Abstract
In this paper, we present a novel dynamic scheme to estimate the execution time of stored procedures inside the Database Management Systems (DBMS). Our estimation model can provide users an accurate estimation of the execution time of stored procedures. The proposed estimation model adopts the basic idea of Radial Basis Function (RBF) network to accurately predict the execution time of the stored procedures based on their input parameters. Our model is self-trained and does not need a separate training set. Moreover, it can automatically adjust itself to adapt to the changes of the database. Furthermore, our proposed model can be implemented by SQL to embed directly in the database which is an important advantage over previous systems. Extensive experimental results show that the proposed approach can accurately predict the execution time of stored procedures with error rate below 10% after getting executed for as least as 20 times. Moreover, the new model can effectively adjust itself to dramatic database updates.

Keywords: Stored procedure, Execution Time, Estimation, SQL Server, Neural Network, RBF Network

1. Introduction

Stored procedures are precompiled database subroutines called by applications. The major benefits of using stored procedures include the substantial performance gains from the precompiled execution, the reduction of database access, development efficiency gains from code reuse and abstraction, and the security controls inherent in granting users permissions on specific stored procedures instead of the underlying database tables [13].

Despite their enormous benefits, there is one major problem in using stored procedures. When a stored procedure is called, the calling application does not have any information on how the execution of the stored procedure is progressing. This is especially important for web applications to inform the user with a realistic estimation. A complex stored procedure may take minutes to execute. Without giving any clue of the execution time, the user may get very frustrated. In addition, it is not efficient in terms of user’s time management. However, most of the commercial DBMSs do not provide the progress information of the running stored procedure.

Estimating the execution time of a stored procedure is challenging. The common method to estimate the execution time of a software program is to examine the complexity of its algorithm. In the case of a stored procedure, however, the complexity analysis is quite different, because the SQL commands constituting the stored procedures is quite different from other commands. Moreover, this estimation needs to consider more factors such as the nature of the input and output variables, the properties of the data in the database, as well as the hardware/operating system capabilities.

There have been several attempts [1, 2, 3, 5, 6] (as detailed in Related Work) to estimate the execution time of stored procedures. These approaches can provide reasonably accurate estimation. However, these approaches suffer from some problems: a) their accuracy decreases when the number of the parameters of the stored procedures changes; b) the estimation software needs to be separated from the DBMS; c) training session has to be performed before real estimation; d) they need longer training time.

To solve the aforementioned problems, we propose an effective estimation model. The proposed model utilizes a Radial Basis Function (RBF) neural network to estimate the execution time of a stored procedure based on input parameters. The estimation model is very simple yet quite efficient. It can intelligently train itself and dynamically adjust its weights to fit the changing data in the database. Our experiments show that this method can quickly train itself to reach an error rate of less than 10%. Most importantly, our estimation model can be implemented with SQL-based stored procedure and be embedded into any DBMSs. Therefore, if an application developer would like to include a progress bar in the application, there is no need to install another software component.

The rest of this paper is organized as follows. Section 2 introduces the related work. Section 3 discusses the problem in more details. Section 4 presents our estimation approach and its implementation and deployment in SQL Server.
section 5, we evaluate the proposed model and analyze the results. Concluding remarks are provided in section 6.

2. Related Work

There have been many approaches proposed for estimating the execution time of a stored procedure (SP). Generally, we divide them into three categories: 1) Cost Model-based, 2) Neural Network-based, and 3) Histogram-based.

The first category is the cost model approach. This approach estimates the execution time of a SP by evaluating the complexity of the relational algebra operations of the SP. Several research works [4, 5, 6] have applied cost model to estimate the cost of querying different DBMSs. In these works, a calibration process has been applied for their predefined analytical model. Then, a benchmark is performed on the system to measure its performance. Later, a mathematical model is constructed to estimate the complexity. This process produces a calibrated cost model for a specific platform. However, as specified in [1] “cost models are still a problematic issue in database research because of their imprecision and the continuous introduction of new data types and processing in extended database system.” Cost models cannot adjust themselves with the database changes because once the model is formed, there is no way to update and change the model.

The second category employs the neural network technologies to estimate the SP execution time. In [1], a curve-fitting mechanism was applied by a neural network to perform the estimation. During the training process, the neural network is fed with input data as well as the corresponding outputs and the actual execution time. After training, the trained network can estimate the execution cost of new data entries. A major limitation of this approach is its complexity, which makes it impractical to be applied in DBMSs.

The third category, the histogram-based approach [2], uses a multi-dimensional histogram to estimate the cost of a specific SP. In these approaches, a multi-dimensional histogram is constructed to estimate the cost of a SP. First, the user-defined method is executed multiple times with different input arguments. Then, these input arguments together with the corresponding execution times are collected. For each group of collected data, the histogram can be formed by changing the values of each argument between its upper and lower limits. The main limitation of this approach is its manual calibration. In other words, determining appropriate input arguments ranges requires human intervention. A human agent should assist the system by providing the upper- and lower-bound of each of the input arguments. In addition, the histogram cannot adapt to a dynamic database, and this approach cannot estimate the execution time of the SP when the database is changed.

Our approach has several advantages over the aforementioned approaches. First, it is simple and it can be implemented inside the DBMS easily. Second, our proposed method does not need initial training session. Indeed, it trains itself as it operates on new data, and it can provide a very accurate estimation of the time using only few training data. Third, our intelligent algorithm is capable of recognizing the changes occurring in the database and adapts itself to the new situation.

3. Problem Statement

The execution time of a stored procedure depends on many factors that are listed below:

1. Complexity of its command structure
2. Input parameters
3. The amount and the structure of data inside the database
4. The hardware and processor specification which the DBMS is running on.

Our proposed method can accurately estimate the execution time of stored procedures with all of the aforementioned factors in consideration. In particular, we focus on stored procedures which their execution time change based on input parameters values.

It is well known that the values of the input parameters have a significant impact on the SP performance. However, how the input parameters affect the execution time (i.e., their relationship definition) is very complex. In such cases, neural network can be a good option to imitate the complex function. Each time when a stored procedure is executed, the neural network is trained using the real execution time and input parameters. Since there are many different types of stored procedures inside a database and each of them needs its own neural network (based on its own parameters), the proposed system therefore, should support training and storing multiple neural networks.

Another question to ask is “which neural network is suitable for this estimation of SP execution time problem”? The selection of the network depends on many factors such as training time, number of data for training, size and complexity of the network, etc. We examined many different neural networks, and eventually selected the General Regression Neural Network (GRNN) as our implementation. GRNN was proposed by Donald Specht [14]. It is based on nonlinear kernel regression in statistics. GRNN can be viewed as a normalized RBF (Radial Basis Function)
network in which there is a hidden unit centered at every training case. These RBF units are called "kernels" and are usually probability density functions such as the Gaussian. According to [8] the hidden-to-output weights are just the target values; therefore the output is simply a weighted average of the target values of training cases close to the given input case.

Specht later proposed a method to formulate the weighted-neighbor method described above in the form of a neural network. He called the network a “Probabilistic Neural Network”. Figure 1 shows a diagram of a PNN/GRNN network.

![GRNN Architecture](image)

As shown in Figure 1, the PNN/GRNN networks normally include four layers [9]:

1. **Input layer** — there is one neuron in the input layer for each predictor variable.
2. **Hidden layer** — this layer has one neuron for each case in the training data set. When presented with the \( x \) vector of input values from the input layer, a hidden neuron computes the Euclidean distance of the test case from the neuron’s center point and then applies the RBF kernel function using the sigma value(s).
3. **Pattern layer / Summation layer** — there are only two neurons in the pattern layer. One neuron is the denominator summation unit the other is the numerator summation unit. The denominator summation unit adds up the weight values coming from each of the hidden neurons. The numerator summation unit adds up the weight values multiplied by the actual target value for each hidden neuron.
4. **Decision layer** — the decision layer divides the value accumulated in the numerator summation unit by the value in the denominator summation unit and uses the result as the predicted target value.

### 4. Methodology

As mentioned, GRNNs can be modeled by RBF Networks. In our system, RBF networks are implemented as stored procedures. As shown in Figure 2, the training part, called “Train” in the figure, is implemented by stored procedures. To train the network, each time a stored procedure is executed inside the DBMS, we record the input values and the corresponding execution time. In order to estimate the execution time, we feed the network with the input values and the execution time. Anytime, if the new inputs show significant changes to the network, then they will be added to the training set. A table is used to store the data used for training. In the current implementation, we assume that the maximum number of parameters a stored procedure could have is 10, but it can be easily changed without affecting the algorithm.

![The training process](image)
time of it. Then, it updates the training data table when it is necessarily.

The training procedure is performed as follows: first, it calls a stored procedure which calculates the estimated execution time based on current training set (line 9). The procedure estimating the time will be explained in detail later in this section. Next, it compares the estimated execution time with the real execution time (10). If the difference is more than the error rate, the procedure then further compares the vector of the input values with the two ends of the ordered training input vectors. If the difference is lower than a threshold, the procedure will add the current vector to the training set. Otherwise, it will set the current input vector ad the new lower end of the training vectors. The difference threshold controls the similarity of the training vectors. In our current implementation, it is set to 1%. The change of the threshold would affect the number of training records. As the threshold value decreases, the number of training records increases. Also, the accuracy of our estimation would increase. On the other hand, the training set update may take longer time as the training dataset size increases. Through our experiments, we found 1% can give us a good result after many trial and errors. The basic principle of setting the threshold value is to make a good tradeoff between the accuracy and the training speed. In our experiments, we set the error rate to one second, which means that if the difference between the estimated execution time and real execution time is beyond one second, then we will add the vector to the training set. The training data table is the hidden layer of our GRNN network, which contains different samples and their values control how many records we will have in this table for each stored procedure. The algorithm that estimates the execution time of the stored procedure is complex. It implements the RBF network functionality. Procedure 2 shows the process of estimating the execution time. It takes the stored procedure name and the input parameters as its input. At the beginning, it calls a procedure “CalDistance” which gets the input parameter vector and returns ten vectors stored in the training data table which has the closest distance to the input vector. Then it uses the Gaussian RBF [11] to calculate the weight:

\[
d_i = \frac{\sqrt{\sum_{j=1}^{10} (x_j - v_j)^2}}{\text{Max Distance}}
\]  

(Eq.2)

\[W_i = k \cdot \exp \left( - \frac{d_i^2}{\text{Max Distance}^2} \right) \]  

(Eq.3)

In which \(c\) represents the center and \(r\) is the radius. Figure 3 illustrates a Gaussian RBF with center \(c = 0\) and radius \(r = 1\). As shown in the figure, the values which are closer to the center get higher weight and as their distance to the center increases the weight decreases.

We use mean square to calculate distances between each of the vectors in the training set and the current input vector as shown in Equation 2. We will repeat this process for all training vectors within the database and then choose the top 10 closest vectors.

Figure 3: Gaussian RBF Function

To calculate the weight of each vector, we need to first find the maximum distance between the input vector and the selected vectors from the training set. We use the Gaussian-based RBF Function (as shown in Equation 3) to compute the weight:

\[W_i = k \cdot \exp \left( - \frac{d_i^2}{\text{Max Distance}^2} \right) \]  

(Eq.3)

\[d_i: \text{Vector i distance from input vector} \]
\[W_i: \text{Weight of vector i} \]
\[k: 1.2 \]

Equation 3 calculates the weight for each vector. The result is a weight between 0 and 1. As the vector gets closer to the input vector, its weight gets higher. We multiply the result by 1.2 (i.e., increasing the result by 20%) so that the result can be either higher than the real value or lower than the real value. Otherwise, the estimated time would always be lower than the real time. In other words, this coefficient causes the output of Gaussian function to be a number between 0 and 1.2. As a result, we can focus on the vectors in between. We set the value of this coefficient based on our experiments to achieve an error rate less than 10%. This 20% increase normalizes the output of RBF network.

\[h(x) = \exp \left( - \frac{(x - c)^2}{r^2} \right) \]  

(Eq.1)
CREATE PROCEDURE [dbo].[Train]
@SPName nvarchar(250), @P1 int,@P2 int,@P3 int,@P4 int,@P5 int,@P6 int,@P7 int,@P8 int,@P9 int,@P10 int, @RealTime float AS

BEGIN
declare @Est float
declare @ErrorRate float
declare @MaxD float
declare @MinD float
set @ErrorRate=1
EXEC [dbo].[EstimateExeTime] @SPName,@P1,@P2,@P3,@P4,@P5,@P6,@P7,@P8,@P9,@P10,@Est OUTPUT
if (abs(@RealTime-@Est)>@ErrorRate)
begin
CREATE TABLE #TempT (TED_ID int, Distance float, TED_ExecutionTime float, Weight float, EstimateT float)
insert #TempT EXEC [dbo].[CalDistance] @SP_Name = @SPName, @Param1 = @P1, @Param2 = @P2,  
@Param3 = @P3, @Param4 = @P4, @Param5 = @P5, @Param6 = @P6, @Param7 = @P7, 
@Param8 = @P8, @Param9 = @P9, @Param10 = @P10, @VectorCounts = 9999999
Select @MaxD = Max(Distance), @MinD = Min(Distance) from #TempT
set @MaxD=isnull(@MaxD,99999)
set @MinD=isnull(@MinD,99999)
if (((@MinD/@MaxD)>0.01) or (@MinD=@MaxD))
begin
insert _TimeEstimateData(TED_StoredProcedure,TED_ExecutionTime,TED_Param1, TED_Param2, 
TED_Param3, TED_Param4, TED_Param5, TED_Param6, TED_Param7, TED_Param8, TED_Param9, TED_Param10) values (@SPName,@RealTime,@P1,@P2,@P3,@P4,@P5, @P6,@P7,@P8,@P9)
end
else
begin
declare @ID int
select top(1) @ID=TED_ID from #TempT
declare @sql nvarchar(100)
set @sql='update _TimeEstimateData set TED_ExecutionTime='+ltrim(str(@RealTime ))+
'where TED_ID='+ltrim(str(@ID))
exec (@sql)
end
end

END

Procedure 1. “Train” stored procedure

CREATE PROCEDURE [dbo].[Train]
@SPName nvarchar(250), @P1 int,

BEGIN

DECLARE @return_value int
CREATE TABLE #TempT(TED_ID int, Distance float, TED_ExecutionTime float, Weight float, EstimateT float)
insert #TempT EXEC [dbo].[CalDistance] @SP_Name = @SPName, @Param1 = @P1, @Param2 = @P2,  
@Param3 = @P3, @Param4 = @P4, @Param5 = @P5, @Param6 = @P6, @Param7 = @P7, 
@Param8 = @P8, @Param9 = @P9, @Param10 = @P10, @VectorCounts = 10
declare @SumD float,
declare @SumW float
declare @MaxD float
declare @MinD float
declare @CountD float
select @SumD=sum(Distance),@MaxD=Max(Distance),@MinD=Max(Distance),@CountD=count(Distance) from #TempT
declare @Decrease float
set @Decrease=0
update #TempT set Distance=Distance-@Decrease
update #TempT set Weight=exp(-1*(square(Distance)/square(@MaxD-@Decrease)))*1.2
update #TempT set EstimateT=Weight*TED_ExecutionTime
declare @EstimateTime float
declare @SumTime float
select @EstimateTime=sum(EstimateT),@SumTime=sum(TED_ExecutionTime) from #TempT
drop table #TempT
select @EstimateTime=sum(EstimateT),@SumTime=sum(TED_ExecutionTime) from #TempT
END

Procedure 2. Estimation of the execution time of stored procedures
After getting the weights for the vectors, these weights and their respective estimated execution time can be used to compute the final estimated time as shown in Equation 4 and 5.

\[ T_{Est} = \sum_{i=1}^{CD} (w_i \times T_i) \]  
**Equation 4**

\[ CD: \text{Count of selected vectors (which is 10 in our current implementation)} \]

\[ T_{Est}: \text{Total Estimation} \]

\[ T_i: \text{Execution time of each vector} \]

Equation 5 normalizes the result of Equation 4 based on the execution time of each vector and returns the estimation.

\[ \text{Estimated Time} = \frac{T_{Est}}{\sum_{i=1}^{CD} T_i} \times CD \]  
**Equation 5**

Procedure 3, the “CalDistance” stored procedure, is responsible to choose the top 10 nearest vectors. Procedure 4 calculates the distance between two vectors. This procedure finds the mean square as described earlier. All of these procedures presented here are designed for supporting 10 input parameters but they can be changed based on the requirements of the system. Currently, we support numeric input only. Therefore, when we get other types of input, we need first convert the input to numeric.

---

5. Experimental Evaluation

To evaluate the proposed mechanism, we have used both an artificially generated dataset and a real large-scale dataset received from a telecommunication company. The generated database contains a list of products with corresponding warehouse locations and their prices. In particular, it contains 20,000,000 products with 10 different types, 4 different locations, and prices ranging from 1 to 100,000. All the values are randomly set to simulate a large distribution of values, and to increase the range of the time of executing stored procedures. The real dataset is a database of a time attendance system of a very large company with over 10,000 employees. This system is responsible to calculate salary of employees based on their time presents.

---

Procedure 4. getDistance function

We have designed four different stored procedures to evaluate our proposed mechanism. The first stored procedure calculates the average and the standard deviation of the price of each product at specified stock location with a minimum price value. The execution time will vary based on the stock location which is chosen and the minimum price value. Procedure 5 is the first testing stored procedure called “GetProductStatus”. The second stored procedure is almost the same as the first one, but has an extra parameter which also specifies the maximum price value. The third stored procedure is a complex statistical stored procedure. Its inputs contain a stock location and two price ranges; therefore, it totally has five parameters. The fourth stored procedure is performed on the real dataset. It is responsible to calculate the amount of unused leaves of the employees. This stored procedure has two input parameters which specify the range of departments and it calculates unused leave time of all personals in those departments. The execution time depends on many factors, such as how many personals are in selected departments and historical information for each of the personals.

On a desktop computer with 4 GB RAM, 2.26 GHz CPU, Windows 7 OS, it takes 14 to 25 seconds to execute the first procedure based on different input parameters, 14 to 28 seconds for the second stored procedure, while it takes 17 to 52 seconds to execute the third stored procedure. Finally it takes between 1 to 38 seconds to execute the last stored procedure. Table 1 shows the execution of the four different stored procedures. In this table procedures are located on each
column and we have compared the Minimum, Maximum, Average and Standard Deviation of real execution time with estimated time for each procedure. Comparing estimated/real Average and Standard deviation time shows that how our approach has successfully mapped the stored procedure execution time range.

```sql
CREATE PROCEDURE [dbo].[GetProductStatus] @PlaceNo int, @PriceMin money
AS
BEGIN
declare @AvgPrice money;
declare @SumPrice money;
declare @ProductCount int;
select P_Place, P_ProductKind, ProductCount , PriceAvg, round(
    sqrt((dbo.GetSumPrice(P_Place,P_ProductKind,PriceAvg, @PriceMin))/ ProductCount),2) as StandardDeviation
FROM (SELECT P_Place, P_ProductKind, COUNT(P_ID) as ProductCount, AVG(P_BuyPrice) as PriceAvg
FROM (SELECT P_ID, P_ProductKind, P_Serial,P_Place, P_BuyPrice FROM dbo._Products
WHERE ((P_BuyPrice >= @PriceMin) and (P_Place=@PlaceNo))) AS derivedtbl_1
GROUP BY P_Place, P_ProductKind) AS derivedtbl_2
END
```

**Procedure 5. Stored procedure used for testing**

<table>
<thead>
<tr>
<th></th>
<th>Proc 1</th>
<th>Proc 2</th>
<th>Proc 3</th>
<th>Proc 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter Count</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Experiment Count</td>
<td>90</td>
<td>90</td>
<td>250</td>
<td>173</td>
</tr>
<tr>
<td>Min Time</td>
<td>14</td>
<td>14</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>Max Time</td>
<td>25</td>
<td>28</td>
<td>52</td>
<td>38</td>
</tr>
<tr>
<td>Average Time</td>
<td>19.36</td>
<td>19.92</td>
<td>36.54</td>
<td>14.03</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>3.09</td>
<td>3.36</td>
<td>9.55</td>
<td>11.52</td>
</tr>
<tr>
<td>Min Time</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Max Time</td>
<td>24.1</td>
<td>26</td>
<td>48.93</td>
<td>36.5</td>
</tr>
<tr>
<td>Average Time</td>
<td>19.11</td>
<td>19.55</td>
<td>36.44</td>
<td>14.23</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>3.24</td>
<td>3.17</td>
<td>7.46</td>
<td>9.41</td>
</tr>
</tbody>
</table>

Table 1. Overview of the execution of the four stored procedures

We have used the Absolute Error rate to calculate our approach error for all stored procedures. Absolute Error rate is a quantity used to measure how close forecasts or predictions are to the eventual outcomes. EQ.6 shows the Absolute Error rate formula:

$$\text{Absolute Error} = \frac{|f - y|}{f} \quad \text{(EQ. 6)}$$

**f: Actual Value, y: Estimated Value**

In order to check the trend of absolute error rate we have used logarithmic regression, and in all Figures from now on beside the error rate trend, logarithmic trend also illustrated.

Figure 4 shows the error rate of the estimated execution time of the first stored procedure as the round of execution increases. As shown, after about 20 times of execution the estimation error rate decreases to less than 10%. This demonstrates that our estimation can quickly converge.

The second stored procedure has three parameters, thus more complex compared with the first one. Figure 5 shows the error rate curve for the second stored procedure. The error rate reaches less than 10% after only 25 times of execution. Figure 4 and Figure 5 show that our neural network can learn the pattern of the SPs very fast. In both figures we have added the logarithmic trend to demonstrate how the estimation error rate changes with time.

The third stored procedure is more complex compared with the previous two: It has five parameters and they cannot be modeled with a simple linear relation. This type of SP is commonly known as one of the hardest types of SPs. Figure 6 shows how the error rate changes with time. The curve has many up and downs. It finally stabilized after 130 times of execution when the error rate drops to less than 10%. To speed up the training of the complex SPs, we need more training data.

We examined the ability of our approach to adjust to dynamic database changes. At the beginning, we had 20,000,000 product records in the database. Then, we added 5,000,000 more products altogether to the database. We executed the first SP 190 times and recorded the results. Figure 7 illustrates the error rate under this dynamic situation. As shown, the error rate decreases exponentially as the number of experiment rounds increases. Error rate drops to 10% after 55 times of execution. The learning period under this dynamic situation is longer compared with a staple database as shown in Figure 4. This is because when
new data is added to the database, most of the trained data become invalid. The system needs to first update the old invalid data, which takes some time. In reality, we normally do not have this dramatic update on a database. Our algorithm can model changes and update itself based on the new database condition.

The forth stored procedure is executing on a real database and can show how good our approach can perform in a real application scenario. It has two parameters and it uses multiple stored procedure and tables in order to calculate unused leave time for each personal. As shown on Table 1 this stored procedure has very wide range of execution time from 1 to 38 seconds and also high standard deviation of 11.5, which makes prediction harder compared to the previous stored procedures. Figure 8 shows how the error rate changes with time. It stabilized after 160 times of execution when the error rate drops to less than 10%. Figure 9 shows the difference between the estimated and the real execution time, the logarithmic trend shows how the difference go toward zero, which means overall estimation results of our approach is in a normalized manner.

After getting the primary results we performed an extensive analysis on the system to find its strengths and limitations. We analyzed the recorded information from different aspects to learn more about the performance and efficiency of the proposed approach.

First, we analyzed changes happen to the training data set. Figure 10 shows how the changes of the training data set as we execute the first SP. From this figure, we can see the following facts: Firstly, more data is added to the training set. Moreover, training data is updated during the experiment. Furthermore, as the execution round increases the slope of adding records decreases, while the slope of updating increases. This means the system tends to update more, instead of adding new records. This is because when the existing training vectors are close enough to the testing vector, the system does not need to add vectors to the training set but updates the vectors values.
can be seen from the figure, when data is added to the
database, the system tends to update the training set
more than adding new records. Basically, it will correct
the values which became invalid as a result of changes
happened to the database.

Third, we analyzed the accuracy of our proposed
mechanism, i.e., the difference between the real
execution time and the estimated time. Figure 12
shows real and estimated execution time and their
difference for the first SP. As the number of execution
increases, the real time and estimated time become
closer and closer. The difference is less than couple of
seconds while the total execution time is about 25
seconds. Figure 13 shows the same data for the third
SP. The average execution time of the third SP is about
36. These results show that our estimation is very
accurate.

At last, we further analyzed the distribution of the
error rate of the estimated execution time. The
experiment is based on the third SP, because it needs
more time for training and it is harder to estimate as it
shown earlier, so we have the most error in estimation
of this stored procedure. Figure 14 shows the result.
Each slice is marked with a number and a percentage,
number is the difference between estimated and real
time in second and the percentage shows the
percentage of the samples which match with the error
rate. About 45% of all estimations have an error rate
below or equal to 2 seconds, and 78% of them have an
error rate below or equal to 5 seconds. Overall the
amount of error is very low most of the time.

6. Conclusion

Stored procedure execution time estimation is a
challenging problem. We proposed a new approach to
address this issue using RBF Networks. This approach
has several advantages. First, the estimation model is
very simple yet quite efficient. Second, it can
intelligently train itself and dynamically adjust its
trained parameters to fit the changing data in the
database. This method can quickly train itself to reach
an error rate of less than 10%. Finally, our estimation
model can be implemented with SQL-based stored
procedures and be embedded into the same DBMS.
One limitation of this approach, however, is that it only
accepts numerical inputs. Therefore, in order to use it
for other data types, first we have to convert them to
numbers. A good application of this model is a
progress bar toolkit added to the DBMS. This bar can
give users a very good estimation of the progress of a
running stored procedure. This progress information is
especially important for executing long and complex
stored procedures.
References


