Mining Conditional Cardinality Patterns for Data Warehouse Query Optimization

Mikołaj Morzy[1] and Marcin Krystek[2]

[1] Institute of Computing Science
Poznan University of Technology
Piotrowo 2, 60-965 Poznan, Poland
`Mikolaj.Morzy@put.poznan.pl`
[2] Poznan Supercomputing and Networking Center
Noskowskiego 10, 61-704 Poznan, Poland
`mkrystek@man.poznan.pl`

Abstract. Data mining algorithms are often embedded in more complex systems, serving as the provider of data for internal decision making within these systems. In this paper we address an interesting problem of using data mining techniques for database query optimization. We introduce the concept of conditional cardinality patterns and design an algorithm to compute the required values for a given database schema. However applicable to any database system, our solution is best suited for data warehouse environments due to the special characteristics of both database schemata being used and queries being asked. We verify our proposal experimentally by running our algorithm against the state-of-the-art database query optimizer. The results of conducted experiments show that our algorithm outperforms traditional cost-based query optimizer with respect to the accuracy of cardinality estimation for a wide range of queries.

1 Introduction

Knowledge discovery is traditionally defined as a non-trivial process of of finding valid, novel, useful, and ultimately understandable patterns and regularities in very large data volumes, whereas data mining is considered a crucial step in the knowledge discovery process, consisting of the application of a given algorithm to a given dataset in order to obtain the initial set of patterns [6]. In this paper we show how data mining can be exploited to enhance the query optimization process. For this purpose we embed a data mining algorithm into the query optimizer of the relational database management system. Somehow contrary to traditional data mining, where domain experts are usually required to assess the quality of discovered patterns or to fine-tune algorithm parameters, embedded data mining solutions do not allow external intervention into the process. Therefore, we make the following assumptions regarding presented solution. Firstly, the data being fed into the algorithm must be cleaned and of high quality as no iteration between data mining phase and data pre-processing phase are possible.

The results of embedded data mining algorithm must be represented in the form that allows automatic validation and verification, because all results are being instantly consumed by subsequent process steps and no human validation or verification of patterns is possible. Finally, data mining algorithms being embedded outside of knowledge discovery process must not require sophisticated configuration and parametrization, ideally, these should be either zero-conf algorithms or auto-conf algorithms.

In this paper we present a way to embed a data mining algorithm into the query optimization process in order to enhance the quality of estimates made by the optimizer. We analyze the schema and discover potential join conditions between database tables based on referential constraints. For all tables that can be meaningfully joined, we compute conditional cardinalities. Our method works best in the data warehouse environment. In the case of queries asked against a snowflake schema or star schema, our ability to accurately predict cardinalities of attributes for a fixed set of other attribute values (i.e., a set of dimension attribute values) in the presence of many joins and selection criteria helps to discover better query execution plans. The original contribution of the paper is the following. We develop a simple knowledge pattern, called the conditional cardinality. We design an algorithm that identifies suitable pairs of attributes that should be included in conditional cardinality computation. The identification of these pairs of attributes is performed by mining the database dictionary. Finally, we show how the discovered conditional cardinality counts can be used to better estimate the query result size. We verify our proposal experimentally using the state-of-the-art database query optimizer from Oracle 10g RDBMS to prove the validity and efficacy of the approach.

The paper is organized as follows. In Section 2 we briefly discuss related work. Section 3 introduces the concept of conditional cardinality count. We present our algorithm for computing conditional cardinalities in Section 4 and we report on the results of the experimental evaluation of our proposal in Section 5. We conclude this paper in Section 6 with a brief summary and a future work agenda.

2 Related Work

There are numerous works on both query optimization and data mining, but, surprisingly, few works have been published on using data mining techniques to enhance query optimization process [1, 9]. Most research focused on enhancing existing statistics, usually by the means of detailed histograms. An interesting idea appeared in [8], that consisted in using approximate histograms that could be incrementally refreshed to reflect the updates to the underlying data. Specialized histograms for different application domains, have been proposed, including data warehouses. For instance, in [3] the concept of using query results for multidimensional histogram maintenance is raised. The dynamic aspect of histograms is addressed in [5] where the authors develop an algorithm for incremental maintenance of the histogram.

Another research domain that influenced our work concerned using complex statistics in query optimization. The use of Bayesian Networks in query optimization is postulated in [7]. The need to reject the attribute value independence assumption is advocated in [10]. Finally, the idea of using query expression statistics for query optimization has been proposed in [2] and a framework for automatic statistics management was presented in [11].

Concepts presented in this paper are similar to the concepts introduced in [4], where the estimation of the aggregated view cardinality is performed using k-dependencies. The main difference is that k-dependencies represent a-priori information derived from the application domain, whereas conditional cardinality patterns are computed automatically from the data.

3 Conditional Cardinality

In this section we formally introduce the concept of conditional cardinality patterns. Let R,S denote database relations, and let $A \in R$, $B \in S$ be attributes A, B of relations R,S, respectively. Let $val(R.A)$ be the number of distinct values of the attribute A in relation R. Traditionally, the selectivity factor for an attribute A is defined as $sel(R.A) = \frac{1}{val(R.A)}$. Let n denote the number of tuples resulting from joining relations R and S on some equality join condition, presumably, using a foreign key constraint. We are interested in finding the number of distinct values of the attribute B in $R \bowtie S$ for a fixed value of the attribute A. Let $\{a_1, a_2, \ldots, a_m\}$ be the values of the attribute A, and let $card(B|a_i) = |\{t \in R \bowtie S : R.A = a_i\}|$ denote the number of distinct values of the attribute B in $R \bowtie S$ where $A = a_i$.

The *conditional cardinality* of the attribute $B \in S$ conditioned on the attribute $A \in R$ is the averaged number of distinct values of the attribute B appearing in the result of the join $R \bowtie S$ for a fixed value of A and is given by

$$card(B|A) = \frac{1}{m} \sum_{i=1}^{m} card(B|a_i)$$

Using conditional cardinality allows for more accurate estimation of the cardinality of a query. Having computed $card(B|A)$ we can estimate the size of the result of a query Q of the form `SELECT * FROM R JOIN S WHERE R.A = 'a' AND S.B = 'b'` to be

$$card(Q) = \frac{sel(R.A) * n}{card(B|A)} \ or, equally, \ card(Q) = \frac{sel(S.B) * n}{card(A|B)}$$

Note that we do not consider the quality of the estimation of n, the cardinality of $R \bowtie S$ and we do not require a specific type of join (e.g., a natural join or an outer join). Computing all conditional cardinalities between any pair of attributes from the schema is obviously unfeasible and computationally prohibitively expensive. We compute compute conditional cardinalities only for pairs of joinable attributes, i.e., pairs of attributes from tables that can be joined by

one-to-one, one-to-many, or many-to-many relationship. We refer to such conditional cardinalities as *conditional cardinality patterns*. We are well aware that using a grandiose term *pattern* to describe such a simple measure may spur criticism and may seem unmerited. We use this term purposefully to stress the fact that these varying counts are fundamental in the entire cardinality estimation procedure.

4 Algorithm for estimating query result size

In this section we present an algorithm for estimating the number of tuples returned by an SQL query using conditional cardinality patterns. Our algorithm works under the following three assumptions. Firstly, the database is in either star or snowflake schema. Secondly, only equality conditions are allowed to appear in the query, both for joining tables and for issuing selection criteria. Lastly, queries consist of `JOIN` and `WHERE` clauses only, with no subqueries, in-line views, set containment operators, range operators, and such. Below we present the outline of the algorithm. These assumptions reflect the current state of our research, but we expect to relax them as more research is conducted and formulas for arbitrary selection conditions are determined.

4.1 Algorithm steps

1. Split attributes from the `WHERE` clause into two element ordered sets $S_i = (A, B)$. In each pair (A, B) the attributes must come from different tables, which are dimensions of the same fact table. If, during set creation, this condition can not be ensured, then the left-over attributes should be placed in singleton sets.
2. Let n be the number of all fact tables used in join clause of query. All n fact tables should be ordered in the way that ensures that the i-th fact table will be in one-to-many relationship with (i+1)-th fact table, for $i = 1, \ldots, n$.
3. For each pair of joinable tables, calculate the join cardinality N. In order to do so, select pairs of attributes from the i-th fact table and one of its dimensions. If it is the first execution of step 3. and step 6. was not executed yet, then N should be equal to the join cardinality of tables from which the attributes come from. Otherwise, if step 6. was executed, N should be equal to the value returned in step 6.
4. Let $A \in R$. C_i is an attribute such that: $C_i \in S_j, S_j \neq R$, where R and S_j are dimensions of the same fact table and C_i belongs to the pair of attributes, which were analyzed in the previous iteration of the algorithm. Let k be the number of such attributes. Then, the selectivity of the attribute A is given by

$$\max \left\{ sel(A), \frac{1}{card(A|C_1)}, \frac{1}{card(A|C_2)}, ..., \frac{1}{card(A|C_k)} \right\}$$

5. calculate the cardinality of the result of the query as

$$L = \frac{N * sel(A)}{card(B|A)}$$

   If the set of attributes has only one element, then L should be computed as $L = N * sel(A)$
6. If there are other pairs of attributes that have not been evaluated for the i-th fact table then go to step 3. Perform computation for the next pair of attributes, with the exception that N should be set to the value returned in step 6.
7. If there exist fact tables that have not been analyzed, then let S be a dimensional table joining current i-th fact table with the (i+1)-th fact table. Let us use the following notation. Let n denote the number of tuples in the table S, let m denote the number of tuples in the (i+1)-th fact table. For current value of L, computed in step 6. do: $L = \frac{L*m}{n}$ and go to step 2.
   Computed value L is the estimation of the number of tuples returned by the query.

4.2 Preprocessing

Analyzing database schema is the first step in the preprocessing procedure. The discovery of relationships between user tables is based on data dictionary view. The view contains information about all user table names, constrain names and their types. Because we are looking for tables that are joined by one-to-many or many-to-many relationships, we only consider tables with primary key and foreign key constraints. Data dictionary is looked up in search of the list of tables that remain in a one-to-many or a many-to-many relationship. Cardinality of the result of join operation on two tables is one of the start parameters in estimation algorithm. To avoid computing this value each time the algorithm analyzes a pair of attributes drawn from these tables, we count the cardinality of the join operation and store this value together with table relationship information.

4.3 Gathering statistics

Estimation algorithm is based on statistical information about data stored in the database. Conditional cardinality patterns are an extension of traditional statistics gathered in database dictionary. The estimation process makes an implicit assumption that values of all attributes have constant distribution. This assumption is seldom true. Disjunctive attribute values distort estimation process, so we have decided to identify such values and process them in a special way. Let $c_i = |\{r \in R : r.A = a_i\}|$ denote the number of tuples in the relation R, for which the attribute A has value a_i, and let p denote the threshold above which an attribute value is considered disjunctive. Let $\bar{c} = \frac{1}{m} \sum_i c_i$ be the average number of tuples for one distinct value of the attribute A, and let σ_c be the standard deviation of c computed over all possible values of the attribute

A. For each $A \in R$ and for each value a_i of A we compute the z-score of c. If z-score falls outside of the range $\langle -p, p \rangle$, then we consider the value a_i of the attribute A as disjunctive. The choice of p is arbitrary and should be guided by the Chebyshev's inequality which states that "no more than $\frac{1}{k^2}$ of the values are more than k standard deviations from the mean". The user should set the value of p accordingly to the desired sensitivity to atypical attribute values. For all attributes we collect information about the number of its distinct values. We also gather information about minimum, maximum and average value of each numerical attribute.

4.4 Conditional cardinality

Condition cardinality is computed for each ordered pair of attributes (A, B). Pair generation process must ensure that both attributes belong to different relations which remain in one-to-many or many-to-many relationship. Relation pairs and correct attributes can be chosen based on information gathered during preprocessing procedure. By creating a cartesian product of attributes, one can easily find all possible pairs of attributes of the two relations. Because pairs are ordered and $(A, B) \neq (B, A)$, so for each pair we must also create its mirror pair, by inverting attribute positions in the pair. Finally, for each pair we check if the first attribute in the pair has any disjunctive value. If so, then this pair is cloned and saved with annotation about which disjunctive value it applies to. Otherwise the pair is saved without any annotation. Condition cardinality is computed accordingly to its definition from Section 3 for each pair of attributes. If a pair has an annotation about disjunctive value of its first attribute, then condition cardinality is computed as the number of distinct values of second attribute in a joined result relation, where the first attribute is equal to the annotated value.

4.5 Query generator

Query generator is a simple tool, prepared specially for the experiment. Queries created by the generator consist of three clauses: a `SELECT`, `JOIN` and `WHERE` clauses. The complexity of the last two clauses is controlled by input parameters. From all available tables one table is randomly chosen and it is inserted into the query's `JOIN` clause. If this table is a fact table, then all its dimensions are also inserted in the `JOIN` clause. Next, we choose some other fact table that can be joined with current one (fact tables are in a many-to-many relationship) and repeat the insertion procedure. If the next fact table does not exist or the `JOIN` clause is long enough, then we assume that generation of the `JOIN` clause is finished. For each table in the `JOIN` clause we select all attributes that meet our requirements. To satisfy each condition type cardinality, some attributes are taken randomly from this set, and put in the query's `WHERE` clause in an appropriate form. Finally, the `SELECT` clause consists of one random attribute form each table in the query's `JOIN` clause.

5 Experiments

The first goal of the experiment was to prove the existence of correlation between the estimated and the real number of tuples. The second goal of the experiment was the comparison of the accuracy gained by using conditional cardinality patterns with the accuracy of the leading commercial solution. As our testbed we have chosen Oracle 10g database management system. All experiments were conducted on a PC with openSUSE 10.2 GNU/Linux and Oracle 10g Enterprise Edition 10.2 database. All queries were generated on top of the default Sales History (SH) data warehouse schema pre-installed in the database. Because of time and technical constraints, the size of the default SH schema was reduced. All tuples from SALES table were grouped by customer id, and only groups counting between 4 and 70 tuples were retained. Next, we have reduced sizes of all dimension tables by deleting tuples missing from SALES table. Statistics of the original and reduced SH schema are presented in Table 1.

table	size before reducing	size after reducing
COUNTRIES	23	18
CUSTOMERS	55500	1808
PRODUCTS	72	72
CHANNELS	5	4
TIMES	1826	1431
PROMOTIONS	503	4
SALES	918843	75922
COSTS	82112	81425

Table 1. Statistics of the SH schema

During the experiment 130 different queries were generated, 44 among them returned at least one tuple and 86 returned no tuples. Query generator allowed only for equality conditions in generated queries. Equality conditions are very selective, therefore, the number of conditions allowed in the WHERE clause was set to 10% of all attributes that could have been used in the clause. On average, this setting resulted in WHERE clauses of 6 conditions (not including table join conditions). To ensure optimal conditions for Oracle optimizer all possible statistics were gathered for schema SH. For the estimation algorithm, detail information about table attributes was gathered, as described above, and conditional cardinality for all possible pairs of attributes was computed.

For each query we compute three values: the true number of tuples returned by the query (denoted `real`), the estimated number of tuples using condition cardinality patterns (denoted `cond.card`), and the number of tuples estimated by the Oracle optimizer (denoted `oracle`). Estimations vary from 0 to over 15 000 000. To present results in a more readable form, queries were divided into bins, depending on the their true cardinality. The first bin contains queries that return no rows, the second bin contains queries returning up to 100 rows, the

third bin contains queries returning up to 1000 rows, and so on. For each bin data aggregation was performed in the following way: evaluated values were scaled relatively to the biggest value from the bin. This dominating value was assumed to represent 100%, and the remaining two values were calculated as the percentage of the biggest value. The results depicted in Figure 1 are averaged over all queries.

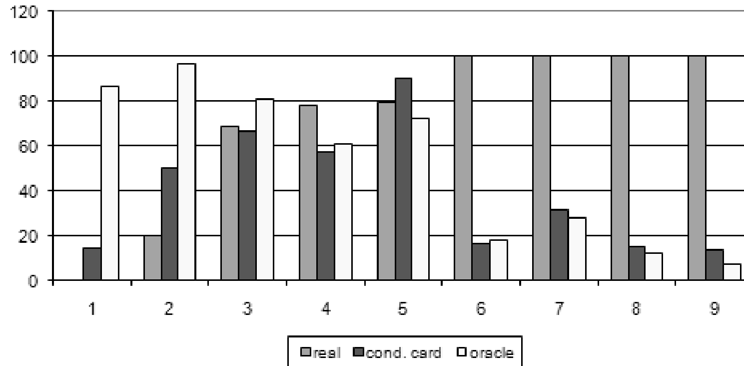

Fig. 1. Precision of cardinality estimates

Based on data received from the test, the Pearson correlation coefficient has been computed between true (random variable X) and estimated (random variable Y) number of returned tuples. The correlation coefficient is $r_{XY} = 0.978$. To prove that the correlation is statistically significant, a t-test has been performed with the null hypothesis H_0 of r_{XY} being insignificant. For the confidence range of 99% the critical value is $t_0 = 2.6148$, whereas the t statistics yields

$$t = \frac{r_{XY}}{1 - r_{XY}^2} * \sqrt{n-2} = 52.75$$

Because $t > t_0$, we reject the null hypothesis and we embrace the opposite hypothesis of the correlation coefficient being significant.

In our experiment many queries return no rows. To assure that a large fraction of random variable X values being 0 does not bias the test, we have repeated it for only non-zero values of the variable X. This time the Pearson correlation coefficient was $r'_{XY} = 0.982$ and the t-statistics as $t' = 33.91$, so the null hypothesis could have been rejected with confidence level of 99%.

Figure 2 presents mistakes committed by estimations. Based on results from Figure 1, each mistake was calculated as the absolute difference between true query cardinality and the respective estimation. To compare the quality of estimates generated by condition cardinality patterns and the Oracle optimizer let us use the following notation. Let x denote the true number of tuples returned by the query, let y denote the number of tuples estimated by using condition cardinality patterns, and let z denote the number of tuples estimated by the Oracle

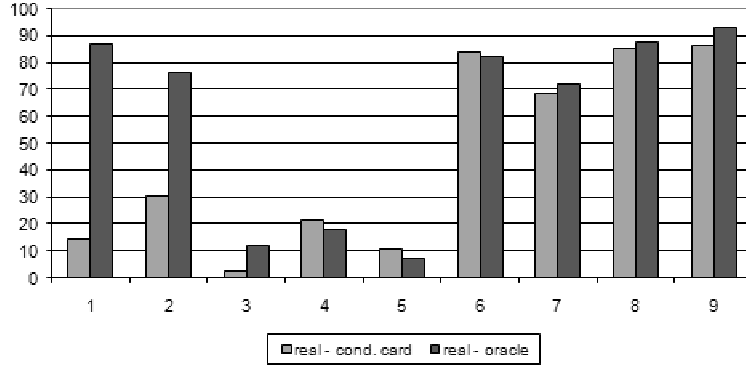

Fig. 2. Comparison of estimate differences

optimizer. In addition, let Δ be the measurement of estimation quality defined as follows: $\Delta_{xy} = |x - y|$ and $\Delta_{xz} = |x - z|$. All estimates tend to minimize the difference Δ, therefore, for each comparison we find the winner as follows: if $\Delta_{xy} < \Delta_{xz}$ then condition cardinality estimation wins, otherwise if $\Delta_{xz} < \Delta_{xy}$, then the Oracle optimizer wins, otherwise if $\Delta_{xy} = \Delta_{xz}$, then we announce the tie. Table 2 summarizes the comparison of the condition cardinality estimation with the Oracle optimizer estimation as the number of wins, losses, and ties.

returned rows	$\Delta_{xy} < \Delta_{xz}$	$\Delta_{xz} < \Delta_{xy}$	total
$x = 0$	72	14	86
$x \neq 0$	29	15	44
total	101	29	130

Table 2. Comparison of condition cardinality with the Oracle optimizer

We can conclude that for 78% of queries condition cardinality estimates are better than the Oracle optimizer. On the downside, we have noticed that when conditional cardinality method miscalculates the cardinality of the query result, usually, the committed error is much larger than the error made by the Oracle query optimizer. We attribute this to the way conditional cardinality is propagated through conditions in the query. When a mistake is made early in the estimation process, this mistake is amplified by subsequent estimations for the remaining selectors, which results in rather a formidable error.

6 Conclusions

In this paper we have used data mining for the query optimization process. We have developed a simple knowledge model, conditional cardinality patterns, and

we have designed an algorithm for identifying promising pairs of attributes. We have used discovered patterns to improve the accuracy of cardinality estimation for typical data warehouse queries. Our experiments were conducted against the state-of-the-art query optimizer from Oracle 10g database management system. The results of conducted experiments show clear advantage of using conditional cardinality patterns in the data warehouse query optimization process. This paper reports on the results of the preliminary research conducted in the field. In the future we intend to extend the framework to handle aggregation queries. We also plan to further utilize database dictionary to mine patterns that might be useful for other database related tasks, such as database maintenance, storage optimization or user management.

References

1. K. Bennett, M. C. Ferris, and Y. E. Ioannidis. A genetic algorithm for database query optimization. In R. Belew and L. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 400–407, San Mateo, CA, 1991. Morgan Kaufman.
2. N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *SIGMOD '02: Proc. of the 2002 ACM SIGMOD international conference on Management of data*, pages 263–274, New York, NY, USA, 2002. ACM.
3. N. Bruno, S. Chaudhuri, and L. Gravano. Stholes: a multidimensional workload-aware histogram. In *SIGMOD '01: Proc. of the 2001 ACM SIGMOD int'l conference on Management of data*, pages 211–222, New York, NY, USA, 2001. ACM.
4. P. Ciaccia, M. Golfarelli, and S. Rizzi. On estimating the cardinality of aggregate views. In *Design and Management of Data Warehouses*, page 12, 2001.
5. D.Donjerkovic, Y.Ioannidis, and R.Ramakrishnan. Dynamic histograms: Capturing evolving data sets. In *ICDE '00: Proc. of the 16th Int. Conference on Data Engineering*, page 86, Washington, DC, USA, 2000. IEEE Computer Society.
6. U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.
7. L. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. *SIGMOD Rec.*, 30(2):461–472, 2001.
8. P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 466–475, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.
9. C.-N. Hsu and C. A. Knoblock. Rule induction for semantic query optimization. In *11th Int. Conf. on Machine Learning*, pages 112–120. Morgan Kaufmann, 1994.
10. V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 486–495, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.
11. S.Chaudhuri and V.Narasayya. Automating statistics management for query optimizers. In *ICDE '00: Proceedings of the 16th International Conference on Data Engineering*, page 339, Washington, DC, USA, 2000. IEEE Computer Society.