Computing Iceberg Queries Efficiently*

Min Fang†, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, Jeffrey D. Ullman
Department of Computer Science, Stanford, CA 94305.
{fangmin, shiva, hector, motwani, ullman}@db.stanford.edu

Abstract

Many applications compute aggregate functions over an attribute (or set of attributes) to find aggregate values above some specified threshold. We call such queries *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). Such iceberg queries are common in many applications, including data warehousing, information-retrieval, market basket analysis in data mining, clustering and copy detection. We propose efficient algorithms to evaluate iceberg queries using very little memory and significantly fewer passes over data, when compared to current techniques that use sorting or hashing. We present an experimental case study using over three gigabytes of Web data to illustrate the savings obtained by our algorithms.

1 Introduction

In this paper we develop efficient execution strategies for an important class of queries that we call *iceberg queries*. An iceberg query performs an aggregate function over an attribute (or set of attributes) and then eliminates aggregate values that are below some specified threshold. The prototypical iceberg query we consider in this paper is as follows, based on a relation *R(target1, target2,..., targetk, rest)* and a threshold T.

```
SELECT  target1, target2, ..., targetk, count(rest)
FROM    R
GROUPBY target1, target2, ..., targetk
HAVING  count(rest) >= T
```

*This work was partially supported by the Community Management Staff's Massive Digital Data Systems Program, NSF grant IRI–96–31952, an IBM Faculty Partnership Award, NSF Young Investigator Award CCR–93–57849, and grants of IBM, Hitachi Corp., Mitsubishi, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

†Currently affiliated with Oracle Systems, Redwood Shores, CA.

Proceedings of the 24th VLDB Conference
New York, USA, 1998

target1	target2	rest
a	e	joe
b	f	fred
a	e	sally
b	d	sally
a	e	bob
c	f	tom

Table 1: Example relation R.

If we apply the following iceberg query on relation R in Table 1, with $T = 3$ (and $k = 2$), the result would be the tuple $\langle a, e, 3\rangle$. We call these iceberg queries because relation R and the number of unique *target* values are typically huge (the iceberg), and the answer, i.e., the number of frequently occurring targets, is very small (the tip of the iceberg).

Many data mining queries are fundamentally iceberg queries. For instance, market analysts execute *market basket* queries on large data warehouses that store customer sales transactions. These queries identify user buying patterns, by finding item pairs that are bought together by many customers [AS94, BMUT97]. Target sets are item-pairs, and T is the minimum number of transactions required to *support* the item pair. Since these queries operate on very large datasets, solving such iceberg queries efficiently is an important problem. In fact, Park et al. claim that the time to execute the above query dominates the cost of producing interesting *association rules* [PCY95]. In this paper, we concentrate on executing such iceberg queries efficiently using compact in-memory data structures. We discuss more examples of iceberg queries in Section 2.

The simplest way to answer an iceberg query is to maintain an array of counters in main memory, one counter for each unique target set, so we can answer the query in a single pass over the data. However as we have already indicated, answering the query in a single pass is not possible in our applications, since relation R is usually several times larger than the available memory (even if irrelevant attributes are projected out as early as possible). Another approach to answer an iceberg query is to sort R on disk, then do a pass over it, aggregating and selecting the targets above the threshold. If the available memory is small relative to the size of R, the sorting can take many passes over the data on disk. For instance, if we use merge-sorting, we produce $|R|/M$ sorted runs, where M is the number of tuples that fit in memory. Then we need $\log_M |R|/M$ merge passes to produce the final sorted run. For each of these passes we need to read and write the entire relation R (or at least all the values for the target attribute). We encounter similar problems if we use other popular techniques such as early aggregation [BD83], or hashing based aggregation.

Until now, we have assumed R is materialized. However, in many cases R may be too large to be materialized, even on disk. For instance, in the market basket application, the input data is often not R itself, but a set of transaction records. Each such record describes a collection of items bought by a customer, and corresponds to multiple R records. For example, suppose we are interested in pairs of items that are frequently bought together in a store, and say a customer bought items $\{a, b, c\}$. Then R would contain tuples $[a, b]$, $[a, c]$, $[b, c]$, representing each association between pairs of items. In general, if the average number of items a customer buys is n, then each customer record generates $C(n, 2) \approx \frac{n^2}{2}$ tuples in R. We can see that even if the initial data with customer transactions is small[1], materializing R may not be feasible due to the quadratic increase in size over the initial input. The situation may get worse when the analyst wants to find popular item triples and quadruples. Thus, when R is very large, it will be useful to execute the iceberg query over the *virtual* relation R without explicitly materializing R, as traditional techniques based on sorting or hashing would require.

The primary contributions of this paper are three-fold:

1. We identify iceberg queries as fundamental data mining queries, and discuss applications where icebergs appear either directly, or as sub-queries in more complex queries. Iceberg queries are today being processed with techniques that do not scale well to large data sets, so it is crucial to develop better techniques.

2. We propose a variety of novel algorithms for iceberg query processing. Our 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. Our techniques avoid sorting or hashing R, by keeping compact, in-memory structures that allow them to identify the above threshold targets. In cases where R is not materialized, we show how to perform the iceberg computation without materializing R.

3. We evaluate our algorithms using a "case-study" approach for three different applications (with real data) and queries. Our results show that the new algorithms can efficiently handle much larger iceberg problems than current techniques. The case study also serves to illustrate the tradeoffs involved in choosing one strategy over another, depending on available system resources (such as size of disk and main memory).

[1] In many cases, input data for WalMart-like stores runs into hundreds of gigabytes.

The rest of the paper is structured as follows. In Section 2 we discuss a few examples of iceberg queries. In Section 3 we present two simple algorithms that can be used to execute iceberg queries. In Section 4 we propose three hybrid algorithms that combine the advantages of the two simple algorithms, in different ways. In Section 5 we propose several orthogonal techniques to optimize the hybrid strategies. In Section 6 we propose some extensions to our algorithms. In Section 7 we evaluate our techniques on three case studies, using over three gigabytes of data – the size of R for some of these scenarios, if materialized, will require 50 to 100 gigabytes of storage. We conclude in Section 9 with some directions for future research.

2 Why are iceberg queries important?

We now illustrate using a few examples why executing iceberg queries efficiently is important, and why traditional techniques such as sorting and hashing can lead to very high query times and inordinately large disk requirements.

EXAMPLE 2.1 PopularItem Query
Consider a TPC-D benchmark [TPC] style relation `LineItem` with attributes `partKey`, the key for parts being sold, `price`, the price of the corresponding item, and `numSales`, the number of units sold in a transaction, in `region`, the area where the part is being sold. The following query computes the keys of popular items and regions, where the item's revenues in the region exceed one million dollars.

```
CREATE VIEW PopularItems as
   SELECT    partKey, region, SUM(numSales * price)
   FROM      LineItem
   GROUP BY  partKey, region
   HAVING    SUM(numSales * price) >= $ 1,000,000
```

It is easy to see that if we apply current techniques such as sorting, to sort the `LineItem` relation to perform the aggregation, the response time for the above query is large – even if most of the items in `LineItem` are not very popular, and have very small revenues. Of course, if the criterion for selecting an item were 10$ of revenue rather than one million dollars, the sorting approach may be best since many items will satisfy the query. We intuitively see that traditional techniques such as sorting and hashing are "over kill" solutions and are not *output sensitive*, in that they perform the same amount of work independent of how small the query's output is. They do not use the given threshold to execute the query faster. Rather, they first perform the aggregation and later apply the thresholding. □

EXAMPLE 2.2 DocumentOverlap Query

Web-searching engines such as AltaVista *cluster* web documents based on "syntactic similarity" of documents [Bro97, BGM97], The goal of clustering is to develop better web crawlers by identifying documents that are replicated or are near-replicas of other documents (such as JAVA 1.1.3 manuals and FAQs [SGM98]).

The engines break up each web document into a set of *signatures*, such as hashed 8-byte integers of sequences of words, or sentences. Then they maintain a relation *DocSign* with tuples $\langle d_i, c_i \rangle$ if document d_i contains signature c_i. Then they identify a document pair to be a copy if they share more than $T2$ signatures in common using the following query.

```
CREATE VIEW DocumentOverlaps
  SELECT    D1.doc, D2.doc, COUNT(D1.chunk)
  FROM      D1 as DocSign, D2 as DocSign
  WHERE     D1. chunk = D2. chunk  AND
            D1.doc NOT = D2.doc
  GROUP BY  D1. doc, D2. doc
  HAVING    COUNT(D1.chunk) >= T2
```

Currently, the DEC prototype [Bro97, BGM97] uses sorting to execute the above self-join, as follows. They first sort *DocSign* on the signatures so that for a given signature s_k, all tuples $\langle d_i, s_k \rangle$ such that document d_i contains s_k will be contiguous. Then for each pair of the form $\langle d_i, s_k \rangle$ and $\langle d_j, s_k \rangle$ they explicitly materialize relation *SignSign* of the form $\langle d_i, d_j \rangle$, indicating that d_i and d_j share a signature in common. Then they sort *SignSign*, so that all tuples for a given document pair are contiguous. Finally, they sequentially scan *SignSign* and count the number of document pairs that occur more than $T2$ times in *SignSign* – these document pairs have more than $T2$ signatures in common.

The above process explicitly materializes *SignSign* (termed R in our discussions), before it sorts *SignSign* and thresholds on $T2$. As we shall see in one of our case-studies, this materialized relation has very large storage requirements. In fact, for a small input *DocSign* of size 500 megabytes, this relation grew to about 40 gigabytes, even though the final answer to the query was only one megabyte worth of document pairs! □

Iceberg queries also arise in many information retrieval (IR) problems. For instance, IR systems often compute *stop words*, the set of frequently occuring words in a given corpus, to optimize query processing and to build inverted indices. Such a query also has the "iceberg" property. IR systems also sometimes compute sets of frequently co-occurring words, and use these to help users construct queries. For instance, the pairs "stock market," "stock price," and "chicken stock" may occur often in a collection of documents. If the user enters the word "stock" in a query, the system may suggest "market, "price," and "chicken" as useful words to add to the query to distinguish the way in which "stock" is used. Computing co-occurring words again involves an iceberg query, where target-sets are pairs of words. We will study this application again in more detail in our experimental case-study.

From the above illustrative examples, we see that iceberg queries occur commonly in practice, and need to be executed carefully so that query times and temporary storage requirements are output sensitive.

3 Techniques for thresholding

For simplicity, we present our algorithms in the next few sections in the context of a materialized relation R, with $\langle target, rest \rangle$ pairs. We assume for now we are executing a simple iceberg query that groups on the single target in R, as opposed to a set of targets. As we will discuss later, our algorithms can be easily extended for unmaterialized R as well as to multiple target sets.

We start by establishing some terminology. Let V be an ordered list of targets in R, such that $V[r]$ is the r^{th} most frequent target in R (r^{th} highest rank). Let n be $|V|$. Let $Freq(r)$ be the frequency of $V[r]$ in R. Let $Area(r)$ be $\sum_{i=1}^{r}[Freq(i)]$, the total number of tuples in R with the r most frequent targets.

Our prototypical iceberg query (Section 1) selects the target values with frequencies higher than a threshold T. That is, if we define r_t to be $\max\{r|Freq(r) \geq T\}$, then the answer to our query is the set $H = \{V[1], V[2], \ldots, V[r_t]\}$. We call the values in H the *heavy* targets, and we define L to be the remaining *light* targets.

The algorithms we describe next answer the prototypical iceberg query, although they can be easily adapted to other iceberg queries. In general, these algorithms compute a set F of *potentially* heavy targets or "candidate set", that contains as many members of H as possible. In the cases when $F - H$ is non-empty the algorithm reports *false positives* (light values are reported as heavy). If $H - F$ is non-empty the algorithm generates *false negatives* (heavy targets are missed). An algorithm can have none, one, or both form of errors:

1. **Eliminating False Positives:** After F is computed, we can scan R and explicitly count the frequency of targets in F. Only targets that occur T or more times are output in the final answer. We call this procedure *Count(F)*. This post-processing is efficient if the targets in F can be held in main-memory along with say 2 – 4 bytes per target for counting. If F is too large, the efficiency of counting deteriorates. In fact, as $|F| \to n$, the post-processing will take about the same time as running the original iceberg query.

2. **Eliminating False Negatives:** In general, post-processing to "regain" false negatives is very inefficient, and may in fact be as bad as the original problem. However, we can regain false negatives efficiently in some high skew cases where most R

tuples have target values from a very small set.[2] In particular, suppose that we have obtained a partial set of heavy targets $H' = F \cap H$, such that most tuples in R have target values in H'. Then we can scan R, eliminating tuples with values in H'. The iceberg query can then be run on the remaining small set of tuples (either by sorting or counting) to obtain any heavy values that were missed in H'.

We now present two simple algorithms to compute F, that we use as building blocks for our subsequent, more sophisticated algorithms. Each algorithm uses some simple data structures such as lists, counters, and bitmaps for efficient counting. For ease of presentation, we assume that the number of elements in each structure is much smaller than $|V|$, and that all structures fit in main memory. In Section 7 we evaluate the memory requirements more carefully.

3.1 A Sampling-Based Algorithm (SCALED-SAMPLING)

Sampling procedures are widely adopted in databases [HNSS96]. (See [Olk93] for a good discussion of sampling techniques to obtain unbiased samples efficiently.) We now consider a simple sampling-based algorithm for iceberg queries. The basic idea is as follows: Take a random sample of size s from R. If the count of each distinct target in the sample, scaled by N/s, exceeds the specified threshold, the target is part of the candidate set, F. This sampling-based algorithm is simple to implement and efficient to run. However, this algorithm has both false-positives and false-negatives, and removing these errors efficiently is non trivial, as we discussed above. We will show how to remove these errors using our HYBRID algorithms in the next section.

3.2 Coarse counting by bucketizing elements (COARSE-COUNT)

"Coarse counting" or "probabilistic counting" is a technique often used for query size estimation, for computing the number of distinct targets in a relation [FM85, WVZT90], for mining association rules [PCY95], and for other applications. The simplest form of coarse counting uses an array $A[1..m]$ of m counters and a hash function h_1, which maps target values from $\log_2 n$ bits to $\log_2 m$ bits, $m << n$. The *CoarseCount* algorithm works as follows: Initialize all m entries of A to zero. Then perform a linear scan of R. For each tuple in R with target v, increment the counter $A[h_1(v)]$ by one. After completing this *hashing scan* of R, compute a bitmap array $BITMAP_1[1..m]$ by scanning through array A, and setting $BITMAP_1[i]$ if bucket i is *heavy*, i.e. if $A[i] \geq T$. We compute $BITMAP_1$ since it is much smaller than A, and maintains all the information required in the next phase. After $BITMAP_1$ is computed, we reclaim the memory allocated to A. We then compute F by performing a *candidate-selection* scan of R, where we scan R, and for each target v whose $BITMAP_1[h_1(v)]$ is one, we add v to F. Finally we remove the false-positives by executing *Count(F)*. Note that there are no false-negatives in our coarse-counting approach.

The candidate-selection scan in this simple coarse-counting algorithm may compute a large F (that may be many times as large as the given memory), since light targets may be hashed into *heavy* buckets. A bucket may be heavy if it has (1) one or more heavy elements, or (2) many light elements whose combined counts are above the specified threshold.

4 HYBRID techniques

We now present three different approaches to combine the sampling and counting approaches we presented earlier. Each approach first samples the data to identify *candidates* for heavy targets; then it uses coarse-counting principles to remove false-negatives and false-positives. By this two-stage approach, we manage to reduce the number of targets that fall into heavy buckets – this leads to fewer light targets becoming false positives. We refer to the three approaches as the HYBRID class of algorithms.

4.1 DEFER-COUNT Algorithm

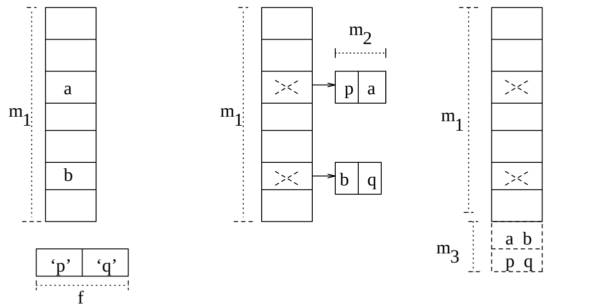
First, compute a small sample (size $s << n$) of the data using sampling techniques discussed in Section 3.1. Then select the $f, f < s$, most frequent targets in the sample and add them to F. (These targets are likely to be heavy, although we do not know for sure yet.) Now execute the hashing scan of COARSE-COUNT, but do not increment the counters in A for the targets already in F. Next perform the candidate-selection scan as before, adding targets to F. Finally, remove false positives from F by executing $Count(F)$.

We see an example of this approach in Figure 1 (a). Consider the case when p and q are heavy targets, and targets a and b are light targets. In this case, p and q were identified in the sampling phase to be potentially heavy, and are maintained explicitly in memory (denote by 'p' and 'q') so they are not counted in the buckets (as are a and b).

The intuition behind the DEFER-COUNT algorithm is as follows. Sampling is very good for identifying some of the heaviest targets, even though it is not good for finding all the heavy targets. Thus, we select f so that we only place in F targets that have a very high probability of being heavy. Then, for each of these targets v that is identified in advance of the hashing scan, we avoid pushing $A[h_1(v)]$ over the threshold, at least on account of v. This leads to fewer heavy buckets, and therefore fewer false positives.

The disadvantage of DEFER-COUNT is that it splits up valuable main memory between the sample

[2] The 80 – 20 rule is an instance of high skew. When the rule applies, a very small fraction of targets account for 80% of tuples in R, while the other targets together account for the other 20% [Zip49].



(a) DEFER-COUNT (b) MULTI-LEVEL (c) MULTI-STAGE

Figure 1: Alternative HYBRID techniques to combine sampling and coarse-counting.

set, and the buckets for counting. Even if f is small, we maintain the explicit target. For instance, if we use DEFER-COUNT to count heavy-item pairs (two-field target set), we need eight bytes to store the item pair. The storage requirement gets progressively worse if we start counting heavy-item triples, or heavy-item quadruples, and so on. Another problem with implementing DEFER-COUNT is that it is hard to choose good values for s and f that are useful for a variety of data sets. Yet another problem with DEFER-COUNT is that for each target, we incur the overhead of checking if the target exists in f during the hashing scan.

4.2 MULTI-LEVEL Algorithm

We now propose an algorithm that does not explicitly maintain the list of potentially heavy targets in main memory like DEFER-COUNT. Instead MULTI-LEVEL uses the sampling phase to identify potentially heavy buckets as follows.

First, perform a *sampling* scan of the data: For each target v chosen during this sampling scan, increment $A[h(v)]$, for hash function h. After sampling s targets, consider each of the A buckets. If $A[i] > T * s/n$, we mark the i^{th} bucket to be *potentially heavy.* For each such bucket allocate m_2 *auxiliary* buckets in main memory. (We will sometimes refer to the A buckets as *primary* buckets, to maintain the distinction.)

Next, reset all counters in the A array to zero. Then perform a hashing scan of all the data. For each target v in the data, increment $A[h(v)]$ if the bucket corresponding to $h(v)$ is not marked as potentially heavy. If the bucket is so marked, apply a second hash function $h_2(v)$ and increment the corresponding auxiliary bucket.

We show an example of this procedure in Figure 1 (b). In the sampling phase, two buckets (marked with dotted X's) are identified to be potentially heavy, and are each allocated $m_2 = 2$ auxiliary buckets. During the subsequent scan, when targets $\{a, b, p, q\}$ fall into the heavy buckets, they are rehashed using h_2 to their corresponding auxiliary buckets. Note that we do not explicitly store the targets in the auxiliary buckets as indicated in the figure; we continue to maintain only counters in the buckets.

The idea behind the MULTI-LEVEL algorithm is very similar to the concept of *extensible indices* commonly used in databases [Ull88] – these indices grow over populated buckets by adding auxiliary buckets dynamically. However, the difference is that in the case of extensible indices the entire key that is being indexed, is stored. Hence when buckets are overpopulated, we can dynamically add auxiliary buckets efficiently. Recall that we cannot afford to store the targets explicitly in main memory, and can only maintain counters. Hence we perform the prescan to preallocate auxiliary buckets for potentially heavy buckets. Also notice that MULTI-LEVEL does not store the sample set explicitly like DEFER-COUNT does – this is useful especially when the size of targets is very large.

One problem with MULTI-LEVEL is that it splits a given amount of main memory between the primary and auxiliary buckets. Deciding how to split memory across these two structures is not a simple problem – we can only empirically determine good splits for datasets. Also, the cost of rehashing into the auxiliary buckets could be expensive, if a second hash function is employed. In practice, however, we can avoid this by using one hash function: we can use fewer bits for the first hashing, and use the residual bits to "hash" the target into the auxiliary buckets.

We now discuss one important detail for implementing the above scheme. In Figure 1, we maintain pointers to auxiliary buckets. In some cases, maintaining eight bytes per pointer may be expensive especially if the number of potentially heavy buckets is high. In such cases, we can allocate all the auxiliary buckets for all potentially-heavy buckets contiguously in main memory starting at base address B. For the i^{th} potentially-heavy bucket, we can store in A the offset into the auxiliary buckets. We can then compute the auxiliary buckets for potentially heavy bucket $A[i]$, to be in locations $[B + (A[i] - 1) \times m_2, B + A[i] \times m_2)$.

4.3 MULTI-STAGE Algorithm

We now propose a new technique that uses available memory more efficiently than the MULTI-LEVEL algorithm. MULTI-STAGE has the same prescan sampling phase as MULTI-LEVEL, where it identifies potentially heavy buckets. However, MULTI-STAGE does not allocate auxiliary buckets for each potentially heavy bucket. Rather it allocates a common *pool* of auxiliary buckets $B[1, 2, \ldots, m_3]$. Then it performs a hashing scan of the data as follows. For each target v in the data, it increments $A[h(v)]$ if the bucket corresponding to $h(v)$ is not marked as potentially heavy. If the bucket is so marked, apply a second hash function h_2 and increment $B[h_2(v)]$.

We present an example of this procedure in Figure 1 (c). We mark the common pool of B arrays arrays using dotted lines. Note that the targets $\{a, b, p, q\}$ are remapped into the auxiliary buckets, using a second hash function that uniformly distributes the targets across the common pool of auxiliary buckets. It is easy to see that in this example there is a 50% chance that both the heavy targets p and q will fall into the same bucket. In such cases, targets a and b are no longer false-positives due to p and q. Indeed in the figure, we present the case when p and q do fall into the same bucket. We have analysed MULTI-LEVEL based on the above intuition, in the full version of the paper [FSGM+97].

The main intuition behind sharing a common pool of auxiliary buckets across potentially heavy buckets is that several heavy targets when rehashed into B could fall into the same bucket as other heavy targets (as illustrated in the example). MULTI-LEVEL does not have this characteristic, since the heavy targets are rehashed into their local auxiliary structures. Hence we can expect MULTI-STAGE to have fewer false-positives that MULTI-LEVEL, for a given amount of memory.

MULTI-STAGE shares a disadvantage with MULTI-LEVEL in that determining how to split the memory across the primary buckets and the auxiliary buckets can only be determined empirically.

5 Optimizing HYBRID using MULTI-BUCKET algorithms

The HYBRID algorithms discussed in the last section may still suffer from many false-positives if many light values fall into buckets with (1) one or more heavy targets, or (2) many light targets. The sampling strategies we outlined in the last section alleviate the first problem to a certain extent. However the heavy targets not identified by sampling could still lead to several light values falling into heavy buckets. Also, HYBRID cannot avoid the second problem. We now propose how to improve the HYBRID techniques of the last section, using multiple sets of primary and auxiliary buckets, to reduce the number of false positives significantly. We analyze the same idea in two different contexts, in the following subsections based on the number of passes required over the data.

For clarity, we describe the techniques of this section, in the context of the simple DEFER-COUNT algorithm, even though the techniques are also applicable to the MULTI-LEVEL, and MULTI-STAGE algorithms. Furthermore, for the techniques we present below we continue to perform the sampling scan to identify potentially heavy targets, and store them in F. We do not count these targets during the hashing scans, but count them explicitly in the candidate-selection phase. After the candidate-selection phase, we continue to execute $Count(F)$ to remove false-positives. Since these steps are common to all the following techniques, we do not repeat these steps in the following discussion.

5.1 Single scan DEFER-COUNT with multiple hash functions (UNISCAN)

We illustrate UNISCAN using two hash functions h_1 and h_2 that map target values from $\log_2 n$ bits to $log_2(m/2)$ bits, $m << n$. The memory allocated is first divided into two parts for the two counting and bitmap arrays. That is, we now have $A_1[1..m/2]$, $A_2[1..m/2]$, $BITMAP_1[1..m/2]$ and $BITMAP_2[1..m/2]$. We then execute the prescan sampling phase in DEFER-COUNT, identify f potentially heavy candidates, and store them in F. Next, we do one pass over the input data, and for each tuple in R with value v, $v \notin F$, we increment both $A_1[h_1(v)]$ and $A_2[h_2(v)]$ by one. Finally we set $BITMAP_1[i]$ to 1 if $A_1[i] \geq T$, $1 \leq i \leq m/2$. We handle $BITMAP_2$ similarly, and then deallocate A_1 and A_2.

In the candidate-selection phase, we do one pass of the data and for each tuple with value v, we add v to F only if both $BITMAP_1[h_1(v)]$ and $BITMAP_2[h_2(v)]$ are set to one. We can easily generalize the above procedure for k different hash functions $h_1, h_2, \ldots, h_k$. As mentioned earlier, for now we assume that A, the k bitmaps, and F all fit in main memory. We will discuss our model for memory usage in Section 7.

Choosing the right value of k is an interesting problem, for a given amount of main memory. As we choose a larger value of k, we have many hash tables, but each hash table is smaller. While the former helps in reducing the number of false positives, the latter increases the number of false positives. Hence there is a natural trade-off point for choosing k. We discuss in [FSGM+97] how to choose a good value of k for UNISCAN.

5.2 Multiple scan DEFER-COUNT with multiple hash functions (MULTISCAN)

Rather than use multiple hash functions within one hashing scan and suffer an increased number of false positives due to smaller hash tables, we can use the same idea across multiple hashing scans as follows. After the sampling prescan, execute one hashing scan with hash function h_1. Store the corresponding $BITMAP_1$ array on disk. Now perform another hashing scan with a different hash function h_2. Store the corresponding $BITMAP_2$ array on disk. After performing k hashing scans, leave the last $BITMAP$ in memory and retrieve the $k-1$ $BITMAP$ arrays from disk. Then execute the candidate-selection scan and add value v to F if $BITMAP_i[h_i(v)] = 1$, $\forall i, 1 \leq i \leq k$.

5.3 Improving MULTISCAN with shared bitmaps (MULTISCAN-SHARED)

In MULTISCAN we performed each hashing scan independent of the previous scans, even though the $BITMAP$ information from previous scans was available. In MULTISCAN-SHARED we assume that in the $(i+1)^{st}$ hashing scan, bitmaps from all i previous

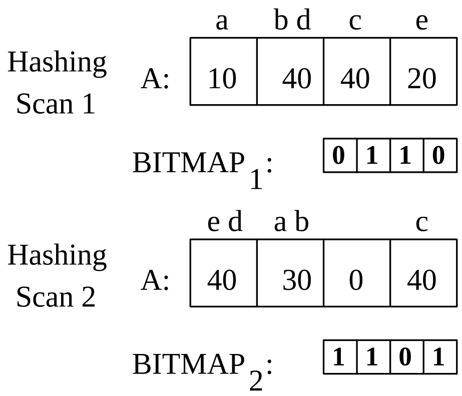


Figure 2: Hashing in MULTISCAN.

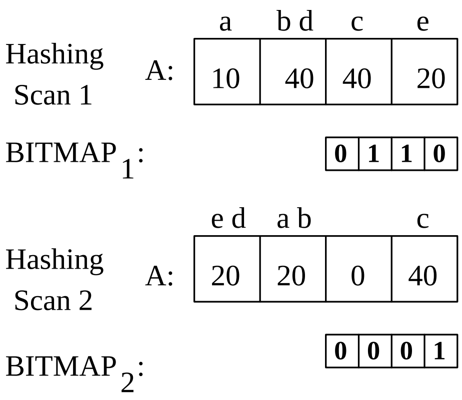


Figure 3: Hashing in MULTISCAN-SHARED.

hashing scans are retained in memory. This optimization works as follows: During the $(i+1)^{st}$ hashing scan, for target v, increment $A[h_{i+1}(v)]$ by one, only if $BITMAP_j[h_j(v)] = 1$, for all j, $1 \leq j \leq i$.

The following example illustrates how MULTISCAN-SHARED reduces the number of false-positives over MULTISCAN. Consider the case when we have the following $\langle$target : frequency$\rangle$ pairs in R: $\langle a : 10\rangle$, $\langle b : 20\rangle$, $\langle c : 40\rangle$, $\langle d : 20\rangle$, $\langle e : 20\rangle$, i.e., target a occurs in ten tuples in R, b occurs in 20 tuples in R, and so on. Let $T = 30$ and $m = 4$. Let h_1 map the targets to the following buckets, set of targets pairs: $[0 : \{a\}, 1 : \{b, d\}, 2 : \{c\}, 3 : \{e\}]$ as shown in Figure 2, i.e., $h_1(a) = 0$, $h_1(b) = h_1(d) = 1$, etc. Similarly h_2 maps the targets to the following buckets $[0 : \{e, d\}, 1 : \{a, b\}, 2 : \{\}, 3 : \{c\}]$. In Figure 2 we show the counts in array A and the corresponding $BITMAP$ after the first hashing scan when we execute MULTISCAN. Similarly we compute A and $BITMAP_2$ after the second hashing scan. Now in the candidate selection scan of MULTISCAN, we would choose $\{b, c, d\}$ to be part of F, since targets b, c, d fall into heavy buckets under both hash functions.

Now consider the execution of MULTISCAN-SHARED in Figure 3. The first hashing scan remains the same as before. The second scan however computes a different bitmap, since the second hashing scan uses the information in $BITMAP_1$ before incrementing A. To illustrate, consider how e is counted by each algorithm in the second hashing scan. In MULTISCAN, $A[h_2(e)]$ is incremented for each of the 20 occurrences of e. However in MULTISCAN-SHARED, $A[h_2(e)]$ is not incremented for the 20 occurences of e, since we already know that e is light (because $BITMAP_1[3] = 0$). Since e does not increment $A[0]$ in the second hashing scan, d is no longer a part of the candidate set. In fact in the candidate-selection scan, the only target chosen by the MULTISCAN-SHARED will be $\{c\}$, as opposed to the $\{b, c, d\}$ chosen by MULTISCAN.

5.4 Variant of MULTISCAN-SHARED (MULTISCAN-SHARED2)

We now propose a variant of MULTISCAN-SHARED that uses less memory for $BITMAP$s. In this variant, we maintain the $BITMAP$'s only from the last q hashing scans while performing the $(i+1)^{st}$ ($q \leq i$) hashing scan, rather than maintaining all i prior $BITMAP$s. The conjecture is that the q latest $BITMAP$s from hashing scans $i - q + 1$ through i have fewer and fewer bits set to one. Therefore these $BITMAP$s have more pruning power than earlier, while using the same storage space. We use MULTISCAN-SHARED2 to denote this algorithm.

6 Extending HYBRID and MULTI-BUCKET algorithms

In this section we briefly describe some variations to the schemes we presented earlier.

1. **Collapsing candidate-selection scan with final counting-scan:** The MULTISCAN algorithm (and its extensions that were proposed in Sections 5.3 and 5.4) performs k hashing scans, one candidate-selection scan, and finally one counting scan where false positives were eliminated. In cases where the size of F is expected to be small, we can collapse the last two scans into one as follows. When executing the candidate-selection scan, we add an in-memory counter for each element of F. In that scan, as we add each target to F (because it appeared in heavy buckets for all k-hash functions), we check if the target was already in F. If so, we increment its counter; if not, we add it to F with its counter initialized to 1. We can dispense with the final counting-scan because we already have a count of how many times each F target appears in R. Targets whose count exceed the threshold are in the final answer.

2. **Parallelizing hashing scans for MULTISCAN:** We can parallelize the hashing scans of MULTISCAN across multiple processes. In such a case, the time for the hashing scans drops from the time for k sequential scans, to the time for

a single scan. Of course, we cannot use the same optimization for MULTISCAN-SHARED and MULTISCAN-SHARED2 since they use bitmaps from previous iterations.

3. **SUM queries:** As we mentioned in Section 1, we can extend our techniques to iceberg queries containing `HAVING SUM(attrib)`. To illustrate, consider query *PopularItem* from Section 2. We can perform this query by performing a hashing scan on the *LineItem* relation. In this pass, we compute $h_1(partKey, region)$, and increment the corresponding counter in A by $numSales * price$. At the end of the hashing scan, compress the A array into $BITMAP_1$, with the definition that bucket i is heavy if $A[i]$ is greater than or equal to the given threshold value of one million. Then perform subsequent hashing scans if necessary and finally produce $partKeys$'s whose total revenues exceed the specified threshold.

7 Case studies

Given the relatively large number of techniques we present in this paper, each of which is parameterized in different ways (such as how much of data we should sample, s, how many values to retain to be potentially heavy, f, and memory allocations), it is difficult to draw concrete conclusions without looking at particular application scenarios. We chose three distinct application scenarios and designed our experiments to answer the following questions: (1) How does each scheme perform as we vary the amount of memory allocated? We report the performance both in terms of the size of the candidate set ($|F|$) produced, and the total time each scheme takes to produce F, as well as to remove the false positives using $Count(F)$. (2) How does each scheme perform as we vary the threshold? As above, we report both $|F|$ and the total time. (3) How do schemes perform for different data distributions? That is, if the input data follows a skewed distribution as opposed to less skewed distributions, how are the schemes affected by sampling?

Before we present our results, we discuss how we allocate memory in our experiments. We experimented with a variety of ways to split the available memory between the sample set of size f (in case of DEFER-COUNT based algorithms), the primary and the auxiliary buckets. We found the following approach to work best for our data.

1. **Allocate f:** For algorithms based on DEFER-COUNT, choose a small f for the sampling scan and allocate memory for that set. We discuss later what should be the value of f, for each application.

2. **Allocate auxiliary buckets:** Allocate p_{aux} percent of the remaining memory after the first step to auxiliary buckets. As the algorithm executes we may discover that this amount of allocated memory was insufficient for the auxiliary buckets. If that happens, we greedily select the buckets with highest A counter values, and assign as many of these as possible to the auxiliary area. The remaining potentially heavy buckets, that could not be assigned to the limited auxiliary area, are treated as any other primary bucket during the hashing scan.

3. **Allocate primary buckets and bitmaps:** Allocate the balance of the memory to the primary buckets and their bitmaps. In case of UNISCAN we need to this memory among the k primary buckets and their bitmaps (based on the value of k chosen by the analysis in the Appendix).

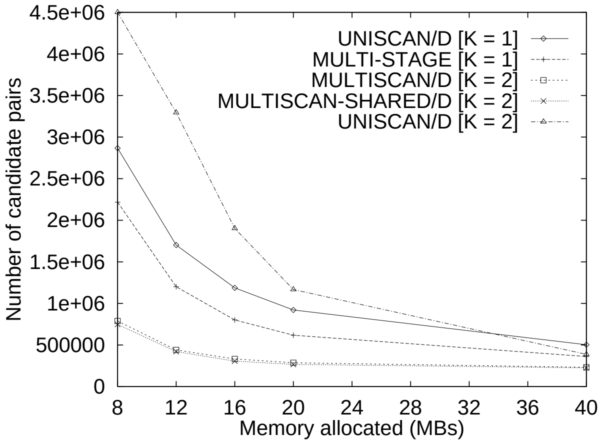


Figure 4: $|F|$ as memory varies ($T = 500$).

In our experiments, we found p_{aux} between $15 - 20\%$ to be good values for splitting up our memory. Before the candidate-selection scan, we reclaim the memory allocated to the primary buckets and allocate that to store F.

In the following experiments, if the final F (input to $Count(F)$) does not fit in main memory, we stream the tuples in F onto disk, and we execute $Count(F)$ using a disk-based sorting algorithm. Our implementation is enhanced with *early aggregation* [BD83] so that it integrates counting into the sorting and merging processes, for efficient execution. As we discussed earlier, this is merely one way to execute $Count(F)$. Hence the reader should not interpret the results of this section as absolute predictions, but rather as illustrations of performance trends. For the following experiments, we used a SUN ULTRA/II running SunOS 5.6, with 256 MBs of RAM and 18 GBs of local disk space.

Case 1: Market basket query

We use the market basket query to find commonly occuring word pairs. For this we use $100,000$ web documents crawled and stored by the Stanford Google webcrawler [BP]. The average length of each document is 118 words. From this data we computed the *DocWord* relation to be $\langle$`docID, wordID`$\rangle$, if document with identifier `docID` had a word with identifier `wordID`. This relation was about 80 MBs, when we

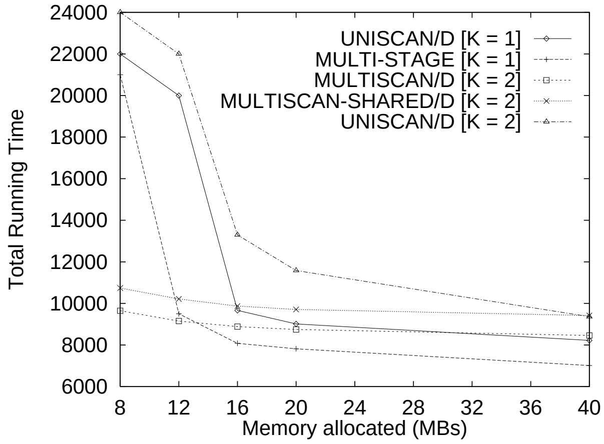


Figure 5: Total time as memory varies ($T = 500$).

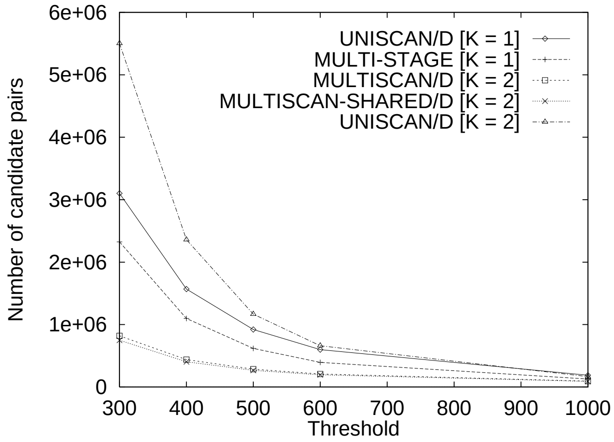


Figure 6: $|F|$ as threshold varies ($M = 20$ MB).

used 4-byte integers for `docIDs` and `wordIDs`. Note that we removed entries corresponding to 500 predefined stop words from this relation [SB88]. Recall that the R over which the iceberg query is to be executed has all pairs of words that occur in the same document. If R were to be materialized on disk, it would require about 29.4 GBs to store R; in addition, we may require temporary storage while performing the aggregation. Since the storage requirements may be impractical, we do not discuss this technique any more in this section.

To avoid explicitly materializing R we use the following technique that we can use in general to execute iceberg queries, when R is not materialized. Typically, tuples that refer to the same document are contiguous in *DocWord*. (This is because *DocWord* is produced by reading and parsing documents one at a time. If entries are not contiguous, we can sort the relation.) Because of this property, we can simply scan *DocWord* and produce $\langle w_i, w_j \rangle$ for each w_i, w_j pair that occurs in the same document. Rather than explicitly storing such tuples, stream the tuples directly to the algorithm we use to execute the iceberg query. For instance, if we use DEFER-COUNT to execute the iceberg query (assume $s = 0$), increment $A[h(w_i, w_j)]$ as soon as tuple $\langle w_i, w_j \rangle$ is produced. Notice that we cannot apply a similar optimization for sorting or hybrid hashing based schemes, since the tuples are materialized explicitly (for sorting), or will need to be stored in the hash table (for hybrid hashing). We can in general use our technique to execute a query over any join of sorted relations. In fact, R can be any expression of sorted relations, as long as we can generate R in one-pass.

We now discuss a few representative schemes for specific values of K to illustrate some of the tradeoffs involved. (We study the performance of all schemes in greater detail, in the full version of this paper [FSGM+97].) Specifically, we present results for MULTISCAN/D, MULTISCAN-SHARED/D and UNISCAN/D, the corresponding multi-bucket optimization of DEFER-COUNT. We also evaluate MULTI-STAGE for $K = 1$. We found a 1% sample of n ($s = 1\%$) and $f = 1000$ to work well in practice for this data.

In Figure 4 we show how $|F|$, the number of candidate pairs, varies as the amount of memory allocated increases. We see that $|F|$ drops as more memory is allocated, as expected. Also we see that MULTISCAN/D [K = 2] and MULTISCAN-SHARED/D [K = 2] perform best, in terms of choosing the smallest $|F|$. This is because when the amount of memory is small, doing multiple passes over the data using most of the available memory for the A array, helps prune the number of false positives significantly. UNISCAN/D [K = 2] performs poorly initially since the amount of main memory is very small, but the difference between UNISCAN/D [K = 1] and UNISCAN/D [K = 2] drops with larger memory. For memory more than about 34 MBs, we see that UNISCAN/D [K = 2] performs better than its $K = 1$ counterpart.

In Figure 5 we see the total time to answer the iceberg query as the amount of memory varies. We see that MULTISCAN/D and MULTISCAN-SHARED/D perform steadily across the different memory sizes, since they do not produce too many false positives. On the other hand, MULTI-STAGE [K = 1] performs badly when memory is limited; beyond about 14 MBs it performs best. This is because (1) the number of false positives is relatively small and hence counting can be done in main memory, (2) MULTI-STAGE scans the data one less time, and uses less CPU time in computing fewer hash functions than the other multi-bucket algorithms (such as MULTISCAN/D).

In Figure 6 we study how $|F|$, the number of candidates, varies as the threshold is varied. We see that MULTISCAN/D [K = 2] and MULTISCAN-SHARED/D [K = 2] tend to have the smallest $|F|$. Again, we see that performing multiple passes over the data using multiple hashing functions helps prune away many false-positives. In Figure 7 we see the corresponding total time to answer the iceberg query. We see that MULTI-STAGE performs the best in this interval, again because (1) F is relatively small, and (2) it performs one fewer scan over the data, and needs to

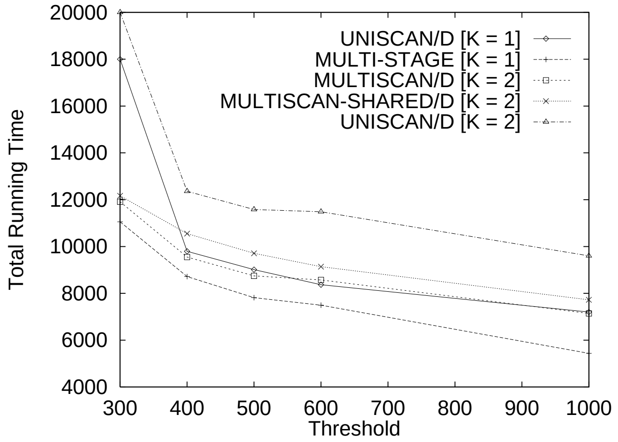


Figure 7: Total time as threshold varies ($M = 20$ MB).

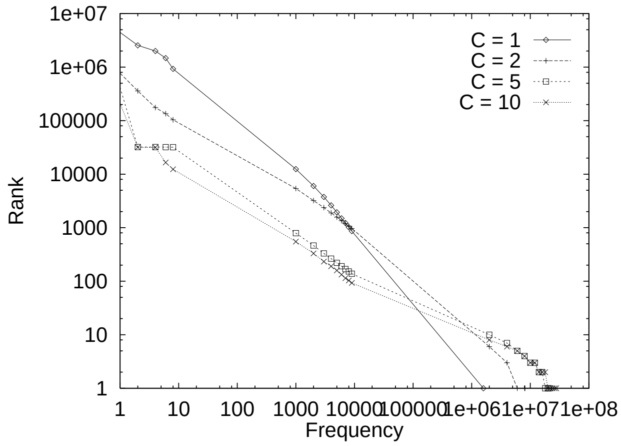


Figure 8: Frequency-rank curves for different chunkings.

compute fewer hash functions than MULTISCAN/D and MULTISCAN-SHARED/D.

In summary, we see that MULTI-STAGE works best since this application had very little data.

Case 2: Computing StopChunks

We now consider how sensitive our schemes are to skews in data distribution, using an IR example. We discussed in Section 2 how IR systems compute a set of stop words for efficiency. In general, IR systems also compute "stop chunks," which are syntactic units of text that occur frequently. By identifying these popular chunks, we can improve phrase searching and indexing. For instance, chunks such as "Netscape Mozilla/1.0" occur frequently in web documents and may not even be indexed in certain implementations of IR systems (such as in [SGM96]), to reduce storage requirements.

For this set of experiments, we used $300,000$ documents we obtained from the Stanford Google crawler (as above). We defined chunks based on *sliding windows* of words as in [SGM96]. We say we use "$C = i$" chunking, if the j^{th} chunk of a given document is the

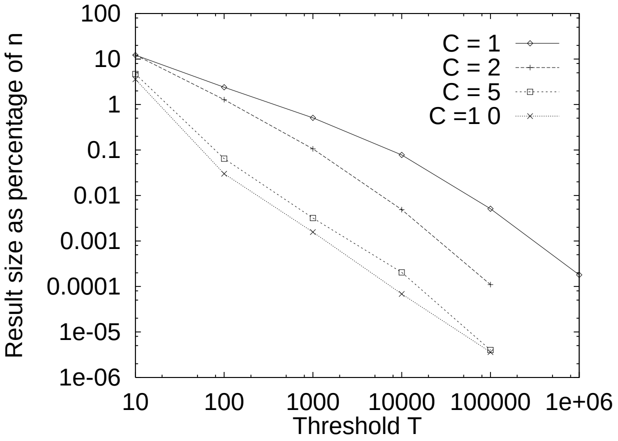


Figure 9: Result sizes for different thresholds.

sequence of words from j through $j+i-1$. For a corpus of documents, we can compute the *DocSign* ($C = i$) relation which contains $\langle d_h, s_j \rangle$, if document d_h contains s_j, the 8-byte hashed version of the j^{th} chunk. For our experiments we computed four different *DocSign* tables for $C = 1, 2, 5, 10$. (Note that the *DocSign* relation for $C = 1$ is the relation used to compute stop words in IR systems.)

Our first two graphs illustrate the nature of the data, and not a specific algorithm. In Figure 8 we show, on a log-log plot, the frequency-rank curves of the four different chunkings. As expected, the smaller the C used to construct a chunk, the fewer the number of distinct target values, and the larger the data skew. For instance, with $C = 1$, the number of distinct chunks, n, is over 1.5 million, and the heaviest target occurs about 4.5 million times in *DocChunk*. For $C = 10$, $n = 27.7$ million, while the heaviest target occurs only 0.21 million times. The size of each *DocSign* relation was about 4.2 gigabytes (Note that we did not remove precomputed stop words from these relations as we did in the market-basket query.)

In Figure 9 we show (again on a log-log plot) what percentage of the n unique terms are actually heavy, for different thresholds. We see in the figure that, as expected, the number of heavy targets (the tip of the iceberg) drops significantly as T increases.

In the following two graphs, Figure 10 and 11, we study how the number of hashing scans K, and the number of hash buckets m affect false-positive errors. Due to lack of space, we present the results only in the context of MULTISCAN-SHARED2/D, with $q = 2$ (the number of previous bitmaps cached in memory). The vertical axis in both figures is the percentage of false positives ($100 * \frac{FP}{n}$, where FP is the number of false positives). As we expected, the percentage of false positives drops dramatically with increasing k. For instance for $C = 1$, the percentage drops from about 70% for $k = 1$ to less than 10% for $k = 4$. Also it is interesting to note that the number of false positives drops as the data is less skewed (from $C = 1$ through $C = 10$), especially as the number of hashing scans increases. We attribute this drop to three factors: (1)

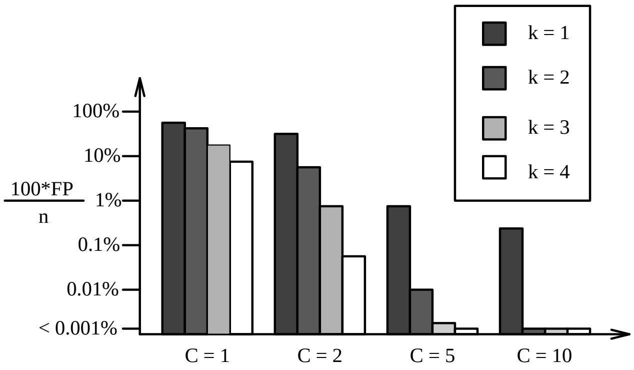


Figure 10: Performance of MULTISCAN-SHARED2/D with k ($T = 1000, m = 1\%$ of n).

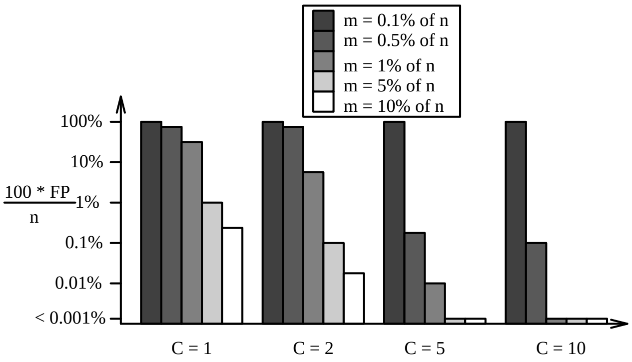


Figure 11: Performance of MULTISCAN-SHARED2/D with m ($T = 1000, k = 2$).

there are fewer heavy targets (Figure 9), (2) since data is not very skewed, fewer light targets fall into buckets that are heavy due to heavy targets, and (3) as more hashing scans are performed, fewer light targets fall into heavy buckets across each of the hashing scans.

In summary, these experiments quantify the impact of skew, and provide guidelines for selecting the number of hashing scans needed by MULTISCAN-SHARED2/D, as the "tip of the iceberg" changes in size. Analogous behavior can be observed for the other schemes.

Case 3: DocumentOverlap Query

In Figure 12 we present the total time to execute the *DocumentOverlap* query (discussed in Section 2) using MULTISCAN and MULTISCAN-SHARED techniques as the amount of memory (M) changes. We executed the query on the *DocSign* relation from Case 2, when $C = 1$. Since the data was unskewed for this query, we avoid the sampling scan, i.e., $s = 0\%$.

In Figure 12 we see that MULTISCAN-SHARED2 [q = 1] performs best, when the amount of memory is small, but progressively becomes inferior to MULTISCAN and MULTISCAN-SHARED as memory increases. MULTISCAN-SHARED [q = 2] is in between MULTISCAN-SHARED [q = 1] and MULTISCAN-SHARED, for small values of memory. The above behavior of MULTISCAN-

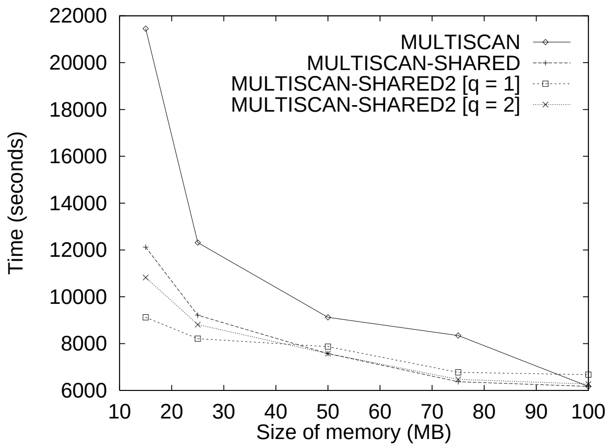


Figure 12: Performance of algorithms with M for DocumentOverlap query for $C = 1$.

SHARED2 compared to MULTISCAN-SHARED is due to the following competing factors: (1) MULTISCAN-SHARED2 uses fewer bitmaps than MULTISCAN-SHARED, thereby allocating more memory for primary buckets. (2) For a given amount of memory, MULTISCAN-SHARED prunes more light targets than MULTISCAN-SHARED2, as we discussed earlier. For small values of memory, MULTISCAN-SHARED2 performs better than MULTISCAN-SHARED, since the first factor dominates. For larger values of memory, the extra space allocated to the additional bitmaps for MULTISCAN-SHARED still leaves enough memory for the primary buckets. Hence the second factor dominates. We also see that MULTISCAN does not perform too well for small memory, since it does not use bitmaps to prune away light targets, as we discussed earlier. Hence we see that choosing $q = 1$ or 2 may be useful for small sized memory while still leaving sufficient main memory for primary buckets.

The size of R, if materialized, is 52 GBs. If we assume disks can execute sequential scans at the rate of 10 MB/sec, it would take $52 * 1024/10 \approx 5300$ seconds each to read and write R. However, notice that MULTISCAN-SHARED2 [q = 1] would finish executing even before R is written once and read once! Of course, since R has to be sorted to execute the iceberg query, it is easy to see that sorting-based execution would require too much disk space to materialize and sort R, and will take much longer than our schemes.

7.1 Summary

Based on our case studies (and from experiments we do not report here due to lack of space [FSGM+97]), we propose the following informal "rules of thumb" to combine schemes from the HYBRID and MULTI-BUCKET algorithms:

1. HYBRID **algorithms:** MULTI-LEVEL rarely performs well in our experiments, while DEFER-COUNT and MULTI-STAGE tend to do very

well under different circumstances. If you expect the data distribution to be very skewed where very few targets are heavy, but constitute most of the relation), use DEFER-COUNT with a small f set. If you expect the data not to be too skewed, use MULTI-STAGE since it does not incur the overhead of looking up the values in f. If you expect the data distribution to be flat, do not use the sampling scan.

2. MULTIBUCKET **algorithms:** In general we recommend using MULTISCAN-SHARED2 with $q = 1$ or $q = 2$. For relatively large values of memory, we recommend UNISCAN with multiple hash functions, since we can choose $K > 1$ and apply multiple hash functions within one hashing scan, as we discuss in the full version of this paper [FSGM+97].

8 Related Work

Flajolet and Martin [FM85], and Whang et al. [WVZT90] proposed a simple form of coarse counting for estimating the number of distinct elements in a multiset. Park et al. [PCY95] proposed coarse counting in the context of mining association rules. All the above approaches use a single hash function for their coarse counting, and hence tend to have many false positives. We extend the above techniques using our HYBRID and MULTIBUCKET algorithms, and perform a comprehensive study of these techniques using a case study approach.

9 Conclusion

In this paper we studied efficient execution techniques for *iceberg* queries, an important class of queries with widespread application in data-warehousing, data mining, information retrieval and copy detection. We proposed algorithms that compute the result, the "tip of the iceberg," much more efficiently than conventional schemes. We evaluated our algorithms using a case study approach in three real applications, and observed that the savings are indeed very significant. Some algorithms in the suite we have provided are better suited to some scenarios, depending on the data skew, available memory, and other factors. We have provided some empirical "rules of thumb" for selecting a scheme and for allocating memory to its data structures.

References

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of International Conference on Very Large Databases (VLDB '94)*, pages 487 – 499, September 1994.

[BD83] D. Bitton and D. J. DeWitt. Duplicate record elimination in large data files. *ACM Transactions in Database Systems (TODS)*, 8(2):255 – 265, 1983.

[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 masket basket data. In *Proceedings of ACM SIGMOD Conference*, pages 255 – 264, May 1997.

[BP] S. Brin and L. Page. Google search engine/ backrub web crawler.

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

[FM85] P. Flajolet and G.N. Martin. Probabilistic counting algorithms for database applications. *Journal of Computer System Sciences*, 31:182 – 209, 1985.

[FSGM+97] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J.D. Ullman. Computing iceberg queries effciently. Technical report, Stanford DBGroup Technical Report, February 1998.

[HNSS96] P.J. Haas, J.F. Naughton, S. Seshadri, and A.N. Swami. Selectivity and cost estimation for joins based on random sampling. *Journal of Computer and System Sciences*, 52(3):550 – 569, June 1996.

[Olk93] F. Olken. *Random sampling from databases.* Ph.D. dissertation, UC Berkeley, April 1993.

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

[SB88] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5), 1988.

[SGM96] N. Shivakumar and H. Garcia-Molina. Building a scalable and accurate copy detection mechanism. In *Proceedings of 1st ACM Conference on Digital Libraries (DL'96)*, Bethesda, Maryland, March 1996.

[SGM98] N. Shivakumar and H. Garcia-Molina. Computing replicas and near-replicas of documents on the web. In *To appear in Workshop on WebDatabases (WebDB'98)*, Valencia, Spain, March 1998.

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

[Ull88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems (Volume 1).* Computer Science Press, 1988.

[WVZT90] K. Whang, B.T. Vander-Zanden, and H.M. Taylor. A linear-time probabilistic counting algorithm for db applications. *ACM Transactions on Database Systems*, 15(2):208 – 229, 1990.

[Zip49] G.K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley Press, Cambridge, Massachusetts, 1949.