Adaptive Statistics in Oracle 12c

Sunil Chakkappen[*] Suratna Budalakoti[†] Ramarajan Krishnamachari[*] Satyanarayana R Valluri[*] Alan Wood[†] Mohamed Zait[*]
*Oracle Corp., †Oracle Labs
400 Oracle Parkway, Redwood Shores, CA 94065, USA
{sunil.chakkappen, suratna.budalakoti, ramarajan.krishnamachari, satya.valluri, alan.wood, mohamed.zait}@oracle.com

ABSTRACT

Database Management Systems (DBMS) continue to be the foundation of mission critical applications, both OLTP and Analytics. They provide a safe, reliable and efficient platform to store and retrieve data. SQL is the lingua franca of the database world. A database developer writes a SQL statement to specify data sources and express the desired result and the DBMS will figure out the most efficient way to implement it. The query optimizer is the component in a DBMS responsible for finding the best execution plan for a given SQL statement based on statistics, access structures, location, and format. At the center of a query optimizer is a cost model that consumes the above information and helps the optimizer make decisions related to query transformations, join order, join methods, access paths, and data movement.

The final execution plan produced by the query optimizer depends on the quality of information used by the cost model, as well as the sophistication of the cost model. In addition to statistics about the data, the cost model also relies on statistics generated internally for intermediate results, e.g. size of the output of a join operation. This paper presents the problems caused by incorrect statistics of intermediate results, survey the existing solutions and present our solution introduced in Oracle 12c. The solution includes validating the generated statistics using table data and via the automatic creation of auxiliary statistics structures. We limit the overhead of the additional work by confining their use to cases where it matters the most, caching the computed statistics, and using table samples. The statistics management is automated. We demonstrate the benefits of our approach based on experiments using two SQL workloads, a benchmark that uses data from the Internal Movie Data Base (IMDB) and a real customer workload.

1. INTRODUCTION

Database query processing refers to the process of compiling and executing SQL statements within a Database Management System (DBMS). The process consists of the SQL Compiler taking a SQL statement text with optional bind variables as input and producing an execution plan. The execution process (performed by the SQL Execution component) takes the execution plan and returns the result of the execution. An execution plan contains the detailed steps necessary to execute the SQL statement. These steps are expressed as a set of database operators that consumes and produces rows. The processing order and implementation of the operators are decided by the query optimizer, using a combination of query transformations and physical optimization techniques.

Proceedings of the VLDB Endowment, Vol. 10, No. 12
Copyright 2017 VLDB Endowment 2150-8097/17/08.

Figure 1 illustrates the lifecycle of a SQL statement inside the *SQL compiler*. A SQL statement goes through the *Parser, Semantic Analysis* (SA), and *Type-Check* (TC) first before reaching the optimizer. The Oracle optimizer performs a combination of logical and physical optimization techniques [1] and is composed of three parts:

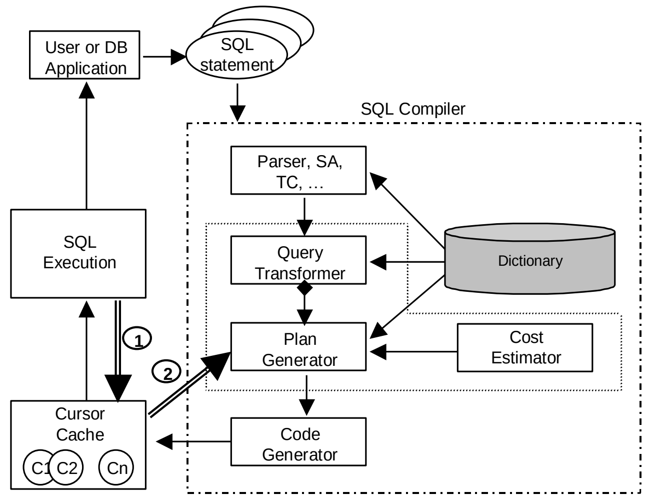


Figure 1. Architecture of SQL Engine

a) The *Query Transformer* (QT) is responsible for selecting the best combination of transformations. Subquery unnesting and view merging are some examples of Query transformations.
b) *Plan Generator* (PG) selects best access paths, join methods, and join orders. The QT calls the PG for every candidate set of transformations and retains the one that yields the lowest cost.
c) The PG calls the *Cost Estimator* (CE) for every alternative access path, join method, and join order and keeps the one that has the lowest cost.

The *Code Generator* (CG) stores the optimizer decisions into a structure called a *cursor*. All cursors are stored in a shared memory area of the database server called the *Cursor Cache* (CC). The goal of caching cursors in the *cursor cache* is to avoid compiling the same SQL statement every time it is executed by using the cached cursor for subsequent executions of the same statement. The *Dictionary* contains the database metadata (definitions of tables, indexes, views, constraints, etc) as well as object and system statistics. When processing a SQL statement, the *SQL compiler* components accesses the *Dictionary* for information about the objects referenced in the statement, e.g. the optimizer reads the statistics about a column referenced in the WHERE clause. At run-time, the cursor corresponding to a SQL statement is identified based on several criteria, such as the SQL text, the compilation environment, and authentication rules. If a matching cursor is found then it is used to execute the statement, otherwise the SQL compiler builds a new one. Several cursors may exist for the same SQL text, e.g. if the same SQL text is submitted by two users that have different authentication rules. All the factors that affect the execution plan, such as whether a certain optimization is enabled by the user running the SQL statement, are used in the algorithm used to match a cursor from the CC.

The execution plan generated for the SQL statement is just one of the many alternative execution plans considered by the query optimizer. The query optimizer selects the execution plan with the lowest cost. Cost is a proxy for performance; the lower the cost, the better the performance (e.g. response time) of the query is expected to be. The cost model used by the query optimizer considers the IO, CPU and network utilization of executing the query. The cost model relies on object statistics (e.g. number of rows, number of blocks, and distribution of column values) and system statistics (e.g. IO bandwidth of the storage subsystem). The quality of the final execution plan produced by the query optimizer depends on the quality of the information used by the cost model and the cost model itself. In the following sections we focus on one important input to the cost model, cardinality.

1.1 Effect of Cardinality on Plan Generation

Plan Generator (PG) is responsible for evaluating various access paths, join methods, and join orders and choosing the plan with the lowest cost. This section describes the PG module and the important role cardinality estimate plays in picking the most efficient plan. Consider the following query, Q1 that retrieves the amount of all ‘Y Box Games’ products under the ‘Electronics’ category sold in California.

```
Q1:

SELECT prod_name, sum(amount_sold) amount_sold
FROM products p, customers c, sales s
WHERE p.prod_category = 'Electronics'
  AND p.prod_subcategory = 'Y Box Games'
  AND p.prod_id = s.prod_id
  AND c.cust_state_province = 'CA'
  AND c.cust_id = s.cust_id
GROUP BY prod_name;
```

1.1.1 Access Path Selection

PG considers different access path for the tables in a query, e.g. some of the access paths considered for *products* table are

- Full Table Scan – Reads all rows in the table and produces rows that qualify for the specified filter condition.
- Index scan – It is used to limit access to rows in the table that qualify for the condition on the index key columns. Filters on non-index columns can be used to further filter rows once they are accessed. In the above example, PG evaluates using an index scan on an index defined with key `prod_category` or key `prod_subcategory`.

The access path with the least cost is selected. The cost is greatly dependent on the cardinality produced after applying the predicate(s). For example, if the number of rows produced with predicate `p.prod_subcategory = ‘Y Box Games’` is very small compared to the total number of rows, using Index Scan on `prod_subcategory` is more efficient that using Full Table Scan.

1.1.2 Join order Selection

PG explores different join orders and chooses the join order with the least cost. For query Q1, the following join orders are possible - (C->P->S), (C->S->P), (S->C->P), (S->P->C), (P->S->C), (P->C->S), where the letters correspond to the aliases of the tables used in the query. The number of rows of each of the tables and intermediate joins is an important input in computing the cost of the join orders. If C and P are joined first (a Cartesian product), the intermediate size, and the resulting cost, will be high, compared to joining P and S first.

Note that given a join between *N* tables, there are at most *N!* possible join permutations. Large values of *N* can cause an exhaustive optimization to be prohibitive. PG employs several heuristics to cut down the optimization time.

1.1.3 Join Method Selection

PG also selects the most efficient join method for every join order based on the cost of feasible join methods. For example, tables P and S can be joined using Nested Loop Join or Hash Join. Their cost depends on the cardinalities of both inputs to the join. Typically, Nested Loop join is the cheapest option if the left input produces a low number of rows.

1.1.4 Query Transformations

The *Query Transformer* (QT) module transforms SQL statements into a semantically equivalent form if the newly transformed form is cheaper than the original form. For example, the query Q1 can be transformed into query Q2 as follows.

```
Q2:

SELECT vw_gbc_3.item_3 prod_name,
       sum(vw_gbc_3.item_2) amount_sold
FROM sh.customers c,
  (SELECT s.cust_id item_1,
          sum(s.amount_sold) item_2,
          p.prod_name item_3
   FROM sh.sales s, sh.products p
   WHERE p.prod_id = s.prod_id
     AND p.prod_subcategory = 'Y Box Games'
     AND p.prod_category = 'Electronics'
   GROUP BY s.cust_id, p.prod_name) vw_gbc_3
WHERE c.cust_state_province='CA'
  AND c.cust_id=vw_gbc_3.item_1
GROUP BY vw_gbc_3.item_3;
```

In this transformation, the `group-by` is placed before the join to the Customers table. In general, the transformed plan will be more efficient than the original plan if performing a group-by earlier reduces the number of rows that joins with the Customers table. The query transformation layer calls PG to get the cost for both forms of the statement and chooses the one with the least cost [1]. The group-by cardinality is a major factor in determining the cost of these plans. Therefore, the quality of the cardinality estimate is important for selecting the optimal transformations.

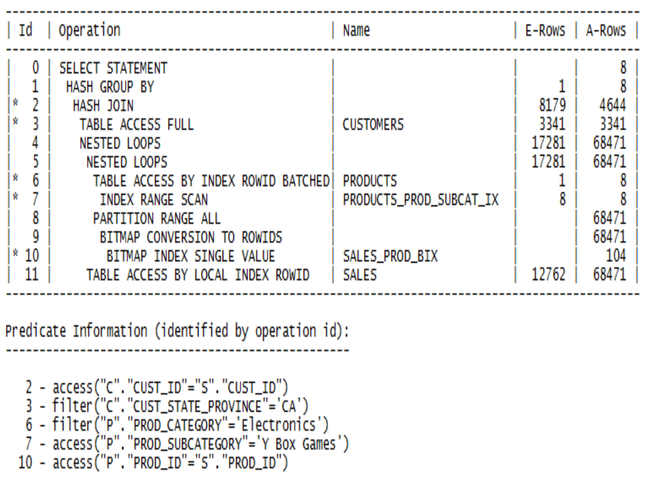
Besides the above major four decisions, the plan generator also makes decisions that depend on the cardinality of the intermediate results. For example, if the SQL statements executes in parallel, the query optimizer decides how the data is reshuffled between processes that perform adjacent steps of the execution plan. Incorrect cardinality estimates can lead to selecting the wrong reshuffling method, which in turn negatively affect the performance of the SQL statement.

It is evident from the above discussion, that improving the accuracy of the cardinality estimation will greatly improve the ability of the plan generator to select the most efficient plan.

1.2 Cardinality Estimation errors

Estimating the number of rows is one of the thorniest subjects in query optimization. It is the Achilles heel of every query optimizer. The formula used to estimate cardinality based on predicates easily breaks when the predicates involve skewed columns, expressions on columns, or complex predicates connected using AND/OR operators. Over time, sophisticated statistics have been added to account for skew (histograms) and correlation (extended statistics [13]). However, pre-computed statistics have limitations that cannot be ignored. For example, in the Oracle database, extended statistics are limited to equality predicates. Furthermore, there will always be query expressions that cannot be represented as first class statistics and that will not be available during the optimization of the SQL statement.

Consider the example query Q1 mentioned in section 1.1. Figure 2 shows the execution plan with estimated and actual cardinality for query Q1.



```
----------------------------------------------------------------------------------------
| Id  | Operation                               | Name                    | E-Rows | A-Rows |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                        |                         |        |      8 |
|   1 |  HASH GROUP BY                          |                         |      1 |      8 |
|*  2 |   HASH JOIN                             |                         |   8179 |   4644 |
|*  3 |    TABLE ACCESS FULL                    | CUSTOMERS               |   3341 |   3341 |
|   4 |    NESTED LOOPS                         |                         |  17281 |  68471 |
|   5 |     NESTED LOOPS                        |                         |  17281 |  68471 |
|*  6 |      TABLE ACCESS BY INDEX ROWID BATCHED| PRODUCTS                |      1 |      8 |
|*  7 |       INDEX RANGE SCAN                  | PRODUCTS_PROD_SUBCAT_IX |      8 |      8 |
|   8 |      PARTITION RANGE ALL                |                         |        |  68471 |
|   9 |       BITMAP CONVERSION TO ROWIDS       |                         |        |  68471 |
|* 10 |        BITMAP INDEX SINGLE VALUE        | SALES_PROD_BIX          |        |    104 |
|  11 |     TABLE ACCESS BY LOCAL INDEX ROWID   | SALES                   |  12762 |  68471 |
----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("C"."CUST_ID"="S"."CUST_ID")
   3 - filter("C"."CUST_STATE_PROVINCE"='CA')
   6 - filter("P"."PROD_CATEGORY"='Electronics')
   7 - access("P"."PROD_SUBCATEGORY"='Y Box Games')
  10 - access("P"."PROD_ID"="S"."PROD_ID")
```

Figure 2. Execution Plan for Q1

The cardinality estimate for access to the *products* table (line 6) is under estimated by a factor of 8. The cardinality for this operation takes into consideration the predicates applied at line 6 and 7, i.e. *prod_subcategory = 'Y Box Games' AND prod_category = 'Electronics'*. The underestimation is due to the strong correlation between the columns involved in these two predicates. *Cost estimator* estimates the cardinality based on statistics available on these columns individually as if they are independent which leads to incorrect estimates. The mis-estimate at line 6 cascades to operations higher up in the plan, e.g. 5, 4. Such mis-estimates can cause the *Plan Generator* to pick a suboptimal plan.

1.3 Contributions

In this paper, we discuss our approach towards improving the quality of statistics used during query optimization. It includes automatic creation of auxiliary statistics structures (called extended statistics) based on workload analysis and validation of optimizer statistics (including that of intermediate results) using actual table data. We mitigate the cost of accessing table data using the following techniques:

- **Adaptive sampling.** When accessing table data to validate optimizer statistics we use sampling. We may read several samples in case earlier samples fail a quality metric. Furthermore, the statistics derived from the samples are cached for later reuse and are automatically refreshed when table's data change.
- **SQL Plan Directives (SPD)**. They are used to limit reading table data for the purpose of validating optimizer statistics, to cases where it matters the most. The latter is implemented by comparing the statistics estimated by the optimizer to the actual values seen during query execution. If there is a significant difference between estimates and actual statistics, then we create an SPD. The optimizer relies on SPDs to decide whether to validate statistics by accessing table data. In addition, SPDs are the basis for identifying extended statistics (e.g. column group statistics) as part of a separate statistics gathering process. Extended statistics will reduce the reliance on reading data for validating optimizer statistics. SPDs are generic database objects that are designed to store other information that can improve the quality of execution plans.

  SPDs are created while executing statements and subsequent queries use them. There can be cases where only a partial set of directives are available for a query, especially in the ramp up stage of an application. For example, a subset of join orders will have directives and optimizer will have the correct estimate for only those join orders. This may create a bias on costs for some join orders and can lead to a suboptimal plan. To avoid this, SQL Plan Management techniques mentioned in [23] can be used.

The organization of the rest of the paper is as follows. Section 2 presents the related work and contributions. Section 3 discusses the techniques proposed in this paper. Experimental results are presented section 4 and finally section 5 concludes the paper.

2. RELATED WORK

Several solutions have been proposed to improve the quality of cardinality estimation, in both the academic world and commercial products. The solutions can be classified into four categories.

1. *Provide better statistics to the optimizer*. For example [18] talks about maintaining histograms using feedback from previously executed SQL statements that gives better cardinality estimates. [5] proposes creating an approximate "synopsis" of data-value distributions based on the feedback obtained from observations on the executed query workload. It combines the technique of histograms with parametric curve fitting leading to a specific class of linear splines.
2. *New type of statistics*. Statistics on individual columns is not sufficient when dealing with the complexity of expressions allowed by the SQL language. The following are some of the new type of statistics used by commercial databases and referenced in the literature.

a. Oracle supports collecting statistics on group of columns to deal with correlation between columns [13]. It also allows finding the group of columns in a given workload [14]. Other database vendors [11][12] also support collecting statistics on a group of columns.
b. Filtered statistics [12] use a filter predicate to select the subset of data that is included in the statistics. It can improve query performance for queries that select from well-defined subsets of data. An experienced database administrator knowledgeable about the workload has to create the statistics that are relevant for the workload.
c. [4] Proposes collecting statistics on views and some commercial systems support collecting statistics on views [10]. Typically, the process of collecting statistics on views is manual in these implementations. However, [3] discusses a statistics advisor that can recommend views for which statistics need to be collected for a given workload.
d. [8] argues for the construction of specialized histograms, where the buckets are constructed to bound the q-error [8] , an error measure (instead of the variance, as is often the case). A rigorous relationship can be established between q-error and the cost of the final plan.

Detecting new statistics that are needed for a workload is a difficult task in general. This poses a serious manageability challenge, especially when the new statistics have to be continuously detected as new SQL statements are added to the workload or new applications are installed in the system. Once the new statistics are identified, gathering and maintaining the statistics poses another challenge. This paper proposes ways to gather and maintain the statistics automatically and continuously for query constructs (including joins) that are necessary for getting good cardinality estimates for the queries. This does not require any input from database administrators.

3. *SQL tuning*. Tuning can be done in several domains: query optimization, to improve the execution plans selected by query optimizer; access design, to identify useful access structures; SQL design, to restructure and simplify the text of a badly written statement. Oracle's Automatic SQL Tuning feature [2] helps to automate the above tuning activities. Some of the techniques proposed in [2] for verifying the cardinality estimate using sampling queries, are used in this paper as well. However, SQL tuning is designed in such a way that it runs offline and is not part of running the workload. Hence reducing the time taken to get accurate cardinality estimates is not one of the primary goals. In contrast, the ideas proposed in this paper are meant to run while optimizing the SQL statement and make sure that it runs in a reasonable amount of time using sampling, time budget, SQL Plan directives, extended statistics. etc.

4. *Feedback systems*. Oracle 11 Release 2 introduced the use of a feedback mechanism [6] for cases where the optimizer cardinality estimates are incorrect. This technique corrects the cardinality estimates for subsequent executions of the same SQL statement using the actual cardinality observed during prior executions of the statement. The actual cardinality is stored in the shared cursor and is used only when the same statement is executed. In addition, the information stored in the cursor is not persistent and hence is lost when restarting the DBMS. The techniques proposed in this paper allow information learned in previous executions to persist and can be used by other statements that share similar SQL constructs.

   Feedback mechanisms were also proposed earlier in LEO-DB2's learning optimizer [19]. This paper proposes computing adjustments based on the deviation between the estimated and actual cardinalities and storing them in dictionary tables. These adjustments are readjusted when new statistics are collected and can be incorrect. [22] proposes a sampling based re-optimization method in which after the query optimizer returns its best estimate plan, an additional step is invoked in which the plan is re-optimized by feeding the optimizer with refined cardinality estimates obtained via sampling. If the re-optimized plan is different from the optimizer plan then the original plan is considered to be based on erroneous cardinality estimates and the re-optimized plan is used for execution. This process of re-optimization is repeated iteratively until the new plan is same as that of the previous iteration.

   The techniques we propose in this paper do not store the adjustments. Instead, we store logical findings like "misestimate has occurred for a SQL construct". Actual adjustments are computed during the compilation of the statement. This accounts for any DML, statistics collection etc., that happened since the misestimate has occurred.

3. ADAPTIVE STATISTICS

3.1 Architectural Overview

Adaptive statistics solve the cardinality misestimate issues that manifest due to the limitations of pre-computed statistics. This technique consists of computing the statistics (selectivity/cardinality, even first class statistics like number of distinct values) during optimization of the SQL statement. This process happens in the *Cost Estimator* module. The statistics are computed by executing a SQL statement against the table with relevant predicates. This technique can be used to estimate cardinality of operations that involve only single table as well as more complex operations that involve join, group by etc. These kinds of queries are referred to as **statistics queries.** Statistics queries are executed in most stages of plan generation. Some example statistics queries executed while optimizing the query Q1 in section 1.1, are:

1. Query Q3 below estimates the cardinality when costing full table scan of products. It provides the cardinality after applying both predicates on this table.

   ```
   Q3:

   SELECT sum(c1)
   FROM
   (SELECT 1 AS c1
    FROM products p
    WHERE (p.prod_subcategory = 'Y Box Games')
      AND (p.prod_category = 'Electronics'));
   ```

2. Query Q4 estimates the cardinality when costing the access using the index on `PROD_SUBCATEOGORY` column.

```
Q4:

SELECT c1
FROM
(SELECT /*+ index(p products_prod_cat_ix) */
   count(*) as c1
 FROM products p
 WHERE (p.prod_category = 'Electronics'));
```

3. Query Q5 estimates the cardinality of the join between sales and products.

```
Q5:

SELECT
/*+opt_estimate(@innerquery,table,p#2,rows=8)
*/
sum(c1)
FROM
(SELECT /*+ qb_name(innerQuery) */ 1 as c1
 FROM sales SAMPLE BLOCK(47,8) SEED(1) s#0,
      products p#2
 WHERE (p#2.prod_subcategory = 'Y Box Games')
   AND (p#2.prod_category = 'Electronics')
   AND (p#2.prod_id = s#0.prod_id)) innerQuery
```

The index, opt_estimate hints and sample clause in these queries are described in later sections. Figure 3 shows the execution plan generated by the optimizer using statistics queries. Note that cardinality estimates for all the operations are accurate. Also, the plan is different from Figure 2. The new plan uses a Hash Join to join Products and Sales since the cardinality is estimated correctly for scan of Products table (operation id 5).

```
-------------------------------------------------------------------------------------------
| Id  | Operation                            | Name                   | E-Rows  | A-Rows |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                     |                        |         |      8 |
|   1 |  HASH GROUP BY                       |                        |       8 |      8 |
|*  2 |   HASH JOIN                          |                        |    4440 |   4644 |
|*  3 |    TABLE ACCESS FULL                 | CUSTOMERS              |    2987 |   3341 |
|*  4 |    HASH JOIN                         |                        |   68455 |  68471 |
|*  5 |     TABLE ACCESS BY INDEX ROWID BATCHED| PRODUCTS             |       8 |      8 |
|*  6 |      INDEX RANGE SCAN                | PRODUCTS_PROD_SUBCAT_IX |      8 |      8 |
|   7 |     PARTITION RANGE ALL              |                        |    918K|    918K|
|   8 |      TABLE ACCESS FULL               | SALES                  |    918K|    918K|
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("C"."CUST_ID"="S"."CUST_ID")
   3 - filter("C"."CUST_STATE_PROVINCE"='CA')
   4 - access("P"."PROD_ID"="S"."PROD_ID")
   5 - filter("P"."PROD_CATEGORY"='Electronics')
   6 - access("P"."PROD_SUBCATEGORY"='Y Box Games')

Note
-----
   - dynamic statistics used: dynamic sampling (level=AUTO)
```

Figure 3. Plan for Q1 using statistics queries

Executing statistics queries as part of optimizing user SQL statements incur additional optimization time. Oracle employs several techniques to reduce this overhead. We describe two of these, adaptive sampling, and SQL plan directives (SPDs), below.

Adaptive Sampling: Use sample of the table in statistics queries to estimate the cardinality. Sampling is done by *Statistics Query Engine* as shown in **Figure 4**. This module is responsible for computing the optimal sample size, executing the statistics queries within a specified time budget, using the full or partial results from statistics queries to derive the cardinality estimate, and storing the result of statistics queries in SPDs**.**

SQL Plan Directives: SPDs are persistent objects that have run time information of SQL or SQL constructs. They are used for the following purposes.

- For tracking the SQL constructs that caused misestimates: This happens in the *Execution Engine* when the cardinality estimate for a particular construct in an operation is significantly different from the actual rows produced by the operation. *Cost Estimator* requests estimates from the *Statistics Query Engine* only for the constructs for which misestimates are recorded as SPDs. This is to avoid executing statistics queries for each and every construct. To avoid the overhead of tracking in the *Execution Engine*, the directives are first recorded in *Directive Cache* in memory (*SGA*) before it is flushed to disk by background process (*MMON).*
- *Statistics collector* (*DBMS_STATS*) also looks at the SPDs for constructs with a misestimate in the *Dictionary* and gathers statistics for them. For example, if the SQL construct has multiple equality predicates, statistics collector will collect statistics for the group of columns in the predicates. This allows the statistics collector to collect statistics only for group of columns that caused the misestimate.

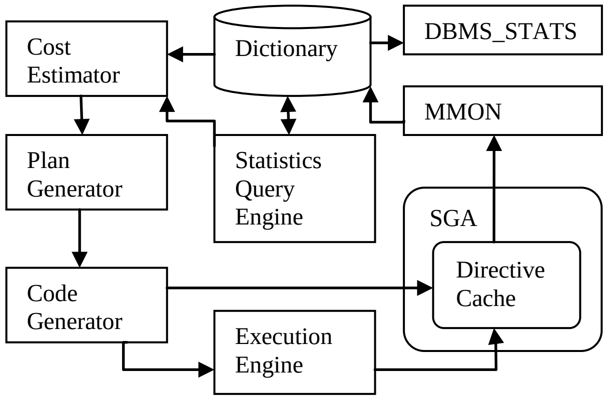


Figure 4. Adaptive Statistics Flow

- For persistently storing the result of statistics queries to avoid repeated execution of the same statistics queries: *Statistics Query Engine* first checks if there is a SPD that has the result of the statistics query in *Dictionary* and uses it if the result is still valid. If the result is stale, it executes the statistics query to get the correct result and stores the new result in directive.

The sections below discuss these techniques in detail.

3.2 Adaptive Sampling Methodology

We rely on sampling to limit the overhead when reading data from tables to validate optimizer statistics. If an access structure (e.g. index) is efficient then we skip sampling. For the latter case, the index is forced using a hint as in example Q4. This section describes the algorithms used to compute an appropriate sample size, and extrapolating the statistics value to the full data .

Formally, the adaptive sampling addresses the following problem: given a table T and a set of operators applied to T, provide an estimate of the cardinality of the resulting dataset, based on a sample. The operators applied to T include table filters, joins, group by etc. The adaptive sampling algorithm consists of iterating through the following four steps, until the quality test at step 3 is successful:

Given n, number of blocks in the initial sample, and a query Q:

1. *Sample:* Randomly sample n *blocks* from T. Apply the operators on the sample.

2. *Cardinality Estimate:* Estimate the cardinality of query Q for the entire dataset, based on the resulting cardinality after applying the operators on this sample and samples from previous iterations (if any).

3. *Quality Test:* Calculate a confidence interval around the cardinality estimate, and perform a quality test on the confidence interval.

4. *Next Sample Size Estimate:* If the quality test succeeds, stop. If the test fails, calculate n_{next}, the number of additional blocks required to be sampled, so that the resultant sample size meets the quality test (with a certain probability). Set $n = n_{next}$. Go to step 1.

As mentioned earlier, we sample a random set of blocks from T, as opposed to a random set of rows. This means that internal correlation within rows in a block have to be taken into account during the variance calculation, possibly resulting in larger required sample sizes. However, block sampling is far cheaper than row sampling, which makes this a reasonable trade-off.

Sampling at the block level introduces another complication: it is expensive to remember for each row which block it originated from, making a straightforward estimate of the block-level variance impossible. To address this problem, we rely on two statistical properties:

1. *Central Limit Theorem* [21]. The mean of a sequence of independently and identically distributed (iid) random variables follows a Normal distribution. This sample mean is an unbiased estimate of the distribution mean. The variance of the mean is σ^2/n, where σ^2 is the distribution's variance, and n is the sample size.
2. The sum of square of K independent standard Normal random variables follows a chi-squared distribution with K degrees of freedom [21].

The basic approach, then, is, to

- Take multiple block samples of sufficient size, so that each can be modeled as a sample from a Normal distribution, and
- Model the variance across samples as a chi-squared distribution, to establish confidence intervals on the variance, and derive the bounds on across-block variance from the bounds on across-sample variance.

We present the details of the approach in the next sub-section.

3.2.1 Mathematical Details

3.2.1.1 Problem Formulation

Successful execution of the adaptive sampling algorithm requires the solution of the following three problems:

a) **Cardinality Estimate and Confidence Interval:** Arrive at an unbiased estimate $\widehat{M}$ of the true cardinality M of query Q. Establish a 95% lower bound $\widehat{M}_L$ and a 95% upper bound $\widehat{M}_U$ $\widehat{M}_L$ on M such that, $\widehat{M}_L \le \hat{M} \le \widehat{M}_U$ with 95% probability.

b) **Quality Test:** For a pre-determined λ, check if $\widehat{M}_U \le (1+\lambda)\widehat{M}$. For example, if λ=1, we can be 95% confident that $\widehat{M}_U \le 2\widehat{M}$. That is, with 95% confidence, the true value of M is not more than twice the estimated value of M.

c) **Next Sample Size Estimation:** Given the current cardinality estimate, and information about the samples taken till date, estimate n_k, the size of the next sample to take, so that the condition $\widehat{M}_U \le (1+\lambda)\widehat{M}$ is likely to be met (with a certain confidence).

The following three sections address each of these problems respectively.

3.2.1.2 Solution Outline

Let μ be the ground truth mean number of rows matching the query per block (referred to as the *per block query cardinality*). Assuming that the number of blocks B constituting the table is known, it is sufficient to estimate μ, as M = μ * B.

After K rounds of adaptive sampling, let the total number of blocks sampled so far be N, and let the number of rows matching Q in the i^{th} block be x_i. Then μ is estimated as:

$$\widehat{\mu} = \frac{\sum_{i=1}^{N} x_i}{N} \qquad (1)$$

The confidence interval around $\widehat{\mu}$ can be calculated using the well-known Central Limit Theorem [9] (ch. 4), which states that, for a simple random sample $x_1,x_2,..,x_N$ from a population with mean μ and finite variance σ^2, the sample mean (calculated as equation (1) above is an unbiased estimator of the population mean μ, and is normally distributed as:

$$\hat{\mu} = N(\mu, \frac{\sigma^2}{N}) \qquad (2)$$

Using properties of the Normal distribution [9] (ch. 4), after N samples, the 100(1-α)% upper confidence bound on μ is given by:

$$\mu_{UB} = \hat{\mu} + z_\alpha * \frac{\sigma}{\sqrt{N}}$$

Here z_α is the 100(1-α)% percentile standard score (or z-score) of the standard normal distribution [9], ch. 5]. We use α=0.025, so that z_α = 1.96. To establish this confidence interval, we need to estimate σ, the query cardinality standard deviation across blocks.

A straightforward estimate of the population variance of per block query cardinality is given by the sample variance of the sampled blocks. However, since maintaining per block information is too expensive, we use an alternative approach, described next, to estimate σ.

3.2.1.3 Variance Estimation of Per Block Query Cardinality

An alternate way to calculate $\widehat{\mu}$ is in terms of the number of matching rows observed per round of sampling. Let the number of rounds of sampling completed be K, K ≥2. Let n_i be the number of blocks sampled in the ith round of adaptive sampling, and let s_i be the number of rows matching Q found in the sample taken in the ith round. Then:

$$\hat{\mu} = \frac{\sum_{i=1}^{K} s_i}{\sum_{i=1}^{K} n_i} \tag{3}$$

An unbiased estimate $\hat{M}$ of M is then given by $\hat{M} = \hat{\mu} * B$. While we do not have access to the across-block variance σ, we can compare how the estimate of the same mean changes from round to round. We use these values to arrive at an estimate of σ. Let x_i be the observed per block query cardinality for the ith sample, defined as $x_i = \frac{s_i}{n}$. By the Central Limit theorem, x_i can be modeled as being sampled from a normal random variable, $X_i = N(\mu, \sigma_i^2)$ where $\sigma_i = \frac{\sigma}{\sqrt{n_i}}$. In other words, $\frac{X_i - \mu}{\sigma_i}$ follows a standard normal distribution. As the sum of square of K standard normal random variables follows a Chi-squared distribution with K degrees of freedom (χ_K), the following holds true after K rounds:

$$\sum_{i=1}^{K} \left(\frac{X_i - \mu}{\sigma_i} \right)^2 \sim \chi_K$$

$$\Rightarrow \sum_{i=1}^{K} n_i \left(\frac{X_i - \mu}{\sigma} \right)^2 \sim \chi_K$$

$$\Rightarrow \frac{1}{\sigma^2} \sum_{i=1}^{K} n_i (X_i - \mu)^2 \sim \chi_K$$

After K rounds, the β = 97.5% upper bound on σ^2, written as $\hat{\sigma}_{UB}^2$, can be calculated as:

$$\Rightarrow \frac{1}{\hat{\sigma}_{UB}^2} \sum_{i=1}^{K} n_i (X_i - \mu)^2 = \chi_{K,\beta}$$

$$\Rightarrow \hat{\sigma}_{UB}^2 = \frac{\sum_{i=1}^{K} n_i (X_i - \mu)^2}{\chi_{K,\beta}} \tag{4}$$

We know that with 95% probability, σ^2 is less than $\hat{\sigma}_{UB}^2$. Here $\chi_{K,0.95}$ (since β=0.95) is the value v such that, $P(\chi_{K,0.95} \geq v)$. That is, $\chi_{K,0.95}$ is the 0.95 p-value, or the 0.05 inverse CDF. For example, based on the chi-squared table at[16], for K=2, $\chi_{K,0.95} = 0.103$.

The 92.5% upper bound on the per block cardinality is then given by the formula:

$$\hat{\mu}_{UB} = \hat{\mu} + 1.96 * \hat{\sigma}_{UB} \tag{5}$$

Similarly, the lower bound $\hat{\mu}_{LB} = \hat{\mu} - 1.96 * \hat{\sigma}_{UB}$. $\hat{\mu}$ can be calculated using either eq. (1).

The reason we arrive at a 92.5% upper bound on the cardinality estimate, is due to the probabilistic approximation we do at two stages: while estimating the standard deviation, and while estimating the mean. Combining the two probabilistic estimates using the *union bound* [17], we get:

$$P(\mu > \hat{\mu}_{UB} \vee \sigma > \hat{\sigma}_{UB})$$
$$< P(\mu > \hat{\mu}_{UB}) + P(\sigma > \hat{\sigma}_{UB}) = 0.075$$

Therefore, since the overall probability of error is less than 7.5%, the result has at least a 92.5% confidence. Similary, it can be shown that:

$$P(\mu > \hat{\mu}_{UB} \vee \mu < \hat{\mu}_{LB} \vee \sigma > \hat{\sigma}_{UB}) < 0.1$$

In other words, setting α=0.025, β=0.95 gives us a 90% confidence interval on the cardinality estimate.

While the above approach requires at least two rounds of sampling before arriving at a confidence interval, it has the following advantage: it can calculate an accurate confidence interval from a block sample, without requiring any block-level information. This is very useful, as storing block level information per row is expensive computationally and in terms of memory usage.

Given the upper and lower bound estimates on μ, it is straightforward to calculate M, and perform the quality test described in Section 3.2.1.1. The next section addresses how the next sample size is calculated, if the quality test fails.

3.2.1.4 Next Sample Size Calculation

Let the number of rounds of sampling completed be K-1, with a total of N_{K-1} blocks sampled. At the end of the Kth round of sampling, we would like the following condition to hold, so that no more rounds are required:

$$\hat{\mu} + z_\alpha \frac{\sigma}{\sqrt{N}} \leq (1 + \lambda)\hat{\mu}$$

Note that our default value for α=0.05. Writing z_α for brevity, the above equation can be rewritten as:

$$\sigma^2 \leq \frac{\lambda^2 \hat{\mu}^2 N}{z^2} \tag{6}$$

Using the transitivity of inequality, and using eq. (4) for the left-hand side, we see that (6) will be true if:

$$\frac{\sum_{i=1}^{K} n_i (X_i - \mu)^2}{\chi_{K,\beta}} \leq \frac{\lambda^2 \hat{\mu}^2 N}{z^2} \tag{7}$$

Solving for N:

$$N \geq \frac{z^2}{\lambda^2 \hat{\mu}^2 \chi_{K,\beta}} \left(\sum_{i=1}^{K-1} n_i (x_i - \hat{\mu})^2 + n_K (x_K - \hat{\mu})^2 \right) \tag{8}$$

Since $(x_K - \hat{\mu})^2$ is not known until after the Kth sample, we use an estimate. By the Central Limit Theorem:

$$E[(x_K - \hat{\mu})^2] = \frac{\sigma^2}{n_K}$$

Replacing this in eq. (8):

$$N \geq \frac{z^2}{\lambda^2 \hat{\mu}^2 \chi_{K,\beta}} \left(\sum_{i=1}^{K-1} n_i (x_i - \hat{\mu})^2 + \sigma^2 \right)$$

Setting an upper-bound on σ^2 using eq. (4), and pulling out the common factor from the terms within the parenthesis:

$$N \geq \frac{z^2}{\lambda^2 \hat{\mu}^2 \chi_{K,\beta}} \left(\sum_{i=1}^{K-1} n_i (x_i - \hat{\mu})^2 \right) \left(1 + \frac{1}{\chi_{K-1,\beta}} \right) \quad (9)$$

Eq. (9) gives us N, the total sample size in number of blocks that would be sufficient to meet the quality test. Using this, the optimal sample size for the K^{th} round can be calculated as $n = N - N_{K-1}$.

3.2.1.5 *Special Case: No Matching Rows*

In the case where no matching rows are found in the two initial samples, we follow the following strategy: a sample of double the size in the previous iteration is taken, till at least one matching row is found, or till the total number of blocks sampled reaches a pre-determined threshold. If no matching rows are found till the threshold is reached, the query cardinality is estimated as zero. If matching rows are found in the j-th iteration, the next sample is calculated using eq. (9), where $j = K-1$ and $x_i=0$ for iteration $i < j$.

3.2.1.6 *Sampling for complex operators*

Statistics queries can be complex, involving joins, group bys, etc. To get optimal plans for complex statistics queries, Oracle sends estimates generated for the parts of the statement earlier. This is done using opt_estimate hints. An example hint can be seen in Q5.

Currently Oracle uses sampling only for the largest table in the complex statistics query and estimates the result using the formulas mentioned in sections 3.2.1.2 – 3.2.1.5. This can be improved using the techniques for join cardinality estimation proposed in [20].

3.2.1.7 *Sampling without Quality Metric (older approach)*

Before the introduction of the quality-metric based approach described previously, the standard approach used by adaptive sampling was to take a single sample of a pre-determined size, and use the cardinality estimate arrived via this sample as the ground truth, without further statistical validation or extra rounds of sampling.

Further rounds of sampling were performed only in the case of the cardinality estimate yielding a value of zero. In this case, the sample size was successively doubled until a non-zero cardinality estimate was arrived at, or the entire table had been read.

3.2.2 *Time budget and enforcement for statistics queries*

Oracle keeps track of performance data for previously executed statements. Historical execution information is also available in Automatic Workload Repository [15]. This information is used for budgeting the time used for statistics queries. Oracle allots a fraction of the time it spends in actually executing the query in the past for executing the statistics queries. Once the limit allotted for a statistics query is reached, the query execution is stopped and the results generated so far are retrieved. The result will not be as accurate as when the statement executes to completion, but can be sufficient for the purpose of query optimization.

3.3 SQL Plan Directives

SPDs are persistent objects that have run time information of SQL or SQL constructs. These objects are stored in the *Dictionary*, which can be used to improve statistics gathering and query optimization on future executions. Currently Oracle has two types of directives – “Adaptive Sampling” and “Adaptive Sampling Result” directives. They are described next.

3.3.1 *Adaptive Sampling Directives*

Adaptive sampling directives are created if execution-time cardinalities are found to deviate from optimizer estimates. They are used by the optimizer to determine if statistics queries (using sampling) should be used on portions of a query. Also, these types of directives are used by the statistics gathering module to determine if additional statistics should be created (e.g. extended statistics). The directives are stored based on the constructs of a query rather than a specific query, so that similar queries can benefit from the improved estimates.

Creation of directives is completely automated. The execution plan can be thought of as a tree with nodes that evaluates different SQL constructs of the query. During compilation of the query (more precisely in *Code Generator*), the constructs evaluated in these nodes are recorded in a compact form in the system global memory area (*SGA*), and can be looked up later using a signature. The signature enables sharing of a construct between queries.

For example consider Figure 2, node 6 of the query plan for Q1 scans the Products table with predicates on columns `PROD_CATEGORY`, `PROD_SUBCATEGORY`. The signature in this case will be built using `PRODUCTS`, `PROD_CATEGORY`, `PROD_SUBCATEGORY`. That is, the signature does not use the values used in the predicates. So if another query has predicates on the same set of columns but with different values, the construct in the SGA can be shared.

At the end of execution of every query, the *Execution Engine* goes over all nodes in the execution plan starting from the leaf nodes and marks those SQL constructs corresponding to node in SGA, whose cardinality estimate is significantly different from the actual value. The nodes whose children have misestimates are not marked, as the misestimate can be caused by a misestimate in the children. For example, in Q1, the optimizer has misestimated the cardinality for products table in node 6. The construct in this node (`PRODUCTS` table with `PROD_CATEGORY` and `PROD_SUBCATEGORY`) is marked while that of the parent nodes 5, 4 etc are not. The SQL constructs that are marked (because they caused a misestimate) are used for creating the directive. The creation is done periodically by a separate background process, called *MMON*. The directives are stored persistently in *Dictionary* along with the objects that constitute constructs. They are called **directive objects**. In our example, PRODUCTS, PROD_CATEGORY, PROD_SUBCATEGORY are the directive objects created for the misestimate in node 6 of Q1. The directive can be used for other queries where these directive objects are present.

Cost Estimator estimates the cardinality for SQL constructs using the available pre-computed statistics in *Dictionary* in the normal way. Once this is done, it will look for any directive that exists for the construct. It will request *Statistics Query Engine* to execute statistics adaptive sampling query and get the more accurate estimate if a directive exists for the construct.

One straight forward way to check if a directive exists for a construct is to build the signature of the construct and see if there exists a directive with the same signature. To maximize the usage of directives and reduce the number of directives created, instead of doing an exact match on the signature, we check if there is a directive that has a subset of objects of the current construct being estimated. If we find such a directive, we execute the statistics adaptive sampling query. For example, the directives created for products table during execution of Q1 can be used by another query with an additional predicate on products table.

As mentioned earlier, we do not create directives for a node if there is misestimate for its children. Instead a directive for the children is created. If the misestimate in the parent node still manifests without any misestimates in child nodes after using the directives for children, a directive for the parent node is created. In this case the misestimate in parent is not caused by children. The overall process is shown in **Figure 4**.

3.3.2 Adaptive Sampling Result Directives

Adaptive sampling directives reduces the number of statistics queries executed in the system by executing statistics queries only if there is a directive created for the construct it is estimating cardinality for. For the statistics queries executed, it still adds an overhead to compilation. The same statistics queries may get executed for several top level SQL statements. We use directive infrastructure to avoid the overhead of this repeated execution.

The result of the statistics query is stored in a directive of type **Adaptive Sampling Result**. This type directive has the following directive objects:

- The tables along with its current number of rows referenced in the statistics query.
- The SQL identifier (sqlid). It is-a hash value created based on the SQL text.
- A signature of the environment (bind variables etc) in which the statistics query is executed.

This type of directive is created immediately after executing a statistics query in *Statistics Query Engine*. The usage of the result stored in these type of directives is as follows:

- The *statistics query engine* first checks if a directive is created for the statistics query before executing the statement. The lookup is done based on the sqlid of the statistics query.
- If there is a directive, we check if the result stored in the directive is stale. The result can be stale if some DML has happened for any of the tables involved in the statistics query. If the current number of rows (maintained in SGA) for any of the tables is significantly different from what is stored in the directive, we consider the directive as stale.
- If a directive is stale, we mark it as such and execute the statistics query to populate the new result in the directive.

3.3.3 Automatic extended statistics

In real-world data, there is often a relationship or correlation between the data stored in different columns of the same table. For example, in the products table, the values in `PROD_SUBCATEGORY` column are influenced by the values `PROD_CATEGORY`. The optimizer could potentially miscalculate the cardinality estimate if multiple correlated columns from the same table are used in the where clause of a statement. Extended statistics allows capturing the statistics for group of columns and helps the optimizer to estimate cardinality more accurately [13]. Creation of extended statistics was manual when it was introduced in Oracle 11g. Oracle had also introduced APIs to find all column groups in a given workload and to create extended statistics for all of them [13].

In Oracle 12c, the extended statistics are automatically created for all the column groups found in the SQL constructs that caused the misestimate. This avoids the creation of extended statistics for unnecessary group of columns that are not causing a misestimate in cardinality and suboptimal plans. The automatic creation of extended statistics relies on the SPD infrastructure explained in section 3.3.1. The adaptive sampling directives maintain different states depending on whether the corresponding construct has the relevant extended statistics or not. It goes through the following state changes, as shown in Figure 5.

- NEW: When a directive is created as described in section 3.3.1 it will be in the NEW state.
- MISSING_EXT_STATS: When optimizer finds directives corresponding to the constructs in the query it will check if there is a column group in the construct. If no extended statistics are created yet for the group then those column groups will be recorded in the dictionary tables. The state of the directive will be changed to MISSING_EXT_STATS.
- HAS_EXT_STATS: The statistics gathering process (either manual, or automatic job) creates extended statistics for the groups that are monitored. If optimizer finds the extended statistics for the column group corresponding to the directive, it will change the state to HAS_EXT_STATS. Statistics queries are not executed for the directives with HAS_EXT_STATS state. If the extended statistics produce more accurate estimate, it avoids the overhead of executing statistics queries.

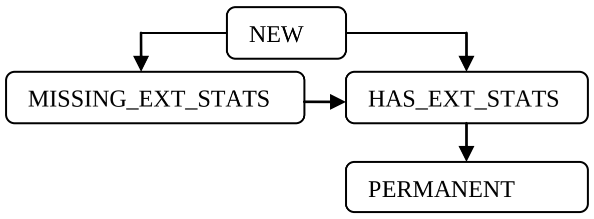


Figure 5. SPD State Transition Diagram

- PERMANENT: If *Execution engine* finds misestimate for a construct and if the construct has a directive with state HAS_EXT_STATS, it goes throgh a state transition to PERMANENT and will use statistics queries from then onwards for the directive. This is because the extended statistics in previous state did not help to correct the misestimate for some queries.

All the states except HAS_EXT_STATS execute statistics queries.

4. PERFORMANCEEVALUATION

Adaptive statistics feature is available in Oracle 12c which has been in production for over 4 years and this section presents a performance study of the feature in Oracle 12c Release 2. The various aspects of adaptive statistics are evaluated on a publicly available workload as well as using a real customer workload.

4.1 Workloads

We ran our experiments on two workloads.

- *IMDB Workload* [7]: We used the IMDB dataset benchmark that uses data from the Internal Movie Data Base (IMDB). We ran our tests on the 113 queries in the benchmark.
- *Customer X Workload*. This is a real-world workload from a large market research company. We ran our tests on a sample of 29 queries with a diverse profile based on the execution time: short, medium and long.

4.2 Experiment Setup

The experiments were run on a 48 CPU, X86 machine running Linux 3.8. The machine has 512GB of physical memory.

For each of the workloads, we ran six different experiments:

1. *Baseline*. Adaptive statistics feature is not used.
2. *Adaptive statistics without quality metrics (AS w/o QM)*. The quality of the results from the statistics queries is not measured in this experiment as described in section 3.2.1.7.
3. *Adaptive statistics with quality metrics (AS w/ QM)*. Quality test is performed on the results of the statistics queries and statistics queries are re-executed with higher sample size as described in section 3.2.
4. *Adaptive statistics with quality metrics and with cache (AS w/ QM+Cache)*. The results of the statistics queries are cached persistently (section 3.3.2) and they are fetched from the cache instead of executing them.
5. *Adaptive statistics with directives (AS w/ SPD)*. Statistics queries are executed only if there is a corresponding SPD (section 3.3.1).
6. *Adaptive statistics with directives and extensions (AS w/ SPD+EXT)*. If there is extended statistics corresponding to a SPD, the corresponding statistics queries are not executed (section 3.3.3) in this setup.

In each experiment, we measure the parse time (time to generate the execution plan), the total execution time (aggregate time spent by all the processes that participated in the execution) and the wall clock run time (the difference between the start and end of the execution of the query). Note that if the query executes in serial then the total execution time is same as the wall clock run time. However, for a parallel query execution, the wall clock time is less than or equal than the total execution time. All the experiments used a degree of parallelism of 16.

4.3 Results

4.3.1 IMDB Workload

Out of 113 queries, 107 queries changed plans with "AS w/ QM". We analyzed the plan changes for a sample of 23 queries. 20 queries had a join order change. Join method changed from nested-loops to hash join in 12 queries and 2 queries had the reverse change, i.e. from hash join to nested-loops join.

Figure 6 shows the wall clock run time and the average hard parse time and Figure 7 shows the total execution time and average hard parse time for the workload.

From Figure 6, it can be seen that compared to baseline, with AS w/o QM, the wall clock run time improves by about 70% at the expense of long parse time. The average parse time worsens further by about 56% if quality metrics are not used for statistics queries. For this workload, the wall clock run time remains almost the same with and without quality metrics. We verified that the cardinality estimated is more accurate when using the Quality Metric. However, this improved cardinality estimate does not result in different plans in this workload.

On the other hand, as can be observed from Figure 7, the total execution time increases when adaptive statistics are used though the wall clock run time decreases. This shows that using adaptive statistics, the optimizer is able to find plans that are more parallelizable. These plans use more resources and finish execution faster.

Note that "AS w/ QM+Cache" experiment has a parse time that is close to that of Baseline. This means that, if the workload is executed multiple times, only the first execution incurs the higher parse time, which is then amortized over later executions. In addition, usage of SPDs reduces the parse time considerably even for the first execution by selecting only the statistics queries that correct cardinality misestimates. Using directives (AS w/ SPD) decreased the wall clock time by about 6.8%. However, the improvement is significantly lower than that of AS w/ QM (no directives) in this workload. This is because, SPDs are created only for the misestimates seen in the final plan. It takes several executions with different plans (e.g. different join orders) to generate more directives. Also there is no guarantee that we will see the join orders that has misestimate in the final plan even if the query is run several times. Hence, we created directives only from the first three executions in this experiment.

Finally, Adaptive Statistics w/Directives & Extension parse time is better than without using Extensions, since statistics queries are not executed if there are extended statistics to help to produce better quality estimates.

Figure 10 shows a scatter graph with the relationship between the run-time of a query and the absolute improvement in the run-time, as a result of AS w/ QM. In the graph, the queries can be divided into two categories based on the difference in the impact of adaptive sampling: the short queries (<30 seconds run-time), and the long queries (>30 seconds run-time). For the long queries, the absolute improvement grows roughly linearly with the run-time, so that the percentage improvement is roughly constant. This is a desirable property, as the long running queries show the greatest improvement. However, the same does not hold for short running queries, with many showing a small improvement, or even a significant increase in the run-time (in terms of percentage). However, with the exception of one query, most queries that show an increase in the run-time have wall clock run-time of less than 30 seconds, and hence do not have a large negative impact on the overall wall clock run-time.

4.3.2 Customer X Workload

Out of 29 queries, 21 queries changed plans with "AS w/ QM". The changes fall in the following categories: (1) access path changes (different indexes, from index scan to sequential table scan) [4 queries], (2) Join order [15 queries], (3) Join method (from nested loop to hash join) [3 queries], Transformation: join predicate push down and Group by placement is not chosen in 3 and 1 queries respectively.

We can see trends similar to the IMDB benchmark in the Customer X workload (Figure 8 and Figure 9) except for the following differences.

- With quality metric, the wall clock time or total execution time is better than not using quality metric. So the quality test makes a difference in this workload.
- The total execution time for "AS w/ SPD" is about 50% better than baseline, while spending a very small amount of extra parse time. However, wall clock time does not show a comparable improvement. This is due to two queries that, based on better estimates choose a plan that distributes the records differently, which causes some parallel processes to run idle. If we exclude these two queries, the wall clock time, is comparable to the total execution time, in being about 50% better than baseline.

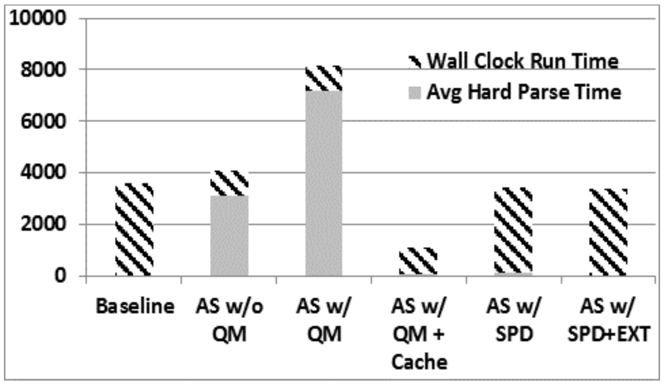


Figure 6. IMDB Workload – Wall Clock Run Time

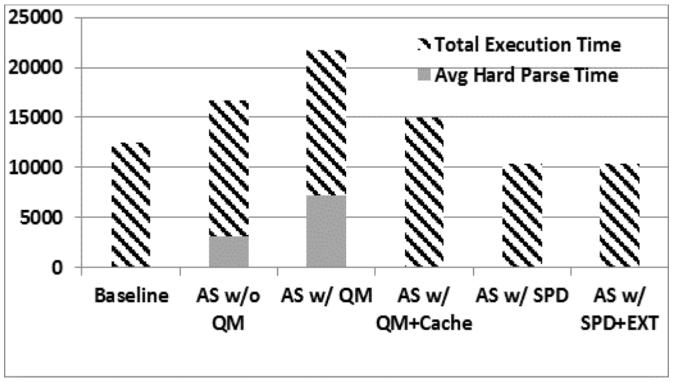


Figure 7. IMDB Workload – Total Execution Time

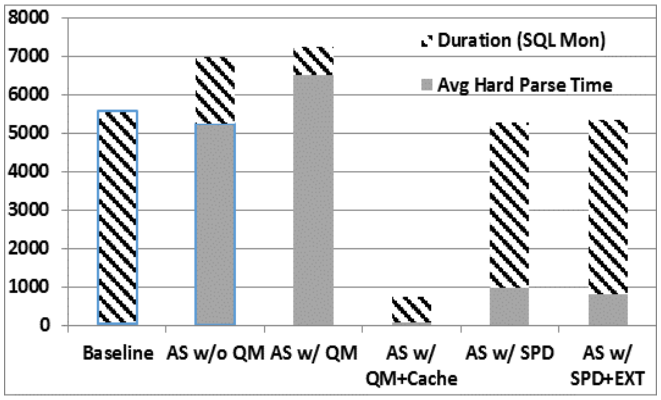


Figure 8. Customer X Workload – Wall Clock Run Time

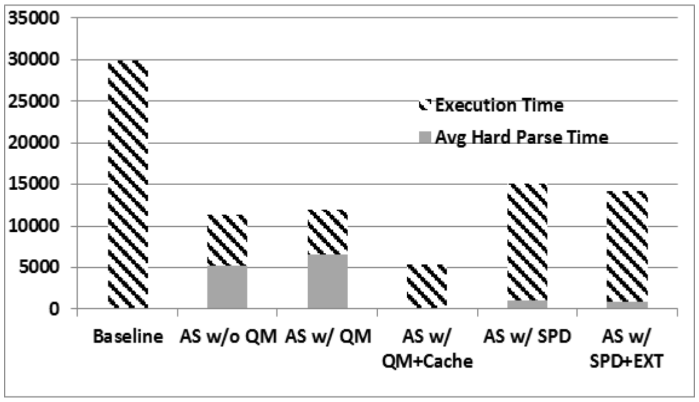


Figure 9. Customer X Workload – Total Execution Time

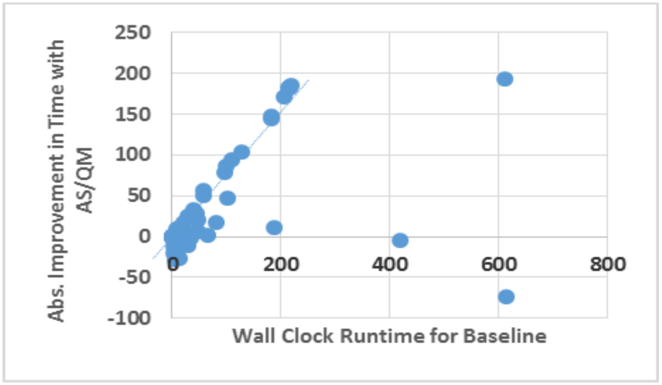


Figure 10. IMDB Workload: Run-time vs Improvement with Adaptive Statistics (using QM)

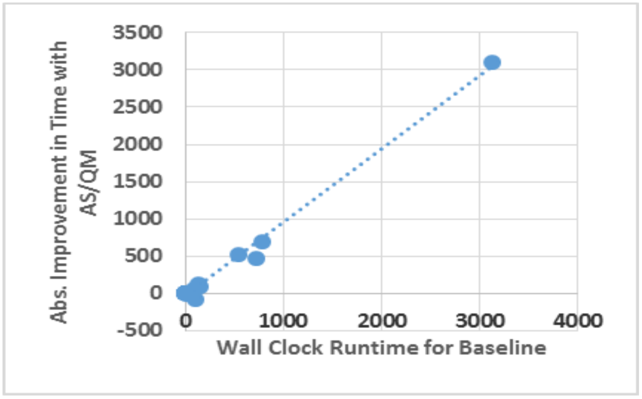


Figure 11. Customer X Workload: Run-time vs Improvement with Adaptive Statistics (using QM)

- In this workload, the parse+total execution time is three times better than using adaptive statistics even without cache, directives and extensions.

Figure 11 shows a scatter graph, displaying the relationship between the query run-time, and the absolute improvement in the run-time. It demonstrates similar pattern as in IMDB workload: the absolute improvement for a query increases linearly with the query runtime.

5. CONCLUSION

We presented the problems caused by incorrect statistics of intermediate results, surveyed the existing solutions and presented the approach we introduced in Oracle 12c. We performed experiments using two SQL workloads, IMDB and a real customer workload. The experiments show that the approach of computing statistics during compilation using statistics queries gives significant improvement on execution time. The techniques used for reducing compilation overhead of statistics queries were effective in these workloads.

6. REFERENCES

[1] R. Ahmed, A. W. Lee, A. Witkowski, D. Das, H. Su, M. Zaït, and T. Cruanes. Cost-based query transformation in oracle. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006,* pages 1026–1036, 2006.

[2] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zaït, and M. Ziauddin. Automatic SQL tuning in oracle 10g. *In (e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004,* pages 1098–1109, 2004.

[3] A. El-Helw, I. F. Ilyas, and C. Zuzarte. Statadvisor: Recommending statistical views. PVLDB, 2(2):1306–1317, 2009.

[4] C. A. Galindo-Legaria, M. Joshi, F. Waas, and M. Wu. Statistics on views. In *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, pages 952–962, 2003.

[5] A. C. König and G. Weikum. Combining histograms and parametric curve fitting for feedback-driven query result-size estimation. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK,* pages 423– 434, 1999.

[6] A. W. Lee and M. Zaït. Closing the query processing loop in oracle 11g. PVLDB, 1(2):1368–1378, 2008.

[7] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? PVLDB, 9(3):204–215, 2015.

[8] G. Moerkotte, T. Neumann, and G. Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. PVLDB, 2(1):982– 993, 2009.

[9] W. Navidi. *Statistics for Engineers and Scientists*. McGraw-Hill, New York, 2006.

[10] IBM Developer Works. Get the most out of DB2 optimizer: Leveraging statistical views to improve query execution performance. https://www.ibm.com/developerworks/data/library/techarticle/dm-1305leverage.

[11] IBM Developer Works. Understand column group statistics in db2. http://www.ibm.com/developerworks/data/library/techarticle/dm-0612kapoor/index.html.

[12] MSDN. Statistics, SQL Server 2016. https://msdn.microsoft.com/en-us/library/ms190397.aspx.

[13] Oracle Blog. Extended statistics. https://blogs.oracle.com/optimizer/entry/extended_statistics.

[14] Oracle Blog. How do I know what extended statistics are needed for a given workload. https://blogs.oracle.com/optimizer/entry/how_do_i_know_what_extended_statistics_are_needed_for_a_given_ workload.

[15] Oracle Documentation. Overview of the automatic workload repository. http://docs.oracle.com/cd/E11882_01/server.112/e41573/autostat.htm#PFGRF02601.

[16] Penn State. Chi-square table. http://sites.stat.psu.edu/~mga/401/tables/Chi-square-table.pdf.

[17] Wikipedia. Boole's inequality. https://en.wikipedia.org/wiki/Boole's_inequality.

[18] U. Srivastava, P. J. Haas, V. Markl, M. Kutsch, and T. M. Tran. ISOMER: consistent histogram construction using query feedback. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA,* page 39, 2006.

[19] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's learning optimizer. *In VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001,* Roma, Italy, pages 19–28, 2001.

[20] D. Vengerov, A. C. Menck, M. Zaït, and S. Chakkappen. Join size estimation subject to filter conditions. PVLDB, 8(12):1530–1541, 2015.

[21] R. R. Wilcox. Fundamentals of Modern Statistical Methods: Substantially Improving Power and Accuracy. Springer, 2010.

[22] W. Wu, J. F. Naughton, and H. Singh. Sampling-based query re-optimization. *In Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016,* San Francisco, CA, USA, June 26 - July 01, 2016, pages 1721–1736, 2016.

[23] M. Ziauddin, D. Das, H. Su, Y. Zhu, and K. Yagoub. Optimizer plan change management: improved stability and performance in oracle 11g. PVLDB, 1(2):1346–1355, 2008.