Optimal Splitters for Database Partitioning with Size Bounds

Kenneth A. Ross[*]
Columbia University
kar@cs.columbia.edu

John Cieslewicz[*†]
Columbia University
johnc@cs.columbia.edu

ABSTRACT

Partitioning is an important step in several database algorithms, including sorting, aggregation, and joins. Partitioning is also fundamental for dividing work into equal-sized (or balanced) parallel subtasks. In this paper, we aim to find, materialize and maintain a set of partitioning elements (splitters) for a data set. Unlike traditional partitioning elements, our splitters define both inequality and equality partitions, which allows us to bound the size of the inequality partitions. We provide an algorithm for determining an optimal set of splitters from a sorted data set and show that it has time complexity $O(k \lg^2 N)$, where k is the number of splitters requested and N is the size of the data set. We show how the algorithm can be extended to pairs of tables, so that joins can be partitioned into work units that have balanced cost. We demonstrate experimentally (a) that finding the optimal set of splitters can be done efficiently, and (b) that using the precomputed splitters can improve the time to sort a data set by up to 76%, with particular benefits in the presence of a few heavy hitters.

1. INTRODUCTION

Partitioning is an important component of a scalable database system. It is commonly used for fundamental operations such as joins, aggregation, sorting, and dividing work into balanced pieces. Balanced pieces of work are relevant for parallel processing (to get good processor utilization) or memory-constrained processing (to get good spatial locality).

Current partitioning paradigms include hash-partitioning and range-partitioning [5]. A problem with both of these paradigms is that neither can guarantee a non-trivial upper-bound on the size of a partition. Common keys, sometimes known as "heavy hitters," may cause a partition to be much larger than the average partition size. This is problematic for two reasons:

- If the complexity of processing each partition is superlinear in the partition size, such as for sorting, unbalanced partitions are potentially inefficient.
- If the partitioning is done to divide a job into pieces to be executed in parallel, an unbalanced partitioning could lead to an inefficient use of the parallel resources.

We propose an alternative range-partitioning scheme in which a table with N records is partitioned using a set of k distinct keys into $2k + 1$ partitions. Unlike traditional range-partitioning, our ranges involve strict inequalities, and we have k additional partitions explicitly devoted to single keys.

There are two main advantages to this scheme, which uses the same number of splitters as traditional range-partitioning. Firstly, we can provide an upper bound on the size of the inequality partitions. By choosing splitters that cover the heaviest hitters, we can ensure that no inequality partition contains more than $\lceil \frac{(N-k)}{(k+1)} \rceil$ elements. (This is a worst-case bound; we can often do better when there are heavy hitters as we will show in Section 4.)

Secondly, in a database context, one often can leverage the knowledge that an equality partition contains a single key to process the partition more efficiently. Examples include:

- For sorting, the partition does not need to be further processed.
- For aggregation, a scalar aggregation (with running aggregates in registers) is likely to be much more efficient than a grouped aggregate (with running aggregates in a hash table).
- For joins of fixed-length records, a matching key in a joined table can use a simple `memcpy` operation for the whole partition, rather than having to do key matching for every record.

The splitters can be computed either on-line or off-line. In an on-line computation, one could sample the column being partitioned. A sample of $O(\sqrt{N})$ elements appears to be a good choice [15]. Since the number of partitioning elements k is likely to be many orders of magnitude smaller than the number N of records, it is feasible to sample say $10k$ records from the table with cost negligible compared to the cost of a table scan. With such oversampling, it is highly likely that heavy hitters can be identified from the sample [7]. The algorithm for computing the splitters would then be the same as the off-line computation described below, but using the sample rather than the full data set.

In an off-line computation, one can compute an *optimal* set of splitters. In Section 2 we provide an algo-

[*]Supported by NSF Grant IIS-0534389

[†]Supported by a U.S. Department of Homeland Security Graduate Research Fellowship

rithm for computing this optimal set from a sorted column in $O(k \lg^2 N)$ time. The sorted input could come from a tree index on the column, or from an explicit sort of the column itself. We have implemented this algorithm, and show that it is practical, taking fractions of a second for realistic memory-resident examples.

The algorithm simultaneously computes the exact count in each partition. The count can be an important piece of information. For example, it allows a partition-based sorting algorithm such as sample-sort [9] to partition the data into contiguous regions, so that each partition can be sorted in place to generate the final sorted result without further data movement.

The splitter structure can provide time savings at a fraction of the space cost of alternatives. For example, the C-Store system advocates the physical storage of multiple versions of a single table according to multiple sort keys [21]. For large tables, the number of physical representations would be limited by the amount of available storage and (depending on the implementation) by the cost of incremental maintenance for updates. Our splitter structure can save a substantial fraction of the sorting cost, using a data structure that is orders of magnitude smaller than a table or an index. (We will quantify these claims in Section 4.)

Precomputed splitters are also useful for partitioning on parallel machines to make sure that work is balanced across processors. In the case of a join, using a splitter set from one of the participating tables may give unbalanced partitions for the other table. We provide a modification of our optimal splitter finding algorithm that chooses an optimal splitter set for a pair of tables using a column (with a common domain) from each. This algorithm uses a cost function to give the best possible cost bound on the generated partitions for a given number of splitters. This algorithm uses $O(k \lg^3 N)$ time.

As well as providing the performance benefits mentioned above, a splitter set possesses many of the statistical properties of an equi-depth histogram, and can be used as such for approximate query processing [12] or for selectivity estimation of range predicates [1]. It can also provide exact selectivities for heavy hitters, like a compressed histogram [19].

An important contribution of this splitter technique is its robustness given any distribution. We provide guarantees on the size of the inequality partitions generated by the splitters, without requiring special knowledge about the input such as the number of unique keys or information about heavy hitters.

Section 2 presents the definition of splitters as well as splitter finding algorithms for single columns and multiple tables. We present refinements to these optimal splitters in Section 3. An experimental evaluation of the splitter finding algorithm, efficient data set partitioning, and an application to sorting can be found in Section 4. Section 5 presents related work. We conclude and discuss future work in Section 6. Proofs that are not in the main text can be found in Appendix A.

2. COMPUTING THE SPLITTERS

2.1 Terminology

We assume an ordered data type, such as a number or string, for the partitioning key. A set of k distinct keys $s_1, \ldots, s_k$ called *splitters* defines $2k+1$ partitions:

- k *equality partitions* of the form $\{x | x = s_i\}$, $i = 1, \ldots, k$.
- $k-1$ *inequality partitions* of the form $\{x | s_i < x < s_{i+1}\}$, $i = 1, \ldots, k-1$.
- Two *inequality partitions* $\{x | x < s_1\}$ and $\{x | x > s_k\}$.

A table T may be distributed to partitions according to a splitter set; each record is mapped to a single partition based on the value of a particular column of T. The *breadth* of a splitter set for a table T is the maximum cardinality among all inequality partitions of T.

A splitter set S is *optimal* for table T and cardinality bound k if, among all splitter sets with at most k elements, S has minimal breadth for T. By limiting the size of the biggest inequality partition, optimality ensures that these partitions are (to the extent possible given the data distribution) well-balanced.

2.2 Splitters for Single Columns

We now present Algorithm 2.1 in Figure 1, whose purpose is to find (if possible) a set of up to k splitters that yield breadth at most b for a given data set.

LEMMA 2.1. *Algorithm 2.1 is correct.*

COROLLARY 2.2. *It is always possible to split N records using k splitters so that at most $\lceil \frac{N-k}{k+1} \rceil$ elements are in each inequality partition.*

COROLLARY 2.3. *A key that occurs at least $\lceil \frac{N}{k} \rceil$ times in a data set of size N must be represented in any optimal splitter set of size k.*

PROOF. *If not, then some inequality range would contain too many elements to be optimal.* □

LEMMA 2.4. *Algorithm 2.1 takes time $O(min\{(N/b), k\} \lg N)$.*

PROOF. *The number of iterations is at most $min\{(N/b), k\}$. In each iteration, the work is $O(\lg N)$: the index of the first nonmatching key can be found by doubling the index range until a distinct key is found, then using binary search to locate the earliest nonmatch.* □

Algorithm 2.2 in Figure 2 uses Algorithm 2.1 repeatedly to find the best bound b for a data set, given k.

THEOREM 2.5. *Algorithm 2.2 returns an optimal splitter set.*

LEMMA 2.6. *Algorithm 2.2 takes time $O(k \lg^2 N)$.*

PROOF. *The number of iterations is at most $\lg \lceil \frac{N-k}{k+1} \rceil$. In each iteration, the work is $O(k \lg N)$ by Lemma 2.4.* □

2.3 Multiple Tables

If two tables R_1 and R_2 are often joined on a shared attribute C, we might try to utilize a precomputed splitter set to make the join process more efficient. This would be a space-efficient alternative to materializing

ALGORITHM 2.1.

Input: *Number of records $N \geq 1$, number of keys $k \geq 0$, bound $b \geq 1$, Records $r[0], \ldots, r[N-1]$ in ascending key order.*

Output: *Either (a) "error", indicating that the records cannot be split using k splitters in a way that ensures all inequality partitions have cardinality no more than b; or (b) A minimal-length sequence of up to k splitters (together with partition counts) that ensures all inequality partitions have cardinality no more than b.*

Method:

```
start=0; i=0;
while (start + b) < N {
   if i≥k return error;
   s[i] = r[start+b];
   next = index j of first record after r[start+b] with r[j]>r[start+b], or N if no such j exists;
   prev = index m of earliest record before or at r[start+b] with r[m] = r[start+b];
   count[2i] = prev-start; // Even index counts are for inequality partitions
   count[2i+1] = next-prev; // Odd index counts are for equality partitions
   start = next;
   i = i+1;
}
count[2i] = N-start; // final inequality partition
return s[0],...,s[i-1],count[0],...,count[2i];
```

Figure 1: Algorithm for finding splitters on a single column.

ALGORITHM 2.2.

Input: *Number of records $N \geq 1$, number of keys $k \geq 0$, Records $r[0], \ldots, r[N-1]$ in ascending key order.*

Output: *A bound b and a sequence of up to k splitters that ensures all inequality partitions have cardinality no more than b.*

Method: *UpperB = $\lceil \frac{N-k}{k+1} \rceil$; /* Corollary 2.2*/*
LowerB = 1;
Do a binary search on b between UpperB and LowerB, calling Algorithm 2.1 at each iteration. Move to the upper half when "error" is returned, and move to the lower half when b is feasible. When the smallest feasible value of b has been found, return b and the splitters provided by Algorithm 2.1 for that value of b.

Figure 2: Using Algorithm 2.1 to find the best bound for a given data set when partitioned by k splitters.

and maintaining the full join result. There are several alternative splitter choices, depending on the efficiency goal. In what follows we focus on the inequality partitions, since the equality partitions are easier to process (no key matching is needed) and they can be replicated/divided among processors if they are large [2].

If the partitioning step is a precursor to a hash join, then the goal might be to bound the size of the build partitions so that they fit in the appropriate level of the memory hierarchy, such as the L2 cache. With this in mind, one could use a precomputed splitter set for the smaller of the two tables (say R_2) to partition both tables. (Since the per-record build time is usually more than the per-tuple probe time, and since less memory is required, building on the smaller table is usually preferred.) Even though the size of the R_1 partitions may vary, the R_2 partitions will be bounded in size.

There are two drawbacks to this approach. Firstly, even if R_2 is the smaller table, it may happen than some R_1 partitions are smaller than the corresponding R_2 partitions. In that case, we have perhaps partitioned too finely, and a coarser set of splitters would have been optimal to achieve the given bound. Secondly, this partitioning scheme can lead to highly unbalanced partitions if the distribution of column C values in R_1 is very different from that of R_2. In the context of partitioning for parallelism, such imbalance can lead to unnecessarily long latencies and/or processor underutilization.

Instead, we propose to construct a splitter set based on the sorted C columns from *both* tables. Let N_1 and N_2 be the cardinalities of R_1 and R_2 respectively. We let $r_1[0..N_1\text{-}1]$ and $r_2[0..N_2\text{-}1]$ denote the sorted lists of C values from R_1 and R_2 respectively.

We look for a set of splitters for the combination of r_1 and r_2 based on a *cost function*. For the join example above, a cost model for partitions of R_1 and R_2 with sizes p_1 and p_2 respectively might be:

$$\begin{aligned} cost(p_1, p_2) &= build\text{-}cost(min(p_1, p_2)) \\ &+ max(p_1, p_2) probe\text{-}cost(min(p_1, p_2)) \end{aligned} \quad (1)$$

Equation 1 expresses the preference for building the hash table on the smaller partition. The functions *build-cost* and *probe-cost* may themselves have parameters. For example, the probe cost may depend on the size of the hash table, such as whether it fits into the L2 cache or not. A more practical cost function would also take into account the number of columns in each of the participating tables. Despite the possible complexity of the *cost* function, it should be fairly obvious that any realistic cost function must be monotonic in both p_1 and p_2.

ALGORITHM 2.3.

Input: *Numbers $N_1, N_2 \geq 1$, number of keys $k \geq 0$, bound $b \geq 1$, Records $r_1[0], \ldots, r_1[N_1 - 1]$ in ascending key order and $r_2[0], \ldots, r_2[N_2 - 1]$ in ascending key order.*

Output: *Either (a) "error", indicating that the records cannot be split using k splitters in a way that ensures all inequality partitions have cost no more than b; or (b) A minimal-length sequence of up to k splitters that ensures all inequality partitions have cost no more than b.*

Method:

```
start1=0; start2=0; i=0;
while cost(N1-start1, N2-start2) > b {
   if i≥k return error;
   find the largest q among {r1[start1], ..., r1[N1 − 1], r2[start2], ..., r2[N2 − 1]} such that
      cost(c1, c2) ≤ b, where
      c1 = number of keys less than q among r1[start1], ..., r1[N1 − 1];
      c2 = number of keys less than q among r2[start2], ..., r2[N2 − 1];
   s[i] = q;
   start1 = index j1 of first record in r1 with r1[j1] > q, or N1 if no such j1 exists;
   start2 = index j2 of first record in r2 with r2[j2] > q, or N2 if no such j2 exists;
   i = i+1;
}
return s[0],...,s[i-1];
```

Figure 3: Finding splitters for multiple tables.

ALGORITHM 2.4.

Input: *Numbers $N_1, N_2 \geq 1$, number of keys $k \geq 0$, Records $r_1[0], \ldots, r_1[N_1 - 1]$ in ascending key order, and records $r_2[0], \ldots, r_2[N_2 - 1]$ in ascending key order.*

Output: *A bound b and a sequence of up to k splitters that ensures all inequality partitions have cost no more than b.*

Method: *$UpperB = cost(N_1, N_2)$;*
$LowerB = cost(0, 0)$;
Do a binary search on b between UpperB and LowerB, calling Algorithm 2.3 at each iteration. Move to the upper half when "error" is returned, and move to the lower half when b is feasible. When the smallest feasible value of b has been found, return b and the splitters provided by Algorithm 2.3 for that value of b.

Figure 4: Using Algorithm 2.3 to find the best bound for a given data set when partitioned by k splitters.

Adding more records to the build and/or probe phases can only increase the cost of the join, all else remaining equal. In what follows, we assume only that we are given an integer cost function that (a) is monotonic, and (b) gives equal weight to all records in a table. (Condition (b) is necessary if we wish to compute the cost based simply on counts.)

A set of splitters defines partitions in both $r_1$and r_2. The *cost* function for a partitioning range can be calculated based on the number of records from each list that fall within the partition's range.

Our problem can now be phrased as follows: Find a set of up to k splitters for r_1 and r_2 such that the biggest *cost* among all inequality partitions is minimized. A splitter set is *optimal* if it meets this condition.

For a fixed monotonic *cost* function, we can modify Algorithm 2.1 to give Algorithm 2.3 in Figure 3. We omit the count calculations for brevity.

LEMMA 2.7. *Algorithm 2.3 is correct.*

The asymptotic complexity is slightly higher that Algorithm 2.1, due to the complexity of the step that determines the largest q value.

LEMMA 2.8. *Let $N = N_1 + N_2$ be the total size of the input. Algorithm 2.3 takes time $O(k \lg^2 N)$.*

PROOF. *The number of iterations is at most k. In each iteration, the work is $O(\lg^2 N)$ for the step to find the largest q. One can use an exponentially expanding search followed by binary search to find the appropriate q. Within that loop, we perform a similar search to find the latest value in each input array that is less than q, in order to compute c_1 and c_2.* □

Figure 4 shows Algorithm 2.4 for finding the best partitioning bound for multiple tables.

THEOREM 2.9. *Algorithm 2.4 returns an optimal splitter set.*

LEMMA 2.10. *Let $N = N_1 + N_2$ be the total size of the input to Algorithm 2.4, and suppose that the cost function is bounded above by a polynomial function of its inputs. Then Algorithm 2.4 takes time $O(k \lg^3 N)$.*

PROOF. *The number of iterations is at most $\lg(cost(N_1, N_2) - cost(0, 0))$. For a polynomially bounded cost function, $\lg(cost(N_1, N_2) - cost(0, 0)) = O(\lg N)$. In each iteration, the work is $O(k \lg^2 N)$ by Lemma 2.8.* □

3. REFINEMENTS

A Hierarchy of Splitters

Unfortunately, there is no guarantee that an optimal set of splitters for $k = x$ has any intersection with the optimal set of splitters for $k < x$. So in general it is not possible to take a subset of the splitters to use for smaller partitioning factors.

Fortunately, both the space and time requirements for computing and representing the splitters is linear in k. Thus, given a maximum k value of K, we can compute splitter sets of size $1, 2, 4, 8, \ldots, K$ with total cost approximately double that of computing just the K-splitters alone. Having a set of variable granularities will allow one to partition to the extent needed for the particular operation (and no more).

Incremental Maintenance

The counts can be incrementally maintained over time as database updates occur. A large number of updates might create new heavy hitters and cause the splitters to no longer be optimal. Periodic recomputation of the splitters would be required to bound the divergence from optimality. In situations such as data warehousing that do updates in batches, new optimal splitters could be computed during the batch update window. Despite the possible divergence from optimality, the maintained counts still provide a bound on the size of the inequality partitions.

Higher Dimensions

Histograms over multiple dimensions are used to estimate selectivities and provide approximate query answers when dimensions are correlated (see [10] for a survey). Multidimensional partitioning might also be useful for making database operators more efficient. Examples include partitioning a data set according to a composite key, and partitioning a table on one attribute while at the same time applying a selection condition on another attribute. Finding optimal partitioning elements in two or more dimensions is NP-complete [17], and so we expect to be satisfied with efficiently computable heuristic algorithms.

There are a number of well-known data structures for multidimensional data access [4]. Many of these data structures take balanced partitioning into account when choosing partitioning dimensions and values. One could construct one of these tree-based data structures and store just the partitioning elements from the higher levels of the tree. However, none of these structures pay special attention to equality on the splitter values, meaning that a heavy hitting point (or plane) in multidimensional space could still cause a major imbalance in the tree.

There are several ways one could generalize our approach to higher dimensions. One could find k_x optimal x-splitters and k_y optimal y-splitters in a one-dimensional fashion, and then define a grid of $(2k_x + 1)(2k_y + 1)$ partitions. Alternatively, one could find k_x optimal x-splitters, and then find k_y optimal y-splitters *within* each x-partition. This second approach requires more splitters, but can better handle correlated dimensions. One could even balance the assignment of y-splitters to x-partitions so that x-partitions with a wider y range get more y splitters.

Even though the one-dimensional choices are optimal, these approaches do not guarantee that the cardinality bound of the two-dimensional regions is optimal. The optimality of one-dimensional projections of a multidimensional partitioning structure is as good a guarantee as one can achieve in polynomial time, given the NP-completeness of the multidimensional problem, unless P=NP.

4. EXPERIMENTAL EVALUATION

We implemented the splitter-finding and partitioning algorithms in C++ and performed an experimental evaluation on real hardware, an unloaded Linux server with an Intel Core 2 Duo processor. The specifications of our experimental platform can be found in Table 1. Although the Core 2 Duo processor has two cores, our implementation is single threaded and only one processor core was used during the experiments. For all experiments, the input is memory resident before timing begins.

Processor	Core 2 Duo 2.4 GHz (E6600)
RAM	4 GB
L2 Cache	4 MB
TLB	256 entries [3]
Operating System	Linux (2.6.18 kernel)
C++ Compiler	GCC 4.1.1
Compiler Options	-O3 -funroll-loops -msse2

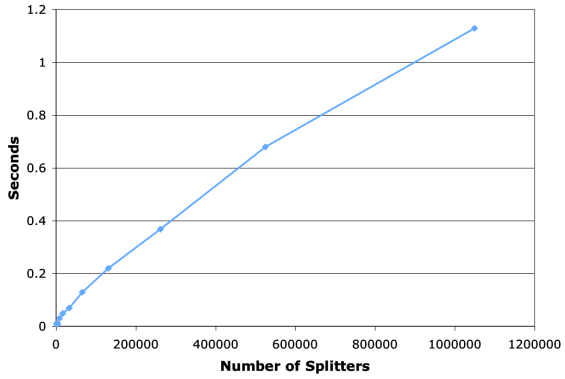
Table 1: Experimental Platform

We conducted experiments with a number of input distributions that are encountered in practice: (1) uniform, (2) sorted, (3) heavy hitter, (4) sequential, (5) zipf, (6) self-similar, and (7) moving cluster. The methods described in Gray et al. [6] were used to generate the probablistic distributions. In all cases, the input consisted of 1 GB of 16 byte tuples, for a total of 2^{26} tuples. Each tuple contains an 8 byte partitioning key and an 8 byte payload, which could be a record ID. For each distribution type we generated input with 2^a unique partitioning key values, where $a = 1, 2, \ldots, 24$.

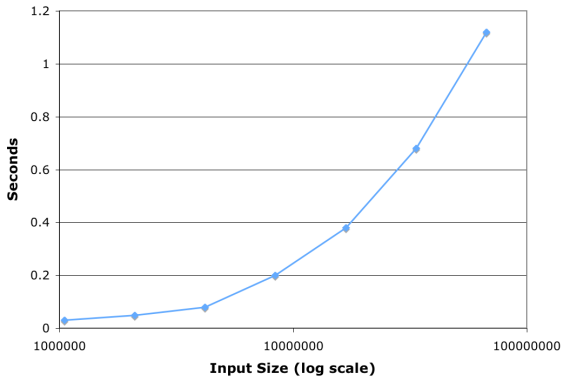
In the heavy hitter input, one value accounts for 50% of the group-by keys, while the other values are chosen uniformly from the other group-by keys. The sequential distribution consists of input records in segments, each consisting of a numerically increasing sequence of group-by values. For example, with 10000 group-by values, the sequence of group-by values would be $1, 2, \ldots, 10000, 1, 2, \ldots, 10000, 1, 2, \ldots$. The self-similar distribution uses an 80-20 proportion, and the Zipf distribution uses an exponent of 0.5. In the moving-cluster distribution with $c \geq W$, record number i is chosen uniformly from the range $\lfloor (c-W)i/r \rfloor$ to $\lfloor (c-W)i/r+W \rfloor$, where c is the target group-by cardinality, r is the number of records, and W is a window size. For $c < W$ moving-cluster reverts to a uniform distribution. We use $W = 1024$.

4.1 The Splitter-Finding Algorithm

We implemented the splitter-finding algorithm in C++ to empirically verify its running time and confirm its

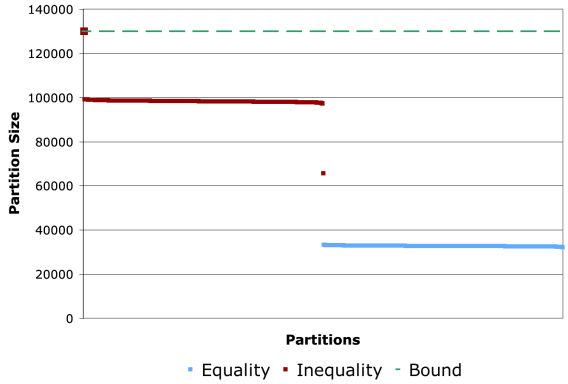


(a) Varying k ($N = 2^{26}$)

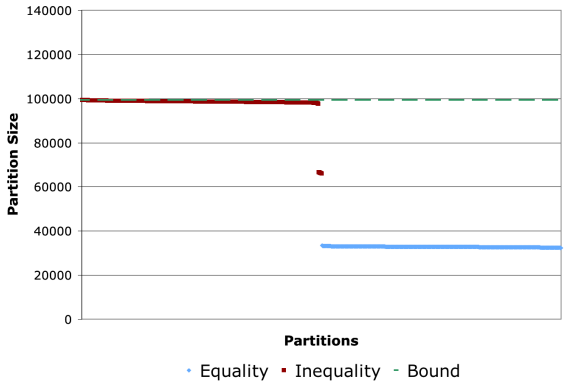


(b) Varying N ($k = 1048575$)

Figure 5: Time to find splitters on uniform input.



(a) 2048 Unique Keys



(b) 2040 Unique Keys

Figure 6: Partition sizes on uniform input using 511 splitters, sorted largest to smallest.

practicability. Figure 5 shows the time required to find k splitters on uniform, sorted input of size N. This time does not include the time to sort the input. The predicted $O(klg^2N)$ running time is confirmed: Figure 5(a) shows the running time to be linear in k and Figure 5(b) shows the running time to be quadratic in lgN. For 1 GB of input, a reasonable number of splitters can be found in one second or less, making splitter-finding a relatively inexpensive database task.

The data structure for partitioning requires an array of the k splitters as well as the starting offsets for each of the $2k + 1$ partitions. Our keys are 8 bytes, so our splitters are 8 bytes as well. We use 4 byte offsets, giving a total data structure size of about $16k + 4$ bytes. For most of our experiments we use $k = 511$, which yields a total size of almost 8KB. The L1 cache size is 32KB, meaning that the search step for partitioning is entirely L1 cache resident. Given that the experimental data sets are 1GB in size, the partitioning data structures for 511 splitters require only 0.00076% storage overhead.

Figure 6 shows the sizes of the partitions created by 511 splitters on uniform input with (a) 2048 distinct keys and (b) 2040 unique keys. The blue data points represent the sizes of equality partitions and the red data points represent the sizes of the inquality partitions. The horizontal, dotted green line shows the maximum inquality partition size bound. For 2048 unique keys, shown in Figure 6(a), the bound does not appear to be very tight, because there are only two inequality partitions whose size is close to the bound. Because the number of splitters is one less than a power of 2 and 2048 is a power of 2, this distribution represents a type of worst case situation. One could attempt to partition using a tighter bound, but it would not succeed. There are 511 equality partitions, which will account for 511 of the 2048 unique keys, leaving 1537 keys to be distributed among the 512 inequality partitions, or just barely more than three keys per partition. Because there are a fixed number of inequality partitions, by the pigeon hole principle one of the partitions would have to hold at least four keys. In contrast, a uniform distribution with 2040 unique keys as shown in Figure 6(b) exhibits a much tighter bound because there are now just under three keys per inequality partition on average.

The robustness of the splitters for distributions with heavy hitters is demonstrated in the case of a Zipf input, as shown in Figure 7. In this case, we see that the largest partitions are equality partitions that effectively capture all heavy hitters. The inequality bound is an order of magnitude smaller than the size of the largest

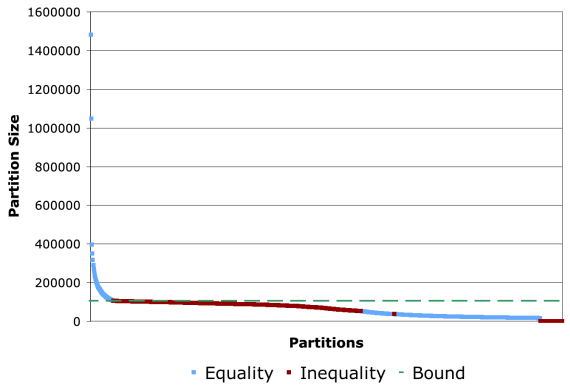


Figure 7: Partition sizes on Zipf input using 511 splitters, sorted largest to smallest.

equality partition. Because there are no or few values between heavy hitter values, some inequality partitions are empty (the flat red line at the right of the chart). We argue that this is a small price to pay for robust guarantees on the size of inequality partitions, as well as the special equality handling of heavy hitters, without needing to know anything about the input distribution.

4.2 Efficient Partitioning

In order to make the partitioning step as efficient as possible, we hand-optimized the partitioning code. We used some of the same optimization methods described by Sanders and Winkel for their sample-sort algorithm [20]: (a) we used conditional move instructions rather than branches, to avoid misprediction penalties and to convert control dependencies into data dependencies; (b) we instructed the compiler to unroll the inner loop (which uses a fixed partitioning depth), and we processed multiple[1] keys at a time to expose a higher degree of instruction-level parallelism; (c) we invoked the compiler with flags allowing it to use instructions (in particular, conditional moves) specific to the hardware platform (see Table 1); (d) we allocated memory to exactly fit the partitions, so no end-of-partition check is needed when incrementing the partition index. However, unlike [20], our algorithm handles both equality and inequality partitions, with no extra work in the inner loop. Our partitioning implementation works for numbers and short ($\leq$ 8 byte) strings. The cost for variable length strings or user-defined data types would be higher. Nevertheless, numeric codes representing row ids are common in databases and are typical of columns on which partitioning would take place. The code for finding the appropriate partition for a key is shown in Figure 8; the generated machine code contained no conditional branch instructions.

In order to ensure that the conditional moves were generated as outlined in Figure 8, using a single compare instruction, we had to write the first three lines of the inner loop in assembly language. We also write to each partition starting at a random location, and then wrap around at the end of the partition. This randomness helps avoid degenerate behaviors, such as repeated conflict misses in the TLB when the partitions have size that is a large power of 2. Other differences from the sample-sort of [20] include: (a) we assume that the splitters and partition counts are precomputed, so we can avoid the sampling and counting steps of sample-sort; (b) our splitters are guaranteed to be optimal; (c) for equality partitions, we avoid the recursive sort step entirely.

Figure 9(a) shows the performance of our partitioning implementation on the various input distributions. The spike in execution times around 511 to 1023 unique keys is due to TLB thrashing. As the number of unique keys in the distributions requires more active output partitions than the TLB can cover, partitioning time increases.[2] The sorted distribution does not experience this TLB coverage problem because consecutive input tuples map to the same partition. Even considering the TLB overhead, partitioning is very efficient on all distributions.

Based on performance measurements, an average of 93 instructions are required to locate an element's partition and then copy it into that partition. On uniform input, our partitioning implementation retires between 1.1 and 1.68 instructions per cycle (IPC), depending on the number of unique keys. This is a reasonable IPC considering that each element processed requires reading from and writing to RAM. An improvement in IPC may be possible, but we leave further optimization to future work.

As the number of splitters used increases, as shown in Figure 9(b), the partitioning cost increases. This is because good partitioning performance relies on fast, cache-resident access to the partitioning data structures, such as the splitters and partition offsets. When that data does not fit in the cache, each tuple processed from a uniformly distributed input causes many cache misses in both the binary search and offset lookup, slowing performance dramatically.

4.3 Using the Splitters to Regenerate a Sorted Data Set

We apply our partitioning implementation to improve the performance of recreating a sorted data set. Suppose that at some point in the past, the data set was sorted and splitters calculated. The data set has since lost that sort property; for instance, it may have been sorted on a different attribute. To sort this data set, we first partition it and then apply stl::sort to each of the inequality partitions. We choose to compare against stl::sort from the GCC STL library, which uses a quicksort variant called Introsort [16], because it is regarded as one of the fastest general purpose sorting implementations [20].

Figure 10 shows the improvement achieved by various numbers of splitters on one uniform data set. The best performance is achieved when the number of splitters is 255 or 511. For 255 and greater numbers of splitters, the inequality partitions fit within the L2 cache of our processor, resulting in cache resident sorting tasks. At

[1]Three keys at a time appeared to work best on our platform.

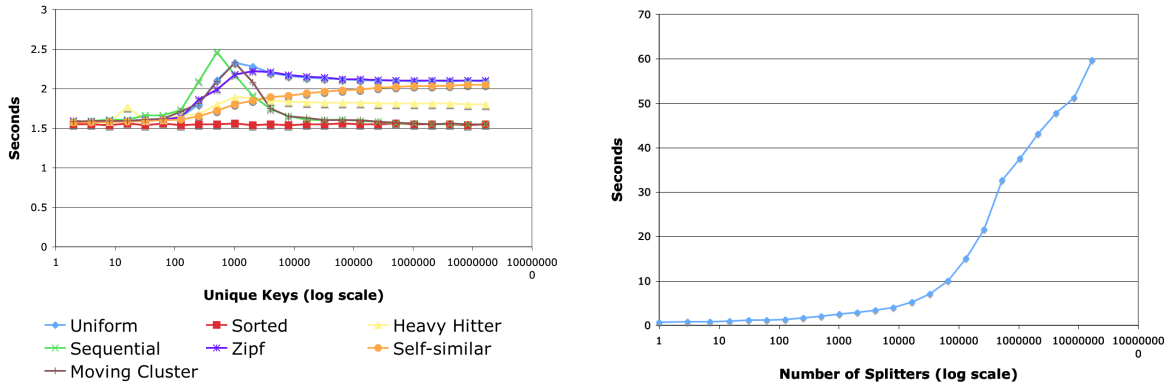
[2]Using large memory pages may help mitigate this problem by increasing the TLB coverage, but such issues are beyond the scope of this paper.

```
int findpartition(N,d,k,p) {
// N = number of keys, one less than a power of 2;  d = lg(N+1);
// k = search key; p[1..N] is the partitioning array, p[0] contains -MAXINT

 low = 0; hi = N+1; mid = hi>>1;  // shift is division by 2

 for(i=0; i<d; i++) {        // loop can be unrolled if d is fixed
   CMP k,p[mid];             // compare key with p[mid]
   CMOVl hi,mid;             // if key smaller, move hi down
   CMOVg lo,mid;             // if key larger, move low up
   mid = (low+hi)>>1;        // if key equal, do nothing to low or hi!
 }
 temp = mid<<1;              // even numbers are inequality partitions
 return temp - (k==p[mid]);  // odd numbers are equality partitions
}
```

Figure 8: Code for finding a partition for a key.



(a) Various Distributions ($k = 511$)

(b) Various k (Uniform keys $= 2^{24}$)

Figure 9: Partitioning performance

511 splitters, the splitter data structures needed for partitioning are also L1 resident, resulting in more efficient partitioning. Increasing the number of splitters beyond this point actually results in lower improvement or even worse performance than stl::sort. This is because the partitioning step becomes more expensive as the splitter data structures cease to be L1 and then L2 cache resident as well increased cache pressure caused by more output partitions. For all subsequent experiments, 511 splitters are used.

Figure 11 shows the improvement of our partition-then-sort approach to simply sorting the entire data set with stl::sort on multiple input distributions. To provide an idea of sort cost on our experimental platform, consider the case of sorting 1 GB of input with 2^{24} unique keys. Sorting with stl::sort takes 10.5 seconds. In contrast, our sort-then-partition approach takes 7.89 seconds in total, with 2.09 seconds spent partitioning and 5.8 seconds spent sorting the inquality partitions with stl::sort.

The partition-then-sort approach performs well, particularly if heavy hitters are present in the input. For instance, regardless of the distribution, when the number of unique keys is less than the number of splitters, partitioning will place almost all tuples into equality partitions that do not require further sorting. In Figure 11, this is the reason for the better than 60% improvement over naive stl::sort for all distributions when the number of unique keys is less than 511. The heavy hitter, Zipf, and self-similar distributions show an improvement of at least 25% in all experiments. This is due in large part to the guarantee that heavy hitters will be placed in equality partitions that do not require further sorting.

Sorted and moving cluster benefit the least from partitioning. In the case of sorted input, partitioning is unnecessary, but it does provide a small benefit by creating smaller chunks of the data set to be sorted, resulting in better cache usage. The clustering present in the moving cluster input provides some of the benefit of partitioning, resulting in faster sort times that make it more difficult for partition-then-merge to show an improvement. This is because elements in the moving cluster data set are already close to their final sorted partition. Therefore, initial quicksort partitioning passes do not need to swap elements, leading to fast execution because there is good branch prediction and no data movement. When elements must be swapped, they are only moved within a small region of the input, which is more likely to be cache resident.

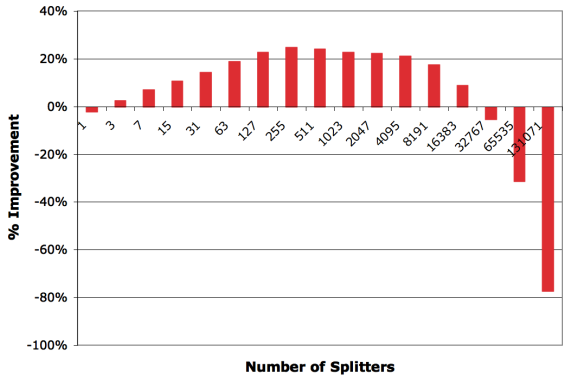


Figure 10: Partition-then-sort improvement over stl::sort (2^{24} unique keys).

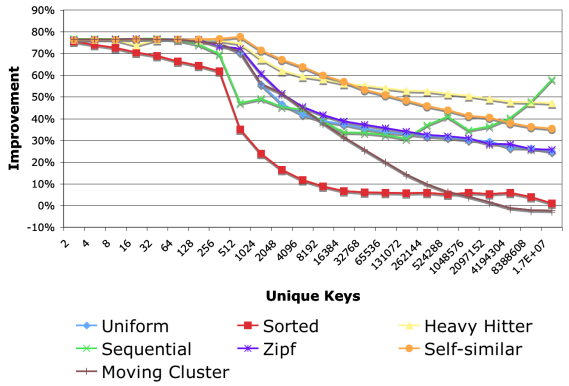


Figure 11: Partition ($k = 511$) and sort performance improvement compared to stl::sort.

After initially falling, the partition-then-sort performance improvement on the sequential distribution increases as the number of unique keys in the data set becomes quite large. This is likely because stl::sort, which uses Introsort [16], chooses poor pivots. Introsort may detect the poor pivot choices and switch to another sorting algorithm (heapsort) that is less efficient than quicksort, but does not have as bad a worse-case running time.

In comparison with Sanders and Winkel [20], our partition-then-sort implementation performs better (to the extent that their Pentium 4 results may be compared with our Core 2 Duo results). One expects this improvement because we do not require a sampling or counting step (the splitters are precomputed), and because we have special handling for equality partitions, which do not need to be sorted after partitioning. Also, unlike [20], we our evaluation considers a variety of data distributions.

5. RELATED WORK

There has been much work on histogram construction for database applications [10], but most of this work has been focused on the problems of approximate query processing [12] and/or selectivity estimation [1]. As a result, the desired error metric may be different. For example, Ioannidis and Poosala define a notion of "V-optimality" based on the sum-squared error [11]. Jagadish et al. provide a dynamic programming algorithm for calculating histograms that are optimal according to an arbitrary error metric [13]. However, this algorithm takes time $O(kN^2)$ in the worst case, making it impractical for large data sets containing tens of millions of records. Also, it does not do any special processing for heavy hitters. Compressed histograms [19] set aside some space to keep values with high frequencies in singleton buckets. However, compressed histograms have not previously been used to save work in database operators for singleton partitions. Muthukrishnan et al. have shown that optimal rectangular partitioning in two or more dimensions is NP-hard [17].

Poosala and Ioannidis study histograms for estimating the size of a join result [18], and show that the lowest sum-squared error is achieved using two V-optimal histograms on the input relations' join columns. They also use a cost function to compute balanced partitions for a parallel join. Their cost function uses the two separate V-optimal histograms on the participating tables to determine the workload distribution. For us, the cost function is used to determine a single splitter set for the tables considered jointly. Further, our notion of optimality that is based on the maximum partition cost more directly matches the load-balancing application, where one has to wait until the termination of the slowest partition. Most other previous work on load balancing joins for parallelism handles skew only on the build relation [14, 8, 2]

Materialized views make queries faster by allowing the query processor to avoid recomputing certain subexpressions of the query. The cost of a materialized view is the space needed to store it, and the time needed to keep it (and its indexes) up to date. Our splitter set can be thought of similarly. The splitter set allows some query operations (such as sorts and joins) to be performed faster than they could before. The maintenance of the counts for a splitter set is straightforward, and the space occupied is (in most practical situations) negligible relative to the size of the original table.

6. CONCLUSION AND FUTURE WORK

In this paper we have introduced a novel method for efficiently extracting optimal splitters for database partitioning from a sorted data set. We use these splitters to create both equality and inequality partitions, and guarantee size bounds on the inequality partitions. The data structures required to support splitter calculation and partitioning require a very small amount of storage overhead, making the calculation and storage of optimal splitters a low cost operation. These optimal splitters can be used to create partitions with guaranteed size bounds, improving the performance of sorting, joins, aggregation and parallel processing.

We have validated the running time and partition bound guarantees experimentally, by implementing these algorithms on real hardware. We also present an efficient partitioning implementation that avoids conditional branches when calculating which partition an element belongs to. We further demonstrate that the splitters are robust regardless of the distribution of the data

set. Finally, we use the optimal splitters and our partitioning implementation to improve the performance of sorting various 1GB data sets by up to 76% over stl::sort.

One avenue for future work is to extend partitioning with size bounds to hash partitioning. Perhaps one could place heavy hitters in special buckets within the hash table. Nevertheless, creating hash buckets that are optimally balanced seems like a difficult problem. We also plan to develop an efficient parallel implementation of the splitter-finding and partitioning algorithms for use on a multi-core architecture.

7. REFERENCES

[1] Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.

[2] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, pages 27–40, 1992.

[3] Jack Doweck. Inside intel core microarchitecture. Viewed on-line December 2007. http://www.hotchips.org/archives/hc18/3_Tues/HC18.S9/HC18.S9T4.pdf.

[4] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[5] Shahram Ghandeharizadeh and David J. DeWitt. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In *VLDB*, pages 481–492, 1990.

[6] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD Conference*, pages 243–252, 1994.

[7] Peter J. Haas and Arun N. Swami. Sampling-based selectivity estimation for joins using augmented frequent value statistics. In *ICDE*, pages 522–531, 1995.

[8] Kien A. Hua and Chiang Lee. Handling data skew in multiprocessor database computers using partition tuning. In *VLDB*, pages 525–535, 1991.

[9] J.S. Huang and Y.C. Chow. Parallel sorting and data partitioning by sampling. In *Proc. of Symposium on Parallel Algorithms and Architectures*, pages 627–631, 1983.

[10] Yannis E. Ioannidis. The history of histograms (abridged). In *VLDB*, pages 19–30, 2003.

[11] Yannis E. Ioannidis and Viswanath Poosala. Balancing histogram optimality and practicality for query result size estimation. In *SIGMOD Conference*, pages 233–244, 1995.

[12] Yannis E. Ioannidis and Viswanath Poosala. Histogram-based approximation of set-valued query-answers. In *VLDB*, pages 174–185, 1999.

[13] H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel. Optimal histograms with quality guarantees. In *VLDB*, pages 275–286, 1998.

[14] Masaru Kitsuregawa and Yasushi Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (sdc). In *VLDB*, pages 210–221, 1990.

[15] Conrado Martínez and Salvador Roura. Optimal sampling strategies in quicksort and quickselect. *SIAM J. Comput.*, 31(3):683–705, 2001.

[16] David R. Musser. Introspective sorting and selection algorithms. *Softw., Pract. Exper.*, 27(8):983–993, 1997.

[17] S. Muthukrishnan, Viswanath Poosala, and Torsten Suel. On rectangular partitionings in two dimensions: Algorithms, complexity, and applications. In *ICDT*, pages 236–256, 1999.

[18] Viswanath Poosala and Yannis E. Ioannidis. Estimation of query-result distribution and its application in parallel-join load balancing. In *VLDB*, pages 448–459, 1996.

[19] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD Conference*, pages 294–305, 1996.

[20] Peter Sanders and Sebastian Winkel. Super scalar sample sort. In *European Symposium on Algorithms*, pages 784–796, 2004.

[21] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented dbms. In *VLDB*, pages 553–564, 2005.

APPENDIX

A. SUPPLEMENTARY PROOFS

DEFINITION A.1. *A splitter s_i in a splitter set $s_0, \ldots, s_m$ satisfies a bound b to the left in a sorted array r, if the inequality interval immediately to the left of s_i in r (either the leftmost inequality partition if $i = 0$, or one defined by s_{i-1} and s_i) has cardinality at most b.*

LEMMA A.1. *Algorithm 2.1 satisfies the following loop invariant immediately before the if statement: Either $i = 0$, or there is no set of i splitters for r[0],...,r[N-1] that extends further to the right than s[i-1] while satisfying the bound b to the left of each splitter.*

PROOF. *The proof is by induction. The base case $i = 0$ is trivial. Suppose the invariant holds for $i = m \geq 0$. We show it also holds for $i = m + 1$. By the induction hypothesis, either $m = 0$, or $m > 0$ and no set of m splitters that satisfies the bound b on the left of each splitter, extends beyond s[m-1].*

If $m > 0$, let j be the index of the first element r[j] with r[j]>s[m-1], as in Algorithm 2.1. The maximal extension beyond s[m-1] (that satisfies the bound b between s[m-1] and s[m]) occurs when s[m]=r[j+b] as ensured by Algorithm 2.1. If a smaller value of s[m] was chosen, it would not extend as far to the right in r. If a larger value of s[m] was chosen, then the inequality partition between s[m-1] and s[m] would violate the bound b. The reasoning for $m = 0$ is similar, considering the leftmost inequality partition. □

PROOF OF LEMMA 2.1: *Algorithm 2.1 is guaranteed to terminate because i is incremented on every iteration, and the loop terminates (at the latest) when $i \geq k$.*

We first need to prove that when Algorithm 2.1 produces a set of i splitters, that there are at most k of them, that i is minimal, and that the inequality partitions they define satisfy the bound b. (The correctness of the counts is straightforward.) By construction, we can only exit the while loop when $i \leq k$, so there are at most k splitters returned. Also by construction, the splitters are located at most b elements to the right of the first element non included as a previous splitter. Thus, all ranges to the left of a splitter satisfy the bound. The minimality of i follows for Lemma A.1. Finally, the loop ends when there are no more than b elements remaining to the right of the final splitter. This concludes the first part of the proof.

We now need to prove that when Algorithm 2.1 returns "error", it is impossible to find any set of at most k splitters that satisfy the bound b. If $k = 0$, then the algorithm returns error if and only if $b < N$, which is correct. For what follows, we assume $k > 0$.

Assume that Algorithm 2.1 returns error, which must happen in the if statement. Since $k > 0$, $i = k > 0$ and by Lemma A.1, there is no set of k splitters for r[0],...,r[N-1] that extends further to the right than s[k-1] while satisfying the bound b to the left of each splitter. By the while condition, (start+b)<N, and so there are more than b elements in the rightmost inequality partition. Thus no set of k splitters that satisfy the bound b exists. □

PROOF OF LEMMA 2.7: *A result analogous to Lemma A.1 holds, due to the monotonicity of the cost function. The remainder of the proof is similar to the proof of Lemma 2.1.* □