Fast Ingest and Query Performance in PostgreSQL with SplinterDB Indexes

Aditya Gurajada Gabriel Rosenhouse Carlos Garcia-Alvarado Rob Johnson
VMware

Abstract

Relational databases make extensive use of B-tree indexes to accelerate query performance. B-Trees offer excellent point-query performance but inserting data into a large B-tree can be costly, especially for a workload of random inserts. In fact, high-velocity tables are often not indexed at all due to the performance limitations of B-tree insertions.

This paper describes how SplinterDB, a novel, open-source, write-optimized key-value store developed at VMware, can be used as an index within a relational database, such as PostgreSQL, to increase ingestion speed without sacrificing query performance.

In our performance evaluation, we show that a SplinterDB-indexed table can ingest data up to 24× faster than a B-tree-indexed table. In fact, SplinterDB ingests data half as fast as a table with no index at all, whereas B-tree indexes can impose a 36× slowdown compared to an unindexed table. Furthermore, SplinterDB matched the point-query performance of Postgres's B-tree and outperformed the builtin B-tree by 50% on large-range queries, although it was about 3× slower on short-range queries.

Our results suggest that a SplinterDB-based index could become a compelling alternative to B-tree indexes for supporting higher ingestion rates like those needed for real-time applications in relational databases like PostgreSQL.

1 Introduction

Databases have typically been optimized for accelerating queries, and since their inception, they have maintained indexes to support fast queries. One of the most common indexing structures is the B-tree [8]. B-Trees are the most commonly used indexing structure in PostgreSQL [4], a powerful, open-source object-relational DBMS engine supporting different index types. B-Tree indexes are widely used indexes by customers because they deliver excellent query performance for both point lookups and range scans. Often, Btrees are preferred as an access method to return order-preserving result sets, avoiding extra sort costs.

Unfortunately, an often-heard customer complaint is that the ingestion rate suffers on huge tables with large B-tree indexes. The larger the table, the slower ingestion gets. For example, some PostgreSQL customers ingesting a large volume of data on a table with a B-tree have reported a reduction in their ingestion rate larger than 10x, which compromises their access to fresh data, business operations, and SLAs.

Recent work has shown that it is possible to build indexes that support much faster insertions than B-trees. SplinterDB [9] is an embedded standalone key-value store based on the mapped B$^{\varepsilon}$-tree data structure designed for high-ingestion rates and good query performance. SplinterDB was designed for high-performance when dealing with small key-value pairs in resource-constrained environments. In direct benchmarks of small key-value pairs, SplinterDB provides 2-10× ingestion speedups over existing key-value stores, such as RocksDB, that are already optimized for insertions. SplinterDB is particularly efficient at handling high ingestion rates of randomly ordered keys (e.g., the customer's SSN), which is where B-trees are particularly weak.

In this work, we empirically demonstrate that SplinterDB can transform insertion performance in PostgreSQL. We show that, for random insertion workloads, switching from B-tree indexes to SplinterDB indexes can increase throughput by a factor of over 20× due to the greater I/O efficiency of insertions into SplinterDB. SplinterDB insertion performance remains steady even as the database grows much larger than RAM, whereas B-tree insertion performance falls off a cliff. But, even when the database fits in RAM, SplinterDB is faster than the B-tree for random insertions. PostgreSQL's B-tree outperforms SplinterDB insertions in only one case—single-threaded sequential insertions—and only by a small margin of 10%.

We also show that SplinterDB insertion throughput scales better than PostgreSQL's B-tree with increasing threads. For example, SplinterDB outperforms the B-tree on sequential insertions (B-tree's best case) with 2 or more threads.

We also show that, in all our workloads, inserting into a table with a SplinterDB index is at least half as fast as inserting into a table with no index at all. In contrast, adding a B-tree index can slow down inserts by over 30×.

For queries, we show that a SplinterDB-based index in PostgreSQL can match the B-tree point-query performance and outperform the B-tree for large range queries. The one place where the B-tree clearly outperforms SplinterDB is short range queries, where SplinterDB can be about 3× slower than the B-tree. However, keep in mind that, for a high-velocity table, the choice that a database adminiatror may face is not between a SplinterDB index or a B-tree index, but between a SplinterDB index or no index at all, for which any range query would have to scan the entire table. In that case, SplinterDB would offer orders-of-magnitude faster small range queries.

In summary, we show that adding SplinterDB as an indexing method in PostgreSQL can offer order-of-magnitude speedups for many workloads, with little or no downside. But, going further, we believe that SplinterDB can fundamentally change how customers use PostgreSQL. A fast index, such as SplinterDB, makes indexing cheap. This means users can maintain indexes for high-velocity tables for which B-tree indexes may not be practical. Thus, SplinterDB indexes have the potential to accelerate queries in databases that just cannot afford the maintenance costs of B-tree indexes. And SplinterDB indexes have the potential to eliminate the need for batch loads, making application code simpler and faster and enabling database queries to return results on the latest data efficiently.

In the rest of this paper, we will make the case that developing a novel index based on the open-source SplinterDB Key-Value store for a commercial-grade open-source DBMS engine, like PostgreSQL,

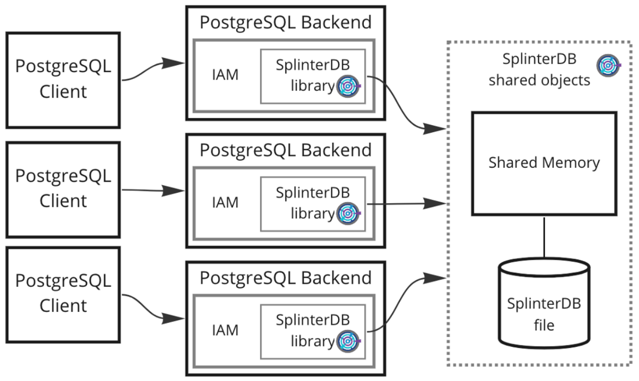


Figure 1: Architecture

could upgrade relational databases to meet the requirements to support applications that need to ingest and query high velocity data.

2 A SplinterDB-Based Index

In this section we describe the challenges to integrating SplinterDB into PostgreSQL and the techniques we use to overcome those challenges. There are two main problems we needed to tackle during the integration:

- Physical integration. PostgreSQL uses a single-threaded multiprocess model augmented with explict shared memory, whereas SplinterDB was written for a multi-threaded model in which all memory is implicitly shared.
- Logical integration. PostgreSQL uses a relational model whereas SplinterDB supports a simple key-value model.

2.1 Physical Integration

PostgreSQL starts with an initial process, called the ***postmaster***, which creates shared memory segments on start-up and begins listening on a network port. All PostgreSQL shared data structures, such as the buffer cache, metadata, lock structures, etc., are allocated from this shared segment. For every database client connection, the postmaster forks a new process called the "backend" dedicated to that client. Each backend retains references to the shared memory established on start-up. Thus, all concurrency and synchronization across multiple clients is managed through shared data structures allocated from the shared segment.

SplinterDB, on the other hand, is a multithreaded library that manages its memory using `mmap`, `malloc`, and `free`, i.e. it assumes that all memory is shared among all threads. Furthermore, it supports launching background threads to perform tree maintenance tasks, whereas PostgreSQL processes are generally single threaded.

Figure 1 summarizes the architecture of how we integrated the SplinterDB library with `postgres`.

Memory Allocation: We enhanced SplinterDB's memory allocation to support a shared-memory model. With our changes, a SplinterDB instance can now be configured to allocate a shared memory region at start-up, and all SplinterDB shared data structures are allocated from this segment. This brings SplinterDB's memory allocation and sharing assumptions in line with PostgreSQL's. We were aided in this effort by the fact that all memory allocation had already been abstracted in the SplinterDB code base.

Note that we did not integrate SplinterDB's page cache with any of the caches in PostgreSQL. Thus, in our prototype, PostgreSQL's tables and B-tree indexes use PostgreSQL's internal cache and the SplinterDB indexes use SplinterDB's cache. However, in our experiments, we always ensured that the total amount of cache (across all caches) was the same, and we further limited total process memory using cgroups to ensure a fair comparison of the different indexing schemes (see Section 3 for details on the experimental setup).

Background Threads: SplinterDB's "background thread" functionality was exposed for library users. Since SplinterDB supports background threads that can asynchronously handle time-consuming maintenance work, such as compacting trees and building filters. This feature is intended to reduce tail latencies on write operations since application-created "foreground threads" (i.e., the client connection performing the inserts) can then skip such maintenance work.

Multi-Process Model: SplinterDB's threading support was enhanced to integrate with PostgreSQL's multi-process model. The existing pthread-based synchronization primitives were changed so that instead of being process-private, they could now be shared across OS processes. This enabled multiple Postgres backends, each a single thread in a separate process, to operate concurrently and alongside any SplinterDB background threads. Furthermore, SplinterDB's asynchronous IO context management was changed from global to per-thread (and thus per-process) so that each PostgreSQL backend could drive its own IO of data managed by SplinterDB.

Linking, startup and shutdown: The SplinterDB library is statically linked with the `postgres` binary.

We exposed several SplinterDB-specific configuration parameters via PostgreSQL's configuration file. This allowed us to enable/disable SplinterDB support and to tune the size of SplinterDB resources (cache, shared memory, device size, etc.) through normal PostgreSQL configuration.

The lifecycle of the SplinterDB-enhanced PostgreSQL is simple: On server start-up, when SplinterDB is enabled, the PostgreSQL's "postmaster" process creates (or opens an existing) SplinterDB device and sets up the required shared memory and other metadata memory for use by SplinterDB. Then each PostgreSQL forked backend will register itself as a thread within SplinterDB. When a client disconnects, the corresponding backend deregisters itself from SplinterDB before exiting. When the postmaster shuts down, it closes the SplinterDB instance.

2.2 Schema-Level Integration

PostgreSQL exposes an Index Access Method (IAM) API that enables the creation of custom index types by implementing a documented interface [2]. All indexes, including built-in BTree indexes, conform to this interface.

SplinterDB, on the other hand, provides a simple key-value interface. Keys and values are opaque blobs of bytes with user-defined comparison and hashing functions.

We implemented functional support for indexing single-column indexes only on integer columns are allowed. Although restrictive, this capability is sufficient to perform a wide range of performance benchmarking to evaluate the two indexing schemes. For such indexes, the maximum size of a KV-pair is 10 bytes. The integer column becomes the key in SplinterDB, and we store the tuple ID as the associated value.

The primary semantic gap between PostgreSQL and SplinterDB is around uniqueness checking of inserted keys. SplinterDB supports only “blind” inserts, i.e. an insert of a new key-value pair with the same key as an existing entry simply overwrites the old entry. SplinterDB does not return whether an insert results in an overwrite of an older key-value pair or not.

PostgreSQL IAM, on the other hand, supports two types of indexes: unique and non-unique. Unique indexes are expected to return an error if the user attempts to insert a record with the same key as an existing tuple. Thus, SplinterDB currently supports only PostgreSQL’s non-unique index mode.

Unfortunately, an index’s uniqueness mode also affects how PostgreSQL makes queries to that index, and this difference has significant performance ramifications in SplinterDB. For non-unique indexes, PostgreSQL uses the index’s iterator method, even for equality queries. For unique indexes, PostgreSQL uses a special point-lookup API to the index. SplinterDB supports much faster point queries than iteration because it maintains quotient filters [9] internally, which are used to narrow down searches during point queries, enabling most lookups to complete in a single I/O.

To get around this mismatch, during our point-query benchmarks, we declare SplinterDB to be a unique index to PostgreSQL. Note that we do this only for the point-query benchmarks—all our insertion and range-query benchmarks were executed with SplinterDB in the non-unique mode.

3 Evaluation

Our evaluation of the SplinterDB index was focused on two major items: One, quantifying the performance of bulk inserts on large datasets, in which BTrees struggle. Two, performance of single-table point-lookups and range scans. The design point was to study the performance of these usages when the data size far exceeds available memory (cache) size.

Our objective was to understand the overheads of maintaining during bulk loads an index (B-tree or SplinterDB) as compared to the raw throughput of such inserts into a table with no index. We also wanted to study any degradation of ingest performance with increasing table sizes. For reads, the objective was to compare the query performance of single-table point-lookups and short- and long-range scans between a B-tree or SplinterDB index as an access method.

We did not include update of non-index key columns in our workload as the performance of such operations is similar to that of read performance, insofar as the SplinterDB index access is concerned. For SplinterDB, update of an index key-column or delete of data rows both essentially result in the insertion of a new tuple(s). Hence, we expect that the results of comparing insert performance will be a good indicator for how these other operations would compare.

3.1 Hardware and Server Configuration

We used the following setup for all experiments described below. Experiments were run on an AWS i4i.16xlarge instance which has 64 logical processors, 378 GiB memory and 4 NVMe SSDs available as local storage. The SSDs were each formatted with the ext4 filesystem. PostgreSQL 15 was deployed with the SplinterDB-based Index Access Method described above. PostgreSQL and SplinterDB data was stored on the ext4 filesystem of one of the SSDs, so that all IO bandwidth is shared by either form of the index structure along with IO for the data pages.

PostgreSQL was launched within a Linux cgroup that enforced an 8GiB limit on the combined memory usage of the server and all of its child-processes, including backends. This cgroups memory-limit was used to simulate the common use-case where the database size far exceeds the available memory on the host, without needing to test with terabyte-scale workloads.

The PostgreSQL buffer cache and SplinterDB cache were sized to fit comfortably within this limit. In all experiments using SplinterDB, its shared memory segment was configured to 1.5 GiB, to avoid failures due to insufficient shared memory. In practice, we observed that SplinterDB’s max shared memory usage was no more than 80% at the end of each experiment.

We performed large data-ingestion throughput experiments for three scenarios: no index on the table, BTree index and SplinterDB index. When testing without an index, or with the BTree index, SplinterDB was disabled (i.e., no shared memory is allocated for SplinterDB) and the PostgreSQL buffer cache was given 6 GiB of memory. When testing the SplinterDB index, it was configured with a 5 GiB cache (in addition to the previously mentioned shared memory segment) and PostgreSQL was configured with an equivalently reduced 1 GiB buffer cache. Note that in both configurations, the same 6 GiB is being distributed across the two caches, and everything is run under a cgroups of 8 GiB.

The Postgres database was created with its default 8KB page size while SplinterDB was created with its default 4KB page size. This may have affected the IO characteristics of our workloads, especially for random insertions, however, we did not try to fine-tune this configuration as both systems were using their default page sizes.

3.2 Workload

All experiments ran on a simple table with a 4-byte integer column and a fixed-length char(10) filler column. The index is defined on the integer column. Logging was enabled for ingestion performance experiments to ensure that we include the overheads of run-time logging needed to support crash recovery.

Data-loading experiments used a custom “workload generator” program written in Go. The workload program was run on the same host as PostgreSQL. It opens a configurable number of concurrent client connections to Postgres and bulk-loads, in batches of 1000 rows, synthetic data via the PostgreSQL `COPY FROM <stdin>` statement. When client concurrency was >1, the total row count was partitioned amongst the clients. The use of the COPY statement, which is a highly optimized batch load PostgreSQL interface, and the use of streaming, in-memory data generation using `FROM <stdin>` helps minimize overheads in the client, socket, and PostgreSQL frontend. In practice, we observed that even with the clients and server running on the same host, the workload generator was able to drive un-indexed tables to much higher throughput than either index, so we were confident that the client program was not the limiting factor in evaluating the performance of the indexes.

The integer column value was generated in one of two ways:

- Sequential integers: When inserting N rows using C concurrent clients, each client generated its sequential keys from an assigned interval of size (N/C) non-overlapping key ranges.

This simulates the ingestion of streaming monitoring data in, say, timestamp order.
- Uniform-Random integer samples: These used a pseudo-random generator with a fixed seed value to ensure that the same sequence of random values is used for all experiments. There was no partitioning of the key space, so at higher row counts, concurrent clients almost certainly inserted some duplicate keys. We did not monitor duplicate key count.

For both cases, the char(10) column was filled with base64-encoded random bytes.

For ingestion performance and range-scan experiments, a non-unique index was created on the integer column.

Point-lookup performance experiments were performed as follows. We loaded a billion sequential keys to the test table defined with a UNIQUE index. SplinterDB does not enforce uniqueness checks, but BTree indexes do. So, by loading sequential data, we avoid running into uniqueness check failures. We did not measure the ingestion measure performance in this load-phase, and just used this load to run point-query performance experiments. We measured scalability of lookup throughput (in terms of number of lookups / sec) across number of clients. Each client performed 1 million random point lookups out of the 1B keys, so each lookup was expected to find a row.

For range scan performance, we loaded a billion sequential keys to the test table defined with a non-unique index. We used a single client to measure range-scan performance. For different ranges of lengths from 10, 10^2 , 10^3 ... 10^7 (10M) keys, a million queries were executed using a random value for the lower-end of the range specified by a `BETWEEN` clause. The lower-bound of the range was chosen randomly from the billion keys inserted. Depending on the lower-bound and the range size, it is possible that not all keys in the range would be found. We did not measure or check whether the query returned the specified number of keys in the selected range.

For both read performance queries, to avoid the overheads of per-statement query optimization, we used PREPARE'ed statements with dynamic parameters to specify the lookup-value or the values limiting the range.

An "experiment" constitutes a test run with: a choice of index type (no index, BTree or SplinterDB index), a key-generator (sequential / random) and a chosen number of concurrent clients. For each experiment, PostgreSQL was re-started, tables and indexes recreated and, when applicable, the Splinter instance (i.e., its device, shared memory, cache etc.) was recreated. This ensures that the database and all caches were in the same initial state for all workloads. Execution times were measured from the workload generator client program.

4 Benchmarking Results

This work was originally motived by anecdotal reports that bulk-loading random data into a table with a B-tree index was roughly 10× slower than inserting into an unindexed table. Therefore, we first aimed to reproduce that claim, and then to determine whether a SplinterDB-based index might perform better.

For the three test cases of "no index", "B-tree " and "SplinterDB", we ran a test which created from an empty table with the specified index (if any), and then used our workload generator to load 1 billion rows in batches of 1000, with either random or sequential key order, into the table. For each run, the workload generator reported samples showing the current table size and an "instantaneous" throughput (averaged over the prior 10 seconds). This helps us understand how the ingestion throughput changes with increasing table size for different index types.

Results are shown in Figure 2. As the test proceeded, the table grew (horizontal axis). Average insert throughput over a 10-second window is reported along the vertical axis.

4.1 Single Client Inserting Data

Figure 2a shows bulk-load performance for a single client inserting sequential keys. As B-trees are highly optimized for sequential keys, the throughput remains consistent. SplinterDB is not specifically optimized for this case but performs nearly as well in average throughput. However, the inserting thread also has to perform tree management, such as compaction, flushing etc., which creates variability in the throughput.[1]

Figure 2b shows bulk-load performance for a single client inserting uniformly random keys. When the test begins from an empty table, the B-tree instantaneous throughput is roughly half that of the no-index load. As the table size grows, B-tree index throughput drops dramatically, ending with an instantaneous throughput (16k rows/sec) that is roughly 30× slower than when inserting without an index (491k rows/sec). This sharp drop in performance is presumably due to the table and index growing beyond the configured memory limits (see Section 3), causing the B-tree inserts to thrash the PostgreSQL buffer cache and/or OS page cache.

On the other hand, throughput of random inserts into the SplinterDB index starts higher than for the B-tree index, and decays only marginally as the table grows exponentially (note logarithmic scale on the horizontal axis). Again, the SplinterDB throughput fluctuates as the inserter threads occasionally perform tree maintenance.

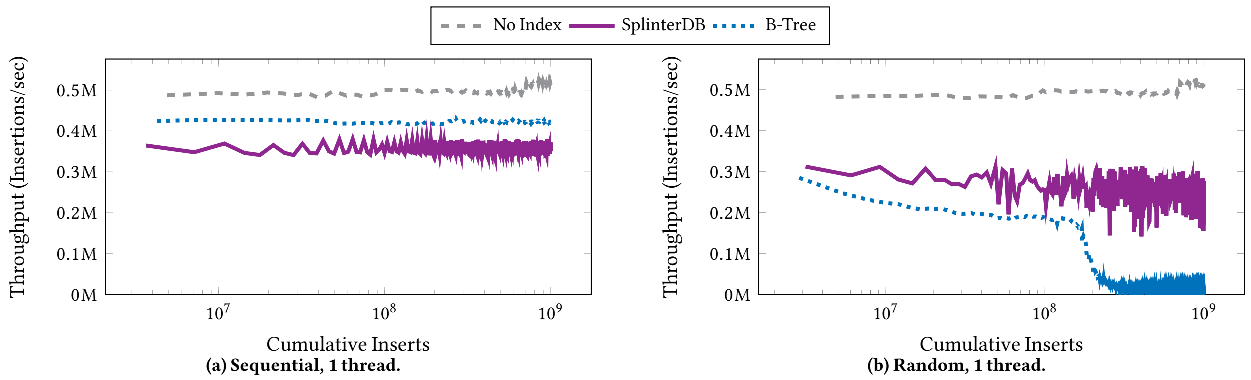
4.2 Multiple Clients Inserting Data Concurrently

Relational databases are designed for concurrent access by many clients and, users often parallelize bulk-loads of large data sets. To quantify the insert performance of indexes in the face of multiple concurrent clients, we repeated the above experiments using a varying number of workload-generating client connections. The same total amount of data was loaded every time (1 billion rows), but this load was split equally across the varying number of clients to study the ingestion performance as a function of concurrency.

Figures 2c and 2d show bulk-load throughput as a function of table size for 32 concurrent clients. Here, SplinterDB's optimizations for high concurrency allow its index to consistently out-perform the B-tree index for both sequentially generated keys (Figure 2c) and random keys (Figure 2d). Again, the B-tree performance drops dramatically at larger table sizes, while SplinterDB performance decays only a small amount as the table grows several orders of magnitude.

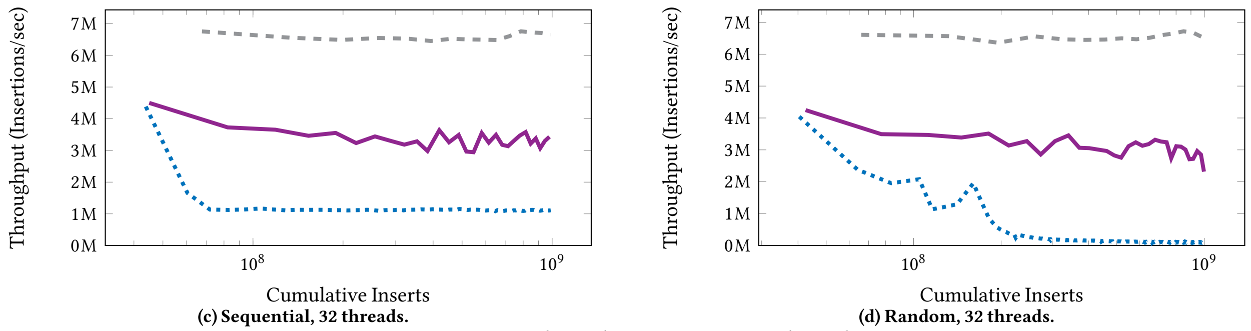
We also repeated the same experiments at other concurrencies between 1 and 32 concurrent clients. Overall average throughput is summarized in Figure 3 for random and sequential . At lower concurrency levels, insertion throughput of sequential data into SplinterDB is close to or slightly better than the throughput into B-tree. At higher concurrency levels, SplinterDB throughput scales

[1]We have preliminary results showing that enabling SplinterDB's background threads for tree maintenance dramatically reduces the variance in throughput seen by foreground threads.



(a) Sequential, 1 thread.

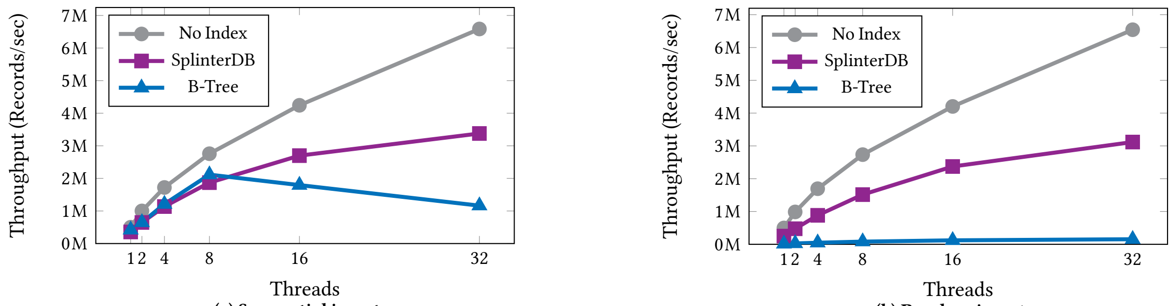
(b) Random, 1 thread.



(c) Sequential, 32 threads.

(d) Random, 32 threads.

Figure 2: Insertion throughput over time. Higher is better.



(a) Sequential inserts.

(b) Random inserts.

Figure 3: Insertion throughput scaling. Higher is better.

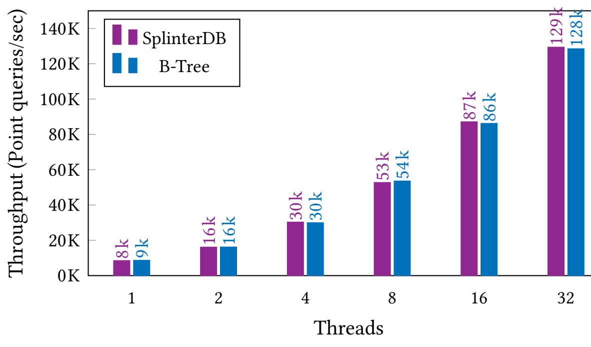


Figure 4: Point query throughput. Higher is better.

up while B-tree throughput tapers off. For insertions of random data, SplinterDB outshines B-trees by a wide margin, delivering up to a 24× performance gain for 32 concurrent clients.

4.3 Point Queries

We now compare point-query performance with SplinterDB indexes and B-tree indexes across a range of concurrency levels. Each thread performs 1 million queries for random keys in a table with 1 billion records. All queries are for keys that are present in the database.

Figure 4 shows that, across all numbers of concurrent threads, query throughput with SplinterDB indexes is virtually identical to query throughput with B-tree indexes. B-Tree queries are asymptotically optimal in the external-memory atomic key comparison model [6], so matching the query performance of B-trees is impressive.

Overall, these results suggest that increased insertion throughput does not have to come at the cost of decreased point query throughput.

4.4 Range Queries

To measure range query performance, we had a single thread perform random range queries for records with primary key

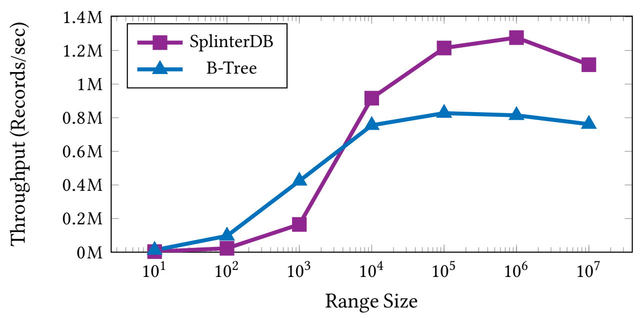


Figure 5: Scan throughput. Higher is better.

between two integers, x and $x + \ell$. The starting point x was chosen uniformly randomly, and we varied the size ℓ of the range from 10 to 10^7. Note that not all integers were present in the database, so a query for a range of length ℓ could return somewhat less than ℓ records (for x values closer to the maximum key inserted). Throughput is measured in terms of returned records per second.

Figure 5 shows throughput as a function of range size. For small ranges, the B-tree index scan was about 1-5 ms faster than SplinterDB which resulted in about 3× greater throughput than the SplinterDB index. For large ranges, (> 10000 keys), SplinterDB was up to 50% faster than the B-tree index.

These results are consistent with the theory behind B-trees and SplinterDB. In the external-memory model [5] where each page can hold B records, a B-tree range query returning k records from a database of N records will need to access $O(k/B + \log_B N)$ pages ($\log_B N$ pages to find the first record in the range, and k/B pages to read the k records). In SplinterDB, the same range query would access $O(k/B + \sqrt{B} \log_{\sqrt{B}} N)$ pages. Thus Splinter has a roughly $\sqrt{B}$ higher startup cost of finding the first item in the queried range, but it is just as efficient at scanning through the subsequent k items.

In fact, SplinterDB is even more efficient than the B-tree at the scan phase because it can use larger I/Os than a B-tree. SplinterDB groups every 32 logically consecutive pages into ***extents*** that can be read using a single I/O. For B-trees, however, the choice of page size entails a trade-off between range-query performance and the performance of other operations [6]. This is why B-trees tuned for OLTP workloads typically use very small nodes (e.g. less than 16KiB), which is too small to amortize per-page I/O overheads during range queries. As a result, SplinterDB is able to outperform the B-tree index on large range queries, where the startup overhead is immaterial.

5 Related Work

BTree indexes have been part of the design of PostgreSQL since its seminal papers in the 1980s [14, 15]. Nowadays PostgreSQL has several types of indexes such as BTree, Hash, GisT, SP-GisT, BiN and BIN that have different capabilities, performance, and are useful in different type of scenarios [3]. Despite this, the focus remains on BTrees due its wide applicability and efficient range querying capabilities. The dramatic slowdown in BTree insert performance at large scales has been observed across the literature and novel BTree variations have been proposed to improve its write performance by algorithmic modifications [7] or adaptation to novel hardware [10].

Prior work as explored using write-optimized data structures to accelerate RDBMS performance, including LSM-based solutions [11], including well-known production grade implementations such as MyRocks [12]. Prior challenges and efforts to enhance PostgreSQL performance with alternatives to BTree indexes were attempted by Jin [13] and Knizhnik [1].

6 Future Work

We plan to extend the support for creating single-column indexes on data types such as VARCHAR and date-time oriented types such as DATE, TIME, and TIMESTAMP. Moreover, we plan to support multi-column indexes and then multiple such indexes defined on multiple tables, all mapped to the same SplinterDB instance. We also plan to study the behavior of executing mixed read-write workloads and the concurrency behavior (in terms of index blocking or conflicts) as seen using B-Tree v/s SplinterDB indexes.

The goal of these enhancements is (a) to understand better and overcome the challenges in tightly integrating a key-value storage engine, such as SplinterDB, with a mature RDBMS engine, and (b) to gather performance measurements for "real-life" mixed read-write workloads.

Many other enhancements would be required, including a more mature shared memory allocator and integrating the logging and recovery mechanisms between PostgreSQL and SplinterDB. Furthermore, packaging this index access method as a separately compiled dynamically-loaded SplinterDB+ library would, perhaps, simplify release-management and deployment of PostgreSQL servers.

7 Conclusion

This paper showed how a SplinterDB-based index could help a system like PostgreSQL support ingesting data at a rate that will help real-time applications. In particular, this novel index performed in the same order of magnitude during ingestion as not using an index.

Our experiments demonstrated that a SplinterDB-based index for PostgreSQL outperforms the standard BTree index across a wide range of workloads, with especially large gains for larger tables and higher concurrency. For data ingestion of random data, the gains were in the order of 21x, and using a SplinterDB-based index reduced the ingestion index overhead from 36x to 2x. Regarding reads, our results show a point-query performance on par with the Btree and a performance improvement of 50% for large scans. The Btree index performed better for short-range scans.

We foresee a path where SplinterDB can be fully integrated into all relational databases to offer an additional index type offering for very high ingestion throughput for sequential and random data loads and enhanced scan/query performance over conventional BTree indexes. Alternatively, a SplinterDB-based index could be a compelling alternative for analytical queries.

References

[1] Benchmarking LSM-Trees for Postgres. PostgreSQL's Message Board. https://www.postgresql.org/message-id/315b7ce8-9d62-3817-0a92-4b20519d0c51%40postgrespro.ru.
[2] PostgreSQL Index Access Methods Interfaces. https://www.postgresql.org/docs/current/index-functions.html.
[3] PostgreSQL Indexes. https://www.postgresql.org/docs/current/indexes.html.
[4] PostgreSQL: The world's most advanced open source database. https://www.postgresql.org/.
[5] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. Commun. ACM, 31(9):1116–1127, 1988.
[6] Michael A. Bender, Alex Conway, Martin Farach-Colton, William Jannen, Yizheng Jiao, Rob Johnson, Eric Knorr, Sara McAllister, Nirjhar Mukherjee, Prashant Pandey, Donald E. Porter, Jun Yuan, and Yang Zhan. Small refinements to the DAM can have big consequences for data-structure design. In Christian Scheideler

and Petra Berenbrink, editors, The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019, pages 265–274. ACM, 2019.

[7] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. utree: a persistent b+-tree with low tail latency. Proceedings of the VLDB Endowment, 13(12):2634–2648, 2020.

[8] Douglas Comer. The ubiquitous b-tree. ACM Comput. Surv., 11(2):121–137, 1979.

[9] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard P. Spillane, Amy Tai, and Rob Johnson. Splinterdb: Closing the bandwidth gap for nvme key-value stores. In Ada Gavrilovska and Erez Zadok, editors, 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020, pages 49–63. USENIX Association, 2020.

[10] Thomas Hardjono, Tadashi Araki, and Tetsuya Chikaraishi. Improving the performance of enciphered b+-trees. IEICE transactions on fundamentals of electronics, communications and computer sciences, 76(1):104–111, 1993.

[11] Chris Jermaine, Anindya Datta, and Edward Omiecinski. A novel index supporting high volume data warehouse insertion. In VLDB, volume 99, pages 235–246, 1999.

[12] Yoshinori Matsunobu, Siying Dong, and Herman Lee. Myrocks: Lsm-tree database storage engine serving facebook's social graph. Proceedings of the VLDB Endowment, 13(12):3217–3230, 2020.

[13] Jin Shichao. Introducing lsm-tree into postgresql, making it as a data gobbler. In Talk from PostgreSQL conference (PGCON-2020).

[14] Michael Stonebraker. The case for partial indexes. ACM Sigmod Record, 18(4):4–11, 1989.

[15] Michael Stonebraker and Lawrence A Rowe. The design of postgres. ACM Sigmod Record, 15(2):340–355, 1986.