DEIM Forum 2018 I4-4

# Random Clustering for Multiple Sampling Units to Speed Up Run-time Sample Generation

Yuzuru OKAJIMA† and Koichi MARUYAMA†

† NEC Solution Innovators, Ltd. 1-18-7 Shinkiba, Koto-ku, Tokyo, 136-8627 Japan
E-mail: y-okajima@bu.jp.nec.com, kou-maruyama@sx.jp.nec.com

**Abstract** Random sample generation from a database can be helpful to exploratory data analysis because of its capability of fast approximate aggregation of large data. For efficient run-time random sample generation, random clustering, a technique to randomly sort table rows on disk in advance, has been extensively used in previous research. However, this technique can randomize rows of a table only for a single unit. Thus, if data analysts require sampling for other units than the one used for clustering, traditional single-unit random clustering is no longer of help and causes a significant slowdown due to many random disk accesses. In this paper, we propose a random clustering technique for multiple units rather than for a single unit. This technique can randomize table rows for multiple units on disk. After clustering, a data analyst can select any one of the units and then samples for the selected unit can be efficiently read from disk until one satisfies the given filtering criteria and size condition. Experiments show that the new random clustering technique for multiple units generated samples significantly faster than traditional single-unit random clustering.

**Key words** Random sampling from databases, Exploratory data analysis, Online aggregation

## 1. Introduction

Random sampling from a database has been studied for a long time, and it is gaining even more significance owing to the emerging demand for large data analyses. Calculating exact aggregates over massive data requires long execution times and vast amounts of resources. Sampling can reduce such costs by calculating approximate aggregates over a small amount of data extracted from the original tables.

Run-time sample generation has been used in approximate query processing (AQP) as well as sample precomputing. In precomputing approaches, samples are precomputed and stored apart from the original data so that approximate aggregates can be calculated from the samples. On the other hand, run-time generation approaches do not store such precomputed samples. They extract rows in random order at query time and calculate approximate answers from them. An advantage of run-time approaches is their high adaptability to ad-hoc queries. Run-time approaches can generate a sample of appropriate size for estimating the answer to a given query by continuously expanding the sample size until it becomes sufficient. This property has made run-time sample generation a key component of *online aggregation* [8], which is a framework for presenting a running estimate based on a sample until the user is satisfied with it.

The time efficiency of run-time sample generation depends on the physical data layout on disk. Without any assumption, randomly sampled rows may be distributed over a large part of the disk-resident original table, wherein the random disk accesses substantially slow the sampling process. To avoid this problem, the first online aggregation paper [8] proposed the *random clustering* technique, which randomly sorts rows in a disk-resident table in a pre-processing phase. If rows are randomly clustered, a sample can be efficiently obtained through a sequential scan because rows in continuous blocks can be considered a random sample and this sample is easily expanded by scanning more blocks. Since then, random clustering has come to be viewed in the online aggregation literature, e.g., [7], [9], [13], as a key technique for accelerating the run-time sample generation process.

AQP approaches based on sampling have been developed to present approximate answers by automatically choosing or generating samples in the background. However, simply giving approximate answers is not sufficient for data analysts familiar with advanced statistical analysis tools. Most statistical analysis tools today have more sophisticated analysis and visualization functions than the aggregate functions executable in database systems. As analysts would want to take full advantage of their favorite tools, they would prefer simply getting samples according to the requirements they give explicitly for further analysis in their own tools, instead of only getting approximate answers.

Generating samples according to explicitly given requirements is a challenging task, as is AQP. In exploratory data analysis, analysts need to perform analyses iteratively and from different perspectives; thus, the sampling unit of concern may be different in each trial. However, random sampling using different sampling units causes a severe slowdown even if the data are randomly clustered. This is because the traditional random clustering technique can only randomize a table according to a single unit (in most cases, a row) and so we call the traditional technique *single-unit random clustering*. If analysts choose other units for sampling than the one used for clustering, traditional single-unit random clustering is no longer of help and causes a significant slowdown due to many random disk accesses. We call this the *unit mismatch problem*. This problem is inherent to single-unit random clustering, and any online aggregation system based on traditional random clustering may suffer from this problem when the aggregation needs to use different units.

In this paper, to avoid the unit mismatch problem, we propose a random clustering technique for multiple sampling units rather than a single unit. Our multi-unit random clustering technique can randomize a table for several sampling units in advance so that samples of different sizes for any one of the units can be read from disk quickly. If a data analyst gives her requirements as to the target sampling unit, the filtering criteria and the size condition, we can iterate the sample generation by increasing the sample size until the sample satisfies the given requirements.

Experiments show that our multi-unit random clustering generate samples for multiple units significantly faster than traditional single-unit random clustering because the traditional method is affected by the unit mismatch problem. By iterative sampling, samples satisfying the given requirements were efficiently generated from the database clustered using our technique even though the requirements involved sampling of distinct values in a foreign key (typical examples that suffer from the unit mismatch problem). The source code of the sampling system is available online (https://github.com/nec-solutioninnovators-ilab/sampling-sql).

The rest of this paper is organized as follows. In Section 2, we discuss the unit mismatch problem. Section 3 explains the multi-unit clustering technique, the iterative sampling algorithm that returns samples according to analytical requirements and the system that executes the iterative sampling. In Section 4, we experimentally evaluate our approach. We discuss related work in Section 5 and give our conclusions in Section 6.

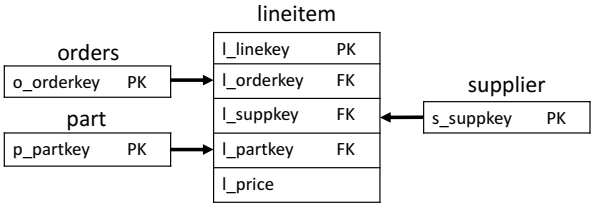


Figure 1 Example database for illustrating the unit mismatch problem. PK represents a primary key, and FK represents a foreign key.

## 2. Unit Mismatch Problem

The unit mismatch problem is a problem of many random disk accesses that occur when the sampling unit differs from the unit of random clustering. To illustrate this problem, let us consider the example database shown in Figure 1. Each row in table lineitem represents a line item of a transactional order and has a primary key l_linekey and foreign keys corresponding to the primary keys of three other tables that represent orders, parts, and suppliers related to the line items. lineitem and the other three tables have many-to-one relationships: An order (part or supplier) may be related to more than one row in lineitem. In addition, we assume that all tables are so large that sampling is needed to analyze them. This means that this database has four entity types that can be chosen as the sampling unit: a line item, an order, a part, and a supplier.

Let us see what happens when all the tables are randomly clustered in the typical way, i.e., when all rows are randomly shuffled in advance. If the analyst wants to estimate the average sales *per line item* by sampling from table lineitem, the sampling can be done efficiently by making a sequential scan of the first rows in lineitem. However, If the analyst next wants to estimate the average total sales *per order*, she needs to randomly choose a small number of orders, collect all line items related to the sampled orders, and then average the sum of the prices for the orders. The orders can be efficiently sampled by making a sequential scan of the first rows in table order. However, the rows related to each order are randomly distributed over table lineitem because the rows in lineitem are randomly shuffled and there is no guarantee that rows having the same l_orderkey value can be found in adjacent disk blocks. Thus, random sampling considering an order as the sampling unit is slowed down by many random disk accesses to lineitem.

In general, by using traditional random clustering, a table can be clustered for only a single unit, and many random disk accesses can be avoided only if the sampling unit matches the unit of the clustering. If one of the other units is chosen as

the sampling unit, random disk accesses are not reduced. Thus, the purpose of this paper is to overcome this problem.

## 3. Multi-unit Clustering

### 3.1 Clustering Procedure

We explain how rows can be randomly clustered for multiple sampling units. Large tables generally have several attributes such that the distinct values in each of them represent numerous entities such as IP addresses or URLs and can be used as sampling units. We call such attributes *unit keys* and our multi-unit clustering technique clusters rows in a table so that we can efficiently sample rows considering the distinct values of the chosen unit keys as the sampling units. In Figure 1, for example, l_linekey, l_orderkey, l_partkey and l_suppkey are good choices for the unit keys of lineitem. Our technique leverages compound clustering indexes, which are supported by many database systems. First, we designate several attributes as the unit keys. Then, we set compound clustering keys in the tables by appending several attributes derived from the given unit keys. We use a pairwise independent hash function $h(v)$ that maps each distinct value $v$ of the unit keys to $[0, 2^L - 1]$. We assigned each distinct value $v$ of the unit keys a *level*:$Level(v) = L - \lfloor \lg h(v) \rfloor - 1$ if $h(v) > 0$ and $Level(v) = L$ for $h(v) = 0$. The minimum level is 0, and the maximum is $L$. The probability that the level of $v$ is at least $l \in [0, L]$ is $Pr(Level(v) \geqq l) = 2^{-l}$. This level function is the same as the one used in distinct sampling [6] to keep the sample size small. We use it to cluster the original table in preparation for sampling using different units. If the table has $U$ unit keys, we append $U$ *level attributes* that store the levels for the unit keys and append a *level sum attribute* that stores the sum of the $U$ levels. Then, we cluster the table by setting a compound clustering key in it. The first component of the key is the level sum attribute, and the successive components are the level attributes. If the table was already assigned a clustering key, the original clustering key is appended as the last component of the new clustering key. For example, if table orders has one clustering key o_orderdate and three unit keys o_orderkey, o_custkey and o_productkey, we set a new clustering key composed of (level_sum, o_orderkey_level, o_custkey_level, o_productkey_level, o_orderdate) after appending the level sum attribute and the three level attributes.

### 3.2 Extract Samples of Different Sizes

Let us see how samples of different sizes can be extracted from a clustered table. First, one of the unit keys of the table is chosen as the *target unit key* that represents the sampling unit. Then, the *target level* $l \in [0, L]$ is chosen to determine the sample size. All distinct values of the target unit key that have levels being at least $l$ are sampled. A sample table $s$ of a target table $t$ is a subset of $t$ that consists of rows in $t$ that have the sampled values in the target unit key of $t$. For example, if the analyst wants to extract a sample from table lineitem in Figure 1 considering an order as the sampling unit, then the target unit key is l_orderkey and sample table l_sample is defined as follows:

```
WITH l_sample AS (SELECT * FROM lineitem
WHERE l_orderkey_level ≧ l)
```

l_orderkey_level stores the levels of l_orderkey. The sample for $l = L$ is the smallest sample, and the sample size increases exponentially as $l$ decreases. Finally, the sample for $l = 0$ contains all the rows of the original table.

The distinct values of the target unit key in a sample table are selected on the basis of the pairwise independent hash function $h(v)$ and thus these selected values can be seen as a random sample of the distinct values of the target unit key in the original table.Thus, the data analyst can estimate the properties about the target unit in the original table by only analyzing a sample table derived from it. For example, the average total sales per order in lineitem can be estimated using the average total sales per order in l_sample because all distinct orders are sampled with the same probability.

We will see that the sample tables of different target levels can be read from disk efficiently. To discuss the efficiency, we call a set of rows sharing the same set of levels in a clustered table a *bucket*. A bucket can be viewed as an I/O unit. All rows in a bucket share the same prefix in their clustering keys, and thus, they are contiguously clustered. If a row in a bucket is included in a sample table, all other rows in the bucket are also included.

The single-unit random clustering suffers from the unit mismatch problem because rows can be sorted in only one random order while each row is related to multiple units. Thus, our approach uses random bucketing rather than ordering. The rows related to the units of the same level are divided into multiple buckets on disk. Given the target unit at query time, our algorithm restores these rows by selecting buckets related to the target level.

Sample extraction is surely efficient if the target table has the target unit key as its only unit key. In that case, all rows are sorted only by the levels of that key and collecting rows above a certain level is efficiently done by making a sequential scan. However, if the table has extra unit keys other than the target unit key, the rows having the same level of the target unit key are divided into many smaller buckets because of the existence of the levels of the other unit keys.

Therefore, we need to estimate the effect of the number of unit keys on the efficiency of our sample generation. For a table having $N$ rows, the expected size of a bucket with levels $(l_0, l_1, ..., l_{U-1})$ is $N \prod_{u=0}^{U-1} 2^{-1-l_u} = N 2^{-U-\sum_{u=0}^{U-1} l_u}$.

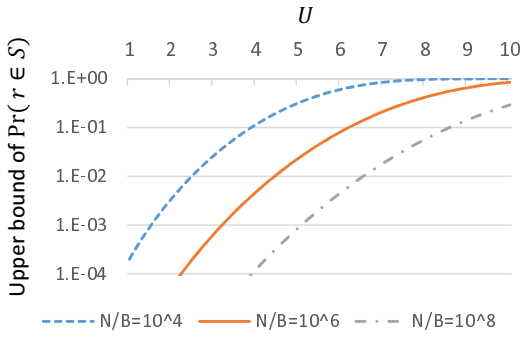


Figure 2 Upper bound of the probability that a row is in a mini-bucket

This is a monotonic function of the level sum. Our technique clusters rows in ascending order of the level sum, i.e., their expected size. Thus, the smallest buckets are probabilistically clustered in the blocks at the bottom of the table.

We assume that each disk block stores $B$ rows. Buckets having at least $B$ rows can be efficiently read from disk because they are stored in at least one contiguous block. On the other hand, buckets with fewer than $B$ rows may reduce the disk-read efficiency. A bucket is called a *mini-bucket* if the expected number of its rows is smaller than $B$. Here, we show that mini-buckets actually occupy a small fraction of the whole table and do not significantly reduce efficiency if the table has many rows and a few unit keys.

Consider a row $r$ in a table $t$ of $N$ rows with $U$ unit keys. Denote by $\mathcal{S}$ the set of all rows in mini-buckets of $t$. The probability that $r \in \mathcal{S}$ is bounded as follows:

$$\frac{\Gamma(U, \ln(N/B))}{(U-1)!} \leqq Pr(r \in \mathcal{S}) < \frac{\Gamma(U, \ln(N/B) - U\ln 2)}{(U-1)!} = P_{ub}(U, N/B)$$

where $\Gamma(a, x)$ is the upper incomplete gamma function. The upper bound $P_{ub}(U, N/B)$ is plotted for varying $U$ and $N/B$ in Figure 2. This graph shows that the probability is reasonably small if $N/B$ is large and $U$ is small. For example, the probability is smaller than $4.8 \times 10^{-3}$ for $N/B = 10^6$ and $U = 4$. The probability is equal to the expected ratio of the number of rows in the mini-buckets to $N$. Thus, the mini-buckets occupy fewer than $N/B \cdot P_{ub}(U, N/B)$ blocks in expectation. Thus, our algorithm can collect buckets of the same level with a small overhead caused by reading the blocks storing mini-buckets.

The expected size of the smallest sample table, which is obtained at $l = L$, is $N2^{-L}$. Thus, $L$ should be at least $O(\log N)$, which makes the expected size of the smallest sample constant. If $L$ is larger than $O(\log N)$, the choice of $L$ does not affect the efficiency, because the samples for $l > O(\log N)$ are very small and clustered in the blocks storing mini-buckets and do not cause many random accesses.

### 3.3 Iterative Sampling

We showed that sample tables of different target levels can be efficiently extracted from a clustered table. However, the sample table probably still includes data irrelevant to the analyses, and the data analyst needs to filter out such data by the filtering criteria. If the remaining data after filtering are too small for statistical analysis, the analyst needs to get larger samples by setting lower target levels. To reduce the burden, we give an algorithm that automatically chooses samples satisfying her requirements, i.e., the filtering criteria and size condition, as well as the target sampling unit. This algorithm iteratively extracts samples from multiple tables by decreasing the target level until the requirements are met.

We define how the analyst can represent her requirements as the input to the system. First, the target sampling unit is represented by choosing one of the unit keys for each target table to be sampled. For example, if sampling of orders is needed, o_orderkey for table orders and l_orderkey for lineitem are chosen as the target unit keys. Next, we introduce the notion of *filtered table* to describe the filtering criteria and size condition. A filtered table is a table derived from sample tables and, if needed, other tables. The filtering criteria is described as a selection from the sample tables to the filtered table. If the analyst wants to exclude a certain kind of unit, she defines the filtered table so that rows related to such units are excluded. The size condition can be described as an aggregation over the filtered table because the filtered table only contains the units that pass the filtered criteria.

Given these requirements, we return a sample of sufficient size by decreasing the target level until the given requirements are satisfied. The pseudo-code is given in Algorithm 1. We are given a query that contains $T$ pairs of clustered tables and their unit keys $(t_0, u_0)$, $(t_1, u_1)$, ..., $(t_{T-1}, u_{T-1})$, filtering criteria $f$, size condition $c$, and main query $q$. First, we initialize the target level $l$ to $L$. For each clustered table $t_i$, we define a sample table $s_i$ as the set of all rows in $t_i$ whose levels of $u_i$ are at least $l$. Next, we ask the database system if the size condition $c$ is satisfied by submitting a query with the definitions of the sample tables and the filtering criteria (*checkCondition* in Algorithm 1). The database system creates the sample tables, filters them with the given criteria, and returns a truth value indicating whether the filtered tables satisfy the size condition. If the condition is not satisfied, we decrement $l$ by one, redefine the sample tables, and submit a size condition query again. The size condition check is repeated until the condition is satisfied or $l$ reaches 0. Finally, we submit the main query $q$ to the database system with the final definitions of the sample tables (*runMainQuery*). If the sampling process reaches the final step $l = 0$, the sample tables are the same as their

**Algorithm 1** Sampling(pairs of a target table and its unit key $(t_0, u_0), ..., (t_{T-1}, u_{T-1})$, filtering criteria $f$, size condition $c$, main query $q$)

```
for l = L to 0 do
  for i = 0 to T − 1 do
    define sample table s_i as the set of all rows in t_i whose
    level of u_i is at least l
  if checkCondition(c, f, s_0, ..., s_{T−1}) then
    break
return runMainQuery(q, f, s_0, ..., s_{T−1})
```

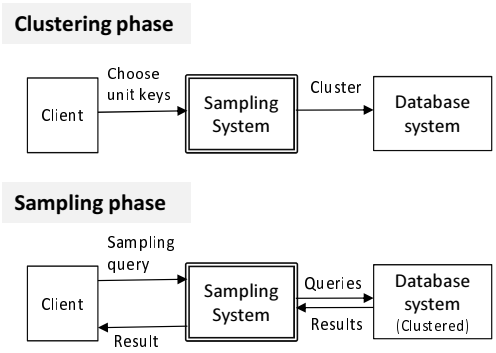


Figure 3 Overview of our sampling architecture

original tables and the main query is exactly answered.

The distinct values of the target unit keys are randomly sampled by the hash function. Thus, if one of the values is included in the sample of one table, all rows related to the same value must be included in the samples of other tables. This assures that we can safely join the sample tables on the sampled values of the target unit keys without lacking related rows. This capability of joining on sampled values is a major advantage of our algorithm.

### 3.4 Sampling System

We here present a sampling system that can bring out the strengths of our clustering technique. This system receives sampling queries from data analysts and runs the iterative sampling algorithm according to it. An overview is shown in Figure 3. To simplify the explanation, we assume that we are given a database system that already stores the data the analyst wants to examine. Our sampling system works as a proxy server between the client application the analyst operates and the database system. It first clusters the data of the tables in the database system (the clustering phase) and then waits for sampling queries to be submitted by the analyst (the sampling phase). In the clustering phase, the analyst sends a clustering request to the sampling system through the client application. In this request, the analyst designates several attributes in large tables as unit keys. Then, the sampling system clusters the data of the tables in the database system by using multi-unit clustering. After the clustering finishes, the sampling system waits for sampling queries to be submitted by the analyst. Once it receives a query, the sampling system runs the iterative sampling algorithm and returns the result to the analyst. The clustering process generally takes a long time, because all rows are sorted on disk. However, once the clustering is done, subsequent sampling queries can be answered in shorter amounts of time.

We extend the SQL grammar so that the analyst can describe her requirements. Figure 4 shows our extended SQL grammar, and Figure 5 shows an example query. The requirements are represented by SAMPLE, WITH and UNTIL clauses, and they are inserted before the main query block. The SAMPLE clause is used to choose which clustered tables to sample and the target unit key for each of them. The names of the sample tables are specified as AS clauses. If they are omitted, the name of the original clustered table is used to refer to its sample table in the following clauses. The WITH clause is used to define filtered tables derived from the sample tables. The filtered tables can be used later in the UNTIL clause and the main query. The WITH clause can be omitted if no filtered tables are necessary. The UNTIL clause specifies the size condition to be satisfied by the sample and filtered tables. Once the condition is satisfied, the main query following the sampling requirements is executed using the sample and filtered tables. Figure 5 shows an example query with the sampling requirements. The query is based on the database schema of the TPC-H benchmark. It calculates the average total price per customer who lived in Japan and ordered in 2015.

## 4. Experimental Evaluation

In the experiments, we compared our multi-unit random clustering with traditional single-unit random clustering.

### 4.1 Setting

The database system used in the experiments was Amazon Redshift dc2.xlarge single node. We used the TPC-H benchmark datasets with a scale factor of 300. To simplify the explanation, we modified table lineitem so that it would have a primary key attribute l_linekey that stores row ids because the original lineitem does not have a single primary key. After that, we created three identical copies of the TPC-H database in the same system and clustered these three databases using different clustering techniques. We set $L = 31$ because it is large enough to keep the smallest sample small. All unit keys had integer values, and we used a pairwise independent hash function of the form $h(v) = (a \cdot v + b) \bmod p \mod 2^L$.

In the first database, MU, each table was clustered using

⟨query⟩ ::= [⟨sampling requirements⟩] ⟨main query⟩
⟨sampling requirements⟩ ::= ⟨sample clause⟩ [⟨with clause⟩] ⟨until clause⟩
⟨sample clause⟩ ::= SAMPLE *clustered_table* [AS *sample_table*] BY *target_unit_key*
{"," *clustered_table* [AS *sample_table*] BY *target_unit_key*}
⟨until clause⟩ ::= UNTIL *size_condition*

Figure 4 SQL grammar with sampling requirements

```
SAMPLE orders AS o_sample BY o_custkey, customer AS c_sample BY c_custkey                    ... (1)
WITH filtered AS (SELECT * FROM o_sample, c_sample, nation WHERE o_custkey = c_custkey AND
  c_nationkey = n_nationkey AND n_nation = 'JAPAN' AND o_orderdate BETWEEN '2015-01-01' AND '2015-12-31')   ... (2)
UNTIL 1000 ≦ (SELECT COUNT(DISTINCT o_custkey) FROM filtered)                                ... (3)
SELECT AVG(sum) FROM (SELECT SUM(price) FROM filtered GROUP BY o_custkey)                    ... (4)
```

Figure 5 Example query that calculates the average total sales in 2015 per customer who lived in Japan

multi-unit random clustering. We designated all primary keys and foreign keys as unit keys for all tables. As a result, lineitem was assigned four unit keys: l_linekey, l_orderkey, l_partkey, and l_suppkey. Next, level and level sum attributes were appended to all tables, and the tables were clustered in the manner explained in Section 3. 1.

In the second database, MU*, each table was clustered using multi-unit random clustering in the same way as MU except that the level sum was not included in the clustering key. This means the buckets in the tables of MU* were not arranged in order of expected size. Thus, unlike MU, MU* did not have the benefit of clustering small buckets.

The third database, SU, was a database that shows the effect of the traditional single-unit random clustering technique that considers a row as the sampling unit. SU was basically the same as MU, but we changed the level attributes *_level to hash attributes *_hash that stored raw hash values $h(v)$ instead of $Level(v)$. Each table was clustered using the hash attribute of the primary key. The tables in this database can be viewed as having been randomly clustered because the rows were shuffled according to the hash values.

These three databases contained exactly the same data except the newly appended attributes for clustering. In the following experiments, we executed identical queries over these databases. The databases returned the same results, but had different query performances because their tables were clustered using different techniques.

### 4. 2 Experiment 1: Sample Table Generation

We evaluated the efficiency of extracting a sample table from a clustered table for different target levels by directly querying the database system with traditional SQL queries. We chose lineitem as the target table from which the sample tables were generated, because lineitem is the largest table in the TPC-H dataset and it has four unit keys, the largest number among the tables. Queries were executed with 'cold cache'; i.e., all selected rows were read from disk. We submitted the following queries to the three databases.

A SELECT COUNT(DISTINCT *unitkey*) FROM lineitem WHERE $unitkey\_level \geqq l$

B SELECT COUNT(DISTINCT *unitkey*) FROM lineitem WHERE $unitkey\_hash < 2^{L-l}$

where $l$ is an integer in $[0, L]$. A is for MU and MU*, and B is for SU. Both queries return the number of units in the sample table, i.e., the sample size. $l$ is a parameter to control the sample size. Upon receiving these queries for the same $l$, the three databases must return the same sample size because the rows satisfying $unitkey\_level \geqq l$ in the MU and MU* are equal to the rows satisfying $unitkey\_hash < 2^{L-l}$ in the SU.

The results are shown in Figure 6. We varied $l$ from L to 0 and plotted the query time versus the sample size. The graph for l_linekey exemplifies the case that the sampling unit matches the unit of single-unit random clustering. For this key, the three databases showed almost the same performance. For the other three keys, however, SU performed significantly worse than l_linekey while MU still worked efficiently, as well as l_linekey. This result shows how the unit mismatch problem degrades the efficiency of single-unit random clustering and that multi-unit random clustering can avoid this problem. The difference between SU and MU was large when the sample size was moderate, but was small in the minimum and maximum limit because the minimum sample required a constant query processing cost and the maximum sample required a full scan of the table.

MU performed more efficiently than MU* for l_partkey and l_suppkey. This difference shows the effect of using the level sum as the first element of the clustering key, which arranges the smallest buckets in contiguous blocks.

### 4. 3 Experiment 2: Iterative Sampling

In Experiment 2, we evaluated the efficiency of the iterative sampling algorithm by submitting sampling queries to our sampling system. We designed the sampling queries on the basis of three TPC-H queries: Q4, Q13, and Q17. These

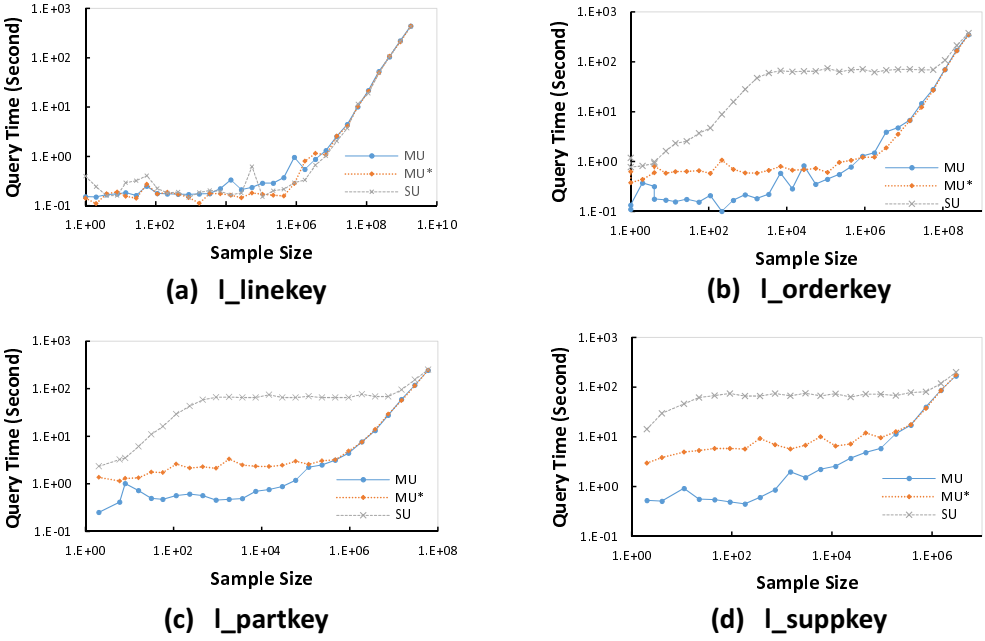


Figure 6 Query times required to generate sample tables of varying sizes from the three databases with respect to the four unit keys in lineitem.

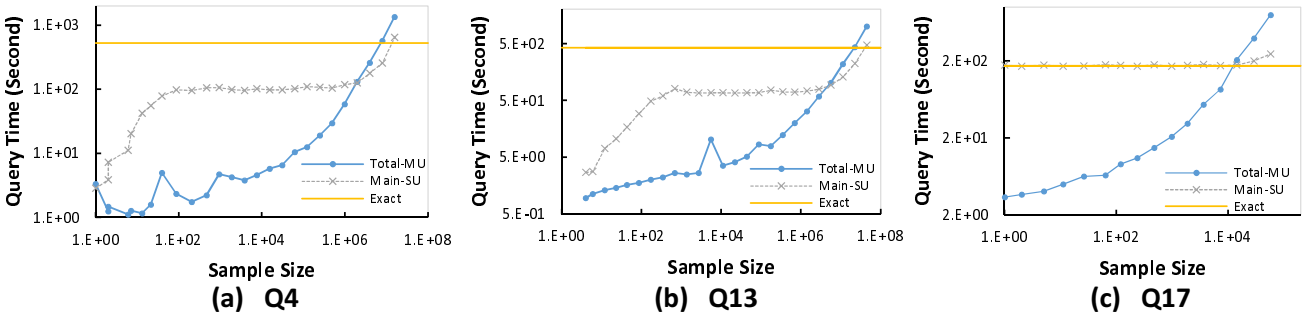


Figure 7 Query times required to answer sampling queries based on TPC-H queries Q4, Q13, and Q17 for varying sample sizes.

queries are typical examples that suffer from the unit mismatch problem because their aggregation units are numerically distinct values in foreign keys. We rewrote these queries in extended SQL by appending sampling requirements. The target sampling unit and the filtering criteria were determined according to the nature of the queries. For all queries, we set the size condition so as to stop sampling when $l = \theta$, where $\theta$ is an integer scaling parameter in $[0, L]$. For each $\theta$, the sampling system ran size condition queries from $l = L$ to $l = \theta$, then performed the main query for $l = \theta$.

For each $\theta$, we compared the performance of our iterative sampling in the MU database with the performance of directly running the main query for the final sample with $l = \theta$ in the SU database without running condition queries. To perform sampling in SU, the main query was executed with tables restricted by $unitkey\_hash < 2^{L-l}$ instead of $unitkey\_level \geqq l$ in the same manner as Experiment 1. The results are shown in Figure 7. Total-MU represents the costs to run iterative sampling on the MU database, i.e., the total costs to run the condition queries and the main queries. Main-SU represents the costs to run the main queries for the final sample in the SU database. Exact represents the costs to get exact answers by running the original TPC-H queries. Although the costs of iteratively running condition queries were included in Total-MU, Total-MU was still significantly more efficient than Main-SU except when the sample size was very large. For Q17, a large number of blocks were read in Main-SU even when the sample size was small because the database system chose to join tables to avoid inefficient random accesses. Nevertheless, Total-MU still worked well for Q17 because the sampled rows were clustered well. These results show that samples can be efficiently extracted from the tables clustered by multi-unit random clustering.

## 5. Related Work

**Sample generation.** There are a number of tech-

niques for randomly sampling rows from disk-resident files or databases. They can be classified as to whether the disk access method is a sequential scan or random access. Reservoir sampling [12] is a typical example of sequential techniques. Among random techniques, the technique proposed by Olkan and Rotem generates samples using an indexing structure like B+-tree [10] or hash file [11]. A drawback of the sequential-scan-based techniques is to need to process the whole data to get a random sample. Random techniques can avoid having to make a whole scan, but still require, at maximum, as many random block accesses as the sample size. To reduce I/Os, several studies randomly collect *blocks* instead of rows, e.g. [5]. However, rows in the same block may have a strong correlation with each other and cannot be seen as an independent random sample.

The most closely related approach to ours is distinct sampling [6], which generates samples by sequential scan considering distinct values of a target attribute as the sampling units and using them to answer queries asking for the number of distinct values. Distinct sampling can deal with different sampling units, but needs to scan the whole data to generate samples, and it only deals with distinct value queries.

**AQP based on precomputed samples.** AQP systems typically store precomputed synopses including samples, histograms, and wavelets for approximate aggregation. AQUA [2] stores join synopses [3] for foreign key joins and congressional samples [1] for group-by queries. BlinkDB [4] is an AQP system storing stratified samples.

**Online Aggregation.** Online aggregation [8] is an AQP framework based on runtime sample generation. It presents to the user a running estimate with an error bound based on the samples retrieved so far. The first online aggregation paper [8] considered aggregation in a single table. This approach has been extended to multi-table joins [7], large tables [9] and nested queries [13]. The single-unit random clustering technique has been used for online aggregation in many studies [7], [9], [13], but it cannot work efficiently if the sampling unit does not match the unit of random clustering. Our clustering technique is a solution to this problem.

## 6. Conclusion

We proposed a random clustering technique that enables fast sample generation from disk-resident tables for multiple sampling units. It is a solution to the unit mismatch problem inherent to single-unit random clustering, a traditional technique used for run-time sample generation. Our technique can be used to help data analysts by generating a sample until it satisfies their requirements. Experimental results showed our multi-unit clustering worked equally efficiently for different sampling units, whereas single-unit random clustering worked well only for a single unit and performed significantly worse for other units because of the unit mismatch problem.

## References


[1] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 487–498, New York, NY, USA, 2000. ACM.

[2] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The aqua approximate query answering system. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 574–576, New York, NY, USA, 1999. ACM.

[3] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 275–286, New York, NY, USA, 1999. ACM.

[4] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42, New York, NY, USA, 2013. ACM.

[5] S. Chaudhuri, G. Das, and U. Srivastava. Effective use of block-level sampling in statistics estimation. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 287–298, New York, NY, USA, 2004. ACM.

[6] P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 541–550, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[7] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 287–298, New York, NY, USA, 1999. ACM.

[8] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 171–182, New York, NY, USA, 1997. ACM.

[9] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the dbo engine. *ACM Trans. Database Syst.*, 33(4):23:1–23:54, Dec. 2008.

[10] F. Olken and D. Rotem. Random sampling from b+ trees. In *Proceedings of the 15th International Conference on Very Large Data Bases*, VLDB '89, pages 269–277, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[11] F. Olken, D. Rotem, and P. Xu. Random sampling from hash files. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 375–386, New York, NY, USA, 1990. ACM.

[12] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, Mar. 1985.

[13] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica. G-OLA: Generalized on-line aggregation for interactive analysis on big data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 913–918, New York, NY, USA, 2015. ACM.