Supporting Statistics In Extensible Databases:
A Case Study*

Arie Segev and Abhirup Chatterjee

Walter A. Haas School of Business
University of California at Berkeley
and
Information and Computing Sciences Division
Lawrence Berkeley Laboratory
Berkeley, CA 94720
E-mail: {segev, abhirup}@ux5.lbl.gov

Abstract

This paper presents a framework for supporting regression in extensible databases. This work is motivated by an actual case study that required probabilistic record matching using regression be supported in a database. In this paper, we discuss how the regression function can be implemented in a database using rules and functions. We present some indexing and caching ideas for efficient data processing for retrieval-intensive applications. We also describe how the regression coefficients can be incrementally updated by materializing some of the intermediate statistical results. Finally, we illustrate how sampling techniques can be used to generate the data sets for regression and highlight some of the performance issues in this regard.

1 Introduction

In order to perform statistical analysis, users need access to both data management as well as statistical tools. Even though its impact in business decision making has long been recognized, little effort has been spent so far in supporting statistical analysis in traditional Database Management Systems (DBMSs).

There are some inherent advantages in integrating statistical analysis tools in a DBMS. Firstly, a DBMS allows many intermediate results to be shared in a multiuser environment [15]. Different users using the same data sets could use the statistical results, such as mean or variance, that may have been computed earlier. If any of these estimates are subsequently updated, the new estimates become available to all users of the database. Secondly, DBMSs are well-equipped to process large data sets through efficient storage and indexing. A typical statistical package does not have any general purpose multi-user query optimization capability, which makes it unsuitable in a transaction environment. Finally, a DBMS improves query response time by allowing one to perform data manipulation and statistical analysis on the same data set at the same time rather than treating these as two disparate steps.

The objective of this paper is to illustrate how statistical operations can be supported in an extensible DBMS. Specifically, we focus on regression analysis, though many of the ideas presented here can be used to perform other statistical operations. The particular focus of this paper was motivated by an actual case study, which required that regression be supported in a database, at least to a limited extent. The approach presented here can be considered as a compromise between (a) using separate packages for analysis and data management, and (b) incorporating an entire statistical package inside a DBMS. Since the analytical tools get continuously refined, a DBMS may not be able to keep up with the latest developments in statistics. Thus, some interfacing between a package and the DBMS may still be necessary. However, once the appropriate statistical model, such as logistic regression, has been identified (presumably using the package), one should be able to apply the tools, that is, run the regression from within the database in an

---

*Issued as LBL Technical Report 35292. This work was supported by the NSF Grant IRI-9116770 and by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098.
Over the last decade, there has been growing interest in statistical database research [16]. Aggregate operators, such as, MAX, MIN, COUNT, and SUM, have long been a part of SQL vocabulary. Researchers have devised efficient algorithms for matrix manipulation in a database [9]. Considerable amount of research has also been published on sampling in databases, e.g., [11, 7]. The situation improved further with the advent of extensible DBMSs, such as Starburst [6], Exodus [2] and PostgreSQL [19]. These systems allow users to define, among other things, procedures, complex objects and rules which contribute significantly towards the feasibility of supporting statistical operations in a database.

The paper is organized as follows. In the next section, we briefly describe the case which motivated the study of implementing regression in a database. In Section 3, we discuss the features of extensible databases, which make them suitable for statistical support. In Section 4, we propose an implementation framework for record matching using regression. The problem of updating the regression coefficients is addressed in Section 5. The paper is concluded in Section 6 with a summary and directions for future research.

2 On-Line Customer Support: A Case Study

In this section, we present a case study about a California based software development company. The objective of the study was to develop a system that would enable the company to respond to customers' complaints on-line. A part of this application involved probabilistic record matching [4] using regression, which motivated us to study it in detail.

2.1 The Environment

In most software support centers, the major obstacle to providing effective customer support is the lack of an effective problem resolution system. These centers are typically characterized by low-head counts and high-workloads. Consequently, there is a high cost for re-inventing a solution than just re-using it, if it already existed. The objective in this case is to design a system that would enable a customer support staff to determine if a solution exists to the problem being reported and if so, identify it.

Under the present system, the set of solutions to the common problems are maintained in the database. When a customer reports a problem, it is recorded in a log file. Subsequently, the set of available solutions are checked, and the one that is considered most appropriate is conveyed to the customer, usually within one to two days.

In order to identify the appropriate solution, one would ideally match the problem description, that is, the log of a customer complaint, to the set of likely solutions. This is not possible in this situation because the records are matched using keywords and, the solution keywords often have very little in common with those of the problems they resolve. Consequently, one has to match the new problem reports to the previously existing ones. The basic idea being, that if there is a close match between a new and an existing log then it is likely that the solution to both problems is the same. If a solution is found to a previously reported problem, then it can be re-used. If the previously reported problem(s) do not have any solutions yet, similar problems are still linked together with the expectation that when one of these gets resolved, the solution will be adopted by others. Thus, the fundamental goal of a problem resolution system is to identify all previously existing logs that closely matches a new one.

2.2 The Schema

In order to explain the design issues, we present here a portion of the company's database using an Entity-Relationship diagram (see Figure 1). For brevity sake, we only present the portion of the database that is relevant to this case. The primary keys are in **bold**, whereas the foreign keys are **underlined**.

Customer complaints are recorded in the relation *Call Log* and are assigned unique identifiers, *Call Id*. The database schema is shown in Figure 1.
Other attributes of Call Log are Platform, Product, Priority, the Date the complaint was received, and the identifier of the customer lodging the complaint. The actual problem description is stored in a long text field. Detailed customer data is recorded in the relation Customer. After a complaint has been recorded, a keyword generation program is executed to extract the keywords from the textual description. The list of keywords for each record in the Call Log is stored in the Key Table. Thus, Keyword represents a set valued attribute.

The Solution table contains a list of solutions that has been suggested so far, where each of these have a unique identifier. Finally, the Linkage relation contains all the complaints (Call Log records) that have been resolved and their corresponding solutions.

### 2.3 The Methodology

As mentioned earlier, the objective of the system is to compare new call logs with the ones existing in the Call Log relation. If a close match is found, then the solution(s) to the matching log(s) can be proposed as the solution to the new problem. The criteria used in matching call logs depend on the customer service staff, but usually the most important attribute is Keyword. Occasionally, one would restrict the search further using other attributes, such as, Platform, Product or Priority.

The problem of document identification using keyword matching has been well investigated by information scientists. Two of the most common models in this regard are the Vector Space and the Probabilistic models. Numerous books and papers have compared the performance of these methods [13, 12]. For our application, we decided to adopt the probabilistic model due to its intuitive appeal and the fact that it is based on solid theoretical foundations. The use of the probabilistic model in record matching was described in detail in [4]. In the rest of this section, we illustrate how this approach can be combined with statistical regression to design a powerful information retrieval tool.

For notational simplicity, let us denote the relation containing the new call logs by \( \mathcal{R}(R) \) and, the one containing the previously logged calls, \( \mathcal{R}(S) \). Let the tuples of \( \mathcal{R}(R) \) be \( r_i, i = 1, \ldots, K \), that is, there are \( K \) new calls yet to be matched. Similarly, let the tuples of \( \mathcal{R}(S) \) be denoted by \( s_i, i = 1, \ldots, L \). Typically, \( K < L \). We want to join relations \( \mathcal{R}(R) \) and \( \mathcal{R}(S) \) to identify the tuples in \( \mathcal{R}(S) \) that best match the ones in \( \mathcal{R}(R) \). Let \( \mathcal{M} \) be a set containing the accurate result of this join, that is, the join result given perfect information.

Let a concatenated tuple be denoted by \( t_{ij} = (r_i, s_j) \). Our objective is to determine if \( t_{ij} \in \mathcal{M} \), \( \forall (i, j) \) combinations. In order to make this determination, we compare the attributes (including the keywords) of \( r_i \) and \( s_j \). Let \( \gamma_k(t_{ij}) \) represent the outcome of comparing attribute \( k \) in the two tuples. For single valued attributes, the \( \gamma_k \)'s are binary, where a ‘1’ denotes a match [3]. For a set valued attribute such as Keyword, the \( \gamma_k \) assumes a value in the range \([0, 1]\) representing the (normalized) number of keywords that match in the two tuples. If there are \( n \) matching criteria, we obtain the following comparison vector:

\[
\gamma(t_{ij}) = \{ \gamma_1(t_{ij}), \gamma_2(t_{ij}), \ldots, \gamma_n(t_{ij}) \}
\]

It should noted that the ultimate decision as to whether a pair of tuples match depends on the user. Thus, it is possible that a pair of tuples may be classified as matched while another pair, unmatched even though they may have the same \( \gamma \). So we want to evaluate the conditional probability:

\[
p_{ij} = \Pr \{ t_{ij} \in \mathcal{M} \mid \gamma(t_{ij}) \}
\]

One way to interpret the \( p \) value is as follows: if \( p_{ij} > p_{jk} \), then tuple \( r_i \) is more likely to match \( s_j \) than \( s_k \).

In order to evaluate this expression, one needs information about the conditional distribution. For most systems however, such distributions are non-standard and consequently, rather difficult to obtain. Secondly, a direct evaluation of probability often involves numerical integration, which is quite processing intensive. We present here a relatively simple yet practical approach to compute \( p \) using a regression model. We know by definition that \( p \) is a function of the observations, \( \gamma \). Consequently, it can be expressed in a functional form as follows:

\[
p_{ij} = \beta_0 + \beta_1 \gamma_1(t_{ij}) + \beta_2 \gamma_2(t_{ij}) + \cdots + \beta_n \gamma_n(t_{ij})
\]

The \( \beta \)'s can be estimated using the data available in the Call Log and Linkage relations. The intuitive approach calls for a linear model for this purpose. However, since \( p \)'s are probabilities, \( 0 \leq p \leq 1 \), a linear model does not guarantee that the predicted values will be in this range. In such situations, one has to fit a ‘S’ shaped curve, as obtained from a logistic regression, to the data [5]. Thus, the probability of matching is given by:

\[
p_{ij} = \frac{e^{\beta_0 + \beta_1 \gamma_1(t_{ij}) + \beta_2 \gamma_2(t_{ij}) + \cdots + \beta_n \gamma_n(t_{ij})}}{1 + e^{\beta_0 + \beta_1 \gamma_1(t_{ij}) + \beta_2 \gamma_2(t_{ij}) + \cdots + \beta_n \gamma_n(t_{ij})}}
\]

A plot of this expression for the two parameter (one variable) case is shown in Figure 2. The variable, \( \gamma_1 \),
Figure 2: The Matching Probability Function

which takes on values in the range [0, 1], represents the (normalized) number of keywords that match between two tuples. Notice that Equation (1) can be easily linearized by substituting

\[ \pi_{ij} = \log_e \left( \frac{p_{ij}}{1 - p_{ij}} \right) \]

which implies

\[ \pi_{ij} = \beta_0 + \beta_1 \gamma_1(t_{ij}) + \beta_2 \gamma_2(t_{ij}) + \ldots + \beta_n \gamma_n(t_{ij}) \]  

Once linearized, the coefficients of the logistic model can be obtained using an approach similar to the linear model. For additional details on regression or the application of the logistic model to information retrieval, the reader is referred to [5, 10].

Once the regression function has been determined, it can be used to match new calls against old ones. This would allow a staff member to suggest possible solutions to the customer at real-time, directly over the phone. In the next section we describe how this can be supported in an extensible DBMS.

3 Statistics in Extensible DBMS

In this section, we discuss the features of extensible DBMSs that make them attractive for statistical operations. We also illustrate how the support for such operations can often be developed in logically separate steps which greatly simplifies the implementation.

3.1 Features of Extensible DBMS

As illustrated in the previous section, regression requires complex mathematical operations to be supported in a database. The extensible DBMSs have certain features which render them particularly attractive for such applications:

- Complex data types. In order for the statistical data to be stored and efficiently manipulated in a database, we use statistical objects, which are a collection of numerous base types, such as, integer or float. Such objects are described in Section 4.

- Functions and Operators. A function enables one to store a computational procedure such as Equation (1), in the database instead of actual values. New operators are needed for adding new functionality, such as updating regression coefficients, to the DBMS.

- Access Methods and Rules. Statistical procedures and operators are frequently expensive and an extensible DBMS allows one to define rules and access methods for their efficient execution. These are illustrated in Section 4.

Several alternative DBMSs were considered for implementing the statistical procedures, namely, ODE [1], Starburst [6], EXODUS [2] and POSTGRES [19]. Each of these were evaluated based on the above criteria, and availability. Though quite a few systems supported several of the above features, only POSTGRES and EXODUS allowed us to modify the source code and integrate the missing elements. Between the two, we chose the former due to the local availability of source code support. General familiarity with the POSTGRES model also contributed to the decision. Consequently, the implementation discussed in the rest of the paper is with respect to POSTGRES though the ideas can be exported to other systems.

3.2 Stepwise Support of Regression

Regression has a functional representation that simplifies its implementation in a database. In order to use regression, one has to first determine the appropriate coefficients (the \( \beta \)'s). Once obtained, they can be used to derive new results (e.g., whether or not a record matching a new call exists in the database). We take advantage of this separability in designing our system. First we assume that the necessary coefficients are available and discuss (in the next section) how the regression function can be supported for matching records. We then analyze the problem of deriving and updating the coefficients in Section 5.
4 Record Matching Using Regression

In this section we discuss how the regression function can be used for processing new records, assuming that the necessary coefficients are available. In order to support regression in a database, three issues need to be addressed: (i) storing the coefficients, (ii) activating the regression function, and (iii) devising access methods for efficient record matching. We discuss these below.

4.1 Storing Coefficients as Objects

A regression function, such as Equation (2), typically has a number of coefficients associated with it, depending on the number of the independent variables and the interactions between them. The decision to use a set of parameters, or for that matter one form of the regression function, depends on the application and the user. During this study, we discovered that different customer service staff preferred using different functions depending on their personal experience, available time and the success rates of the functions in identifying the proper matches. The following regression function was illustrated in Figure 2:

\[ \pi_{ij} = \beta_0 + \beta_1 \gamma_1(t_{ij}) \]

where \( \gamma_1(t_{ij}) \) represents the (normalized) number of keywords that match between records \( r_i \) and \( s_j \). A more comprehensive form of the function would be the following:

\[ \pi_{ij} = \beta_0 + \beta_1 \gamma_1(t_{ij}) + \beta_2 \gamma_2(t_{ij}) + \beta_3 \gamma_3(t_{ij}) \]

where \( \gamma_1(t_{ij}) \) has the same interpretation as before; \( \gamma_2(t_{ij}) \) equals 1 if the \( Platform \) attribute matches in the two tuples, zero otherwise; and \( \gamma_3(t_{ij}) \) is defined in the same way for the \( Product \) attribute.

Our objective is to store the different functions (and their coefficients) on-line, so that they can be called by the users at any time. Our primary choice was to store these in a relation. However, one would use:

\[ \text{append C VALUES (Cid = 101, Value = 2.314, description = "Product")} \]

Next, we create a second relation \( PARAMETER \) to store the different functions (and their coefficients) on-line, so that they can be called by the users at any time. Our primary choice was to store these in a relation. However, one would use:

\[ \text{create PARAMETER (Pid = int4, Validity = abstime, Updated = abstime, Values = proc[])} \]

Here, \( Pid \) is the identifier associated with the current list of parameters. \( Validity \) and \( Updated \) represent the actual times at which these coefficients are valid and updated respectively. These fields allow versioning in a sense that they let the user define and use parameters which were computed over specific time ranges, such as last week. In addition, \( Updated \) is useful for updating the coefficients, as described in Section 5.

The \( Values \) field is actually an array of procedures. Currently, \( POSTGRES \) only supports arrays of basic types. Here, we extend the concept to include procedure data types as well. These procedures would point to the appropriate data in the \( C \_VALUES \) relation. To append data to \( PARAMETER \), one would use:

\[ \text{append PARAMETER (Pid = 12, Validity = 01/01/94, Updated = "now", Values = "(get(101), get(25), get(79))")} \]

where, \( get() \) is a function (defined below), that does the appropriate table look up:

\[ \text{define function get (language = "postquel", returntype = float) arg is (int4) as retrieve (C \_VALUES.Value) where C \_VALUES.Cid = $1)} \]

Here, \( $1 \) refers to the function argument and \( now \) incorporates the current date in the corresponding field. Finally, retrievals to the array data type can be handled effectively using rules of the form [18]:

\[ \text{define rule rule_1 on retrieval to PARAMETER.Values[1] where PARAMETER.Pid = 12 then do instead execute get(101)} \]

These rules are activated by \( Event \) locks placed on the \( Values \) field. Event locks are tagged with the type of
event, in this case a retrieval, which will trigger an action, along with the identifier of the rule. When a retrieval is attempted, the qualification in the appropriate rule is checked and if true, the action is executed. Next we show how these parameters can be used in the regression function.

4.2 Matching Records using Functions

Once the coefficients are available in the database, the regression function can be implemented as a Postgres function which uses the coefficients as arguments. Due to the complexity, the regression functions were coded in the general purpose programming language, C. Let a generic regression function \texttt{match_fn()} be defined as:

\begin{verbatim}
define function match_fn
  (language = "c", returntype = float)
  arg is (data type, ..., data type)
  as "/usr/postgres/match_fn.0"
\end{verbatim}

The function \texttt{match_fn.c} can be defined in various different ways. Here we assume that the \( \gamma \) values are computed outside the function and passed on to it as arguments, along with the identifier of the parameter set. The function uses some embedded SQL-type statements to retrieve the actual parameter values, and computes the match probability as defined in Equation (1). The value returned by the function represents this probability.

It should be noted that the number and the data type of the arguments varies with different functions. Let us consider a specific application where all the new calls are stored in the relation \texttt{NCALL}, and all previously matched calls, in \texttt{OCALL}. Then a match query can be stated (in SQL) as:

\begin{verbatim}
Select N.Call.Id, O.Call.Id, \gamma_1 = kcount(N.Call.Id, O.Call.Id), p = match_fn(\gamma_1, 12)
From NCALL N, OCALL O, PARAMETER
Where N.Product = O.Product
And p > 0.65
And PARAMETER Update > 1/10/94;
\end{verbatim}

The objective of this query is to compare the tuples which have a common entry in the \texttt{Product} field, and retrieve only those which have a matching probability \( p \) greater than 0.65. The probability is to be computed using (i) the \( \gamma_1 \) variable, and (ii) the parameter set \( \text{Pid} = 12 \) that was computed on or after January 10, 1994.

Finally, \texttt{kcount()} function is defined for this case study to compute \( \gamma_1 \), the (normalized) number of matching keywords between the two records. Since the keywords are precomputed and stored in the Key Table, the function can be defined as follows, where \$i$ refers to the \( i^{th} \) function argument, and \( m \), the normalizing constant:

\begin{verbatim}
define function kcount (language = "sql", returntype = int) arg is (int, int) as select count (+)/m from Key Table N, Key Table O
  where N.Call.Id = S1 and O.Call.Id = S2
  and N.keyword = O.keyword;
\end{verbatim}

4.3 Access Methods and Caching

Response time is a key measure for an effective customer support system. In this section we propose some indexing and caching schemes that could improve system performance when comparing customers’ complaints.

4.3.1 Access Methods

The coefficients in \texttt{C.VALUES} are identified by \texttt{Cid} and the parameter set in \texttt{PARAMETER}, by \texttt{Pid}. Since these are retrieved each time a record matching query is executed, we propose a \( B^+ \) tree index for efficient retrieval on both these fields. Many DBMSs including Postgres allow users to define indices on most frequently accessed fields.

The main bottleneck to efficient retrieval could be the \texttt{match_fn()} itself. The probability computation, as illustrated by Equation (1) involves time consuming operations such as, exponentiation and floating point divisions. These computations are likely to be repeated however, as the inputs to the functions vary over a limited range. Thus, it is possible to obtain better response times by materializing the functions' output for various inputs. This approach, which we refer to as \textit{function indexing} is particularly useful when a function has discrete inputs, as is the case here.

Let us consider a function \texttt{match_fn()} with a single argument, \( \gamma_1 \). For such a function, a \( B^+ \) tree index can be constructed in the following way:

- All non-leaf nodes including the root node will contain the values of \( \gamma_1 \) as if the index is on the possible values of \( \gamma_1 \).
- All leaf nodes will contain (or point to) \texttt{match_fn(\gamma_1)}.

An example of such an index for a generic function \( f(\bullet) \) is shown in Figure 3. If such an index is available, it would no longer be necessary to evaluate \texttt{match_fn(\). Once the arguments to the function are available, the DBMS would search
down to the proper leaf node and retrieve the tuple probability. This would be considerably cheaper than the floating point arithmetic, specially if the index is small enough to be stored in the main memory.

It should be noted that the materialized values become invalid and need to be refreshed when the function coefficients are updated. This is similar to restructuring an ordinary index. However, in this case the tree does not grow (i.e., no node splitting occurs). Only the materialized values in the leaf node needs to be recomputed, which can be done on demand, incrementally and during off-peak periods. Performance benefits from function indexing is currently being further investigated.

4.3.2 Caching

The performance of a DBMS can be dramatically improved by caching frequently used objects so that they do not have to be repeatedly rematerialized [17]. The PARAMETER table contains an array of procedures. Instead of executing them each time a set of parameters are retrieved, the actual values can be cached and stored in those fields. This has been referred to in the literature as Cache in Tuple (CT) strategy. Alternately, one can store the cached results separately [8]. In either situation, one avoids executing the procedures in PARAMETER. We prefer the CT strategy due to the small size and high access rate of the tuples in PARAMETER.

We use Invalidation Locks (in short, I-locks) to indicate the validity of a cached procedure. These locks stay in place until the coefficients are updated in C.VALUES. Once the coefficients change, the I-locks are broken, indicating that the procedures need to be recomputed next time the corresponding parameter set is retrieved. We are currently studying the improvements in the system performance resulting from this approach.

5 Updating Regression Coefficients

In this section we discuss how the coefficients for the regression function can be derived and updated. We first assume that the data is available in a format appropriate for the statistical operations. Later we identify some simple steps that were used for formatting the data for this case study.

5.1 Deriving the Coefficients

In order to derive the coefficients we adopt an approach similar to the one used in the previous section, that is, using procedures. The input to the procedure is the data set. The procedure does not produce any explicit output, it simply writes (or updates) the coefficients in the appropriate relation. The data set used for deriving the coefficients are stored in a two dimensional matrix. The regression equation can be represented in matrix notation as:

$$\Pi = \Gamma \beta,$$

where $\beta$'s are the unknown coefficients. A typical data set would look like the following:

- $\pi_1 = \beta_0 + \beta_1 \gamma_{1,1} + \cdots + \beta_{n-1} \gamma_{1,n-1}$
- $\pi_2 = \beta_0 + \beta_1 \gamma_{2,1} + \cdots + \beta_{n-1} \gamma_{2,n-1}$
- $\vdots$
- $\pi_m = \beta_0 + \beta_1 \gamma_{m,1} + \cdots + \beta_{n-1} \gamma_{m,n-1}$

where all $\gamma$'s and $\pi$'s are known. In a matrix form, these would be represented as:

$$\Pi = \begin{bmatrix} \pi_1 \\ \vdots \\ \pi_m \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 \\ \vdots \\ \beta_{n-1} \end{bmatrix}$$

$$\Gamma = \begin{bmatrix} 1 & \gamma_{1,1} & \cdots & \gamma_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \gamma_{m,1} & \cdots & \gamma_{m,n-1} \end{bmatrix}$$

In order to store these arrays, their structure first needs to be defined. POSTGRES provides built-in functions for easy creation of arrays. If an array and a two dimensional matrix data type be denoted as intarray and int4matrix respectively, we can define a relation to store the input data:
create DATA (Pid = int4, \( II = \text{int2array}, \; \Gamma = \text{int4matrix} \))

Here, the \( Pid \) field, which contains the identifier of the data set, is also the identifier of the coefficient set that would be generated and stored in the \textbf{PARAMETER} relation (defined in Section 4). Note that a new data set is used each time the coefficients are updated.

Once the data set is stored in \textit{DATA}, it can be passed to a procedure that would compute the coefficients and append (or update) them in \textbf{PARAMETER}. Such a procedure can be defined as:

\[
\text{define function } \text{regress} \\
\text{(language = "c", returntype = bool)} \\
\text{arg is (DATA) as "} /usr/postgres/regress.o" \\
\text{and executed as:} \\
\text{execute regress (DATA)} \\
\text{where DATA.Pid = 12.}
\]

Here we assumed \textit{regress.c} to be a C function. Its detailed discussion is beyond the scope of this paper, however it is worthwhile to point out that this results in an efficient modular design. Obtaining the coefficients involve solving for \( \beta \) in the equation: 
\[
\beta = (\Gamma^T \Gamma)^{-1} \Gamma^T \Pi 
\]
Consequently, the procedure can incorporate the algorithms that have been proposed in the literature for efficient matrix manipulation [9].

The procedure can even be used to call up a statistical package, if such an interface exists. We intend to analyze the performance of the system using different designs as a part of our future research.

It should be noted that we do not consider the issues of numerical or statistical stability in this paper. Sophisticated and stable procedures have been reported in statistics literature for treating newly arrived data to update prior distributions. We chose to describe a simple approach here because it is easier to understand and yet highlights the general steps required to implement regression in a database.

5.2 Incremental Updates

The coefficients in the regression function need to be periodically updated. However, as more complaints get logged, the size of the data set increases. The update process requires manipulation of a \( m \times n \) matrix, a step which becomes increasingly expensive with the size of \( m \). By sampling, one can select (and reduce) the number of data points to be processed. Still this is equivalent to re-deriving the coefficients with a new set of input, which is as expensive as the initial derivation. The problem with this approach is that it does not take advantage of the earlier computations and the fact that only the effect of the complaints logged since the previous update needs to be incorporated in the computation.

For such situations, we propose a scheme for incrementally updating the coefficients. By storing some of the intermediate results, one can reduce the effort in updating the coefficients drastically. Moreover, these intermediate results can serve as inputs to other statistical computations, such as, confidence interval estimates. It has been illustrated that such sharing improves the response time for all statistical operations [15].

In case of regression, we take advantage of the special properties of the \( \Gamma^T \Gamma \) and the \( \Gamma^T \Pi \) matrices:

1. Size: Even though the initial data set may be large, \( m \gg n \), the size of the \( \Gamma^T \Gamma \) matrix is merely \( n \times n \), and that of \( \Gamma^T \Pi \) vector, \( n \times 1 \). Typical value for \( n \) is unlikely to exceed 4 or 5.

2. Symmetry: The \( \Gamma^T \Gamma \) matrix is symmetric requiring one to store only \( \frac{n(n+1)}{2} \) elements instead of \( n^2 \).

In order to incrementally update the coefficients, we propose to materialize these two matrices. During the initial computation, the elements of these matrices would be stored in a relation, with the corresponding identifier, \( Pid \). This can be easily achieved using a matrix data type as discussed earlier in the section. While recomputing the coefficients, the individual elements of these matrices are updated using the relationship:

\[
\Gamma^T \Gamma := \Gamma^T \Gamma + A^T A - D^T D
\]

where, ':=' indicates assignment and, \( A \) and \( D \) denote matrices corresponding to the data appended and deleted respectively, since the last update. (Note that any update can be modeled as a deletion followed by an insertion.) Of course, the \textit{regress} function has to be modified to accept the intermediate results from the previous computation along with the new data set.

So far we have assumed that one can identify the data that was modified since the last update of a set of coefficients. In general, this could be a tricky problem since there are multiple sets of coefficients that may be incrementally updated by different users at different points in time. However, in this case study, the problem is solved in a rather simple way. We use the date attributes in the \textit{Call Log} and \textbf{PARAMETER} to select the tuples which have been appended to the former relation since the last update of the latter. In general, if the date attributes are not available,
one may need more sophisticated algorithms involving
time stamps, as discussed in temporal database liter-

5.3 Generating the Data Set

Up to this point in the section, we had conveniently
assumed that the input data sets for estimating the
regression coefficients are available in a readily usable
format. While this may be true in some situations,
more often than not these data sets need to be care-
fully constructed. We describe here the steps that
were undertaken in generating the data sets for this
case study.

Using the information in the Linkage relation, one
can identify all the matching record pairs and derive
d their $\gamma$ values, by applying the simple rule that two
records (complaints) are similar if they use the same
solution. This however may not generate sufficient
number of data points. In order to obtain data points
at each level (or value) of $\gamma$, the following two sam-
pling techniques [11] are being considered:

1. Sample first: This is a simple approach where two
   sets of records are first randomly selected (with-
   out replacement) from Call Log. The tuples in
   these two data sets are then compared and their
   $\gamma$ values computed. This process is continued
   until sufficient number of data points have been
   obtained for all levels of $\gamma$.

2. Query first: In this approach, selection queries are
   used to identify potentially close tuples, such as,
tuples having the same Product or Platform char-
acteristics. These are then compared, and their
   corresponding $\gamma$ values computed. By carefully
posing the initial queries, this approach can be
used to obtain data points for all levels of $\gamma$. In
   a sense, this is equivalent to stratified sampling,
where each level of $\gamma$ can be considered as a stra-

In general, the sample first approach is inefficient
as it tends to generate large number of data points
around the mean value of $\gamma$. The query first approach
is comparatively efficient and can take advantage of
available indices during sampling. However in situa-
tions as in this case study, it may not be very effective
since the similarity between tuples is measured using a
function such as $\text{count}()$ which requires both tuples
as its argument. We are currently investigating the
performance of these and other sampling approaches
in generating the regression data set.

6 Conclusions And Future Re-
search

Recently, the data management issues associ-
ated with accessing and processing large statistical
databases have become an active research area. Tra-
ditionally, the data management needs of a statisti-
cal database user have been met by the restricted
data management capabilities of statistical packages
[9]. With the recent developments in space research,
large quantities of scientific data is being collected and
it is no longer efficient to simply delegate the data
analysis step to a statistical package. In the business
environment, statistics have long been a part of deci-
sion support applications. With the advancement in
the database technology and the development of new
applications, organizations are now interested in ap-
plying statistical tools at real-time as a part of regular
transactions.

This paper introduced a framework for processing
statistical queries in extensible databases. This re-
search was motivated by a case study where the Cus-
tomer Support Department in a software development
company desired a system that would enable them to
respond to customers' complaints at real-time. Such
a system required a probabilistic record matching ca-
pability using regression.

We illustrated in this paper how such a system can
be developed using an extensible DBMS. In particular,
we focused on supporting regression in a database. Re-
gression can be implemented in two logical stages: (i)
determination of the regression function (or coefficient
estimation), and (ii) using the function for predicting
new values. We illustrated how these could be sup-
ported in a database using procedures, functions and
rules. We also discussed access methods, caching and
sampling techniques which would reduce the response
time for statistical operations.

We are looking at several areas for further research.
In this paper, we presented numerous alternative im-
plementation plans for materialization, access meth-
ods, and sampling. As a part of our future research, we
intend to analyze the performance of these plans and
determine the most efficient implementation. Also, we
had restricted ourselves to regression in this paper.
We want to extend this model to a more general case
involving other statistical operations. Particularly, we
are interested in the cost savings that can be achieved
in a multi-user environment by efficient sharing of the
intermediate results. We are also interested in devis-
ing efficient algorithms for incrementally updating the
regression coefficients.
Acknowledgement

The authors would like to thank Sukumar Rathnam of Scopus Technology, Inc. for providing us with the case study described in Section 2.

References


