Speeding up DB2 UDB Using Sampling

Peter J. Haas
IBM Almaden Research Center
peterh@almaden.ibm.com

1 Introduction

The vast amount of data in warehouses and on the web poses a major challenge for users who want to run BI or OLAP queries, execute advanced analytical, mining, and statistical algorithms, or interactively explore their data. Most of these tasks simply do not scale to the hundreds of terabytes of data often found in modern repositories. At the same time, users are demanding that decision support systems be increasingly fast, flexible, and responsive. This pressure by users arises both from the ever-increasing pace of e-business and from the development of applications that support real-time interaction with data, such as spreadsheets and OLAP tools. Although increases in CPU and disk speeds are helpful in dealing with massive data, hardware improvements alone do not suffice. Indeed, there is evidence that computer systems are getting slower in that the volume of online data is growing at a rate faster than Moore's law.

To help users address these increasingly difficult scalability problems, DB2 UDB will support random sampling in SQL queries, starting with Fixpack 2 of V8.1. Sampling techniques permit the computation of approximate query results — which often suffice in practice — in a fraction of the time required to compute an exact answer. For example, Figure 1 shows the result of executing the simple SQL aggregation query

```
SELECT SUM(sales) FROM transactions GROUP BY year
```

on the entire `transactions` table and on a 1% sample of the table. As can be seen, the results for the sampled table are almost indistinguishable from those for the entire table, while the required processing time is reduced by over two orders of magnitude.

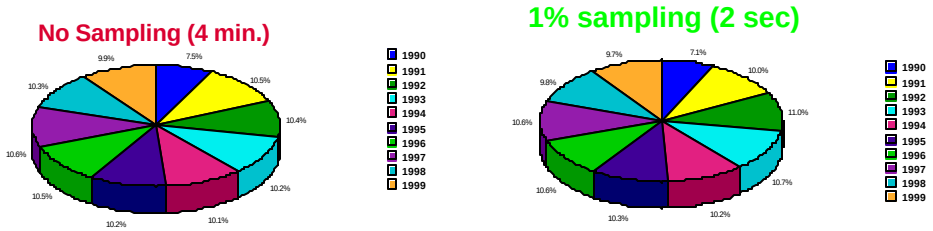

Figure 1: Effect of sampling on an aggregation query

Figure 2 shows how sampling can be combined with more sophisticated analyses. Specifically, we use the DB2 linear regression functions to fit a line to a set of data points by executing the query

```
SELECT REGR_SLOPE(t.y,t.x), REGR_INTERCEPT(t.y,t.x) FROM t
```

on a million row table and on an 0.01% sample from the table. Again, the results are almost indistinguishable, and the query against the sampled table executed orders of magnitude faster than the query against the original table. In general, sampling techniques are ideally suited to discovering general trends and patterns in data.

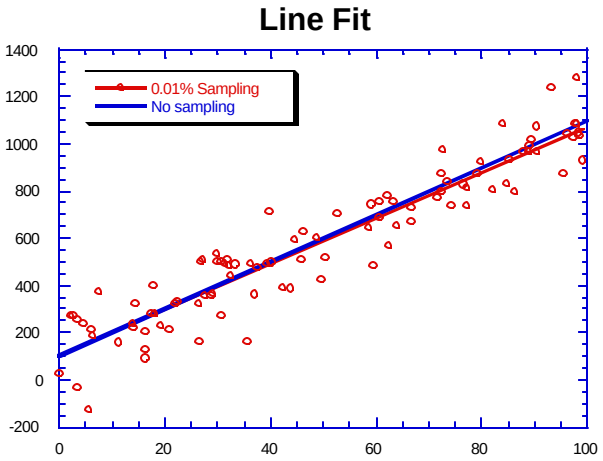

Figure 2: Effect of sampling on regression query

Perhaps the most common application of sampling is for aggregation queries, that is, queries having column functions such as AVG, SUM, and COUNT in the SELECT list. A sample can also be used for auditing purposes, interactive data exploration, or as input to a mining or analysis application. In these latter settings, a sample can be viewed as a synopsis or compressed version of a set of data.

In the following sections, we describe how sampling will work in DB2 UDB and show how the power of sampling can be greatly enhanced by combining DB2's basic sampling functionality with the expressive power of SQL.

2 Sampling Syntax and Semantics

2.1 The New SQL Sampling Clause

Any stored table appearing in the FROM clause of a SELECT query can have an appended sampling clause. The basic syntax of a sampling clause is as follows (illustrated here for a single table):

```
SELECT select_list
FROM table_name TABLESAMPLE sampling_method (P) [ REPEATABLE (S) ]
```

Here TABLESAMPLE is a new SQL keyword that tells DB2 to process only a sample of rows from *table_name* rather than all of the rows, *sampling_method* specifies the method used to sample the rows, and *P* specifies the target percentage of rows to

retrieve (between 0 and 100). The optional REPEATABLE clause, discussed below, is useful for debugging sampling queries. DB2's syntax is consistent with the ISO SQL-200n Change Proposal that has nominally been approved by ISO's Database Languages Working Group.

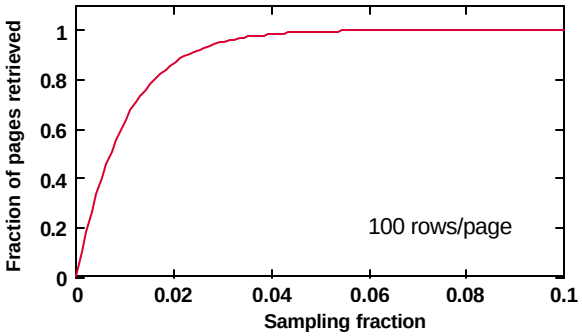

Figure 3: Minimum fraction of pages retrieved for BERNOULLI sampling as a function of fraction of rows sampled

Although the proposed ISO syntax permits a sampling clause to be associated with any table reference, DB2 will initially restrict sampling to stored tables. Thus a materialized query table (MQT/AST) or global temporary table can be sampled, but sampling will not be allowed on logical views, nicknames (in federated databases), table expressions, table functions, and so forth.

DB2 supports the two sampling methods currently specified in the ISO proposal: BERNOULLI and SYSTEM. BERNOULLI sampling (with sampling percentage *P*) "flips a weighted coin" for each row individually, including that row in the sample with probability *P*/100 and excluding it with probability 1- *P*/100, independently of the other rows. For this reason the BERNOULLI sampling method is sometimes called *row-level* Bernoulli sampling. Although on average the sample contains *P* percent of the rows in the table, the actual sample size is random, and hence may differ on subsequent executions of the query.

When there is no index available, BERNOULLI sampling retrieves each row in the table, so that there is no I/O savings and sampling performance can be poor. When there is an index on one or more of the columns on the table, then DB2 performs the Bernoulli "coin flips" on the RIDs in the index leaf pages, thereby retrieving only those pages that contain at least one sampled row. As shown in Figure 3, however, even under the most efficient implementation of BERNOULLI sampling, a large fraction of the pages must be retrieved unless the sampling fraction is extremely low.

SYSTEM sampling permits the query optimizer to determine the most efficient manner in which to perform the sampling. In most cases, SYSTEM sampling applied to a table means that each page of the table is included in the sample with probability *P*/100 and excluded with probability 1 - *P*/100. For each page that is included, all rows on that page qualify for the sample. This sampling method is called *page-level* Bernoulli sampling. SYSTEM sampling generally executes much faster than BERNOULLI sampling, since many fewer data pages need to be retrieved. SYSTEM sampling can, however, yield less accurate estimates for aggregate functions, e.g., SUM(SALES), especially if there

are many rows on each page or if the rows of the table are clustered on any of the columns referenced in the query. The optimizer may, in certain circumstances, decide that it is more efficient to perform SYSTEM sampling as if it were BERNOULLI sampling, for example when a predicate can be applied by an index and is much more selective than the sampling rate *P*.

In general, BERNOULLI sampling is most appropriate when materializing a sample of rows that is to be used for auditing purposes or as an input to a sequence of sophisticated mining or analysis procedures. In this setting, the higher cost of producing the sample can often be amortized over the multiple analyses. BERNOULLI sampling is also appropriate when the column values are clustered on pages and it is important to obtain a sample that is as representative of the set of rows in the table as possible. SYSTEM sampling is most appropriate when the goal is to produce quick approximate answers.

The optional REPEATABLE clause allows the results of a sample to be reproducible over repeated executions of the query. For a fixed value of the argument *S*, the same set of rows is sampled from the table each time the query is run. Thus *S* performs the same role as the seed in a random number generator. The REPEATABLE clause is very useful for purposes of debugging. Of course, repeatability over subsequent query executions is only guaranteed if the data is not updated, reorganized, repartitioned, etc. between executions. To guarantee that the same sample from a table is used across multiple queries, the use of a global temporary table is recommended. Alternatively, the multiple queries could be combined into a single query having multiple references to the sample, which is defined using a WITH clause.

Semantically, sampling of a table occurs before any other query processing, such as applying predicates or performing joins. One can envision the original tables referenced in a query being initially replaced by temporary "reduced" tables containing sampled rows, and then normal query processing commencing on the reduced tables. (For performance reasons, actual processing may not occur in exactly this manner.) It follows, for example, that repeated accesses of a sampled table within the same query, such as in a nested-loop join or a correlated subquery, will see the same sample of that table within a single execution of that query.

2.2 Some Illustrative Examples

The following examples illustrate the syntax and semantics of sampling.

Suppose we wish to estimate the sum of sales over a set of transactions using a 2% SYSTEM sample. Then we can use the following query:

```
SELECT SUM(sales) / 0.02
FROM transactions TABLESAMPLE SYSTEM(2);
```

Because we have computed the sum of sales over only about 2% of the rows in the table, we need to scale up our answer by a factor of 50 (i.e., 1/0.02) to estimate the sum over the entire table. (Such a scaleup would not have been needed had we used AVERAGE instead of SUM.) If we modifiy the query to look as follows:

```
SELECT SUM(sales) / 0.02
FROM transactions TABLESAMPLE SYSTEM(2) REPEATABLE(3);
```

then we will take the exact same sample from `transactions` every time that we execute the query, and hence compute the same estimate every time (assuming that `transactions` is not modified between query executions). If we change the argument of REPEATABLE from 3 to 17, then we will get the same estimate every time we run the modified query, but this latter estimate will differ from the estimate that we obtain using a REPEATABLE argument equal to 3.

Here is a slightly more complicated example, in which we estimate the sum of sales of widgets for each city:

```
SELECT SUM SALES / 0.02
FROM transactions AS t TABLESAMPLE BERNOULLI(2)
WHERE t.product = `widget`
GROUP BY t.city;
```

Note that the operations of sampling and application of predicates (including predicates implicit in a GROUP BY query) are interchangeable.

We can also compute the join of two samples:

```
SELECT s.a, t.b
FROM s TABLESAMPLE SYSTEM(1), t TABLESAMPLE SYSTEM(1)
WHERE s.key = t.fkey;
```

In general, the join of samples has very different statistical properties from the sample of a join, e.g., as computed by the following query:

```
CREATE USER TEMPORARY TABLESPACE;
DECLARE GLOBAL TEMPORARY TABLE r(a REAL, b REAL);
INSERT INTO SESSION.r
  (SELECT s.a, t.b FROM s,t WHERE s.key = t.fkey);
SELECT a, b FROM SESSION.r TABLESAMPLE SYSTEM(0.01);
```

(We use a global temporary table for this query because such tables are inexpensive to create and modify: no entries are created in the system catalog and the table persists only for the duration of the database connection.)

Sampling can be used to obtain quick estimates of simple aggregates that are computed over joins, provided that the appropriate scaleup factor is used; e.g., a SUM must be scaled up by a factor of 1 / (product of sampling rates). For example, suppose we want to quickly estimate the answer to the following query:

```
SELECT SUM(s.a * t.b)
FROM s, t
WHERE s.c > s.d;
```

To this end, we can execute the following sampling query:

```
SELECT SUM(s.a * t.b) / (0.05 * 0.01)
FROM s TABLESAMPLE SYSTEM(5), t TABLESAMPLE SYSTEM(1)
WHERE s.c > s.d;
```

The estimate computed by this query is *unbiased* for the true value in that if we execute the sample query over and over, then we obtain the correct answer on average.

In the following example, we want to select each employee whose salary exceeds the estimated average salary for his department:

```
SELECT name, salary FROM emp e
WHERE salary > (SELECT AVG(salary)
                FROM emp e1 TABLESAMPLE BERNOULLI(1.0)
                WHERE e1.deptno = e.deptno
               );
```

The semantics of DB2 sampling imply that the estimated average salary is *fixed* for each department within a single query execution, because `e1` is sampled only once. Of course, the estimated average salary for each department varies over different executions of the query, because the REPEATABLE clause is not present.

3 Combining Sampling and SQL

The scope of DB2's basic sampling tools can be greatly enhanced using the expressive power of SQL. In this section we provide some examples that show tricks for implementing non-native sampling methods, estimating the precision of quick approximate answers, drilling down into "fuzzy" datacubes, and other tasks.

We have already introduced two useful techniques. As we have seen, sampling can be combined with DB2's analytical functions, such as linear regression, to scale these functions to large datasets. We have also shown one technique for sampling from complex query expressions, namely, the use of global temporary tables. We now look at other ways of combining sampling and SQL.

3.1 Estimating the Precision of Quick Approximate Answers

Quick approximate answers to an aggregation query vary between successive query executions because (in the absence of the REPEATABLE clause) the samples of rows on which the aggregate is based vary between executions. Assessing the variability, and hence precision, of a sampling-based estimate is therefore an important task. One common statistical measure of an estimate's variability is the associated *standard error*. A rough rule of thumb is that, with high probability, the true value of an aggregate with lie within plus or minus two standard errors of the estimated value. Therefore, the smaller the standard error, the more precise and less variable the estimate. The following query

shows how to compute the standard error of a estimated SUM when using BERNOULLI sampling on a single table, using a well known formula for the standard error:

```
SELECT SUM(sales) / :samp_rate AS est_sales,
  SQRT((1e0/:samp_rate)*((1e0/:samp_rate) - 1e0)
     *SUM(sales*sales)) AS std_err
FROM transactions TABLESAMPLE BERNOULLI(100 * :samp_rate);
```

(This example assumes that the sampling rate is contained in a host variable.) The following query is similar, but for AVERAGE instead of sum:

```
WITH dt AS
 (SELECT COUNT(*) / :samp_rate as est_count,
         SUM(sales) / COUNT(*) AS est_avg,
         (1e0/:samp_rate)*((1e0/:samp_rate)-1e0)
            *SUM(sales*sales) AS v_sum,
         (1e0/:samp_rate)*((1e0/:samp_rate)-1e0)*COUNT(*)
            AS v_count,
         (1e0/:samp_rate)*((1e0/:samp_rate)-1e0)*SUM(sales)
            AS cov_cs
  FROM trans TABLESAMPLE BERNOULLI(100*:samp_rate))
SELECT est_avg,
       SQRT(v_sum-2e0*est_avg*cov_cs+est_avg*est_avg
          *v_count)/est_count AS std_err
FROM dt;
```

This query is more complicated than the previous one because an AVERAGE is a SUM divided by a COUNT, which makes the standard error computation more difficult.

The current proposed ISO standard is not quite powerful enough to permit standard error computations under SYSTEM sampling. IBM and others are currently working on enhancing the proposed standard to address this issue. Under one enhancement tentatively proposed by several vendors, the following query could be used to estimate the sum of sales, along with the standard error of the estimate:

```
WITH
  dt1 AS (SELECT sales, SAMPLE UNIT FOR trans AS s_u
          FROM trans TABLESAMPLE SYSTEM(100*:samp_rate)),
  dt2 AS (SELECT SUM(sales) AS s_sales
          FROM dt1
          GROUP BY s_u),
  dt3 AS (SELECT SUM(s_sales*s_sales) AS ss_sales,
                 SUM(s_sales) AS tot_samp_sales
          FROM dt2)
SELECT tot_samp_sales / :samp_rate AS est_sales,
       SQRT((1e0/:samp_rate)*((1e0/:samp_rate)-1e0)
         *ss_sales) AS std_err
FROM dt3;
```

The new extension to SQL is the `SAMPLE UNIT` clause. The idea is to assign to each sampled row a vendor-defined “sample-unit ID” having the following property: rows r and

s have the same sample-unit ID if and only if they are always jointly excluded from or included in the sample. So under BERNOULLI sampling each row has a unique sample-unit ID, whereas under page-level Bernoulli sampling two rows have the same sample-unit ID if and only if they lie on the same page. Using the `SAMPLE UNIT` clause, the following query estimates average sales, along with a standard error:

```
WITH
  dt1 AS (SELECT SUM(sales) AS s_sales, COUNT(*) AS c_sales
          FROM transactions TABLESAMPLE
SYSTEM(100*:samp_rate)
          GROUP BY SAMPLE UNIT FOR transactions),
  dt2(est_count, est_avg, v_sum, v_count, cov_cs) AS
    (SELECT SUM(c_sales) / :samp_rate,
            SUM(s_sales) / SUM(c_sales),
            (1e0/:samp_rate)*((1e0/:samp_rate)-1e0)
              *SUM(s_sales*s_sales),
            (1e0/:samp_rate)*((1e0/:samp_rate)-1e0)
              *SUM(c_sales*c_sales),
            (1e0/:samp_rate)*((1e0/:samp_rate)-1e0)
              *SUM(s_sales*c_sales)
     FROM dt1)
SELECT est_avg,
       SQRT(v_sum-2e0*est_avg*cov_cs+est_avg*est_avg*
         v_count)/est_count AS std_err
FROM dt2
```

The foregoing SAMPLE UNIT functionality is under consideration for a future release of DB2, when the ISO standard becomes finalized.

3.2 Implementing Non-Native Sampling Schemes

BERNOULLI and SYSTEM sampling can be combined with SQL to implement other sampling schemes. For example, the method of *simple random sampling* (without replacement) selects a specified number of rows, say *n*, randomly and uniformly from among all of the rows in a table. More precisely, the probability of selecting any two subsets of *n* rows is the same. To materialize a simple random sample of rows, we can first select a BERNOULLI sample of $m > n$ rows and then select a random subset of *n* rows. To select the random subset, we randomly sort the *m* rows and then return the first *n* of these rows. Here is a sample query for selecting a simple random sample of 75 rows from a 10,000 row table:

```
WITH dt AS (SELECT * FROM mytable TABLESAMPLE BERNOULLI(1))
SELECT * FROM dt ORDER BY RAND()
FETCH FIRST 75 ROWS ONLY
```

Here `RAND` is the built-in function that returns a pseudorandom number between 0 and 1.

Another popular sampling method is *stratified sampling*, in which the rows are divided into disjoint sets, or strata, such that the column values of interest are relatively

homogeneous within each stratum. Then a fixed number of rows are sampled from each stratum. This sampling method can lead to highly precise estimates, because observation of a few random column values from within a stratum yields a large amount of information about all of the column values within the stratum. Another reason why stratified sampling is effective is that it helps prevent any given column value from being over- or under-represented in the sample due to sheer chance. The following query stratifies rows by the `gender` attribute and collects a simple random sample of 75 rows from each stratum:

```
(SELECT name,address,salary,gender
  FROM census TABLESAMPLE BERNOULLI(10)
  WHERE gender = 'M'
  ORDER BY RAND() SELECT FIRST 75 ROWS ONLY)
UNION
(SELECT name,address,salary,gender
 FROM census TABLESAMPLE BERNOULLI(10)
 WHERE gender = 'F'
 ORDER BY RAND() SELECT FIRST 75 ROWS ONLY)
```

3.3 Drilling Down in Fuzzy Datacubes

As a final example of how sampling and SQL can work together synergistically, we provide a query that permits drilldown into a "fuzzy" datacube of sales data:

```
SELECT country, state, city, year, month,
  AVG(value) AS avg_sales,
FROM trans TABLESAMPLE SYSTEM(:samp_rate), loc
WHERE trans.locid = loc.locid
GROUP BY ROLLUP(country, state, city),
         ROLLUP(year, month)
HAVING COUNT(*) > 100
```

The idea is to sample from the fact table `trans` and drill down by increasing the sampling rate. Observe that this query returns the average sales only for those cells in the ROLLUP cube where the average is based on at least 100 sampled rows. Thus at low sampling rates, the query will only return values for highly aggregated cells such as (year), (country), or (year, country). At higher sampling rates, we will start seeing average sales for less aggregated cells such as (year, month, country, state, city) and so forth. The sampling-based approach is very natural in the sense that drilling down to greater depths requires processing of more data; in contrast, the usual (non-sampling-based) ROLAP computation processes more data for highly aggregated cells than for less aggregated cells.

4 Conclusion

Sampling will be a powerful enhancement to DB2's capabilities, potentially speeding up query processing by orders of magnitude. DB2 will support almost the entire proposed ISO standard; we expect that additional sampling methods and other enhancements will be added both to the standard and to DB2 over time. By judiciously combining DB2's

native sampling support with the SQL language, the DB2 user can effectively apply BI, OLAP, mining, and analysis techniques even in the face of massive data volumes.