Query co-planning for shared execution in key-value stores

Josué Ttito[1], Renato Marroquín[2], Sergio Lifschitz[3], Lewis McGibbney[4], José Talavera[5]

[1] Universidad Católica San Pablo, Perú
josue.ttito@ucsp.edu.pe
[2] Oracle, Switzerland
renato.marroquin@oracle.com
[3] Pontifícia Universidade Católica do Rio de Janeiro, Brazil
sergio@inf.puc-rio.br
[4] Jet Propulsion Laboratory-NASA, USA
lewis.j.mcgibbney@jpl.nasa.gov
[5] LuizaLabs, Brazil
jose.herrera@luizalabs.com

Abstract. Key-value stores propose a straightforward yet powerful data model. Data is modeled using key-value pairs where values can be arbitrary objects and written/read using the key associated with it. In addition to their simple interface, such data stores also provide read operations such as full and range scans. However, due to the simplicity of its interface, trying to optimize data accesses becomes challenging. This work aims to enable the shared execution of concurrent range and point queries on key-value stores. Thus, reducing the overall data movement when executing a complete workload. To accomplish this, we analyze different possible data structures and propose our variation of a segment tree, Updatable Interval Tree. Our data structure helps us co-planning and co-executing multiple range queries together and reduces redundant work. This results in executing workloads more efficiently and overall increased throughput, as we show in our evaluation.

Categories and Subject Descriptors: H.2 [**Database Management**]: Miscellaneous; H.3 [**Information Storage and Retrieval**]: Miscellaneous

Keywords: range queries, database, key-value stores, embedded data stores, shared execution

1. INTRODUCTION

The need for storing and analyzing vast amounts of data motivates the design of data management systems tailored for each application's needs. For instance, relational databases are used ubiquitously across digital and non-digital businesses. Graph databases allow more complex data models to be stored while enabling them to express more complex queries. On the other hand, key-value stores offer a much simpler interface for retrieving and storing data, and such simplicity has made them very popular across domains.

The data model used in key-value stores consists of storing (K, V) pairs where K represents a unique identifier of a value V. The actual value V can range from byte arrays to complex JSON documents depending on the key-value implementation. Their standard interface consists of methods such as `GET` for obtaining a value given a key, a `PUT` method for writing a key-value pair, and a `RANGE-SCAN` for retrieving multiple values given a range of keys.

Although the simple interface of key-value stores might be seen as an advantage, the granularity of its

Josué Ttito was supported by grant 234-2015-FONDECYT (Master Program) from Cienciactiva of the National Council for Science,Technology and Technological Innovation (CONCYTEC-PERU). Sergio Lifschitz was partially supported by CAPES (institutional) and CNPq (personal) grants.

operations makes it challenging to optimize more than the execution of a single process. For example, it is unclear how to apply known techniques to optimize multiple queries or even the execution of an entire query workload in one go. In addition to that, several other aspects affect the performance of executing an entire query workload, e.g., each query selectivity, the arrival rate of queries, the underlying data distribution, the query scheduling mechanism for used by the underlying data store, among many others.

This work focuses on optimizing the entire execution of a read-only workload to reduce redundant data accesses. We achieve this by co-planning and co-executing queries that access common subsets of data. More specifically, given a range and point queries workload, we index the query workload predicates to determine overlapping data accesses and then co-plan and co-execute the before-mentioned queries. This results in a series of shared scans that are executed against the data store. Thus, removing redundant data accesses and retrieving required data only once.

The main contributions of this paper are:

—Demonstrate that shared execution is indeed possible even when dealing with a very limiting data access API (only point and range queries).

—Design and analysis of a data structure for enabling shared execution, the Updatable Interval Tree. We use it for allowing shared query execution because it will enable us to minimize redundant data access while maximizing query overlap.

—Evaluation of our shared execution approach using different key-value stores.

The paper is structured as follows: Section 2 introduces work-sharing and explains the traditional approach taken to apply it. In section 3, we describe the available data structures for indexing intervals, introduce our data structure, Updatable Interval Tree. Additionally, we evaluate the complexity of inserting ranges in the Updatable Interval Treein the average and worst case. We evaluate our approach using two popular embedded databases, RocksDB and DuckDB, in section 4. Section 5 outlines related work where shared execution techniques are studied and applied. Finally, we conclude in section 6 and outline future work.

2. WORK-SHARING

There has been a long line of research to reduce redundant work carried out during query processing. This ranges from traditional common sub-expression elimination [Finkelstein 1982] to execute entire query workloads with a shared plan in cloud data services [Marroquin et al. 2018]. Moreover, shared-workload optimization (SWO) [Giannikis et al. 2014; Giannikis et al. 2012] was proposed as an alternative to multi-query optimization (MQO) [Sellis 1988] to enable the generation of shared execution plans suitable not just for a group of queries but also for an entire query workload. In contrast to MQO, where the main goal is to achieve the execution plan with the least cost for a subset of queries, SWO aims to produce an execution plan with the least cost for the entire query workload.

A different optimization point is trying to reduce the overall data movement across the memory hierarchy. For instance, different caching strategies can use temporal and spatial data locality and keep relevant data closer to the CPU. Although caching works the best when the hot part of the data set fits totally in the cache, there are scenarios (e.g., ad-hoc analytics) in which different strategies can help reduce data movement further. When supporting ad-hoc querying or a largely distinct set of queries, simply caching data might not be very effective since keeping a diverse group of queries would require a predictable query frequency or a large cache. Shared execution enables the possibility of improving the overall system's throughput by not carrying out redundant work multiple times but, at the same time, by increasing individual query latencies. In this work, we focus on enabling work-sharing over key-value stores with a workload of `GET` and `RANGE`.

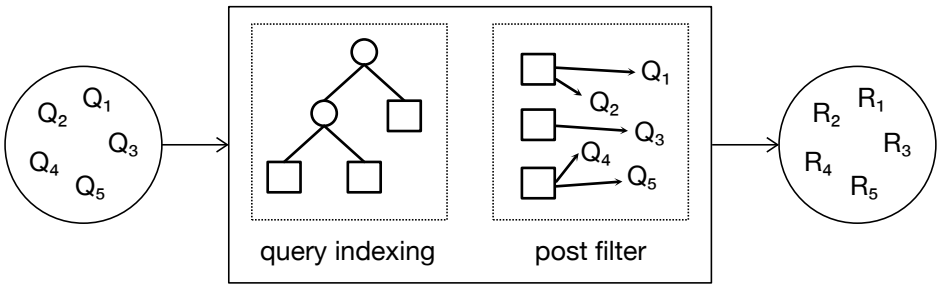


Fig. 1: Shared workload execution.

Figure 1 depicts how shared execution works at a high level. First, all queries are collected before their execution. Then, a shared plan for that group of queries is generated and executed against the data store. Once the results are obtained, they are filtered and returned to each specific query.

Shared execution is similar to regular query batching in the sense that for it to work, multiple queries need to be executed concurrently. However, they differ in shared execution; a shared execution plan is generated for a batch of queries. Such a query plan minimizes the overall redundant work and solves all queries in the batch. In our work, the shared plan generation consists of building an interval tree using the predicates of range and point queries, i.e., indexing the workload queries. This shared plan represents the final range of queries to be executed without any redundant work.

3. SHARED-SCANS IN KEY-VALUE STORES

Typically, data is indexed to speed up access to large amounts of data. On the other hand, query indexing has been proposed when dealing with highly concurrent workloads and stream-processing scenarios where incoming data needs to be checked against many installed queries [Unterbrunner et al. 2009]. Their goal is to reduce the number of operations when determining the queries for which the incoming data is of interest. Our work indexes range and point queries from a query workload to find overlapping opportunities by creating an interval tree with their predicates, thus reducing the overall work. In the paragraphs below, we describe two well-known data structures for storing and retrieving intervals: Segment Tree and Centered Interval Tree. Figures 2 and 3 show a segment and a centered interval tree, respectively, built using the set of intervals below.

$$\begin{array}{llll} S_1 : (100, 670) & S_2 : (230, 450) & S_3 : (120, 340) & S_4 : (430, 760) \\ S_5 : (800, 920) & S_6 : (100, 120) & S_7 : (230, 340) & S_8 : (450, 760) \end{array} \tag{1}$$

A **segment tree** is a balanced binary tree that supports storing information about a set I of n intervals. All interval endpoints are sorted to build this data structure, and elementary intervals are obtained from those elementary intervals. Then, a balanced binary tree is constructed from those elementary intervals. For each node v in the segment tree, the actual interval it represents is determined. Finally, to compute the canonical intervals from I for all nodes, the intervals from I are inserted one by one into the segment tree. Therefore, each leaf node maintains an elementary interval and canonical intervals that overlap with that elementary interval. Building it has a time complexity of $O(n\ log\ n)$ while searching for intervals containing a query point can be done in $O(log\ n\ +\ k)$ time where k is the number of retrieved intervals or segments [1].

A **centered interval tree** is also a binary tree, which given a set I of n intervals, can be constructed in $O(n\ log\ n)$ time. To build it, the entire range of all intervals is divided into half x_{center}. As a

[1] https://en.wikipedia.org/wiki/Segment_tree

consequence, we have three sets of intervals, one to the left of x_{center}, which we call $left$, one to the right which we call $right$, and those intervals overlapping with x_{center} which we call $center$. Then, the intervals in $left$ and $right$ are recursively divided in the same way until there are no intervals left. Overlapping intervals with $center$ are stored in a separate data structure but linked to the node in the interval tree. This data structure contains two lists, one with all intervals sorted by their starting point and another with all intervals sorted by their ending point. Searching for intervals containing a query point can be done in $O(\ log\ n\ +\ m\ )$ time where m is the number of overlapping intervals [2].

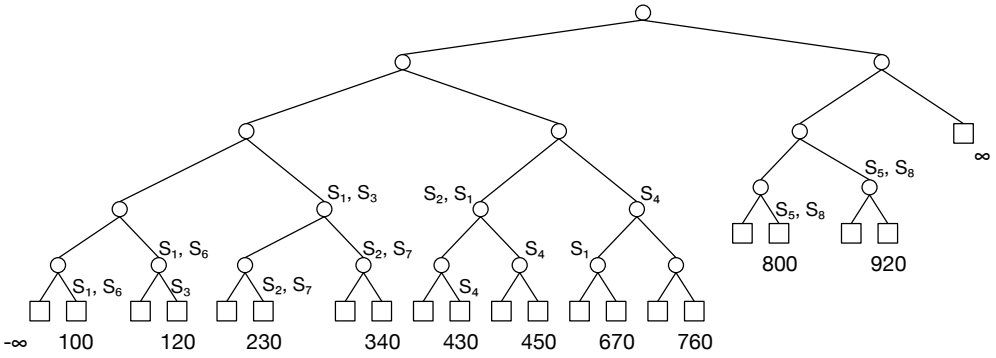


Fig. 2: Segment tree.

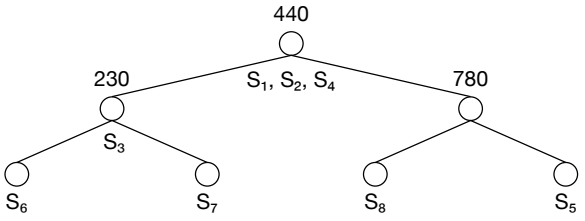


Fig. 3: Centered interval tree.

3.1 Updatable Interval Tree

We define our Updatable Interval Tree (UIT) as a variation of the segment tree. The two notable differences are: (i) the UIT can actually be incrementally updated, and (ii) the UIT aims to minimize its total number of nodes by checking for overlapping intervals and updating its structure accordingly. Additionally, all nodes maintain their intervals whose size is controlled by a parameter M, that represents the maximum size of an interval that any node can hold. We use M to control the total amount of nodes created.

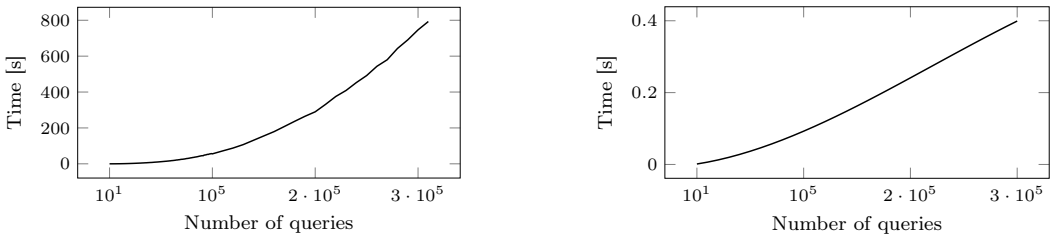
The insertion of an interval $i\ = (i_{low}, i_{high})$ begins at the root node. If the interval i exceeds M, then the interval is split into multiple ranges that do not exceed M. At this point, we check if the new interval overlaps with the interval j of the current node c. If overlap exists and c is a leaf node, then the interval j is updated to j', which copes with the new interval, and *no* new node is inserted. This means that the current node's interval will be $(j_{low'}, j_{high'})$ where $j_{low'} = i_{low}$ if $j_{low} \geq i_{low}$ or j_{low} otherwise, and $j_{high'} = i_{high}$ if $j_{high} \leq i_{low}$ or j_{high} otherwise. If there is no overlap, then, similarly to a centered interval tree, we check whether or not the new interval i is to the left or the right of the current node's interval and insert a new node accordingly. When inserting i, we check the size of the resulting interval of j' to determine if it is smaller than M or not. If it is, we only update the

[2] https://en.wikipedia.org/wiki/Interval_tree#Centered_interval_tree

node with the new j'. Otherwise, the interval j' is split. We create two new nodes, one containing an interval of size M and one with the remaining interval, i.e., $M - j'$. Next, the node with the M sized interval is inserted. The node with an interval of size $M - j'$ is added to the queue of nodes still to be inserted.

After the insertion, we enforce the segment tree property that intervals from the same level must not overlap. We do this by recursively verifying if any interval from non-leaf nodes needs to be updated. Overall, similarly to other non-self-balancing binary tree data structures, building our UIT has a complexity of $O(\ n^2\ )$ in the worst-case, and $O(\ n\ log(n)\ )$ in the average-case.

In figure 4a, we inserted ranges into our UIT. Here, the ranges limits are chosen using a zipf-distribution to study and verify the $O(\ n^2\ )$ complexity when inserting elements in the worst case. Basically, the test consisted of inserting non-overlapping ranges where each new range is strictly larger than the previous one. This forced the UITto degenerate into a linked-list. On the other hand, figure 4b shows the running time when inserting a set of ranges in an average case where range limits are chosen uniformly at random from a continuous domain.



(a) UIT insertion worst case.

(b) UIT insertion average case.

Fig. 4: Updatable Interval Tree building time (s)

The UIT does not create new nodes in all cases. There are two cases when we may merge nodes, and this depends on the maximum interval size nodes can hold, i.e., the parameter M. One is when the new interval overlaps, and the length of the resulting j' is smaller than M. The other case is if the resulting interval is contiguous to its sibling node's interval, which we inspect after insertion. Suppose both intervals are contiguous and the resulting interval j' is smaller than M. In that case, we merge the two nodes into a single one with an adjacent interval composed of both original intervals. Figure 5 shows the example intervals stored in the UIT.

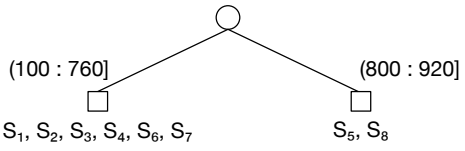


Fig. 5: Updatable Interval Tree representing example intervals

Like a segment tree, the non-leaf nodes of our data structure keep more segments towards the root node, and nodes at the same level only have non-overlapping intervals. The leaf nodes represent the actual intervals we need to retrieve from the database. To determine which final intervals match the original queries, we use only the leaf nodes to create a second interval tree. Then, we compare the leaf nodes intervals to the initial queries. We also make the leaf nodes keep pointers to their sibling nodes to avoid traversing the entire data structure when retrieving only the leaf nodes.

It is worth noting that we are solely interested in the leaf nodes for our particular use case because these nodes maintain non-overlapping segments of at most size M. We can directly use the leaf nodes segments to execute range queries against the data stores. Additionally, we augment each node to

store the identifiers of the queries requesting the node's segment (either entirely or partially). This helps us improve the process of associating the data stores execution results with the corresponding original queries.

Finally, searching for intervals is similar to how we would proceed on segment trees. However, searching intervals in the UITree is not optimized further since its primary goal is to create a set of disjoint intervals on its leave nodes. Then, these leaf nodes are used for creating a new query plan without redundant work.

4. EXPERIMENTAL EVALUATION

Although there are other popular data processing benchmarks, such as TPC-DS and TPC-H, they evaluate more complex queries. The publicly available benchmark which is the closest to our setting is the YCSB benchmark which is designed to assess different aspects of key-value stores. The YCSB benchmark proposes two types of read-only query workloads, one for pure point queries (workload C) and one for querying small ranges (workload E). However, there are no workloads mixing point and range queries, nor the parameters for varying the selectivities of range queries. Thus, we design our set of experiments inspired by the YCSB benchmark and accommodate different types of read-only query workloads to suit our needs.

In general, our benchmark consists of two phases: a load and a query phase. The load phase consists of loading a total of E key-value pairs into a key-value store where both key and value are chosen uniformly at random from a domain D. The query phase executes Q queries (a combination of range or point queries) against a key-value store. We compare our proposed solution to the other only possible way of executing read queries against a key-value store, which is one query-at-a-time (`QAT`). Each query is executed independently from the other; thus, parallelization is possible. Even though it is possible to exploit parallelism, both approaches, `QAT` and `Shared-Execution`, would be benefited from it by a factor of the total number of threads used. For simplicity, we do single-threaded execution throughout the experiments.

The C++ implementation of our shared execution strategy is used as a middleware to make it applicable to other key-value stores. However, it could also be implemented as an additional component of any key-value store. We execute all experiments in a machine running Ubuntu 20.04 with 16GB of DDR4 RAM (2666 Mhz), an Intel Core i5-8400 CPU, a 500GB hard drive. We run them three times and report the average end-to-end execution time.

Systems-under-test:

We chose two popular embedded data stores to evaluate our solution, RocksDB [Inc. 2020] (optimized for fast reads) and DuckDB [Informatica 2020] (optimized for analytical queries). Although DuckDB is not a key-value store, we wanted to evaluate how much we could improve other data store throughput using our work-sharing approach.

Rocks-DB is an in-process key-value store that is optimized for fast, low latency accesses targeting flash drives and high-speed disk drives. It uses a log-structured database engine for achieving fast reads and writes operations of arbitrarily-sized byte streams. Like other log-structured storage engines, RocksDB keeps recently inserted data in memory, and while data is inserted, it is moved into different levels of the underlying data structure.

DuckDB is an in-process database management system specialized for analytical workloads. It is optimized for ingesting and updating data in large batches rather than optimizing single row insertions or updates. DuckDB uses a columnar-vectorized query execution engine that processes large batches of values (vectors of data) at a time. This vectorized execution model makes DuckDB more efficient for analytical queries than other traditional systems such as PostgreSQL, MySQL, or SQLite, which process each row sequentially.

We configure RocksDB to use only RAM and not the local disk to improve its overall performance in all our experiments. For DuckDB, we use its default configuration. We load the same amount of data and use a similar schema in both systems. More specifically, for DuckDB, we use a schema with two integer attributes for simulating a key-value store. DuckDB only has two types of indexes: (i) Min-max indices and (ii) Adaptive radix tree indexes. The former index type is always defined for all general-purpose tables, and the latter is recommended for highly selective queries. We decided not to define additional indexes because we wanted to evaluate how our shared execution techniques could help the underlying systems.

Although both systems are not comparable (designed and optimized for different workloads), we wanted to have a rough idea of how each system performed when executing a series of range scans. Figure 6 shows the execution time of range queries in both systems. This experiment uses a domain size of $D = 10^6$ and load 10^6 key-value pairs. Our query workload consists of groups of range queries. Each range query has a fixed selectivity of 10%. We observe that although RocksDB is not fully optimized for range scans, it is comparable to DuckDB. This is because RocksDB takes full advantage that its data set fits in memory and that range queries are not very selective. We believe that for more complex queries or queries, selecting more data DuckDB would outperform RocksDB, but a more in-depth comparison of both systems is orthogonal to our work.

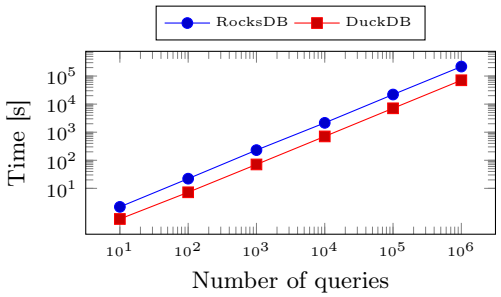
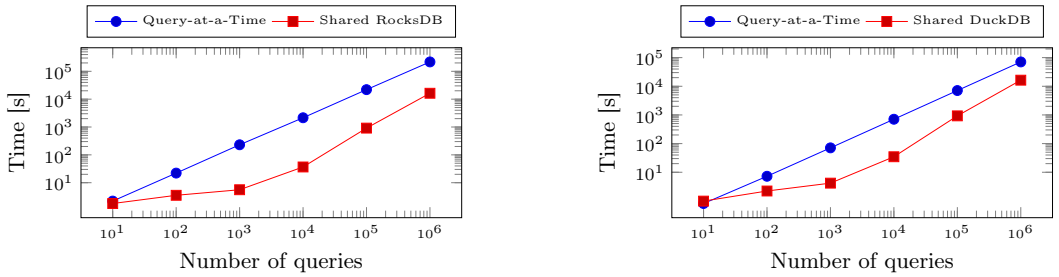


Fig. 6: Range scans query execution.

A) Shared execution vs Query-at-a-time: This experiment uses a domain size of $D = 10^6$, which the loading phase uses to insert 10^6 key-value pairs. We compare the `QAT` approach against the `Shared` execution approach to understand how they compare. The query workload used here consists only of range queries where each query has a fixed selectivity of 10%, i.e., selects 10% of data. We use this selectivity to determine the maximum interval size, M, for nodes of our UIT tree. We report the end-to-end execution time of the entire workload while varying its size, i.e., the total number of queries, Q, executed.

Figures 7a and 7b show the execution time of the two approaches, query-at-a-time (`QAT`) and shared-execution (`Shared`). Regarding the `QAT` approach, we observe that the total execution time increases linearly with the query workload size, which is expected. Although the execution time of the `Shared` execution approach also increases linearly, it is still 15X faster than executing the entire query workload with a `QAT` approach. There are two main reasons which make the execution time grow linearly in the `Shared` approach. The first one is related to the complexity of building our UIT. The second one is the amount of time it currently takes us to perform results separation. Our current implementation consists of a linear scan through our result set while assigning results to each specific query. We also plan to apply similar techniques to query indexing to improve this part.

Figures 8a and 8b show the query indexing break down when compared to the actual query execution for RocksDb and DuckDB, respectively. Here, we show the time it takes for query indexing and executing the resulting query workload. We observe that with smaller workloads, less than 10^6 queries, building our query indexing structure is in the order of tens of milliseconds. Therefore, when

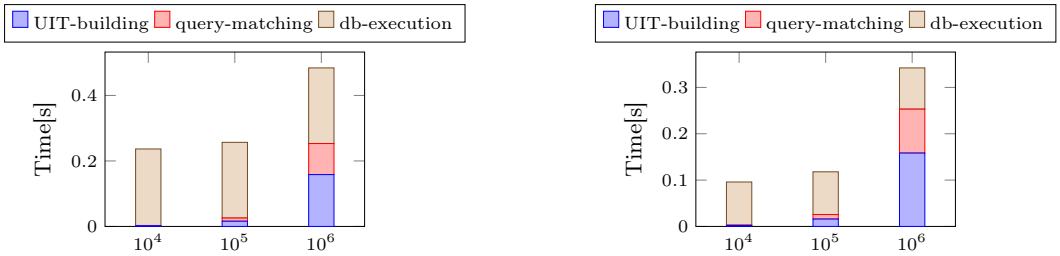


(a) RocksDB (b) DuckDB

Fig. 7: Experiment varying number of queries

integrating our approach within a more extensive system, we could batch up to 10^5 with a *negligible* time penalty. On the other hand, when dealing with larger query workloads, the time taken for query indexing is comparable to the query execution. There are mainly two reasons for this: (i) the time complexity of our indexing strategy, i.e., building our data structure will grow sublinearly with the number of queries; and (ii) the total number of resulting queries accounts for only 15% of the original workload size of 10^6. The reduction of the workload size occurs because once the UIT becomes a complete binary tree, the leaf nodes intervals actually cover the entire domain. Thus, when adding new queries, no additional insertions or updates are needed for the UIT. This results in a significant reduction of redundant work needed for solving the entire query workload.

It is worth pointing out that in the case of DuckDB (figure 8b), the time taken for building our UITis similar to the execution time of the resulting query workload. This is because the resulting query workload has been dramatically reduced (by 75%). After all, this query has redundant work that is now avoided. DuckDB can efficiently use its min-max indices to solve the queries.



(a) RocksDB (b) DuckDB

Fig. 8: Query indexing break down

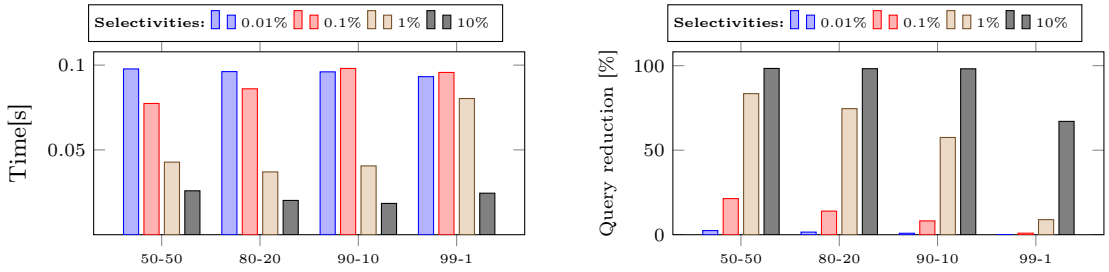
B) Varying query workload mix:

It is unlikely that a single type of query would always be the one executed against the key-value store in real-world settings. However, we can expect that the ratio between point and range queries changes over time. Thus, we investigate the cost of building the UIT with a workload containing both range and point queries.

Our observation is that if a workload consists only of range queries, many of them might overlap depending on their selectivity. But on the other hand, if the query workload consists of only point queries, there might be little to no chance to reduce redundant work.

For this experiment, we use domain $D = 10^9$ and a query workload of $Q = 10^5$ queries. We vary the percentage of range and point queries in the query workload and the range query selectivities (from 0.01% to 10%) to show how the query selectivity impacts our shared execution strategy. We

name our query workloads to denote the range and point query ratio, e.g., a workload called `90-10` means having 90% of point queries and 10% of range queries.



(a) UI-tree construction times

(b) Percentage of queries reduced

Fig. 9: Experiment varying workload mix

Figure 9a depicts the building time of the UIT under different workload mixes. It is important to note that regardless of the amount of range queries present in the workload, if such range queries are very selective, it takes the longest to build our indexing data structure. The main reason is that As a consequence, we obtain a UIT with a more significant number of nodes. On the other hand, if range queries select more data, then many queries might overlap, reducing our data structure's building time. The effect of queries overlapping is reflected in the total number of resulting queries to be executed (see Figure 9b). For example, the query workload labeled as `99-1`, which contains only 1% of range queries, can reduce the amount of redundant work by up to 70% if range queries select at least 10% of the data.

5. RELATED WORK

Key-value stores These systems have been widely studied both by academia [Escriva et al. 2012; Dayan et al. 2017; Dayan and Idreos 2018] and by industry [Sivasubramanian 2012; Li et al. 2017; Chandramouli et al. 2018]. For instance, HyperDex [Escriva et al. 2012] proposes the design of a searchable key-value store by introducing a search primitive over the values stored using hyperspace hashing for mapping objects with multiple attributes into a multidimensional hyperspace. While KV-Direct [Li et al. 2017] presents the design of a key-value store that leverages hardware accelerators for doing in-network processing of incoming requests, thus, increasing overall throughput for `GET` and `PUT` operations. Moreover, the Tell project [Pilman et al. 2017] shows that many popular key-value stores are not suitable for performing analytical queries as such queries require access to most of the underlying data. This is accomplished through full table scans in traditional databases, an operation not well supported by key-value stores. Pilman et al. propose a key-value store design that supports both full table scans and high throughput of `GET` and `PUT` operations. These efforts aim to improve different aspects of key-value stores and show considerable interest in improving them.

Sharing the scan operator CoScan Sharing is a scheduling framework proposed by [Wang et al.] that eliminates redundant processing in data processing workflows. This system was built over Nova (a higher level of abstraction over the Hadoop-Distributed-File-System) using Pig-Latin as execution engine and its primary goal is to combine Pig-Latin dataflow jobs and create a new one where data access is shared among the input jobs. These input jobs could be merged or split depending on a strategy set for a greedy algorithm. A particular benchmark was designed for the CoScan framework, consisting of 17 Pig-Latin queries, PigMix.

A different approach for reducing redundant work is by using query operators when solving multiple queries. Crescando [Giannikis et al.] is a notable in-memory data processing system in this line of work. Its most crucial design choice is the use of an "always-on" scan operator, which continually

goes over all available data. The authors add transactions to the system by using epochs. Every time the scan operator starts from the beginning of the data, a new epoch begins. Then, when new queries arrive in the system, they all share the same epoch and consume data produced by the scan operator for that epoch. Additionally, the authors implement a predicate indexing algorithm to reduce the overall work of filtering the data produced by the scan operator.

Sharing more operators A different line of research aims to share the scan operator and all possible operators among a group of queries. For example, SharedDB is a database architecture proposed by [Giannikis et al. 2012], which solves batches of queries accessing a common data set. SharedDB introduces the following main concepts to enable executing multiple queries sharing relational operators: (i) The data query model: It enhances the data read with an additional attribute that represents the query identifier of the query interested in a particular record. The authors add this extra attribute at execution time to keep data in Normal Form 2 and avoid replicating data for each query involved. This allows SharedDB to solve different concurrent queries. (ii) A global query plan instead of a single one per query: Instead of compiling every query into a separate query plan (as traditional database systems do), SharedDB compiles the whole query workload into a single global query plan. The intuition behind this design decision is that specific operators of this global plan can be activated to solve such queries based on the queries received. And lastly, (iii) shared join plans to amortize the cost of computing join results. SharedDB achieves very high throughput by grouping queries that access the same data and executing them in one go. This approach is most advantageous when dealing with many (possibly even hundreds) of concurrent queries, which will probably overlap in the sets of data they need to process.

Sharing operators execution With the increased popularity of software-as-a-service databases, where the cost is proportional to the amount of data accessed, it is essential to reduce the amount of data accessed to reduce costs. This is especially important when this additional cost comes from redundant data accesses. For instance, the authors [Marroquin et al. 2018] focus on applying shared execution techniques to serverless cloud databases, which have a business model to charge for every byte that is accessed from storage. Thus, having as a goal to reduce the overall monetary cost when solving an entire query workload. The authors [Marroquin et al. 2018] based their work on the query data model proposed by [Giannikis et al. 2012], but because this work targets cloud-based systems, all work-sharing has to be done at the SQL level. Moreover, the authors define the shared operator using relational algebra and a set of rewriting techniques to enable the shared execution of queries. They show that even using plain SQL, the most relevant relational algebra operators could be rewritten to be used by multiple queries simultaneously. Applying these shared execution techniques in the context of serverless databases results in significant cost reductions.

6. CONCLUSION AND ON-GOING WORK

We presented the design and implementation of a data structure to enable shared work execution of a query workload consisting of range and point queries. The resulting UIT helps us determining the overlapping intervals, which represent the redundant work, while minimizing the total number of nodes created. This helps us finding out the required queries to be executed and then to execute only the non-overlapping queries. We show that using our Updatable Interval Tree for co-planning and co-executing an entire query workload helps reducing the redundant work and improving the overall throughput of the system.

There are a few lines of work we plan to continue improving. We are continually working on improving our overall query-indexing approach. One specific aspect we are trying to improve is to remove the query matching process we currently do. By doing this, we will further improve the efficiency of our query co-planning step.

Moreover, we plan to use the maximum interval size, M, to introduce a more flexible interval

partitioning. For instance, if applying our techniques with a Log-Structured-Merge tree, we could use the LSM min-max values per level to improve query grouping, i.e., grouping together queries that belong to the same level, and disallowing queries that span multiple levels. In a distributed setting, we want to explore how we could improve distributed query execution by using the underlying data distribution across a cluster of nodes to reduce overall data movement.

REFERENCES

Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J., Hunter, J., and Barnett, M. Faster: A concurrent key-value store with in-place updates. In *2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18), Houston, TX, USA*, 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18), Houston, TX, USA ed. ACM, 2018.

Dayan, N., Athanassoulis, M., and Idreos, S. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Association for Computing Machinery, New York, NY, USA, 2017.

Dayan, N. and Idreos, S. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Association for Computing Machinery, New York, NY, USA, 2018.

Escriva, R., Wong, B., and Sirer, E. G. Hyperdex: A distributed, searchable key-value store. *SIGCOMM Comput. Commun. Rev.* 42 (4), Aug., 2012.

Finkelstein, S. J. Common subexpression analysis in database applications. In *Proceedings of the 1982 ACM SIGMOD*, M. Schkolnick (Ed.), 1982.

Giannikis, G., Alonso, G., and Kossmann, D. Shareddb: Killing one thousand queries with one stone. *Proc. VLDB Endow.* 5 (6), 2012.

Giannikis, G., Makreshanski, D., Alonso, G., and Kossmann, D. Shared workload optimization. *Proc. VLDB Endow.* 7 (6), 2014.

Giannikis, G., Unterbrunner, P., Meyer, J., Alonso, G., Fauser, D., and Kossmann, D. Crescando. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10.

Inc., F. rocksdb, 2020.

Informatica, C. W. duckdb, 2020.

Li, B., Ruan, Z., Xiao, W., Lu, Y., Xiong, Y., Putnam, A., Chen, E., and Zhang, L. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, Proceedings of the 26th Symposium on Operating Systems Principles ed. ACM, 2017.

Marroquin, R., Müller, I., Makreshanski, D., and Alonso, G. Pay one, get hundreds for free: Reducing cloud costs through shared query execution. In *ACM SoCC 2018*, 2018.

Pilman, M., Bocksrocker, K., Braun, L., Marroquín, R., and Kossmann, D. Fast scans on key-value stores. *Proc. VLDB Endow.* 10 (11), Aug., 2017.

Sellis, T. K. Multiple-query optimization. *ACM Trans. Database Syst.* 13 (1), 1988.

Sivasubramanian, S. Amazon dynamodb: A seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Association for Computing Machinery, New York, NY, USA, 2012.

Unterbrunner, P., Giannikis, G., Alonso, G., Fauser, D., and Kossmann, D. Predictable performance for unpredictable workloads. *Proc. VLDB Endow.* 2 (1), 2009.

Wang, X., Olston, C., Sarma, A. D., and Burns, R. Coscan: Cooperative scan sharing in the cloud. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. SOCC '11. New York, NY, USA.