Implementation and Tuning of a Multi-Core Version of the Radix-Clustering

Gabriel Campero
University of Magdeburg
gabrielcampero@gmail.com

Angel Yordanov
Technical University of Sofia
angel.yordanov@fdiba.tu-sofia.bg

Svetoslav Ermakov
Technical University of Sofia
svetoslav.ermakov@fdiba.tu-sofia.bg

***Abstract* — In this research we examine alternatives for designing a multi-core version of the radix-join algorithm. More specifically, we focus on its most time-consuming stage: partitioning. Traditional parameters are discussed here for their capabilities of allowing performance-tuning. Considering that different implementation alternatives have an impact on performance too, we suggest in this study that they could also be used for tuning. Concretely, we propose schemes of affinity setting and adjustable parallelism as ways for tuning with thread-placement strategies and task-based parallelization alternatives. A prototypical implementation using adjustable parallelism was evaluated for varying bits per pass and number of threads. Our results confirm the potential for speedup and higher throughput of the parallel algorithm. Good memory use was also observed within the limited scope of our tests, suggesting that the cache-conscious properties of the serial algorithm might be preserved when using task-based parallelism. The suboptimal speedups and reduced scalability observed in our study suggest there's still space for further improvement through better implementations and more careful tuning of the algorithm.**

Keywords—radix-join; parallelising; CoGaDB; radix clustering;

I. INTRODUCTION

In recent years, the hardware industry has produced a significant number of developments, offering vast possibilities for improving the performance of traditional programs. Nowadays, off-the-shelf CPUs provide various levels of parallelism by supporting single instruction over multiple data (SIMD) processing or simultaneous multi-threading (SMT). General-purpose computing with GPUs, multi-cores and other novel architectures are quickly becoming a mainstream practice. To gain from the new trends in hardware, traditional DBs have required a great deal of redesign and refinements from the database community. One of the major areas of research for the community has been query optimization in regards to the novel hardware.

Since all database queries are necessarily translated into relational operators (i.e. sort, join, etc.), with several alternative physical realizations, the evaluation and selection of an efficient physical operator, in regards to the input, computing platform and existing workload, is one of the key current challenges for DBMS.

In this paper we'll address several aspects from such a challenge, by studying one of the most common and costly relational operators: the join operator; and one of its specific physical realizations: the radix-join algorithm.

Ever since its introduction in a serial version, almost 15 years ago [1], the radix-join algorithm has stood out from among the 2 most commonly used joins (sort-merge join and hash-join), for its memory access pattern, presenting the inherent capability of making an efficient use of caches and the TLB. For this reason it is considered a cache-conscious algorithm. Given the high penalties incurred by cache-misses, this property makes the algorithm quite advantageous.

Similar to partitioned-hash join, the radix-join algorithm consists of three stages: partition, build and probe (Figure 1). The partitioning stage embodies the essence of the algorithm. In this stage the input relations are scanned and clustered through several passes, by considering in each pass a number of its most significant or less significant bits. The second stage is optional, in this part the resulting partitions can be explored to build a hash-table by using the bits defining membership to each partition. The final phase (Stage 3) uses the hash-table, or alternatively a nested-loop, to probe the partitioned relations and perform the join.

The canonical algorithm offers some variables for tuning the algorithm to performance: the number of passes (*P*), the total number of bits considered (*B*), the number of bits considered per pass (*Bi*) and the resulting number of clusters (*H*). These variables are highly related to each other, through the following relations*:* $\sum i=P$, $\sum Bi=B$, $H=2B$.

Past research covering serial implementations have shown great benefits from adapting the cluster number and sizes to fit the TLB and caches [2].

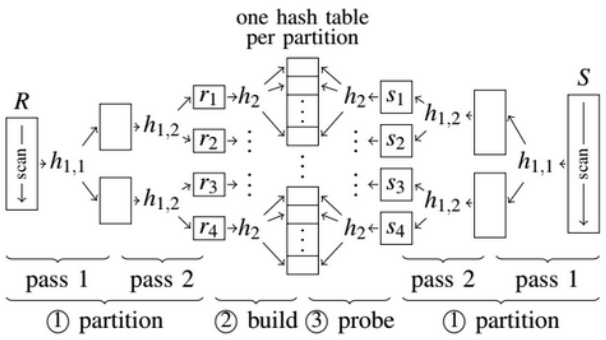

Figure 1: Stages of Radix-Join [6]

Several studies have focused on exploiting the data parallelism between partitions, in order to present parallel implementations. For them, a new configurable variable appears: number of threads used [6], [7]. These studies follow similar parallelization patterns, and thus perhaps there might be space for studying other implementation alternatives. In this study we will consider some of them for multi-core platforms.

Additionally, several configurable variables of the algorithm seem to have been neglected in previous studies - mainly, the effect of different distributions of bits per pass. This aspect might also be of interest for research.

Some studies have also claimed to find optimal configurations of the radix join capable of outperforming other join alternatives [6], but currently there is no clear consensus on a guideline for achieving the optimal performance through implementation alternatives and configurable variables

One overriding conclusion of studies so far is that for most cases, the partitioning stage requires more execution time than others. Furthermore, it has been observed that as *H* diminishes, the partitioning phase requires less time, but the join becomes more costly. So a tradeoff must be reached between both over this value. Given this insight it would be of very valuable to develop a very fast radix-partitioning stage, which allows a large number of clusters. With the aim of contributing to this goal, we present our study of the radix-partitioning stage.

In order to study the parallelization alternatives of the algorithm for multi-cores we implement a prototypical parallel version based on previous studies [2], [7], but instead of exploring highly specialized optimizations [6], we propose a novel parallelization alternative from the second pass onwards. Since our proposal is based on the idea of adjustable join parallelism [9], which consists on adjusting the parallelism to the partitioning, we call our proposal: adjustable parallel radix-clustering.

Through our presented prototype we aim to facilitate further analysis on both configurable variables and implementation aspects, and their effect on performance.

The remainder of this paper is organized as follows: first, we introduce related work in the field. In this way we aim to establish some key terms that will be used for the subsequent section where the radix-join algorithm and its configurable variables are analyzed in further detail. Next, we discuss some considerations regarding the implementable parallelization. At this point we also present some of our original ideas for tuning to the underlying hardware. Then, we describe our prototypical implementation of the algorithm, as well as our experimental setting. Following it, we assess the results from our experiments and, to conclude the paper, we examine some potentially rewarding areas for further research.

II. RELATED WORK

In this section we describe works by others that are related to our current project. We begin by introducing the earliest found mention to the radix-join concept and the early evaluations of cache-consciousness for database operators by Shatdal el al. [3]. Next we present the original radix-join implementation by Manegold et al. [2], and some additional ideas that they have developed [1], [10]. Then we'll introduce a study considering the parallelization of partitioning algorithms, moving on to some key studies on parallel versions of radix-join. Finally we mention a small number of research offering observations that can be of help for studying the performance of the partitioning stage of radix-join.

The first published proposal for designing an algorithm similar to radix-join arose from studies covering the traditional radix-sort algorithm. In radix-sort, keys are compared in a multi-pass way, according to a radix *r*. The most common and by default radix value is the binary or bit radix r=2. The passes are done whether starting from the most significant or the less significant bit of each key. In each pass the keys are partitioned according to a consecutive number of their i-th bits. This goes on, for a number of passes until all bits of every key are considered. The result is an ordered array. The earliest proposed idea for radix-join was, to sort the keys by using this algorithm and then to perform a join operation over the resulting keys [11].

Another important study that influenced the creation of the radix-join algorithm was done by Shatdal et al. [3]. By examining different join and aggregation operations in terms of their cache performance, they discovered the positive impact that a cache-conscious implementation can have on improving the performance of database operators. Observing that traditional methods incurred in may cache misses, they proposed a cache-conscious variant of the partitioned hash-join. In it, both input tables are initially partitioned in such a fashion that the hash tables which are later built for them, can fit into the cache, thus reducing the number of cache misses of the algorithm. More specifically, they used a technique called AlphaSort [12] to distribute the data in partitions, as in external sorting, while increasing in-memory sorting speed. Despite the overhead introduced by the partitioning operation, their tests found significant improvements by using this algorithm over a regular bucket chained hash join. They concluded that when partitions to be joined are small and numerous enough, so each can fit in the cache, the join algorithm performs much faster by incurring in less cache misses, hence that it might be worthy to trade more CPU cycles (required for the partitioning stage) for a better cache use during the probe stage.

Building on the work of Shatdal et al. [3], Manegold et al. [1] presented the first proposal for the radix-join algorithm. The main component of the algorithm is the configurable partitioning stage, also known as radix-clustering. In this operation, both tables are partitioned during *P* passes by using radix-sort, but without necessarily considering all bits.

The fact that radix-sorting is stable implies that whatever partitions are obtained from running the operation on one table are hash-equivalent to the corresponding partition generated by the same operation on the other table. This interesting behavior allows joining the results by easily building a hash-

table over each partitioned relation, and then doing a hash-join. However, in the evaluation of Manegold et al. [1] this is not considered, for they omitted the build stage, and instead of hashing opted for following the partitioning with a nested-loop join, not using structures additional to the partitioned input itself.

In the serial implementation this partitioning is done by performing a first partitioning pass over all the input, clustering the keys according to a number B_i of bits. For the next pass the function calls itself recursively, in a depth-first fashion, over each cluster generated in that pass, considering the next set of B_{i+1} bits. The algorithm stops when both the full number of passes *P*, and the total number of bits considered *B* are reached.

By performing a depth-first recursion after the first pass, the partitioning stage of the algorithm can tend to have a good memory-access pattern, because it clusters together memory addresses that will later be accessed together within the algorithm, thus tending to reduce cache misses.

The tests of Manegold et al. [1] also confirmed the findings of Shatdal et al. [3]. If the generated partitions fit into the cache, the cache misses can be reduced, leading to performance gains. They further developed this idea by considering the translation look-aside buffer (TLB) during the partitioning stage. Usually the TLB is used for improvement of virtual transaction speed. Nearly all modern hardware relies on TLBs for paged or segmented virtual memory use TLBs. The authors evaluated that when the number of clusters *H* exceeds the number of TLB entries, performance suffers a sharp decrease. This led them to propose a performance tuning improvement of their algorithm by limiting the number of clusters in each pass H to the size of the TLB. The number of available TLB entries can be considered the maximum potential partition number that can be accessed simultaneously without loss of efficiency.

Additionally they present cost functions to model the behavior of the partitioning stage. These will be discussed further ahead.

The complexity of the radix-clustering has been modeled as $O((|R| + |S|)\ P)$ [6]. *R* and *S* stand for both relations to be joined.

In a subsequent study Manegold et al. [10] used hardware counters to measure the performance of the radix-clustering for one pass, and different number of bits considered. They measure a breakdown of the execution time, and establish that actual computation tends to represent most of the time, while there is a tendency for low overheads related to memory misses. However time consumed by handling more TLB and cache misses does indeed increase with rising numbers of partitions generated for that one pass. On the other side, rising number of partitions can improve the probe stage, as Shatdal et al. showed [3]. Therefore a tradeoff must be reached regarding this value, with better performance in the probe stage meaning a potentially more costly partitioning stage.

Radix-Declustering, an additional optimization for the required tuple materialization that follows the join itself was also introduced by Manegold et al. [10]. In the context of this work they also propose optimal configurations for total bits considered *B*, w.r.t the expected cardinality among input relations.

The research of Cieslewicz and Ross [13] highlights some meaningful observations on the different performances of alternative choices for processing in parallel the output array of a partitioned hash-join. While the scanning of the input array can easily parallelized, the authors identify that the writing of the output partitions can be done in at least 4 ways: by writing to independent copies of the output, by doing a first scan determining the write positions and then a second scan for performing such write (this alternative is called count-them-move), by synchronizing through locks the writing access to the output array or by the use of parallel buffers. In their experiments the count-then-move method displayed, for small number of partitions, a lower throughput than that of parallel buffers and independent copies, but as the number of partitions increased beyond the number of threads used, it showed a throughput only surpassed by that of the concurrent method. When the number of threads matched the number of partitions, the performance of the count-then-move method proved to be similar to the one from other methods, but with different behaviors for different page sizes.

Significantly, their evaluation also confirms results from other studies showing that the performance of the 4 schemes is limited by the big performance impact of increasing the number of partitions beyond the size of the TLB.

Another meaningful observation from their research has to do with scaling. All techniques stop scaling once cache thrashing, produced by the memory use of all threads, starts occurring. However without thrashing, no technique managed to display a linear scaling, and for most techniques their ability to scale lowered as partition numbers increased.

With regards to multipass partitioning, the authors show that for a large number of partitions, a higher throughput can be achieved by performing more passes using the parallel buffers and concurrent write strategies, whereas for these strategies performing more passes with a lesser number of partitions does not lead to throughput improvements. The performance during multipass partitioning is not studied by them for the other techniques.

Parallel versions of the radix-join algorithm have been studied in the context of 2 important debates: understanding if sort merge join can achieve better performance than partitioned hash join, or if the hardware conscious radix join is better than hardware oblivious alternatives.

While comparing sort merge join to partitioned hash join, Kim et al. [7] introduced a parallel version of radix clustering. In their implementation they use the aforementioned count-then-move strategy for writing to the output buffer. Among their results, achieved by using a small number of threads, they confirm the number of partitions/performance of the join stage tradeoff. They also present some considerations on the effect that skewness in the input data can have over the performance of the algorithm.

Another parallel version of radix-join was implemented by Blanas et al. [4], in the context of comparing it to a hardware

oblivious non-partitioned hash join. They study a partitioned hash join using the independent output writing strategy and the concurrent write strategy. For the later strategy they find a poor performance, and claim that it has to do with higher penalties for synchronization in SMT processors. The cache-conscious radix-join is found to underperform the hardware oblivious non-partitioned alternative, because of this they claim that some new architectures are good enough at hiding latencies and thus, display no benefits from partitioning to improve memory use in join operations. Further research has contradicted some of these findings, by suggesting that their results are due to a narrow combination of parameters and architecture, and fail to reflect the expected performance of radix-join [6].

In their study of a parallel radix-clustering Teubner et al. [6], found that in several architecture its throughput under most configurations and its scalability within the number of available hardware contexts, can be significantly higher than the observed for hardware-oblivious approaches. These results are obtained with an implementation that uses a count-then-move strategy, and additional improvements during the partitioning stage such as prefetching and load-balancing through task queuing. The authors prove that the parallel-radix join is fairly robust to a wide range of skew and to a range of parameter misconfigurations such as some partitions being bigger than cache size, or the number of partitions being larger than TLB. Although performance of a parallel non-partitioned join can be better for small table relations (when an input table is significantly smaller than other), the parallel radix-join algorithm proves to be robust over a wide range of relation size ratios. Finally, the authors also propose that cache use can be improved if the build and probe stages are done successively per each partition, instead of the less efficient alternative of processing the build stage over all partitions and then the probe stage over all.

Optimization possibilities of the partitioned hash-join by the use of pre-fetching have also been studied [14], [15].
The effect of data skewness on the performance of radix-join has been studied by Albutiu et al. [16] and Yadan et al. [5]. The later have presented some improvements by adopting different strategies based on data size at runtime.

III. FACTORS AFFECTING THE PERFORMANCE OF RADIX-JOIN

Building on the preceding section, we can identify 3 meaningful performance measures to evaluate a parallel version of radix-clustering: scalability, speedup and throughput.

Previous studies have shown that platform characteristics such as the design of the memory system, number of available hardware contexts, the current load of the system, and threading start-up/management costs can influence the values of these performance measures for the algorithm. Characteristics of the input data can have a significant effect over the performance of the algorithm too. Data size, skewness, cardinality and relation size ratio are some of these factors.

In order to tune the performance of the algorithm to platform and input characteristics, the use of configurable parameters of the algorithm and different implementation alternatives have been considered.

In the next section we discuss some factors with tuning potential for the algorithm. We conclude the section by proposing the idea that implementation alternatives could also be offered in a configurable manner, so the algorithm can be gainfully adapted at call-time to different input and platform characteristics; in particular we suggest a scheme of adjustable clustering parallelism and affinity setting options. In future research more configuration alternatives might be explored.

A. *Configurable parameters*

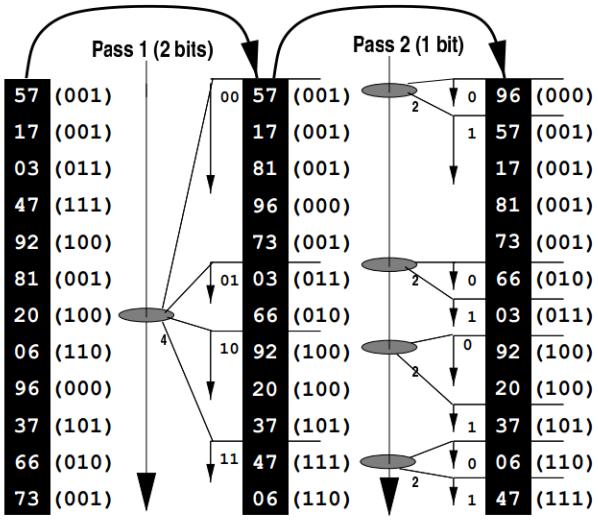

Figure 2: 2pass/3bit Radix Clustering [2]

Following the canonical radix-clustering algorithm introduced by Manegold et al. [2] (Fig. 2.), 4 variables can be presented as configurable parameters of the algorithm, having an impact on its execution and performance. These variables are: the number of passes (*P*), the total number of bits considered (*B*), the number of bits considered per pass (*Bi*) and the resulting number of partitions (*H*). As was mentioned in the introduction, these variables are inextricably related to each other, through the following relations: $\sum i=P$, $\sum Bi=B$ and $H=2B$. Additionally if Hi is the number of clusters in a given pass, we have that $H_i=2B_i$ and $H=\prod H_i$.

Both the distribution of partitions *Hi* and *Bi* define the fan-out of the algorithm. It has been observed that for performance tuning of the partitioning step, the number of partitions in any pass should not exceed the number of TLB entries. This can be easily guaranteed by making the resulting number of clusters fit into the TLB. Given that the resulting number of partitions can be the same while having different *Hi* and *Bi* distributions, it is plausible to study what effect different distributions of these values might have on the performance.

Manegold et al. [2] suggests that a uniform distribution of B_i implies a more even processing and thus is to be preferred.
For improving the join stage, each resulting partition should also fit in the TLB and L1 cache, next to the corresponding partition from the other input [2]. To achieve this, the creators of the algorithm have suggested the tuning of *H* to the cardinality divided by a small constant. For unskewed data, the existence of more partitions at this stage implies a lower size for each partition, and in turn a lower cost for the join stage. For a higher number of partitions performance can be expected to decrease during the partitioning stage, and improve during the join stage. To reach an agreeable compromise it must be necessary to clearly establish the effect of both given a platform and input data.

A defining issue for scaling up the algorithm would be the efficiency of the platform in using additional TLBs and handling memory transfers. If this is achieved, and under conditions of low overhead or thread startup/management, then more partitions might useful if they can be exploited for additional parallelism up until the number of available hardware contexts, or until memory thrashing starts to occur.
Apart of the effect that the choice of *B* might have on the number of clusters, it can also have an important effect in improving the memory performance of the join stage for certain column projections. For this case, it has been proposed that a proper tuning of *B* could allow a performance similar to the one of a sorted join but at a much lower cost; specifically it was suggested the tuning of this variable to $B=1+log2(|column|)-log2(C/⌉column)$ [10].Here *C* is the cardinality, |column| - the number of tuples in the column and ⌉column - its bit-width.

The reasoning behind adjusting this variable can be illustrated through an example: If the cache size is of 64 KB, and the tuple values are of 4 bytes, then 16.384 tuples would fit in the cache. If the input size is of 10M tuples, 2^{10} partitions would be necessary for an average size of 10.000 tuples, assuming unskewed data. To achieve this, the total number of bits *B* can be adjusted to be at least 10.
With regards to the last parameter, an increase in the number of passes *P* implies more instructions executed, but also, potential improvements of the cache use of the algorithm, by reducing partition sizes under the assumption of unskewed data. Therefore this parameter could be valuable for performance tuning, and the experimental study of its properties for parallel versions would also be of interest.

B. Implementation alternatives

Join algorithm

Perhaps the most evident alternative for implementing the radix-join refers to the choice of the build and join-algorithms used for the stages following the partitioning. Traditionally hash-join and nested-loop join have been used. It might be worthwhile to expand the scope of possible choices, by testing other join algorithms capable of benefiting from the partitioning.

Parallelization alternatives

Multi-threaded parallel versions of radix-clustering can opt to assign different tasks to each of the used threads. Through literature research and the study of open-sourced versions, we've identified some of the possible parallelization choices. They can be classified by their being based on task-parallelism or on data-parallelism. For the case of this specific algorithm, the later exploit parallelism beginning at the scanning of the input, while the former intend to exploit parallelism at the moment when recursive calls are made.

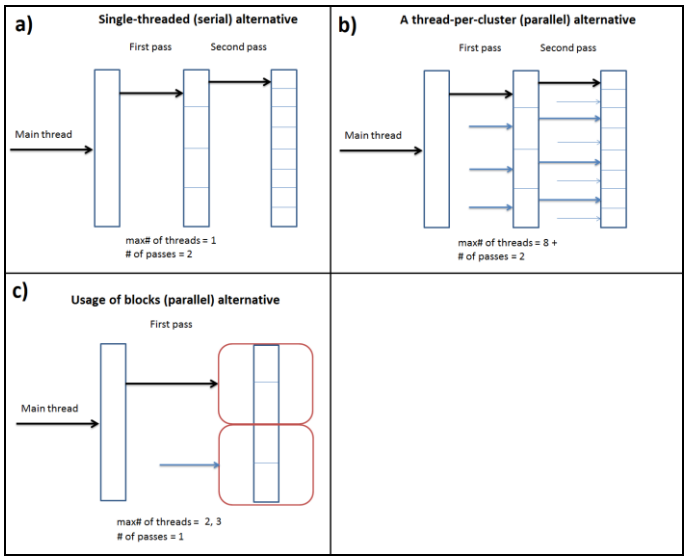

Figure 3: Parallelization alternatives

In order to execute additional passes after a given pass of the radix-clustering, recursive calls to the algorithm over each partition have to be made. Task-based alternatives target the possibility of executing those calls in additional threads. For this strategy to be efficient, the current thread can run by itself one of such recursive calls.

We've found at least 4 alternatives based on task-parallelism (Figure 3):

a) no-parallelization: the algorithm would run on a single thread. After performing a first clustering pass over the given input, it would call itself recursively over each generated cluster.

b) per-cluster parallelization: one-thread-per-cluster would handle the recursive call to a new pass over a given cluster generated by the first pass.

c) per-block parallelization: each thread would handle a number n of recursive calls for a new pass over a given number n of the currently generated clusters. This can be called an n-block parallelization, with n being the number of clusters/recursive calls executed by each thread.

An idea similar to per-block parallelization has been suggested by Rodiger et al. [17]. They use this strategy to group together the processing of small partitions, achieving a better work distribution for skewed data. They call this method adaptive radix-partitioning.

d) several-threads-per-cluster parallelization: several threads are assigned to handle a recursive call over a cluster. In order to realize this, one thread would handle the call, while the other threads would be passed as a thread queue for use during this next recursive call.

Additionally, we've found 2 parallelization possibilities based on data-parallelism:

a) sequential scanning, in this alternative the input tuples are scanned sequentially, without parallelization.

b) range-partitioning: this is perhaps the most common choice for parallelizing the first pass. In this setting each thread is responsible for scanning, clustering and writing to the output array, a given number of tuples from the input array. So as to continue processing successive passes, threads under range-partitioning must turn to one of the task-based granularity choices.

In several parallel versions [7], [6] the number of threads used for range-partitioning of the first pass are usually the same as the number of generated clusters in this pass, so then each thread can run the recursive call over a cluster, which corresponds to per-cluster parallelization.

For increments in the number of passes *P*, the mentioned alternatives for parallelization exhibit coarser granularities if done in the first pass. When called from successive passes, the granularity is lower since the work by the thread extends from its call, up until the last pass. If there is a large cost for thread startup and management, coarser grain parallelization might be preferred for better speedup and throughput.

Examining the performance for different parallelization alternatives, as well as their relation to configurable parameters from the algorithm, might be of interest. The prototypical implementation produced in our research could be employed for such a study.

Number of threads

There's a significant interdependence between the number of threads which can be used and the parallelization selected.

In concordance with previous research, favorable scalability, speedup and throughput can be expected by increasing the number of threads, up until the number of available hardware contexts is reached, or when cache-thrashing starts taking place.

The potential benefits from increasing the number of threads are also weakened by overheads of thread startup/management, poor scheduling and the effect of concurrent system workload.

Thread-placement strategies

So as to reduce cache-thrasing, a good implementation should aim to reduce the memory use of each thread. Local variables should be kept to a minimum.

Thread-placement strategies could also be examined as a tool for reducing cache thrashing, and thus improving speedup. This would be an important achievement, for high speedups in the partitioning stage might suggest potential speedups in the probe stage.

The motivating idea behind thread-placement strategies would be to place together (so they share a cache) threads accessing contiguous memory blocks. When this is achieved, better memory use can be expected. For each parallelization choice, certain thread-placement strategy might be more beneficial than others.

Processor affinity setting or CPU pinning (i.e. the binding of a thread or process to a specific CPU or set of CPUs), a service supported by several operating systems, might be employed to support this implementation alternative.

Handling of possible data dependence violations

From the parallelization options discussed, only threads belonging to a range-partitioning parallelization use shared data elements in such a way that might violate data dependence (i.e. the output array).

In a previous section we described the results of Cieslewicz and Ross [13]. In their work they consider 4 possible solutions for realizing the parallel writes to the output array done in radix-clustering: a count-then-move strategy, independent copies, parallel buffers and synchronized or concurrent access. Following their results, we could anticipate to observe a similar throughput for all solutions, save for the concurrent case, which, in turn, might display stability for a growing number of partitions and better results for larger page sizes.

In addition to the characteristics discussed in this section, scheduling and strategies for improving the load-balancing could also be considered as design alternatives deserving of further study.

Through our review of implementation alternatives we intended to analyze their viability to be presented as configurable parameters, which could then be passed as input to the algorithm. With the exception of the minimization of local variables per thread, all of the discussed options might plausibly be rendered as configurable parameters. In the next section we present our specific ideas for doing this w.r.t. number of threads, parallelization alternatives and thread-placement strategies.

a. Adjustable parallel radix-clustering and affinity setting

The fanout of the radix-clustering can be processed in parallel, while doing an even work-distribution per pass. This can be achieved by potentially using all of the task-based parallelization alternatives (except for the several-threads-per-cluster parallelization) and limiting the parallelization to a maximum number of threads.

For realizing this, we propose a straightforward scheme that takes into account both the bits per pass B_i (as they define the number of clusters per pass H_i) and the maximum number of threads *T*. The scheme consists on considering the number of

threads available (starting from *T-1*) for each pass. If the number of available threads is enough, per-cluster parallelization is done. If not, per-block parallelization is done. When no alternatives are left, no-parallelization is done.
We call this scheme adjustable parallel radix-clustering.
A valuable result of this scheme is that the parallelization alternatives can be defined by T and B_i, in this way, this implementation option is presented as a configurable parameter, which could be used for performance tuning.

By considering the presented scheme, we can observe that threads are created at a given pass. A very simple thread-placement strategy would be the selection of placing all threads generated in a certain pass on the same CPU, or not. This naive strategy can be expected to be moderately beneficial, unless the thread managing library already does a better job at thread-placement.

Through storing an array of bits A_i, which we'll call affinity vector, with i=number of passes, we can flag to the algorithm at thread-creation time whether the thread should be created with affinity to the others in the pass.

These two simple proposals: the affinity setting and the adjustable parallel radix-clustering illustrate the feasibility of turning implementation alternatives into configurable parameters amiable for performance tuning.

With this we conclude our discussion on configurable parameters and implementation alternatives. In the upcoming sections we'll introduce our experiments, present our results and discuss them.

V. EVALUATION

In this section we present our prototypical implementation and setting for the experiments.

A. Prototype

To examine the performance of the algorithm w.r.t. discussed parameters and options, we implemented a prototypical version of the algorithm in C, using the pthreads-library for thread parallelization.

Our version was strictly based on the canonical implementation, although we extended it to accept skewed data.

The scheme for adjustable parallel-radix-clustering is a key feature that we implemented in our prototype.
Support for the proposed affinity setting was also included.
Range-partitioning parallelization was not implemented for any pass. Because of this, obtained results will differ from the ones by Kim et al. [7] and Teubner et al. [6]. By avoiding range-partitioning our prototype is inadequate for studying choices in handling possible data dependence violations.
All of the previously discussed parameters, in addition to the number of threads T, and the affinity vector A_i, constitute the input to our prototype.

B. Experimental settings

With our prototype we set out to study the relations between discussed parameters and resulting performance. Our experiments were run by using 2 Intel Xeon E5-2609 v2 CPUs at 2.5GHz clock speed, with 256 KB L1 cache, 1 MB L2 cache, 10 MB L3 cache with 4-way associativity and a 64-bit instruction set. Data-TLBs at the different levels consisted of 2M/4M pages, 32 entries; 4K pages, 64 entries and 4K, 512 entries. Each CPU had 4 cores and a single hardware-context per core.

Wall-clock time was measured by using system libraries from the C language (sys/time.h). Memory use was studied by using Valgrind's tool Cachegrind.

Following common practice in previous research, we used 32 bit keys for our experiments. The values for these keys were generated with random skewness by using the QuEval framework [18]. Datasets consisting of 50K, 100K and 500K tuples were selected, representing a size of 200KB, 400KB and 2MB respectively.

The results of our experiments could be limited by our choice of using only a small number of CPU performance counters. Most notably we did not study TLB misses. Adding to this, the datasets used might not represent realistic workloads as some standard benchmarks might do. Measures from our prototype could also be limited by the fact that the prototype has not been yet adapted to an existing DBMS, as a result it could fail to reflect more accurate implementation characteristics for the algorithm. Lastly, the choice of an a platform with more parallelism and the pursuit of more specialized optimizations could better reflect the performance enhancing opportunities of the algorithm.

Through our tests we aimed to assess the potential of the adjustable parallelism scheme for improving speedup, scaleup and throughput. From all standard algorithm parameters, we focused on studying the effect on performance of changing the distribution of bits-per-pass. The later seemed an interesting parameter to consider, because it has an important impact in the adjustable parallelism scheme. The potential gains by affinity setting could not be satisfactorily evaluated, and they are not part of our results.

Execution times for the algorithm were determined by averaging over a series of repeats. For the 50K, 100K and 500K datasets, 1000, 500 and 100 repeats of the algorithm were used.

VI. DISCUSSION

Here we present the results of our experiments using 5 passes and considering a total of 11 bits per clustering. With this configuration we studied 4 different settings of bits-per-pass B_i*:* 11117, 31511, 53111 and 71111. Each digit of these numbers represents the amount of bits considered for clustering during a given pass. The leftmost value corresponds to the first pass and each next digit to the right relates to a successive pass, up until the last pass when the number of overall bits considered adds up to 11.

Conforming to the scheme of adjustable parallelism, the values of *Bi* next to the total number of threads *T*, determine the parallelism to be executed. For example, 11117 with 32 threads means that 2 threads will be used for processing via per-cluster parallelism the second pass. 4 threads would also adapt to that form of parallelism for the third pass. 8 threads would serve the forth pass, and finally 16 threads would perform the last pass. In this example only 16 of the maximum 32 threads available could be used.

For the different *Bi* choices we found out that execution time improves only slightly in the single-threaded case for a higher number of partitions at an early stage. However these differences are not significant.

For all cases studied we also found out that execution times tend to grow when the number of threads starts to exceed the number of available hardware contexts in our platform (Figures 4 to 6).

Sublinear speedup was observed for most cases (Figure 7 and 8), perhaps due to the time required for running the unoptimized first pass, but also to thread startup/management overheads.

As expected from the interrelation between the number of threads *T*, B_i and the achievable parallelization, we observe differences in the performances for each choice of these values. The best speedups fell in the 2.2x range. They were achieved by the 71111 option, running on 4 threads (the number of hardware contexts of one of the processors used). Interestingly, no alternative of B_i can achieve better speedups for 8 threads, which should be able to use the maximum number of contexts. Since for this case no significant change on the percentage of data misses was observed (Figure 11), we speculate that this behavior could be caused by operating system configurations or thread management overhead. However these assumptions were not studied.

Scale up, defined as the ability of the program to process N-times more input on N-times more threads, was also studied (Figure 10). An ideal scale up chart would display similar execution times when increasing the problem size and the number of threads by a same proportion. This was not the case for the current algorithm, a result that could be also attributed to the unoptimized first pass and other unoptimized variables. No significant variations in instruction cache misses were observed.

An important finding is that cache-thrashing was not observed (Figure 11). This occurrence relates to the cache-conscious properties of the algorithm itself and to the beneficial platform characteristics and data sizes. The 50k and 100k datasets both fit into the L2 caches, and after the first pass the generated partitions should fit into the L1 caches. The 500k dataset fits into the L3 cache and after the first pass its partitions could be in the L2 caches. After 8 partitions more (no matter the number of passes), the partitions from the 500k dataset will also fit in the L1 caches. Coherent with this logic, lower-level misses were almost 0% for all cases, and this was continued when increasing the number of threads.

These observations regarding memory usage might suggest that the cache-conscious pattern of the serial-algorithm could be preserved and even improved by task-based parallelization of the original program, when partitions can be distributed in more L1 and L2 caches.

Another meaningful finding is that given a number of threads, throughput is maintained, independently from the number of tuples considered (Figure 9). This, in addition to the findings regarding memory usage might suggest that for relations within the size range evaluated, and for similar platforms, the performance of the presented adjustable parallel radix-clustering would be less defined by memory usage and more dependent on optimization of the computation itself. Additional experiments should be performed to confirm these early results.

To conclude this section, it should be noted that our results do indeed confirm the potential of this parallelization for achieving speedups and increased throughput, albeit at suboptimal levels in our studies. However the ability of the algorithm for scaling up cannot be ascertained from our results and should be further studied.

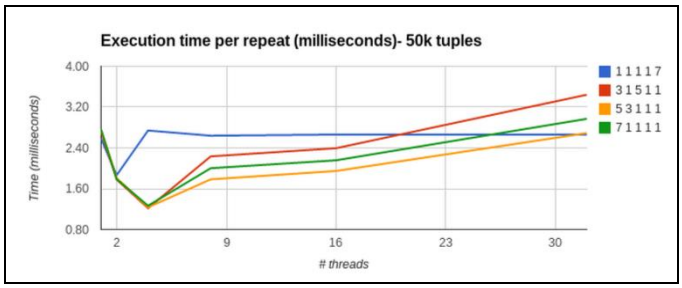

Figure 4: Execution time per repeat, 50 000 tuples

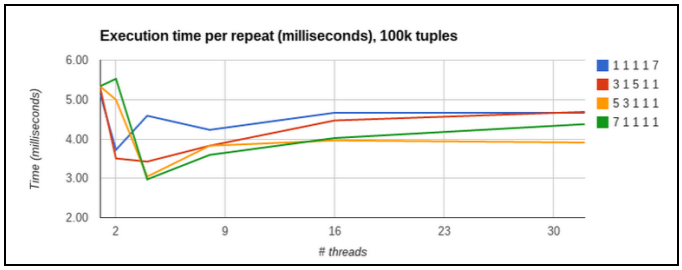

Figure 5: Execution time per repeat, 100 000 tuples

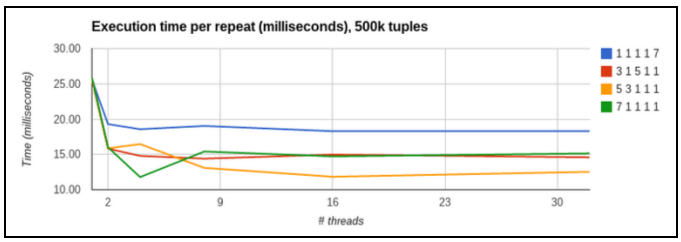

Figure 6: Execution time per repeat, 500 000 tuples

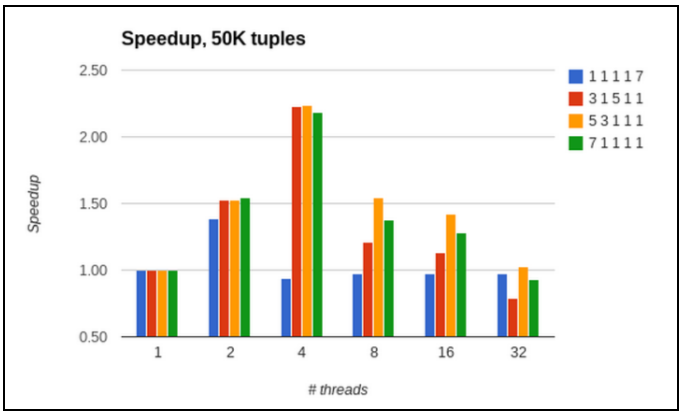

Figure 7: Speedup, 50 000 tuples

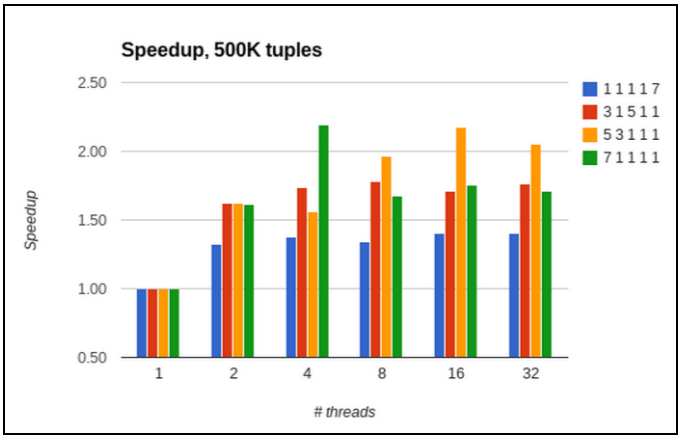

Figure 8: Speedup, 500 000 tuples

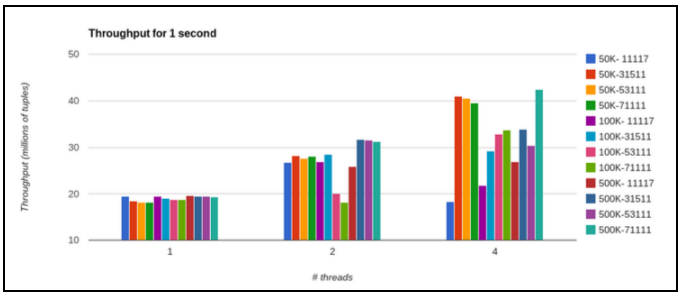

Figure 9: Throughput for 1 second

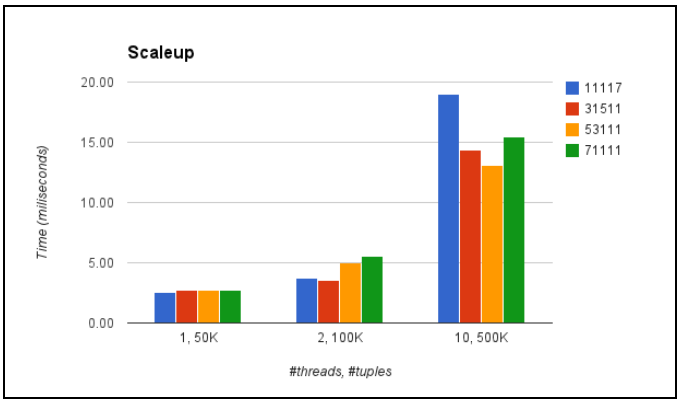

Figure 10: Scale up

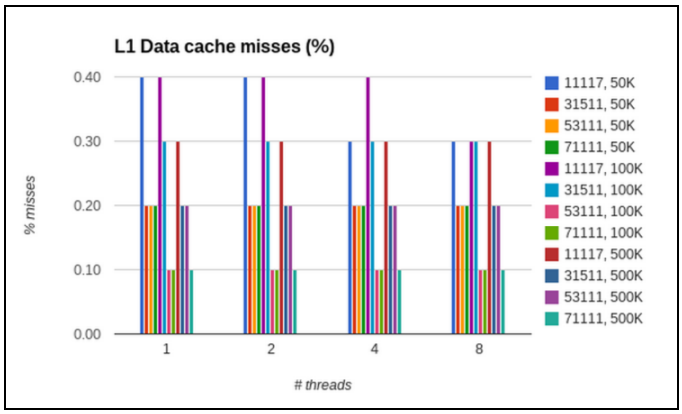

Figure 11: L1 Data cache misses (%)

VII. CONCLUSION AND FUTURE WORK

In this study we've aimed to identify variables and implementation options affecting the performance of parallel versions of radix-clustering. Among the existing configurable parameters, we've identified: number of clusters, bits per pass, total number of bits, number of passes and number of threads. With regard to the implementation alternatives, we considered: the choice of parallelism to exploit, thread-placement strategies and methods for handling data dependence violations. We tried to summarize results from diverse research describing the impact of these parameters and alternatives on the performance of the resulting algorithm.

An overreaching proposal of the current project was to present implementation options as configurable parameters, so as to facilitate their use for performance tuning. By proposing a scheme of adjustable parallelism we proved that this was possible for configuring the parallelization itself. We also suggested affinity setting as an alternative for tuning by alternating thread-placement strategies. Further studies should be made for turning other performance-impacting implementation choices into tunable variables.

By means of experimenting with an early prototype, we've confirmed the potential for improved throughput and speedup by parallelizing the original clustering. No evidence was found for such a potential regarding its scalability.

The tuning of the parallel version by using standard parameters of the algorithm was not thoroughly explored in our tests. This subject demands more research. The affinity setting options made available by our prototype should be tested in future work as well.

Good memory usage was found throughout our experiments, and within the limited scope of our tests, cache thrashing was not observed.

Given that speedup was found to be suboptimal, and this was not due to cache thrashing, we suggest that future research should target the challenge of optimizing the parallel computation. A great deal of options is available for this. Range-partitioning should be specially studied.

The parallelization of the clustering could also be examined in architectures offering different and higher kinds of parallelism, such as GPUs. An innovative approach might be

to devote for range-partitioning as much threads as generated clusters for the penultimate pass. The first pass would occur as usual with range-partitioning. Up until the penultimate pass recursive calls would be handled with several-threads-per-cluster parallelization, dividing the existing threads to all clusters. The potential benefits from using range-partitioning in this way are not entirely clear. This approach might be useful in GPUs, because all threads would have a similar granularity. In addition, for this approach it might also be worthwhile to test independent copies and parallel buffers as potential performance-contributing choices for the handling of possible data dependence violations.

Other areas demanding future work are the grouping of the parallel clustering to the successive phases of radix-join, and the deployment of this version of radix-join into a set of alternative physical operators fit for use by query-optimizers.

ACKNOWLEDGMENT

This project stems from the HyPE query processor research initiative [8], aiming to contribute to its ongoing evaluation of physical operators.

The authors acknowledge the helpful guidance of their tutor David Broneske and the referees.

REFERENCES

[1] S. Manegold, P. A. Boncz, and M. L. Kersten, “Optimizing main-memory join on modern hardware”, IEEE Trans. Knowl. Data Eng., 2002

[2] P. A. Boncz, S. Manegold, and M. L. Kersten, “Database architecture optimized for the new bottleneck: Memory access,” In Proceedings of VLDB Conference, 1999

[3] A.Shatdal, C. Kant, J. Naughton, “Cache Conscious Algorithms for Relational Query Processing”, In Proceedings of the International Conference on Very Large Databases, 1994

[4] S.Blanas, Y.Li, J. Patel, “Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs”, In SIGMOD Conference, 2011

[5] D.Yadan, J.Ning, X.Wei,C.Luo, C. Hongsheng, “Hash Join Optimization Based on Shared Cache Chip Multi-processor”, Procedings from *Database Systems for Advanced Applications,* 2009

[6] C.Balkesen, J.Teubner, G.Alonso, “Main-Memory Hash JoinS on Multi-core CPUs: Tuning to the Underlying Hardware”, IEEE 29th International Conference on Data Engineering (ICDE), 2013

[7] C.Kim, E.Sedlar, G.Chhugani, “Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs”, In SIGMOD Conference, 2009

[8] S.Bress, I.Geist, E.Schallehn, M.Mory and G.Saake, “A framework for cost based optimization of hybrid CPU/GPU query plans in database systems”, In Procedings of Control & Cybernetics; Vol. 41 Issue 4, p715, 2012

[9] D. DeWitt and R. Gerber, ”Multiprocessor hash-based join algorithms”. In*Proceedings of the 11th international conference on Very Large Data Bases - Volume 11* (VLDB '85), Alain Pirotte and Yannis Vassiliou (Eds.), Vol. 11. VLDB Endowment 151-164., 1985

[10] S.Manegold, P.Bonz, N.Nes, M.Kersten, “Cache-Conscious Radix-Decluster Projections”, In Procedings of The 30th International Conference on Very Large Databases, 2004

[11] L. Gotlieb, ”Computing joins of relations”. In Proceedings of the 1975 ACM SIGMOD international conference on Management of data (SIGMOD '75). ACM, New York, NY, USA, 55-63, 1975

[12] C.Nyberg, T.Barclay and J. Gray, “AplhaSort: A RISC Machine Sort”, In Procedings Of the 1994 ACM SIGMOD, 1994

[13] J. Cieslewicz and K. Ross, “Data partitioning on chip multiprocessors”. In Proceedings of the 4th international workshop on Data management on new hardware (DaMoN '08), ACM, New York, NY, USA, 25-34, 2008

[14] C. Shimin, A. Ailamaki, P. Gibbons, and T. Mowry, “Improving hash join performance through prefetching”. *ACM Trans. Database Syst.* 32, 3, Article 17, 2007.

[15] P. Garcia and H. Korth., “Database hash-join algorithms on multithreaded computer architectures”. In *Proceedings of the 3rd conference on Computing frontiers* (CF '06). ACM, New York, NY, USA, 2006

[16] M. Albutiu, A. Kemper and T. Neumann, “Massively parallel sort-merge joins in main memory multi-core database systems”. In Procedings of VLDB Endow. 5, 10, 2012

[17] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper and T. Neumann, “Locality-sensitive operators for parallel main-memory database clusters”. In Procedings of ICDE, 2014.

[18] M. Schäler, A. Grebhahn, R. Schröter, S. Schulze, V. Köppen and G. Saake, “QuEval: beyond high-dimensional indexing à la carte”. In Procedings of VLDB Endow. 6, 14, 2013