_i | s_i \in S, & (s_i = s_{i+1}) \\ & \vee (margin(s_i)/size(s_{i-1}, si) > \theta) \\ & \vee (margin(s_i)/size(s_i, s_{i+1}) > \theta)\} \end{aligned}$$

The first term $(s_i = s_{i+1})$ means two target sizes hit the same split point, resulting in an empty partition range. For example, if the number of split point candidates m is smaller than the number of partitions P, a selection will have such unqualified partitions. This is a candidate of drill down because having more fine-grained keys may make the two target sizes hit different points. The remaining two terms imply that the size margin at s_i is too large for the error margin ratio θ with respect to the two partition splits here. In either case, the problem of an unqualified point is its size margin being too large relative to the partition size. A large size margin suggests that we need a drill down here to identify more keys at a granular level as split point candidates. If U is empty, the given split points in S meet the quality threshold.

5.6 Drilldown selection

For each u in the unqualified points U, we find entries to drill down, expecting that we will find better partitioning result from updated $\mathcal{E}$. Figure 6 illustrates an example of an unqualified point and entries. Seven entries from 4 deltas are merged as split point candidates, and the split point candidate at K4 is considered unqualified. We need to identify entries that involve the imprecision (i.e., size margin) at point K4. Such entries are included in the maximum cumulative size but not included in the minimum cumulative size at K4 (i.e., 3 entries $[K2, K5]$, $[K3, K6]$, $[K4, K8]$). We call them the entries cut by point i.

The entries in $\mathcal{E}$ that are cut by point i are given by:

$$\textsc{Cut}(\mathcal{E}, i) \quad = \quad \{e | e \quad \in \quad \mathcal{E}, e.\text{start} \quad \leq \quad k_i \quad < \quad e.\text{end}\}$$

Now we consider drill down candidates $DD(\mathcal{E}, U)$ for a set of unqualified points U from entries $\mathcal{E}$.

$$DD(\mathcal{E}, U) = \{e | e \in \textsc{Cut}(\mathcal{E}, u), \exists u \in U \wedge e.\text{level} > 0\}.$$

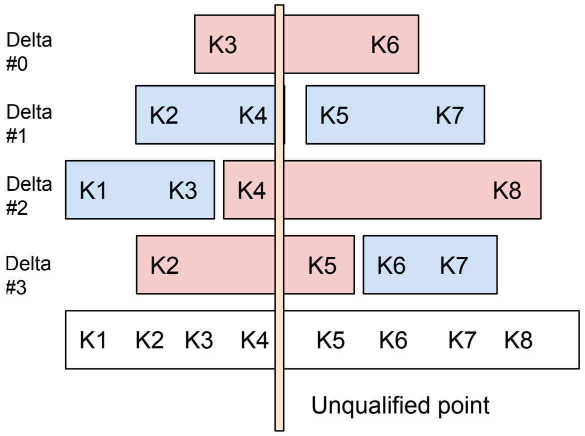

Figure 6: Entries cut by an unqualified point at K4.

Note that we only consider an entry with e.level > 0 (i.e., an entry we can drill down). If e.level is 0, the entry already provides the finest granular information available in the index.

A naive way to find $DD(\mathcal{E}, U)$ involves key comparison between entry start/end keys and k_u for Cut$(\mathcal{E}, u)$, which can be expensive when a key has many columns. We should be able to avoid key comparison because we have done that when we generate distinct keys $K = k_1, \cdots, k_m$. The initial approach we implemented was to track and record entries that have started but have not ended at each key when we generate K. For each key $k_i \in K$, we have a set of entry indices that covers split point candidate i. In the example, the entries corresponding to K4 can be represented as $\{0, \_, 1, 0\}$ (entry #0 of delta #0, entry #1 of delta #2, entry #0 of delta #3). Similarly, the entries at K5 are $\{0, 1, 1, \_\}$. A drawback of this approach is that the required memory size for this mapping is $O(D \times K)$ and therefore expensive in production for a large number of deltas D. When we split D deltas into P partitions, we usually need to have $K = O(D \times P)$ keys (i.e. $O(P)$ keys from each delta). Then the size of the mapping would be $O(P \times D^2)$.

An alternative approach is to maintain a mapping M from each entry $\mathcal{E}$ to keys K: We remember that the start and end keys of each entry appear in K (i.e., i and j if e.start $= key_i$, e.end $= key_j$). In the example, the mapping has $(2, 4, 5, 7)$ for delta #1, indicating that they have keys K2, K4, K5, K7. This approach consumes less memory $O(|\mathcal{E}|) \sim O(P \times D)$ (i.e., $O(P)$ entries from each delta) but needs search to find entries for an unqualified point: For each unqualified point u, we take $O(D \log P)$ (instead of $O(D)$) to find Cut$(\mathcal{E}, u)$. Note that, compared to multi-column key comparison, the binary search $O(\log P)$ only requires integer comparisons. Given that the drill down involves potentially expensive I/O, the extra CPU cost of search was negligible in practice (for P up to thousands), providing a better trade off to conserve memory.

We have at most D entries (one from each delta) in Cut$(\mathcal{E}, u)$ for each unqualified point. Instead of drilling down all the entries, we employ the following heuristic: select entries in Cut$(\mathcal{E}, u)$ where level is the maximum. The underlying intuition is as follows: deltas typically have exponential size distributions (multiple small deltas will be compacted into a larger delta), ending up with B-trees with different sizes. When we have two entries with different levels, it means one is much larger than the other. The contribution of drilling down the smaller entry will be much smaller than the one from drilling down the larger entry.

5.7 Putting it together: The Progressive Partitioning Algorithm

The overall partitioning algorithm is illustrated in Algorithm 2. AccumulateKeysAndSizes will find distinct keys K, the corresponding cumulative sizes C, and mapping M from entry keys in $\mathcal{E}$ to keys in K. We find $P - 1$ split points S to cut C evenly and check whether there are unqualified points U. If all the points are qualified (U is empty), we have found good enough partitions. Generate split keys from K and S and estimate partition sizes from C and S. Otherwise, we find entries to drill down as DD. If DD is empty, we can no longer improve $\mathcal{E}$. Return partitioning based on the current result S. DrillDown() will perform index block retrieval in parallel for DD and matching with the query Q to update $\mathcal{E}$ with new matching entries. In our implementation, the retrieval for individual deltas is independently done in parallel, and the retrieval of blocks in each delta is done in a batch through our proprietary distributed file cache.

Algorithm 2 The Progressive Partitioning Algorithm

```
$\mathcal{E} \Leftarrow e0_1, \cdots, e0_D$;                          ▷ virtual root entry.
$P \Leftarrow$ num. split points
repeat:
    $\langle K, C, M \rangle \leftarrow$ AccumulateKeysAndSizes($\mathcal{E}$);
    $S \leftarrow$ FindSplitPoints(C, P);
    $U \leftarrow$ FindUnqualifiedPoints($S, C, \theta$);
    if $U = \varnothing$ then                          ▷ good enough result.
        return CreatePartitions(K, C, S);
    end if
    $DD \leftarrow$ FindEntriesToDrilldown(U, M, $\mathcal{E}$);
    if $DD = \varnothing$ then                       ▷ no more improvement.
        return CreatePartitions(K, C, S);
    end if
    $\mathcal{E} \leftarrow$DrillDown($\mathcal{E}, DD, Q$);
until True
```

The above algorithm takes the number of partitions P as the initial input. In practice, we use other variations of the inputs, which can be handled in the framework of this algorithm. (1) Minimum partition size and target number of partitions. We provide the user a way to specify the number of workers (i.e., parallelism), which lets the user control the trade-off between the latency (benefit) and resource usage (cost). However, since the user has little idea on the actual query cost (e.g., the total size to read), the specified number of workers can be too large. Unnecessarily high query parallelism can hurt query performance due to having higher chance of hitting random tail events. The system can prevent such over-partitioning by enforcing the minimum partition size. (2) Maximum partition size and maximum number of partitions. In a batch-oriented workload with large scans, it is often more convenient to specify the desired scan size rather than the number of partitions.

When the partitioning input involves minimum or maximum partition size, the algorithm will derive the target number of partitions

P based on the total estimated size. Notice that extra information acquired through drill down operations can provide more accurate input size estimation. Thus, although Algorithm 2 takes P as an input, the actual implementation derives P every time when FindSplitPoints is called.

6 EVALUATION

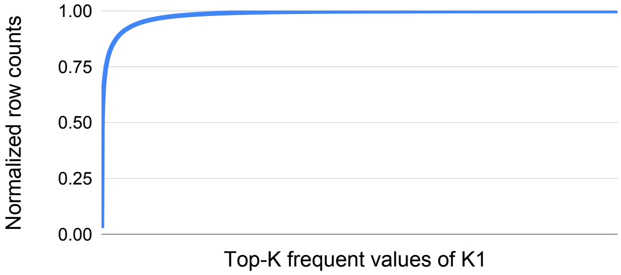

Figure 7: The cumulative number of rows for top-k distinct values: (the number of rows having the top-k values) / (the total number of rows) with varying k for the entire table

We have implemented progressive query-specific partitioning in Napa. The partitioning algorithm described in this paper has been used for both low latency queries and large-scale batch queries for multiple years. In the experiments, we use our testing environment that runs the production code under controlled resource provisioning. This environment is developed for performance regression tests: although the wall time of a query execution is different from the production, the relative performance reflects the actual performance when the proposed partitioning scheme is deployed in production.

6.1 Implementation in Napa

Napa uses F1 Query [10] to perform general SQL queries. Whereas the F1 system drives the distributed processing of the entire query (e.g., join), the Napa query server decides the Napa scan parallelism and the resulting partitioning requirements for the query. Partitioning of the data scan is determined at the planning time: F1 requests Napa to find partition ranges for each query and dispatches them to parallel F1 workers to execute the query. Note that the planning time is included in the end-to-end query latency. The algorithm must find good partitions within a limited time budget since slow partitioning will result in higher query latency. On the other hand, fast but inaccurate partitioning will lead to data skew in the parallel scan, and the slowest worker affected by the skew will determine the overall query latency.

Napa's deltas and indices are stored on the Colossus File System [7] and its metadata is stored in Spanner [5]. The query servers employ caching and prefetch data and metadata. The servers read index file blocks through a distributed caching layer both for partitioning and querying. The caching layer reduces the number of storage I/Os but cannot eliminate them, as the total working set size for Napa's query serving is significantly larger than the aggregated cache memory available [9]. Hence, reducing the amount of index I/Os for the partitioning phase is critical for robust performance. If partitioning uses only a few levels of the B-trees, it is more likely that it requires few expensive storage I/Os. Our implementation of the partitioning algorithm also uses prefetching to reduce the impact of distributed cache misses.

The production system employs additional optimizations that are specific to Napa's data model. In the experiments, we disable these optimizations. In production, we use different values for θ depending on the workloads. For example, when the workload has sub-second latency SLO, we can use θ = 1.0 allowing a 2x larger partition than the estimation for faster partitioning. On the other hand, if the workload is for large-scale data processing, we can use θ = 0 to require most accurate partitioning. In this case, the query is expected to take more than a few seconds, we can afford extra drill down to the leaf level (reading $O(P \times D)$ leaf nodes). We use θ = 1.0 in the experiments on latency-sensitive lookup queries, which is the focus of this paper.

6.2 Data And Query Sets

All our experiments use tables with real production data. For detailed analysis, we use a benchmark query template abstracted from common production queries. The referenced table holds statistics with a key $< K1, K2, ..., K15 >$, representing 15 dimensions. The first key K1 represents ids (e.g. customer id, product id, etc.) and K2 represents a temporal dimension (e.g. date). The table contains multi-year data with K1 having hundreds of thousands of keys while K2 has tens of thousands of keys. Figure 7 shows the distribution of distinct values of K1 for this table, showing high skew (which is very common for ids).

Benchmark query template.

For detailed analysis of the algorithms, we use a query looking up on prefix ranges $< K1, K2 >$ of a table:

```
SELECT K1, K2, SUM(V1),... SUM(Vn) FROM T
    ↪ WHERE K1 IN (a1,···, a8)
    ↪ AND K2 BETWEEN b1 AND b2
    ↪ GROUP BY K1, K2;
```

We select 8 ids for K1 to demonstrate skewed distribution: 1 "large" id (sampled from the most frequent values) and 7 "median" ids (sampled from the median frequent values). The number of rows having the large id is 57K times larger than the other ids combined. We change the size of the range of K2 to control the selectivity of the query. We use relative selectivity compared to the number of rows having K1 IN (a1,..., a8) (i.e., no filter on K2).

As is the case in production, the number of workers (= P) is specified as an input parameter to the query. To provide a good trade-off between the resource consumption and latency, the query should speed up as P increases. The actual number of partitions can be smaller than P if there are not enough keys available from the indices. We call this actual parallelism *the effective number of workers*.

To measure the performance of a benchmark query instance while minimizing unrelated noises, we run it multiple times, discard initial runs, and acquire the average performance after the distributed cache layer warms up.

Production queries. We also sample queries from the production query logs to confirm that the observations in the benchmark are consistent with production queries. We use our performance testing

Table 1: Skew ratio (max worker time / average worker time) of each partitioning scheme (CPU time). The skew ratio 1.0 means perfect partitioning and a larger value indicates skew. The number in a parenthesis indicates the effective number of workers.

	Progressive	S = 250MB	S = 500MB	S = 1GB	S = 2GB
P = 10	1.48 (10)	1.49 (10)	1.87 (10)	2.21 (10)	4.31 (10)
P = 100	1.64 (100)	2.34 (60)	3.09 (32)	2.94 (19)	3.33 (12)

mechanism to automatically run a large number of different queries for different system configurations. Given the diversity of queries in the set, we report (50p, 90p) percentiles in addition to the average.

6.3 Evaluation of Query-specific Partitioning

First, we demonstrate the challenge in fast parallel lookup queries that lead us to develop query-specific partitioning. We compare query-specific partitioning to the approach of a fixed set of partitioning boundaries (e.g., data is range partitioned and stored into separate files). At run time, the query will find partitions that overlap with the query prefix ranges. To emulate this partitioning scheme, we implement a modified version (called "fixed-size partitioning") of the partitioning component of Napa. The algorithm merges entries from the B-tree indices and finds boundary keys for each size *S*. The algorithm checks each partition between boundaries whether it overlaps with the query prefix ranges. Finally it returns only the partitions that overlap (if the number of overlapping partitions is larger than *P*, they will be concatenated into *P* partitions). This partitioning algorithm emulates the case of pre-determined boundaries, but actually takes runtime cost to generate them. Hence, the execution time of partitioning is not useful in this experiment. We focus on the quality of partitioning (i.e., how evenly the prefix range scan is split). The performance is measured as the speed up relative to the sequential execution of the query: (wall time of a sequential scan) / (max wall time of P parallel scans). We measure the speed up for different partition sizes S and compare them with the progressive partitioning (θ = 1). In the experiments the size S ranges from 250MB to 2GB (in logical size after decompression and decoding of physical data).

Figure 8 shows the results for the query with a relative selectivity 0.05. The fixed-size partitioning underperforms the progressive partitioning for all the sizes for $P > 10$.

For further investigation, Table 1 shows the skew ratio (max worker time / average worker time) in terms of CPU time (which will indicate whether the amount of work is evenly distributed across the workers) and the effective number of workers. For P = 100, all the fixed sizes failed to achieve the effective number of workers desirable. S = 250MB is still too large to speed up this query with P = 100. For P = 10, all the fixed sizes achieve the effective number of workers. However, only S = 250MB splits partitions as even as the progressive partitioning. The larger the fixed size is, the more skewed the results are. This result indicates that the fixed partition size should be much smaller than the scan partition size of a query in order to avoid skew.

The experiment result indicates that, if we want to achieve a sub-second latency for this query with 100 workers, we will need fixed-size partitions smaller than 250MB, which means we have to store a peta-byte table into the order of 10 million partitions. Such fixed-size partitioning would be very inefficient, especially under continuous data ingestion: the ingested data may be stored into a large number of tiny files.

Instead of physically partitioning the data into files, we may maintain the logical partition boundaries for fixed-size partitions. Then we do not need physical partition alignment in LSM structure that causes file fragmentation. However, maintaining and retrieving such logical boundary keys in this scale is challenging by itself. This is the problem we have addressed in this paper using B-tree per delta within LSM-Tree structure.

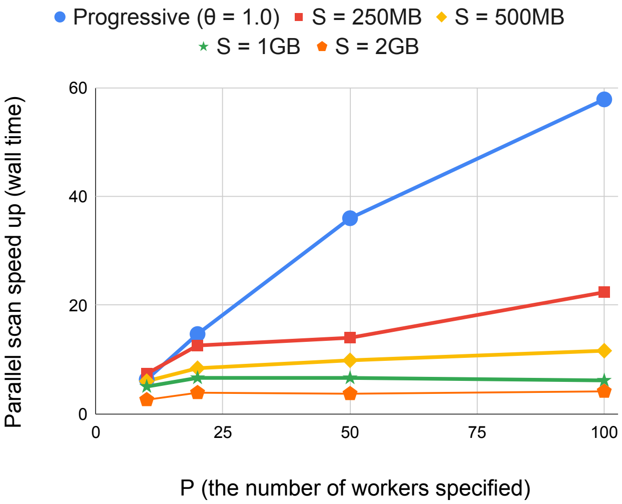

Figure 8: Speed-up of the parallel execution time (max worker wall time relative to P = 1).

6.4 Evaluation of Progressive Partitioning

This experiment compares the performance of progressive algorithms against approaches that maintain flat equi-samples of the tables to aid partitioning. To emulate such flat samples, we implement a non-progressive version of query-dependent partitioning as another baseline, called fixed-level partitioning. The fixed-level partitioning with level = K will retrieve matching index entries at level K of each B-tree index (or highest possible level if the tree is shorter). Level 0 is the leaf level, meaning that the keys of all the matching data blocks are used in partitioning. Level 1 will retrieve sparser key sampling from the next level of the B-trees. Given the set of matching index entries, we perform the same partitioning algorithm for only one round (i.e., no further drill down). In Napa tables, the capacity (the number of entries) of B-tree nodes has been tuned for the progressive partitioning algorithm: A leaf node has 10 entries, and a node at a higher level has 1000 entries. If the algorithm drills down one entry at level 1, it will get up to 10 more matching entries from a level 0 node. We consider the fixed-level

Table 2: Detailed performance comparison at two selectivity points (0.005 and 0.5): the partitioning/total wall time ratio and the effective number of workers.

Relative Selectivity	Algorithms	Partitioning time/total wall time ratio	Effective number of workers
0.005	Progressive	59.66%	33
	Level 0	55.50%	44
	Level 1	26.26%	11
0.5	Progressive	19.70%	200
	Level 0	43.35%	200
	Level 1	21.07%	200

partitioning with level = 0 and 1. We have measured performance of level = 2 but excluded from the results as it clearly underperforms having only 1/1000 of the level 1 keys.

In this experiment, we consider end-to-end wall time including the time taken in partitioning. Note that if the partitioning takes significant time, the performance will be limited even if the scan is split perfectly even. Figure 9 shows the relative wall time of the fixed level partitioning level 0 and 1 against the progressive algorithm (θ = 1) for varying the selectivity of the benchmark query template for P = 200. When the relative selectivity is small (< 0.1), the fixed level 1 is slower than the progressive and level 0. This is because the fixed level 1 does not have enough keys for partitioning. When the relative selectivity is large (> 0.4), fixed level 0 is slower than progressive and level 1.

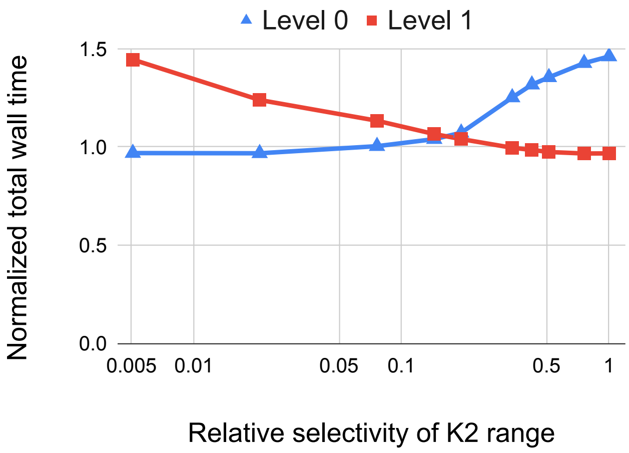

Figure 9: The total wall time of Level 0 and 1 relative to Progressive partitioning with verified selectivity of prefix ranges

Table 2 shows the detailed performance at a small selectivity (0.005) and a large selectivity (0.5). At the small selectivity (0.005), Level 1 uses a smaller number of workers than Progressive and Level 0, which should affect its slow execution. To parallelize this small query, we need to get keys from the leaf index nodes. Progressive and Level 0 have similar total execution time because Progressive also needs to visit read index nodes. At the large selectivity (0.5), all the methods achieve the max number of workers (200) and Progressive and Level 1 outperform Level 0, suggesting that there are enough keys at level 1 index nodes to split this large query. Level 0 reads all the matching leaf index blocks getting more keys than required, resulting in significantly higher partitioning/total wall time ratio.

While this experiment shows a general trend, the actual degree of performance difference can be different in production. Recall that the measured performance is with warm distributed caches to reduce noise. In the actual production, the leaf index blocks can be colder, and the impact of extra index reads to the tail performance can be even more significant. In addition, the impact of skewed distribution to prefix sizes will be more complex in the real queries: it will be hard to determine the fixed level of sampling without actually trying to split in a progressive way.

6.5 Evaluation of Production Queries

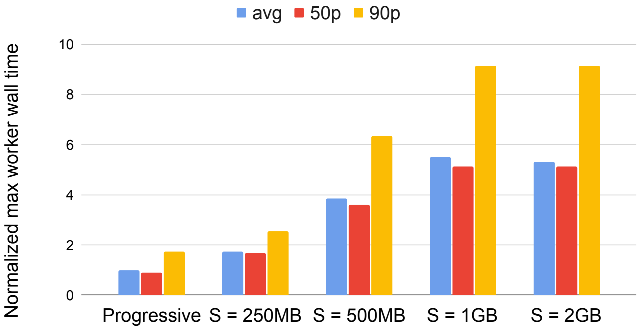

Figure 10: Max worker wall time of progressive and fixed size partitioning normalized relative to the average wall time of progressive partitioning.

We use logged production queries on one representative table to evaluate the performance of progressive partitioning. The cost of the queries varies: from small queries that only need one worker (no parallelism) to queries that use 100 workers. For small queries, the exact partitioning scheme does not make significant performance difference. Hence we exclude small queries (the effective number of workers is less than 10) in the result. Even though small queries don't necessarily need the techniques described in this paper, most of our query evaluation effort goes into large queries that need these techniques and benefit greatly from them. Overall, these techniques enable meaningful savings in our data centers. Figure 10 and Figure 11 are comparisons of progressive partitioning with fixed size partitioning and fixed level partitioning, respectively. Similar to the benchmark, we use the max worker wall time (i.e. excluding partitioning) for fixed size partitioning and the total wall

Table 3: Detailed performance comparison: the partitioning/total wall time ratio and the effective number of workers. The partitioning time is not available (meaningful) for fixed size partitioning (S).

	Partitioning Time / Total Wall Time Ratio			Effective Number of Workers			
	avg	50p	90p	avg	50p	90p	Max
Progressive	3.53%	3.22%	5.37%	27.52	18	64	100
Level 0	3.98%	3.67%	6.67%	30.103	20	69	100
Level 1	0.99%	0.79%	1.73%	5.192	4	10	77
S = 250MB	N/A	N/A	N/A	12.093	8	27	100
S = 500MB	N/A	N/A	N/A	6.8	5	14	100
S = 1GB	N/A	N/A	N/A	3.897	3	8	51
S = 2GB	N/A	N/A	N/A	2.479	2	4	29

time (i.e., including partitioning) for fixed level partitioning. Table 3 shows detailed performance that affect the time difference between the progressive and baseline partitioning methods.

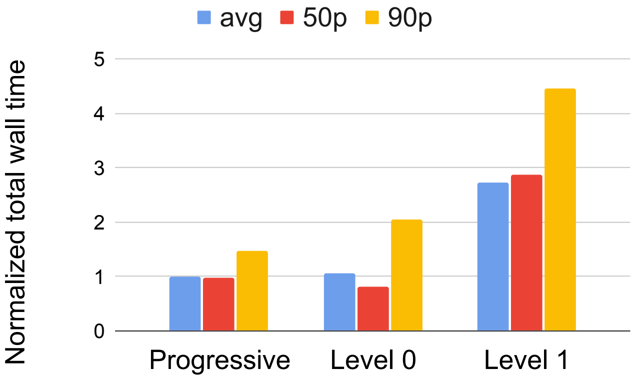

Figure 11: Total wall time of progressive and fixed level partitioning normalized relative to the average wall time of progressive partitioning.

As shown in Table 3, the fixed size partitioning (from 250MB to 2GB) fails to gain as much parallelism (the effective number of workers) as the progressive partitioning can get, resulting in the larger max worker wall time in Figure 10. Similarly, Level 1 suffers from reduced parallelism and takes longer total wall time in Figure 11. Notice that the effective number of workers is smaller than 100 for most of the queries even with Level 0. For such queries, parallelizing more than Level 0 would not gain much improvement: A leaf index node has keys per data block, and a partition that is much smaller than one data block will increase the read overhead per row. Only a small fraction ($<$ 10%) of the sampled queries from the log (even after removing very small queries) are large enough to use 100 workers. For such small queries, the behavior of Progressive partitioning would be very similar to Level 0 as we observe in Figure 11 and Table 3.

The benchmark result indicates that the fixed level 0 would suffer from extra index read to run a large query with high parallelism. Such queries are relatively infrequent and hard to get statistics from the automatic replay of the query log. Instead, we picked up 5 largest queries from query log, ran individual queries multiple times in the same manner as the benchmark query, and observed similar results as the benchmark: The fixed level 0 took 2.6 ~ 3.9x more partitioning time than the progressive, resulting in 13 ~ 26% more total wall time under the warm distributed cache.

7 CONCLUSION

Napa's progressive query-specific partitioning helps it achieve exceptional quality of service on billions of queries per day, with sub-second response time and high efficiency. Our query-specific technique partitions petabyte tables with onerous data skews, to best meet the per-query latency and resource budget requirements. The proposed approach is progressive in the sense it carefully balances the tradeoffs between "perfect" but time consuming versus quick yet "uneven" during the partitioning process. B-tree supports both indexed lookups as well as aids in the partitioning, thus aiding both integral parts of query execution. Experimental results demonstrate the effectiveness of our approach.

ACKNOWLEDGMENTS

We thank the F1 Query team, Andrew Lamb, Ankur Agiwal, Hakan Hacigumus, Haoyu Gao and Yao Liu for their support. We are also grateful to Abdul Quamar, Biswapesh Chattopadhyay and Vijayshankar Raman for their help in proofreading and commenting on the paper draft.

REFERENCES

[1] S. Agrawal, V. R. Narasayya, and B. Yang. 2004. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In *SIGMOD* (Paris, France). 359–370.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. 2001. Weaving Relations for Cache Performance. In *VLDB* (Rome, Italy). 169–180.

[3] D. Bitton, H. Boral, D. J. DeWitt, and W. K. Wilkinson. 1983. Parallel Algorithms for the Execution of Relational Database Operations. *ACM Trans. Database Syst.* 8, 3 (1983), 324–353.

[4] H. Boral and D. J. DeWitt. 1980. Design Considerations for Data-flow Database Machines. In *SIGMOD* (Santa Monica, CA). 94–104.

[5] J. C. Corbett *et. al.* 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3 (2013), 8.

[6] C. Curino, Y. Zhang, E. P. C. Jones, and Sa. Madden. 2010. Schism: A Workload-Driven Approach to Database Replication and Partitioning. *PVLDB* 3, 1 (2010), 48–57.

[7] J. Dean. 2010. Evolution and Future Directions of Large-scale Storage and Computation Systems at Google. https://research.google.com/people/jeff/SOCC2010-keynote.html

[8] J. Dean and S. Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI* (San Francisco, CA). 137–150.

[9] A. Agiwal *et. al.* 2021. Napa: Powering scalable data warehousing with robust query performance at Google. *PVLDB* 14 (07 2021), 2986–2997.

[10] B. Samwel *et. al.* 2018. F1 Query: Declarative Querying at Scale. *PVLDB* 11, 12 (2018), 1835–1848.

[11] S. Fushimi, M. Kitsuregawa, and H. Tanaka. 1986. An Overview of The System Software of A Parallel Relational Database Machine GRACE. In *VLDB* (Kyoto, Japan). 209–219.

[12] F. Halim, S. Idreos, P. Karras, and R. Yap. 2012. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *PVLDB* 5 (02 2012), 502–513.

[13] B. Hilprecht, C. Binnig, and U. Röhm. 2020. Learning a Partitioning Advisor for Cloud Databases. In *SIGMOD* (Portland, OR, USA). 143–157.

[14] S. Idreos, M. Kersten, and S. Manegold. 2007. Database Cracking. In *CIDR 2007* (Asilomar, CA). 68–78.

[15] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. 2011. Merging What's Cracked, Cracking Whatś Merged: Adaptive Inexing in Main-Memory Column-Stores. *PVLDB* 4 (06 2011), 585–597.

[16] C. Luo and M. J. Carey. 2019. LSM-Based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (jul 2019), 393–418.

[17] R. V. Nehme and N. Bruno. 2011. Automated partitioning design in parallel database systems. In *SIGMOD* (Athens, Greece). 1137–1148.

[18] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. 1996. The Log-Structured Merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[19] A. Pavlo, C. Curino, and S. B. Zdonik. 2012. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD* (Scottsdale, AZ). 61–72.

[20] A. Quamar, K. A. Kumar, and A. Deshpande. 2013. SWORD: Scalable workload-aware data placement for transactional workloads. In *EDBT* (Genoa, Italy). 430–441.

[21] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. 2002. Automating physical database design in a parallel database. In *SIGMOD*. 558–569.

[22] F. Schuhknecht, A. Jindal, and J. Dittrich. 2013. The Uncracked Pieces in Database Cracking. *PVLDB* 7 (10 2013), 97–108.

[23] F. Schuhknecht, A. Jindal, and J. Dittrich. 2015. An experimental evaluation and analysis of Database Cracking. *The VLDB Journal* 25 (08 2015), 27–52.

[24] A. Shanbhag, A. Jindal, S. Madden, J. Quiané-Ruiz, and A. Elmore. 2017. A robust partitioning scheme for ad-hoc query workloads. In *ACM Symposium on Cloud Computing*. 229–241.

[25] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. J. Storm, C. Garcia-Arellano, and S. Fadden. 2004. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB* (Toronto, Canada). 1087–1097.