Improving the Performance of List Intersection

Dimitris Tsirogiannis
University of Toronto
dimitris@cs.toronto.edu

Sudipto Guha
University of Pennsylvania
sudipto@cis.upenn.edu

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

ABSTRACT

List intersection is a central operation, utilized excessively for query processing on text and databases. We present list intersection algorithms for an arbitrary number of sorted and unsorted lists tailored to the characteristics of modern hardware architectures. Two new list intersection algorithms are presented for sorted lists. The first algorithm, termed *Dynamic Probes*, dynamically decides the probing order on the lists exploiting information from previous probes at runtime. This information is utilized as a cache-resident micro-index. The second algorithm, termed *Quantile-based*, deduces in advance a good probing order, thus avoiding the overhead of adaptivity and is based on detecting lists with non-uniform distribution of document identifiers. For unsorted lists, we present a novel hash-based algorithm that avoids the overhead of sorting.

A detailed experimental evaluation is presented based on real and synthetic data using existing chip multiprocessor architectures with eight cores, validating the efficiency and efficacy of the proposed algorithms.

1. INTRODUCTION

Set intersection is a central operation in query processing and text analytics. All modern query processors inherently support set operations. Additionally, set intersection is employed during the evaluation of conjunctive selection conditions. Consider for example the evaluation of the following conjunctive selection condition: $a_1 =$ 'X' $\wedge\, a_2 =$ 'Y', where a_1, a_2 are two relation attributes. Assume that for each attribute a_i there is an index. Utilizing the index on each attribute, one can retrieve a set of pointers to records satisfying an individual condition. The conjunctive selection condition is evaluated by taking the intersection of all sets of record pointers and then fetching the qualifying records from memory. Sets of record pointers may be sorted or unsorted.

Text data are generated at unprecedented rates. This is corroborated by user generated content in the form of blogs and social networks. The huge amount of text data as well as the increasing need to exploit textual information in order to solve business problems necessitate the use of efficient query processing techniques.

The fundamental data structure for indexing and query processing on text is the inverted index [4]. Query processing consists of retrieving the inverted indices corresponding to query keywords and intersecting them to identify relevant documents. Each inverted index for a keyword q consists of a document identifier and commonly a score associated with each document identifier. The score represents a measure of the importance of q at each document identifier (commonly computed via several scoring functions for document scoring purposes [4]). Depending on the platform and the application, query processing may involve the intersection of a different number of inverted indices which may or may not be ordered on document identifier.

Text analytics queries can be answered efficiently using inverted index (list) intersection. Consider for example the case where company *X* wants to know what male bloggers under the age of 25 and located in a specific geographical area think of a new product *Y*. To answer questions like this, we have to retrieve all blog posts that satisfy certain criteria. Assuming that both demographics (age, geographic location) and text keywords are indexed using inverted indices, text analytics queries like this one can be answered efficiently computing the intersection of a number of inverted indices, each corresponding to a particular criterion, i.e. age = '25'.

Given the importance of this basic operation, list intersection has attracted significant research attention in the past [3, 5, 7, 8, 13, 14, 27]. Prior research on list intersection algorithms has ignored the characteristics of modern hardware architectures: a) large on-chip caches, and b) on-chip parallelism in the context of chip-multiprocessors. It is imperative to incorporate modern hardware trends in algorithm design to improve the performance of this increasingly important operation.

1.1 Hardware Trends

The use of 64-bit computer architectures increased the amount of memory that a system can utilize and today it is common to have servers with many GBs of RAM performing tasks in memory-resident data. However, increasing the size of memory also increased its random access latency to hundreds of CPU cycles introducing the "memory-wall" problem [1]. Significant amount of research has been conducted toward "cache-aware" query processing techniques that effectively utilize multi-level cache hierarchies to reduce the overhead of random memory accesses [10, 24, 28]. On the other hand, traditional techniques for list intersection have not been adapted to modern processor architectures with large on board caches mostly due to the random memory access patterns that they exhibit.

On-chip parallelism was recently introduced in the context of chip multiprocessors (CMPs). Chip-multiprocessors allow concurrent execution of multiple threads of control through the use of multiple simple processing units (*cores*). Most vendors have adopted this design paradigm [2, 21, 23]. Compared to multi-processor ar-

VLDB '09, August 24-28, 2009, Lyon, France

chitectures, CMPs offer different computation and communication tradeoffs. In a CMP, inter-processor communication is relatively inexpensive as it is performed through some shared level of the cache hierarchy, typically the L2 cache. On the other hand, more shared resources, such as cache and memory bandwidth, must be carefully managed to avoid the competition among cores which may result in severe performance degradation.

Several research efforts have studied how to unleash the computational power of CMPs in order to improve the performance of various data management tasks [11, 16]. On the contrary, prior research work on list intersection algorithms assumes a uniprocessor architecture, ignoring the potential for performance improvement that CMPs offer.

1.2 List Intersection on Modern Hardware

In this paper, we study the design of in-memory list intersection algorithms, tailored to the characteristics of modern hardware architectures. Our goals are: a) to reduce the overhead of random memory accesses by exploiting the large on-chip cache hierarchy, and b) to take advantage of the on-chip parallelism offered in CMPs.

For the case of sorted lists, the majority of algorithms proposed follow a similar pattern in the way they compute intersection. The elements of one list are used as *eliminators* and *probes* are conducted in the other lists using a search method (e.g. binary or interpolation search) to locate these elements. If an eliminator belongs to the intersection, the probes will locate it in all lists. If this is not the case, the eliminator is simply ignored. The algorithms proposed in the literature vary in how they select the eliminators and in the order in which they probe the lists (*probing order*).

All search methods utilized for probing lists exhibit a random memory access pattern, thus lacking temporal and spatial locality. Consequently, the multi-level cache hierarchy is underutilized and every probe incurs a number of expensive memory accesses, wasting hundreds of CPU cycles.

In this paper, we propose a list intersection algorithm for sorted lists, termed *Dynamic Probes*, that utilizes previous probes to construct a cache-resident *micro-index* of the lists. The micro-index is utilized to reduce the search range of each probe and consequently, the number of random memory accesses performed. The micro-index is continuously updated during the computation of the intersection, allowing the algorithm to adapt to the characteristics of each list by dynamically changing its probing order.

An important consideration when designing parallel algorithms is load balancing. Primarily, load has to be distributed evenly among cores. If this is not the case, load disparity underutilizes the computational power available, thus reducing the speedup achieved. Furthermore, the overhead of load balancing should be kept to a minimum, otherwise it may offset the benefits of parallelization. It is important to notice that executing a list intersection algorithm in parallel may not always be the best solution due to the load balancing overhead introduced. In cases where computing the intersection is not time consuming and throughput is the primary performance objective, a serial algorithm may be preferred.

Exact algorithms for partitioning a number of lists ensure perfect load balancing. However, when a large number of lists is intersected, the overhead of such algorithms is high [22]. Approximate algorithms with explicit error guarantees offer attractive alternatives as they trade accuracy for speed. Load is not perfectly balanced but the overhead is lower than the overhead of exact algorithms. Even when approximate algorithms are utilized, the overhead of load balancing is non-negligible as these algorithms are not optimized for cache performance. Furthermore, for the case of unsorted lists, they deploy an expensive sorting phase.

We propose a CMP adaptation of an approximate quantile identification algorithm demonstrating how it can be utilized to efficiently partition an arbitrary number of sorted (*Partition Sorted* algorithm) and unsorted lists (*Partition Unsorted* algorithm) into any number of independent partitions. The quantile identification algorithm used has explicit error guarantees, allowing us to control the load disparity across cores.

Applying a quantile identification algorithm for load balancing purposes provides a unique opportunity to inspect list elements and identify lists with highly non-uniform distribution of elements. As Demaine et al. pointed out, this non-uniformity may result in severe performance degradation of list intersection algorithms [13]; they proposed adaptive intersection algorithms to address this issue. However, when a large number of lists is intersected, their approach introduces notable overhead because it tends to be "over-adaptive" [14, 27]. In this paper, we propose a list intersection algorithm for sorted lists, termed *Quantile-based*, that takes advantage of quantiles, computed by the load balancing algorithm, to detect lists with highly non-uniform distribution of elements and determine a good probing order at low cost, thus eliminating the overhead of adaptivity.

Finally, for the case of unsorted lists, all known approaches deploy a sorting algorithm to sort the lists. Then, a list intersection algorithm for sorted lists is applied. Instead, we propose *Hash-based*, a list intersection algorithm that is tailored to CMPs. It utilizes the quantile identification algorithm to avoid sorting the lists and computes their intersection in a cache-conscious manner using hashing.

1.3 Contributions and Paper Outline

To the best of our knowledge, this is the first study on how to improve the performance of list intersection for sorted and unsorted lists on modern hardware architectures. The contributions of this paper are the following:

- We demonstrate how we can exploit the characteristics of modern hardware architectures to improve the performance of list intersection for sorted and unsorted lists.
- We present *Dynamic Probes*, a novel list intersection algorithm that takes advantage of multi-level cache hierarchies to reduce the overhead of random memory accesses.
- We study the problem of load balancing during the parallelization of list intersection algorithms and we propose an effective and efficient load balancing algorithm based on quantiles.
- We present *Quantile-based*, a list intersection algorithm that exploits the "free" inspection of lists, when a load balancing algorithm is employed, to reduce the cost of intersection.
- We present *Hash-based*, a list intersection algorithm for unsorted lists tailored to CMPs that avoids the overhead of sorting.
- We present a detailed experimental evaluation using synthetic and real data on a chip multiprocessor with eight cores. We demonstrate that all algorithms proposed herein scale almost linearly as the number of cores increases. For sorted lists, the proposed techniques exhibit the best performance compared to existing list intersection algorithms. When the lists are unsorted, the *Hash-based* algorithm performs better than the combination of sorting and any list intersection algorithm for sorted lists.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 describes the approximate quantile identification algorithm and presents cache-conscious list partitioning algorithms for sorted and unsorted lists. Section 4 presents intersection algorithms for sorted lists and the hash-based intersection algorithm for unsorted lists. We present the experimental evaluation of the proposed algorithms in Section 5 and conclude in Section 6.

2. RELATED WORK

The problem of intersecting two sorted lists has been studied in the past by Hwang and Lin [20, 19]. Demaine et al. proposed the *Adaptive* algorithm for computing the intersection of a collection of sorted sets [13]. Their algorithm computes the intersection by repeatedly cycling through the sets in a round-robin fashion. On a follow-up study they compared the *Adaptive* algorithm with existing algorithms using real data and demonstrated that in practice their algorithm does not always exhibit the best performance due to the overhead of adaptivity (repeatedly cycling through the sets) [14]. Based on these findings, a number of improvements were proposed, the most significant of which are: a) galloping search [9], and b) a limited form of adaptivity, compared to the initial version. The resulting algorithm, termed *Small Adaptive*, intersects the two smallest sets and it starts cycling through the remaining sets only when a match is found.

Barbay et al. repeated the study of [14] using a larger dataset [8]. In their experiments they included a list intersection algorithm proposed in [7]. Their study verified that *Small Adaptive* has the best performance in practice and that interpolation search improves the performance of all intersection algorithms.

An adaptive row id list intersection strategy for query processing has been proposed by Raman et al. [27]. In order to avoid the risk of using wrong cardinality estimations during the computation of an AND-tree, they use an adaptive n-ary intersection algorithm based on the *Adaptive* algorithm [13]. Instead of using a round-robin probing policy as suggested in [13], they propose a probabilistic probing policy to determine which list to examine next based on historical data from previous probes. In their experimental evaluation they compare their approach against the *Adaptive* algorithm and against the traditional pipelined approach that utilizes the best AND-tree. As we will show in Section 5, the performance of *Small Adaptive* [14] in many cases surpasses or is equivalent to the performance of their algorithm.

Iyer et al. proposed an exact percentile identification algorithm for an arbitrary number of sorted lists [22]. The overhead of the algorithm increases by a factor of $m \log m$, where m is the number of sorted lists. Hence, it is not applicable to a large number of lists. Furthermore, in order to apply the algorithm in unsorted lists, a sorting algorithm must be utilized at first, introducing significant overhead.

Several algorithms for computing approximate quantiles of large datasets have been proposed [25, 18, 12]. Manku et al. presented single pass algorithms for computing approximate quantiles of large datasets with explicit approximation guarantees [25]. Greenwald and Khanna proposed a distributed single-pass algorithm to compute an ϵ-approximate quantile summary over sensor data [18]. Cormode et al. proposed deterministic algorithms for biased quantiles, ranked queries and targeted quantile queries that have guaranteed space bounds [12].

3. LOAD BALANCING ON A CMP

A major issue when designing algorithms for CMP architectures is load balancing. Since load disparity among cores and the overhead of the load balancing technique limit the achievable speedup through parallelism, we seek low overhead load balancing techniques with explicit error guarantees. Techniques that achieve perfect load balancing usually introduce considerable overhead and in many cases the benefit derived from the elimination of the load disparity does not compensate for the overhead introduced [15]. We verify that effect in Section 5.

A quantile identification algorithm computes any order statistic over a set of elements[1] S. Such an algorithm is used as a building block in our framework to divide (sorted and unsorted) lists into a number of *independent* partitions of approximately equal size. Each partition is assigned to a core and in this way the computation of list intersection is parallelized in a load balanced way. For the case of sorted lists, list elements are accessed once while computing the intersection and the only requirement is load to be evenly partitioned among cores. Hence, the number of partitions is set to be equal to the number of cores utilized. For the case of unsorted lists, we require each partition to be cache-resident. Consequently, for a system utilizing C cores, we choose the number of partitions such that each partition fits in $1/C$-th part of L2 cache.

Let us first discuss how a quantile identification algorithm can be utilized to partition $k \geq 2$ sorted lists into P *independent* partitions of approximately equal size on a uniprocessor architecture (Section 3.1). Next, we will demonstrate how these quantiles can be computed efficiently on CMPs (Section 3.2).

3.1 List Partitioning on a Single Processor

The *independence* property states that for any pair of partitions $P_i, P_j (i < j, i, j = 1, \ldots, P)$, all the elements of P_i must be strictly smaller or larger than all the elements of P_j. Consider for example two sorted lists A_1 = {1, 2, 3, 5, 9, 10, 12, 15, 18, 20, 40} and A_2 = {4, 8, 11, 13, 14, 16, 17, 39, 41, 42, 50}. A possible partitioning of these two lists in two partitions that satisfies the independence property could be: $P_1 = \{\{1, 2, 3, 5\}, \{4\}\}$ and $P_2 = \{\{9, 10, 12, 15, 18, 20, 40\}, \{8, 11, 13, 14, 16, 17, 39, 41, 42, 50\}\}$. In this partitioning, P_1 contains four elements of A_1 and one element of A_2; P_2 contains seven elements of A_1 and ten elements of A_2. Notice, that all the elements of P_1 are strictly smaller than the elements of P_2.

In order to identify the elements of each partition P_i, it is sufficient to determine its boundary values $lo(P_i), hi(P_i)$. The following invariant is true for boundary values: if element id belongs to P_i then $lo(P_i) \leq id \leq hi(P_i)$. In the example described, the boundary values are: $lo(P_1) = 1, hi(P_1) = 5$, $lo(P_2) = 8, hi(P_2) = 50$. In general, a partition is a "set" of sublists - one from each of the original lists - where each of the sublists shares the same boundary values. The size of a partition is the sum of sizes of the sublists.

Note that although this is one possible partitioning of the two lists, it is not necessarily the best as P_1, P_2 are not of equal size. In order to divide an arbitrary number of sorted lists into P partitions of equal size, it is sufficient to compute $P-1$ order statistics of the elements in the union of the lists; specifically, the elements with rank $\frac{i}{P}, i = 1, \ldots, P-1$. The $P-1$ order statistics along with the extreme values (min, max) of the lists determine the boundary values of the partitions. In our example, we need to compute the median in the union of A_1 and A_2 (value 13). Hence, the boundary values of the partitions in the optimal partitioning of A_1, A_2 are: $lo(P_1) = 1, hi(P_1) = 13, lo(P_2) = 14, hi(P_2) = 50$, i.e. $P_1 = \{\{1, 2, 3, 5, 9, 10, 12\}, \{4, 8, 11, 13\}\}$, $P_2 = \{\{15, 18, 20, 40\},$

[1]Element and document identifier (id) are used interchangeably in the text.

$\{14, 16, 17, 39, 41, 42, 50\}\}$. The resulting partitions, P_1, P_2 have the same size (11 elements each).

Existing exact quantile identification algorithms can be used to compute order statistics and consequently to partition the lists in a load balanced way. However, these algorithms introduce notable overhead. Several approximate quantile algorithms with explicit error guarantees have been proposed in the literature [25, 18, 12]; any of these algorithms can be utilized in our setting. In this study we choose to elaborate utilizing the algorithm proposed by Greenwald and Khanna [18] since it has better space requirements compared to the algorithm in [25]. It is easy to see that if the size of the dataset N is known, the space requirements of the algorithm in [18] is $O(\frac{1}{\epsilon}\log\epsilon N)$ compared to the algorithm in [25] which requires $O(\frac{1}{\epsilon}\log^2\epsilon N)$ memory, where ϵ is the error with which any order statistic is computed. Furthermore, the algorithm in [18] is utilized because we want the simplest algorithm for deterministic unbiased quantiles; the algorithm in [12] is for biased quantiles.

An ϵ-approximate quantile summary Q for a set of elements S is an ordered subset of S that can be utilized to answer any order statistic query over S with error ϵ. The resulting summary Q contains $(1/2\epsilon + 1)$ elements. If we query Q for an element with rank r with error ϵ, then the output element is guaranteed to have rank within $r \pm \epsilon|S|$. Next, we describe the computation of an ϵ-approximate quantile summary for a set of elements residing in two sorted lists in a uniprocessor architecture using the algorithm in [18]. For brevity, we refer to an ϵ-approximate quantile summary as ϵ-*summary*.

We will describe the basic operations of the algorithm using a simple example. Assume that we wish to compute an ϵ-summary Q for the elements residing in two sorted lists A_1, A_2 (same as in the previous example) with error $\epsilon = 0.1$. Initially, the algorithm computes two ϵ-summaries Q_1, Q_2, for the elements in A_1 and A_2, respectively. An ϵ-summary Q_i for the elements of A_i is computed by choosing at most $\frac{1}{2\epsilon} + 1$ elements. These are the elements with ranks $1, 2\epsilon|A_i|, 4\epsilon|A_i|, \ldots, |A_i|$, where $|A_i|$ is the number of elements in A_i. The ϵ-summaries Q_1, Q_2 contain six elements each, Q_1 = {1, 3, 9, 12, 18, 40} and Q_2 = {4, 11, 14, 17, 41, 50}. In order to compute the final ϵ-summary Q, we apply the *combine*$(Q_1, \ldots, Q_m)$ operation [18]. The *combine* operation produces a new summary Q by merging the smaller summaries $Q_1, \ldots, Q_m$ provided as input. The error of Q is the maximum error of the input summaries in *combine*. The size of Q is equal to the union size of summaries provided as input in *combine*. In our example, we apply the *combine* operation using as input the ϵ-summaries Q_1, Q_2 obtaining Q = {1, 3, 4, 9, 11, 12, 14, 17, 18, 41, 40, 50} which is an ϵ-summary that can be used to answer any order statistic over all the elements in A_1 and A_2 with error 10%.

3.2 List Partitioning on CMPs

Given the description of the approximate quantile algorithm for uniprocessors, we now propose an adaptation tailored to CMPs, resulting in *Partition Sorted* algorithm for sorted lists (presented in Section 3.2.1) and *Partition Unsorted* algorithm for unsorted lists (presented in Section 3.2.2). The adaptation aims to: a) enhance the algorithm so that it exploits all the available processing elements, and b) improve its cache locality, thus reducing the number of expensive memory accesses performed.

We consider a CMP architecture consisting of C cores and an on-chip cache hierarchy with a private L1 cache per core and a shared L2 cache which can store $|L2|$ elements. Our algorithms are general enough to function on any type (organization) of the on-chip cache hierarchy. We refer to L2 as the lowest level of the on-chip cache hierarchy which is shared among cores.

3.2.1 Partition Sorted Algorithm

To parallelize the computation of an ϵ-summary from k sorted lists, we assign the lists to cores (approx. $\frac{k}{C}$ lists per core). Let k_i be the number of lists assigned to core i. Note, that there is no physical transfer of lists because they reside in memory which is shared among cores. Core i computes k_i ϵ-summaries $Q_1, \ldots, Q_{k_i}$, one from each assigned list. Then, it merges the k_i ϵ-summaries using the *combine* operation to produce a single ϵ-summary Q^i. A single core is responsible for merging the resulting ϵ-summaries $Q^1, \ldots, Q^C$ using the *combine* operation and produces the final ϵ-summary Q. The ϵ-summaries are small enough to reside in cache and since cache is shared among cores, the intermediate ϵ-summaries $Q^1, \ldots, Q^C$ as well as the final ϵ-summary Q are produced without the need for expensive off-chip memory accesses.

If $k < C$, we "greedily" partition the lists in order of decreasing length. The partitioning requires C steps and at every step $\min\{$# of remaining unassigned elements of current list, $\frac{N}{C}\}$ contiguous elements from the currently examined list are assigned to a core; N is the total number of elements in the lists. Each core produces an ϵ-summary and the ϵ-summaries are merged by a single core to produce the final ϵ-summary Q.

3.2.2 Partition Unsorted Algorithm

In contrast to the sorted case, when the lists are unsorted, it is not possible to create an ϵ-summary by simply choosing a set of elements with specific rank. The naive solution is to sort the lists and then compute an ϵ-summary utilizing the *Partition Sorted* algorithm (Section 3.2.1). We demonstrate that it is possible to avoid a possibly expensive sorting phase.

In Algorithm 1, we present pseudo-code for the *Partition Unsorted* algorithm. The main idea is to partially sort each list into a number of sorted *runs* and create an ϵ-summary from each sorted run (lines 1-7). Once the ϵ-summaries have been produced, they are merged into the final ϵ-summary Q (lines 9-21).

Each run contains $\frac{|L2|}{C}$ elements from a list and it is sorted using Quicksort which is an in-place sorting algorithm. The size of each run is chosen so that C runs can fit simultaneously in L2. Therefore, the generation of sorted runs does not cause any L2 capacity cache misses. Only compulsory cache misses are incurred when the elements are fetched for the first time into the cache. An ϵ-summary is produced from a sorted run immediately after its creation. The overhead of producing the ϵ-summary is negligible as the run already resides in cache.

Once the initial ϵ-summaries of all the runs have been produced, they are merged in parallel by the cores to produce the final ϵ-summary Q. Let M be the number of initial ϵ-summaries. The M ϵ-summaries are assigned to cores (approximately $\frac{M}{C}$ ϵ-summaries per core) and each core merges its assigned ϵ-summaries using the *combine* operation; an ϵ-summary Q^i is produced by core i. The C resulting ϵ-summaries $Q^1, \ldots, Q^C$ are merged by a single core to produce Q.

If all ϵ-summaries can fit in L2, the cores do not experience any L2 cache misses while the ϵ-summary Q is produced (lines 10-13). If this is not the case, we resort to a multi-phase merging (lines 15-21). In each phase the cores read a sufficient number of summaries fitting in L2. When in cache, the summaries are merged by the cores and a single summary is produced. Then, the size of the resulting summary is reduced using the *prune* operation [18].

The *prune* operation reduces the size of a quantile summary. It takes as input an ϵ-summary Q for a set of elements S and a parameter B and produces a new ϵ'-summary Q' of size at most $B + 1$ elements with error ϵ' by querying Q for the elements of rank $1, |S|/B, 2|S|/B, \ldots, |S|$. The error of Q' is $\epsilon' = \epsilon + \frac{1}{2B}$.

In the next phase, a sufficient number of pruned summaries fitting in L2 is read by the cores. The summaries are merged and a new summary is created which is again pruned. In the last phase all the summaries fit together in L2 and the final summary Q is computed (line 21).

Since the number of initial summaries is M, the *prune* operation will be applied at most $\log M$ times, each time increasing the error of the resulting quantile summary by $\frac{1}{2B}$. Hence, in order for the final quantile summary to have error ϵ, we must create the initial quantile summaries with error $\frac{\epsilon}{2}$ ($\frac{\epsilon}{2}$-summaries) and set the value $B = \frac{1}{\epsilon}\log M$. Consequently, if we apply the *prune* operation $\log M$ times, we will generate a quantile summary with error ϵ.

Algorithm 1 *Partition Unsorted*

Input: k unsorted lists $A_1, \ldots, A_k$, $|A_i|$ elements in list A_i, $|L2|$ elements in L2 cache, C cores, error ϵ
Output: an ϵ-approximate quantile summary Q

1: **for** $l = 1, \ldots, k$ **do**
2: **while** $\exists$ unread elements in A_l **do**
3: Read the next $\frac{|L2|}{C}$ elements from A_l.
4: Sort the elements in-place using Quicksort and generate a sorted run r.
5: Compute an ϵ-approximate summary for r containing at most $\frac{1}{2\epsilon}+1$ elements.
6: **end while**
7: **end for**
8: **Barrier** // Synchronization point - There are $M = \sum_{l=1}^{k} m_l$ quantile summaries, where $m_l = \lceil \frac{|A_l|\cdot C}{|L2|} \rceil$ is the number of runs of list A_l.
9: **if** $M \cdot (\frac{1}{2\epsilon}+1) < |L2|$ **then**
10: The quantile summaries are assigned to cores, i.e. $\frac{M}{C}$ quantile summaries are assigned to each core.
11: Each core merges its assigned quantile summaries using the *combine* operation.
12: **Barrier** // Synchronization point - C quantile summaries have been created.
13: The C quantile summaries are *combined* by a single core into the final quantile summary Q.
14: **else**
15: **while** summaries cannot fit in L2 cache **do**
16: Each core reads $\frac{M'}{C}$ summaries in L2 cache. // Let M' be the number of summaries that can fit in L2 cache.
17: Each core merges its assigned summaries using *combine* operation.
18: **Barrier** // Synchronization point - C quantile summaries have been created.
19: The C quantile summaries are *combined* by a single core into the quantile summary Q'. *prune* is applied to Q' to reduce its size to at most $B+1$ elements.
20: **end while**
21: Q is computed. // Lines 10 - 13
22: **end if**

4. LIST INTERSECTION ALGORITHMS

In this section, we present several algorithms to compute the intersection of sorted and unsorted lists. Section 4.1 presents the *Dynamic Probes* and *Quantile-based* algorithms for sorted lists. Section 4.2 presents the *Hash-based* algorithm for unsorted lists.

Assume we wish to compute the intersection of k lists $A_1, \ldots, A_k$, where $k \geq 2$. Each list A_i contains $|A_i|$ unique elements (ids); the total number of elements in the lists is $N = \sum_{i=1}^{k} |A_i|$. We assume that the lists reside in main memory and the result of the intersection is also written in memory.

4.1 The Case of Sorted Lists

4.1.1 Dynamic Probes Algorithm

The basic idea explored by most of the algorithms proposed for intersecting k lists is to use the elements of one list as eliminators and conduct *probes* (lookups) in the remaining lists using a search method (binary search, galloping search, etc). The *probing order* (order in which the lists are probed) and the search interval length of each probe affects the performance of a list intersection algorithm.

For every probe, the number of steps (comparisons) required to either locate an element or exclude it from the intersection depends on the length of the search interval. Due to the random access pattern that most search methods exhibit, every step results in a random memory access that with high probability cannot be resolved through cache. Furthermore, an effective probing order can speedup the computation of intersection. Probing the lists that are less likely to contain an element first, reduces the number of probes that is required to exclude an element from the intersection.

The *Dynamic probes* (DP) algorithm dynamically decides the probing order and the length of each probe using information from previous probes. This information is utilized as a cache-resident *micro-index*. Let us describe how the DP algorithm computes the intersection of k sorted lists $A_1, \ldots, A_k$. The pseudo-code for the DP algorithm is presented in Algorithm 2. Initially, the lists are sorted by increasing length ($|A_1| \leq |A_2| \leq \ldots \leq |A_k|$) and the first element of the smallest list is set as eliminator, $e = A_1[0]$. A binary search is performed for e in the remaining lists $A_2, \ldots, A_k$ and two arrays Val, Pos are utilized to store the *ids* and the *positions*, respectively, of the elements that are "touched" during a binary search. For every list A_i, the ids are stored in the i-th row of array Val and the positions are stored in the i-th row of array Pos (lines 3-5). Val and Pos store the ids and positions of only the elements with ids larger than e. Val and Pos are two $(k-1) \times \log n_{max}$ arrays, where n_{max} is the number of elements in the largest list. Next, we describe how the information stored in these two arrays is utilized as a micro-index to speedup following probes.

In the next step, the successor of e in the current smallest unexplored list is used as eliminator, $e = succ(e)$ (line 6). Instead of using a round robin probing policy as the one applied in [14], we dynamically decide the probing order and the search interval length for e as follows. For every list A_i, we search in Val for the ids $lo\_val^i(e)$, $hi\_val^i(e)$ that bound e, i.e. $lo\_val^i(e) < e < hi\_val^i(e)$ (line 9). The corresponding entries in Pos ($lo\_pos^i(e)$, $hi\_pos^i(e)$) of $lo\_val^i(e)$ and $hi\_val^i(e)$, define a range of positions to search for e in list A_i. e will either be found in range $[lo\_pos^i(e), hi\_pos^i(e)]$ or it does not exist in that list. Given their small size and the fact that they are frequently accessed, Val and Pos will reside with high probability in some level of the on-chip cache hierarchy. Consequently, computing the search intervals for e imposes minimal overhead. Once we determine for every list the search interval of e, we sort the intervals by increasing length (line 15) and we greedily probe the lists in that order using interpolation search (line 20). The same procedure is applied repeatedly until one of the lists is exhausted.

In Figure 1, we illustrate a sample run of DP in three lists A_1, A_2, A_3 with 5, 100 and 150 elements (document identifiers), respectively. DP uses the first id (100) of list A_1 as eliminator and performs a binary search in A_2 and A_3. A binary search for id 100 in A_2 locates id 100 at position 6 and "touches" the following position-id pairs: (50-750), (25-200), (12-150). Hence, the second row of Val is populated with ids (150, 200, 750) and the second row in Pos is populated with their positions (12, 25, 50). In accordance, after performing a binary search for id 100 in list A_3, the

third row of Val stores ids (350, 500, 700) and the third row of Pos stores their positions (18, 37, 75).

Now, consider a search for the second id (400) of A_1. Any list intersection algorithm will consider as search interval the range $(6, 100]$ of positions in A_2 and the range $(9, 150]$ of positions in A_3. Moreover, if the probing order is determined using the remaining unexplored portion of each segment, then A_2 is probed first and A_3 second. Instead, the DP algorithm operates in a less conservative manner exploiting the entries in Val and Pos to refine the search interval for id 400. By comparing the id searched with the ids stored in Val, it sets the search interval to be the range $[25, 50]$ of positions in A_2 and the range $[18, 37]$ of positions in A_3. Furthermore, since the search interval in A_3 is smaller, A_3 is probed first and A_2 second.

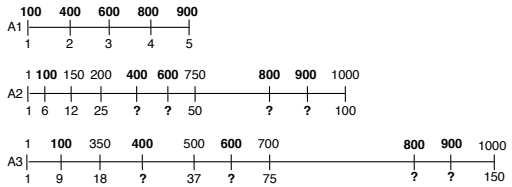


Figure 1: Example of *Dynamic Probes* algorithm.

The entries in arrays Val and Pos are updated in two cases using binary search: a) when the length of the search interval (number of elements) is larger than a threshold T_{DP}, and b) when for a given eliminator e and a list A_i, it is not possible to identify a search interval in A_i using the entries in Val and Pos (lines 10-13). The latter case occurs when the value of e is larger than all the values in the i-th row of Val. In that case, the remainder unexplored portion of A_i determines the search interval of e.

For example, consider searching for id 800 of A_1 in Figure 1. Id 800 is larger than all the ids stored in Val for both A_2 and A_3. Hence, we set $lo\_pos^i(e)$ to be the maximum between the position in A_i where the search for the predecessor of e halted and the last entry in the i-th row of Pos (maximum position in i-th row). We set $hi\_pos^i(e)$ to be the length of list A_i, $|A_i|$. Thus, for id 800, the search interval is the range $[50, 100]$ of positions in A_2 and the range $[75, 150]$ of positions in A_3. When probing list A_i for e, we perform a binary search and we update the entries in the i-th row of Val and Pos to benefit subsequent probes (lines 17-18); a binary search is performed in A_2 and A_3 for id 800. In Section 5 we describe how the value of T_{DP} is determined.

4.1.2 Quantile-based Algorithm

When we are dealing with real life documents such as news articles or blogs, we may often encounter situations in which the distribution of document identifiers (ids) to lists is highly non-uniform. Demaine et al. referred to this non-uniformity as "burstiness" and argued that although adaptive list intersection algorithms may be theoretically superior to static algorithms, in practice the effectiveness of adaptivity depends on the burstiness of the actual data [14]. Based on these findings, we are trying to detect as early as possible lists with non-uniform distribution of document identifiers and select in advance a good probing order to eliminate the overhead of adaptivity.

In order to determine the existence of lists with non-uniform distribution of ids, some form of data inspection is required. When a serial list intersection algorithm is employed, inspecting the lists introduces notable overhead that offsets any benefit attained. On the contrary, computing the intersection in a CMP environment provides a unique opportunity to examine the lists during the execution of a load balancing algorithm. The load balancing algorithms

Algorithm 2 *Dynamic Probes*

Input: k sorted lists, $A_1, \ldots, A_k$, threshold T_{DP}
Output: The elements in list intersection
1: Sort the lists by length ($|A_1| \leq |A_2| \leq \ldots \leq |A_k|$).
2: Choose the eliminator e to be the first element in A_1.
3: **for** each list A_i, $i = 2 \ldots k$ **do**
4: Perform a binary search for element e and store the values and the positions of the elements "touched" by the binary search in arrays Val and Pos, respectively.
5: **end for**
6: Set new eliminator $e = succ(e)$.
7: **while** the eliminator $e \neq \infty$ **do**
8: **for** each list A_i, $i = 2 \ldots k$ **do**
9: Using Val and Pos, find the search interval of e in list A_i.
10: **if** a search interval is not found for list A_i using Pos and Val **then**
11: Set the maximum possible search interval of A_i.
12: Mark A_i for binary search.
13: **end if**
14: **end for**
15: Sort lists by increasing search interval ($|A_2| \leq \ldots \leq |A_k|$).
16: **for** $i = 2 \ldots k$ **do**
17: **if** A_i was marked for binary search **or** the search interval is larger than T_{DP} **then**
18: Perform binary search for e in A_i and update Pos and Val entries.
19: **else**
20: Perform interpolation search in the search interval of A_i.
21: **end if**
22: **if** e is found in k lists **then**
23: Output e.
24: **end if**
25: **end for**
26: Set new eliminator $e = succ(e)$.
27: **end while**

presented in Section 3 generate a statistical description of the lists in the form of approximate quantiles.

In this section, we present a list intersection algorithm, termed *Quantile-based* (QB), that utilizes quantiles to detect lists with non-uniform distribution of ids. QB recursively splits the partitions created by the load balancing algorithm into sub-partitions for which it determines a "good" probing order. The probing order of a partition remains fixed while that partition is being processed, thus avoiding the overhead of adaptivity. Pseudo-code for the QB algorithm is presented in Algorithm 3.

Let us illustrate the QB algorithm with a simple example. In Figure 2, we present parts of two sorted lists A_1, A_2 that belong to a partition p. We use the same notation (A_i) to refer to a list and the part of that list that belongs to a partition. The median of the ids in p is 7. Given the position of the median in each list, it is evident that the distribution of ids in A_1, A_2 is non-uniform; A_1 contains six elements with values which are smaller than 7, while A_2 contains only two such elements. Now, consider how different probing policies would work in this setting. The adaptive probing policy [14] performs 10 probes to compute the intersection (shown in Figure 2). A probing policy with a fixed probing order requires either 9 or 10 probes depending on the order used.

Algorithm QB on the other hand, utilizes the median to detect the existence of non-uniformity in the distribution of ids (lines 3-8). The median splits every list A_i in two sub-lists A_i^1 and A_i^2, e.g. A_1 is divided into $A_1^1 = \{1, 2, 3, 4, 5, 6, 7\}$ and $A_1^2 = \{8, 11, 13\}$. For every list, QB compares the length difference of its sub-lists with a threshold T_{QB} (*non-uniformity* condition) to identify non-uniformity in the distribution of ids. In Section 5 we demonstrate how the value of T_{QB} is determined.

If at least one list A_i satisfies the *non-uniformity* condition, i.e.

$||A_i^1| - |A_i^2|| > T_{QB}$, QB divides p in two sub-partitions p_1, p_2 (line 6). p_1 consists of sub-lists A_1^1, A_2^1 with elements which are smaller or equal than the median (shown in Figure 2). Accordingly, p_2 consists of sub-lists A_1^2, A_2^2 with elements which are larger than the median. For each sub-partition p_i, QB greedily selects a probing order by sorting the sub-lists of p_i in increasing length ($A_2^1 \rightarrow A_1^1$ for p_1 and $A_1^2 \rightarrow A_2^2$ for p_2) resulting in 6 probes.

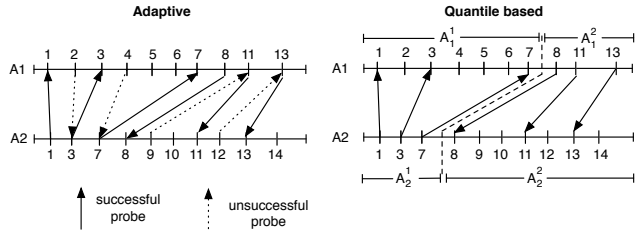


Figure 2: Different probing policies in two segments with non-uniform distributions of document identifiers.

In general, the same procedure is applied recursively for every newly created sub-partition until either a threshold of maximum number of generated partitions allowed is exceeded or until none of the lists satisfies the *non-uniformity* condition (lines 2-10). List intersection is computed in every (sub-)partition using the elements of the current smallest unexplored list as eliminators and probing the remaining lists in the probing order derived (lines 11-12). Each probe is performed using interpolation search.

Algorithm 3 *Quantile-based*

Input: ϵ-approximate quantile summary Q, k lists of partition P, $A_1, \ldots, A_k$, threshold T_{QB},
Output: The elements in list intersection
1: enqueue(P); $subpartitions = 0$
2: **while** queue is not empty() && $subpartitions <$ maximum number of sub-partitions allowed **do**
3: P' = dequeue()
4: Compute the approximate median m of P' using Q.
5: Search for m in each list A_i, $i = 1, \ldots, k$, of P'.
6: **if** at least one list of P' is split by the median into sub-lists with length difference which is larger than T_{QB} **then**
7: Split P' in two sub-partitions P_1, P_2 using m.
8: enqueue(P_1); enqueue(P_2); $subpartitions+ = 2$
9: **end if**
10: **end while**
11: Sort the lists by increasing length to determine the probing order for that (sub-)partition.
12: Use the elements of the current smallest unexplored list as eliminators. Probe the remaining lists using interpolation search.

4.2 The Case of Unsorted Lists

When the lists are unsorted, one way to compute the intersection would be to sort the lists and then apply any algorithm for sorted lists. However, sorting imposes an overhead. In this section we present *Hash-based*, an algorithm that computes the intersection of unsorted lists in a CMP context. It is a cache-conscious algorithm that uses hashing and avoids the overhead of sorting.

4.2.1 Hash-based Algorithm

The *Hash-based* (HB) algorithm utilizes the *Partition Unsorted* algorithm (Section 3.2.2) to partition the lists. Furthermore, it exploits the partial sorting of the lists, that *Partition Unsorted* performs, to locate the elements of each partition at low cost without examining all the elements in the lists. Then, hashing is utilized to identify the elements of intersection.

Algorithm 4 presents pseudo-code for the HB algorithm. The *Partition Unsorted* technique (Section 3.2.2) is utilized to compute an ϵ-summary Q of the elements in k unsorted lists $A_1, \ldots, A_k$ (line 1). Using Q, the lists are divided into P partitions, where $P = \lceil \frac{N \cdot C}{|L2|} \rceil$ (line 2). The size of each partition p is chosen such that C partitions can fit in L2 simultaneously, i.e. $|p| = \frac{|L2|}{C}$. Note, that since the lists are unsorted, at the present time we only know the boundary values $(lo(p), hi(p))$ of p. Next, the P partitions are assigned to cores so that each core processes approximately the same number of partitions, i.e. $\frac{P}{C}$.

The next step, for a core processing a partition p, is to locate the positions of the elements that belong to p in the lists. The naive approach is to read all the elements from the lists and filter out those that do not belong to p. However, this approach imposes an overhead, which is the cost of reading all the elements in the lists. Instead, the HB algorithm exploits a partial sorting of the lists (performed by *Partition Unsorted*) to locate the elements of p without reading undesirable elements, i.e. elements that belong to other partitions.

Recall that the *Partition Unsorted* algorithm generates, along with a quantile summary Q, a number $m_l = \lceil \frac{|A_l| \cdot C}{|L2|} \rceil$ of sorted runs per list A_l ($r_1, \ldots, r_{m_l}$), each containing $\frac{|L2|}{C}$ elements. The core that processes partition p utilizes these sorted runs to locate the elements of p as follows (lines 4-8). In every sorted run of each list, it performs two binary searches for the boundary values of p $(lo(p), hi(p))$. For a sorted run r, the two binary searches determine the range of positions in r ($[lo\_pos(p), hi\_pos(p)]$) with the elements that belong to p.

In Figure 3, we present the execution of the HB algorithm for two unsorted lists A_1, A_2 using a CMP with two cores. Let us assume for simplicity that the number of partitions is two ($P = 2$) and the boundary values are $(1, m)$ for P_1 and (m, MAX_ID) for P_2, where m is the median in the union of A_1, A_2 and *MAX_ID* the maximum document identifier in A_1, A_2.

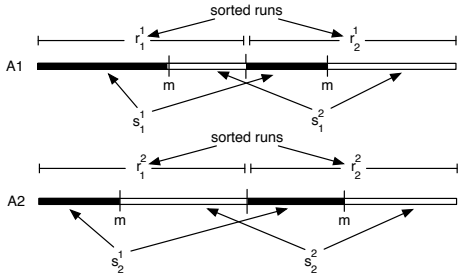


Figure 3: Example of *Hash-based* algorithm for two unsorted lists on a CMP with two cores.

Assume that every list in Figure 3 consists of two sorted runs, produced by *Partition Unsorted* algorithm. Furthermore, assume that core 1 processes partition P_1 and core 2 processes partition P_2. Core 1 performs a binary search for the median m on each sorted run of A_1 (r_1^1, r_2^1) and on each sorted runs of A_2 (r_1^2, r_2^2). The position of m in a sorted run r divides the elements of r in two partitions, P_1 (denoted in black) and P_2 (denoted in white). The elements of A_1 that belong to P_1 are distributed in two sorted runs, i.e. the black parts of r_1^1 and r_2^1. In accordance, the elements of A_2 that belong to P_1 are distributed in the two sorted runs of A_2 (black parts of r_1^2 and r_2^2).

Next, we describe how a partition p is processed to compute the intersection. We refer to the set of elements of list A_i that belong to

partition P_j as *segment* s_i^j. Let $s_1, \ldots, s_k$ be the segments of p (superscript is omitted for brevity). Initially, the segments are sorted by increasing length, $|s_1| \leq \ldots \leq |s_k|$ (line 10). Algorithm HB uses the elements of the smallest segment, s_1 to build a hash table and the elements of the remaining segments, $s_2, \ldots, s_k$ to probe that hash table and compute the intersection (lines 11-20). The entries of the hash table are $\langle$*id, counter*$\rangle$ pairs. *Id* is used as a key and *counter* is the payload recording the number of lists in which this *id* exists; *counter* is initialized to one. Every probe to an existing *id* increments the corresponding counter. When the value of *counter* becomes equal to the number of lists, the corresponding *id* belongs to the intersection. In Figure 3, core 1 (processing partition P_1) utilizes segment s_2^1 to build a hash table and segment s_1^1 to probe the hash table and compute the intersection (assuming $|s_2^1| < |s_1^1|$). In a similar fashion, core 2 processes partition P_2.

Since the size of each partition is chosen appropriately, the cores are not competing for L2 while the intersection is computed. With high probability, each hash table will reside in the L2 cache and hash probes will not cause any off-chip memory accesses. The cores are experiencing only compulsory misses while reading the elements of a partition and conflict misses due to cache associativity.

Algorithm 4 *Hash-based*

Input: N elements in total in k lists $A_1, \ldots, A_k$, C cores, $m_l = \left\lceil \frac{|A_l| \cdot C}{|L2|} \right\rceil$ sorted runs $r_1, \ldots, r_{m_l}$ per list A_l
Output: The elements in list intersection
1: Compute an ϵ-approximate quantile summary Q using *Partition Unsorted* algorithm.
2: Using Q, divide the lists into P approximately equal sized partitions, where $P = \left\lceil \frac{N \cdot C}{|L2|} \right\rceil$.
3: **while** $\exists$ unprocessed partitions **do**
4: Process the next unprocessed partition p. // Each core processes a disjoint set of partitions.
5: **for** $l = 1$ to k **do**
6: **for** $j = 1$ to m_l **do**
7: Search for the boundary values of p in r_j^l. // The output is a range of positions in r_j^l, $[lo\_pos(p), hi\_pos(p)]$.
8: **end for**
9: **end for**// The segments $s_1, \ldots, s_k$ of p have been computed.
10: Sort the segments of p by increasing length ($|s_1| \leq \ldots \leq |s_k|$).
11: Build a hash table h using the elements of s_1 (smallest segment).
12: **for** $i = 2$ to k **do**
13: Probe h using the elements of s_i.
14: **if** a match e is found **then**
15: Increment $counter(e)$.
16: **if** $counter(e) == k$ **then**
17: Output element e.
18: **end if**
19: **end if**
20: **end for**
21: **end while**

5. EXPERIMENTAL EVALUATION

In this section we present a series of experiments on sorted and unsorted lists using a chip multiprocessor with eight cores. We compare the *Dynamic Probes* (DP) and the *Quantile-based* (QB) algorithms with *Small Adaptive* (SA) [14], *Probabilistic Probes* (PP) [27] and *Linear Merge* (LM). The latter computes the intersection of sorted lists using merging similar to the execution of a sort-merge join algorithm. Alternatively, one could view the problem of intersecting k lists as a special case of joining k relations and compare the performance of DP and QB algorithms against static AND-trees. However, as Raman et al. demonstrated, adaptive list intersection algorithms perform in most cases at least as good as the best AND-tree and are not prone to cardinality estimation errors [27]. Hence, we compare our algorithms only against other adaptive list intersection algorithms.

For unsorted lists, we evaluate the performance of the *Hash-based* (HB) algorithm. Due to the lack of a multi-way hash join algorithm for CMPs, we compare HB against an adaptation of a parallel and cache-conscious sorting algorithm [15] for CMPs (PS for brevity). PS is a variant of AlphaSort [26] that utilizes an exact partitioning technique [22]. The sorting algorithm can be used in conjunction with any algorithm for sorted lists. We experiment with synthetically generated datasets as well as with real data. Section 5.1 discusses the datasets, the hardware configurations, and implementation details. The results are presented in Section 5.2.

5.1 Methodology

We experiment with real and synthetic datasets. Our synthetic data capture uniform and correlated distributions of document identifiers. Uniform synthetic data were generated utilizing the guidelines on [17]. For real data we use a sample corpus from a web search engine[2] [6]. The size of the corpus is 3.1GB of raw text and the query log contains 1000 multi-keyword queries.

All the experiments were conducted on a Dell PowerEdge 2950 server with dual quad-core Intel E5355 CPUs (8 cores) running at 2.6GHz. It has a 4MB of shared L2 cache per quad cores (8MB in total) and each core has a private 64KB L1 cache. It runs the Linux operating system with kernel 2.6.15 and has 32GB of main memory. All the algorithms were implemented in C using Posix threads (Pthreads). We used the Intel icc-10 compiler with optimization level O3. In all the experiments conducted, the *sched_setaffinity* function of the Linux OS is employed to schedule the threads to specific cores and assure that they are not re-scheduled during the execution of the algorithms.

5.2 Experimental Results

In all the experiments we measured the wall-clock time required to read the lists from main memory, conduct partitioning using the quantile algorithms (for sorted and unsorted lists), compute the intersection and write the intersection result in memory. For the case of unsorted lists, where we evaluate intersection time utilizing sorting, the time includes sorting the lists, computing the partitioning using the quantile algorithm, computing the intersection using the best algorithm for sorted lists and writing the intersection result in memory. All the numbers reported are averages over 100 runs. The standard deviation for all the algorithms was less than 2%.

The maximum possible size of the intersection of k lists is the length of the smallest list. We consider the *selectivity* of a list intersection to be the ratio of the size of the intersection to the maximum possible size of the intersection. Unless stated otherwise, all the lists in the synthetic datasets have the same size, containing one million elements. Each element is a four byte integer. In Table 1, we present the range of parameter values examined in our experiments.

List size	1 - 100 million ids
Cores	1 - 8
Lists	2 - 16

Table 1: Range of parameter values in our experimental design.

All the algorithms examined utilize the list partitioning techniques presented in Section 3 to achieve load balancing. In all the experiments that follow, we set the error threshold at $\epsilon = 0.01$.

[2]www.blogscope.net

The maximum load disparity across cores observed in all the experiments conducted (using eight cores) never exceeded 10%.

5.2.1 Sorted Lists

In the first set of experiments, we assess the performance of list intersection algorithms at different selectivities and different number of sorted lists using uniform synthetic data. Figures 4, 5, 6 present the results for 4, 8 and 16 lists, respectively when four cores are utilized. We present the normalized time of each algorithm with respect to LM. The latter exhibits the worst performance with running times reaching 0.3 seconds (for 16 lists and 4 cores).

For four sorted lists (Figure 4) and when the selectivity is small (1%), SA and QB have approximately the same performance with a small advantage for the latter (10%). SA cycles repeatedly through the lists only when a match is found in the two smallest lists currently being intersected. When the selectivity is small, the frequency at which matching elements are found is proportionally small too. Hence, SA changes its probing order less frequently and does not pay the overhead of cycling through the lists. For larger selectivities though, where matching elements are discovered more frequently, SA acts more adaptively, continuously updating its probing order. That has a negative impact on its performance compared to algorithms that update their probing policy more effectively, such as DP and QB. As the number of lists increases (Figures 5, 6), the overhead of repeatedly cycling through a large number of lists results in SA performing worse than DP and QB even for small selectivities.

For small selectivities and small number of lists SA performs better than DP. However, as selectivity increases (more than 10% in our experiment), DP performs better than SA and the performance difference between these two algorithms is more pronounced. For every eliminator, DP consults the micro-index in order to refine the search range for every list and to determine the best probing order. Even though the micro-index is cache-resident, there is still some overhead involved in this procedure. On the other hand, when selectivity is small, SA discards most of the eliminators by performing a single probe on the second smallest list. Hence, it is more efficient than DP. However, as selectivity increases and the lists are probed more frequently, the use of the micro-index starts paying off and DP performs better than SA.

For small selectivities, DP and QB have approximately the same performance. However, for large selectivities the DP algorithm performs worse than QB. The latter determines a good probing order at low cost which remains fixed during its execution. The benefit of the simple probing policy that QB applies is more profound for larger selectivities and larger number of lists where more adaptive algorithms such as SA, DP and PP experience large overheads.

Algorithm LM has the worst performance overall for the range of selectivities and the number of lists examined. Compared to other intersection algorithms, it reads the lists sequentially resulting in better utilization of cache lines. Furthermore, the sequential memory access pattern of LM is detected by the hardware prefetcher of the Intel architecture. For small number of lists the sequential memory access pattern of LM compensates for the larger number of comparisons it performs (compared to other intersection algorithms); for two sorted lists the performance of LM is equivalent to the performance of other list intersection algorithms (results omitted due to space constraints). As the number of sorted lists increases, the execution time of LM is dominated by the excessive number of comparisons and the performance difference between LM and the rest of the algorithms increases proportionally.

The PP algorithm, although it performs significantly better than LM, is $1.6X$ slower than QB and $1.44X$ slower than SA for four lists and small selectivities (1%). The PP algorithm repeatedly cycles through the lists with some probability p and chooses the next "promising" list to probe with probability $1 - p$. Unless there is a probing order that is highly beneficial, PP will behave like the *Adaptive* algorithm [13]; the latter has been shown to be inferior to SA. The results in Figures 4, 5, 6 demonstrate that the PP algorithm is consistently worse than SA regardless of selectivity and number of lists.

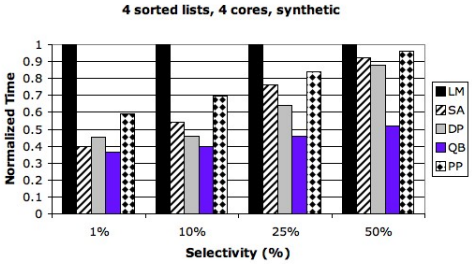


Figure 4: Performance of intersection algorithms for four sorted lists with synthetic data using four cores.

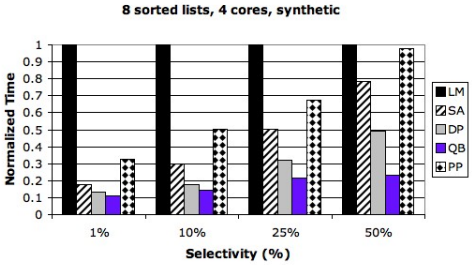


Figure 5: Performance of intersection algorithms for eight sorted lists with synthetic data using four cores.

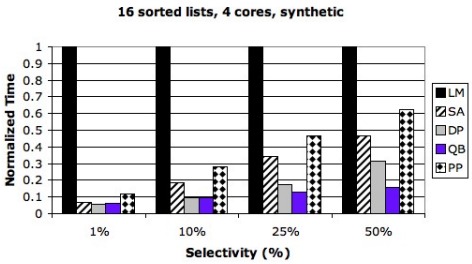


Figure 6: Performance of intersection algorithms for sixteen sorted lists with synthetic data using four cores.

We conducted the same set of experiments utilizing eight cores. Figure 7 presents the results for sixteen lists (normalized time with respect to LM). The results for smaller number of lists are consistent with those presented in Figures 4 and 5. The most important observation is that, when eight cores are utilized, the relative performance of the intersection algorithms does not change, indicating the effectiveness of the load balancing technique applied.

Next, we wanted to verify that our results are consistent when larger lists are intersected. Figure 8 presents the performance of list intersection algorithms for four lists when every list contains 100 million elements; four cores are used in this experiment. The results presented in Figure 8 are similar to those in Figure 4, verifying that our results are consistent irrespective of list size.

5.2.1.1 Anti-correlated Data.

In all experiments presented thus far, the document identifiers that belong to the intersection were positioned randomly in each

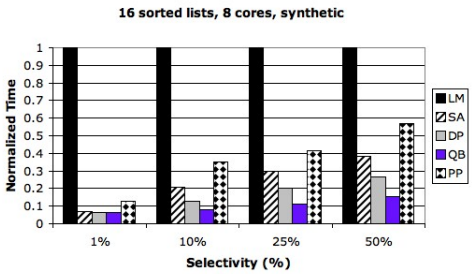


Figure 7: Performance of intersection algorithms for sixteen sorted lists with synthetic data using eight cores.

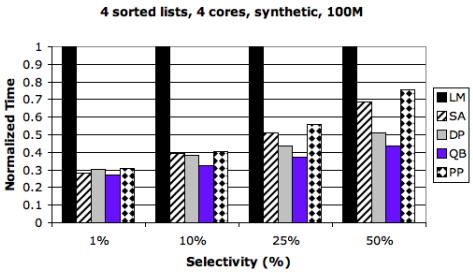


Figure 8: Performance of intersection algorithms for four sorted lists with synthetic data using four cores when each list contains 100 million elements.

list. In the next experiment, we study the performance of intersection algorithms for sorted lists in the case where the positions of the document identifiers that belong to the intersection are anti-correlated. This means that these document identifiers are clustered (placed close to each other in each list) and the clusters are in different positions in each of the lists. The position of each cluster is selected randomly.

Figure 9 presents the results for 16 sorted lists with anti-correlated data when eight cores are utilized. LM is not included in these results as its performance does not depend on the distribution of the document identifiers in the lists. We present the normalized time of each algorithm with respect to PP; its running times reach 0.03 seconds. DP and QB perform consistently better than SA and PP. As selectivity increases, the performance difference of the proposed algorithms with SA and PP is more pronounced. QB performs consistently better than DP and SA as it is able to detect the non-uniformity in the lists and select from the beginning a good probing strategy. For large selectivities and large number of lists (50% in our experiments) QB is $2.5X$ faster than DP and $6X$ faster than SA and PP. The anti-correlated document identifiers degrade the performance of SA due to its conservative round robin probing policy. PP is slightly worse than SA. These results are consistent for different number of lists as well as for correlated data (omitted due to space constraints). In all cases examined, QB is the algorithm of choice.

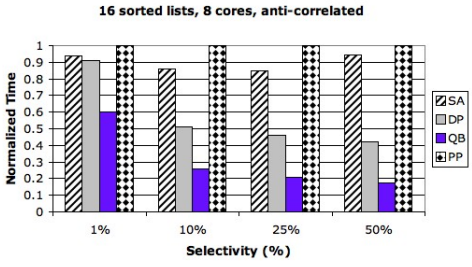


Figure 9: Performance of intersection algorithms for sixteen sorted lists with anti-correlated data using eight cores.

5.2.1.2 *Real Data.*

The performance comparison of intersection algorithms as a function of the number of sorted lists using real data is presented in Figure 10; four cores were utilized. For each algorithm we measured the average time to execute a set of queries with the same number of keywords. We present the normalized average time of each algorithm with respect to PP; its running times reach 0.05 seconds. One characteristic exhibited by real data distributions on inverted lists is that the resulting selectivities of intersections is low. Hence, in practice the LM algorithm is not considered to be a competitive algorithm. SA performs better than DP for small number of lists but as we increase the number of lists the conservative probing policy of SA reduces its performance. QB has the best performance and the performance difference with the other algorithms is increasing for larger number of lists. Notice that for 6 lists QB is $1.6X$ faster than SA and PP; the latter has approximately the same performance as SA.

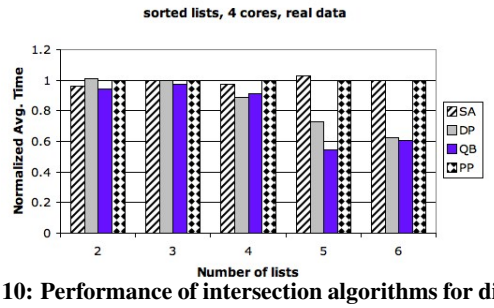


Figure 10: Performance of intersection algorithms for different number of sorted lists with real data using four cores.

5.2.1.3 *Speedup.*

In the next set of experiments we measure the speedup of the proposed algorithms for sorted lists as well as the performance of the parallel partitioning algorithm. In Figures 11 and 12 we demonstrate the performance of DP and QB algorithms, respectively, for computing the intersection of eight lists as we increase the number of cores; selectivity is 10%. We present the normalized running time with respect to the case when a single core is utilized (serial algorithm). We also present the fraction of time spend on the partitioning technique.

Both algorithms achieve adequate speedup as we increase the number of cores from one to four, verifying the effectiveness of the proposed partitioning technique. However, we observe that as we increase the number of cores to eight, the reduction in total running time of the DP algorithm is merely 27% compared to the case where four cores are utilized. Interestingly, the intersection time is reduced approximately 50%, indicating the ability of the partitioning technique to balance the load evenly. The problem however, as we increase the number of cores, is the increase in the partitioning cost. Recall that our partitioning technique is tailored to CMP architectures and relies on the fact that the cores share the L2 cache. However, the hardware that we utilized contains two quad-cores which communicate through main memory. This has the following implication. In the last step of the partitioning technique a single core is producing the final ϵ-summary by merging a number of summaries. Since each quad-core has its own shared L2 cache, half the accesses to the quantile summaries will have to be resolved through main memory, thus increasing the partitioning cost.

In CMP architectures in which all the cores share the L2 (L3) cache, all the quantile summaries will reside on the same cache hierarchy –as it happens when four cores are utilized in our experiment–

and the partitioning cost will be significantly lower. Analogous is the case for the QB algorithm (presented in Figure 12).

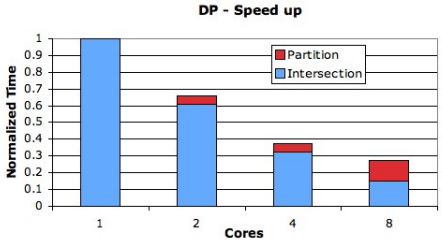


Figure 11: Performance of the DP algorithm as we increase the number of cores.

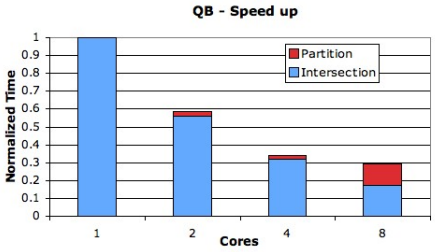


Figure 12: Performance of the QB algorithm as we increase the number of cores.

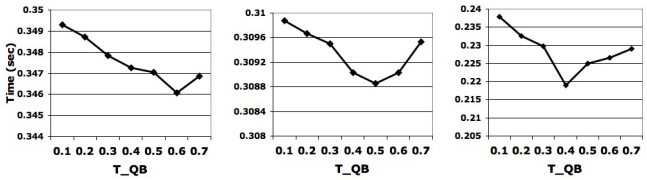
5.2.1.4 *Micro-benchmarks.*

We run several micro-benchmarks to identify the best values for the thresholds used in algorithms QB and DP. Recall that algorithm QB utilizes threshold T_{QB} in a *non-uniformity* condition, to decide whether a partition should be divided into smaller sub-partitions. The length of the search interval currently examined is compared to threshold T_{DP} in DP, to decide whether binary search should be applied to update the cache resident index.

We measured the running time of the QB algorithm using real data for increasing number of lists as a function of the value of T_{QB}. We report the average of 100 runs in Figure 13 for two, three and four lists. As observed, a single global optimal value for T_{QB} does not exist. The value of T_{QB} that results in the best performance depends on the number of lists intersected. A second observation is that the optimal value of T_{QB} is shifted towards smaller values as the number of lists increases. As the number of lists increases, a wrong decision in the probing policy applied has an even larger (negative) impact on performance. For small values of T_{QB}, QB tends to be less conservative, changing its probing order more often to handle fluctuations in the distribution of document identifiers.

The results for larger number of lists are consistent with those presented here (omitted due to space constraints). We choose to set the value of T_{QB} in our experiments according to these micro-benchmarks. Namely, for each experiment involving a number of lists k, we set T_{QB} to the best value observed in the micro-benchmarks for k lists.

Similar trends are observed in the micro-benchmarks conducted for the DP algorithm (omitted due to space constraints). As the number of lists increases, the optimal value of T_{DP} that results in the best performance for DP is shifted towards smaller values. For small values of T_{DP}, DP updates its cache-resident index more often and probes the lists more effectively. Similarly, in our experiments we set T_{DP} values according to the values yielding the best performance in these micro-benchmarks.



(a) 2 lists (b) 3 lists (c) 4 lists

Figure 13: Impact of threshold T_{QB} on the performance of QB algorithm on real data.

5.2.2 *Unsorted Lists*

In this section we evaluate the performance of intersection algorithms for unsorted lists using real and synthetic data. We compare the performance of *Hash-based* (HB) with that of sorting (PS) in conjunction with the best intersection algorithm for sorted lists; the QB algorithm is used in all the experiments. The selectivity of the intersection does not affect the performance of HB and PS algorithms. Their cost depends solely on the number of elements in the lists. Hence, in all the experiments we fix the selectivity at 10%.

In Figure 14 we present a performance comparison of HB and PS in conjunction with the QB algorithm as we increase the number of unsorted lists from 2 to 16, when eight cores are utilized. All the lists have the same length (1 million unique ids).

As it is evident in Figure 14, HB performs consistently better than the combination of PS and QB for different number of unsorted lists. HB utilizes the *Partition Unsorted* algorithm to leave the data partially sorted and computes the intersection without sorting the lists completely. Each core reads only the elements of the partition currently processed. Hence, the elements of the lists are read only once. However, HB pays the overhead of building and probing hash tables. We reduce this overhead by building hash tables using the elements of the smallest segments. Moreover, since the partitions are cache resident, probing the hash tables results in smaller overhead compared to the cost of sorting the lists (performed by PS). In addition to reading the elements once for performing the final merge phase, PS pays the overhead of storing the final sorted run in memory. Finally, we notice that the cost of computing the intersection using QB is relatively small compared to the cost of sorting the lists.

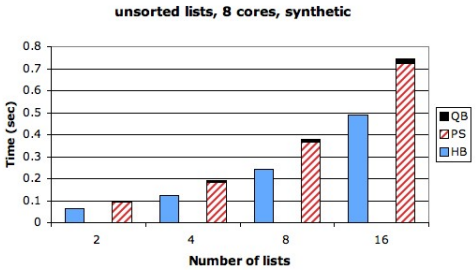


Figure 14: Performance of intersection algorithms for different number of unsorted lists with synthetic data using eight cores.

A performance comparison of the algorithms for unsorted lists using real data, when eight cores are utilized is presented in Fig 15. HB performs consistently better than the combination of sorting and QB (best algorithm for sorted lists with real data), verifying the efficacy of HB in practice.

In the next experiment we measure the speedup of the HB algorithm as well as the cost of the partitioning technique as we increase the number of cores (Figure 16). The first observation is the good

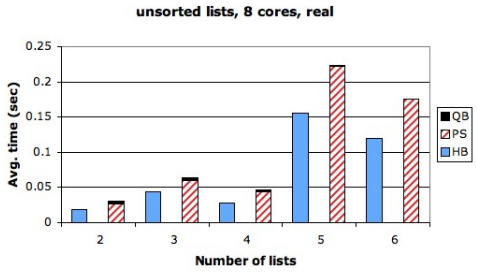


Figure 15: Performance of intersection algorithms for different number of unsorted lists with real data using eight cores.

speedup achieved by the HB algorithm, verifying the effectiveness of the partitioning technique. The second observation is that the total running time of HB is dominated by the partitioning cost and that the cost is significantly reduced as we increase the number of cores, demonstrating the necessity for a parallel partitioning technique. Note that, in contrast to the case of sorted lists, the partitioning technique is also employed when a single core is utilized, in order to improve cache locality.

In contrast to the speedup experiments of DP and QB algorithms (Figures 11 and 12), when eight cores are utilized we achieve significant reduction in the partitioning cost even though the cores do not operate on the same shared cache. The reason is that for unsorted lists, that cost is dominated by the sorting phase which is performed independently by each core.

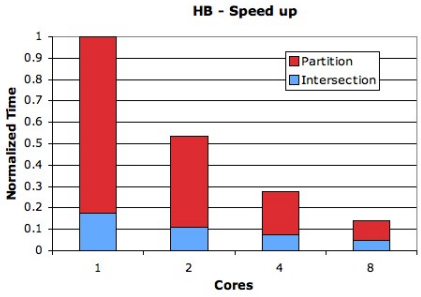


Figure 16: Performance of the HB algorithm as we increase the number of cores.

6. CONCLUSIONS

We have presented algorithms to compute the intersection of an arbitrary number of sorted and unsorted lists tailored to commodity chip-multiprocessors.

The problem of load balancing is initially studied and cache-conscious algorithms for partitioning sorted and unsorted lists in a CMP context have been presented. Two new intersection algorithms have been proposed for sorted lists: *Dynamic Probes*, and *Quantile-based*. *Dynamic Probes* exploits information from previous probes as a cache-resident index. *Quantile-based* utilizes quantiles to detect lists with non-uniform distributions of document identifiers and select in advance and at low cost a good probing policy. For unsorted lists we have proposed a cache-conscious intersection algorithm, termed *Hash-based*, that computes the intersection of unsorted lists using hashing.

In a detailed experimental evaluation using real and synthetic data on a CMP with eight cores, we demonstrate that our intersection algorithms for sorted and unsorted lists have superior performance and achieve very good speedup due to the effectiveness of the load balancing strategy.

7. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, pp. 266–277, 1999.

[2] AMD. Multi-core processors - the next evolution in computing, AMD white paper, 2005.

[3] R. A. Baeza-Yates. A fast set intersection algorithm for sorted sequences. *LNCS*, 3109:400–408, 2004.

[4] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[5] R. A. Baeza-Yates and A. Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *SPIRE*, pp. 13–24, 2005.

[6] N. Bansal and N. Koudas. Blogscope: A system for online analysis of high volume text streams. In *VLDB*, pp. 1410–1413, 2007.

[7] J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *SODA*, pp. 390–399, 2002.

[8] J. Barbay, A. Lopez-Ortiz, and T. Lu1. Faster adaptive set intersections for text searching. *LNCS*, 4007:146–157, 2006.

[9] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5:82–87, 1976.

[10] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Inspector joins. In *VLDB*, pp. 817–828, 2005.

[11] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, pp. 339–350, 2007.

[12] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Space- and time-efficient deterministic algorithms for biased quantiles over data streams. In *PODS*, pp. 263–272, 2006.

[13] E. D.Demaine, A. L. Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *SODA*, pp. 743–752, 2000.

[14] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *ALENEX*, pp 91–104, 2001.

[15] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Parallel and Distributed Information Systems*, pp. 280–291, 1991.

[16] B. Gedik, P. S. Yu, and R. R. Bordawekar. Executing stream joins on the cell processor. In *VLDB*, pp. 363–374, 2007.

[17] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD*, pp. 243–252, 1994.

[18] M. B. Greenwald and S. Khanna. Power-conserving computation of order-statistics over sensor networks. In *PODS*, pp. 275–285, 2004.

[19] F. K. Hwang and S. Lin. Optimal merging of 2 elements with n elements. *Acta Informatica*, 1(2):145–158, 1971.

[20] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal on Computing*, 1(1):31–39, 1972.

[21] Intel. Intel multi-core processor architectur development backgrounder, white paper, 2005.

[22] B. R. Iyer, G. R. Ricard, and P. J. Varman. Percentile finding algorithm for multiple sorted runs. In *VLDB*, pp. 135–144, 1989.

[23] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM POWER5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[24] S. Manegold, P. A. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. In *VLDB*, pp. 191–202, 2002.

[25] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *SIGMOD*, pp. 426–435, 1998.

[26] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. Alphasort: a risc machine sort. In *SIGMOD*, pp. 233–242, 1994.

[27] V. Raman, L. Qiao, W. Han, I. Narang, Y.-L. Chen, K.-H. Yang, and F.-L. Ling. Lazy, adaptive RID-list intersection, and its application to index anding. In *SIGMOD*, pp. 773–784, 2007.

[28] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB*, pp. 510–521, 1994.