LittleTable: A Time-Series Database and Its Uses

Sean Rhea
Cisco Meraki
srhea@meraki.com

Eric Wang
Cisco Meraki
ericwang@meraki.com

Edmund Wong
Cisco Meraki
elwong@meraki.com

Ethan Atkins
Cisco Meraki
eatkins@meraki.com

Nat Storer
Cisco Meraki
nat@meraki.com

ABSTRACT

We present *LittleTable*, a relational database that Cisco Meraki has used since 2008 to store usage statistics, event logs, and other time-series data from our customers' devices.

LittleTable optimizes for time-series data by clustering tables in two dimensions. By partitioning rows by timestamp, it allows quick retrieval of recent measurements without imposing any penalty for retaining older history. By further sorting within each partition by a hierarchically-delineated key, LittleTable allows developers to optimize each table for the specific patterns with which they intend to access it.

LittleTable further optimizes for time-series data by capitalizing on the reduced consistency and durability needs of our applications, three of which we present here. In particular, our applications are *single-writer* and *append-only*. At most one process inserts a given type of data collected from a given device, and applications never update rows written in the past, simplifying both lock management and crash recovery. Our most recently written data is also *recoverable*, as it can generally be re-read from the devices themselves, allowing LittleTable to safely lose some amount of recently-written data in the event of a crash.

As a result of these optimizations, LittleTable is fast and efficient, even on a single processor and spinning disk. Querying an uncached table of 128-byte rows, it returns the first matching row in 31 ms, and it returns 500,000 rows/second thereafter, approximately 50% of the throughput of the disk itself. Today Meraki stores 320 TB of data across several hundred LittleTable servers system-wide.

1. INTRODUCTION

We live in an age of Internet-connected devices. At Cisco Meraki we produce enterprise-class wireless access points, switches, firewalls, VOIP phones, and security cameras, all of which customers configure and monitor almost exclusively via our website, which we call *Dashboard*. In the consumer space the variety of devices is even greater. The authors of this paper, for example, own Internet-connected smoke alarms, baby monitors, pedometers, and thermostats, which we likewise configure and monitor over the Internet.

Within the management applications for these Internet-connected devices there exists a common need to store *time-series data*—such as packet counters, security events, carbon monoxide levels, or step counts—and retrieve it quickly for customers to view in graphs, tables, and other visualizations. In our experience, customers have a near-limitless appetite for such data. Although most requests cover only summaries of recent data—e.g., bytes transferred per client in the last hour or a monthly ranking of the top network applications observed—customers also use Dashboard and applications like it for ad hoc exploration, root-cause analysis, and forensics. They thus place substantial value in the ability to look further back in time and drill down into greater detail.

This insatiable demand for high-resolution historical data motivates applications like Dashboard to minimize costs by using inexpensive, high-volume spinning storage. All else being equal, more storage is always better, and solid state drives are still significantly more expensive per byte than spinning ones. At the same time, customers digging into their data expect web pages to load in a few seconds or less, and increased data resolution or retention duration should not dramatically affect interactivity.

In this paper we present *LittleTable*, a relational database optimized for time-series data that has been in production use at Meraki since early 2008. A key insight behind LittleTable's design is that time-series data admits a natural clustering by a combination of timestamp and a hierarchically-delineated key. For example, Dashboard organizes wireless access points into groups called networks, and it tracks bytes transferred per device in a table keyed by network and device identifier. LittleTable clusters this table such that any rectangle defined by a continuous range of the key space over a continuous range of time is likely to be contiguous on disk, as shown in Figure 1. Dashboard can thus efficiently render multiple visualizations from the same table. For example, it can display a graph of the total bytes transferred by all devices in a network in the last week or a graph of the bytes transferred by a specific device in the first two hours of last Monday. Both queries are efficient in two senses: they read the requested data from a mostly contiguous area of disk, minimizing seek latency, and they avoid reading unrelated data in the process, minimizing pressure on the page cache. As a consequence, retaining infrequently-read data does not affect the access performance of data queried more often.

A second key insight behind LittleTable's design is that time-series data used by Dashboard allows for much weaker

SIGMOD'17, May 14 - 19, 2017, Chicago, IL, USA

ISBN 978-1-4503-4197-4/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3035918.3056102

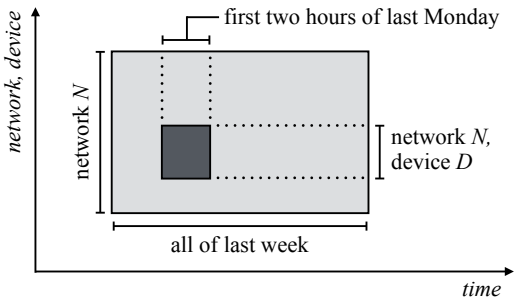

Figure 1: Two-dimensional clustering. LittleTable clusters data so that any rectangle defined by a continuous range of the hierarchically ordered key space over a continuous range of time is likely to be contiguous on disk.

consistency and durability guarantees than those commonly provided by relational databases, simplifying locking and crash recovery while improving performance. In particular, time-series data in Dashboard is *single-writer*, *append-only*, and *recoverable*. At most one process per device inserts any given metric into LittleTable, and data gathered from different devices are independent. There is no need to update rows, as each row represents a measurement taken at a specific point in time. Instead, LittleTable merely ages data out after a configurable time-to-live. Finally, any data recently inserted into LittleTable can generally be re-read from the devices themselves. LittleTable can thus safely lose some amount of recently-written data in the event of a crash.

Although working with LittleTable's weaker consistency and durability guarantees and thinking about data clustering a priori impose additional effort on developers, most LittleTable applications follow one of a few common use patterns, and we illustrate LittleTable's utility by presenting three representative applications. One example fetches byte and packet counters from devices and stores them into LittleTable so that Dashboard can display graphs of transfers per network, client, interface, and destination. Another example pulls and stores logs of events such as wireless associations and disassociations for Dashboard users to browse or search. In the 9 years since LittleTable entered production use, Meraki developers have added 68 additional applications, including the third we present here, which fetches and stores motion vectors from security camera video streams so that users can search for motion in subsections of a camera's field of view. We thus suspect that LittleTable will accommodate a wide variety of similar applications that require high performance storage and retrieval of time-series data.

In return for this additional developer effort, LittleTable is fast and efficient, even on inexpensive spinning storage. Our microbenchmarks show that when querying a table of 128-byte rows and using only a single processor and disk, LittleTable returns the first matching row in 31 ms, and it returns 500,000 rows/second thereafter, approximately 50% of the disk's peak throughput. Inserts are also efficient: LittleTable accepts batches of 512 128-byte rows—common in our application—at 42% of the disk's peak throughput.

In this paper we make the following contributions. We present the design, implementation, and limitations of a relational database optimized for time-series data. We demonstrate via microbenchmarks the performance and efficiency of our design, and we present several applications of LittleTable and discuss how they work within the limitations of its semantics. Throughout, we present lessons we have learned since placing LittleTable into production use.

This paper is organized as follows. Section 2 places LittleTable in context by describing Dashboard in more detail. Sections 3–5 present LittleTable itself, its applications, and its performance in microbenchmarks and production use. Section 6 presents related work, and Section 7 concludes.

2. DASHBOARD OVERVIEW

In this section we place LittleTable in context and motivate its design by describing the architecture of the Meraki system as a whole.

2.1 Shards

Dashboard is implemented as a collection of mostly independent servers called *shards*, each of which implements the entirety of Dashboard's functionality for some subset of Meraki's customers and their devices. Most shards host several hundred Meraki customers. Our largest shards each host over 30,000 Meraki devices and support over two million network clients per day. For geographic proximity and legal reasons such as data sovereignty, we group shards by *region*—e.g., North America, Europe, etc. Customers choose their region when they first create their Dashboard accounts.

Meraki's devices communicate with their hosting shard through a proprietary virtual private network, called *mtunnel*, that secures their communication and allows the shard to initiate communication with devices even when they have no publicly-routeable IP address. The shard stores devices' configurations in PostgreSQL, and devices fetch their configurations over mtunnel. Daemon processes on the shard, called *grabbers*, periodically fetch time-series data from devices over mtunnel and store them in LittleTable. The Dashboard webapp runs directly on the shards, and our customers connect to it over TLS to view these statistics or reconfigure their devices.

Dashboard maintains a mapping of customers and devices to their respective shards. When a customer first logs into `dashboard.meraki.com`, it sends them via HTTP redirect to the shard that hosts their data, and from then on the customer interacts directly with that shard. A similar mechanism redirects devices to the correct shard upon initial boot.

2.2 Fault Tolerance and Load Balancing

To protect against data loss, every shard has a warm *spare*, hosted on equally capable hardware in the same region, but managed by a different provider in a different datacenter in a different city. Dashboard uses PostgreSQL's built-in continuous archiving functionality to keep the spare's PostgreSQL instance consistent by replaying the shard's write-ahead log on the spare. To allow recovery in the case of programming or operational errors, each spare also takes hourly backups that it stores locally. Finally, to protect against simultaneous failure of the multiple replicated drives on both shard and spare, every night the spare signs and encrypts a backup of each database and stores it in Amazon S3.

When a shard fails or becomes unreachable due to a network partition, Meraki's operations team initiates an automated failover sequence that brings the spare out of continuous archival mode and redirects traffic to it by updating DNS records. Once initiated, this process takes only a minute

or two, including the DNS cache TTL. During failover, customers cannot view or reconfigure their networks, but except for a few specific features (e.g., splash pages, packet traces), Meraki's devices tolerate temporary losses of connectivity to Dashboard without incident.

Although shards experience some variation in load due to the mix of device types they host and the features their customers have enabled, as well as diurnal and other regular variations, the primary determinant of a shard's load is the number of Meraki devices it hosts. Consequently, a shard's load tends to change slowly, as it takes time for customers to physically install additional devices. In contrast, quick changes in load are usually a result of code changes or operational errors that should be investigated before being acted upon. Meraki's operations team thus balances load among shards manually. To keep Dashboard responsive, the team *splits* overloaded shards by mapping roughly half of their customers to each of two new child shards. To maintain high resource utilization, the operations team assigns new customers to underloaded shards during customer sign-up.

2.3 Design Considerations

Based on the existing Dashboard architecture and our intended use cases for LittleTable, we now outline the considerations that guided our design. Section 3 describes the design itself and how it meets these considerations.

2.3.1 Horizontal Scalability

At the time that we began building LittleTable, in early 2008, the size of our existing codebase, its reliance on PostgreSQL, and the lack of mature alternatives for storing configuration data all led us to scale out Dashboard via manually partitioning users' data onto independent shards. Dashboard's architecture required that LittleTable support continuous archiving, but there was never any requirement for LittleTable itself to partition data across servers.

Although Meraki's business has grown substantially since 2008, the processing power, memory, and storage per server has grown as well, and today only a select few Meraki customers have so many devices that Dashboard distributes their data across multiple shards. In the few places Dashboard shows a rollup of such customers' LittleTable data, one of the involved shards pulls data from the others and aggregates it before forwarding the result to the client. A more distributed database would simplify implementing such pages, but would likely introduce performance and operational challenges of its own.

2.3.2 SQL Support

Another consequence of Meraki's pre-existing use of PostgreSQL that initially surprised us was the importance of an SQL interface to LittleTable. Our first implementation of LittleTable had an XML-based query language, and developer uptake was sluggish until a subsequent version added SQL support. While alternative data models and their associated query languages—e.g., graph databases [7], multidimensional sparse maps [8, 14], etc.—are quite popular today, using a well-understood and commonly-known query language was extremely valuable to our developers.

2.3.3 Insatiable Storage Requirements

It is almost impossible to store too much time-series data. There is always some customer who wishes they had more history or higher resolution. At the time we built LittleTable, this consideration definitively ruled out solid state drives. Even today, spinning storage remains significantly less expensive per byte, although the gap continues to close.

To achieve high performance on spinning disks, LittleTable must cluster data such that it can retrieve the rows for any given graph or table from a mostly contiguous area of disk, and there are two dimensions along which such clustering can be fruitful. First, we observe that customers most frequently visit Dashboard in order to view relatively recent data about their networks—to debug a client's current connectivity issues, for example, or to generate a monthly summary of network usage. This insight implies that LittleTable should cluster data by timestamp, so that recent data is not interleaved with data from the distant past. Second, most views in Dashboard display data for only a subset of the devices hosted on the relevant shard—usually a single device, network, or customer—and these categories nest: devices belong to networks that belong to customers. This insight implies that LittleTable should also cluster data according to some organizational hierarchy. It should not interleave one customer's, network's, or device's data with another's.

2.3.4 Weaker Consistency and Durability

By design, the time-series data Dashboard stores in LittleTable requires relatively weak consistency and durability guarantees. At most one Dashboard process collects and inserts each type of data for any given device, and that process manages any relationship between a single device's rows (e.g., computed differences in the value of an increasing counter), so there is no need for reader-writer consistency within tables. Measurements of different metrics are generally independent of each other, so there is little need for consistency across tables. Because each row represents a measurement of state at a particular time, and Dashboard only deletes rows by aging them out after a configurable interval, there is no need to update rows once written. Finally, data written within the past few minutes is largely recoverable: in the event of a database crash, Dashboard can generally re-collect such data from the devices themselves.

These weaker semantics are in contrast with the stronger requirements on the configuration data Dashboard stores in PostgreSQL. Updating a configuration may require modifying multiple tables atomically, and updates may occur concurrently, as many users interact with Dashboard at any given time. Database backups must reflect a consistent view across tables. Atomicity, consistency, isolation, and durability are all vitally important for configuration data.

3. LITTLE TABLE

In this section we present LittleTable's data model and API, its basic architecture, and a number of challenges we had to overcome in designing and implementing it.

3.1 Data Model and API

LittleTable is a relational database, run as an independent server process. Clients interact with LittleTable by loading a custom adaptor into SQLite's virtual table interface [4], allowing them to perform inserts, queries, aggregates, and other operations in SQL. Internally, the adaptor communicates with the server over TCP to get a list of available tables, determine the schema and sort order of each table, and perform inserts or queries.

The schema of a table in LittleTable consists of a set of columns, each of which has a name, type, and default value. An ordered subset of these columns form the table's *primary key*. The final column in this subset must be of type timestamp and named "ts". Although other columns may also be of type timestamp, we hereafter refer to this column as *the* timestamp column. LittleTable enforces the uniqueness of primary keys, and the server always returns query results to SQLite in ascending or descending order by primary key.

LittleTable clusters data by timestamp and sorts data by primary key within each cluster. Applications choose their primary keys carefully to control storage layout. The primary key in Figure 1, for example, is (network, device, ts).

The SQLite adaptor takes clients' inserts and transmits them to the LittleTable server in batches. Each inserted row's timestamp may be in the past or in the future. A client may also omit a row's timestamp entirely, in which case the server sets it to the current time.

LittleTable does not provide any way for clients to determine whether their inserts have reached stable storage. It guarantees only that if it retains a particular row after a crash, it will also retain all rows that were inserted into the same table prior to that row. Note that this guarantee is made relative to rows' insertion order, not their timestamps. The SQLite adaptor maintains a persistent TCP connection to the server in order to detect server crashes. After a disconnection, clients issue queries to determine which rows, if any, to re-insert (see Section 4.1).

The SQLite adaptor takes clients' queries and works with the LittleTable server to execute them. On the server side, every query in LittleTable is an ordered scan of rows within a two-dimensional bounding box of timestamps in one dimension and primary keys or prefixes thereof in the other. These bounds may be inclusive or exclusive. In response to a query, the server returns all rows that lie within the specified bounds, sorted by primary key.

Continuing the example from Figure 1, consider a query that requests the sum of bytes transferred from each device in a network N over the last week. The SQLite adaptor will request from LittleTable all rows whose primary keys begin with network N and whose timestamps are in the last week. Having loaded the table's schema on initialization, the SQLite adaptor will know that the resulting data will be sorted by device identifier, and it can thus perform the aggregation without resorting the data.

LittleTable provides no guarantees as to whether a query that executes concurrently with an insert will return any of the inserted rows. It may return some, all, or none of them. In the absence of an intervening server crash, however, a query that starts after an insert completes will always return all of the insert's rows that fall within the query's bounds.

Tables in LittleTable have a set time-to-live (TTL) period, and LittleTable discards rows whose timestamps fall more than a TTL in the past. This age-based expiration of rows is currently the only form of deletion in LittleTable.

3.2 Design Overview

LittleTable implements each table as a union of sub-tables, called *tablets*, of two types.[1] It places newly inserted rows into an *in-memory* tablet, implemented as a balanced binary tree. When an in-memory tablet reaches a configurable maximum size or age, LittleTable marks it as read-only, adds it to a list of tablets to flush to disk, and allocates another in-memory tablet to receive new rows.

Once flushed to disk, a tablet becomes an *on-disk* tablet. LittleTable writes an on-disk tablet as a sequence of rows sorted by their primary keys and grouped into 64 kB blocks. It also writes to disk an index for each tablet that records the last key in each of the tablet's blocks. On average, these indexes are only 0.5% of their tablets' sizes, so LittleTable caches them almost indefinitely in main memory.

LittleTable caches the range of timestamps each tablet contains, which we call a tablet's *timespan*, and it writes the list of on-disk tablets and their timespans to a *table descriptor file* after every change. Once written, LittleTable atomically renames this file to replace the previous version.

To execute a query, LittleTable selects the set of tablets whose timespans overlap the query's timestamp bounds. It traverses each in-memory tablet's binary tree to find the first key within the query's key bounds. It performs a similar lookup in each on-disk tablet by binary search within the tablet's index to find the relevant block, and another binary search within that block to find the relevant row. Using these starting points, LittleTable opens a cursor on each tablet, filters any rows that fall outside the query's timestamp bounds (which generally do not align exactly with the tablets' timespans), and merge-sorts the resulting streams to form a single result stream ordered by primary key.

3.3 Initial Analysis

Based on the above overview, we now briefly itemize the benefits of LittleTable's design. Assume for the purposes of exposition that all clients let LittleTable set the timestamps on their rows to the current time, and that timestamps have infinite precision. We remove these assumptions about timestamps in Section 3.4.3.

First, note that rather than writing individual disk blocks, LittleTable flushes entire in-memory tablets to disk at once. Based on the average seek time and sequential throughput we observe on our disks (see Section 5.1.1), we set the default flush size to 16 MB, which is large enough to sustain roughly 95% of the disk's peak write rate.

Second, by placing rows into only a single in-memory tablet as it receives them, LittleTable clusters rows by timestamp. By sorting the rows within each tablet by primary key, LittleTable further clusters them by key. When performing a query that covers only a small time range, LittleTable reads from a correspondingly small fraction of tablets, and it uses each tablet's index to start reading at the block corresponding to the query's minimum key bound. As with inserts, reading from each tablet into sufficiently large buffers ensures that LittleTable sustains a high fraction of the disk's peak throughput.

Third, with appropriately-chosen primary keys, LittleTable clusters a single table for a variety of applications. In the example from Figure 1, both the rows corresponding to a particular network and those corresponding to a particular device within a network are contiguous within their tablets.

Finally, a background process in LittleTable efficiently re-

[1] We apologize that this terminology is confusing to readers familiar with the related work. What LittleTable calls in-memory or on-disk tablets, Bigtable and related systems call memtables or SSTables. Bigtable uses "tablet" to refer to a partition of rows assigned to a particular server. For brevity of exposition, we delay further discussion of related work until Section 6.

claims disk space by first removing from the table descriptor and then deleting from the file system any tablet whose rows have all passed their TTL. Because some rows in a tablet may expire before others, the server also filters expired rows from query results.

3.4 Challenges

We now discuss some challenges with the LittleTable design as we have described it so far, as well as how we extend the design to handle them.

3.4.1 Merging Tablets

To limit the number of rows it will lose in a crash, LittleTable by default flushes an in-memory tablet no longer than 10-minutes after it first adds a row to the tablet.

While this approach provides acceptable durability for our applications, it creates a problem for queries. Consider an application that draws graphs over one-week time periods. To read the relevant data for this application, LittleTable must open a cursor on every tablet whose timespan overlaps the week in question. To do so, it must at a minimum read one block from each tablet, and because they are stored in separate files, this likely implies a disk seek for every tablet. As there are over one thousand 10-minute periods in a week, and a modern disk averages around 8 ms per seek, this query would take 8 *seconds* just to return its first row.

The problem does not end with the first row, either. To make efficient use of the disk's peak throughput, LittleTable must read in long runs. A modern disk reads sequentially at around 120 MB/s, so to spend at most half of its time seeking, LittleTable must read about 1 MB at a time. A query that reads from 1,000 tablets would thus need around 1 GB of buffer space to execute efficiently.

LittleTable could reduce the number of tablets a query needs to read from by flushing data less often, but doing so requires more memory to hold in-memory tablets as it fills them and risks losing even more data in the event of a crash.

Instead, a background process within LittleTable periodically *merges* two or more existing on-disk tablets. LittleTable performs such merges efficiently in a single pass by merge-sorting the input tablets and writing the results to disk as a new tablet. Afterwards it rewrites the tablet descriptor file and removes the source tablets from disk.

While merging reduces the number of tablets a query must read, when performed indiscriminately it can dramatically increase the write load on the disk. If, for example, LittleTable merged tablets so as to maintain only a single, large tablet, it would end up rewriting all of the existing rows of a table every time it merged in a newly flushed on-disk tablet.

To merge tablets efficiently, LittleTable instead orders tablets by their timespans' lower bounds and merges the oldest adjacent pair such that the newer one is at least half the size of the older. It includes in this merge any newer tablets adjacent to this pair, up to a maximum tablet size. By merging only adjacent tablets, this approach does not affect the disjointness of tablets' timespans. In the appendix we further prove this process is efficient in that both the number of tablets that remain when no more tablets can be merged and the number of times any one row is rewritten to disk are logarithmic in the total number of rows they contain.

3.4.2 Application-Driven Timespans

By keeping the number of tablets small via merging, LittleTable prevents query execution from being dominated by seek time. There is, however, a cost to having too few tablets. Consider a query over a particular day. If the relevant rows were stored in a single tablet whose timespan covered a year, in executing this query LittleTable might scan 365 times more rows than it returned to the client.

Anecdotally, most queries ask for anthropocentric ranges of time: an hour, a day, a week. These periods also generally increase as one looks further back in time: a network operator debugging a connection problem usually looks only at the most recent hour or two, whereas a CIO might at year's end draw up monthly summaries of network usage.

Following this intuition, LittleTable groups time into three ranges, each measured in even intervals from the Unix epoch: the six 4-hour periods of the most recent day, the seven days of the most recent week, and all the weeks previous to that. It inserts only rows from the same period into any one in-memory tablet, and it will not merge tablets from different periods. To prevent this policy from producing a surge of merge activity as the tablets from a smaller period roll over into the next largest one, LittleTable spreads the merge load across tables by delaying each merge by a pseudorandom fraction of the larger period.

We can thus further bound the number of seeks required to return the initial row from a query, as well as the amount of buffer space the query needs to make efficient use of the disk, by noting that the number of tablets per time period covered by the query will be logarithmic in the number of rows in that period. In practice, most tables in our system contain half a dozen or so tablets per period.

This heuristic is not perfect, but it works well in practice. As we discuss in Section 5.2.4, the average ratio of rows scanned versus rows returned for most of our production tables is less than 1.4.

3.4.3 Timestamps Other Than Now

For various reasons, a number of LittleTable applications insert rows with timestamps other than the current time. In a typical example that we discuss further in Section 4.2, a grabber fetches events such as DHCP leases from devices and inserts them into LittleTable with timestamps marking the time at which the events occurred *on the device*. If a device is disconnected from Dashboard for an extended period, the resulting timestamps may be arbitrarily far in the past.

In LittleTable's design as we have described it so far, allowing inserted rows to have timestamps other than the current time threatens the disjointness of the receiving tablets' timespans and, consequently, the clustering of data by time. To keep tablets' timespans mostly disjoint in the presence of such inserts, LittleTable fills several in-memory tablets at once. It bins rows into these tablets using the same time periods it uses to limit merging: there is one filling tablet for each four-hour period in the most recent day, one for each day in the current week, etc. While this approach can produce tablets with overlap, they end up adjacent when ordered by their lower time bounds, and thus they are often merged together with others of similar timespans.

Adding rows to more than one filling tablet at a time complicates flush ordering, since LittleTable guarantees that if it retains a row inserted prior to a crash, it will also retain *all* rows inserted prior to that row on the same table. When filling only a single in-memory tablet at a time, providing this guarantee is straightforward: LittleTable need

only flush tablets in the order they were filled. With more than one, particularly in the case of applications that insert out of order like the application described above, the rows a client inserts may interleave between tablets.

To maintain its flush ordering guarantee with multiple tablets, LittleTable tracks for each table the tablet t that most recently received an insert. When it processes an insert to a different tablet $t' \neq t$, it adds a flush dependency $t \rightarrow t'$, meaning t must be flushed before t'. These dependencies form a directed graph that may have cycles. Before flushing a tablet t, either because t has reached its size or age limit, LittleTable first traverses this dependency graph to find the transitive closure of tablets that must be flushed first. LittleTable flushes those dependencies that have not already been flushed to disk along with t to form one new on-disk tablet each, and it adds all of those on-disk tablets to the tablet descriptor file in a single atomic update.

3.4.4 Enforcing the Uniqueness of Primary Keys

As discussed in Section 3.1, LittleTable guarantees the uniqueness of primary keys. It enforces this guarantee by looking for duplicates of each inserted key in existing tablets, and in many cases it can do so without blocking on disk. For example, it can confirm that a row's timestamp is newer than any other—and that its primary key is thus unique—using only the table descriptor file. Many of our applications insert rows with timestamps set to the current time, so this case is quite common. LittleTable can also confirm that a row has a larger primary key than any other in the same time period using only the indexes of the tablets in that period. This check is particularly beneficial for aggregators (see Section 4.1.2), which by design insert the rows of each aggregation period in ascending primary key order.

For all remaining inserts, LittleTable must perform a query to determine if the row's primary key is unique, and unlike the two cases above, such queries may wait on disk to retrieve their results. To avoid holding a mutex during disk reads, LittleTable instead implements a small in-memory lock table that blocks other inserts to the same table while allowing queries to continue unencumbered.

3.4.5 Finding the Latest Row for a Key Prefix

Dashboard applications occasionally need to find the latest row in a table for some prefix of the primary key. Section 4.2 describes one such example. Although such queries contain no explicit timestamp bounds, LittleTable can still take advantage the way that tablets partition a table by time to avoid opening a cursor on all tablets simultaneously.

To find the latest row with key prefix k in a table, LittleTable works backwards sequentially in timespan order through each group of tablets whose timespans overlap. It opens on each group a cursor that returns all rows with prefix k in descending key order. If k contains all columns of the primary key except the timestamp, the first row returned by this cursor is the latest with prefix k. Otherwise, because the timestamp column is the last in the primary key, LittleTable must scan through the rows returned by the cursor to find the one with the latest timestamp. If the cursor returns no rows at all, LittleTable moves on to the next group.

While this approach is correct, it is far from efficient. Because the latest row may occur arbitrarily far in the past, the query might eventually open a cursor on every tablet in the table. To further optimize for this case, we are considering storing with each on-disk tablet a Bloom filter summarizing the tablet's keys, as in bLSM [22]. This change would eliminate the need to check 99% of the tablets that do not contain any matching key at a storage cost of only 10 bits per row. These Bloom filters would also be useful to check for duplicate keys during inserts, as described above.

3.5 Implementation Details

LittleTable is currently 20,672 lines of C++, including comments, blank lines, and tests. It supports as column types 32–bit and 64–bit integers, double precision floating point numbers, timestamps, variable length strings, and byte arrays (i.e., blobs). Unlike most SQL databases, it does not support null values; few of our applications need them, and those that do instead use sentinel values (e.g., –1).

LittleTable stores the indexes for on-disk tablets as a footer within the same file as the tablet's rows and reloads them into memory on demand after a restart. It compresses the footer and the tablet's blocks with LZO1X-1 [2], and it uses the final two words of the file to store the footer's decompressed size and its offset within the file. Linux's `ext4` file system will usually store tablets of 1 GB or less in a single extent. It thus takes three seeks to read a tablet's footer: one to read the inode, one to read the footer's offset from the end of the file, and one to read the footer itself.

Our implementation supports only a few schema manipulations. Clients can append columns to the tail of a table's schema, increase the precision of 32–bit integer columns to 64 bits, and alter a table's TTL. They can also drop a table and recreate it with a new schema, an approach we use frequently during new feature development.

To implement schema changes, LittleTable stores each tablet's schema in the tablet's footer, and it stores a table's current schema in its table descriptor file. When reading from a tablet with a previous schema version, LittleTable translates its rows to the latest version, extending the precision of cells or filling them in with the default values from the table schema as necessary. In this way schema changes impose a small CPU cost on queries, but they do not require rewriting any existing on-disk tablets.

LittleTable allows clients to specify a query direction—ascending or descending—and a limit on the number of rows to return. Internally, the server enforces its own limit on the number of rows it will return and sets a *more-available* flag in any query result that hits this limit. The SQLite adaptor retrieves additional rows up to any client-requested limit by updating the starting key bound in a query to the key of the last row returned and re-submitting.

To implement continuous archival of LittleTable data, every 10 minutes Dashboard runs `rsync` from shard to spare repeatedly until a sync completes without copying any files, indicating that shard and spare have identical contents. This approach works because an `rsync` that copies no files is quick relative to the rate of new tablets being written to disk.

4. APPLICATIONS

In this section we discuss three LittleTable applications, all of which gather time-series data from Meraki devices, store it in LittleTable, and display it in Dashboard. We have simplified each application for brevity of exposition, but we pay particular attention to the techniques they use to handle LittleTable's weak durability guarantees, as well as how they handle temporary device unavailability due to

problems with customers' uplinks or the broader Internet. Surprisingly, the strategies for overcoming these two challenges often compliment each other. Throughout this section we assume that LittleTable returns query results at 500,000 rows/second. Section 5.1.5 presents the related benchmark.

4.1 Network Usage

As one would expect of a network monitoring application, Dashboard provides customers with graphs of bytes and packets transferred per network, device, and client. Less typical is that it also provides graphs of transfers per *network application*: by port number (e.g., port 22 for `ssh`), by hostname (e.g., `meraki.com`), or by deep packet inspection (e.g., HTTP headers). Customers use these graphs to monitor usage and to decide whether they should add traffic shaping or firewall rules to their networks' configurations.

4.1.1 Usage Retrieval and Storage

UsageGrabber is a daemon process within Dashboard that periodically fetches counts of bytes and packets per client and application from Meraki devices and stores them in LittleTable for Dashboard to display. For brevity of exposition, we present here a simplified version of UsageGrabber that fetches only byte counters for a single network interface per device. Extending this design to handle multiple clients and applications per device is tedious, but straightforward.

Every minute UsageGrabber fetches from each device D in network N a 64-bit count of the number of bytes the device has transferred, and it keeps an in-memory cache of the previous time t_1 and count c_1 it fetched from each device. UsageGrabber does not insert any row into LittleTable for the very first response it receives from a device, but when it fetches a subsequent count c_2 at time t_2, it calculates an average transfer rate of $r = (c_2 - c_1)/(t_2 - t_1)$ and stores in LittleTable the key (N, D, t_2) and value (t_1, c_2, r), indicating that the device transferred at rate r over the interval $[t_1, t_2)$. Note that because this table is keyed on both network and device, Dashboard can efficiently load the data for either an entire network or a single device within that network.

After a temporary loss of connectivity to a device, UsageGrabber handles the device's next response differently according to the duration of its unavailability. If the unavailability was short—e.g., several minutes—UsageGrabber proceeds as normal. If the unavailability continues for hours, on the other hand, it feels disingenuous to show that the device maintained a steady rate of transfer over the entire period. Instead, if $t_2 - t_1$ exceeds some threshold T, UsageGrabber caches t_2 and c_2 in memory but does not insert any row in LittleTable, and Dashboard users observe a gap in usage for this period. The optimal value of T is subject to taste; Dashboard sets T to an hour.

UsageGrabber reuses the threshold T to compensate for LittleTable's weak durability guarantees. Note that UsageGrabber's behavior on receiving a response from a device after a period of unavailability longer than T is identical to its behavior after receiving a response from that device for the very first time. As such, UsageGrabber can remove from its in-memory cache the entry for any device with which it last communicated further than T in the past. Similarly, after a LittleTable crash UsageGrabber can rebuild its in-memory cache by querying LittleTable for the maximum timestamp and associated counter value for each device from the current time minus T forward. Assuming 30,000 devices per shard, one row per device per minute, and a threshold $T = 1$ hour, this query takes under four seconds.

We now consider the properties of UsageGrabber using the terminology from Section 2.3.4. The rows that UsageGrabber inserts into LittleTable are *append-only*. If necessary for performance, UsageGrabber can be multithreaded, but any given device can be handled by a single thread, so for each device there exists a *single writer*. Finally, usage data is *recoverable*. After a LittleTable crash, UsageGrabber can rebuild its in-memory cache and resume fetching counter values from devices in four seconds. A LittleTable crash thus appears to customers in Dashboard as no more than temporary unreachability of their devices, and the less data LittleTable drops in the crash, the shorter this period of unreachability appears. So long as LittleTable's data losses are shorter and less frequent than Internet connectivity problems, customers are unlikely to notice such crashes at all.

4.1.2 Aggregation and Rollups

Recall from above that UsageGrabber stores one sample per device per minute. If Dashboard were to render a graph of cumulative bytes transferred on a network of 100 devices over the last month, it would read over four million rows from the source table. This read would take an estimated 8 seconds, a long delay for a web page. Furthermore, there is little reason to transfer so many points to a web browser that will use them to draw a graph only a few thousand pixels wide. Instead, background processes within Dashboard aggregate this source table to a new table of cumulative bytes transferred per network over ten-minute periods. Rendering the same graph from this derived table yields only a few thousand points, and it reduces resource usage across the stack: from client-side CPU to server disk bandwidth.

We originally intended to build aggregation directly into LittleTable, in the style of rrdtool [3], but we later realized that by computing aggregates in a separate background process, called an *aggregator*, we were able to iterate more quickly on new aggregation schemes. For example, several features within Dashboard track clients using HyperLogLog [10], a fixed-size, probabilistic representation of a set that permits unions and provides cardinality estimates with bounded relative error.

Another unanticipated benefit of computing aggregates in a separate process was the ability to join source data from LittleTable with dimension tables from our configuration data stored in PostgreSQL. For example, Dashboard allows users to add *tags*—the meanings of which users define for themselves—to clients, devices, and networks. A school might tag its wireless access points with the tags "classrooms", "playing-fields", etc. An aggregator reads the tags for each access point from PostgreSQL and writes a new table of usage keyed on customer and tag, allowing Dashboard to efficiently render graphs of usage per tag. Yet another aggregator annotates client transfers with the likely operating systems of those clients so that Dashboard can produce a chart of usage per operating system.

In some cases, such as with the events logs discussed in the next section, Dashboard allows customers to view the source data of some table far into the past. Even when it does not, however, we are in the habit of retaining source data for as long as disk space allows, as it is hard to predict what aggregates we might later want to compute when implementing new features. By partitioning data by timestamp and auto-

matically and inexpensively aging it out, LittleTable affords us this flexibility at no cost except disk space.

We discuss above how UsageGrabber copes with LittleTable's weak durability guarantees. Aggregators illustrate two additional ways that applications do so. First, because LittleTable flushes rows to disk in insertion order, if an aggregator can find any row from a particular aggregation period in its destination table after a LittleTable crash, then it knows that all prior aggregating periods are complete. At that point, it can simply re-process the period for the row it found and all subsequent periods. Unfortunately, LittleTable provides no built-in, efficient way to find the most recent row in a table. To compensate for this deficiency, aggregators query their destination tables over exponentially longer periods in the past until they find some row. They then find the most recent row via binary search.

Second, aggregators must take care not to insert rows derived from source data that might not yet be persisted on disk. Because LittleTable currently provides no built-in mechanism to accomplish this task, aggregators simply assume that data written more than 20 minutes in the past has reached disk. To remove this assumption, we are considering adding a new command to LittleTable that flushes to disk all tablets with timestamps before a given value.

4.2 Event Logs

In addition to network usage, Dashboard also tracks devices' logs, which include events such as DHCP leases, wireless (dis-)associations, and 802.1X authentications. These logs are particularly useful for diagnosing network connectivity issues or performing forensic analysis. We present this application because it demonstrates yet another way to work around LittleTable's weak durability guarantees—one that has elements in common with those used by both UsageGrabber and the aggregators.

To implement event logs, a Meraki device assigns each event a unique id from a monotonically increasing counter. A daemon process within Dashboard, called *EventsGrabber*, keeps an in-memory cache of the most recent event id, if any, it has fetched from each device. It periodically connects to each device and supplies this value, and the device replies with any more recent events. EventsGrabber then inserts a row for each event into LittleTable, with the network and device as the key and the event id and contents as the value.

Like UsageGrabber, after a restart EventsGrabber must rebuild its in-memory cache. It initiates this process by performing a query over a fixed duration of recent rows, storing the latest event id it finds for any device. If a device has been unreachable from Dashboard or powered off for an extended period, however, its most recent row in LittleTable may be arbitrarily far in the past. In such cases, EventsGrabber fetches from the device without providing any previous event id, and the device responds with the oldest event it has stored. Using that event's timestamp to bound how far back in time to search, EventsGrabber then queries LittleTable to find the latest row for that network and device.

As we note in Section 3.4.5, such queries are not particularly efficient for LittleTable to execute. In addition to adding Bloom filters as described in that section, another way to improve applications like EventsGrabber is to regularly insert for each device a sentinel value that contains the latest event id inserted for that device. So long as the rate of inserting sentinel values is a small fraction of the rate of real events, this approach costs little, and it allows an application to query no further back in time after a restart than a single sentinel period.

As with UsageGrabber, we note that the rows EventsGrabber inserts into LittleTable cluster by timestamp and key and are append-only, single-writer, and recoverable.

4.3 Video Motion Search

Video motion search is one of our newest LittleTable applications. Although it is far from the type of application we had in mind when designing LittleTable, it shares a surprising amount of functionality with event logs. We briefly present it here to demonstrate the diverse range of applications for which LittleTable is appropriate.

Unlike most security cameras, Meraki's cameras store the video they record in flash inside the cameras themselves. A web browser on the same network subnet as the camera can stream live video feeds and historical footage directly from the camera, and Dashboard proxies video for any browser without a local route. When a security incident occurs, a Dashboard user can select any rectangular area of interest in a camera's video frame and search backwards in time for motion events within that area. Dashboard also uses these motion events to draw heatmaps of motion over time.

To implement these features, a background process on each camera processes each video frame and encodes motion events as follows. It divides each 960×540 frame into 60×34 *coarse cells*, each of which contains six columns and four rows of 16×16 pixel *macroblocks*. When a coarse cell changes between frames, the process creates a motion event and encodes it as a single 32–bit word: a nibble each for the row and column of the coarse cell within the frame, and a bit each to indicate the presence or absence of motion in the 24 macroblocks. If a coarse cell contains motion in successive frames, the process coalesces those events, OR'ing together their bit vectors to create a single event and duration.

A daemon process in Dashboard, called *MotionGrabber*, fetches these motion events from the device similarly to how EventsGrabber fetches other events. It stores each bit vector and duration in LittleTable keyed on the camera's identifier. Over a recent week, MotionGrabber stored an average of 51,000 rows/camera system-wide. Again assuming a query throughput of 500,000 rows/second, searching a week's worth of video on a single camera takes approximately 100 ms.

5. EVALUATION

In this section, we analyze LittleTable's performance via microbenchmarks and present measurements of our production deployment.

5.1 Microbenchmarks

We first present microbenchmarks to quantify LittleTable's performance and validate its design.

5.1.1 Experimental Setup and Procedure

We ran our microbenchmarks on a machine with two Intel Xeon E5-2630 v2 6-core processors, 64 GB of 1600 MHz DDR3 RAM, and a software RAID 1 array of two SATA Western Digital WD2000FYYZ 2 TB, 7,200 RPM hard drives with 64 MB of cache per drive. The drives have a single `ext4` partition. To measure performance using only a single spindle, we mark one drive in the RAID array as faulty. The

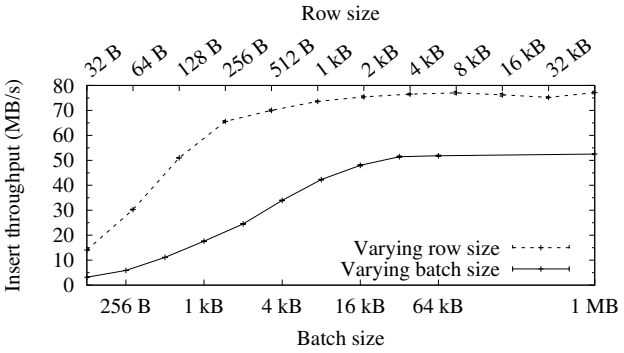

Figure 2: Insert throughput vs. row and batch size. The solid line shows insert throughput with 128-byte rows and varying batch size. The dashed line shows insert throughput in 64 kB batches and varying row size.

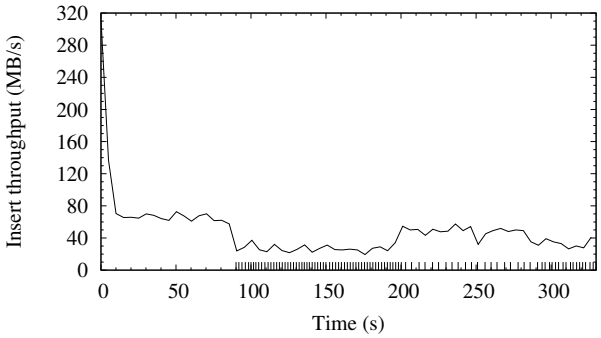

Figure 3: Insert throughput with active tablet merging. Merge events are shown as impulses along the x-axis.

machines run Linux kernel version 3.16.0 using the default file system readahead of 128 kB unless otherwise noted.

Before each benchmark we `sync` all writes, clear Linux's disk cache and the drive's internal cache, and write and read 64 MB of data to a random location on disk. Except for the multiple writer benchmark in Section 5.1.4, our benchmarks are single-threaded. We use `nice` and `ionice` to increase the CPU and I/O priority of our benchmark processes to the highest priority. We run each benchmark on each set of parameters 26 times, and we plot the average of the y-axis metric with a 95% confidence interval, computed using the Student's t-distribution.

We tested the sequential performance of our disks using `dd`, clearing the cache beforehand and syncing the filesystem afterwards, and observed approximately 120 MB/s read and write throughput. We measured an average combined seek and rotational latency of 8 ms when reading random 512-byte blocks from the area of disk used for our benchmarks.

Our insert benchmarks insert rows with timestamps set to the current time to emulate how Dashboard generally uses LittleTable. The benchmarks generate all other input data using a xorshift pseudorandom number generator, effectively disabling LittleTable's LZO compression.

5.1.2 Single Writer Throughput

We evaluate the performance of LittleTable with a single client inserting 500 MB of data into a single table in two experiments. In our first experiment, we explore what effect the number of rows in a single command has on insert throughput. In this experiment, each row is made up of 32-bit integers and is 128 bytes long. The solid line in Figure 2 shows that write throughput increases with larger batch sizes as the relative fraction of per-command overhead and round-trip time decreases.

In our next experiment, we fix the batch size at 64 kB and vary the size of the rows from 32 bytes to 64 kB by varying the size of a single blob value column. We fix the number of key columns to six to keep the amount of work for performing key comparisons constant. The dashed line in Figure 2 shows the results of running this microbenchmark. In this test, LittleTable achieves an insert throughput anywhere from 12% (with 32-byte rows) to 63% (with 4 kB) of peak disk throughput. Overall, LittleTable's insert throughput steadily improves with increasing row sizes, as the relative cost of per-row operations decreases.

5.1.3 Insert Throughput and the Impact of Merging

Since log compaction has been shown to be a significant cost in other log-structured storage systems [23], in this experiment we investigate the cost of merging tablets during inserts into LittleTable. To clearly illustrate the effects of merging, we maximize initial throughput by inserting 4 kB rows in 64 kB batches into a single table, and we limit the memory used by LittleTable so that at any time there are at most 100 outstanding tablets waiting to be flushed to disk. LittleTable flushes tablets at 16 MB and limits merged tablet sizes to 128 MB, its default settings for those parameters.

Figure 3 shows the result of this benchmark. We insert 16 GB of data over 350 seconds and measure the average insert throughput over 5-second windows. LittleTable is CPU-limited and achieves very high throughput until it hits the 100-tablet limit, after which it becomes disk-bound, and throughput levels off around 70 MB/s.

To maximize the number of tablets available to any one merge, LittleTable waits until 90 seconds after a tablet is written before merging it. Consequently, the merge thread becomes active at 90 seconds into this test. (We indicate individual merge events as impulses along the x-axis.)

Once merging begins, insert throughput drops as flushes compete with merges for disk bandwidth. This competition slows inserts down, leaving less data to merge. Eventually the rate of merges decreases, allowing the insert rate to climb again. Near the end of the test these two processes reach more of an equilibrium, at which point insert throughput vacillates between 30-40 MB/s.

Given the high insertion rate in this test, the merge thread always has eight tablets available to merge, and each iteration writes out a 128 MB tablet that is never merged again. We thus observe a write amplification factor of 2, and the final throughput is half the initial disk-bound value. At lower insertion rates LittleTable suffers higher write amplification, but the effects of merging are less obvious, as the disk more easily keeps up with the lower aggregate write rate.

5.1.4 Multiple Writer Throughput

LittleTable's insert performance is CPU-bound for smaller batch sizes. Because the server shares almost no state between tables, we thus expect insert performance to increase with multiple processes writing to different tables. This sit-

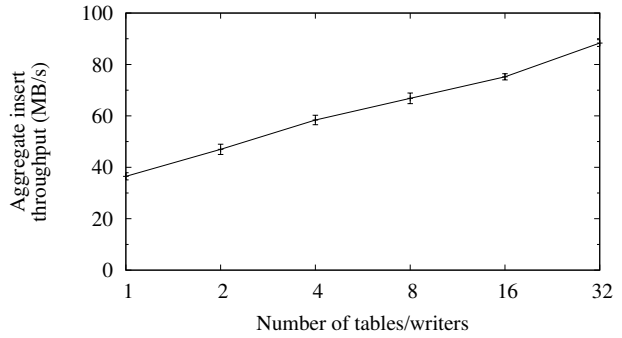

Figure 4: Insert throughput vs. number of writers.

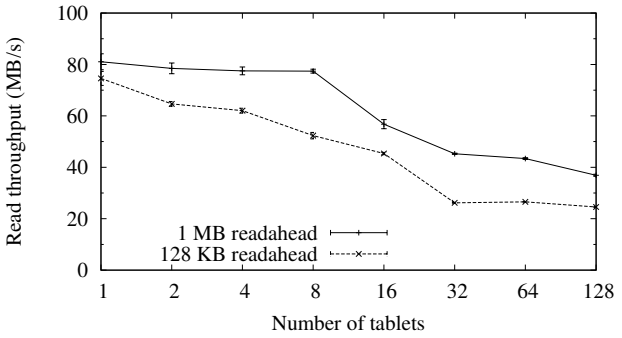

Figure 5: Query throughput vs. number of tablets.

uation more closely matches Dashboard itself, where many separate processes each write to their own table.

In this benchmark, each writer writes 500 MB to a separate table. Each insert is a batch of 32 128-byte rows. We compute the aggregate insert throughput by dividing the total amount of data written by the total time for all requests to finish. Figure 4 shows the results of this microbenchmark. With a single writer, LittleTable sustains 37 MB/s, and each additional writer increases the aggregate throughput. With 32 writers, LittleTable sustains almost 75% of the peak disk write throughput.

5.1.5 Query Throughput

LittleTable's read throughput is relatively independent of both row size and number of readers. In contrast, as Figure 5 shows, it is much more sensitive to the number of tablets LittleTable is simultaneously reading from, as the disk arm must seek back and forth between tablets. This effect is the motivation behind merging tablets as discussed in Section 3.4.1.

In this experiment, we fix the table's total size to 2 GB and each row to 128 bytes while varying the number of tablets. A single reader queries the entire table. We first run this experiment with the default file system readahead of 128 kB, which takes 1 ms to read at the disk's sequential throughput of 120 MB/s. With a sufficiently large number of tablets, it becomes increasingly certain that the disk arm must move to read the next 128 kB for a given tablet. The combined seek and rotational latency is approximately 8 ms, so in the limit the disk should spend only 1/9 of its time reading, for an expected throughput of 12-13 MB/s including LittleTable overheads. We suspect that LittleTable's performance levels off somewhat higher, at 24 MB/s, because the disk's internal 64 MB cache is providing additional readahead. When we rerun the experiment with a larger readahead of 1 MB, LittleTable's performance levels off at around 40 MB/s, which is much closer to what we would expect.

5.1.6 Query First-Row Latency

The latency to retrieve the first row from a query in LittleTable is dominated by the time to read the relevant block from each tablet that overlaps the query's time range. As discussed in Section 3.5, we expect LittleTable to read a tablet's footer in three seeks. Once the footer is cached, we expect it to take only one additional seek to read any block in the tablet.

To confirm this expectation we perform queries for random keys in a table with 128-byte rows and 16 MB tablets. We vary the number of disk tablets from 1 to 32 by adjusting the query's timestamp bounds. We perform two queries, each to a different random key, and we clear all system caches and the buffer cache before each pair. The first query reads the tablet's footer and a single block. The second query reads a different block (in expectation).

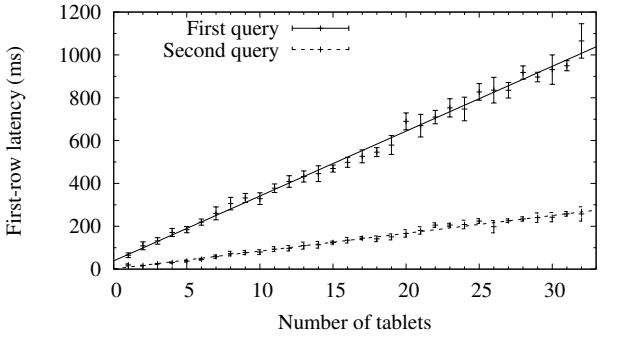

Figure 6: First-row latency vs. number of tablets.

Figure 6 shows a linear regression on both first query and second query performance versus the number of tablets. The slopes of the two lines are 30.3 ms and 8.3 ms per tablet, very close to the 4 and 1 seek times we expect in each case.

5.2 Production Metrics

In this section, we present measurements of our production PostgreSQL and LittleTable deployments.

5.2.1 Database Sizes

As discussed in Section 2.1, Dashboard horizontally partitions customers across shards. Meraki's operation team generally splits a shard when its PostgreSQL database size exceeds its available RAM, as in our usage PostgreSQL performs poorly otherwise. They also split a shard when its LittleTable data begins to fill its disks. As a result, Dashboard stores approximately 20 times more data in LittleTable than in PostgreSQL, roughly corresponding to the ratio of disk to main memory on our servers.

Because Dashboard is composed of multiple generations of server hardware, we visualize the range of PostgreSQL and LittleTable sizes across our production shards as a cumulative distribution function (CDF) in Figure 7. As of January 4, 2017, Dashboard stores a total of 320 TB in LittleTable, with the largest instance storing 6.7 TB. In com-

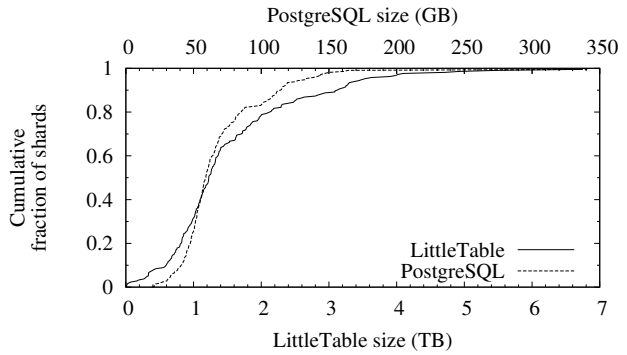

Figure 7: Distribution of PostgreSQL and LittleTable sizes in production.

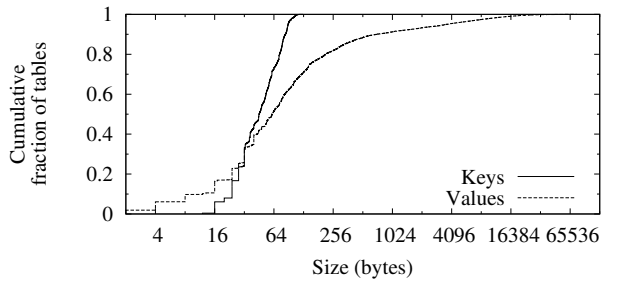

Figure 8: Distribution of key and value sizes per table in production.

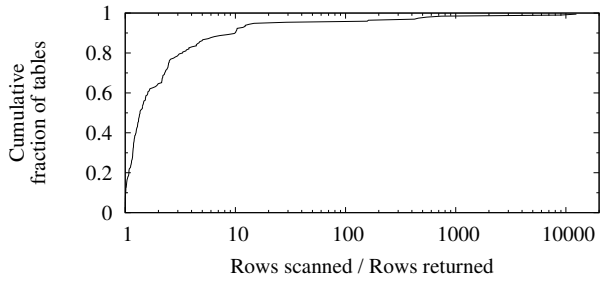

Figure 9: Distribution of rows scanned / rows returned by table in production.

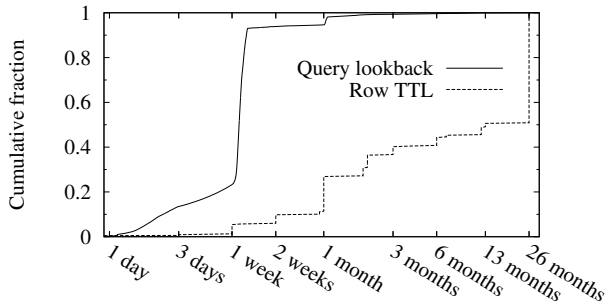

Figure 10: Distributions of row TTL by table and lookback period by query in production.

parison, Dashboard stores only 14 TB in PostgreSQL, with the largest shard storing 341 GB.

5.2.2 Table, Key, and Value Sizes in LittleTable

The features for which Dashboard uses LittleTable vary widely in their storage needs. As of January 4, 2017, Dashboard has approximately 270 LittleTable tables on each production shard. The median table size is about 875 MB compressed, and the largest table in LittleTable, at 704 GB compressed, far exceeds any shard's entire PostgreSQL database.

Figure 8 shows a CDF of the key and value size per table across all tables. Overall, tables have small keys: the median key size is only 45 bytes and all keys are less than 128 bytes. Most values are small as well: the median value is only 61 bytes, and 91% of LittleTable tables have an average value size of 1 kB or less. The largest values store large, probabilistic representations of sets of clients, however, and those have values as large as 75 kB. The average row is 791 bytes, large enough to write at 72 MB/s according to the microbenchmark results shown in Figure 2.

5.2.3 Long-Term Insert and Query Rates

The microbenchmarks in Section 5.1 show the peak read and write throughput LittleTable can sustain. On account of diurnal usage patterns, relatively idle weekends, and intentionally over-provisioning our production system, we observe much lower average rates in production. Between October 10, 2016, and January 7, 2017, LittleTable accepted an average of 14,000 rows/second per shard in inserts and returned an average of 143,000 rows/second per shard to queries. The workload is read-heavy in part due to aggregation: multiple aggregators (see Section 4.1.2) read each source table and write substantially smaller destination tables.

5.2.4 Query and Insert Efficiency

As discussed in Section 3.2, because LittleTable clusters rows by timestamp but sorts rows within a cluster by primary key, a query may scan through many rows that lie within its key bounds but fall outside of its timestamp bounds. Figure 9 shows a CDF of the average ratio per table of rows scanned versus rows returned across all queries processed by LittleTable in a day. On average, queries are very efficient, scanning only 1.4 rows for every row they return, and 80% of tables see a ratio of 3.3 or less. A small minority of queries, however, are from applications looking for the latest value for a prefix of the primary key. As described in Section 3.4.5, unless a query specifies every column in the primary key except the timestamp, LittleTable must scan through many rows with the given prefix to find the latest one, since rows are sorted within tablets by primary key, and the timestamp is the last of the primary key columns. As in any system, it is possible to carelessly construct queries that are not optimized for LittleTable's strengths, and we are not always diligent about tracking them down and correcting them.

The number of rows per insert likewise varies widely by application. Half of our tables see an average batch size of 128 rows or more, and the top 20% see batches of over 6,000 rows, but the bottom 20% insert only a single row at a time.

5.2.5 The Importance of Clustering

As discussed in Section 2.3.3, customers have a nearly insatiable demand for high-resolution historical data, even though they mostly query data from the recent past. By

clustering data by timestamp, LittleTable co-locates recent data, increasing the odds that it can satisfy most queries from the system page cache. By simultaneously clustering data by customer, network, or device, LittleTable ensures that it can satisfy queries for older, less-frequently queried data with mostly sequential disk reads.

Figure 10 quantifies the importance of such clustering. The upper, solid line in the figure shows a cumulative distribution of the oldest times requested by queries to a representative page in Dashboard over the six months ending with December, 2016. The lower, dashed line in the figure shows the distribution of TTLs across all tables in production use. While over 90% of requests are for data from the most recent week, Dashboard is able to retain data in most tables for a year or longer, removing old rows only when limited by the available disk space. Together, these two lines illustrate the importance of and the opportunity created by clustering time-series data in both dimensions.

6. RELATED WORK

Like a number of recent storage systems, LittleTable's design was inspired by the log-structured merge tree [19], which itself was inspired by LFS [21]. Other log-structured storage systems include Bigtable [8], bLSM [22], Cassandra [14], Diff-Index [24], HBase [5], LevelDB [1], LHAM [18], LogBase [25], Megastore [6], the PE file [12], Spanner [9], and TileDB [20]. Recent versions of both MongoDB [26] and MySQL [17] also include log-structured storage managers. These systems vary in whether they are centralized or distributed, whether they are run as a separate server process or linked into client processes as a library, whether their data model is relational or semi-structured, whether they vertically partition data by column, whether they contain a separate write-ahead log for recovery, and whether they support secondary indexes.

LittleTable is a relational database, run as a separate server process. By popular demand, its interface is SQL. LittleTable itself is not a distributed database; instead, we shard Dashboard at the application layer. Because data recently inserted into LittleTable can be recovered from the devices from which Dashboard originally gathered it, LittleTable does not use a separate write-ahead log, and it flushes data infrequently, thereby trading reduced durability for less disk write load. LittleTable does not support secondary indexes; to efficiently lookup data by multiple columns, Dashboard stores it more than once. Finally, as Dashboard generally accesses most columns of each row together, there is little motivation for LittleTable to vertically partition data.

LittleTable differs from most other log-structured systems in that it explicitly clusters data into two dimensions by partitioning data into tablets by timestamp and sorting data within each partition by primary key. While other systems' merge policies aim to combine as many tablets as possible, LittleTable groups tablets into time periods that are larger the further they are in the past, and LittleTable's merge policy explicitly avoids merging together tablets from different time periods in order to maintain a clustering by time. Recent versions of Cassandra include a similar policy [11]. LittleTable also uses a novel merge policy within time periods that maintains the logarithmic scaling of prior work while only merging tablets whose timespans are adjacent. In this way it maintains a clustering of data by timestamp even within a period.

An alternate approach to clustering data in more than one dimension is taken by TileDB, a log-structured array storage manager for scientific data. TileDB allows applications to control how data is grouped on disk in each dimension, how it is ordered within each group, and how groups are ordered relative to each other on disk. We believe this approach could be used to emulate LittleTable's clustering strategy except that the time periods covered by each group would not change size depending on how far in the past they were.

A number of indexing schemes for multiversion databases also partition data by both primary key and timestamp, including the Time-Split B-tree [15, 16] and the MD/OD R-tree [13]. Unlike LittleTable and other log-structured systems, however, these schemes do not take advantage of long sequential writes to maximize insert throughput.

LHAM [18] introduced the idea of moving older data in a log-structured system to write-once media. This approach is especially attractive for time-series data, where very old values are accessed infrequently but remain valuable, and we are considering using Amazon S3 or another cloud service as an additional backing store for old LittleTable data.

7. CONCLUSION

At the time we designed and built LittleTable, Meraki was a startup with only 13 software engineers, and LittleTable's design was thus heavily optimized for ease of implementation. Although we have occasionally made small enhancements, and we are currently investigating a bulk delete feature to simplify compliance with regional privacy laws, the same basic design continues to meet Dashboard's evolving time-series storage needs nine years later.

LittleTable's design succeeds primarily because it clusters data in two dimensions. By clustering rows by timestamp, LittleTable allows Dashboard to quickly display recent measurements without any penalty for retaining older history. Developers use this older data to allow infrequent searches further into the past or to re-aggregate old measurements and reveal new patterns. By further clustering each table by a developer-chosen key, LittleTable allows developers to optimize a table for the specific features they intend to build on top of it. High-quality features take time to code, from the firmware on devices to the JavaScript in the browser. A little thought about storage layout up front is a relatively small cost to pay for snappy performance down the line.

Central to both LittleTable's implementation simplicity and its high performance are the weaker consistency and durability guarantees it provides. These weaker semantics certainly present occasional challenges for developers, although we frequently find ad hoc solutions to common difficulties, such as the new flush command proposed in Section 4.1.2. Developing a more general, yet still performant, system with strong semantics would be a fruitful direction for future research.

8. ACKNOWLEDGMENTS

We thank Mehul Shah, our shepherd, and Sam Madden for their detailed feedback on this paper. We also thank Cliff Frey for his advice and feedback on the original LittleTable design, Varun Malhotra and Xiao Yu for their substantial contributions to the LittleTable code, and the many Meraki engineers who have helped improve LittleTable over the years, either as developers or users.

9. REFERENCES

[1] LevelDB. http://leveldb.org/.

[2] LZO real-time data compression library. http://www.oberhumer.com/opensource/lzo/.

[3] Round robin database tool. http://oss.oetiker.ch/rrdtool/.

[4] The virtual table mechanism of SQLite. https://sqlite.org/vtab.html.

[5] Welcome to Apache HBase. https://hbase.apache.org/.

[6] 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 CIDR*, 2011.

[7] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. In *Proceedings of USENIX ATC*, 2013.

[8] 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 *Proceedings of OSDI*, 2006.

[9] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, 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. In *Proceedings of OSDI*, 2012.

[10] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm. In *Proceedings of the International Conference on Analysis of Algorithms*, 2007.

[11] B. Hegerfors. Date-tiered compaction in Apache Cassandra. https://labs.spotify.com/2014/12/18/date-tiered-compaction/, Dec. 2014.

[12] C. Jermaine, E. Omiecinski, and W. G. Yee. The partitioned exponential file for database storage management. *The VLDB Journal*, 16(4):417–437, Oct. 2007.

[13] C. Kolovson and M. Stonebraker. Indexing techniques for historical databases. In *Proceedings of ICDE*, 1989.

[14] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.

[15] D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proceedings of SIGMOD*, 1989.

[16] D. Lomet and B. Salzberg. The performance of a multiversion access method. In *Proceedings of SIGMOD*, 1990.

[17] Y. Matsunobu. MyRocks: A space- and write-optimized MySQL database. https://code.facebook.com/posts/190251048047090/myrocks-a-space-and-write-optimized-mysql-database/, Aug. 2016.

[18] P. Muth, P. O'Neil, A. Pick, and G. Weikum. The LHAM log-structured history data access method. *The VLDB Journal*, 8(3-4):199–221, 2000.

[19] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.

[20] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson. The TileDB array data storage manager. In *Proceedings of VLDB*, 2017.

[21] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[22] R. Sears and R. Ramakrishnan. bLSM: A general purpose log structured merge tree. In *Proceedings of SIGMOD*, 2012.

[23] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *Proceedings of the USENIX Technical Conference*, 1995.

[24] W. Tan, S. Tata, Y. Tang, and L. Fong. Diff-index: Differentiated index in distributed log-structured data stores. In *Proceedings of EDBT*, 2014.

[25] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. LogBase: A scalable log-structured database system in the cloud. *Proceedings of the VLDB Endowment*, 5(10):1004–1015, 2012.

[26] T. Wolpe. MongoDB CTO: How our new WiredTiger storage engine will earn its stripes. http://www.zdnet.com/article/mongodb-cto-how-our-new-wiredtiger-storage-engine-will-earn-its-stripes/, Nov. 2014.

APPENDIX

As discussed in Section 3.4.1, LittleTable prevents inefficient merging with a simple algorithm. In this appendix we formalize that algorithm and prove its efficiency.

Let t_i be the tablets in a table, $|t_i|$ be the size of tablet t_i in bytes, and T be the size of the table. We number tablets such that for all i, the minimum timestamp in t_i is at most the minimum timestamp in t_{i+1}. In every iteration of the merge algorithm, LittleTable finds a consecutive sequence of merge candidates—consisting of the first two tablets t_i, t_{i+1} in the sequence such that $|t_i| \leq 2|t_{i+1}|$—and merges those tablets together. LittleTable repeats this process until there are no more tablets to merge.

Our first claim is that this algorithm results in a final number of tablets that is logarithmic in T. The proof is straightforward: if $t_1, \ldots, t_n$ represent the final set of tablets after merging is over, there is nothing to merge if and only if $|t_1| > 2|t_2| > 4|t_3| > \ldots > 2^{n-1}|t_n|$. Because the size of the table is equal to the size of the tablets, we have

$$\begin{aligned} T &= |t_1| + \ldots + |t_n| \\ &> 2^{n-1}|t_n| + 2^{n-2}|t_n| + \ldots + 2|t_n| + |t_n| \\ &\geq 2^n - 1 \end{aligned}$$

It follows then that $n = O(\log T)$.

Our second claim is that the number of times any one row is merged is at most logarithmic in T. Let m be the maximum number of times any particular tablet is merged during the merge algorithm, and consider any row in any tablet, before any merging has occurred, that is merged m times by the merge algorithm. In any given iteration of the

merge algorithm, there are two cases in which a tablet t_m containing the aforementioned row is merged:

- t_m is the first tablet (i.e., the one with the smallest index) in the sequence of merge candidates. This situation occurs either if t_m is larger than t_{m+1} but not double t_{m+1}'s size:

  $$|t_{m+1}| \leq |t_m| \leq 2|t_{m+1}|$$

  or t_m is smaller than t_{m+1}:

  $$|t_m| \leq |t_{m+1}|$$

  In either case, the merged tablet is at least 3/2 the size of t_m. It follows that this type of merge can only occur $O(\log T)$ times to t_m.

- t_m is not the first tablet in the sequence of merge candidates. Consider the first tablet t_ℓ in the sequence of merge candidates. If $\ell = 1$, then t_m will become part of the first tablet after this merge, and this type of merge will never happen again. Otherwise, if $\ell \geq 2$, all tablets before t_ℓ cannot be eligible to be merged, i.e., $|t_i| > 2|t_{i+1}|$ for all $i \leq \ell - 1$. As in the first claim, there can only be $O(\log T)$ such tablets to the left of t_ℓ; therefore, this type of merge can only occur $O(\log T)$ times to t_m.

Note that, while the described merge policy only considers and merges the first two tablets t_i, t_{i+1} that satisfy $|t_i| \leq 2|t_{i+1}|$, the logarithmic bounds and proof continue to hold even if LittleTable merges any number of tablets that immediately follow t_{i+1}, regardless of their sizes.