Niv Dayan

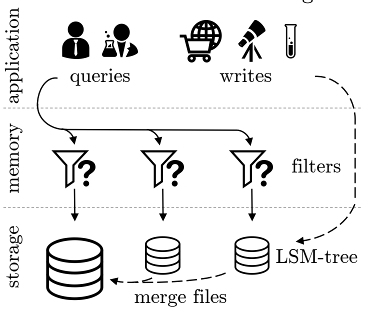
Research Statement
Scaling Storage Engines for 100x Big Data

Our society is creating and storing exponentially increasing amounts of data. What is often less thought of are the storage engines that maintain this data and facilitate the extraction of knowledge from it. In the late-2000s, a new class of storage engines emerged that prioritize the efficiency of ingesting new data. They include Google's BigTable, Amazon's DynamoDB, Facebook's RocksDB, as well as Apache Cassandra and HBase. These engines have become indispensable for a wide range of applications, including cloud storage, blockchain, machine learning, etc. Nevertheless, a lingering problem is that their performance deteriorates with respect to the amount of data they store. This, in turn, causes applications running on top to have to spend disproportionately more time, energy, and hardware in order to, say, transact on a blockchain, train a deep learning model, or add a photo to the cloud.

My research allows for such storage engines to function more efficiently as the big data that they store continues getting bigger. It does this by revisiting an age-old question in computer science: how to scale core dictionary data structures such that their performance deteriorates as slowly as possible with respect to the data size. The work constructs mathematical models of how all constituent data structures within a modern storage engine interact to define the engine's performance properties. It then uses these models to systematically identify and address performance bottlenecks. This involves co-designing traditional data structures in novel ways that lead to scalability improvements, as well as introducing new data structures with more scalable properties. This document summarizes this work as well as plans for carrying it forth.

LSM-Tree. Modern storage engines streamline new application data into storage (disk or SSD) as small sorted files, which are later merged into larger sorted files. This organization is known as a log-structured merge-tree (LSM-tree) and it is illustrated in Figure 1. With LSM-tree, merging files more eagerly creates higher overheads for writes but allows for faster queries as there are fewer files to search. This trade-off is controlled by a *compaction policy*, which dictates which files to merge under which conditions.

Figure 1



Filters. Each file of an LSM-tree is assigned a "filter" in fast memory (DRAM chips) as shown in Figure 1. A filter is a compressed approximate representation of a file that takes up little space. Filters can be quickly searched to rule out files that do not contain the target data. Thus, they eliminate unnecessary accesses to slower storage. The more space a filter is assigned, the more accurate it becomes thus allowing queries to rule out the file with a higher probability. An LSM-tree implements a *filtering policy* to decide how much memory to assign each filter. Together, the filtering and compaction policies govern a three-way trade-off between the overheads of queries, writes and space.

Overarching Question. As with most tree structures, LSM-tree's query and write overheads grow logarithmically with respect to the data size. The intuition is that as the data grows, the number of files that must be queried and merged grows too. In our current era of exponential data growth, logarithmic scalability implies linearly increasing overheads with respect to time. The outcome is rapidly deteriorating performance. While it is possible to offset one of the overheads growing by another (e.g., by merging more eagerly or allocating larger filters to prevent query overheads from increasing), it is impossible with existing designs to keep all three overheads steady at the same time as the data grows. This begs a question: is it possible to achieve sub-logarithmic query and write costs for an LSM-tree, all without hurting space?

Step 1: Scaling Queries. Our first work on this topic shows that existing storage engines assign memory to the different filters in a manner that does not minimize the overall rate at which the filters make mistakes (false positives). These false positives, in turn, result in unnecessary storage accesses that amplify query costs. The work frames a constrained optimization problem where the objective is to minimize the sum of false positive rates across all filters by allocating a finite memory budget across them. It shows that optimal solution is for the filtering policy to assign relatively more memory to filters of smaller files [1, 2]. The intuition is that since smaller files contain a small fraction of the overall data, their false positive rates can be drastically reduced by allocating only a slightly higher proportion of the overall memory budget to them. The approach is shown to reduce query cost from $O(\log N)$ to $O(1)$ storage accesses by causing the sum of false positive rates to converge with respect to the number of files. This work received a citation as one of the best four papers of SIGMOD 2017.

Step 2: Scaling Writes. Our next effort was to reduce and better scale write overheads to allow applications to ingest new data more efficiently. A naïve solution to this problem is tuning the compaction policy to merge files less eagerly so that less work has to be done to organize new data. The problem with less eager merging, however, is that it leads to having more files in the system for queries to have to search, or it requires having larger filters to keep queries fast. The research challenge is therefore to improve write costs without hurting query speed or space. To address this problem, we observed that smaller files in the LSM-tree are merged more frequently and are therefore responsible for most compaction overheads. At the same time, as discussed in Step 1, smaller files have fewer entries, and so it is feasible to drastically reduce their filters' false positive rates by allocating slightly more of the overall memory budget to them. Hence, we co-designed the compaction and filter policies to merge smaller files less eagerly while assigning their filters relatively more memory [3, 4]. The solution is shown to reduce write overheads from $O(\log N)$ to $O(\log \log N)$ storage accesses while keeping query and space overheads unharmed. In addition, this work explores the compaction design space for LSM-tree and proposes a parameterized generalization that allows assuming different trade-offs between query, write and space costs for different application workloads.

Step 3: Scaling Memory Accesses. A traditional assumption in the design of storage engines is that memory access (i.e., on DRAM) takes negligible time relative to storage access (i.e., on disk or SSD). Modern SSDs, however, call this assumption into question as they are rapidly getting faster. For an LSM-tree with tens to hundreds of files, the cumulative time taken to access all filters in memory can exceed the time taken to fetch the target data entry from a high-end storage SSD. Hence, we identified a new research challenge: preventing filtering from becoming a performance bottleneck as SSDs continue to evolve. Towards this end, the work proposes a unified filter over the whole LSM-tree that takes less time to query. This filter is structured as a compact hash table, which maps from an approximate representation for each key in the LSM-tree to the ID of the file whereon that key resides. To keep the memory requirement fixed, we compress the file IDs within the hash table using Huffman coding. The new filter brings memory access cost from $O(\log N)$ to $O(1)$ for queries while keeping all other overheads unharmed [5]. I pursued this work while working at Pliops, a startup working on a hardware-accelerated storage engine [6].

Step 4: Filtering Range Queries. Filters in modern storage engines support *point* queries, which take as input one key and expect one data entry as output. They do not support *range* queries, which take as input a range, say one to ten, and expect as output all data entries whose keys fall in-between. Hence, range queries are generally slow as they have no means of ruling out files that do not contain relevant data. To address this problem, we introduced a range filter, which indexes every prefix of every key in a file and processes queries by checking if any of the relevant prefixes covering the target range are present [7]. This new range filter brings as much as a 40x improvement in throughput relative to standard filters and a 2-5x improvement relative to the best competition from the literature.

Step 5: Creating New Use-Cases. Alongside the work on LSM-tree's scalability, I worked on extending the applicability of LSM-tree to new application domains. The crux of these projects has been to find ways of representing particular application data in a manner that lends itself to efficient indexing. This included indexing the location of invalid data within an SSD for the purpose of garbage-collection [8, 9, 10], creating a new representation for time series that lends itself to fast similarity search using an index [11, 12, 13], and accelerating statistical queries by indexing aggregations of the raw data [14].

Future Directions. The cost contentions between writes, queries, and space in database indexes are intrinsic, and we will likely never overcome them completely. Nevertheless, there is a need to better understand how these overheads interact. For example, if one is willing to merge each data entry α times and has β bits of memory, what is the lowest one can hope for query overheads to be? Today, there is no way of answering such questions, and yet they are necessary to determine whether performance and space overheads are air-tight across all fronts when designing a new system.

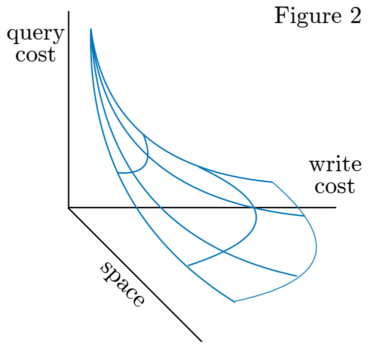


Figure 2

Towards this end, I am inspired by information theory, which provides a lower bound over how much data can be compressed with respect to the data's distribution yet independently of any particular compression algorithm. In a similar fashion, I envision a theory that provides lower bounds for the relevant indexing cost metrics with respect to each other yet independently of any particular data structure. We may conceive these lower bounds as a multi-dimensional Pareto curve as shown in Figure 2, yet where the precise shape of the curve depends on the data size and distribution. The work described above in Steps 1 to 4, which proved the existence of several untapped optimization opportunities, indicates that existing data structures still hover above the curve in Figure 2. Hence, they provide sub-optimal trade-offs to applications running on top. A core goal with a theory of indexing would be to provide guidance on where further optimization opportunities might lay. The ultimate ambition is making every point along the curve in Figure 2 habitable by a known data structure with a particular tuning.

A complementary yet equally important challenge is that application workloads are often not known in advance when a system is first deployed. Once an application is already running, switching its storage engine to a different option, which may be more appropriate, is often disruptive and thus prohibitive. Towards this end, my colleagues and I have envisioned amorphous data structures, which can assume different instances during runtime depending on the application workload [16, 15, 17, 18]. Much work remains to implement amorphous systems in practice, and to allow them to assume provably optimal trade-offs (i.e., along the curve in Figure 2).

With our society coming to increasingly rely on data and its efficient management, the design and performance of storage engines are more important than ever. I am excited by the prospect of leading a research lab in this field at such a foundational moment.

References

[1] **Niv Dayan**, Manos Athanassoulis, Stratos Idreos. Monkey: Optimal Navigable Key-Value Store. In **ACM SIGMOD** International Conference on Management of Data. 2017.

[2] **Niv Dayan**, Manos Athanassoulis, Stratos Idreos. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. In **ACM Transactions on Database Systems**. 2018.

[3] **Niv Dayan**, Stratos Idreos. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In **ACM SIGMOD** International Conference on Management of Data. 2018.

[4] **Niv Dayan**, Stratos Idreos. The Log-Structured Merge-Bush & the Wacky Continuum. In **ACM SIGMOD** International Conference on Management of Data. 2019.

[5] **Niv Dayan**, Moshe Twitto. Chucky: Huffman Coded Key-Value Store. In **ACM SIGMOD** International Conference on Management of Data. 2021.

[6] **Niv Dayan**, Moshe Twitto, Yuval Rochman, Uri Beitler, Itai Ben Zion, Edward Bortnikov, Shmuel Dashevsky, Shmuel Dashevsky, Evgeni Ginzburg, Igal Maly, Avraham Meir, Mark Mokryn, Iddo Naiss, Noam Rabinovich. The End of Moore's Law and the Rise of The Data Processor.. In Proceedings of the Very Large Database Endowment (**PVLDB**). Industry track. 2021.

[7] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, **Niv Dayan**, Wilson Qin, Stratos Idreos. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In **ACM SIGMOD** International Conference on Management of Data. 2020.

[8] **Niv Dayan**, Philippe Bonnet, Stratos Idreos. GeckoFTL: Scalable Flash Translation Techniques for Very Large Flash Devices. In **ACM SIGMOD** International Conference on Management of Data. 2016.

[9] **Niv Dayan**, Martin Kjaer Svendsen, Matias Bjoerling, Philippe Bonnet, Luc Bouganim. EagleTree: Exploring the Design Space of SSD-Based Algorithms (Demo). In Proceedings of the Very Large Database Endowment (**PVLDB**). 2013.

[10] Matias Bjoerling, Philippe Bonnet, Luc Bouganim, **Niv Dayan**. The Necessary Death of the Block Device Interface. In the Biennial Conference on Innovative Data Systems Research (**CIDR**). 2013.

[11] Haridimos Kondylakis, **Niv Dayan**, Kostas Zoumpatianos, Themis Palpanas. Coconut: A Scalable Bottom-Up Approach for building Data Series Indexes. In Proceedings of the Very Large Database Endowment (**PVLDB**). 2018.

[12] Haridimos Kondylakis, **Niv Dayan**, Kostas Zoumpatianos, Themis Palpanas. Coconut: Sortable Summarizations for Scalable Indexes over Static and Streaming Data Series. In the **VLDB Journal**. 2019.

[13] Haridimos Kondylakis, **Niv Dayan**, Kostas Zoumpatianos, Themis Palpanas. Coconut Palm: Static and Streaming Data Series Exploration Now in your Palm (Demo). In **ACM SIGMOD** International Conference on Management of Data. 2019.

[14] Abdul Wasay, Xinding Wei, **Niv Dayan**, Stratos Idreos. Data Canopy: Accelerating Exploratory Statistical Analysis. In **ACM SIGMOD** International Conference on Management of Data. 2017.

[15] Stratos Idreos, **Niv Dayan**, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, Zichen Zhu. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In the Biennial Conference on Innovative Data Systems Research (**CIDR**). 2019.

[16] Stratos Idreos, Kostas Zoumpatianos, Manos Athanassoulis, **Niv Dayan**, Brian Hentschel, Michael S. Kester, Demi Guo, Lukas Maas, Wilson Qin, Abdul Wasay, Yiyou Sun. The Periodic Table of Data Structures. **IEEE Data Eng. Bull.** 2018.

[17] Stratos Idreos, Kostas Zoumpatianos, Subarna Chatterjee, Wilson Qin, Abdul Wasay, Brian Hentschel, Mike Kester, **Niv Dayan**, Demi Guo, Minseo Kang, Yiyou Sun Learning Data Structure Alchemy. **IEEE Data Eng. Bull.** 2019.

[18] Stratos Idreos, Kostas Zoumpatianos, Subarna Chatterjee, Stratos Idreos, Manos Athanassoulis, **Niv Dayan**, Demi Guo, Mike S. Kester, Lukas Maas, Kostas Zoumpatianos. Past and Future Steps for Adaptive Storage Data Systems: From Shallow to Deep Adaptivity. In Proceedings of the International Workshop on Business Intelligence for Real-Time Enterprises (**BIRTE**). 2016.