Real-time Approximate Range Motif Discovery & Data Redundancy Removal Algorithm

Ankur Narang
IBM Research India
New Delhi, India
annarang@in.ibm.com

Souvik Bhattcherjee
IBM Research - India
New Delhi, India
souvikbh@in.ibm.com

ABSTRACT

Removing redundancy in the data is an important problem as it helps in resource and compute efficiency for downstream processing of massive (10 million to 100 million records) datasets. In application domains such as IR, stock markets, telecom and others there is a strong need for real-time data redundancy removal of enormous amounts of data flowing at the rate of 1*Gb/s* or higher. We consider the problem of finding Range Motifs (clusters) over records in a large dataset such that records within the same cluster are approximately close to each other. This problem is closely related to the approximate nearest neighbour search but is more computationally expensive. Real-time scalable approximate Range Motif discovery on massive datasets is a challenging problem. We present the design of novel sequential and parallel approximate Range Motif discovery and data de-duplication algorithms using Bloom filters. We establish asymptotic upper bounds on the false positive and false negative rates for our algorithm. Further, time complexity analysis of our parallel algorithm on multi-core architectures has been presented. For 10 million records, our parallel algorithm can perform approximate Range Motif discovery and data de-duplication, on 4 sets (clusters), in $59s$, on 16 core Intel Xeon 5570 architecture. This gives a throughput of around $170K$ records/s and around $700Mb/s$ (using records of size $4K$ bits). To the best of our knowledge, this is the highest real-time throughput for approximate Range Motif discovery and data redundancy removal on such massive datasets.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: [information filtering]; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*performance evaluation*

General Terms

Algorithm, Design, Performance

Keywords

Bloom Filter, Range Motif, Locality Sensitive Hash Function, Data Redundancy Removal, , Multi-core Architecture

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT 2011, March 22–24, 2011, Uppsala, Sweden.
Copyright 2011 ACM 978-1-4503-0528-0/11/0003 ...$10.00

1. INTRODUCTION

Data intensive computing has evolved into a central theme in research community and industry. There has been a tremendous spurt in the amount of data being generated across diverse application domains such as IR, telecom (call data records), telescope imagery, online transaction records, web pages, stock markets, medical records (monitoring critical health conditions of patients), climate warning systems and others. Processing such enormous data is computationally expensive. Removing redundancy in the data helps in improving resource utilization and compute efficiency especially in the context of stream data, which requires real-time processing at 1 Gb/s or higher. We consider the problem of eliminating redundant records present in massive datasets in real-time. A record may be considered redundant, if there exists another record present in that data stream earlier, which matches approximately (intuitively, only ϵ fraction of bits differ, for a small $\epsilon < 1$) or exactly with this record. This is also referred to as the *approximate data de-duplication* (de-dup) problem. Approximate Data redundancy removal (*ADRR*) and approximate de-duplication are used interchangeably in this paper. We also solve the more generic problem of real-time approximate Range Motif (*ARM*) discovery, over massive datasets. Intuitively, Range Motif discovery involves finding elements of a set that are closer to each other compared to elements of other sets.

For many practical application domains it suffices to find an approximate match for the query objects rather than an exact match. This observation underlies a large body of research in databases, including using random sampling for histogram estimation [7] and median approximation [25], using wavelets for selectivity estimation [26] and approximate SVD [20]. When relevant answers are much closer than the irrelevant ones, then the approximate algorithm (with a suitable approximation) will return the same result as the exact algorithm. The approximate algorithm can also provide time-quality trade-off, using appropriate tuning of the parameters.

Real-time Approximate Range Motif discovery (ARM) and Approximate de-duplication (ADRR) over massive datasets ($10M$ to $100M$ records) are computationally challenging problems. Straightforward approaches for approximate data redundancy removal (ADRR) involve pair-wise string comparisons, leading to quadratic complexity. This prohibits real-time redundancy removal over massive (10 to 100 million) number of records. ADRR can also be performed using repeated calls to the approximate nearest neighbour search (Approximate NNS) routine with each element in the given database as a single query. In the NNS problem, one needs to find K nearest neighbors to a given query object (record). For

approximate NNS, techniques based on space partitioning such as k-d trees, SR trees, etc., all degenerate to linear search, per record to be matched, for large number of dimensions [33], and hence are not scalable. However, locality sensitive hash (*LSH*) functions are known to provide scalable solution [15] for ϵ-NNS (approximate nearest neighbour search) problem. For this the known time complexity [15] is $O(d \cdot n^{1/(1+\epsilon)})$, where d is the dimension of the space of the objects in the database and n is the number of items in the database. For ADRR, using *LSH* based repeated queries, one for each element in the database, one gets the time complexity for complete ADRR as, $O(d \cdot n^{(2+\epsilon)/(1+\epsilon)})$. This is still a computationally expensive solution, for $\epsilon \sim 0$ and n in the range of millions of objects (records). In order to handle this computational challenge, we used locality sensitive hashing with Bloom filters to achieve approximate de-duplication over the complete dataset in $O(d \cdot n)$ time complexity.

Bloom filters [6] are space-efficient probabilistic data structures that provide fast set membership queries but with a small false positive rate (*FPR*). When used with locality sensitive hash functions, the resultant bloom filter is referred to as *distance sensitive* [21] Bloom filter. The distance sensitive bloom filter has both a false positive rate (*FPR*) and a false negative rate (*FNR*). The FPR and FNR can be tuned based on the requirement of the underlying application domain. While [21] deals with set membership detection for queries with respect to a single set, we deal with complete approximate de-duplication (ADRR) of massive number of records/objects in the database. Further, we consider the more general problem of discovery of approximate Range Motifs (ARM) or clusters of close points, develop asymptotic upper bounds of FPR and FNR for r sets (clusters) and also demonstrate scalable performance over multi-core architectures. For the ARM problem, we assume that the maximum number of underlying sets in the database is a small constant relative to the number of records/objects (n) in the database.

Further, in order to achieve high throughput (1 - 10 *GB/s*) approximate Range Motif discovery (ARM) and approximate data redundancy removal (ADRR), one needs an efficient parallel algorithm with scalable performance. We present the design of a parallel pipelined multi-threaded algorithm based on Bloom filters. Typical parallel Bloom filter approaches incur k cache-misses with every record, where k is the number of hash functions computed per record to check the bits of the Bloom filter array. This leads to poor cache performance. Thread pipeline throughput is a critical issue in parallel Bloom filter design, which if not addressed, can lead to lower overall performance. Further, there is a trade-off between the cache performance and memory efficiency [29] in the Bloom filter design. These issues make parallel real-time Range Motif discovery and approximate data redundancy removal (de-duplication) over massive datasets a very challenging problem.

In order to achieve high throughput and real-time performance, we optimized our parallel ARM (ADRR) algorithm on parallel multi-core architectures. Emerging and next generation many-core architectures have massive number of hardware threads and multiple levels of cache hierarchy. In order to provide real-time ARM (ADRR), we consider cache performance at multiple levels of cache hierarchy as well as thread pipeline throughput with increasing number of threads. The parallel algorithm makes appropriate trade-offs between cache locality and memory efficiency. We demonstrate scalable performance on $10M$ records using multi-core Intel Xeon 5570 architecture with 16 cores and $64GB$ memory. This paper makes the following contributions:

- We present a novel sequential algorithm for in-memory real-time approximate Range Motif discovery (ARM) and approximate data redundancy removal (ADRR) over massive datasets (10 million records). We prove asymptotic bounds on the false positive rate and the false negative rate for ARM and hence establish theoretical soundness of our algorithmic approach.
- We present parallel pipelined algorithm for in-memory real-time approximate Range Motif discovery (ARM) and approximate data redundancy removal (ADRR). Asymptotic parallel time complexity analysis proves scalable performance of our parallel algorithm.
- We demonstrate real-time in-memory parallel ARM & ADRR on 16-core Intel Xeon 5570 multi-core architectures. Our in-memory algorithm, delivers a throughput of $170K$ records per second (around 700 *Mb/s* for 512 byte records) for approximate de-duplication and Range Motif discovery (using 4 sets) over $10M$ records (with negligible false positive and false negative rates). We also study the impact of variation of parameters on false positive rate and false negative rate. To the best of our knowledge, this is the best known throughput for approximate Range Motif discovery and approximate data redundancy removal for such large number of records.

2. PRELIMINARIES & BACKGROUND

A Bloom filter is a space-efficient probabilistic data structure that is widely used for testing membership queries on a set [4]. The efficiency is achieved at the expense of a small false positive rate, where the Bloom filter may report falsely the presence of an element (record) in the set. However, it does not report false negatives i.e. elements which are actually present in the set are always recognized as being present. Representing a set of n elements by a Bloom filter requires an array of m bits, initially all set to 0. To insert an element β into the filter, one requires to set k bits (locations) in the Bloom filter array (BFA) denoted here by *INSERT*. These k locations are obtained by the evaluation of k independent hash functions $h_1(\beta), \ldots, h_k(\beta)$. We denote this indexing operation by *HASH*. If all the locations are already set to 1, then either the element β is already a member of the set or a false positive. The probability of the false positive rate [6] for a standard Bloom filter is:

$$FPR \approx \left(1 - e^{-kn/m}\right)^k \tag{2.1}$$

Given n and m the optimal number of hash functions $k = \ln 2 \cdot (m/n)$.

DEFINITION 1. *A family $H = h : U \to V$ is (r_1, r_2, p_1, p_2)-sensitive with respect to a metric space (U, d) if $r_1 < r_2$, $p_1 > p_2$, and for any $x, y \in U$,*

- *if $d(x, y) \le r_1$ then $Pr_{h\leftarrow H}(h(x) = h(y)) \ge p_1$, and*
- *if $d(x, y) \ge r_2$ then $Pr_{h\leftarrow H}(h(x) = h(y)) \le p_2$.*

We say that any such family is a (U, d)-locality-sensitive hash (LSH) family, omitting (U, d) when the meaning is clear.

This definition can be generalized by the following distance-sensitive hash family.

DEFINITION 2. *Let (U, d) be a metric space, and let $p_L : R_{\geq 0} \rightarrow [1, 0]$ and $p_H : R_{\geq 0} \rightarrow [1, 0]$ be non-increasing. A hash family $H : U \rightarrow V$ is called (p_L, p_H)-distance sensitive (with respect to (U, d)) if for all $x, y \in U$*

$$p_L(d(x, y)) \leq Pr_{h \leftarrow H}(h(x) = h(y)) \leq p_H(d(x, y)).$$

This generalizes Definition 1, which can be obtained as follows: Set $p_L = p_1$, if $r < r_1$ and 0 otherwise; and set $p_H = p_2$, if $r > r_2$ and 1 otherwise.

In order to determine whether an element u is close to any element x of the set S, a distance-sensitive Bloom filter [21] can be used. To answer the membership query for u to a given set S, the distance sensitive Bloom filter is constructed in the following fashion. Let $H : U \rightarrow V$ be a (p_L, p_H) distance sensitive hash function, fix some $S \subset U$ with n elements, and let A be an array consisting of k disjoint m'-bit arrays, $A[1, 1], ..., A[k, m']$ (for a total of $m = km'$ bits) where k and m' are parameters chosen heuristically for low false-positive and false-negative error rates. To initialize the filter, the hash functions: $h_1, h_2, ..., h_k \leftarrow H$ are chosen independently. Then, all bits in A are set to zero. Finally, for $x \in S$ and $i \in [k]$, $A[i, g_i(x)] = 1$. In order to answer the query, whether u is close to any $x \in S$, one checks the number ($B(u)$) of u's hash locations that are set to 1. It can be seen that $B(u) = sum_{i \in [k]} A[i, g_i(u)]$. Since, $A[i, g_i(u)]$ are independent and identically distributed bits, $B(u) \sim Bin(k, q_u)$ for some $q_u \in [0, 1]$ ($Bin(t, r)$ denotes the binomial distribution with t trials and common success probability r). It can be shown that [21] for any $u \in U$, $B(u)$ lies in the range:

$$\begin{aligned} &B(u) \geq_{st} Bin(k, p_L(d(u, S))) \\ &B(u) \leq_{st} Bin(k, sum_{x \in S} p_H(d(x, u)) + \frac{nk}{m} \cdot 1(V \neq [m'])) \end{aligned} \tag{2.2}$$

The Range Motif [28] with range r is the maximal set of points that have the property that the maximum distance between them is less than $2r$. Formally, for points in database, D, a Range Motif is described as follows.

DEFINITION 3. *S is a Range Motif with range r iff*
$\forall P_x, P_y \in S,\ dist(P_x, P_y) \leq 2r,$
and $\forall P_d \in (D - S) dist(T_d, T_y) > 2r.$

Range Motifs correspond to dense regions or high dimensional *bumps* in the space of the points considered.

3. RELATED WORK

Data deduplication has been studied extensively in the context of storage systems. Here data deduplication was primarily used to improve the compression ratio rather than achieving high throughput. Storage systems used file-level hashing to detect duplicate files [1, 32, 12], but such approaches achieved a low compression ratio. Secure hashes were used by Venti [30] to remove duplicate fixed-size data blocks. This work used a disk-based hash table divided into buckets where a hash function is used to map chunk hashes to appropriate buckets. However, it used an on-disk index cache which had no locality and thus ended up with a throughput of less than 7 MB/sec. Dividing a file into content-based data segments and using such segments for deduplication was shown by [5]. The TAPER system [19] was the first to use Bloom Filters to detect duplicates instead of detecting a new segment. However these studies did not investigate deduplication throughput issues. [34] addresses the issue of throughput in deduplication systems. They used a combination of techniques such as *Summary Vector* (Bloom Filter) and *Locality Preserved Caching* to reduce disk index lookups and simultaneously to obtain throughput values over 210 MB/s. Recently, [23] uses a combination of content-based chunking and *sparse-indexing* to solve large scale disk based deduplication. They also claim to use less RAM compared to all other existing approaches but don't focus on throughput issues. In this paper, we consider Range Motif discovery and appoximate de-duplication algorithm design to achieve low false positive and false negative rates. Further, we performed optimizations to deliver real-time throughput and scalable performance.

Bloom filters have been proposed for various purposes. These include, counting Bloom filters [13], compressed Bloom filters [27], hierarchical Bloom filters [35], space-code Bloom filters [22] and spectral Bloom filters [31]. Counting Bloom filters replace an array of bits with counters in order to count the number of items hashed to a particular location. We design and analyze a variation of the counting Bloom filter for approximate Range Motif discovery and optimize its throughput.

Bloom filters have been broadly applied to network-related applications. Bloom filters are used to find heavy flows for stochastic fair blue queue management scheme [14]. Bloom filters provide a useful tool to assist network routing, such as packet classification [2], per-flow state management and the longest prefix matching [11]. [21] presents the design of distance-sensitive Bloom filters using LSH functions. It addresses the problem of solving membership queries to a *single set*, using *threshold* based decision for deciding closeness. We present a novel sequential algorithm that uses *maximum* count based decision, in Bloom filter based data structure with LSH functions, to solve a more generic problem of real-time Range Motif discovery and approximate de-duplication over the complete dataset. While [21] presents results for only $N = 10K$ records, we demonstrate real-time (single pass over the data) scalable performance along with extremely low FPR/FNR for $N = 10M$ records. We also prove theoretical upper bounds on the false positive and false negative rates for the Range Motif discovery problem. [9] deals with streaming data (limited buffer space and infinite stream of data), where it is necessary to remove stale state using a novel data structure referred to as *Stable Bloom Filter* (SBF). It considers matching exact strings such as *apple* with itself, though approximately with FPR/FNR because SBF with limited buffer space is used. We consider approximate matching of strings such as *apple* with *apxle* which are close strings. For some applications in data mining this is what is needed instead of looking for exactly the same string. Due to this, we have to use LSH and further we use bloom filter to make it fast and hence also do this work approximately because we get finite but small FPR/FNR due to LSH function usage with Bloom filters. We also consider the general version of ADRR problem i.e. Approximate Motif Discovery (ARM) (see Introduction section) which is not considered by [9]. We also demonstrate the low FPR/FNR achieved by our algorithm and their variation with the parameters such as the size of the bloom filter used and the number of hash functions used. [16] considers the problem of detecting duplicates in a single pass over a data stream

of length n over an alphabet $[m]$ where $n > m$. It provides the first randomized algorithm for this problem that uses sub-linear ($O((\log m)^3)$) space. This algorithm also solves the more general problem of finding a positive frequency element in a stream given by frequency updates where the sum of all frequencies is positive.

[10] presents the design of parallel bloom filters, implemented in hardware, to match patterns in network packet payload. It demonstrates matching $10,000$ strings on the network data at a line speed of 2.4 *Gbps* using state-of-the-art FPGAs. We perform real-time approximate Range Motif discovery and de-duplication on large datasets ($10M$ and higher) using a novel parallel bloom filter algorithm, with performance optimizations for multi-core architectures. [8] proposes a new design of Bloom filter in which every two memory addresses are squeezed into one I/O block of the main memory. With the burst-type data I/O capability in the contemporary DRAM design, the total number of memory I/Os involved in the membership query is reduced to half. This leads to a reduction in query delay and an improvement in overall performance, with a small increment in the false positive rate. Our in-memory algorithm design supports any desirable false positive rate and false negative rate while providing scalable performance.

[18] proposes a new Bloom filter structure that can support the representation of items with multiple attributes and allow the false positive probability of the membership queries at a very low level. The new structure is composed of parallel Bloom filters and a hash table to support an accurate and efficient representation for querying of items. [17] extends *Bloomjoin*, the state-of-the-art algorithm for distributed joins, in order to minimize the network usage for the query execution based on database statistics. [24] discusses how Bloom filters can be used to speed up name to location resolution process in large scale distributed systems. The approach presented offers trade-offs between performance (the time taken to resolve an object's name to its location) and resource utilization (the amount of physical memory to store location information and the number of messages exchanged to obtain the object's address). Our parallel bloom filter design for multi-core architectures can be leveraged to accelerate these network processing applications that use bloom filters.

4. APPROXIMATE RANGE MOTIF DISCOVERY ALGORITHM

The approximate Range Motif discovery and data redundancy removal algorithm uses a novel data structure. The data structure consists of k BFAs (Bloom Filter Arrays), where k is the number of locality sensitive hash functions used by the algorithm. Each BFA$[q]$ ($q \in [1..k]$) is an array of structures. Each structure consists of the following fields:

- *Bloom Bit*: One bit that represents whether any prior seen record in the input set that bit. This bit is set to 1 in case any prior record set it, else it has value 0.

- *Set Membership Array*: Array of r bits that represents the membership of records to the k sets (Range Motifs) over which the input data needs to be partitioned. When an input record hashes (using q^{th} LSH, $q \in [1..k]$) onto a particular bit location, say i, in the BFA$[q]$, and it is determined that this record belongs to set, j, then the bit j in the Set Membership Array at BFA$[q][i]$, is set to 1.

Table 1: Notation for Time Complexity

Symbol	Definition
N	Total number of input records
D	Number of records per batch
T_p	Number of pre-processing threads
T_f	Number of front-end threads
T_b	Number of back-end threads
k	Number of locality sensitive hash functions
X	Size (length) of a record
X'	Size of each input to secondary hash functions
l'	Number of bits for the output of one LSH function
$m' = 2^{l'}$	Size of each partition of the Bloom Filter array
$m = km'$	Size of the total Bloom Filter array
r	Number of sets / Range Motifs in the input data
c	Collision array size (pre-processing (PP) phase)
K_e	Number of hash indices in PP phase

The sequential algorithm for hashing the input records and assigning them Range Motifs consists of two main modules: *Frontend Module* and *Backend Module*. Each input record goes through both the Frontend and the Backend Modules in sequence. In the Frontend module, k locality sensitive hash functions are computed for each record. For the qth ($q \in [1..k]$) locality sensitive hash function computation, l' bits are chosen from locations independently and randomly selected from the input record. Here, the size of each BFA, m' equals $2^{l'}$. The l' bits thus obtained are concatenated to form an integer, $i = h_q$. This integer is used to index into BFA$[q][i]$.

In the Backend Module, we decide to which this input record belongs, whether an existing populated set or a new set. Here, for each of the k locations across the k BFAs, we count the number of $1s$ in each set. The set that has maximum number of $1s$ is referred to as the *winning set*. In case there is a clear single winning set and the number of $1s$ is not small, then the input record is assigned to that set and the corresponding bits in the Set Membership Arrays for that set are set to 1. In case there are multiple winner sets but the number of $1s$ is small, then a new set is populated, and its correspondng locations in the Set Membership Arrays are set to 1. The last case, in which there are multiple winner sets, and the number of $1s$ is not small, then we assign that record to any of these winner sets randomly. We could also (though not necessary) use an additional threshold as an input data dependent parameter, to generate more accurate Range Motifs.

4.1 Asymptotic Bounds on FPR & FNR

In this section, we present the asymptotic bounds on the false positive rate and the false negative rate achieved by our approximate Range Motif discovery algorithm. The notation used is given in Table 1.

The locality sensitive hash function H (section 2) is a (p_L, p_H)-distance sensitive hash function for $p_L(z) = p_H(z) = (1-z)^{l'}$.

THEOREM 4.1. *The hash function, H, for any fixed $u \in U$, achieves the following bounds on the false positive rate and the false negative rate.*

(a) *False Positive Rate(S^*),* $\leq e^{-((l'\epsilon - l'\delta)^2 k/2}$ *if,* $\forall j \in [0..(r-2)]$ $d(u, S_j) \geq \delta$, *and,* $d(u, S_{r-1}) \leq \epsilon$.

(b) *False Positive Rate(S^*)* $\leq \prod_{j=[0..(r-1)]} e^{-(l'\epsilon - l'z_j)^2 k/2}$ *if,* $d(u, S_j) = z_j$ *(general case)*

(c) *False Negative Rate(S^*)* $\leq (r-1) * e^{-(l'\epsilon - l'\delta)^2 k/2}$

PROOF. The False Positive Rate is the probability that an element $u \in U$ is not close to any element $x \in S_0$, i.e. $d((u, S_0) \geq \delta$, but, $\forall j \in [1..r-1] B(u, S_0) > B(u, S_j)$.
First consider the case (a), where u is close to one Set, w.l.o.g. say $S_{(r-1)}$, but u is far from other set, i.e.
$\forall j \in [0..(r-2)]\ d(u, S_j) \geq \delta$, and,
$d(u, S_{r-1}) \leq \epsilon$.

Let, E_j denote the even that $B(u, S_0) > B(u, S_j)$. The FPR is given by the following equation.

$$\begin{aligned} FPR &= Prob(B(u,S_0) \text{ is maximum} \mid d(u,S_0) \geq \delta) \\ &= Prob(E_1 \cap E_2 \cap ... \cap E_{(r-1)})) \\ &= \prod_{j=1..(r-1)} Prob(E_j) \quad \because B(u,S_j) \text{are all independent} \\ &= Prob(E_{(r-1)}) * \prod_{j=1..(r-2)} Prob(E_j)) \\ &\leq e^{(l'\epsilon - l'\delta)^2 k/2} * 1 \end{aligned} \tag{4.1}$$

The last inequality follows from Hoeffding inequality. To determine the bound on the difference between two Binomial variables, $Y \sim Bin(k, q)$ and $X \sim Bin(k, p)$, we express it as the sum of k differences between bernoulli random variables, $B_p(i) - B_q(i)$.

$$\begin{aligned} Prob((Y - X) \geq 0) &= Prob((Y - X + k(p-q)) \geq k(p-q)) \\ &\leq e^{-2k^2(p-q)^2/4k} \\ &\leq e^{-(p-q)^2 k/2} \end{aligned} \tag{4.2}$$

Using the above Hoeffding equality, we get:

$$\begin{aligned} &\forall j \in [0..r-2] Prob(B(u,S_0) > B(u,S_j)) \\ &\qquad \leq e^{-(p_H - p_H)^2 k/2} \leq 1 \end{aligned} \tag{4.3}$$

Further, we get,

$$\begin{aligned} &Prob(B(u,S_0) > B(u,S_(r-1))) \\ &\quad \leq e^{-(p_L - p_H)^2 k/2} \\ &\quad \leq e^{-((1-\epsilon)^{l'} - (1-\delta)^{l'})^2 k/2} \\ &\quad \leq e^{-(l'\epsilon - l'\delta)^2 k/2} \end{aligned} \tag{4.4}$$

Next, consider the general case, when, $d(u, S_j) = z_j \in [\epsilon, \delta]$. In this case, one can see that the False Positive Rate becomes:

$$FPR \leq \prod_{j=1..(r-1)} e^{(l'z_j - l'\epsilon)^2 k/2} \tag{4.5}$$

False Negative rate for a Set, (w.l.o.g.) say S_0, is given by:

$$\begin{aligned} FNR(S_0) &= Prob(B(u,S_0) \text{is not maximum}) \\ &= Prob(\exists j \in [0..(r-1)] : B(u,S_0) \leq B(u,S_j)) \end{aligned} \tag{4.6}$$

Now, let E_j denote the event that $(B(u, S_0) \geq B(u, S_j))$. So,

$$\begin{aligned} FNR(S_0) &= 1 - Prob(B(u,S_0) \text{is maximum}) \\ &= 1 - Prob(E_1 \cap E_2 \cap ... \cap E_(r-1)) \\ &= 1 - Prob(E_1) * Prob(E_2) * Prob(E_(r-1)) \end{aligned} \tag{4.7}$$

The last equality follows since E_j are mutually independent. Now,

$$\begin{aligned} Prob(B(u,S_0) \geq B(u,S_j)) &= 1 - Prob(B(u,S_0) < B(u,S_j)) \\ &= 1 - e^{-(l'\epsilon - l'\delta)^2 k/2} \\ &\quad (\because \text{using Hoeffding bound}) \end{aligned} \tag{4.8}$$

Using this in equation (4.11), we get,

$$\begin{aligned} &Prob(B(u,S_0) \text{is not maximum}) = \\ &\qquad 1 - \prod_{j\in[1..(r-1)]} (1 - e^{-(l'\epsilon - l'\delta)^2 k/2}) \\ &\qquad = 1 - (1 - e^{-(l'\epsilon - l'\delta)^2 k/2})^{(r-1)} \end{aligned} \tag{4.9}$$

Let, $\alpha = e^{-(l'\epsilon - l'\delta)^2 k/2}$, then,

$$Prob(B(u,S_0) \text{is not maximum}) \approx (r-1) * \alpha \tag{4.10}$$

In the above equation, we used the approximation:
$(1-\alpha)^{(r-1)} \sim 1 - (r-1) * \alpha + \Omega(\alpha^2)$..

Hence, we get that the false negative rate for set, S_0 is given by:

$$FNR(S_0) \leq (r-1) * e^{-(l'\epsilon - l'\delta)^2 k/2} \tag{4.11}$$

One can intuitively understand and also verify with equation (4.11) that the false negative rate decreases with:

- Increase in k, the number of locality sensitive hash functions used,
- Increase in difference between ϵ and δ, which represents how separated the sets (range motifs) are in the underlying data.
- Decrease in the number of sets.
- Increase in the size of the sub-arrays of the Bloom filter, $m' = 2^{l'}$.

□

5. PARALLEL BLOOM FILTER BASED ALGORITHM

We present here the design of our parallel Bloom filter (PBF) for approximate Range Motif discovery over r-sets. For the purpose of modularity, we process batches of size D instead of processing the N elements, as a whole. The PBF has three modules (phases) namely Pre-Processing (PP) module (phase), Front-End (FE) module (phase) and Back-End (BE) module (phase). We assign T_p, T_f and T_b threads to each of these modules respectively. These phases

work in a pipelined fashion to deliver high performance on multi-threaded multi-core architectures.

5.1 Pre-Processing Module

The PP module performs the task of detecting duplicates within the batch being currently processed, by using hash tables in parallel. To begin with, the PP module assigns to each record in the batch, a unique identification number (UID) by using a cryptographic hash function (such as MD5). This assignment is used to avoid costly compare operations due to large size of the records. Next, the batch is partitioned into T_p parts, each containing around (D/T_p) records ([3]). This record based partitioning strategy (Fig. 1) gives equal processing load to each of the T_p threads. Each of the T_p threads, computes a hash function on the records assigned to it and stores them in the hash table associated with it. Thereafter, hash index based partitioning is used to merge the hash tables generated by each thread (Fig. 1). During merge, the duplicate records are discarded. Thus, at the end of the PP module, the batch being processed consists of only unique records (within itself).

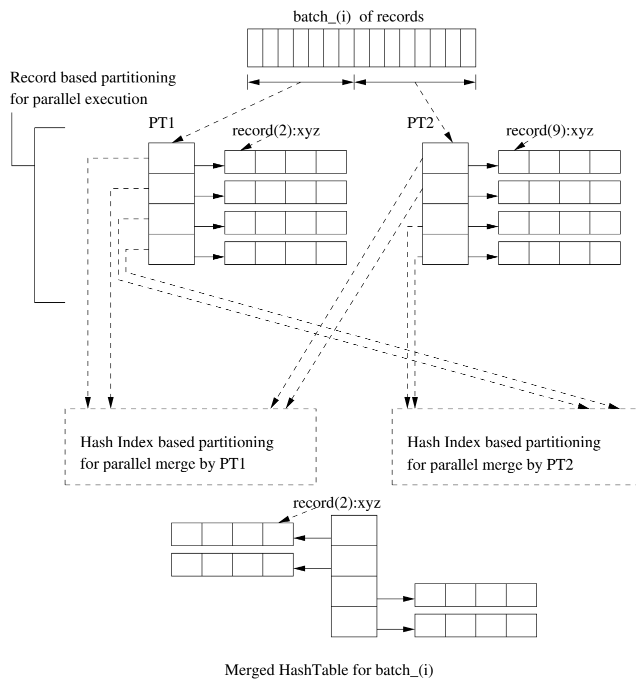

Figure 1: Parallel Hash Table Merge ($\alpha = 2$)

5.2 Frontend and Backend Modules

5.2.1 *Frontend Module*

The FE module is responsible for computing the k independent locality sensitive hash (LSH) functions for each record and storing them in the queue assigned for each record. (Fig. 2). The LSH is computed by selecting l' bits of the input record at random and then concatenating them to form the resulting integer. Each of these k hash functions is used to access the k separate bloom filter sub-arrays *BFA*).

5.2.2 *Backend Module*

The k BFA's are assigned to T_b backend threads, such that each thread gets roughly equal number of BFAs. Here, each thread determines locally the count of the number of 1's in each of the r-sets. Thereafter, a reduction operation is performed to determine the global count of the number of 1's across all the sets. This is achieved by either (a) by equally assigning r sets to T_b threads and each backend thread determines the global count of the number of the number of $1s$ for each set assigned to it; or, (b) by parallel reduction across the threads where each thread looks at all sets. The choice of (a) vs. (b) depends upon whether $r > T_b$ or otherwise to deliver scalable performance in the backend phase. Finally, the set with the highest count is declared as the winner set. Ties are broken randomly. One can also use a threshold, t, to make sure that in the initial phases of input data read, new sets get created. Here, if there are multiple winner sets and all of their number of 1's count is less that the threshold (t), then the record might belong to a new set, hence a new set is created. The bit representation for set membership per bit of the BFAs, is efficient so it can model moderate number of sets, though it is assumed that the number of sets, r, is much smaller than the total number of records, N, in the input dataset. Finally, all unset bits corresponding to corresponding k locations in the winner set are set in parallel by the T_b threads.

The process of assignment of a record to a set proceeds sequentially, though the intermediate steps from hash function generation till the determination of the winner set and setting the bits in that set proceeds in parallel as described above. This entire process helps in discovering Range Motifs over r-sets. When $r = 1$, this problem reduces to that of approximate de-duplication over the complete dataset.

We now explain why the PP module performs a local de-duplication operation (within the records of each batch). Consider the set: $\{a, b, d, b, f, h\}$, where the index of the records in the set starts from 0. The record b at position 3 (denoted by b_3) in this set is a duplicate of b_1. If the number of FE threads, $T_f \geq 2$, then the thread T_{p_0} computes k hashes of $\{a, b, d\}$ and T_{p_1} does so for $\{b, f, h\}$. If the order of k accesses by the two threads is interleaved, then they might not be able to see the duplicate presence of the record b. This might lead to false negative results, which is an undesirable outcome. Also, if the computation for b_3 is completed and enqueued before that of b_1, then the BE thread responsible for record b will report b_1 as the duplicate instead of b_3. This might not be desirable for some application domains, where all but the first occurrence of a record need to be marked as duplicate and discarded. In order to take care of both these issues, we need to have local de-duplication (within the records of each batch) in the PP module. However, we use this PP module whenever we attempt to assign records to different sets in parallel.

The PP module does the pre-processing operation independent of the remaining modules. After it completes operation on a batch, it signals the corresponding FE module to start the HASH operation. As soon as, the FE module starts filling up the queues, it sends a signal to the BE module to start the INSERT operations. However, the FE module waits for the BE module to complete before it takes up the next batch. This pipelined parallelism results in high throughput and scalable performance of our parallel ADRR algorithm.

5.3 Asymptotic Parallel Time Complexity

This section presents asmyptotic analysis of the time complexity of the parallel ADRR algorithm. For each of the three phases: (a) Pre-Processing phase, (b) Frontend phase and (c) Backend phase, the parallel time complexity is analyzed below.

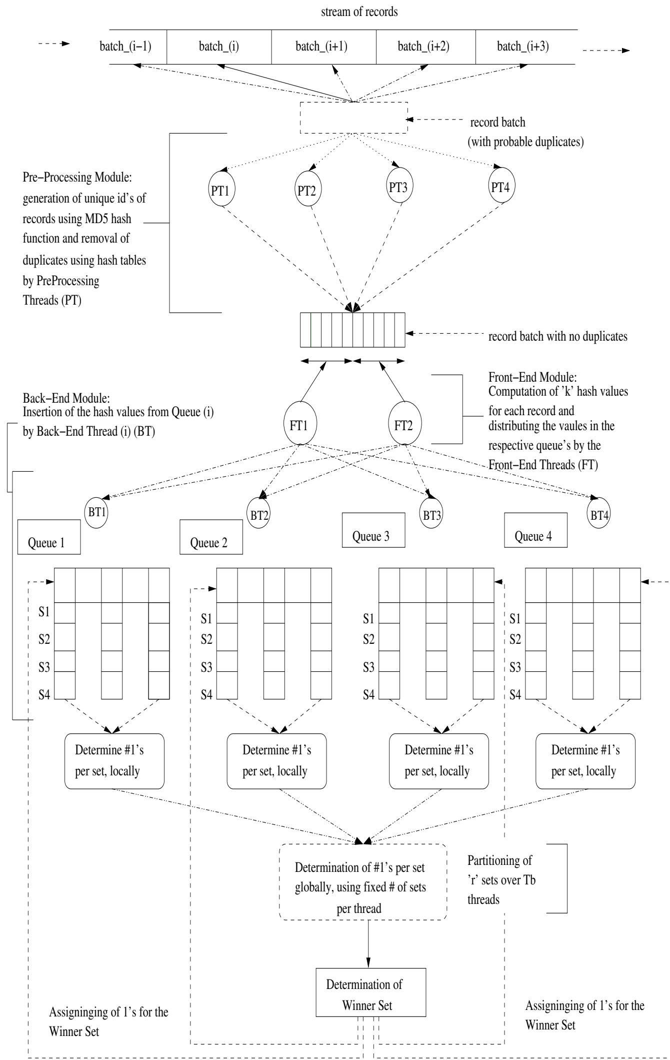

Figure 2: Parallel Bloom Filter Design ($T_p = 4$, $T_f = 2$, $T_b = 4$)

5.4 Pre-Processing Phase

In the PP (Pre-processing) phase, the MD5 computation requires overall $O(N \cdot X)$ work. Using T_p threads, the time taken for the MD5 computations over the complete dataset is $O(N \cdot X/T_p)$. Further, every batch (or block) of D records (each of size X'), is processed by T_p threads/cores, and checked for existence of exact duplicates with that batch (block). Here, first the records/objects within the block are partitioned equally to each thread/core for insertion into a hash-table per core. The time for insertion into the hash-table per thread/core is $O(D \cdot X'/T_p)$. Then, using hash key based partitioning, each thread/core is assigned equal number of keys in the hash-table. Each thread uses the same hash function and key space and we assume that the records/objectsin each thread get evenly distributed across the keys. Now, each thread gtes same number of keys $= K_e/T_p$. Each thread looks at all the T_p hash tables and goes over all the keys assigned to it in a sequential fashion and checks for any duplicate records. hence, the time taken per thread is $O(clog(c) \cdot K_e/T_p + c \cdot K_e + D/T_p)$. Thus, the total time for the Pre-processing phase is $O(NX/T_p + (N/D) * [clog(c) \cdot K_e/T_p + cK_e])$.

5.5 Frontend Phase

The first step in the FE (Frontend) phase consists of computing k Bloom filter locations for each record. Each of the k Bloom filter locations is computed using using l' bits randomly chosen from the record. Since the input records are equally partitioned across T_f threads, therefore, the time complexity per block is $O\left(\frac{D \cdot k \cdot l'}{T_f}\right)$. Hence, the overall complexity of the FE phase is $O\left(\frac{kNl'}{T_f}\right)$.

5.6 Backend Phase

In the BE (Backend) phase, for each block the following operations happen sequentially per record. First, each thread works on k/T_b sub-arrays and determines the number of $1s$ seen per set. Here, the time taken by each thread is $O(r \cdot k/T_b)$. Then, the threads synchronize and start reducing the number of $1s$ for each set, with each working on (r/T_b) sets. Since, there are (k/T_b) values to reduce. The time taken by each thread here is $O(rk/T_b^2)$. Finally, after determination of the winner set, each thread works on (k/T_b) sub-arrays and assigns 1 to appropriate locations in the sub-arrays. The time taken per thread is $O(k/T_b)$. Hence, the total time taken for all records in a block is $O(D(rk/T_b + rk/T_b^2))$, which makes the total time for Backend phase over the complete dataset as $O(N * rk/T_b)$.

Since the three phases work in a pipelined fashion, the overall time of our parallel ADRR algorithm is bounded by the maximum of time taken by PP, FE and BE phases. Hence, the overall execution time for the parallel ADRR ($(_{ADRR}$)algorithm is:

$$\begin{aligned} T_{ADRR} = \max\{&Nrk/T_b, \\ &Nkl'/T_f, \\ &(NX/T_p + N/D * (clog(c)K_e/T_p + cK_e))\} \end{aligned} \tag{5.1}$$

This model can be used to distribute the threads across the pipeline phases to maximize ADRR throughput, by appropriately considering the constants in the equation, on various multi-core architectures. The constants would reflect the impact of multi-level cache hierarchy and memory bandwidth on multi-core performance.

6. RESULTS & ANALYSIS

We implemented our parallel ARM / ADRR algorithm using Posix Threads / NPTL(Native Posix Thread Library) API. Random test data, with variable number of records and record sizes, was used to evaluate the performance and scalability of our algorithm. The experiments were performed on 16-core Intel architecture, with four Quad-core Xeon 5570 chips. Each core has a private L1 instruction and a L1 data cache of size 32KB and a private 256KB L2 cache. Four cores share 8MB L3 cache. The affinity of each thread was set to a different core using the thread affinity API in NPTL. We use the following notation in this section. X represents the size of a record in bytes; D represents the size of a batch in number of records (per batch in in-memory execution); N represents the total number of records over which to perform de-duplication; N_0: represents the total number of records used to determine the size of the bloom filter array. Note, $N \leq N_0$ to maintain the false positive rate (FPR); T: represents the total number of threads used.

6.1 FPR/FNR Analysis

In this section, we study the impact of variation of the parameters l' ($m' = 2^{l'}$ is the size of each sub-array of the Bloom filter) and k

(number of hash functions) on the total FPR and FNR, as well as on the maximum FPR over all sets and maximum FNR over all sets. In the first configuration, (a), we chose $N = 1M$ records, $r = 4$ sets, $\delta = 0.25$, $\epsilon = 0.1$, $l = 512$ bits and threshold $t = \lceil k/2 \rceil$. In the second configuration, (b), we chose $N = 1M$ records, $r = 8$ sets, $\delta = 0.125$, $\epsilon = 0.05$, $l = 512$ bits and threshold $t = \lceil k/2 \rceil$.

6.1.1 FPR / FNR variation with l'

Fig. 3 presents (for configuration (a)) the impact of variation in l' on the total FPR across all 4 sets and the maximum and minimum FPR over all sets. Here, k is kept constant as 8 locality sensitive hash functions. The total FPR decreases exponentially with increase in l'. As l' (respectively $m' = 2^{l'}$ bits) varies from 12 ($m' = 4096$ bits) to 19 ($m' = 512K$ bits), the FPR total decreases from 19% to 0. The maximum FPR per set over all the 4 sets decreases from 15% to 0. The minimum FPR per set, over all sets, remains low for all sets with a small increase in between for $l' = 15$. This can be intuitively explained by the fact that as the number of bits in the Bloom filter ($m = m'k$) increases, the system will consume more memory but the error rate should go down as more bits are used to represent each set. This exponential decrease of FPR with increase in l' also agrees with the theoretical bound on FPR given by Theorem 4.1.

Fig. 4 presents (for configuration (a)) the impact of variation in l' on the total FNR across all 4 sets and the maximum and minimum FNR over all sets. Here, k is kept constant as 8 locality sensitive hash functions. The total FNR decreases exponentially with increase in l'. As l' (respectively $m' = 2^{l'}$ bits) varies from 12 ($m' = 4096$ bits) to 19 ($m' = 512K$ bits), the FNR total decreases from 19.48% to 0. The maximum FNR per set over all the 4 sets decreases from 19% to 0. The minimum FNR per set, over all sets, remains low for all sets. This exponential decrease of FNR with increase in l' also agrees with the theoretical bound on FNR given by Theorem 4.1. As explained above, with increase in Bloom filter size, the memory consumed increases, but the error rate also goes down due to more bits that represent each set.

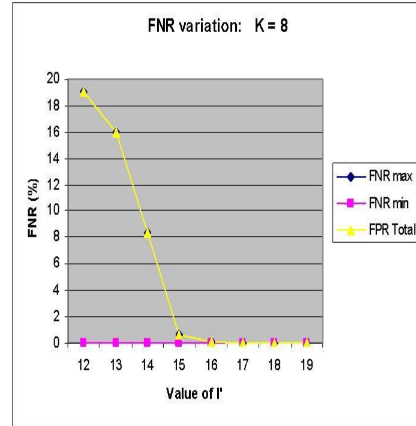

Figure 4: FNR Variation with l' (4 sets)

Fig. 5 presents (for configuration (b)) the impact of variation in l' on the total FPR across all $r = 8$ sets and the maximum and minimum FPR over all sets. Here, k is kept constant as 12 locality sensitive hash functions. The total FPR decreases exponentially with increase in l'. As l' (respectively $m' = 2^{l'}$ bits) varies from 16 ($m' = 4096$ bits) to 26 ($m' = 64M$ bits), the FPR total decreases from 25% to 0. The maximum FPR per set over all the 8 sets decreases from 25% to 0. The minimum FPR per set, over all sets, remains low for all sets. This can again be intuitively explained by the fact that as the number of bits in the Bloom filter ($m = m'k$) increases, the false positive rate goes down as more bits are used to represent each set. This exponential decrease of FPR with increase in l' also agrees with the theoretical bound on FPR given by Theorem 4.1.

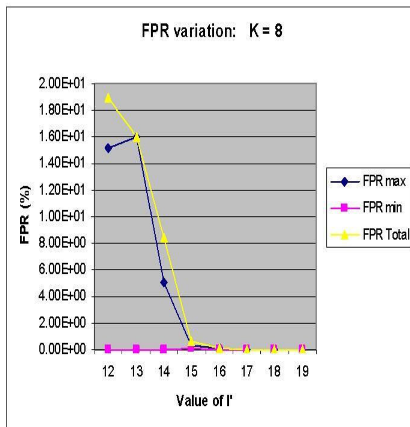

Figure 3: FPR Variation with l' (4 sets)

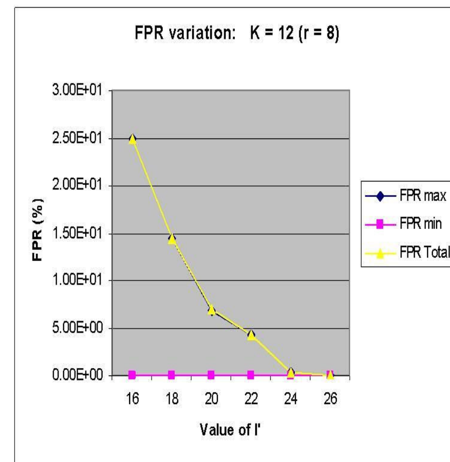

Figure 5: FPR Variation with l' (8 sets)

Fig. 6 presents (for configuration (b)) the impact of variation in l' on the total FNR across all $r = 8$ sets and the maximum and minimum FNR over all sets. Here, k is kept constant as 12 locality sensitive hash functions. The total FNR decreases exponentially with increase in l'. As l' (respectively $m' = 2^{l'}$ bits) varies from 16 ($m' = 4096$ bits) to 26 ($m' = 64M$ bits), the FNR total decreases from 25% to 0. The maximum FNR per set over all the 8 sets decreases from 11.4% to 0. The minimum FNR per set, over all sets, remains low for all sets. This exponential decrease of FNR with increase in l' also agrees with the theoretical bound on FNR given by Theorem 4.1.

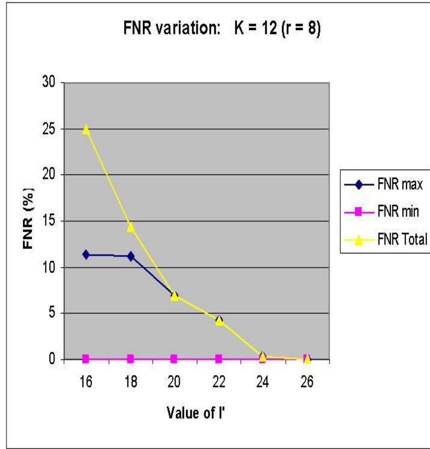

Figure 6: FNR Variation with l' (8 sets)

6.1.2 FPR / FNR variation with K

Fig. 7 presents the impact (for configuration (a)) of variation in k (number of LSH functions) on the total FPR across all 4 sets and the maximum and minimum FPR over all sets. Here, l' is kept constant as 12, i.e. $m' = 2^{l'} = 4096$ bits. The total FPR decreases exponentially with increase in k. As k varies from 12 LSH functions to 24 LSH functions, the total FPR decreases from 10% to 0. The maximum FPR per set over all the 4 sets decreases from 4.8% to 0. The minimum FPR per set, over all sets, remains low for all sets with a small increase in between for $k = 14$. This can be intuitively explained by the fact that as the number of LSH functions used increases, the error rate should go down as more hash functions are used to determine the membership of an element to a set. This exponential decrease of FPR with increase in k also agrees with the theoretical bound on FPR given by Theorem 4.1. Note that the decrease of FPR with k is slower than w.r.t. l' with k constant, since l' term comes with exponent of 2, while k comes with exponent of 1.

Fig. 8 presents the impact (for configuration (a)) of variation in k on the total FNR across all 4 sets and the maximum and minimum FNR over all sets. Here, l' is kept constant as 12. The total FNR decreases exponentially with increase in k. As k varies from 12 to 24 LSH functions, the total FNR decreases from 10.5% to 0. The maximum FNR per set over all the 4 sets decreases from 10% to 0. The minimum FNR per set, over all sets, remains low for all sets. This exponential decrease of FNR with increase in k also agrees with the theoretical bound on FNR given by Theorem 4.1. As explained above, with increase in the number of hash functions, the memory consumed increases, and the error rate also goes down as more LSH functions are used to determine the membership per set.

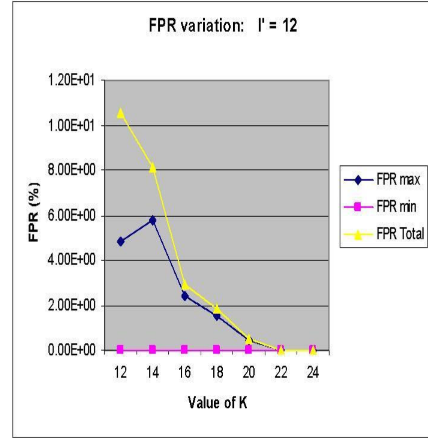

Figure 7: FPR Variation with K (4 sets)

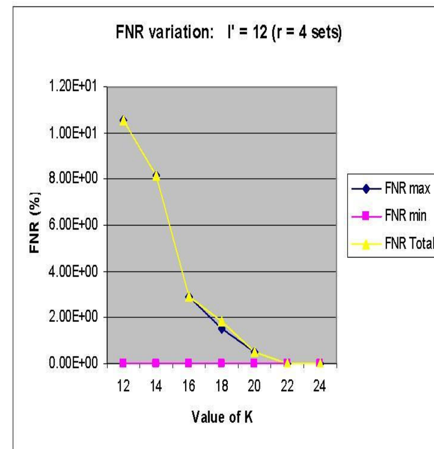

Figure 8: FNR Variation with K (4 sets)

Fig. 9 presents (for configuration (b)) the impact of variation in k (number of LSH functions) on the total FPR across all $r = 8$ sets and the maximum and minimum FPR over all sets. Here, l' is kept constant as 16, i.e. $m' = 2^{l'} = 4096$ bits. The total FPR decreases exponentially with increase in k. As k varies from 14 LSH functions to 30 LSH functions, the total FPR decreases from 12.9% to 0. The maximum FPR per set over all the 8 sets decreases

from 12.8% to 0. The minimum FPR per set, over all sets, remains low for all sets. This exponential decrease of FPR with increase in k also agrees with the theoretical bound on FPR given by Theorem 4.1. Note that the decrease of FPR with k is slower than w.r.t. l' with k constant, since l' term comes with exponent of 2, while k comes with exponent of 1.

Fig. 10 presents (for configuration (b)) the impact of variation in k on the total FNR across all $r = 8$ sets and the maximum and minimum FNR over all sets. Here, l' is kept constant as 16. The total FNR decreases exponentially with increase in k. As k varies from 14 to 30 LSH functions, the total FNR decreases from 12.9% to 0. The maximum FNR per set over all the 8 sets decreases from 10.5% to 0. The minimum FNR per set, over all sets, remains low for all sets. This exponential decrease of FNR with increase in k also agrees with the theoretical bound on FNR given by Theorem 4.1. As explained above, with increase in the number of hash functions, the memory consumed increases, and the error rate also goes down as more LSH functions are used to determine the membership per set.

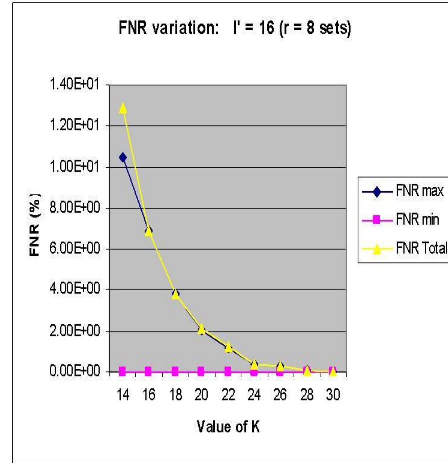

Figure 10: FNR Variation with K (8 sets)

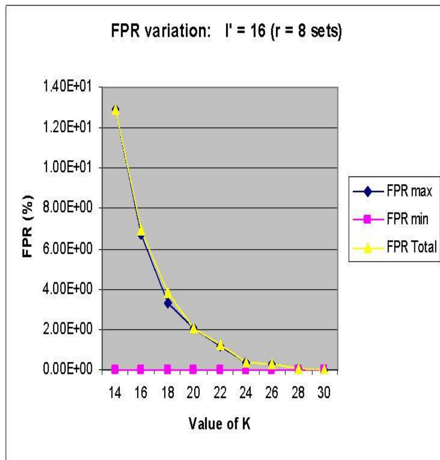

Figure 9: FPR Variation with K (8 sets)

6.2 Performance Analysis

This section presents the strong scalability, weak scalability and data scalability analysis for in-memory execution with constant $X = 512$ bytes, $l' = 25$ bits and $k = 32$ hash functions. The FPR and FNR observed in all these experiments were very close to zero ($< 1e - 5$). The time measured for these experiments represents the total time for approximate Range Motif discovery. In all these experiments we obtained FPR (false positive rate) and FNR (false negative rate) as zero as the parameters l' and k were chosen to be large.

6.2.1 Strong Scalability Analysis

For strong scalability, we keep both N and N_0 constant, while increasing the number of threads, T, from 4 to 16 (with increments of 4 threads). Fig. 11 displays the variation of the total time (with $N = 10M$, $N_0 = 10M$) with increasing number of threads. On Intel Xeon 5570, The total time decreases from $129s$ for 4 threads to $59s$ for 16 threads. This gives a relative speedup of around $2.2\times$ with $4\times$ increase in the number of threads (cores). This demonstrates the strong scalability of our parallel algorithm. Further, the Intel Xeon performance for $T = 16$ threads, gives the processing rate (for all $10M$ record processing) as $10M/59 \approx 170K$ records/s and $(170K * 4096bits) \approx 700Mb/s$ (since each record is of the size 4096 bits).

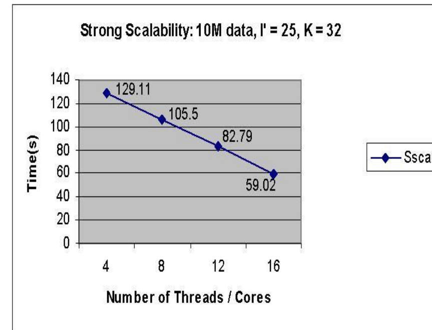

Figure 11: Strong Scalability: Range Motif

6.2.2 Weak Scalability Analysis

For weak scalability, we increase N as well as N_0, from $2M$ to $8M$ (in increments of $2M$) while maintaining $N = N_0$. At the same time, the number of threads, T, is also increased from 4 to 16 (in increments of 4 threads). Fig. 12 displays the variation of the total time with the increasing number of threads and data. In

case of Intel Xeon 5570, the total time increases from $25.76s$ for 4 threads to $47s$ for 16 threads. Thus, there is only $1.82\times$ increase of time with $4\times$ increase in the number of threads and the number of input objects/records. Thus our parallel algorithm demonstrates weak scalability as well.

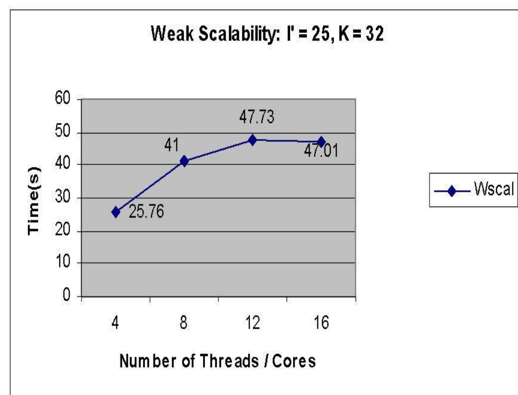

Figure 12: Weak Scalability: Range Motif

6.2.3 Data Scalability Analysis

For data scalability, N as well as N_0, from $1M$ to $8M$ (increasing by factor of $2\times$) while maintaining $N = N_0$. The number of threads, T, is kept constant at 16. Fig. 13 displays the variation of the total time with the increasing input data (number of records). The total time increases from $5.84s$ for $1M$ records to $47s$ for $8M$ records. The time increases by $8\times$ with $8\times$ increase in data. Thus, there is linear increase in time with increase in the input data. Thus our parallel ADRR algorithm also demonstrates data scalability.

7. CONCLUSIONS & FUTURE WORK

Real-time approximate Range Motif discovery and data redundancy removal for huge datasets ($10s$ to $100s$ of millions of records) is a very challenging problem. We have presented novel sequential and parallel algorithms for real-time approximate Range Motif discovery and data redundancy removal. Theoretical analysis for the asymptotic upper bounds on the false positive and false negative rates has been provided. Further, performance model for analysis of overall ARM / ADRR throughput has been provided. We demonstrated real-time approximate parallel data redundancy removal and Range Motif discovery using a random dataset of $10M$ records. We delivered a throughput of around $170K$ records/second or around $700Mb/s$ for records of size 4096 bits. In-depth study of the variation of FPR and FNR with changing design parameters has been provided. To the best of our knowledge, this is the best known throughput for approximate data redundancy removal. We hope our research will lead to further advances into data redundancy removal and motif discovery research. In future, we intend to investigate more general locality sensitive hash functions as well as scalability on many-core hybrid systems.

8. REFERENCES

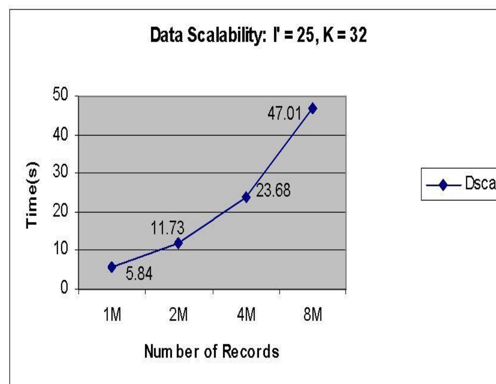

Figure 13: Data Scalability: Range Motif

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, 2002.

[2] F. Baboescu and G. Varghese. Scalable packet clasification. In *ACM SIGCOMM*, pages 199–210, 2001.

[3] S. Bhattacherjee, A. Narang, and V. K. Garg. High throughput data redundancy removal algorithm with scalable performance. In *HiPEAC*, 2011.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[5] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *SIGMOD Conference*, pages 398–409, 1995.

[6] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

[7] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: How much is enough? In *SIGMOD*, pages 436–447, 1998.

[8] Y. Chen, A. Kumar, and J. Xu. A new design of bloom filter for packet inspection speedup. In *GLOBECOMM*, pages 1–5, 2007.

[9] F. Deng and D. Rafiei. Approximately detecting duplicates for streaming data using stable bloom filters. In *SIGMOD Conference*, pages 25–36, 2006.

[10] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood. Deep packet inspection using parallel bloom filters. *IEEE Micro*, 24(1):52–61, 2004.

[11] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor. Longest prefix matching using bloom filters. In *ACM SIGCOMM*, pages 201–212, 2003.

[12] F. Douglis, J. Lavoie, J. M. Tracey, P. Kulkarni, and P. Kulkarni. Redundancy elimination within large collections of files. In *In USENIX Annual Technical Conference, General Track*, pages 59–72, 2004.

[13] L. Fan, P. Cao, J. Almeida, and Z. Broder. Summary cache: a

scalable wide area web cache sharing protocol. In *IEEE/ACM Transaction on Networking*, pages 281–293, 2000.

[14] W. Feng, D. Kandlur, D. Sahu, and K. Shin. Stochastic fair blue: A queue management algorithm for enforcing fairness. In *IEEE INFOCOM*, pages 1520–1529, 2001.

[15] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *25th International Conference on Very Large Databases (VLDB)*, pages 518–529, 1999.

[16] P. Gopalan and J. Radhakrishnan. Finding duplicates in a data stream. In *SODA*, pages 402–411, 2009.

[17] T. Hofmann. Optimizing distributed joins using bloom filters. *Distributed Computing and Internet technology (Springer / LNCS)*, 5375:145 – 156, 2009.

[18] Y. Hua and B. Xiao. A multi-attribute data structure with parallel bloom filters for network services. In *International Conference on High Performance Computing*, pages 277–288, 2006.

[19] N. Jain, M. Dahlin, and R. Tewari. Taper: Tiered approach for eliminating redundancy in replica synchronization. In *FAST*, 2005.

[20] K. R. Kanth, D. Agrawal, and A. Singh. Dimensionality reduction for similarity searching in dynamic databases. In *SIGMOD*, pages 166–176, 1998.

[21] A. Kirsch and M. Mitzenmacher. Distance-sensitive bloom filters. In *Eighth Workshop on Algorithm Engineering & Experiments(ALENEX)*, 2006.

[22] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li. Space-code bloom filter for efficient per-flow traffic measurement. In *IEEE INFOCOM*, pages 1762–1773, 2004.

[23] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *FAST*, pages 111–123, 2009.

[24] M. Little, N. Speirs, and S. Shrivastava. Using bloom filters to speed-up name lookup in distributed systems. *The Computer Journal (Oxford University Press)*, 45(6):645 – 652, 2002.

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

[26] Y. Matias, J. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimations. In *SIGMOD*, pages 448–459, 1998.

[27] M. Mitzenmacher. Compressed bloom filters. In *IEEE/ACM Transaction on Networking*, pages 604–612, 2002.

[28] A. Mueen, E. Keogh, Q. Zhu, S. Cash, and B. Westover. Exact discovery of time series motifs. In *Siam International Conference on Data Mining (SDM09)*, 2009.

[29] F. Putze, P. Sanders, and J. Singler. Cache-, hash-, and space-efficient bloom filters. *ACM Journal of Experimental Algorithmics*, 14, 2009.

[30] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *FAST*, pages 89–101, 2002.

[31] C. Saar and M. Yossi. Spectral bloom filters. In *ACM SIGMOD*, 2003.

[32] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. C. Bressoud, and A. Perrig. Opportunistic use of content addressable storage for distributed file systems. In *USENIX Annual Technical Conference, General Track*, pages 127–140, 2003.

[33] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity search methods in high dimensional spaces. In *24th International Conference on Very Large Databases (VLDB)*, pages 194–205, 1998.

[34] B. Zhu, K. Li, and R. H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST*, pages 269–282, 2008.

[35] Y. Zhu, H. Jiang, and J. Wang. Hierarchical bloom filter arrays (hba): A novel, scalable metadata management system for large cluster-based storage. In *5th IEEE International Conference on Cluster Computing (Cluster)*, pages 165–174, 2004.