Query co-planning for shared execution in Key-Value Stores

Josué Ttito[1], Renato Marroquín[2], Sergio Lifschitz[3]

[1]Dept. Computer Science
Universidad Católica San Pablo
Arequipa–Perú
josue.ttito@ucsp.edu.pe

[2]Oracle
Zürich-Switzerland
renato.marroquin@oracle.com

[3]Informatics Dept.
Pontifícia Universidade Católica-RJ
RJ-Brazil
sergio@inf.puc-rio.br

Abstract. *Key value stores propose a very simple yet powerful data model. Data is modeled using key-value pairs where values can be arbitrary objects and can be 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. This data structure helps us co-planning and co-executing multiple range queries together, as we show in our evaluation.*

1. Introduction

The need for storing and analyzing vasts amounts of data motivates the design of data management systems tailored for each application's needs. For instance, 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 both for retrieving and storing data. 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 arrays of byte 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, its operations' granularity makes it challenging to optimize more than the execution of a single operation. For example, it is not clear how to apply known techniques to optimize multiple queries or even optimize the execution of an entire query workload together. In this work, we focus 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 workload consisting of range and point queries to be executed, we index the 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.

[1]Josué's work was supported by grant 234-2015-FONDECYT (Master Program) from Cienciactiva of the National Council for Science,Technology and Technological Innovation (CONCYTEC-PERU).

2. Work-sharing

There has been a long line of research for reducing the amount of redundant work carried out during query processing. This ranges from traditional common sub-expression elimination [Finkelstein 1982] to executing 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.

In high concurrency environments, many different queries are executed (each query with its own frequency) against a shared pool of data. Approaches such as caching might not be very effective in such scenarios because supporting a diverse set of queries would require either a large cache or a predictable query frequency. On the other hand, 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. Here, we focus on enabling work-sharing over key-value stores with a workload of `GET` and `RANGE`.

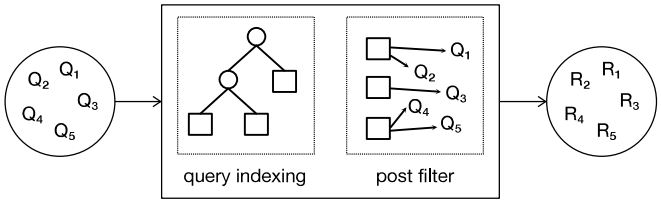


Figure 1. Shared workload execution.

Figure 1 depicts how shared execution works at a high level. First, all queries are collected prior to 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 that 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 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. In our work, we index 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.

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

A **segment tree** is a balanced binary tree that supports storing information about a set I of n intervals. 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 [2].

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. Searching for intervals containing a query point can be done in $O(\ log\ n\ +\ m\ )$ time where m is the number of overlapping intervals [3].

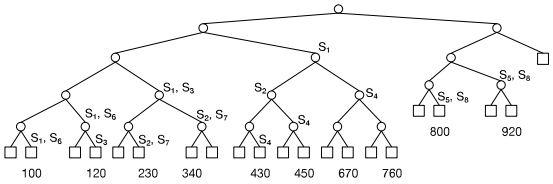


Figure 2. Segment tree.

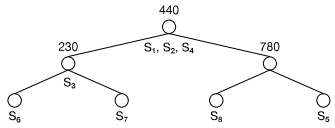


Figure 3. Centered interval tree.

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. 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. Moreover, 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, then we only update the node with the new j', otherwise, the interval j' is split, and a new node is created containing half of the j' interval. 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, in the worst-case, building our UIT has a complexity of $O(\ n^2\ )$ as other similar non-self-balancing data structures.

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

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

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. If both intervals are contiguous, then we merge the two nodes into a single one which has a contiguous interval composed of both original intervals. Figure 4 shows the example intervals stored in the UIT.

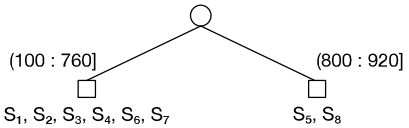


Figure 4. Updatable Interval Tree representing example intervals

Similarly to a segment tree, the non-leaf nodes of our data structure keep larger 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 match the leaf nodes intervals to the initial queries. We also make the leaf nodes to keep pointers to its sibling nodes to avoid traversing the entire data structure when retrieving only the leaf nodes.

4. Experimental evaluation

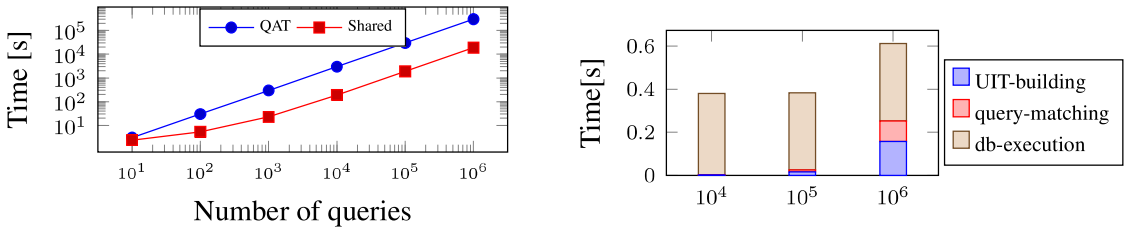
We design a set of experiments for evaluating different aspects of our solution. Our benchmark consists of two phases: a load, and a query phase. The load phase consists of loading a total number of E key-value pairs into a key-value store where both key and value are chosen uniformly at random from a domain D. We use as key-value store *Rocks-DB*. The query phase consists of executing Q queries (a combination of range or point queries) against a key-value store. Moreover, 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 each 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, an Intel Core i5-8400 CPU, a 500GB of hard-drive. We run them three times and report the average end-to-end execution time.

A) Shared execution vs Query-at-a-time: In this experiment, we use 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. Moreover, 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.

Figure 5a shows 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 to 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 for improving this part.

Figure 5b shows the query indexing break down when compared to the actual query execution. Here, we show the time it takes for query indexing and the time to execute 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 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 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 this particular workload.



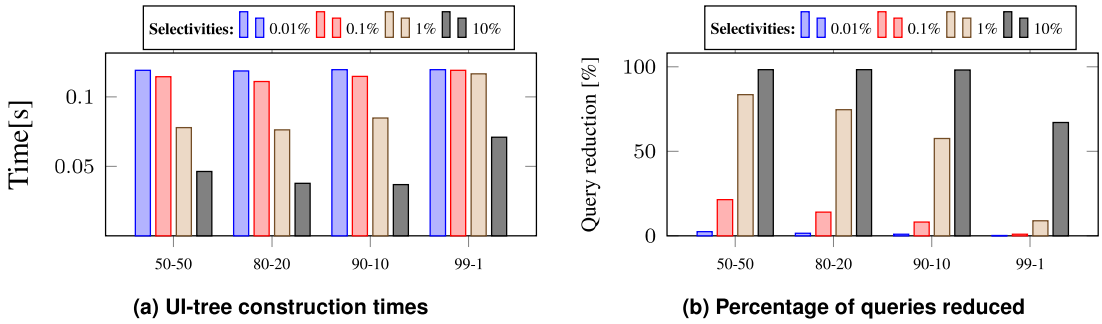
(a) Shared execution vs Query-at-a-time

(b) Query indexing break down

Figure 5. Experiment varying number of queries

B) Varying query workload mix: We investigate the cost of building the UIT with a workload containing both range and point queries. This is because if a a workload consists only of range queries, many of them might overlap depending on their selectivity. On the other hand, if the query workload consists of only point queries, then there might be little to no chance to reduce redundant work. For this experiment, we use as domain $D = 10^9$ and a query workload of $Q = 10^5$ queries. Here, 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. Moreover, a workload named `90-10` means having 90% of point queries and 10% of range queries.

Figure 6a 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



(a) UI-tree construction times **(b) Percentage of queries reduced**

Figure 6. Experiment varying workload mix

if such range queries are very selective, it takes the longest to build our indexing data structure. The main reason is that there is less opportunity for queries overlapping with a mix of very selective range queries and point queries. 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, which reduces the building time of our data structure. The effect of queries overlapping is reflected in the total number of resulting queries to be executed (see Figure 6b). For example, the query workload labeled as `99-1`, which contains only 1% of range queries, can reduce the amount of redundant work in up to 70% if range queries select at least 10% of the data.

5. 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. There are a few lines of work we are still working on. For instance, we are working on removing the query matching process we currently do. By doing this, we will further improve the efficiency of our query indexing approach. Moreover, we plan to use the maximum interval size, M, to introduce a more flexible interval partitioning scheme. We also plan to leverage information from underlying the key-value (e.g., LSM min-max values per layer, data distribution per server in a distributed data store) to achieve this.

References

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

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

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

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

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

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