A Comparison of Five Probabilistic View-Size Estimation Techniques in OLAP

Kamel Aouiche and Daniel Lemire
LICEF, Université du Québec à Montréal
100 Sherbrooke West
Montreal, Canada
kamel.aouiche@gmail.com,
lemire@acm.org

ABSTRACT

A data warehouse cannot materialize all possible views, hence we must estimate quickly, accurately, and reliably the size of views to determine the best candidates for materialization. Many available techniques for view-size estimation make particular statistical assumptions and their error can be large. Comparatively, unassuming probabilistic techniques are slower, but they estimate accurately and reliability very large view sizes using little memory. We compare five unassuming hashing-based view-size estimation techniques including Stochastic Probabilistic Counting and LOGLOG Probabilistic Counting. Our experiments show that only Generalized Counting, Gibbons-Tirthapura, and Adaptive Counting provide universally tight estimates irrespective of the size of the view; of those, only Adaptive Counting remains constantly fast as we increase the memory budget.

Categories and Subject Descriptors

H.3.2 [**Information Storage and Retrieval**]: Information Storage; G.3 [**Probability and Statistics**]: Probabilistic algorithms

General Terms

Algorithms, Performance, Experimentation, Reliability.

Keywords

OLAP, materialized views, view-size estimation, data warehouse, random hashing.

1. INTRODUCTION

View materialization is one of the most effective technique to improve query performance of data warehouses. Materialized views are physical structures which improve data access time by precomputing intermediary results. Typical OLAP queries consist in selecting and aggregating data with grouping sets (GROUP BY clauses) [13]. By precomputing many plausible groupings, we can avoid slow responses due to aggregates over large tables. Many queries, such as those containing conditions (HAVING clauses) can also be computed faster using these preaggregates. However, materializing views requires additional storage space and induces maintenance overhead when refreshing the data warehouse. Moreover, the number of views is large: there are 2^d views in a d-dimensional data cube lattice [13]. Hence, one of the most important issues in data warehouse physical design is the selection of the views to materialize, an NP-hard problem [14]. Most heuristics for this problem depend on view-size estimation.

Some view-size estimation techniques make assumptions about the data distribution and others are “unassuming.” A common statistical assumption is uniformity [12], but any skew in the data leads to an overestimate. Generally, while statistically assuming estimators are computed quickly, the most expensive step being the random sampling, their error can be large and it cannot be bounded a priori. We consider several state-of-the-art statistically unassuming estimation techniques: Probabilistic Counting [10], LOGLOG Probabilistic Counting [7], Adaptive Counting [6], Generalized Counting [5], and Gibbons-Tirthapura [11]. While relatively expensive, unassuming estimators tend to provide good accuracy and reliability [4].

To use these techniques, we need to hash rows quickly and our theoretical bounds require at least pairwise independent hash values. Fortunately, while there can be several dimensions ($d > 10$) in a data cube, the number of attribute values in each dimension is often small compared to the available memory. Hence, we can hash dimensions separately, store the result in main memory, and combine these fully independent unidimensional hash values into d-wise independent multidimensional hash values.

Typically, as we allocate more memory, our algorithms become more accurate, but also slower. We are concerned with two different usage scenario. Firstly, we want rough estimates, with errors as large as 10%, as quickly as possible. In such cases, we can use tiny memory budgets (less than 1 MiB). Secondly, we want highly accurate estimates with errors less than 1% or 0.1%. In these instances, we use several megabytes of memory.

The main result of this paper is an exhaustive theoretical and experimental comparisons of a wide range of unassuming view-size estimation techniques. We also present practical theoretical results on Generalized Counting, a novel algorithm. Finally, we make some recommendations.

2. RELATED WORK

Sample-based, statistically assuming estimations are typically fast, but can be inaccurate and can still use a lot of memory. Indeed, in the worst-case scenario, the histogram of the sample might be as large as the view size we are trying to estimate. Moreover, it is difficult to derive unassuming accuracy bounds since the sample

DOLAP '07 Lisbon, Portugal

might not be representative and the model might not be a good fit. However, a sample-based algorithm is expected to be an order of magnitude faster than an algorithm which processes the entire data set. Haas et al. [15] estimate the view size from the histogram of a sample: adaptively, they choose a different estimator based on the skew of the distribution. Faloutsos et al. [8] obtain results nearly as accurate as Haas et al., that is, an error of approximately 40%, but with a simpler algorithm.

Stochastic Probabilistic Counting [10], LOGLOG Probabilistic Counting (henceforth LOGLOG) [7] and Adaptive Counting [6] have been shown to provide very accurate view-size estimations quickly for very large views, but their estimates assume we have independent hashing. Because of this assumption, their theoretical bound may not hold in practice.

Gibbons and Tirthapura [11] derived an unassuming bound, for an algorithm we will refer to as Gibbons-Tirthapura or GT, that only requires pairwise independent hashing. It has been shown recently that if you have k-wise independent hashing for $k > 2$ the theoretically bound can be improved substantially [17]. Bar-Yossef et al. [5, Section 2] presented a new scheme which they described as a generalization of Probabilistic Counting, assuming only pairwise independent hashing. The benefit of these new schemes is that as long as the random number generator is truly random and the hashed values use enough bits, the theoretical bounds have to hold irrespective of the size of the view or of other factors. We can be certain to have high accuracy and reliability, but what about speed?

3. ESTIMATION BY MULTIFRACTALS

We implemented the statistically assuming algorithm by Faloutsos et al. based on a multifractal model [8]. Given a sample, all that is required to learn the multifractal model is the number of distinct elements in the sample F_0, the number of elements in the sample N', the total number of elements N, and the number of occurrences of the most frequent item in the sample $m_{\max}$. Hence, a very simple implementation is possible (see Algorithm 1). The memory usage of this algorithm is determined by the GROUP BY query on the sample (line 6): typically, a larger sample will lead to a more important memory usage.

Algorithm 1 View-size estimation using a multifractal distribution model.

```
1: INPUT: Fact table $t$ containing $N$ facts
2: INPUT: GROUP BY query on dimensions $D_1, D_2, \ldots, D_d$
3: INPUT: Sampling ratio $0 < p < 1$
4: OUTPUT: Estimated size of GROUP BY query
5: Choose a sample in $t'$ of size $N' = \lfloor pN \rfloor$
6: Compute $g$=GROUP BY($t'$)
7: let $m_{\max}$ be the number of occurrences of the most frequent tuple $x_1, \ldots, x_d$ in $g$
8: let $F_0$ be the number of tuples in $g$
9: $k \leftarrow \lceil \log F_0 \rceil$
10: while $F < F_0$ do
11:     $p \leftarrow (m_{\max}/N')^{1/k}$
12:     $F \leftarrow \sum_{a=0}^{k} \binom{k}{a} (1 - (p^{k-a}(1-p)^a)^{N'})$
13:     $k \leftarrow k + 1$
14: $p \leftarrow (m_{\max}/N)^{1/k}$
15: RETURN: $\sum_{a=0}^{k} \binom{k}{a} (1 - (p^{k-a}(1-p)^a)^{N})$
```

4. UNASSUMING ESTIMATION

All unassuming methods presented in this paper use the same probabilistic idea. Whereas the initial data has unknown distribution, if we use an appropriate random hashing method, the hashed values are uniformly distributed (see Fig. 1).

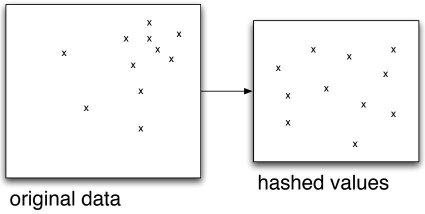


Figure 1: Irrespective of the original data, the hashed values can be uniformly distributed.

4.1 Independent Hashing

Hashing maps objects to values in a nearly random way. We are interested in hashing functions from tuples to $[0, 2^L)$ where L is fixed ($L = 32$ or $L = 64$ in this paper). Hashing is uniform if $P(h(x) = y) = 1/2^L$ for all x, y, that is, if all hashed values are equally likely. Hashing is *pairwise independent* if $P(h(x_1) = y_1 \wedge h(x_2) = y_2) = P(h(x_1) = y_1)P(h(x_2) = y_2) = 1/4^L$ for all x_i, y_i. Pairwise independence implies uniformity. Hashing is k-wise independent if $P(h(x_1) = y_1 \wedge \cdots \wedge h(x_k) = y_k) = 1/2^{kL}$ for all x_i, y_i. Finally, hashing is (fully) independent if it is k-wise independent for all k. Fully independent hashing of F_0 distinct values requires $\Omega(F_0)$ units of memory [1] and is thus impractical if F_0 is large.

We can compute k-wise independent hash values efficiently in a multidimensional data warehouse setting. For each dimension D_i, we build a look-up table T_i, using the attribute values of D_i as keys. Each time we meet a new key, we generate a random number in $[0, 2^L)$ and store it in the look-up table T_i. This random number is the hashed value of this key. This table generates (fully) independent hash values in amortized constant time. In a data warehousing context, whereas dimensions are numerous, each dimension will typically have few distinct values: for example, there are only 8,760 hours in a year. Therefore, the look-up table will often use a few Mib or less. When hashing a tuple $x_1, x_2, \ldots, x_k$ in $D_1 \times D_2 \times \ldots D_k$, we use the value $T_1(x_1) \oplus T_2(x_2) \oplus \cdots \oplus T_k(x_k)$ where $\oplus$ is the EXCLUSIVE OR operator. This hashing is k-wise independent and requires amortized constant time. Tables T_i can be reused for several estimations: we can simultaneously estimate the size of a GROUP BY on D_1 and D_2, and the size of a GROUP BY on D_2 and D_3 while using a single table T_2.

4.2 Probabilistic Counting

Our version of (Stochastic) Probabilistic Counting [10] (or just Counting for short) is given in Algorithm 2. LOGLOG (see Algorithm 3) is a faster variant [7]. The main difference between the two algorithms is that LOGLOG only keeps track of the maximum number of leading zeroes, whereas Probabilistic Counting keeps track of all observed numbers of leading zeroes and is thus more resilient to outliers in the hashing values (see Fig. 2). For the same parameter M, the memory usage of the two algorithms is comparable in practice: Probabilistic Counting uses a $M \times L$ binary matrix and LOGLOG uses M counters to store integer values ranging from 1 to $L - \log M$. Assuming independent hashing, these algorithms have (relative) standard error (or the relative standard deviation of the error) of $0.78/\sqrt{M}$ and $1.3/\sqrt{M}$ respectively (see Fig. 3). These theoretical results assume independent hashing which we cannot realistically provide. They also require the view size to be very large. Fortunately, we can detect the small views. A small view compared to the available memory (M), will leave several of the M

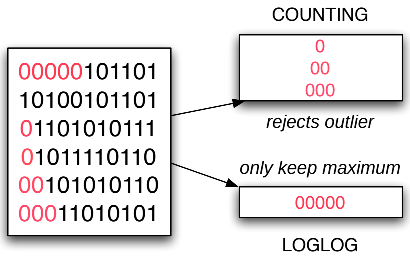


Figure 2: Probabilistic counting methods.

counters unused (array $\mathcal{M}$ in Algorithm 2). Thus, following Cai et al. [6], when more than 5% of the counters are unused we return a linear counting estimate [20] instead of the LOGLOG estimate: see last line of Algorithm 2 (henceforth Adaptive Counting). Finally, Alon et al. [2] presented a probabilistic counting variant using only pairwise independent hashing, but the error bounds are large: for any $c > 2$, the relative error is bounded by $c-1$ with reliability $1-2/c$ (an error bound of 3900% 19 times out of 20). We do not expect these algorithms to be very sensitive to the size of the memory M.

Algorithm 2 View-size estimation using Probabilistic Counting.

```
INPUT: Fact table $t$ containing $N$ facts
INPUT: GROUP BY query on dimensions $D_1, D_2, \ldots, D_d$
INPUT: Memory budget parameter $M = 2^k$
INPUT: Independent hash function $h$ from $d$ tuples to $[0, 2^L)$.
OUTPUT: Estimated size of GROUP BY query
$b \leftarrow M \times L$ matrix (initialized at zero)
for tuple $x \in t$ do
    $x' \leftarrow \pi_{D_1,D_2,\ldots,D_d}(x)$ {projection of the tuple}
    $y \leftarrow h(x')$ {hash $x'$ to $[0, 2^L)$}
    $\alpha = y \bmod M$
    $i \leftarrow$ position of the first 1-bit in $\lfloor y/M \rfloor$
    $b_{\alpha,i} \leftarrow 1$
$A \leftarrow 0$
for $\alpha \in \{0, 1, \ldots, M-1\}$ do
    increment $A$ by the position of the first zero-bit in $b_{\alpha,0}, b_{\alpha,1}, \ldots$
RETURN: $M/\phi 2^{A/M}$ where $\phi \approx 0.77351$
```

Algorithm 3 View-size estimation using LOGLOG and Adaptive Counting.

```
INPUT: fact table $t$ containing $N$ facts
INPUT: GROUP BY query on dimensions $D_1, D_2, \ldots, D_d$
INPUT: Memory budget parameter $M = 2^k$
INPUT: Independent hash function $h$ from $d$ tuples to $[0, 2^L)$.
OUTPUT: Estimated size of GROUP BY query
$\mathcal{M} \leftarrow \underbrace{0, 0, \ldots, 0}_{M}$
for tuple $x \in t$ do
    $x' \leftarrow \pi_{D_1,D_2,\ldots,D_d}(x)$ {projection of the tuple}
    $y \leftarrow h(x')$ {hash $x'$ to $[0, 2^L)$}
    $j \leftarrow$ value of the first $k$ bits of $y$ in base 2
    $z \leftarrow$ position of the first 1-bit in the remaining $L-k$ bits of $y$ (count
      starts at 1)
    $\mathcal{M}_j \leftarrow \max(\mathcal{M}_j, z)$
(original LOGLOG) RETURN: $\alpha_M M 2^{\frac{1}{M}\sum_j \mathcal{M}_j}$
  where $\alpha_M \approx 0.39701 - (2\pi^2 + \ln^2 2)/(48M)$.
(Adaptive Counting) RETURN: $\begin{cases} \alpha_M M 2^{\frac{1}{M}\sum_j \mathcal{M}_j} & \text{if } \beta/M \geq 0.051 \\ -M \log \beta/M & \text{otherwise} \end{cases}$
  where $\beta$ is the number of $\mathcal{M}_j$ for $j = 1, \ldots, M$ with value zero
```

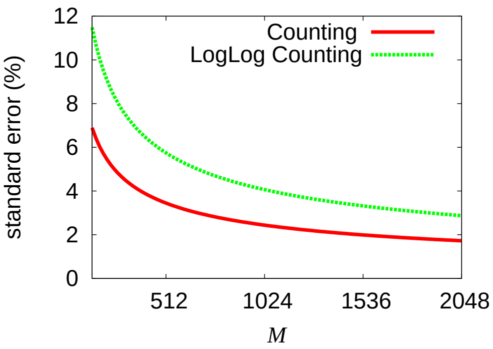


Figure 3: Standard error for Probabilistic Counting and LOGLOG as a function of the memory parameter M.

4.3 Generalized Counting

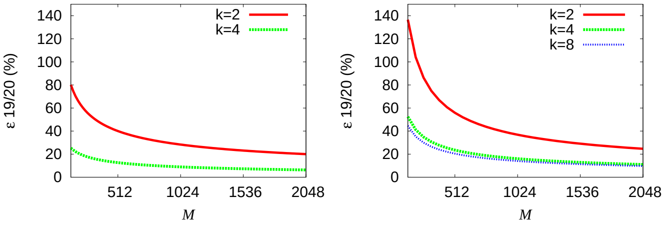
We modified a generalization to Probabilistic Counting [5, Section 2] (henceforth GC), see Algorithm 4. The tuples and hash values are stored in an ordered set, and since each tuple is inserted (line 14), the complexity of processing each tuple with respect to M is in $O(\log M)$. However, for small M with respect to the view size, most tuples are never inserted since their hash value is larger than the smallest M hash values (line 13).

The original algorithm [5] used many hashing bits: $L \geq 3\sum_i \log|D_i|$ where $|D_i|$ is the number of attribute values in dimension D_i. The main problem is that the number of required bits depends on the volume of the cuboid, which is typically far larger than the view-size. As the next result shows, with our modified algorithm, few bits are necessary. For example, when hashing with $L = 64$ bits, using a memory budget of $M = 10000$, and with relative accuracy of $\varepsilon = 0.1$, we can estimate view sizes far exceeding anything seen in practice (2×10^{21} facts). Moreover, we show that the accuracy bounds improve substantially if the hashed values are more than pairwise independent (see Fig. 4(a)).

Proposition 1 *For $L \geq 1 + \log F_0/(\varepsilon M)$ and $M \geq 2k \geq 4$, Algorithm 4 estimates a view size F_0 within relative precision $\varepsilon < 1/2$ with reliability $1-\delta$ where δ is given by $(4k/(e^{2/3}\varepsilon^2 M))^{k/2}$.*

PROOF. Suppose we have F_0 distinct tuples in the GROUP BY and assume that $F_0 > M$. If $M \leq F_0$, we can modify the algorithm so that an exact count is returned.

First, consider the case where we overestimate the true count by ε, that is $2^L M/\max(\mathcal{M}) \geq (1+\varepsilon)F_0$, hence we have at least M hashed values smaller than $2^L M/((1+\varepsilon)F_0)$. Hashed values take integer values in $[0, 2^L)$. Assuming $L \geq 1 + \log F_0/(\varepsilon M)$, the probability that a hashed value is smaller than $2^L M/((1+\varepsilon)F_0)$ is less than $M/((1+\varepsilon)F_0) + 2^{-L} \leq M/((1+\varepsilon)F_0) + \varepsilon M/(2F_0) \leq M(2+\varepsilon+\varepsilon^2)/(2(1+\varepsilon)F_0) = Mp/F_0$ where $p = (2+\varepsilon+\varepsilon^2)/(2(1+\varepsilon))$. Let X_i for $i = 1, \ldots, F_0$ be 1 with probability p/F_0 and zero otherwise. Write $X = \sum_{i=1,\ldots,F_0} X_i$, we have that $\bar{X} = \sum_{i=1,\ldots,F_0} E(X_i) = Mp$ whereas, by pairwise independence, $\sigma^2 = var(X) = \sum_{i=1,\ldots,F_0} var(X_i) = F_0(Mp/F_0 - M^2p^2/F_0^2) = Mp(1 - Mp/F_0) \leq Mp$. By a Chernoff-Hoeffding bound [18, Theorem 2.4] and the k-wise independence of the X_i's, $P(X \geq M) \leq P(|X - Mp| > M - Mp) \leq \left(\frac{kMp}{e^{2/3}(1-p)^2M^2}\right)^{k/2} = \left(\frac{kp}{e^{2/3}(1-p)^2M}\right)^{k/2}$. We have that $p \leq 1$ and $1-p \geq \varepsilon/2$ for $\varepsilon < 1/2$, hence $P(X \geq M) \leq \left(\frac{k4}{e^{2/3}\varepsilon^2 M}\right)^{k/2}$. Finally, observe that $P(2^L M/\max(\mathcal{M}) \geq (1+\varepsilon)F_0) \leq P(X \geq M)$.



(a) Generalized Counting

(b) Gibbons-Tirthapura

Figure 4: Bound on the estimation error (19 times out of 20) as a function of the number of tuples kept in memory (M) with k-wise independent hashing.

Algorithm 4 Generalized Counting view-size estimation.

```
INPUT: Fact table $t$ containing $N$ facts
INPUT: GROUP BY query on dimensions $D_1, D_2, \ldots, D_d$
INPUT: Memory budget parameter $M$
INPUT: $k$-wise hash function $h$ from $d$ tuples to $[0, 2^L)$.
OUTPUT: Estimated size of GROUP BY query
$\mathcal{M} \leftarrow$ empty sorted sequence, $\max(\mathcal{M})$ returns an element with largest hashed value
$t \leftarrow 0$
for tuple $x \in t$ do
  $x' \leftarrow \pi_{D_1, D_2, \ldots, D_d}(x)$ {projection of the tuple}
  $y \leftarrow h(x')$ {hash $x'$ to $[0, 2^L)$}
  if size($\mathcal{M}$)$< M$ then
    insert $x'$ with hashed value $y$ in $\mathcal{M}$
  else if $y < \max(\mathcal{M})$ then
    insert $x'$ with hashed value $y$ in $\mathcal{M}$ {$x'$ may already be in $\mathcal{M}$}
    if size($\mathcal{M}$)$> M$ then
      remove $\max(\mathcal{M})$ from $\mathcal{M}$
RETURN: $2^L \mathrm{size}(\mathcal{M})/\max(\mathcal{M})$
```

Similarly, suppose that we underestimate the true count by ε, $2^L M/\max(\mathcal{M}) \leq (1-\varepsilon)F_0$, hence we have less than M hashed values smaller than $2^L M/((1-\varepsilon)F_0)$. The probability that a hashed value is smaller than $2^L M/((1-\varepsilon)F_0)$ is less than $M/((1-\varepsilon)F_0) + 2^{-L} \leq M/((1-\varepsilon)F_0) + \varepsilon M/(2F_0) \leq M(2+\varepsilon-\varepsilon^2)/(2(1-\varepsilon)F_0) = Mp/F_0$ where $p = (2+\varepsilon-\varepsilon^2)/(2(1-\varepsilon))$. Let X_i for $i = 1, \ldots, F_0$ be 1 with probability p/F_0 and zero otherwise. Write $X = \sum_{i=1,\ldots,F_0} X_i$, we have that $\bar{X} = Mp$ whereas $\sigma^2 = Mp$. Finally, $P(X \leq M) \leq P(|X - Mp| > M - Mp) \leq \left(\frac{kp}{e^{2/3}(1-p)^2 M}\right)^{k/2}$. By inspection, we see that $p/(1-p)^2 \leq 2/\varepsilon^2$, hence $P(X \leq M) \leq \left(\frac{k4}{e^{2/3}\varepsilon^2 M}\right)^{k/2}$ which completes the proof. □

4.4 Gibbons-Tirthapura

Originally, the GT algorithm was proposed in the context of data streams and parallel processing [11] (see Algorithm 5). If the view size is smaller than the memory parameter (M), the estimation is without error. For this reason, we expect GT to perform well when estimating small and moderate view sizes compared to the available memory. We can processing most tuples in (amortized) constant time with respect to M (line 13) using a hash table, however the occasional pruning of tuples requires (amortized) linear time with respect to M (line 16).

The original theoretical bounds [11] assumed pairwise independence. However, more independent hashing, as is possible in our context for views with many dimensions, allow for better theoretical bounds [17] as illustrated by Fig. 4(b). Comparing Fig. 4(b) and 4(a), we may be tempted to conclude that GC is far superior to GT. We will compare them experimentally.

Algorithm 5 Gibbons-Tirthapura view-size estimation.

```
1: INPUT: Fact table $t$ containing $N$ facts
2: INPUT: GROUP BY query on dimensions $D_1, D_2, \ldots, D_d$
3: INPUT: Memory budget parameter $M$
4: INPUT: $k$-wise hash function $h$ from $d$ tuples to $[0, 2^L)$.
5: OUTPUT: Estimated size of GROUP BY query
6: $\mathcal{M} \leftarrow$ empty look-up table
7: $t \leftarrow 0$
8: for tuple $x \in t$ do
9:   $x' \leftarrow \pi_{D_1, D_2, \ldots, D_d}(x)$ {projection of the tuple}
10:  $y \leftarrow h(x')$ {hash $x'$ to $[0, 2^L)$}
11:  $j \leftarrow$ position of the first 1-bit in $y$ (count starts at 0)
12:  if $j \leq t$ then
13:    $\mathcal{M}_{x'} = j$
14:    while size($\mathcal{M}$) $> M$ do
15:      $t \leftarrow t + 1$
16:      prune all entries in $\mathcal{M}$ having value less than $t$
17: RETURN: $2^t \mathrm{size}(\mathcal{M})$
```

Proposition 2 *Algorithm 5 estimates the number of distinct tuples within relative precision ε, with a k-wise independent hash for $k \geq 2$ by storing M distinct tuples ($M \geq 8k$) and with reliability $1 - \delta$ where δ is given by*

$$\delta \quad \leq \quad \frac{k^{k/2}}{e^{k/3} M^{k/2}} \left(\frac{\alpha^{k/2}}{(1-\alpha)^k} + \frac{4^{k/2}}{\alpha^{k/2} \varepsilon^k (2^{k/2} - 1)} \right).$$

for $4k/M \leq \alpha < 1$ and any $k, M > 0$.

For the case where hashing is 4-wise independent, as in some of experiments below, we derived a more concise bound [4].

Corollary 1 *With 4-wise independent hashing, Algorithm 5 estimates the number of distinct tuples within relative precision $\varepsilon \approx 5/\sqrt{M}$, 19 times out of 20 for ε small.*

5. EXPERIMENTAL RESULTS

To benchmark the accuracy and speed of our implementation of the view-size estimation algorithms, we have run tests over the US Census 1990 data set [16] as well as on synthetic data produced by DBGEN [19]. The synthetic data was produced by running the DBGEN application with scale factor parameter equal to 2 except where otherwise stated. The characteristics of data sets are detailed in Table 1. We selected 20 and 8 views respectively from these data sets: all views in US Census 1990 have at least 4 dimensions whereas only 2 views have at least 4 dimensions in the synthetic data set. Statisticians sometimes define the standard error to be the standard deviation of the measures, but when the exact value can be known, it is better to use the deviation from the true value or $\sqrt{E((X-c)^2)}/c$ where c is the value we try to estimate. The (relative) standard error, defined as the standard deviation of the error, was computed from 20 estimates using this formula where c, the exact count, was computed once using brute force.

	US Census 1990	**DBGEN**
# of facts	2458285	13977981
# of views	20	8
# of attributes	69	16
Data size	360 MiB	1.5 GiB

Table 1: Characteristic of data sets.

We used the GNU C++ compiler version 4.0.2 with the "-O2" optimization flag on an Apple MacPro machine with 2 Dual-Core Intel Xeon processors running at 2.66 GHz and 2 GiB of RAM. No thrashing was observed. To ensure reproducibility, C++ source code is available freely from the authors. For the US Census 1990

Algorithm 6 Test protocol.

```
for GROUP BY query $q \in Q$ do
  for memory budget $m \in M$ do
    for random seed value $r \in R$ do
      Estimate the size of GROUP BY $q$ with $m$ memory budget and $r$ random seed value
      Save estimation results (time and estimated size) in a log file
```

data set, the hashing look-up table is a simple array since there are always fewer than 100 attribute values per dimension. Otherwise, for the synthetic DBGEN data, we used the GNU/CGI STL extension `hash_map` which is to be integrated in the C++ standard as an `unordered_map`: it provides amortized $O(1)$ inserts and queries. All other look-up tables are implemented using the STL `map` template which has the computational complexity of a red-black tree. We used comma separated (CSV) (and pipe separated files for DBGEN) text files and wrote our own C++ parsing code.

The test protocol we adopted (see Algorithm 6) has been executed for each unassuming estimation technique, GROUP BY query, random seed and memory size. At each step corresponding to those parameter values, we compute the estimated GROUP BY view sizes and time required for their computation. Similarly, for the multifractal estimation technique, we computed the time and estimated size for each GROUP BY, sampling ratio value and random seed.

In Subsection 5.1, we consider the first use case: the user is satisfied with a moderate accuracy (such as 10%). In Subsection 5.2, we address the case where high accuracy (at least 1%) is sought, maybe at the expense of memory usage and processing speed.

5.1 Small memory budgets

5.1.1 Accuracy

Test over the US Census 1990 data set

Fig. 5 represents the standard error for each unassuming estimation technique and memory size $M \in \{16, 64, 256, 2048\}$. For the multifractal estimation technique, we present the standard error for each sampling ratio $p \in \{0.1\%, 0.3\%, 0.5\%, 0.7\%\}$. The X axis represents the size of the exact GROUP BY values and the Y axis, the corresponding standard error. Both of the X and Y axis are in a logarithmic scale. The standard error generally decreases when the memory budget increases. However, for small views, the error can exceed 100% for Probabilistic Counting and LOGLOG: this is caused by a form of overfitting where many counters are not or barely used (see Section 4.2) when the ratio of the view size over the memory budget is small. In contrast, Fig. 5(a) shows that GT has sometimes accuracy better than 0.01% for small views. For the multifractal estimation technique (see Fig. 5(d)), the error decreases when the sampling ratio increases. While the accuracy can sometimes approach 10%, we never have reliable accuracy.

Test over synthetic data

Similarly, we plotted the standard error for each technique, computed from the DBGEN data set (see Fig. 6). The five unassuming techniques have the same behaviour observed on the US Census data set. The model-based multifractal technique (see Fig. 6(d)) is especially accurate because DBGEN follows a uniform distribution [19]. For this reason, DBGEN is a poor tool to benchmark model-based view-estimation techniques, but this problem does not carry over to unassuming techniques since they are data-distribution oblivious.

We also performed experiments on very large data sets (5, 10, 20 and 30 GiB) generated by DBGEN. Table 2 shows that the accuracy is not sensitive to data and view sizes for small M. In addition, for very large views, Probabilistic Counting has a small edge in accuracy.

Table 2: Standard error over large data sets.

(a) Probabilistic Counting

		Memory budget		
Data size	**View size**	64	128	256
5 GiB	1000000	11%	8%	5%
10 GiB	2000000	10%	7%	6%
20 GiB	4000000	8%	6%	5%
30 GiB	6000000	9%	7%	7%

(b) Gibbons-Tirthapura

		Memory budget		
Data size	**View size**	64	128	256
5 GiB	1000000	10%	8%	7%
10 GiB	2000000	9%	7%	6%
20 GiB	4000000	10%	8%	6%
30 GiB	6000000	14%	8%	5%

5.1.2 Speed

The time needed to estimate the size of all the views by the unassuming techniques is about 5 minutes for the US Census 1990 data set and 7 minutes for the synthetic data set. For the multifractal technique, all the estimates are completed in roughly 2 seconds, but it takes 1 minute (resp. 4 minutes) to sample 0.5% of the US Census data set (resp. the synthetic data set – TPC H), in part because the data is not stored in a flat file. We ran further experiments on the data generated by DBGEN (with a scale factor equal to 5, i.e., 5 GiB of data) to highlight the time spent by each processing step: loading and parsing the data, hashing and computing estimated view sizes. As shown in Table 3, the running time of the algorithms is sensitive to the number of dimensions. For a low (resp. high) number of dimensions, relatively more time is spent reading data (resp. hashing data). However, the time spent hashing or reading is in turn much larger than the rest of the time spent by the algorithms (counting). This explains why all the unassuming estimation algorithms have similar running times and why timings are not sensitive to the memory parameter (M), as long as it is small.

5.2 Large Memory Budgets

When the memory budget is close to the view size, estimation techniques are not warranted. Hence, we did not use the US Census data set since it is too small.

Table 3: Wall-clock running times.

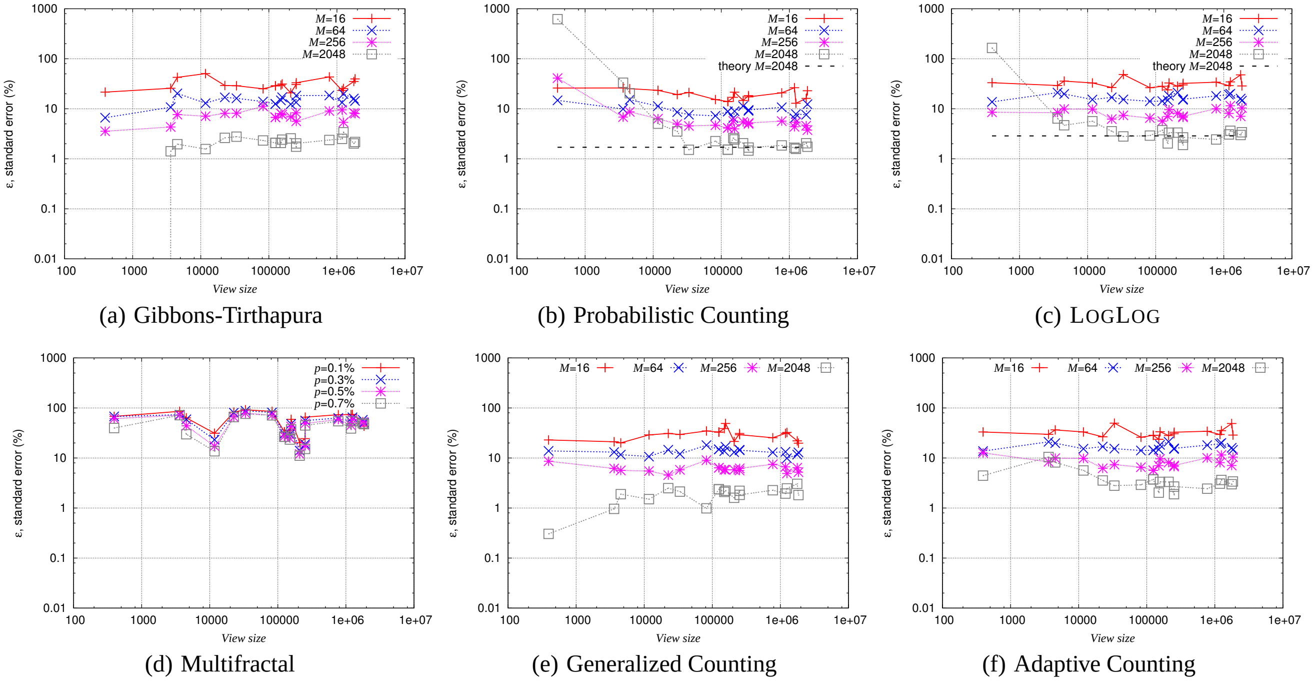
(a) Unidimensional view (view size $= 7.5 \times 10^5$)

		Loading		Hashing		Counting		Time (s)	
Memory		m_1	m_2	m_1	m_2	m_1	m_2	m_1	m_2
Technique	(1)	50%	52%	42%	45%	7%	3%	72	68
	(2)	54%	40%	45%	35%	1%	26%	68	90
	(3)	53%	52%	46%	45%	1%	3%	67	68
	(4)	54%	17%	46%	14%	–	69%	67	215
	(5)	54%	20%	46%	18%	–	62%	68	175

(b) tridimensional view (view size $= 2.4 \times 10^7$)

		Loading		Hashing		Counting		Time (s)	
Memory		m_1	m_2	m_1	m_2	m_1	m_2	m_1	m_2
Technique	(1)	29%	15%	68%	13%	3%	72%	239	240
	(2)	30%	13%	70%	11%	1%	76%	235	277
	(3)	29%	15%	71%	13%	–	72%	237	240
	(4)	30%	5%	70%	5%	1%	90%	235	652
	(5)	29%	6%	71%	5%	–	88%	238	576

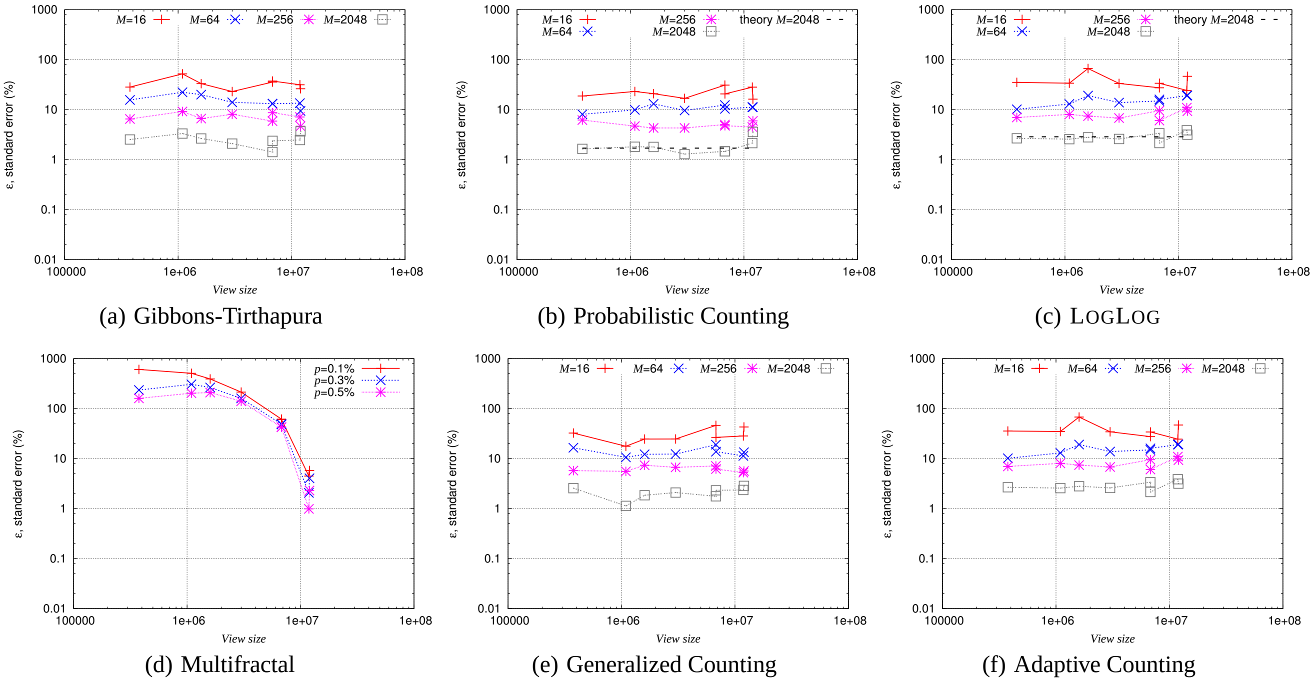
$m_1 = 256$ $m_2 = 8388608$
(1): LOGLOG (2): Probabilistic Counting (3): Adaptive Counting
(4): Gibbons-Tirthapura (5): Generalized Counting



(a) Gibbons-Tirthapura (b) Probabilistic Counting (c) LOGLOG

(d) Multifractal (e) Generalized Counting (f) Adaptive Counting

Figure 5: Standard error of estimation as a function of exact view size for increasing values of *M* (US Census 1990).



(a) Gibbons-Tirthapura (b) Probabilistic Counting (c) LOGLOG

(d) Multifractal (e) Generalized Counting (f) Adaptive Counting

Figure 6: Standard error of estimation as a function of exact view size for increasing values of *M* (synthetic data set).

5.2.1 Accuracy

Fig. 7 shows the behavior of the five probabilistic schemes over a moderately small synthetic unidimensional view. While all five schemes have similar accuracy when the memory budget is small relative to the size of the view, as soon as the memory budget is within an order of magnitude of the view size, they differ significantly: LOGLOG and Counting are no longer reliable whereas the three other schemes quickly achieve nearly exact estimates. As we increase the memory budget, this phenomenon happens somewhat later with LOGLOG than Counting. Adaptive Counting still has a

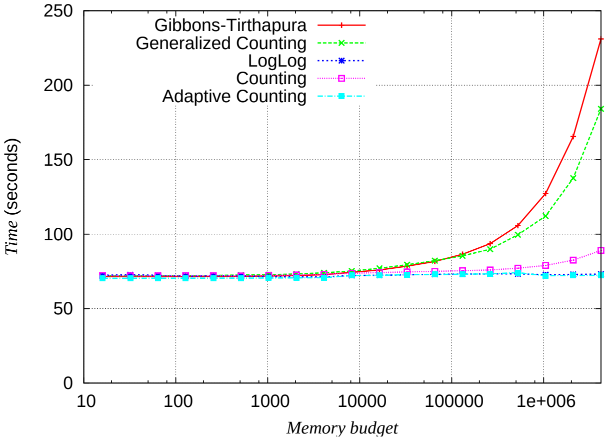


Figure 9: Estimation time for a given view (four dimensions and 1.18×10^7 distinct tuples) as a function of memory budgets M (synthetic data set).

good accuracy for large M because it switches from LOGLOG estimates to linear counting estimates [20] (see Algorithm 3). The accuracy of GC is limited by the size of L. Finally, we ran some tests over a large view using large values of M (see Fig. 8): these values of M still translate in memory usages well below 1 GiB. The main difference with the large view being that LOGLOG and Adaptive Counting performance seems to be substantially worst than Probabilistic Counting unless we increase the number of bits ($L = 64$).

5.2.2 Speed

We also computed the time required to estimate a large view using various memory budgets M (see Fig. 9). For small values of M ($M \leq 65536$) all techniques are equally fast: most processing time is spent hashing and parsing the data (see Table 2). For larger values of M, the time spent counting the hash values by GC and GT eventually dominates the processing time (see Table 3). Probabilistic Counting scales well with large values of M whereas LOGLOG does not slow down with increasing values of M, but their accuracies do not necessarily improve either. Adaptive Counting remains fast and gets increasingly accurate as M becomes large.

6. DISCUSSION

Our results show that Probabilistic Counting and LOGLOG do not entirely live up to their theoretical promise. For small view sizes relative to the available memory, the accuracy can be very low. One implication of this effect is that we cannot increase the accuracy of Probabilistic Counting and LOGLOG by adding more memory unless we are certain that all view sizes are very large. Meanwhile, we observed that GC, GT, and Adaptive Counting accuracies are independent of the view size and improve when more memory is allocated, though they also become slower, except for Adaptive Counting which remains constantly fast. When comparing the memory usage of the various techniques, we have to keep in mind that the memory parameter M can translate in different memory usage. The memory usage depends also on the number of dimensions of each view. Generally, GC and GT will use more memory for the same value of M than either Probabilistic Counting, Adaptive Counting, or LOGLOG, though all of these can be small compared to the memory usage of the look-up tables T_i used for k-wise independent hashing. When memory usage is not a concern (M and L large), GC, GT, and Adaptive Counting have accuracies better than 0.1%. For large values of M, which of GT and GC is more accurate depends on the number of hashing bits used (L). GC is the only scheme guaranteed to converge to the true view size as M grows. View-size estimation by sampling can take minutes when data is not laid out in a flat file or indexed, but the time required for an unassuming estimation is even higher. For small values of M, streaming and hashing the tuples accounts for most of the processing time so for faster estimates, we could store all hashed values in a bitmap (one per dimension).

7. CONCLUSION AND FUTURE WORK

We have provided unassuming techniques for view-size estimation in a data warehousing context. We adapted distinct count estimators to the view-size estimation problem. Using the standard error, we have demonstrated that among these techniques, GC, GT, and Adaptive Counting provide stable estimates irrespective of the size of views and that increasing the memory usage leads to more accuracy. For small memory budgets, all unassuming methods have comparable speeds. For large memory budgets, however, only Adaptive Counting remains constantly fast. For large view sizes, using more hashing bits ($L = 64$) is important, particularly when using Adaptive Counting.

There is ample room for future work. Firstly, we plan to extend these techniques to other types of aggregated views (for example, views including HAVING clauses including icebergs [9]). Secondly, we want to precompute the hashed values for fast view-size estimation. Furthermore, these techniques should be tested in a materialized view selection heuristic [3].

8. ACKNOWLEDGEMENTS

The authors wish to thank Owen Kaser from UNB for his contribution to the software and manuscript as well as Robert Godin from UQAM for his comments. This work is supported by NSERC grant 261437 and by FQRNT grant 112381.

9. REFERENCES

[1] N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. Algorithms*, 7(4):567–583, 1986.

[2] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *STOC '96*, pages 20–29, 1996.

[3] K. Aouiche, P. Jouve, and J. Darmont. Clustering-based materialized view selection in data warehouses. In *ADBIS'06*, volume 4152 of *LNCS*, pages 81–95, 2006.

[4] K. Aouiche and D. Lemire. Unassuming view-size estimation techniques in OLAP. In *ICEIS'07*, pages 145–150, 2007.

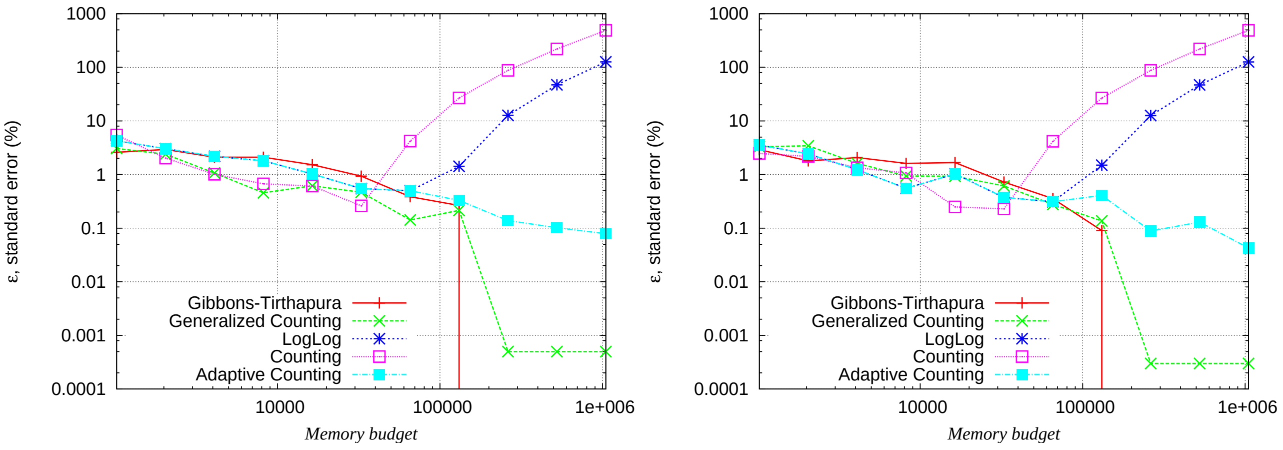
[5] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *RANDOM'02*, pages 1–10, 2002.

[6] M. Cai, J. Pan, Y.-K. Kwok, and K. Hwang. Fast and accurate traffic matrix measurement using adaptive cardinality counting. In *MineNet'05*, pages 205–206, 2005.

[7] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In *ESA'03*, volume 2832 of *LNCS*, pages 605–617, 2003.

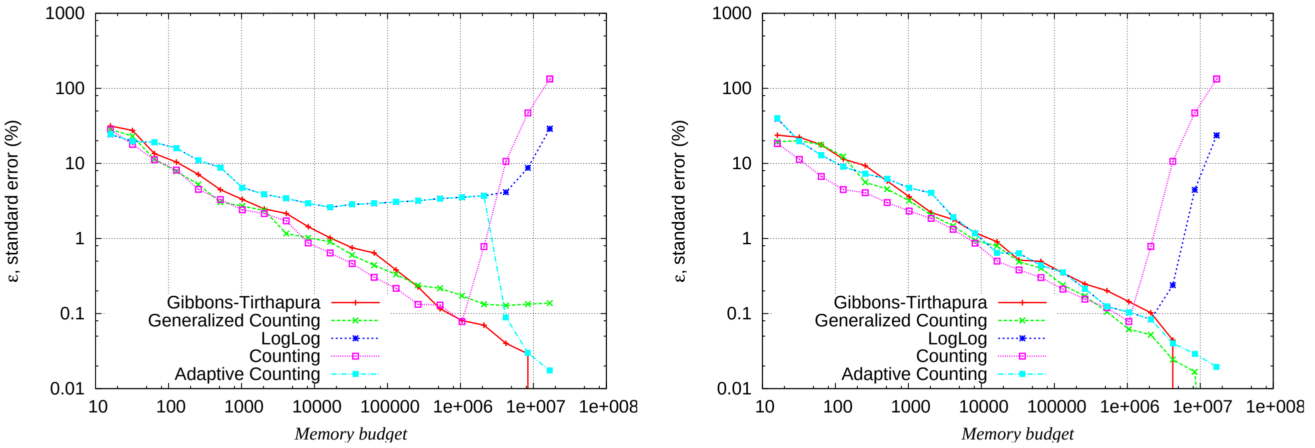
[8] C. Faloutsos, Y. Matias, and A. Silberschatz. Modeling skewed distribution using multifractals and the 80-20 law. In *VLDB'96*, pages 307–317, 1996.

[9] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *VLDB'98*, pages 299–310, 1998.



(a) $L = 32$ (b) $L = 64$

Figure 7: Standard error accuracy for a small unidimensional view (250,000 items) as a function of memory budgets M.



(a) L=32 (b) L=64

Figure 8: Standard error of estimation for a given view (four dimensions and 1.18×10^7 distinct tuples) as a function of memory budgets M (synthetic data set).

[10] P. Flajolet and G. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[11] P. B. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *SPAA'01*, pages 281–291, 2001.

[12] M. Golfarelli and S. Rizzi. A methodological framework for data warehouse design. In *DOLAP'98*, pages 3–9, 1998.

[13] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE '96*, pages 152–159, 1996.

[14] H. Gupta. Selection of views to materialize in a data warehouse. In *ICDT'97*, pages 98–112, 1997.

[15] P. Haas, J. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *VLDB'95*, pages 311–322, 1995.

[16] S. Hettich and S. D. Bay. The UCI KDD archive. `http://kdd.ics.uci.edu`, last checked on 23/10/2006, 2000.

[17] D. Lemire and O. Kaser. One-pass, one-hash n-gram count estimation. Technical Report TR-06-001, Dept. of CSAS, UNBSJ, 2006. available from `http://arxiv.org/abs/cs.DB/0610010`.

[18] J. Schmidt, A. Siegel, and A. Srinivasan. Chernoff-Hoeffding bounds for applications with limited independence. In *SODA'93*, pages 331–340, 1993.

[19] TPC. DBGEN 2.4.0. `http://www.tpc.org/tpch/`, last checked on 23/10/2006, 2006.

[20] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst.*, 15(2):208–229, 1990.