Partitioning based algorithms for approximate and exact Iceberg Queries

Yossi Matias* **Eran Segal†**
Department of Computer Science
Tel-Aviv University
{matias,erans}@math.tau.ac.il

November, 1998

Abstract

In many applications it is necessary to identify items which occur frequently within the data set, which may be a materialized or non-materialized relation. Such queries were recently denoted as *iceberg queries*. Several algorithms for computing iceberg queries were presented, including an approximation algorithm based on concise sampling, and an exact algorithm based on sampling combined with multiple hash functions. In this paper, we propose a new approach for approximating iceberg queries, using a hash based partitioning technique. The data set is partitioned using a hash function into value independent subsets, resulting with suitably smaller independent sub-problems that can be handled effectively with guaranteed performance. We realize our method by deriving two new algorithms that allow a user to specify the threshold, as well as approximation parameters. The algorithms efficiently compute the iceberg queries, guaranteeing that the specified parameters are satisfied for any input data, and any given memory constraints. These are features which are not guaranteed by current techniques. The algorithms can be used to provide quick and rough estimates for iceberg queries, and gradually refine the estimates until reaching satisfactory bounds, or complete and exact solutions. Our first algorithm is fully in-place, requiring no additional working disk space. Our second algorithm is an efficient one-pass algorithm, and can be applied on-line for streaming data, without accessing a data base, and without materializing data sets that are specified implicitly. The analysis of the algorithms is supported by experimental study.

*Department of Computer Science, Tel-Aviv University, Tel-Aviv 69978 Israel. Email: matias@math.tau.ac.il.
†Department of Computer Science, Tel-Aviv University, Tel-Aviv 69978 Israel. Email: erans@math.tau.ac.il.

1 Introduction

A growing number of applications require efficient query execution on massive data sets, using limited memory. An important class of queries which has recently received increasing attention is the class of queries that find aggregate values over an attribute (or set of attributes) above some specified threshold. This class of queries was denoted by Fang *et al.* [FSGM+98] as *iceberg queries*, because the number of above threshold results is often very small (the tip of an iceberg) relative to the large amount of input data (the iceberg). More concretely, consider the following iceberg query: Given a sequence, S, of N items, a memory bound M, and a specified frequency threshold s, find all items whose frequency in S exceeds s.

Iceberg queries have many applications in decision support systems and in data mining applications for large data warehouses. These applications include market basket queries on large data warehouses that store customer sales transactions, to derive association rules [AS94]. A number of papers propose techniques to accelerate the execution of such queries [AIS93, AS94, SA95, Toi96, BMUT97]. In these applications, the sequence S given as input to the iceberg query is typically described in an implicit way: for a given sequence S' consisting of N' u-tuples, the sequence S may be defined as the $N = \binom{u}{3}N'$ items, consisting of all possible triplets from the same u-tuples. As argued by Park *et al.* [PCY95], the above query dominates the execution time of finding association rules.

Other applications illustrated in [FSGM+98] are finding popular item queries, related to the TPC-D benchmark [TPC], and document overlap queries used in web-searching engines [Bro97, BGM97] and in many information retrieval problems. Related problems are those of finding *hot spots*, studied in [BSL92], and *hot-list* queries, studied in [GM98a], in which the ρ most popular items in a data set are to be (approximately) identified, for a prespecified ρ.

Fang *et al* [FSGM+98] propose a variety of algorithms for iceberg query processing. Their algorithms use as building blocks well-known techniques such as *sampling* and *multiple hash functions*, but combine them and extend them to improve performance and reduce memory requirements. Their techniques avoid sorting or hashing the data, by keeping compact, in-memory structures that allow them to identify the threshold targets. However, in some unfavorable data distributions, their algorithms may experience reduced performance and the memory usage may be large. Furthermore, they do not give guaranteed performance for all data sets.

In [GM98a], new on-line sampling techniques were presented for approximating frequent items in data sets. The algorithms there effectively use available memory for identifying frequent items. While their algorithms may work well in practice in many situations, it can guarantee good approximation only for certain circumstances, and there may be many cases in practice for which the performance would not be satisfactory. In contrast, we seek algorithms which works for any given data set, available memory, and confidence and error parameters.

1.1 A case for approximate iceberg queries

We propose and advocate the use of approximate iceberg queries, defined as follows: Given a sequence, S, of N items, a memory bound M, prespecified frequency thresholds s_p, s_n, and confidence levels c_p, c_n, compute a list of items such that every item whose frequency exceeds s_p in S is included with a probability of at least c_p and every item whose frequency is below s_n in S is excluded with a probability of at least c_n.

For example, assume that we have a sequence, S, of size N for which we want to find association rules, involving items whose frequency in S is at least 1000. In all current algorithms the first step is to identify all the large item sets whose frequencies exceed 1000. The next step is to find the

interesting rules among these item sets. Note that the rules that are eventually reported contain only a fraction α of the item sets whose frequencies exceed 1000. We can set c_p to 99% and s_p to 1000. Then, for any α expected fraction of association rules which are missed is at most 1/100. To reduce the amount of item sets which are falsely reported, we can also specify that we want 99% of all items whose frequency is below 950 to be excluded from the report, by setting c_n to 99% and s_n to 950. The result is an approximation, which can be as accurate as we have specified, and can take benefit in the approximation to provide enhanced performance in terms of both execution time and memory usage.

Arguably, for many applications of frequent item set identification, an approximation algorithm would suffice, especially if the approximation can be as good as we want (while, of course, performance degrades as the accuracy of the approximation increases). For instance, in the application of mining association rules, the requirement to find large item sets with certain frequency is of heuristic nature, implying that the level of accuracy can be reduced. Furthermore, only a fraction of the large item sets found eventually participate in the final report of the association rules. In the application of document overlap queries, the search results can typically be of a lowered accuracy level, since in many cases only a fraction of the search results are indeed examined.

The approximation algorithms fit naturally into an approximate query answer framework [GM98a, GMP97]. They can be executed while performing only one scan over the data, without materializing it, using a framework related to synopsis data structures discussed in [GM98b]. This implies better performance. It also implies that the new algorithm can be used in situations where the iceberg algorithm of [FSGM+98] cannot. Namely, it can be used when the data set is seen once, but in which there is no access to the data afterwards, such as in routers, host computer for database applications, proxy servers for web traffic (*cf* [FJS97, GM98b]).

1.2 Algorithms for approximate and exact iceberg-queries

The algorithms given in [FSGM+98] for computing iceberg-queries can be adapted to compute approximate iceberg queries as well. No bounds on performance and accuracy are known for such adaptation. It can be seen that at least in some cases, where the memory is insufficient to hold the necessary data structures, the execution time may be quite high. The on-line sampling techniques of [GM98a], denoted as *concise sampling* and *count sampling*, can also be used to compute approximate iceberg queries. A natural adaptation of the algorithms to meet some prespecified approximation parameters would be to execute several passes over the data, where each new pass adds to the list of output items, and marks these items to be excluded from future consideration. Having 3ℓ passes over the data, using memory of size M, is roughly equivalent to having a single pass with memory of size ℓM

In this paper we propose a new approach for approximating iceberg queries, using a hash based partitioning technique. The data set is partitioned using a hash function into value independent subsets, resulting in suitably smaller independent sub-problems that can be handled effectively with guaranteed performance. Specifically, we take the concise sampling technique of [GM98a] as a building block, and we analyze its merits and limitations. The general idea is to identify the problem size for which concise samples with a given space can provide output within the approximation parameters which are pre-specified. Then, the hash based partitioning is used to obtain sub-problems of the appropriate size. An important feature is that by solving the sub-problems independently, the combined solutions of these sub-problems consist an appropriate solution to the original problem.

Given the partitioning-based approach, the algorithmic challenge is to derive algorithms that can handle simultaneously many independent sub-problems. Note that the sub-sequences, which

are the inputs for the sub-problems, are interleaved. Consequently, a single one-pass application of the concise sampling algorithm of [GM98a] cannot work, since the available memory cannot hold all concise samples simultaneously. We derive two new algorithms that realize the partitioning-base approach.

A bucket-sorting based algorithm

The first algorithm is based on the following simple observation: Suppose that the sequence is to be permuted, so that the sub-sequences are contiguous, without interleaving. Then, one pass over the sequence scans each sub-sequence at a time, allowing to hold in memory the concise sample of only one sub-problem at a time, without conflicts. The above permutation problem can be described as an integer-sorting problem, where items are sorted according to the sub-problem to which they belong. Specifically, for k sub-problems, the input to the integer-sorting problem consists of N keys from $[1 \ldots k]$. In a companion paper [MS98] we derive an integer-sorting algorithm which for $k \ll N$ (this is the case here) is substantially faster than general sorting, requiring at most $3 \lg_{M/B} k$ passes over the data set, where M is the available memory and B is the transfer block size. The integer sorting algorithm is an in-place algorithm, requiring no additional working space. As a result, we obtain an efficient algorithm for approximate iceberg-queries, which requires only very few passes over the data (depending on the integer sorting algorithm) and requires no additional working disk space. It is interesting to note that 3ℓ passes using the simple adaptation of the concise sampling algorithm, could be reduced to $3 \lg_{M/B} \ell$ passes, using our integer sorting-based algorithm.

A one-pass algorithm

A tougher challenge is to compute approximate iceberg queries in one-pass over the sequence without materializing the sequence, using a limited amount of memory and working disk space. The challenge includes reducing the amount of I/Os to the local disk and using the limited available memory in an efficient manner.

Our second algorithm accepts this challenge and computes the approximation query while performing only one pass over the sequence. Hence, it can be applied on-line for streaming data, without accessing a data base, and without materializing data sets that are specified implicitly. The algorithm takes the partitioning approach as well but rather than sorting the sequence and deal with each partition separately, it maintains all concise samples (one for each partition) simultaneously. Since memory constraints may prevent all concise samples to be held together in main memory, the algorithm resorts to additional working disk space. The main memory then serves as a buffer for the concise samples which now reside on disk. As long as the memory suffices, it holds all the inserts that were supposed to be to the concise samples, hence delaying their update and reducing the number of I/Os to the disk. Once becoming full, we stream the largest partition from memory into the disk. As a technical note, the memory available for each concise sample is also a parameter that needs to be resolved based on some trade-off and we show how it is used in an optimal way.

Some features of the new algorithms

We give rigorous analysis for the guarantees of both algorithms. We show that, under reasonable assumptions that occur in many practical cases, our one-pass algorithm performs a constant number of accesses to the working space for each block in the sequence. The analysis is for a worst-case distribution of frequencies in the input data set (in which most frequencies are near the threshold frequencies). In most cases, the distribution of the input data set would be more favorable, and

the performance (both in terms of time execution, accuracy and confidence) would be substantially better. This is supported by experiments on U.S. census data and on synthetic data generated by the IBM test data generator. The final paper will include more experiments on larger and other types of data sets.

The algorithms efficiently compute the iceberg queries, guaranteeing that the specified parameters are satisfied for any input data, and any given memory constraints. These are features which are not guaranteed by current techniques. The algorithms can be used to provide quick and rough estimates for iceberg queries, and gradually refine the estimates until reaching satisfactory bounds, or complete and exact solutions similar to the framework of on-line aggregation [HHW97]. Hence, our algorithms enables us to derive at exact solutions with enhanced performance (since guarantees that the data set is of prespecified characteristics).

If the sequence is materialized and can be accessed more than once, then we realized an algorithm that can sort in-place the data efficiently and then perform one pass over the data to answer the query. Note that in this latter case, subsequent executions of the algorithm can be executed in only one pass over the data, since it is already sorted. In fact, the bucket sorting-based algorithm can be naturally adapted to provide gradually refined approximations. After every pass, the quality of output improves in terms of accuracy and confidence.

The rest of the paper is organized as follows. Section 2 presents concise sample and analyzes their performance in detail. Section 3 defines approximate iceberg queries, and outlines the partition-based approach we take in solving it. Section 4 and 5 detail the in-place and one-pass algorithms, respectively. Section 6 shows how to extend both algorithms to provide exact answers and how to extend the in-place algorithm to provide gradually improving answers. Section 7 describe the experiments and conclusions are provided in Section 8.

2 Concise samples

In this section we present concise samples as introduced in [GM98a] but we give a more detailed analysis including studying their limitations in detail. We will use concise samples as a basic ingredient in the algorithms we develop in later sections and the analysis will be used to optimize the design of our algorithms. Consider the class of queries that ask for the frequently occurring values for an attribute in a sequence of size N. A possible synopsis data structure is the set of attribute values in a uniform random sample of the items in the sequence: any value occurring frequently in the sample is returned in response to the query. However, note that any value occurring frequently in the sample is a wasteful use of available space. We can represent k copies of the same value v as the pair $\langle v, k\rangle$, thereby freeing up space for $k-2$ additional sample points. Formally we have:

Definition 2.1 *A concise sample is a uniform random sample of a sequence such that values appearing more than once in the sample are represented as a value and a count.*

Concise samples are never worse than traditional samples, and can be exponentially or more better depending on the data distribution. A concise sample can be viewed as a sequence S: $S = \{\langle v_1, c_1\rangle, \ldots, \langle v_j, c_j\rangle, v_{j+1}, \ldots v_l\}$. We denote the sample size and footprint of S as: $\mathsf{sample\text{-}size}(S) = i - j + \sum_{i=1}^{j} c_i$, and $\mathsf{footprint}(S) = l + j$. The sample size is the number of elements selected to be in the sample. The footprint is the space consumed.

Computation. We describe an algorithm from [GM98a] for extracting a concise sample of footprint at most m, from a sequence. First, set up an entry threshold r (initially 1) for new items

to be selected for the sample. Let S be the current concise sample and consider a new item x. With probability $1/r$, we add x to S as follows: We do a lookup on x in S; if it is represented by a pair, we increment its count. Otherwise, if x is a singleton in S, we create a pair, $\langle x, 2\rangle$, or if it is not in S, we create a singleton $\langle x\rangle$. In these latter two cases we have increased the footprint by 1, so if the footprint for S was already equal to the predefined footprint bound m, then we need to evict existing sample points to create room. In order to create room, we raise the threshold to some r' and then subject each sample point in S to this higher threshold. Specifically, each of the sample-size(S) sample points is evicted with probability $1 - r/r'$. We expect to have sample-size$(S) \cdot (1 - r/r')$ sample points evicted. Note that the footprint is only decreased when a $\langle value, count\rangle$ pair reverts to a singleton or when a value is removed altogether. If the footprint has not decreased, we raise the threshold and try again. Subsequent inserts are selected for the sample with probability $1/r'$. It was shown in [GM98a] that for any sequence of insertions, the above algorithm maintains a concise sample.

The algorithm maintains a concise sample regardless of the sequence of increasing thresholds used. Thus, there is a complete flexibility in deciding, when raising the threshold, what the new threshold should be. A large raise may evict more than is needed to reduce the sample footprint below its upper bound, resulting in a smaller sample-size than there would be if the sample footprint matches the upper bound. On the other hand, evicting more than is needed creates room for subsequent additions to the concise sample, so the procedure for creating room runs less frequently. Note that instead of flipping a coin for each item, we can flip a coin that determines how many such items can be skipped before the next item encountered must be placed in the sample (as in Vitter's reservoir sampling Algorithm X [Vit85]): the probability of skipping over exactly i elements is $(1 - 1/r)^i \cdot (1/r)$. As r gets large, this results in a significant savings in the number of coin flips.

2.1 Using concise samples for reporting frequent items

We consider applying concise samples to approximating queries of the type: given a sequence, report items whose frequency exceeds a predefined frequency, s. We can choose to report frequent items in the following manner:

Reporting frequent items. Once a concise sample has been extracted from a sequence, we report all pairs with counts at least δ in the sample, where $\delta \simeq s/r$. Consider a value v whose frequency in the sequence is f_v. Since a concise sample maintains that each item is inserted into the sample with probability of $1/r$, we expect that the count of v in the sample will be f_v/r. Thus, for every item in the sample whose count is δ, we estimate that its frequency in the sequence is at least $\delta \cdot r$.

Since concise sample is an approximation, we want to quantify its use for reporting frequent items. Assume we want items whose frequency exceeds some frequency s_p to be reported and we do not want items whose frequency is below some frequency s_n to be reported. We want to be able to measure the two errors that are then produced by the approximation: (1) 'false negatives' – those items whose frequency is above s_p that are not reported, and (2) 'false positives' – those items whose frequency is below s_n that are reported. Throughout the paper we denote c_p as the probability that every item whose frequency is above s_p is reported (so $1 - c_p$ is about the fraction of false negatives) and c_n as the probability that every item whose frequency is below s_n is not reported (so $1 - c_n$ is about the fraction of false positives). We call these probabilities the *reporting confidence level* and the *non-reporting confidence level*, respectively. For a given δ and r, we can either specify s_p and s_n and derive the corresponding c_p and c_n or vice versa. In the next lemma we show how this derivation is computed.

Lemma 2.1 *Let S be an arbitrary sequence and let r be the current threshold for a concise sample. Then:*

1. *Let δ_p be the count in the sample above which we report items. Then, for any ϵ_p, $0 < \epsilon_p < 1$, the probability that a value v with frequency f_v, such that $f_v \geq r\delta_p/(1-\epsilon_p)$, is reported, will be at least $1 - e^{\frac{-\epsilon_p{}^2\delta_p}{2(1-\epsilon_p)}}$.*

2. *Let δ_n be the count in the sample below which we do not report items. Then, for any ϵ_n, $0 < \epsilon_n$, the probability that a value v with frequency f_v, such that $f_v \leq r\delta_n/(1+\epsilon_n)$, is not reported, will be less than $e^{\frac{-\epsilon_n{}^2\delta_n}{3(1+\epsilon_n)}}$.*

Proof. We use Chernoff bounds. Recall that Chernoff bounds state that for a sequence of independent Bernoulli trials, X_i, with probabilities P_i, $\mu = E[\sum X_i] = \sum P_i$, the number of successful trials Y satisfies:

1. $Pr[Y \leq (1-\epsilon)\mu] \leq e^{\frac{-\epsilon^2\mu}{2}}$
2. $Pr[Y \geq (1+\epsilon)\mu] \leq e^{\frac{-\epsilon^2\mu}{3}}$

Consider a value v whose frequency, f_v, is $f_v \geq r\delta_p/(1-\epsilon_p)$. Recall that in a concise sample, each item is inserted with probability $1/r$. It follows that the expected number of successful trials for v is: $\mu_v = f_v/r \geq \delta_p/(1-\epsilon_p)$, and hence, substituting μ in equation 1 we obtain:

$$Pr[Y \leq \delta_p] \leq Pr[Y \leq (1-\epsilon_p)\mu_v] \leq e^{\frac{-\epsilon_p{}^2\delta_p}{2(1-\epsilon_p)}} \;\Rightarrow\; Pr[Y > \delta_p] > 1 - e^{\frac{-\epsilon_p{}^2\delta_p}{2(1-\epsilon_p)}}$$

Similarly, consider a value whose frequency, f_v, is $f_v \leq r\delta_n/(1+\epsilon_n)$. It follows that $\mu_v = f_v/r \leq \delta_n/(1+\epsilon_n)$, and hence

$$Pr[Y \geq \delta_n] \leq Pr[Y \geq (1+\epsilon_n)\mu_v] \leq e^{\frac{-\epsilon_n{}^2\delta_n}{3(1+\epsilon_n)}}$$

■

From the lemma above, we can see that there is a tradeoff: we can set only one parameter in each of the pairs $\langle s_p, c_p\rangle, \langle s_n, c_n\rangle$ thereby imposing the value of the other parameter. In Section 3 we show how to extend concise sample to give the user control over all four parameters.

Since s_p and s_n are such essential parameters we want to study their relationship with c_p and c_n more carefully. More specifically, we show how to compute the coefficients $t_p = \delta_p/(1-\epsilon_p)$ and $t_n = \delta_n/(1+\epsilon_n)$ since, as can be seen in Lemma 2.1, they factor the threshold, r, in computing s_p and s_n from c_p and c_n. Throughout the paper we refer to t_p and t_n as the *positive confidence coefficient*, and the *negative confidence coefficient*, respectively. In the following lemma we show how t_p and t_n are computed.

Lemma 2.2 *Let S be an arbitrary sequence, r be the current threshold for the concise sample. Then:*

1. *Let δ_p be the count in the sample above which we report items and let $\beta = \ln(1-c_p)$. If we set t_p, as:*

$$t_p = \frac{\delta_p}{1-\epsilon_p} \quad where \quad \epsilon_p = \frac{\beta + \sqrt{\beta^2 - 2\beta\delta_p}}{\delta_p}$$

then any value v, whose frequency, f_v, is $f_v \geq rt_p$, will be reported with a probability of at least c_p.

2. *Let δ_n be the count in the sample below which we do not report items and let $\beta = \ln(1 - c_n)$. If we set t_n, as:*

$$t_n = \frac{\delta_n}{1 + \epsilon_n} \quad where \quad \epsilon_n = \frac{-3\beta + \sqrt{9\beta^2 - 12\beta\delta_n}}{2\delta_n}$$

then any value v, whose frequency, f_v, is $f_v \leq rt_n$, will not be reported with a probability of at least c_n.

Proof. Using Lemma 2.1, the first part is this: for any value v, with frequency f_v, such that $f_v \geq r\delta_p/(1 - \epsilon_p)$ where $0 < \epsilon_p < 1$, we comply with the reporting confidence level c_p so that:

$$1 - e^{\frac{-{\epsilon_p}^2 \delta_p}{2(1-\epsilon_p)}} = c_p$$

Substituting β and applying simple algebra we achieve:

$${\epsilon_p}^2 \delta_p - 2\beta\epsilon_p + 2\beta \geq 0$$

Since $0 < c_p < 1$ it follows that $\beta < 0$ and thus: $\frac{\beta - \sqrt{\beta^2 - 2\beta\delta_p}}{\delta_p} < 0$ imposing:

$$\epsilon_p = \frac{\beta + \sqrt{\beta^2 - 2\beta\delta_p}}{\delta_p}$$

Similarly, the second part is this: for any value v, with frequency f_v, such that $f_v \leq r\delta_n/(1 + \epsilon_n)$ where $\epsilon_n > 0$, we comply with the non-reporting confidence level c_n so that:

$$e^{\frac{-{\epsilon_n}^2 \delta_n}{3(1+\epsilon_n)}} = 1 - c_n$$

We obtain the result using a similar computation as done for the first part. ∎

As stated earlier, we will use concise samples as a basic ingredient for approximating the general case query for finding frequent items in a given sequence. In the algorithms we develop we give the user control over the confidence level by which items are reported. We allow the user to set the pairs $\langle s_p, c_p \rangle$ and $\langle s_n, c_n \rangle$ which specify that every item whose frequency in the sequence is above s_p is reported with a probability of at least c_p and every item whose frequency is below s_n is not reported with a probability of at least c_n. The actual values depend on the specific application. In applications in which it is crucial to minimize the errors, the gap between s_p and s_n will be small resulting in a larger memory requirement for our algorithms. In applications in which we are mainly concerned with the positive items, the gap will be larger and would result in enhanced performance. We address these issues in the experimentation section, and show the tradeoffs there.

2.2 Estimating the concise sample gain

As explained in the previous section, a concise sample is never worse than a traditional sample and thus the sample size is always greater than the actual memory footprint and can typically be much greater depending on the data distribution. The gain achieved by using concise samples over traditional samples is the measure of the ratio between the sample size and the footprint. In [GM98a] it was shown that for any data set, when using a concise sample S with sample-size m' and footprint M, the expected gain is:

$$E[m' - M] = \sum_{k=2}^{m'} (-1)^k \binom{m'}{k} \frac{F_k}{n^k}$$

where F_k is the frequency moments defined as $F_k = \sum_j n_j^k$, j is taken over the values represented in the set and n_j is the number of set elements of value j. We derive a bound for the gain depending only on F_0 which is the number of distinct items in the sequence. We use a pessimistic case in which all the multiplicities of the items are the same and arrive at a lower bound for the sample size. This bound will later be used to analyze the performance of our algorithms for approximating iceberg queries. We show this in the next lemma:

Lemma 2.3 *Let S be an arbitrary sequence, let F_0 be the number of distinct items in S, let M be the footprint for a concise sample maintained on S and let m' be the sample size such that $m' \leq \epsilon N$. Then, m' satisfies:*

$$E(m') \geq (1-\epsilon)\left(M + \frac{M^2}{2F_0}\right)$$

Proof. We assume a worst case scenario in which the multiplicities of all F_0 distinct items in S are the same: N/F_0, where N is the size of the data set. We describe a simple probabilistic process which generates a uniform random sample S' of S, so that the number of distinct values in S' is M, and we then evaluate $m' = |S'|$. To generate S', we repeatedly select uniformly at random an item from S, until the number of distinct values in S' reaches M. The process can be considered as consisting of M *phases*, where phase i begins immediately after the number of distinct values in S' has become $i-1$, and ends when the ith new value is selected. Let s_i be the number of steps in phase i. Then, clearly

$$m' = \sum_{i=1}^{M} s_i$$

Let $X = N/F_0$. The probability ρ_i of selecting a new value in phase i is

$$\rho_i \geq \frac{XF_0 - X(i-1)}{XF_0 - m'} = \frac{XF_0 - X(i-1)}{XF_0(1-\epsilon)} = \frac{F_0 - (i-1)}{F_0(1-\epsilon)}$$

and therefore

$$E(s_i) = 1/\rho_i = \frac{F_0(1-\epsilon)}{F_0 - (i-1)} = (1-\epsilon)\left(1 + \frac{i-1}{F_0 - (i-1)}\right) \geq (1-\epsilon)\left(1 + \frac{i-1}{F_0}\right)$$

Hence,

$$E(m') = \sum_{i=1}^{M} E(s_i) \geq (1-\epsilon)\left(M + \sum_{i=1}^{M} \frac{i-1}{F_0}\right) = (1-\epsilon)\left(M + \frac{M^2}{2F_0}\right)$$

■

Note that when F_0 approaches N, the expected sample size becomes close to the footprint since in this extreme case there can be no gain from using concise samples. As a technical note, when the sampling is done with replacements the sample size satisfies

$$E(m') \geq \left(M + \frac{M^2}{2F_0}\right)$$

We now evaluate the threshold r maintained by the concise sample and its relationship to the sequence size N and the memory footprint M. Since the threshold, r, is raised in accordance to the sample size, it will satisfy

$$r \simeq N/m' \leq \frac{2NF_0}{(1-\epsilon)M(M+2F_0)}$$

3 Approximating Iceberg Queries

In this section we present the problem at hand formally and outline the general idea of our approaches to solving it. The analysis and techniques given in this section will be used in Section 4 and 5 where our two detailed algorithms are presented. We present the problem more formally:
Input A sequence, S, of N items, a memory bound M, frequencies s_p and s_n and confidence levels c_p and c_n.
Output A list of items such that every item whose frequency exceeds s_p in S is included with a probability of at least c_p and every item whose frequency is below s_n in S is excluded with a probability of at least c_n.

We now give an example that illustrates the use of the above parameters. Assume that we have a sequence, S, of size N. A typical application can be interested in finding items in S whose frequency is above 1000. Furthermore, we can specify that we want the query to report at least 95% of all items whose frequency exceeds 1000. In this case we set s_p to 1000 and c_p to 95%. To avoid unwanted reports, we can also specify that we want 98% of all items whose frequency is below 950 to be excluded from the report, by setting c_n to 98% and s_n to 950. Another typical application can set s_n to 1000 and c_n to 95%, meaning that at least 95% of all reported items exceed a frequency of 1000.

The algorithms we present use concise samples since they are powerful in answering the above query using a confined memory. However, as shown in the previous section, we can control only one parameter in each of the pairs $\langle s_p, c_p\rangle, \langle s_n, c_n\rangle$ simultaneously. Thus, given all the above as constraints, it should be clear that one concise sample may be insufficient in providing control over all the parameters as required.

To emphasize this, let r be the entry threshold for a concise sample. Recall that each item is drawn into the sample with probability of $1/r$. It is fairly easy to note that an item must have a frequency of at least r in order to have a good chance at getting into the sample. In fact, as shown in the previous section, in order to comply with a given confidence level, r must be factored by some constant (t_p and t_n for the reporting and non-reporting confidence levels, respectively). For instance, if we want a reporting confidence level of at least 90% then $t_p \simeq 8$. This means that for an item to be reported with a confidence level of at least 90%, its minimum frequency must be $\sim 8r$ which can typically be $\sim 8N/m'$, where m' is the sample size of the concise sample.

Note that there is an interval that includes items whose frequency is between s_p and s_n, in which the probability that an item will be reported is unknown. To receive more controlled results we would like to minimize this interval as much as we can. Since the footprint is predefined, it is easy to observe that as the size of the sequence increases the concise sampling algorithm looses flexibility and for increasing c_p and c_n we will require the interval between s_p and s_n to increase, resulting in reduced accuracy. Conversely, for a diminishing interval, we impose decreasing c_p andc_n. We therefore conclude that in the general case, concise samples cannot cope with a memory footprint constraint combined with a constraint of either a predefined confidence level or a predefined frequency threshold. It is thus required to develop algorithms that achieve this flexibility.

Our strategy will therefore be a divide and conquer technique: We compute the maximum sequence size for which we could answer the query under the given constraints. We then partition the sequence into subsequences of fairly equal size using a hash-based partition. All instances of a given item will be partitioned into the same subsequence, resulting in value-independent subsequences. The problem of finding frequent items is then reduced to finding frequent items in each subsequence and combining the results. Since the size of each partition is such that a single concise sample can be maintained for it, within the specified constraints, we now find the frequent items in each

subsequence by maintaining a single concise sample for it. In effect, partitioning the sequence results in reducing r, since we saw, in the previous section, that r is proportional to the size of the sequence.

In Section 3.1 we show how the maximum partition size is computed. In Section 3.2 we explain how the partition is done into equal sized subsequences. The analysis and techniques given in these sections will be used in both our algorithms presented in Sections 4 and 5.

3.1 Computing the maximum sequence size for a single concise sample

Until now, we have treated false positives and false negatives separately. As explained above, our goal is to give the user control over all parameters simultaneously. The implication of this in a concise sample is that $\delta_n = \delta_p - 1$, since we can only choose one count in the sample above which items will be reported and below which items will not be reported. Subjecting all our analysis so far to this constraint will result in the combined query as required. Hence, the analysis provided here will be used in the following sections when we present the two algorithms.

We now show how to compute the maximum partition size that can be handled in a single concise sample, given the frequency thresholds, memory size and confidence levels as constraints. The maximum partition size will be used to determine the number of partitions into which we will divide the sequence, in order to satisfy the above constraints. Throughout the paper, let μ' be the expected sample size: $\mu' = E[m']$. Clearly, for a concise sample with threshold r, $\mu' = N/r$.

Lemma 3.1 *For a reporting confidence level c_p, a non-reporting confidence level c_n, a minimum frequency threshold s_p and a maximum frequency threshold s_n, we can maintain a concise sample with threshold r, for a sequence of size at most $N' = \min(s_p\mu'/t_p, s_n\mu'/t_n)$ where t_p and t_n are the confidence coefficients as computed in Lemma 2.2.*

Proof. Given the above constraints, we shall see the maximum sequence size imposed by $\langle s_p, c_p\rangle$ and $\langle s_n, c_n\rangle$ separately and then subject the combined maximum sequence size to the minimum of both. For s_p and c_p, we want items of frequency at least s_p to be reported with confidence level c_p. Using Lemma 2.2 it is sufficient to have:

$$s_p = rt_p = \frac{N't_p}{\mu'}$$

thus, for any sequence of size at most $N' = s_p\mu'/t_p$ we can maintain a concise sample satisfying s_p and c_p. Similarly, to satisfy s_n and c_n, we must use a sequence of size at most: $N' = s_n\mu'/t_n$. Combining the results, in order to satisfy all the above constraints, we must use a sequence of size at most:

$$N' = \min(s_p\mu'/t_p, s_n\mu'/t_n)$$

■

From the lemma above, we observe that in order to satisfy the above constraints for a sequence of size N the number of equal size partitions can be at most: $N/(\mu'\phi)$ where $\phi = \min(s_p/t_p, s_n/t_n)$.

3.2 Hash-based partitioning

In this section we show how to partition the sequence given as input the number of partitions required. The sequence must be partitioned such that all instances of an item will be partitioned together. This will result in independent subsequences and effectively reduces the problem into

smaller sub-problems. Our goal is to divide the sequence into partitions as nearly equal as possible to each other in terms of the number of distinct items in each partition.

Given the number of partitions, k, and a sequence comprised of items $\{a_1, a_2, \ldots a_n\}$, we use a mapping function, f, such that: $f : \{a_1, \ldots a_n\} \rightarrow \{1 \ldots k\}$. Then, partition j will be: $\{a_i | f(a_i) = j\}$. It is left to study that we can choose f such that for any given sequence and k, f will partition the sequence into fairly equal partitions. We will select the function f at random from a class of *universal hash functions* [CW79]. For instance, f could be a linear hash function, or a multiplicative hash function [Knu73] (which was recently shown to be 2-universal).

It can be shown that with high probability, regardless of the data distribution, the number of different values mapped by f to every partition is the same, within a small order error. If we only consider items whose frequency is well below the average size of a partition, then again, with high probability, the number of items mapped by f to every partition is the same, within a small order error. The same statement does not necessarily apply to items whose frequencies are at the order of the average partition size. We note, however, that such items are to be reported (w.h.p.) in their respective concise sample. Recall that the analysis given for concise sample is for worst case distribution, in which items to be reported have frequencies near the threshold frequency. For the purpose of the hash-partition analysis, we will assume that frequencies are indeed bounded by near threshold frequencies, and note that if an item is reported, then having actually a higher frequency does not influence the performance of concise sampling technique for these items nor for other items. The reason being that once an item is selected to be in the sample and exceeds the count of 2, any subsequent selection of the same item increases the count, but has no effect on the footprint usage. A detailed analysis and proof of the above will be given in the full paper.

4 In-place algorithm

We first describe an efficient sorting algorithm developed in a companion paper for in-place sorting in external memory that we will use for the algorithm.

In-place sorting in external memory. Let N be the size of the sequence to sort, B be the transfer block size, M be the available memory and k be the number of distinct items in the sequence. Compute the maximum number of transfer block sizes that can simultaneously fit into memory, $\lfloor M/B \rfloor$. Then, sort the sequence into $\lfloor M/B \rfloor$ buckets, by the following steps: initially count the number of distinct items for each of the $\lfloor M/B \rfloor$ buckets; next, load $\lfloor M/B \rfloor$ blocks, one from each bucket, from their final positions in the sorted sequence and swap items between buckets; when a block is full from items belonging to its bucket, write that block to disk and load the next block for that bucket. The sorting ends when we finished reading the whole sequence. In the case that $k > \lfloor M/B \rfloor$, perform this same sorting technique for each of the sorted buckets. It is easy to note that the whole sequence is sorted after $\lg k$ sorting iterations. In a companion paper [MS98] we show that the number of I/Os performed in such a sort is at most $O(N/B \lg_{\lfloor M/B \rfloor} k)$. Specifically we show the following:

Theorem 4.1 *We can sort in place a sequence residing on disk while the number I/O accesses is at most:*

$$min(\lceil 4N/M \rceil \cdot \lg k, \lceil 3N/B \rceil \cdot \lg_{\lfloor M/B \rfloor} k)$$

We are now ready to present the in-place algorithm. The general idea is this: first compute the number of partitions required as shown in Lemma 3.1. Next, in-place sort the sequence according to the partitions, using the above sorting algorithm. Note that at this point we have partitions,

each containing a distinct set of values and a size that is less than the maximum size that can be handled in a single concise sample. We now perform one pass over the sequence and maintain a concise sample for each partition separately, while reusing memory. We report the frequent items found in each partition and combine the results.

In-place algorithm
let L equal the number of partition required. Compute L (see Lemma 3.1)
sort the items into L partitions using the in-place sorting algorithm
let *item_buffer* be a buffer into which we load items from the sequence, initialized to the first buffer in the sorte
let *partition_number* be the examined partition, initialized to 1

while (*item_buffer* is not the last buffer in the sequence)
    if the *item_buffer* is done with **then** load the next buffer into *item_buffer*
    if the item encountered is not in *partition_number* **then**
        output items in the concise sample that passed the confidence test and clean the sample
        partition_number ++
    maintain a concise sample that includes the item

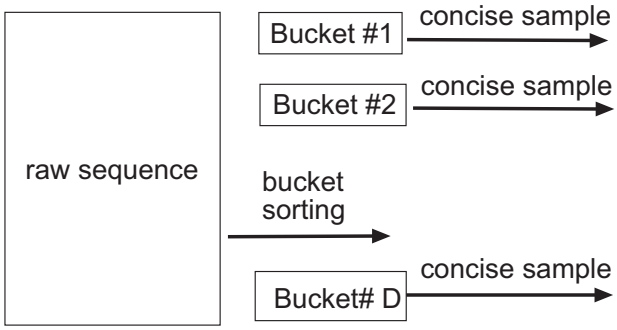

Figure 1: Finding large item sets using bucket sorting

In the following theorem we provide an analysis of the algorithm. For simplicity, we assume that the memory bound is at least twice the transfer block size. We later show the additional refinement required when this is not the case.

Theorem 4.2 *Let S be a sequence of size N, M be the memory bound, B be the transfer block size, s_p be the minimum frequency threshold, s_n be the maximum frequency threshold, c_p be the reporting confidence level and c_n be the non-reporting confidence level. Then, under the assumption that the memory bound is at least twice the transfer block size, the above algorithm answers the approximation query for finding frequent items with a total number of disk accesses less than:*

$$\min\left(\lceil\frac{2N}{B}\rceil \cdot \lg L, \lceil\frac{3N}{B}\rceil \lg_{\lfloor \frac{M}{B} \rfloor} L\right) + \frac{N}{B},$$

where $L = N/\min(s_p\mu'/t_p, s_n\mu'/t_n)$, and μ' is the expected sample size for each concise sample maintained in the algorithm.

Proof. To show that this algorithm indeed answers the approximation query, it suffices to show that every item is monitored by a single concise sample, for the given constraints. Observe that

when we run over the sorted sequence, we maintain a concise sample for each partition using a memory bound of $M - B$. Since the number of partitions are computed in accordance with Lemma 3.1, a suitable concise sample is maintained for each partition. We conclude that each item is matched to exactly one partition and that all partitions are examined and hence, each item is handled in exactly one concise sample.
We divide the disk accesses into two parts: the first is the sorting part, where the number of disk accesses, obtained from a companion paper using the assumption that $M > 2B$, is at most:

$$\min\left(\lceil\frac{2N}{B}\rceil \cdot \lg L, \lceil\frac{3N}{B}\rceil \lg_{\lfloor\frac{M}{B}\rfloor} L\right)$$

The second part is for the running pass over the sorted sequence. To minimize the disk accesses we allocate a memory of size B for reading items from the sequence. This results in an additional N/B disk accesses. By combining both disk accesses computed we obtain the claim in the theorem. ∎

For example, when $M > LB$, the total number of disk accesses is $4N/B$ which is the minimum disk accesses required to run over the sequence 4 times.

When the memory bound is less than twice the transfer block size a slight change in the analysis is required. Instead of the first addendum in Theorem 4.2 we require the full formula from our companion paper: $\min(\frac{2N \lg L}{\min(M/2,B)})$ total disk accesses. As for the second addendum, we need to divide the memory bound in two: one part for maintaining a concise sample of size m_1 and another part for the buffer size, denoted m_2 into which we load items from the sequence. The second addendum will therefore be: N/m_2. There is a tradeoff in deciding the ratio between the two: while increasing the size of the latter results in less disk accesses when maintaining the concise samples, it increases the number of buckets and thus the disk accesses required for the sorting process.

5 One pass algorithm

In this section we develop a technique to solve the problem without sorting the sequence while performing only one pass on the sequence and with the use of working space on disk, if required. We assume that the size of the sequence, N, is known a priori. In Section 5.1 we extend the algorithm to work even when N is not known beforehand while still performing only one pass. In Section 5.2 we extend the algorithm to work with tuples without materializing the sequence.

The general idea of the algorithm is this: first compute the number of partitions, L, required as in lemma 3.1 and derive a hash function that maps all items according to L. For each partition we maintain a separate concise sample; next, perform one pass over the sequence and monitor each item encountered by the concise sample to which it belongs. Depending on the maximum memory allowance, it might be infeasible to maintain all concise samples in memory. In this case we stream information to working space on disk as we run out of memory. We hold two tables in memory: a partition status table that keeps status information about the concise sample maintained by each partition, and an updates table that serves as a buffer and keeps all inserts to all concise samples until it needs to be streamed to disk due to lack of memory. The updates table is streamed when we run out of memory to hold the table. We formalize the algorithm:
Input A sequence, S, of size N; a maximum memory allowance M; reporting confidence level, c_p, and non-reporting confidence level, c_n; minimum frequency threshold, s_p, and maximum frequency threshold, s_n.
Output A list of items such that every item whose frequency exceeds s_p in S is included with

Partition#	Item ID	Item Count
3	A382	1
L	B423	8
9	A932	5

Table 1: Updates table

a probability of at least c_p and every item whose frequency is below s_n in S is excluded with a probability of at least c_n.

Memory usage. We maintain two tables in memory. The first is a partition status table, described below, whose size depends on the number of partitions. The second is a dynamic table that accumulates the updates for each partition.

Updates table. The structure of the table can be seen in Table 1. This table serves as a buffer for delaying updates to each concise sample. The partition number is the ID of the partition that the item is hashed into. The Item ID is a short identification of the item and the item count is the number of occurrences that this item has in the updates table. Note that when implementing the table there is no need to actually store the partition number since this information can be extracted by executing the hash function on the item. Also, the count of an item needs to be saved only in the cases where its count in the updates table is at least two.

Partition status table. The structure of the table can be seen in Table 2. Each partition is numbered and is given an entry in the table. For each partition we store information on its concise sample status. The threshold, r, defines the probability by which an item is inserted into the partition. The partition size is the total number of items hashed into the partition. Instead of flipping a coin for each insert into the sample, we flip a coin that determines how many such inserts can be skipped before the next insert must be placed into the sample (As in Vitter's reservoir sampling algorithm X [Vit95]) and store that number in the fourth column. Recall that we confine each sample to a maximum footprint. We store the memory footprint left for each sample in the fifth column. The last column stores the footprint of this partition in the updates table. As we will see later this information will be useful when streaming the updates table to disk. We denote this table as PS (partition status) and refer to each entry as PS(partition number, attribute) where attribute can be one of: threshold, partition size, Inserts to skip, footprint left or updates footprint.

Tables 1 and 2 show a typical snapshot of the algorithm. The updates table consists of three items, each belonging to its corresponding partition. The count for item $B423$, for instance, in the updates table is 8 and it consumes a footprint of 2 in the updates table. The partition status table shows the status of all partitions. For example, partition L currently consumes a footprint of 2 in the updates table, due to item $B423$. The number of inserts to skip in partition 2 is 1, indicating that the next item encountered belonging to partition 2 will be inserted into the updates table (and eventually to the concise sample corresponding to partition number 2).

Initializations.

1. Compute the confidence coefficients, t_p and t_n in compliance with the required confidence levels c_p and c_n, respectively, as in Lemma 2.2.

2. Compute the number of partitions, denoted L, as described in lemma 3.1. The memory allowance for each bucket, m, is: $m = -B + \sqrt{B^2 + MB}$ where M is the total memory bound

Partition#	Threshold	Partition Size	Inserts to skip	Footprint Left	Updates footprint
1	28.2	5345	32	61	0
2	25.6	4976	1	231	0
.					
.					
L-1	34.7	5032	19	58	0
L	38.9	5923	24	55	2

Figure 2: Partition status table

and B is the working space transfer block size. The rest of the memory is reserved for the updates table.

3. Initialize the partition status table. The table contains L entries. For each entry, the threshold is initially set to N/m. The partition size is set to zero. A coin is flipped and the number of inserts to skip is determined for each partition. The footprint left is set to m.

Running the algorithm. When running the algorithm we perform one pass on the sequence, reading the items one by one. For each item we execute the steps described in the diagram in Figure 3.

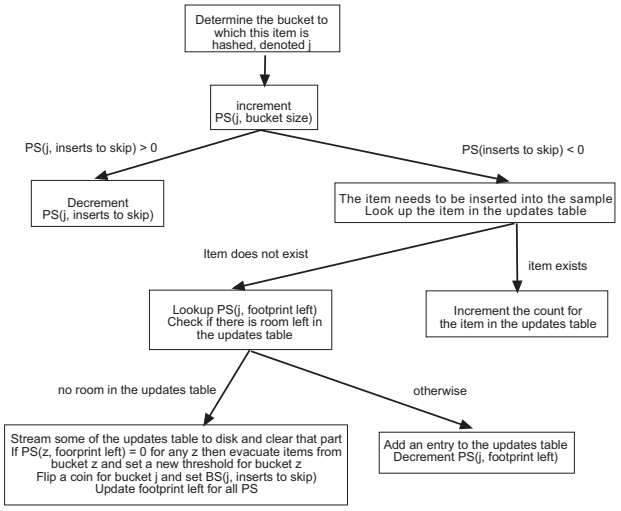

Figure 3: Finding Large Item Sets Using External Memory

Streaming the updates table to disk. Depending on the available memory, the updates table may not be able to maintain all concise samples simultaneously. In this case, as described above, we work with additional working disk space. On disk we save a concise sample for every partition keeping each such concise sample contiguous on disk. To minimize the number of disk accesses to the working disk space, we stream the updates table in a 'lazy streaming' method: whenever the updates table is full we find the partition number whose footprint is the largest in the updates table. This is done by observing the partition status table which stores the number of items from each partition. We then handle only this partition. We load its current concise sample from the

working disk space into memory and update it with all the items belonging to it in the updates table. Note that the partition size was chosen such that each partition could fit in memory. Thus we can load into memory a complete concise sample and insert all the corresponding items into it from the updates table. This results in an efficient streaming of the updates table into the disk. The deferred streaming reduces the number of times the streaming is done, since it results in more sample points being streamed in one update.

Analysis. The above algorithm performs only one pass on the sequence and uses a fixed amount of memory. This is provided that there is enough working disk space to save all the concise samples. Note that if there is not enough disk space we can perform more passes on the sequence, and deal with as many partitions as we can during each such pass. We now compute the total performance of the algorithm in terms of the maximum memory used and the maximum accesses performed to the working disk space. For simplicity, we ignore the memory used for storing the partition status table in our analysis since it is small compared to the size of the updates table and the size left for the concise sample loaded from disk. The partition status table uses a constant amount of memory for each partition and thus we can add an additional αL memory to the total memory to count for the storage of the partition status table.

Theorem 5.1 *Let S be an arbitrary sequence of size N, c_p and c_n be the reporting and non-reporting confidence levels respectively, s_p and s_n be the minimum and maximum reporting frequencies respectively, M be the available memory and B be the working space block transfer size. Let $\phi = \max(t_p/s_p, t_n/s_n)$, let α be the average count of an item in the sample and let r be the concise sample threshold. Then, the above algorithm answers the approximation query for finding frequent items in S while:*

1. *Using a total memory and working disk space of at most: $N\phi/\alpha$*
2. *The total number of disk accesses to the working disk space are at most:*

$$O\left((r\phi)^2/\sqrt{MB}\right)$$

Proof. For the first part, let μ' be the expected number of sample points that we maintain for each concise sample. Thus, the total expected sample points, M', maintained for all the partitions combined is:

$$M' = L\mu' = \frac{N\phi}{\mu'} \cdot \mu' = N\phi$$

where L is the number of partitions computed as shown in Lemma 3.1. Recall that a concise sample stores items as a pair of $\langle itemID, itemcount \rangle$ and thus the storage space for the sample points is factored down by the average count of an item in the sample, α. Thus, the total memory required for storing $N\phi$ sample points is: $N\phi/\alpha$.
For the second part, we need to compute the most efficient way that a memory bound of size M can be used in the algorithm.
Let m be the footprint that we allow for each bucket, where $m = \beta M$, for some $0 < \beta < 1$.
Let L be the number of partitions computed as in Lemma 3.1, $L = N\phi/\mu'$.
Then the memory is divided as follows: m memory is always reserved for one partition to fully reside in memory. Recall that this is required for the streaming process to be able to work efficiently and fully load a concise sample from disk into memory. The rest of the memory is left for the updates table, denoted U. Thus:

$$U = M - m$$

The streaming process is done each time the updates table is full and by our lazy streaming approach at least U/L memory footprint from the updates table is streamed every time. It follows that at least $(U\mu')/(Lm)$ sample points will be streamed every time. As shown above, the total expected number of sample points, M', is: $N\phi$. It follows that the total number of times the updates table is streamed is at most:

$$\frac{N\phi}{U\mu'/Lm} = \frac{N\phi Lm}{U\mu'}$$

In each such streaming we load one partition and thus perform $2\lceil\frac{m}{B}\rceil$ accesses to the working space on disk incurring a total of $\frac{2N\phi Lm}{U\mu'} \cdot \lceil\frac{m}{B}\rceil$ accesses to the working space on disk. Note that all but m are given as constraints to the algorithm and we would thus like to choose m such that the total number of working space disk accesses is minimal. It follows that the total number of disk accesses, ρ, is:

$$\rho \leq \frac{2N\phi Lm}{U\mu'} \cdot \lceil\frac{m}{B}\rceil \leq \frac{2N\phi Lm}{U\mu'} \cdot (\frac{m}{B}+1) = (N\phi)^2 \cdot \frac{2m}{U\mu'^2} \cdot (\frac{m}{B}+1)$$

We would like to minimize the right hand side of the above equation and since $\mu' = \lambda m$, where λ depends on the skewness of the data, it follows that:

$$\min\left(\frac{1}{mU} \cdot (\frac{m}{B}+1)\right) = \min\left(\frac{m+B}{mBU}\right) \Rightarrow m = -B + \sqrt{B^2+MB}$$

Then, the number of accesses to te working space satisfies:

$$\rho \leq \frac{2N\phi Lm}{U\mu'} \cdot \lceil\frac{m}{B}\rceil \leq \left(\frac{N\phi}{\mu'}\right)^2 \cdot \frac{2(m+B)}{B(M-m)} \text{ , } where\ m = -B + \sqrt{B^2+MB}$$

by substituting N/μ' with r and with some algebra manipulation we obtain:

$$\rho = O\left((r\phi)^2/\sqrt{MB}\right)$$

■

We now assume that $M \geq B$ and $F_0 \geq M$. The first assumptions is a common case, since usually the memory is indeed larger than the transfer block size. The second assumption is trivial, since in the case of $M \geq F_0$ one concise sample can be maintained in memory to count exactly the frequencies of all items.

Theorem 5.2 *Let S be an arbitrary sequence of size N containing F_0 distinct values, let M be the available memory, and let $s = \min(s_p, s_n)$ where s_p and s_n are the minimum and maximum reporting frequencies respectively. Then, under the assumption that $M \geq B$ and $F_0 \geq M$, we can maintain, using the above algorithm, a concise sample fulfilling the required confidence levels with no more than*

$$O\left(\frac{NF_0}{s^2M^3}\right)$$

accesses to the working space in terms of passes over S.

Proof. We use Theorem 5.1. Recall Section 2.2 where we showed that r satisfies:

$$r \simeq N/m' \leq \frac{2NF_0}{(1-\epsilon)M(M+2F_0)}$$

where ϵ is the fraction of the sample size from N such that $m' \geq \epsilon N$. When $\epsilon \geq 1/2$, the concise sample counts at least half of the sequence and the problem is trivial. Notice that ϕ was defined in Theorem 5.1 as $\phi = \max(t_p/s_p, t_n/s_n)$. While t_p and t_n are typically small constants, s_p and s_n can be typically large, depending on the frequencies in which we are interested. Then, by substituting r with the above bound in theorem 5.1 and by taking $\epsilon \leq 1/2$ we get that the total number of disk accesses to the working disk space is at most:

$$O\left(\left(\frac{2NF_0\phi}{sM(M+2F_0)}\right)^2/\sqrt{MB}\right)$$

By employing our assumptions and by dividing with N/B we get the number of working space accesses in terms of passes performed over the whole sequence:

$$O\left(\frac{NF_0}{s^2M^3}\right)$$

■

Note that in the case where $F_0 = cM$, the above bound is $O(N/(sM)^2)$ which means that for all practical cases, the number of accesses to the local disk is constant per disk block of the sequence.

5.1 Extension for working on-line

We describe an extension for the algorithm to deal with the case that the size of the sequence is not known a priori. This extension may be used when the algorithm is required to work fully online, in a setting where the sequence is streamed and its size is hence not known in advance.

The idea is to initially estimate a size for the sequence and run the algorithm from the previous section in the same manner while complying to the confidence, frequencies and memory constraints. We keep a counter to count the actual size of the sequence. Whenever the actual sequence size exceeds the estimate we increase our estimate by some factor. In order to still comply with the given constraints it is then necessary to increase the number of partitions but without rereading the sequence. In any case, we derive a new hash function to partition the sequence into the increased number of partitions. We now rearrange all the items currently inserted into all the concise samples into the new increased number of concise samples. This can be done by taking all items currently inserted and sorting them according to the new hash function, using our efficient in-place sorting algorithm. Note that now, since the number of concise samples has increased, each concise sample will contain less items and thus the footprint left for each concise sample is increased. As we stream the sequence, we perform as many such expansions as necessary. When we finish streaming the sequence, the combination of all the concise samples is the same as it was if the size of the sequence was known a priori.

The above method relies on the fact that no matter what the final actual size of the sequence is, the algorithm maintains the answer for the approximation query at any given point. In fact, this property of the algorithm allows us to report approximations for the sequence in any point of the algorithm. In the case that the order in which the sequence is streamed is random, a report at a given point can be fairly accurate.

5.2 Extension for working with tuples

We show how to extend our algorithms to work with tuples. The input now consists of records, each effectively representing more than one item and we are interested in frequent items for a given tuple size, w. For example, we may be receiving records of size 5 and we are interested in the frequent items that are triples. Each such record represents 10 triples. This is a common case in many application such as market basket analysis in data mining, clustering and more.

Note that depending on w, we have a blowup in the size of the sequence. Let b_i be the tuple size of the $i-th$ element in the sequence of size N. Then for any w, the number of tuples in the sequence of size w is: $\sum_{i=1}^{N} \binom{b_i}{w}$.

If we were to expand each tuple into all tuples of size w we are looking for, then the problem would reduce to the exact problem outlined in Section 3. However, the blowup in the number of items can be a large factor of N and thus in database systems it is infeasible to expand every tuple and thus materialize the sequence. Therefore we can not use the in-place algorithm since it requires the of the sequence. As it turns out, the one-pass algorithm is suitable for this case and requires minor modifications to fit here.

We run the one-pass algorithm in the same way using working disk space external memory as described in Section 5. The only modification is this: for every item encountered, extract all the tuples of size w which it represents. Then subject each such tuple to the same process that each item goes through in the algorithm of Section 5.

Note that the algorithm reduces the problem to the exact one solved by the algorithm in Section 5 and therefore indeed answers it under the specified frequencies, confidence levels and memory constraints. This is due to the fact that the sequence is handled as if it were expanded on disk. In fact, the only parameter affected in the algorithm from Section 5 by this approach is the size of the sequence.

To analyze the performance of the algorithm, note that the only difference from Theorem 5.1 is in the size of the sequence N which is now expanded as shown above. Hence we conclude that this extended algorithm can answer the approximation query under the same constraints as in Theorem 5.1 while:

1. Using a total memory and working disk space of at most: $N'\phi/\alpha$
2. The total number of disk accesses to the working disk space are at most:

$$O\left((r\phi)^2/\sqrt{MB}\right)$$

where α is the average count of an item in the concise sample, r is the threshold of the concise sample and $N' = \sum_{i=1}^{N} \binom{b_i}{w}$. Note that if for all i, $b_i = b$, then $N' = N \cdot \binom{b}{w}$. The blowup factor $\binom{b}{w}$ may be quite significant. It should be clear that r will now depend on N' as opposed to N as in Theorem 5.1.

6 Deriving exact answers

In this section we show how our algorithms can be extended to give exact answers and we show the performance costs involved. We also show, in Section 6.3, how the sorting algorithm can be used to gradually refine the approximation until finally, upon request, derive at exact answers.

In general, both our algorithms may suffer from false positives (items whose frequency is below s_n that are reported) and false negatives (items whose frequency is above s_p that are missed). We present techniques for eliminating false positives and regaining the false negatives. Depending on

the type of application, it might be useful to only eliminate false positives or only regain false negatives and this can be done without a performance degradation, since these are two separate processes. Also note that in some settings it may be advantageous to initially eliminate the false positives and report, with high probability, a c_p fraction of the items above s_p and only then to regain the false negatives and report them separately.

6.1 Eliminating false positives

This can be done in both algorithms in an additional pass over the sequence. In both algorithms we consider only the items that were reported as exceeding a frequency of s_p. We perform an additional pass over the sequence and measure the exact count of all these items. Items whose count is below s_p are deleted in this post-processing phase from the set of items that we report.

For the sorting based algorithm, we use the sequence that we already sorted. We perform one pass and for each segment of the sequence (each segment is a partition into which the sequence was sorted) consider only those items reported for that segment. Since the number of items never exceeds the available memory (recall that this is how the concise samples were designed) this process is very efficient since it requires no additional working space.

For the algorithm that uses working space, we have the items that we wish to measure residing on blocks on the working space on disk. Note that we wish to measure the count of only those items that were reported. If all the reported items fit into memory then we can initially load these items and perform one pass over the sequence while measuring the count of these items. However, if this is not the case then we have two alternatives:

1. perform one pass over the data and swap blocks on working space as required.
2. perform more than one pass over the data while in each pass we load as many reported items as we can into memory.

There are tradeoffs, of course, and the best approach is probably using a hybrid approach. The exact tradeoffs are subject of an on going research that we are currently conducting.

In any case, by eliminating the false positives we report only items that indeed exceed the frequency threshold s_p. As shown in Lemma 2.1, we expect that a c_p fraction of all items that indeed exceed s_p to be reported.

6.2 Regaining false negatives

After eliminating the false positives we are left, with a high probability, with at least a c_p fraction of the items that indeed exceed the frequency threshold. When trying to regain those items that were missed, we can disregard all those items that are already reported and thus, with a high probability, we are seeking for no more than a $1 - c_p$ fraction of the total items that exceed s_p.

To regain the items that were missed, we run the same techniques as in [FSGM+98], except that they can now be run with enhanced performance. This is due to the fact that those techniques hash items into buckets and eliminate buckets whose count did not exceed the frequency threshold. Thus, the more frequent items are eliminated, the better these algorithms will perform. Hence, by disregarding frequent items which we already identified, we can use these algorithms with enhanced performance.

We can clearly see a tradeoff in using this technique. If we initially execute our algorithm in search for lower frequencies, we can achieve higher performance when executing the algorithms from [FSGM+98] but this degrades performance in our algorithms. We raise the question of deriving at an optimal algorithm and study this question in our on going research.

6.3 Gradually refining the approximation

We show how the in-place algorithm can be used to gradually refine the approximation, until finally, upon request, deriving at exact answers. The sorting phase of the in-place algorithm may perform more than one pass over the sequence when sorting it. The actual number of passes depends over the available memory for the sorting process. Note that by the nature of the in-place sorting algorithm, after the $i-th$ pass on the sequence, the sequence is sorted into $(M/B)^i$ partitions. The sorting ends, of course, when $(M/B)^i = L$ where L is the number of partitions into which we were aiming the sort.

We observe that the in-place sorting algorithm actually sorts the sequence gradually, until finally the sequence is sorted into the required number of partitions. We take advantage of this feature to derive an approximation algorithm, that gradually refines its answers until finally it can even reach at exact answers. After each iteration of the sort, we perform an additional pass over the sequence and maintain a concise sample for each partition. Since the sequence is still not sorted into the required number of partitions, it is of course not guaranteed that the samples maintained will satisfy the required approximation parameters. However, we can report the frequent items that we find and even the confidence levels that are currently satisfied. After each iteration, the number of partitions is further refined and hence, the accuracy of each concise sample increases, resulting in a better approximation for each iteration of the sorting algorithm. If an exact answer is finally required, then the sorting phase can further refine the sort until each partition is small enough to be fully counted in memory, resulting in an exact answer.

Clearly, executing a concise sample pass after each iteration of the sorting algorithm will degrade the total performance of the algorithm, since we perform more passes over the sequence. On the other hand, in settings in which gradual improving approximations are handy, this algorithm is useful. We can even give the user control over the number of sorting iterations to perform before executing a concise sample phase. This will give the user better control over the performance of the algorithm.

7 Experiments

We conducted a number of experiments comparing the accuracy and overheads of the algorithms introduced in Sections 4 and Section 5. We tested with two types of data: synthetic data generated by the IBM test data generator (http://www.almaden.ibm.com/cs/quest) and U.S. census data (http://icg.fas.harvard.edu/~census). All experiments were run on a Pentium2, 300 Mhz, 128MB RAM machine.

The census data used consists of $\sim 32K$ records with 15 columns in each record. We have tested the algorithm from Section 5 while identifying pairs of frequent items in the data. The total number of pairs was $\sim 3.15M$ including $\sim 320K$ different pairs. Since the algorithm computes approximates for the real answers, we are interested in testing the accuracy of the results as well as the performance. Figure 4 shows the accuracy of the results when the algorithm was run with different confidence levels and with different frequency thresholds. The results show that the approximate answer was correct within less than 0.5% of error. This is true even when the algorithm was run with a $c_p = 50\%$ confidence level. The reason is that our computations for confidence levels ignore the skewness of the data, i.e. the coefficient is determined only by items whose frequency is equal to the minimum frequency threshold specified. Items exceeding this frequency will actually have a much higher confidence at the computed coefficient. We can therefore conclude that performance can be enhanced by using less working disk space if the skewness of the data is known a priori.

Figures 5 and 6 show the number of false reports and the number of in-gap reports for each of the various confidence levels while the gap (the relative difference between the upper frequency threshold, s_p, and the lower frequency threshold, s_n) was set to 0.4. The number of false reports was much less than expected for the different confidence levels, due to a similar assumption in the calculations as explained above. Nevertheless, it can be observed that at lower confidence levels the number of false reports increased, as expected.

Since the algorithm completes only one pass on the data, the performance is bounded by the number of I/Os performed and the amount of working disk space used. Figures 7 and 8 show, respectively, the total number of disk accesses and the total footprint consumed. As can be seen, there are major differences in the number of disk accesses and working space consumed between the various confidence levels. The results encourage us to further research in the direction of adaptive techniques since we can see that the amount of working space used is less than expected. This is due to the fact that when calculating the coefficients we ignore the fact that a concise sample is efficient in storing sample points, i.e. the fact that the average count for an item in the sample exceeds 2 causing the total footprint consumed to fall below the expected. Again, a better knowledge of the skewness of the data could help in devising more efficient algorithms.

Figures 10 and 9 show the performance degradations as the gap is reduced. That is, we show how the footprint consumed and the number of disk accesses vary as a factor of the gap between the upper frequency threshold and the lower frequency threshold It can be observed that as the gap is reduced the performance in both measures degrades. This is expected, since we need a larger sample and thus a greater number of buckets as the range is modified. We can also see how these changes are intensified at each of the confidence levels.

8 Conclusions

We have proposed the problem of computing approximate iceberg queries and developed efficient algorithms to compute them as well as algorithms for computing exact iceberg queries. We have realized our algorithms by a hash based partitioning technique: the data set is partitioned using a hash function into value independent subsets, resulting with suitably smaller independent sub-problems that can be handled effectively with guaranteed performance. The answer to the query is then the combined answer of each of the individual partitions. Our algorithms use as a basic ingredient concise samples as presented in [GM98a].

More specifically, we realized two algorithms. The first algorithm is based on sorting the sequence into the required number of partitions. Then, one pass over the sequence scans each partition at a time, allowing to hold in memory the concise sample of only one sub-problem at a time, without conflicts. The sorting is based on an integer-sorting algorithm derived in a companion paper [MS98], which is proved to be substantially faster than general sorting, when the number of distinct keys to sort by (hence, in this application, the number of partitions) is relatively small compared to the size of the sequence. The integer sorting algorithm is an in-place algorithm, requiring no additional working space.

Our second algorithm computes the approximation query while performing only one pass over the sequence. Hence, it can be applied on-line for streaming data, without accessing a data base, and without materializing data sets that are specified implicitly. The algorithm takes the partitioning approach as well but rather than sorting the sequence and deal with each partition separately, it maintains all concise samples (one for each partition) simultaneously. Since memory constraints may prevent all concise samples to be held together in main memory, the algorithm resorts to additional working disk space. The main memory then serves as a buffer for the concise samples

which now reside on disk. As long as the memory suffices, it holds all the inserts that were supposed to be to the concise samples, hence delaying their update and reducing the number of I/Os to the disk. Once full, we stream the largest partition from memory into the disk. We have also shown how memory can be used in an optimal way in this setting.

We gave rigorous analysis for the guarantees of both algorithms. It is important to note, however, that as can be seen in our experiments, the analysis is for a worst-case distribution of frequencies in the input data set (in which most frequencies are near the threshold frequencies). In most cases, the distribution of the input data set would be more favorable, and the performance (both in terms of time execution, accuracy and confidence) would be substantially better. This is supported by experiments on U.S. census data and on synthetic data generated by the IBM test data generator. We are thus encouraged to investigate adaptive techniques that can exploit the more favorable data distributions. We are currently conducting extensive experiments on larger data sets. The results will be reported in the final version of this paper.

Comparison between the two algorithms

In analyzing both algorithms we notice advantages and disadvantages when applying each of them. The bucket sorting based algorithm performs the approximation in one pass over the sequence after initially sorting the sequence and without the need for additional working disk space. Thus, its performance depends primarily on the sorting performance. The sorting performance depends on the available memory, which affects the number of buckets that can be sorted in each pass on the sequence, and hence affects the total number of I/Os performed. As the cost of an I/O operation to the sequence decreases, this algorithm performs better. The algorithm also performs an in-place sort and thus does not require additional storage space. In very large databases, this can be crucial. In addition we have shown how to extend this algorithm to provide quick and rough estimates for iceberg queries, and gradually refine the estimates until reaching satisfactory bounds, or complete and exact solutions. This extension is very intuitive by the nature of the bucket sorting algorithm, which sorts the sequence gradually: after the the $i-th$ pass over the data set, the sequence is fully sorted into $(M/B)^i$ keys. We have shown that we can utilize this feature of the sorting algorithm and after every sorting pass, execute a pass that maintains a concise sample for each partition. This results in gradually improved approximations that could be refined until an exact answer is achieved.

The one pass algorithm computes the approximation while performing only one pass over the sequence without materializing the sequence. Thus, it can be used as an online algorithm, constantly maintaining the approximation as it reads the records. In setups in which the sequence can only be read once (for example, in routers, host computer for database applications and proxy servers for web traffic) only this algorithm is applicable. There are also settings in which the sequence cannot be materialized and thus sorting is not an option and only this algorithm can be used. Also, when the cost of an I/O operation to the local working space is much cheaper than an I/O operation to the sequence, the performance will be bound by I/O operations performed on the sequence in which cases this algorithm performs the least I/Os possible on the sequence (only one pass).

We conclude that in choosing between the algorithms, one must notice the settings in which these algorithms are to be run. When sorting is not applicable, or when only one pass is allowed on the sequence, or when an online algorithm is required, the one pass algorithm is the only choice. When this is not the case, the difference in costs between I/O operations to the sequence and I/O operations to the working space should be considered, and in most cases, the sorting based algorithm will result in better performance

Parallel algorithm using external memory

We consider executing our algorithms in parallel in the shared-nothing model, in which we have a number of CPUs, each with a working disk space and available memory of its own. The sorting based algorithm works in two phases. In the first phase, the sequence is sorted according to the number of partitions required. In the second phase, one pass is performed over the sequence, in which a concise sample is maintained separately for each partition while reusing memory. A natural and efficient way to extend this algorithm to work in parallel is to initially perform one sorting pass over the sequence and sort the sequence into η partitions, where η is the number of processors available. Then, each processor can execute the sorting based algorithm independently without the need for any synchronization between the processors. Each processor reports the results and the combined reports provides the answer to the approximation query. Besides the initial pass that partitions the sequence into η partitions, this parallel approach takes full advantage of all the available processors, resulting in a substantial performance gain.

The one pass algorithm yields very intuitively to parallelism. Observe that clustering the sequence into partitions reduces the problem of finding frequent items in the sequence to finding large items in each partition and combining the results. Fortunately, handling a partition is independent of other partitions since we use a hash based partitioning techniques that combines all instances of the same item into one partition. Assume that we have a sequence S of size N, η processors each with a memory bound M, and we are given the same approximation parameters as usual. We derive a hash function that maps all items in the sequence into $1 \ldots \eta$. In a setting in which only one processor reads the sequence, each item is transferred to its corresponding processor. In a setting in which each processor can independently read the sequence, each processor handles only those items that are hashed to it and ignores the rest. In any case, each processor executes the same one pass algorithm as detailed in Section 5. Since there is no need for any synchronization between the η processors, the parallel algorithm can improve the performance in direct proportion to the number of processors available.

Future work will involve a refined analysis which takes into account the statistical characteristic of the data, with the goal of obtaining more accurate predictions, rather than bounds that apply in worst case distributions. We are also looking into incorporating count samples, suggested in [GM98a], as alternatives to concise samples, and study their merits and limitations.

References

[AIS93] R. Agrawal, T. Imilineski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 207–216, May 1993.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. 20th International Conf. on Very Large Data Bases*, pages 487–499, September 1994.

[BGM97] A. Broder, S.C. Glassman, and M.S. Manasse. Syntactic clustering of the web. In *Sixth International World Wide Web Conference*, April 1997.

[BMUT97] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 255–264, May 1997.

[Bro97] A. Broder. On the resemblance and containment of documents. Technical Report DIGITAL Systems Research Center Tech. Report, Tech. Report, 1997.

[BSL92] D. Barbara, K. Salem, and R.J. Lipton. Probabilistic diagnosis of hot spots. In *Proceedings of the Eight Conference on Data Engineering*, February 1992.

[CW79] L. Carter and M. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[FJS97] C. Faloutsos, H. V. Jagadish, and N. D. Sidiropoulos. Recovering information from summary data. In *Proc. 23rd International Conf. on Very Large Data Bases*, pages 36–45, August 1997.

[FSGM+98] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *Proc. 24th International Conf. on Very Large Data Bases*, pages 299–310, August 1998.

[GM98a] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 331–342, June 1998.

[GM98b] P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. To appear in *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, AMS., 1998.

[GMP97] P. B. Gibbons, Y. Matias, and V. Poosala. Aqua project white paper. Technical report, Bell Laboratories, Murray Hill, New Jersey, December 1997.

[HHW97] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 171–182, May 1997.

[Knu73] D.E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, 1973.

[MS98] Y. Matias and E. Segal. Efficient bundle sorting. Manuscript, October 1998.

[PCY95] J.S. Park, M.S. Chen, and P.S. Yu. An effective hash based algorithm for mining association rules. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 175–186, May 1995.

[SA95] R. Srikant and R. Agrawal. Mining generalized association rules. 1995.

[Toi96] H. Toivonen. Sampling large databases for association rules. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, 1996.

[TPC] TPC-Committee. Transaction processing council (TPC). `http://www.tpc.org`.

[Vit85] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.

A Figures

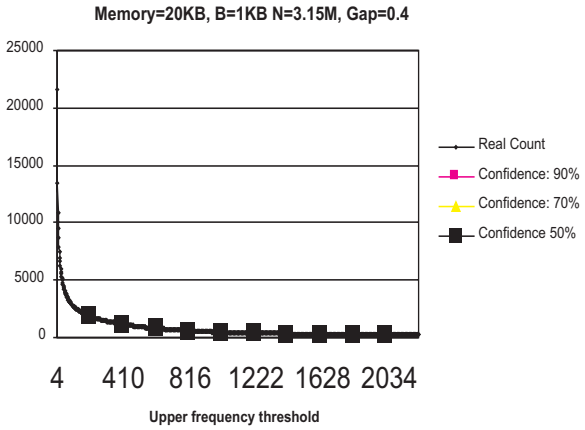

Figure 4: True reports vs. real frequent items. The x-axis is the upper frequency threshold s_p and the y-axis is the number of frequent items found. As can be observed, in all 3 confidence levels the number of frequent items detected matches the real number of frequent items

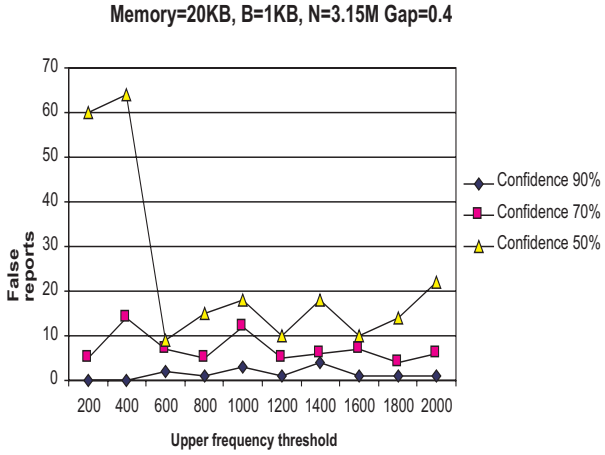

Figure 5: False Reports. The x-axis is the upper frequency threshold s_p and the y-axis is the number of false reports. As expected, the number of false reports increases with lower confidence levels

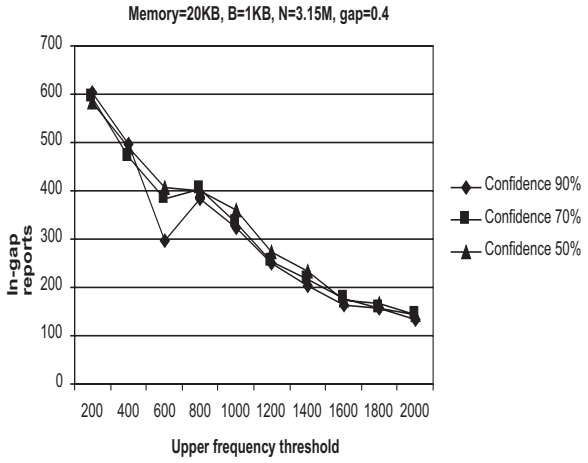

Figure 6: In-Gap Reports. The x-axis is the upper frequency threshold s_p and the y-axis is the number of reports that fall between the true and the flase reports.

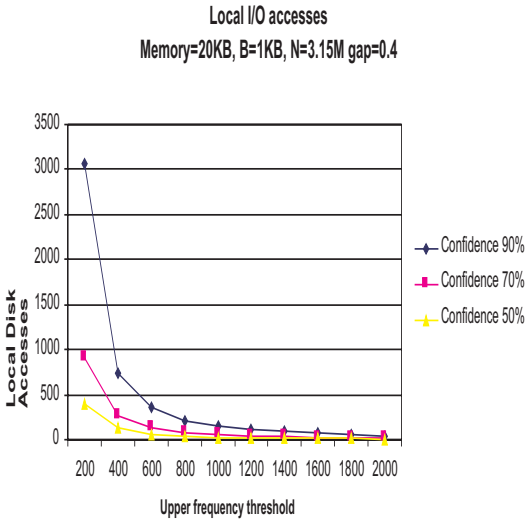

Figure 7: Local disk accesses. The x-axis is the upper frequency threshold s_p and the y-axis is the number of accesses to the working space

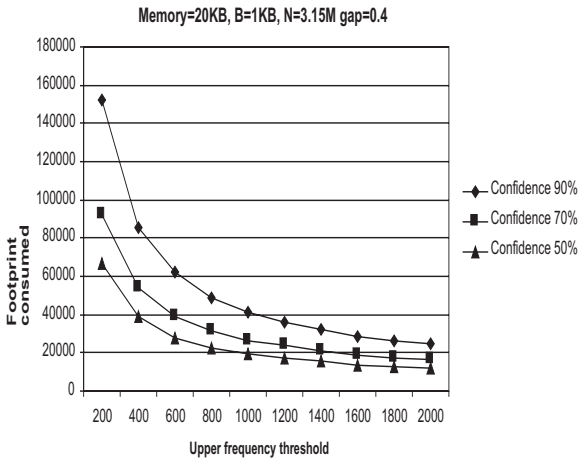

Figure 8: Footprint consumed. The x-axis is the upper frequency threshold s_p and the y-axis is the footprint consumed for each of the confidence levels

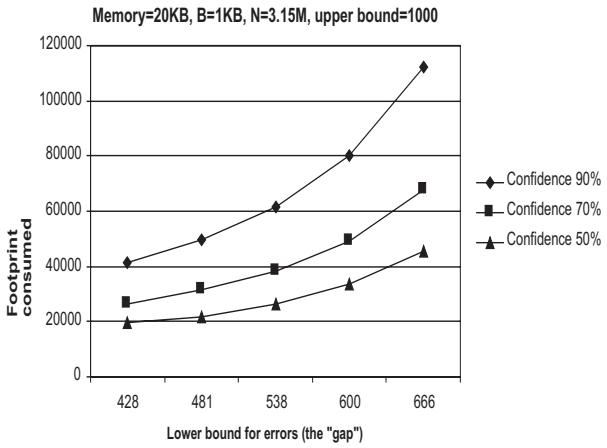

Figure 9: Footprint consumed for various gaps. The x-axis is the lower frequency threshold s_n when the upper frequency was set to 1000. This changes the gap. The y-axis is the footprint consumed in the various gap levels and for different confidence levels

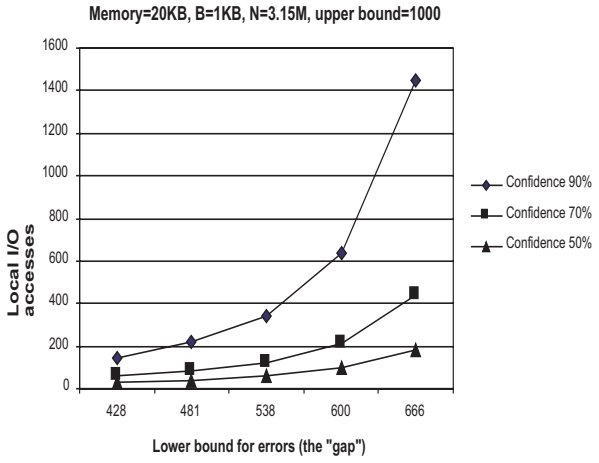

Figure 10: Local disk accesses for various gaps. The x-axis is the lower frequency threshold s_n when the upper frequency was set to 1000. This changes the gap. The y-axis is the accesses to the working space in the various gap levels and for different confidence levels