Limiting Result Cardinalities for Multidatabase Queries using Histograms*

Kai-Uwe Sattler[1], Oliver Dunemann[1], Ingolf Geist[1], Gunter Saake[1], and Stefan Conrad[2]

[1] Department of Computer Science, University of Magdeburg
P.O. Box 4120, D-39016 Magdeburg, Germany
fusion@cs.uni-magdeburg.de
[2] Department of Computer Science, University of Munich
Oettingenstr. 67, D-80538 München, Germany
conrad@dbs.informatik.uni-muenchen.de

Abstract. Integrating, cleaning and analyzing data from heterogeneous sources is often complicated by the large amounts of data and its physical distribution which can result in poor query response time. One approach to speed up the processing is to reduce the cardinality of results – either by querying only the first tuples or by obtaining a sample for further processing. In this paper we address the processing of such queries in a multidatabase environment. We discuss implementations of the query operators, strategies for their placement in a query plan and particularly the usage of histograms for estimating attribute value distributions and result cardinalities in order to parameterize the operators.

Keywords: Result Cardinality, Histograms, Multidatabase, Optimization

1 Introduction

In a large number of application areas integration of data from heterogeneous sources is required, e.g., for building federated information systems or data warehouses. Besides integration of data there is often also a need for cleaning and analyzing the data in order to obtain qualitatively appropriate results.

By means of multidatabase languages (e.g MSQL [GLRS93], SchemaSQL [LSS96], FRAQL [SCS00]) we have the tools at hand which are needed for querying across diverse data sources. Querying using multiple data sources usually produces complete result sets. This requires processing large amounts of data and, thus, results in very poor query response time taking the physical distribution of the data into account.

In data analysis applications such as OLAP, data mining or information fusion we often want to get the 'first' results quickly, e.g., in order to find interesting regions in data or to parameterize the methods and tools. Another example is the case of identifying conflicting values for semantically related attributes stored in different databases. One does not want to obtain all conflicting pairs of values because there might be too many of them for manual inspection. Instead, a certain number of conflicting pairs

* Research was supported by the grant FOR 345/1 from the DFG.

given as examples can help to understand the basic problem (e.g. different scaling of values in the different data sources). Then, adding a corresponding conflict reconciliation function to the multidatabase query used for detecting this conflict should show whether there are no further conflicts (i.e., the reconciliation function is working for all data) or whether we have to modify the reconciliation function in order to capture more conflicts.

Thus, if multidatabase features are combined with techniques for reducing query response time by limiting result cardinalities, more explorative and interactive data integration and analysis will be possible. Unfortunately, all multidatabase languages so far proposed do not allow requests for a specified number of resulting tuples as examples instead of the complete result set. Therefore, we are seeking suitable extensions of multidatabase languages leading to efficient retrieval of example data. In this paper we will explore two ways of getting such example data:

1. asking for the *first* n results of an integrating multidatabase query and
2. asking for a *sample* containing n tuples of the complete result (or for a certain percentage of resulting tuples).

These two possibilities have already been considered in other contexts. For instance asking for the first n (or the best n) results is typical for information retrieval. Much of the work regarding this subject proposes optimization for evaluating such queries (cf. e.g. [CK97,TGO99]). In contrast to these approaches we have to deal with the problem that in a multidatabase environment there are usually legacy systems acting as local data sources.

These local systems may have their own query processing and optimization engine (in particular, if they are database management systems). If such a system offers the possibility to retrieve the first n tuples or a sample of a result, we obviously should try to use this possibility instead of transferring the complete result set for a query to the multidatabase system and computing the first n tuples or the sample there.

Another aspect that we give detailed consideration to in this paper is the case that statistical data (histograms) about the data stored in a local source is available and can be accessed by the multidatabase system. In this case we develop a global query optimization taking this meta-data into account – focusing on the processing of 'first n' and 'sample' queries.

Our work is based on the object-relational multidatabase language FRAQL [SCS00] which in particular allows the dynamic addition of user-defined conflict reconciliation functions. For this language a query engine has been implemented which is able to access heterogeneous database systems by means of specific database adapters. In our current prototype environment we are using native adapters for Oracle and MySQL and access other data sources via ODBC. So, the main contribution of this paper is the application of histogram-based techniques for optimization and processing of 'first n' and 'sample' queries under the special circumstances of a multidatabase system.

The remainder of this paper is organized as follows. In the next section we briefly present related work. Section 3 gives an overview of basic techniques for limiting result cardinalities and sampling described in the related literature and discusses their suitability in multidatabases. In Section 4 we describe the usage of histograms for estimating

query parameters such as intermediate result sizes and attribute value distributions in the FRAQL system, which are essential for optimizing and processing first-n and sample-n queries. Some evaluation results for our approach are presented in Section 5. Finally, we conclude by summarizing the main insights and by pointing out future work.

2 Related Work

Statistical methods have been used in central database systems for twenty years, predominantly in the area of the query optimization and query result size estimation. Recently, strongly associated to data warehouse techniques, several works on this matter investigate how to limit the query results and how to provide approximate answers to user queries. An overview of these data reduction methods is given in [BDF+97].

[CK97,CK98] discuss an approach to restrict the result set by allowing the user to specify the desired result set size. This is accomplished by the SQL extension STOP AFTER. The intermediate results are limited by placing a stop node in the query tree. The authors propose two optimization strategies, namely a conservative and an aggressive. In addition they recommend a restart node in cases in which the original stop node did not produce the desired result size. Several commercial database management systems provide a similar technique to compute the top-n results.

Sampling is another technique for data reduction. The authors of [OR86] and [Olk93] describe different kinds of uniform random sampling techniques in a DBMS, because the integration of sampling in a database system can increase the performance of the sample computation. They discuss several techniques for uniform random sampling from base relations or the output of relational operators.

In [CMN99] and [AGPR99] the join sampling problem is pointed out as an example of the problem of commuting the sample operator with relational operators. [AGPR99] uses precomputed join samples, so-called join synopses, to provide approximate answers for join aggregation queries. These synopses are well suited for star or snowflake schemas which are usual in the data warehouse area. This approach is implemented in the AQUA system [AGP99], which works on top of any commercial DBMS and stores its precomputed statistical data in relations within the DBMS. For providing fast approximate answers for user queries, the system rewrites the query using the AQUA relations instead of the base relations and scales the aggregated query in the desired manner.

Using precomputed histograms for determining approximate answers is yet another possibility to reduce the query result size and to achieve short query response times. This technique is among others described in [PGI99]. The authors claim that it is possible to execute non-aggregate and aggregate queries using this method. Thereby the queries are rewritten using the histograms instead of the base relations.

An interactive and iterative way to provide approximate answers for aggregated queries, called online-aggregation, is described in [HHW97]. Here the user starts with a relative imprecise answer provided by a first small random sample of the data. This initial value will be improved during the processing. The user can observe online the value changes and the error bounds in order to decide when the exactness of the answer is sufficient for his needs.

3 Result Cardinality Limitation in Query Processing

As discussed in the previous section there are several approaches to limit the result size of a query in order to improve the response time of query evaluation. In the following we will focus on two approaches which are implemented as part of the multidatabase system FRAQL query processor [SCS00]: `LIMIT FIRST` and `LIMIT SAMPLE`. Both are extensions to the standard SQL `SELECT` statement:

```
SELECT <projection list>
FROM <table expression>
[ WHERE <condition> ]
[ ORDER BY <order spec> ]
LIMIT FIRST|SAMPLE <value expr> [PERCENT]
```

The parameter `<value expr>` can be any expression which represents a positive integer value including zero. Thus it can be a constant, a functional expression or a sub-query that is not correlated with the main query. If the keyword `PERCENT` is given, `<value expr>` denotes the percentage of the desired result size.

3.1 First n Tuples

With `LIMIT FIRST` at most `<value expr>` tuples are retrieved from the result set, if they exist. Please note that grouping and ordering have higher priority than cardinality reduction, so these operations are performed before any kind of limitation. Conversely, projection as well as aggregations have lower priority, i.e., the query

```
SELECT avg(balance)
FROM Accounts
ORDER BY balance
LIMIT FIRST 10 PERCENT
```

computes the average of the top 10% of the account balances.

Obviously, the cardinality limitation could be performed on top of the database engine in the application by closing the database cursor when the limit is reached. However, the performance benefits would be rather low. Thus, a special query operator is required which can be placed in the query execution plan and 'cuts' the tuple stream after the desired cardinality. Following the operator introduced in [CK97], we added an operator *stop* to our query engine, which implements the iterator model [Gra93] and passes a given number of tuples from the input stream. At physical level the stop operator has several implementations: a simple pipelined scan-stop operation for unordered limitations and a blocking sort-stop operation, that collects the top or bottom n tuples from the input stream in a sorted heap and produces the result set after processing the whole input.

In order to minimize the costs for query execution the stop operator should placed low in the operator graph. In [CK97] two placement strategies are discussed. With the conservative policy, the stop operator is inserted at a point in the query plan where no tuple is discarded that might be part of the final result. Let Op_i be an operator of

a plan $P = Op_1 Op_2 \ldots Op_{i-1} Op_i \ldots Op_r$ with Op_r as root operator and $\mathrm{card}(Op)$ the cardinality of the result produced by Op, then Op_i is *cardinality-preserving* if the following condition holds:

$$\mathrm{card}(Op_i) \geq \mathrm{card}(Op_{i-1})$$

The aggressive approach tries to place the stop operator earlier in the plan, i.e., even where it could provide a cardinality reduction. This requires estimating the stop cardinality by using database statistics as well as a restart operator which ensures that the desired number of tuples is produced even if the estimated stop cardinality was too low. In this case, the sub-query below the restart operator has to recompute the missing tuples.

The scenario which we support with our FRAQL system contains some special characteristics. At first, we operate in a multidatabase environment, i.e., parts of the query are performed by the local component databases which are often full-fledged DBMS. Thus, we want to exploit the ability of the sources to limit the result cardinality in order to reduce the communication costs and the query evaluation effort at global level. Restarting a query could be very expensive, so a safe estimation of the cardinality limit is needed. The second specialty of our scenario is a relaxed requirement regarding the exactness of the cardinality limitation. For supporting explorative data analysis tasks it is more important to get results meeting a given criterion very quickly. In addition, if a percental limit was specified, a small discrepancy is often tolerable. Thus, we embark on a strategy for placing the stop operator in the query plan, which is enforced by the following rules:

1. The main goal is to insert stop operators as deep as possible in the query execution tree according to the query's semantics and the capabilities of the component databases.
2. A *safe placement* is possible if the subsequent operators are cardinality-preserving and contain no sorting (cf. the conservative policy mentioned above). In this case the limit parameter need not be modified.
3. If the query contains a sort operator which cannot be performed by the component databases, this operator is replaced by a sort-stop operator.
4. If the remaining operators are not cardinality-preserving an *unsafe placement* can be performed, i.e. the limit parameter has to be recomputed.
5. For unsafe placement an additional stop operator has to be inserted near the root of the global plan respecting the higher priority of grouping operators.
6. For a join operation the stop operator is inserted only in one of the branches, either according to the safe placement policy, i.e., in the branch, for which the join predicate is cardinality-preserving, or – if no advantage can be taken from referential integrity constraints – in an unsafe manner by choosing the branch where it effects the largest decrease of costs.

A plan for a query containing a `LIMIT FIRST` clause is constructed as follows: after substituting global view definitions, performing the usual transformations (e.g., standardization, simplification [JK84]) and decomposing the query into sub-queries processable by the sources, the optimizer seeks to insert a stop operator according to the

rules given above at the root of the sub-query. If the global remaining operations are not cardinality-preserving, the limit parameter has to be adjusted by estimating the cardinality of the sub-query, which is computed from the selectivity of the operations and the histograms of the base relations as well as the intermediate results. Let $\mathrm{card}(P_{all})$ be the estimated result size without limitations and n the limit specified in the `LIMIT FIRST` query. So, the proper cardinality limit for the stop operator above operator Op_i is as follows:

$$L_{stop} = \frac{\mathrm{card}(Op_i) * n}{\mathrm{card}(P_{all})} \tag{1}$$

An additional stop operator is placed at the root of the global plan just to ensure that not too many tuples are produced. In case of a `LIMIT FIRST ... PERCENT` clause the unlimited result size is estimated from the histograms and the percentage needed for parameterizing the stop operator is computed.

3.2 Random Sampling

By using the notation `LIMIT SAMPLE <value expr> [PERCENT]` the system generates a simple uniform sample of size `n` of the query result. An efficient computation requires a sample operator, as described in [Olk93], being applied as low as possible in the query plan.

As we are in a multi database environment, there are several constraints, which have to be considered. Our system uses virtual integrated relations, so there are no indexes or complete statistics available. Furthermore an efficient random access is not possible. But on the other hand, the different sources can have different features, which have to be exploited. The histogram capabilities of the FRAQL system have to be taken into account for optimizing sampling queries.

Because of missing random access to the data, sequential sampling algorithms have to be utilized. There are two types of scenarios: known and unknown relation sizes. In the first case, algorithms as described in [Vit87] can be used, which have the advantage of not blocking. In the second case sampling with reservoir [Li94] is necessary. These algorithms do not need the relation size, but provide the first tuples only after the complete scan of the relation.

After the description of our environment and constraints, we now want to show which approaches can be adapted to our scenario. The crucial point is sampling of join and union operations.

Several approaches of random join sampling exists in literature. The objective of such strategies is to push down the sampling operator on one side of the operator tree since it is not possible to use sampling on both relations [CMN99]. Possible strategies are:

- *Naive sampling* includes a first complete computation of the join of R and T followed by the application of the sample operator.
- The second strategy is proposed in [Olk93] and includes the following steps. Consider the computation of a join of R and T. First sample uniform randomly one

tuple from R and join it with T and getting the result V. Select randomly one tuple from V and accept it with the probability proportional to the cardinality of V. These steps are repeated until the required sample size n is obtained.

– In [CMN99] further join sample strategies were proposed, which only require statistics or partial statistics on one relation. *Group-sample* is one strategy of these and consists of following steps for the join of the relations R and T:
  1. First produce a weighted WR-sample of relation R of the size n. The weight $\omega(t)$ for a tuple t is the number of distinct tuple with value v in join attribute $t.A$. This sample is denoted by S_1.
  2. Join S_1 with T and group the join after the tuples of S_1. The result is S_2.
  3. The last step consists of picking out one tuple from each group of S_2 using a unweighted random sample algorithm.

With the above constraints, the sampling approach according to [Olk93] cannot be applied in our environment because it requires an index as well as full statistics. However, the naive sampling algorithm or group-sample is possible. To support the latter strategy the FRAQL system supports a histogram scheme, the application of which is described in section 4.

There are different approaches to obtain samples of the union operation. These techniques require indexes and statistics and from there a materialization. So they cannot be used in our scenario.

4 Using Histograms

In relational database systems, information about cardinalities of the relations and the distributions of attribute values is essential for calculating the costs of query execution plans. Thus, modern systems maintain statistical information mostly in the form of histograms which are particularly well suited to the representation of approximations of non-parametric distributions. Using this information the query optimizer is capable of estimating selectivities of operators and cardinalities of result sets. A histogram consists of a set of buckets representing a subset of values of an attribute. Each bucket contains the number of tuples having the value associated with the bucket. In the following we denote the number of buckets as B and the number of tuples in the i-th bucket as $bucket(i)$ for $i = 1 \ldots B$.

There are various classes of histograms used in database systems nowadays. In *equi-width histograms* the size of the range of values in each bucket is the same, whereas in *equi-height histograms* the frequencies of the attribute values contained in each bucket are equal. Typically, equi-height histograms are used in database systems because of the lower worst case error [PSC84]. In [IP95] serial and end-biased histograms are proposed as optimal solution in many cases, but they are currently not very common. However, using histograms in a heterogeneous database environment entails several difficulties:

– the representation of histograms in system catalogs differs for the various available database systems,
– the availability and efficient access to histograms are crucial factors for processing global queries,

– there are data sources which do not maintain histograms or for which histograms cannot be calculated due to limited query capabilities.

Thus, for our FRAQL system we have chosen an approach for histogram maintenance where histograms of global visible relations, i.e., the integrated relations, are kept in the global layer of the multidatabase. When a relation is 'imported' from a source, i.e., a global virtual view is defined in FRAQL on this relation, the histograms for this relations are retrieved. The adapters for the individual database systems participating on the multidatabase are responsible for a uniform access to the system-specific histograms. So, each adapter provides a method for obtaining a histogram which can be implemented in one of the following ways:

– get the histogram directly from the database catalog of the source,
– trigger the computation of the histogram in the source (e.g., 'compute statistics' in Oracle),
– compute the histogram in the adapter itself,
– construct a trivial histogram representing an equipartition if neither a histogram is available nor can it be computed.

Obviously, caching histograms in the federation layer is a compromise between efficient access as well as availability and the actuality of statistics on data. However, for our intended application scenario – data analysis in heterogeneous databases – this approach seems to be practical.

In the following subsection we describe the usage of histograms for supporting result cardinality limitation techniques introduced in Section 3. In particular we discuss the calculation of estimations for intermediate result cardinalities and distributions as implemented in the FRAQL query processor.

4.1 Using Histograms for Estimating Stop Cardinalities

In Section 3 we have identified cardinality estimation as an important task for parameterizing the stop operator, if only an unsafe placement is possible. For this purpose histograms are very helpful. Based on histograms of the base relations, the distribution and cardinality of the intermediate results after applying the particular operators of the query plan can be estimated by the global query optimizer. Finally, the limit parameter for the stop operator can be derived according to equation (1). This approach is implemented in FRAQL system by means of the following three steps:

1. Traversing the operator tree top-down, all attributes are determined for which histograms are needed, i.e., attributes referenced in expressions of join or selection conditions for example.
2. For each operator node the attribute value distribution in form of the histogram, the cardinality of the result set as well as the selectivity of the operator are calculated. This is performed bottom-up for all attributes identified in Step 1.
3. The limit parameter for the stop operator is calculated using equation (1).

Fig. 1 illustrates these steps for the operator tree of the query:

```
SELECT *
FROM customers c, insurances i, accounts a
WHERE c.cust_id = i.cust_id AND
      c.cust_id = a.account_no AND
      a.balance > 6000
LIMIT FIRST 10 PERCENT;
```

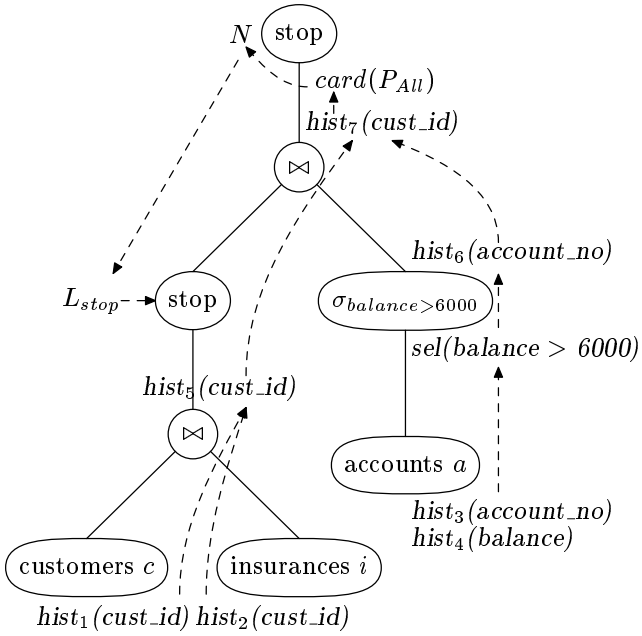


Figure 1. Histogram estimation for a `LIMIT FIRST`-query

Assuming that coincided histograms are available for the attributes `c.cust_id` and `i.cust_id` (denoted as $hist_1(cust\_id)$ and $hist_2(cust\_id)$) of the base relations the buckets of the histogram $hist_5(cust\_id)$ for the join $c \bowtie i$ can be calculated using the following formula[SS94]:

$$\forall j = 1 \ldots B : bucket_{c \bowtie i}(j) = \frac{bucket_c(j) * bucket_i(j)}{\max(d_c, d_i)}$$

Here, d_c and d_i are the numbers of distinct values present in the join column from c or i respectively. If the histograms do not coincide a preceding normalization step has to be performed.

For the selection operator on relation `accounts` the histogram can only be derived indirectly by calculating the selectivity of the operator. According to the formulas presented in [PSC84] we can estimate the selectivity sel for the expression `balance > 6000`. Let $sel_{\theta c}$ be the estimated selectivity of comparison $attr\ \theta\ c$ for

$\theta = \{<, >, \leq, \geq, =\}$. Since

$$sel_{>c} = 1 - sel_{\leq c} \quad \text{and}$$
$$sel_{\leq c} = sel_{<c} + sel_{=c}$$

we have only to estimate $sel_{<c}$ and $sel_{=c}$. In [PSC84] the following formulas are given for the case where the value c is in the k-th bucket:

$$sel_{<c} = \frac{k - 1 + 1/3}{B + 1}$$
$$sel_{=c} = \frac{1/3}{B + 1}$$

Based on these formulas the selectivity of the expression can be computed using the histogram of `a.balance`. This value is used to adjust the histogram for `a.account_no` (denoted as *hist$_6$(account_no)*) by reducing the height of each bucket assuming independence of the attributes:

$$\forall j = 1 \ldots B : bucket(j) \leftarrow bucket(j) * sel_{>6000}$$

Next, the histogram *hist$_7$(cust_id)* for $(c \bowtie i) \bowtie a$ is calculated as shown above, whereas the stop operator is ignored for the moment. The cardinality of the final result set at the root of the operator tree is equal to the sum of the heights of all buckets of this histogram:

$$\text{card} = \sum_{j=1}^{B} bucket(j)$$

Finally, the limit parameter for the stop operator is calculated. This requires firstly estimating the percentage of the cardinality of the whole relation. This value then can be inserted as parameter n in equation (1).

A special treatment is required for histograms of relations containing attributes which are transformed by applying so-called mapping functions [SCS00] as part of the view definition. Because the mapping is implemented as a special query operator in the query plan the involved histogram also has to be mapped. A straightforward solution is to apply the mapping function to each bucket boundary value.

4.2 Using Histograms for Sampling

In this section the use of histograms to support the sample operation is discussed. As shown in section 3 there is a need to apply a weighted sampling algorithm to overcome data skew and problems with join sampling. Calculating these weights requires knowledge regarding frequencies of the distinct values which can be provided by histograms.

The following example query computes a random sample of size 1,000 of the join between the relations `customers` and `insurances`.

```
SELECT *
FROM customers c, insurances i
WHERE c.cust_id = i.cust_id
LIMIT SAMPLE 1000
```

In order to improve the performance the sample operator has to be moved towards the leaves in the operator tree. One strategy, the group sample mentioned above, is introduced in [CMN99]. Figure 2 shows how we support this strategy with histograms.

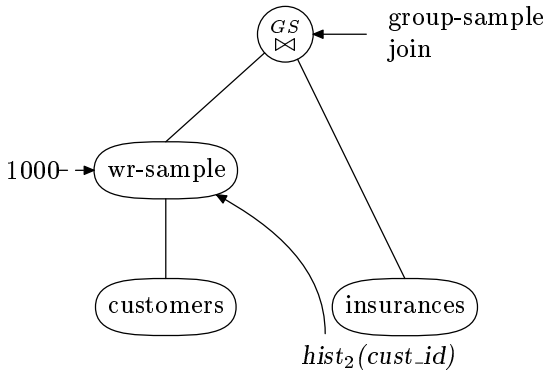


Figure 2. Histogram estimation for a `LIMIT SAMPLE`-query

We want to sample the join of `customers` and `insurances` on the attribute `cust_id`. According to the group-sample strategy we therefore need frequency information about the distinct values of `cust_id` in relation `insurances` as provided by a histogram. A weight $\omega(t)$ of a tuple t of the relation c is calculated as follows:

$$\omega(t) = card(P_{all}) * sel_{=t.cust\_id}(i).$$

$sel_{=t.cust\_id}(i)$ denotes the selectivity of the value in relation `i` and is computed by the statistical data in the histogram for the column `insurances.cust_id`. So the first step of the group-sample strategy is accomplished. The second step is performed in the group-sample-join operator, whose output is a sample of the join of the relations `customers` and `insurances`.

5 Evaluation

The main focus of the following empirical evaluation is not on the possible performance gains, because these depend strongly on the characteristics of the involved data sources. Instead, we evaluate the quality of estimations and results, which rely on statistical information contained in histograms as previously described. For this purpose the following schema is used:

```
db#1: customers (cust_id, income)
      insurances (insurance_id, cust_id)
db#2: accounts (account_id, account_no, balance)
```

Database `db#1` consists of two relations `customers` and `insurances`. It stores information about 250,000 customers, each one having one, two or no insurances with

the average of one. No special distribution describes the attribute `cust_id`, but there is a foreign key constraint from `insurances` to `customers`. The second database `db#2` to be integrated to a global view consists of a table of about 325,000 `accounts`. The attribute `account_no` matches to `customers.cust_id` and for about 57% of all customers at least one account can be found. The balances are normally distributed with a mean of 10,000 and a standard deviation of 5,000. For all involved attributes equi-height histograms consisting of 10 buckets are calculated.

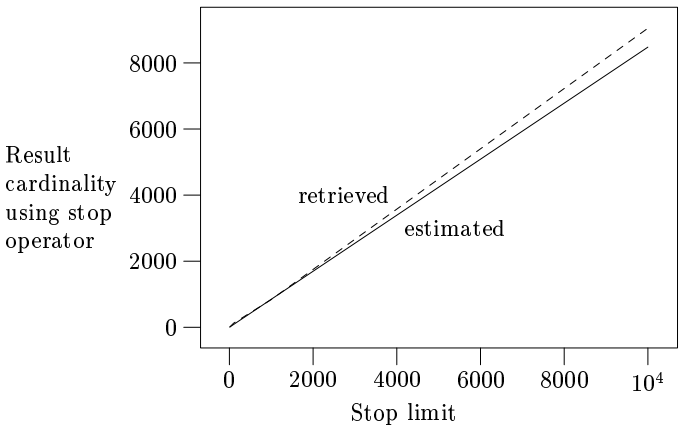
The example query executed on this constellation leads to the access plan shown in table 1. Here, x in step 7 stands for the requested cardinality in percent. Because in the average case there exists one insurance per customer, the cardinality of the index scan in step 2 is approximately the same as the number of tuples in `customers`. The

Step (i)	Operator(Op_i)	Cardinality($card(Op_i)$)
7	Select	$\approx L = card(P_{all}) \cdot \frac{x}{100}$
6	Join results from Step 4 and 5	see section 4.1
5	Table access [`accounts`]	$card$(`accounts`)
4	Stop L_{sub}	$\leq L_{sub}$
3	Join [`customers`] and [`insurances`] using index on [`insurances`]	$\approx card$(`customers`)
2	Unique index range scan [`Primary key of insurances`]	$\approx card$(`customers`) $\approx card$(`insurances`)
1	Table access [`customers`]	$card$(`customers`)

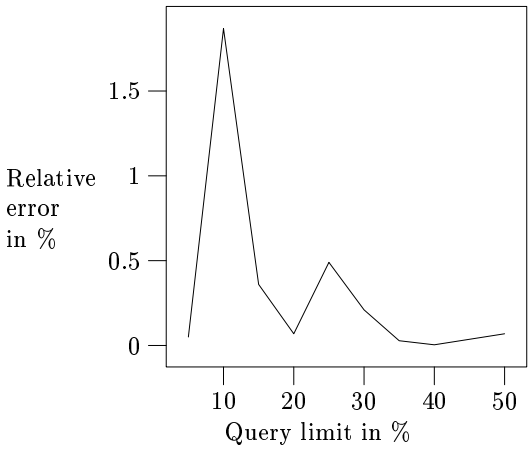
Table 1. Access plan: verification of cardinality estimation

limit L_{sub} for the subplan depends on the choice of the percentage of data sets to be retrieved. It is calculated with support of the histogram. To verify, whether this calculation step leads to a correct limitation, the desired and the actual generated result sizes for estimated L_{sub}, computed as illustrated in section 4.1, are compared in Fig. 3(a). The number of tuples of the exact calculated percentage is compared with the number of tuples retrieved using the estimation techniques in Fig. 3(b). It can be seen that the histogram supported estimation leads to a result size which is near to the requested cardinality. The difference increases with higher limit values. For the considered query, the number of retrieved tuples is underestimated in all cases, so that at least the desired cardinality is provided.

The quality of the sample operation is verified by testing, whether the existing data distributions are maintained or not. For this test a sample of 1,000 tuples of the joined relations `customers` and `insurances` is generated. The access plan is shown in table 2. The methodology of at first generating a weighted sample and applying the modified join operation on the result is described in detail in [CMN99] as stream-sample strategy. The attribute `income` is in the base relation [`customers`] approximately normally distributed with a mean of 5,000 and a standard deviation of 1,000. Using a goodness-of-fit χ^2 statistic we test the hypothesis that this distribution is maintained by the join operation on a sample of $n_1 = 100$ and $n_2 = 1000$ tuples. Let the level of significance be $\alpha = 0.05$ and the test intervals $A_1 = (-\infty; 1000], A_2 =$



(a) Comparison between estimated and retrieved size



(b) Relative estimation error

Figure 3. Evaluation results

Step (i)	Operator(Op_i)
5	Select
4	Join
3	Weighted sample with replacement of [`customers`] Weights are frequencies from Step 2
2	Histogram access (frequency of `cust_id` [`insurances`]
1	Table access [`customers`]

Table 2. Access plan: verification of maintaining of distribution

$(1000; 2000); \cdots ; A_{10} = (9000; \infty]$. Further, let $freq_j$ be the observed frequency of a value of the sample in interval j and p_j the theoretical expected count. Then, the value of the test function

$$v = \frac{1}{n} \cdot \sum_{j=1}^{10} \frac{freq_j^2}{p_j} - n \tag{2}$$

calculates to 8.69 in the case of the sample size n_1 and to 6.31 in the case of 1000 data sets to be retrieved. Because the value of the $x_{1-\alpha}$ fractile of the distribution $\chi^2(k - 1) = \chi^2(9) = 16.92$ is larger than these values, we cannot reject the hypothesis that the original distribution is maintained. Consequently, sampling constitutes an adequate way to reduce the data for analysis purposes.

References

[AGP99] S. Acharya, P.B. Gibbons, and V. Poosala. Aqua: A Fast Decision Support Systems Using Approximate Query Answers. In M.P. Atkinson, M.E. Orlowska, P. Valduriez, S.B. Zdonik, and M.L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 754–757. Morgan Kaufmann, 1999.

[AGPR99] S. Acharya, P.B. Gibbons, V. Poosala, and S. Ramaswamy. Join Synopses for Approximate Query Answering. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 275–286. ACM Press, 1999.

[BDF+97] D. Barbará, W. DuMouchel, C. Faloutsos, P.J. Haas, J.M. Hellerstein, Y.E. Ioannidis, H.V. Jagadish, T. Johnson, R.T. Ng, V. Poosala, K.A. Ross, and K.C. Sevcik. The New Jersey Data Reduction Report. *Data Engineering Bulletin*, 20(4):3–45, 1997.

[CK97] M.J. Carey and D. Kossmann. On Saying "Enough Already!" in SQL. In J. Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 219–230. ACM Press, 1997.

[CK98] M.J. Carey and D. Kossmann. Reducing the Braking Distance of an SQL Query Engine. In A. Gupta, O. Shmueli, and J. Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 158–169. Morgan Kaufmann, 1998.

[CMN99] S. Chaudhuri, R. Motwani, and V.R. Narasayya. On Random Sampling over Joins. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania,USA*, pages 263–274. ACM Press, 1999.

[GLRS93] J. Grant, W. Litwin, N. Roussopoulos, and T. Sellis. Query Languages for Relational Multidatabases. *The VLDB Journal*, 2(2):153–171, April 1993.

[Gra93] G. Graefe. Query Evaluation Techniques For Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[HHW97] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In J. M. Peckman, editor, *Proc. of the 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Arizona, USA*, volume 26 of *ACM SIGMOD Record*, pages 171–182. ACM Press, June 1997.

[IP95] Yannis E. Ioannidis and Viswanath Poosala. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 233–244. ACM Press, 1995.

[JK84] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, 1984.

[Li94] K.-H. Li. Reservoir-sampling algorithms of time complexity O(n(1 + log(N/n))). *ACM Transactions on Mathematical Software*, 20(4):481–493, December 1994.

[LSS96] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian. SchemaSQL – A Language for Interoperability in Relational Multidatabase Systems. In T.M. Vijayaraman, A.P. Buchmann, C. Mohan, and N.L. Sarda, editors, *VLDB'96, Proceedings of 22nd International Conference on Very Large Data Bases, September 3-6, 1996, Bombay, India*, pages 239–250. Morgan Kaufmann, 1996.

[Olk93] F. Olken. *Random Sampling from Databases*. PhD thesis, UC Berkeley, April 1993.

[OR86] F. Olken and D. Rotem. Simple Random Sampling from Relational Databases. In W.W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases,August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 160–169. Morgan Kaufmann, 1986.

[PGI99] V. Poosala, V. Ganti, and Y.E. Ioannidis. Approximate Query Answering using Histograms. *IEEE Data Engineering Bulletin*, 22(4):5–14, 1999.

[PSC84] Gregory Piatetsky-Shapiro and Charles Connell. Accurate Estimation of the Number of Tuples Satisfying a Condition. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 256–276. ACM Press, 1984.

[SCS00] K. Sattler, S. Conrad, and G. Saake. Adding Conflict Resolution Features to a Query Language for Database Federations. In M. Roantree, W. Hasselbring, and S. Conrad, editors, *Proc. 3nd Int. Workshop on Engineering Federated Information Systems, EFIS'00, Dublin, Ireland, June*, pages 41–52, Berlin, 2000. Akadem. Verlagsgesellschaft.

[SS94] Arun N. Swami and K. Bernhard Schiefer. On the Estimation of Join Result Sizes. In Matthias Jarke, Janis A. Bubenko Jr., and Keith G. Jeffery, editors, *Advances in Database Technology - EDBT'94. 4th International Conference on Extending Database Technology, Cambridge, United Kingdom, March 28-31, 1994, Proceedings*, volume 779 of *Lecture Notes in Computer Science*, pages 287–300. Springer, 1994.

[TGO99] Kian-Lee Tan, Cheng Hian Goh, and Beng Chin Ooi. On Getting Some Answers Quickly, and Perhaps More Later. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, pages 32–39. IEEE Computer Society, 1999.

[Vit87] J.S. Vitter. An Efficient Algorithm for Sequential Random Sampling. *ACM Transactions on Mathematical Software*, 13(1):58–67, March 1987.