

Dartmouth College Computer Science Technical Report TR2001-396

**Implementing a Database Information System for an Electronic  
Baseball Scorecard**

Tiffany Wong  
Dartmouth College  
Department of Computer Science  
June 2001

Advisor: Thomas H. Cormen

## **Abstract**

We present our design and implementation of a database system of information storage and retrieval for an electronic baseball scorecard. The program uses the relational MySQL database to hold information and a Tcl API to handle interactions between the database and the user interface code. This paper discusses the inner workings of how information storage was broken down inside the database, how queries were internally constructed in accordance with the user's input, and how statistics for players and teams were calculated and returned to the user. Finally, we discuss some limitations attached to our current implementation of the program and propose improvements that can be made in future versions.

## **1 Introduction**

The game of baseball has perhaps the richest history of all sports in popular culture today. Among the most important manifestations of this history are the record books and statistics about teams and players that have been kept since professional baseball began. Keeping a scorecard for each game is an essential part of contributing to this ongoing history because it provides the necessary information to reconstruct the progression of any game, and the statistics of any player or team. It also serves as an important resource for coaches and managers in making strategic decisions in both the short and long term, for sports media personnel in analyzing a team's or player's performance, and for some fans as merely an enjoyable part of the game as a whole.

A proposed version of an electronic scorecard with a graphical user interface (GUI) and an information database [4] can have several advantages over the traditional paper model. It allows the user to efficiently store information about multiple games in a central location that can later be accessed for future use. Also, all events that are recorded during the course of a game will be instantaneously added to the database, so that game statistics will be kept current without any additional work needing to be performed by the user. In addition to holding data solely relating to specific game events, the database will also store information such as team rosters and player profiles in order to help further automate the process of initially setting up the scorecard. Finally, access to all this data allows for an extensive query system to be implemented that will enable the user to retrieve various statistics that have been filtered according to the user's specifications. In this paper, we focus specifically on the implementation of this database

query system. We first discuss how information in the various data tables will be stored and outline how we will construct the necessary information to answer queries. We then present our actual implementation of the query system and describe how it interacts with GUI. Finally, we discuss limitations that software currently imposes and propose some future improvements that can be made.

## **2 Design and Overview**

In this section we present an overview of how the various data tables that will be used to hold different pieces of information are structured. We then discuss in general terms the processes of calculating statistical information from the database.

### **2.1 Data Tables**

We have designed our program to use a relational database system as its means of information storage. The nature of the relational database model is a collection of tables that represents both data and the relationships among those data [3].

We take advantage of this model's ability to store data very efficiently by minimizing the amount of repeated information in the individual tables and instead using the `join` operation to establish relationships among the data whenever information retrieval becomes necessary. All game information will be stored in one of five table categories: one Roster table for all teams, a player list table for each individual team, one Games table for all games, a plate appearance table for each individual game, or one Game Events table for all events in all games.

There will be only one instance of the Roster table and it will contain the names of all teams who have been entered into the database. Each new team name entry will subsequently trigger the creation of a new table, referenced by the team's name, that will be used to store the name, handedness, position number, and uniform number of every player on the team. The name, position, and number fields will all be text boxes allowing the user to enter the information in whatever form he wishes. Examples of both of these tables are depicted in Figure 1.

Roster Table		Dartmouth Table			
name		name	hand	pos	no
Dartmouth		Hill	r	9	13
NWU		Wilkins	r	2	23
Vanderbilt		Crawford	l	1	3
UChicago		Thompson	r	6	29
UCSD		Sandberg	r	4	7
		Webster	l	3	10

**Figure 1. Examples of the Roster table and a team table.**

Figure 2 gives an example of the Games table, which holds situational information for each game being scored. Each table entry stores data about the date and time of the game, as well as the names of the two teams participating. Additionally, each game is assigned a unique identification number that will be used as a reference back to that specific game in subsequent tables.

GAMES Table							
ID	mo	day	yr	time	Home	Visitor	
300	05	12	01	day	Dartmouth	NWU	
301	05	16	01	day	UChicago	UCSD	
302	05	17	01	day	UCSD	Dartmouth	
303	05	20	01	night	NWU	Vanderbilt	
304	06	01	01	day	Dartmouth	UChicago	
305	06	01	01	day	Vanderbilt	UCSD	
306	06	04	01	night	UChicago	NWU	

**Figure 2. Example of the Games table.**

We now need to store more specific information about the actual events that take place during each game. We begin by creating a new table for each game, *GameID*, that is referenced by the game's unique identification number. These tables will hold information about all plate appearances that happen during the game. An example of one of these individual game tables is given below in Figure 3.

Game300 Table											
AB	I	Outs	Batter	b_team	Pitcher	Catcher	p_team	on_1st	on_2nd	on_3rd	
1	1	0	Miller	NWU	Crawford	Wilkins	Dartmouth	NULL	NULL	NULL	
2	1	1	Davis	NWU	Crawford	Wilkins	Dartmouth	NULL	NULL	NULL	
3	1	1	Lee	NWU	Crawford	Wilkins	Dartmouth	Davis	NULL	NULL	
4	1	2	Roberts	NWU	Crawford	Wilkins	Dartmouth	NULL	Davis	NULL	
5	1	0	Webster	Dartmouth	Davis	Miller	NWU	NULL	NULL	NULL	
6	1	0	Hill	Dartmouth	Davis	Miller	NWU	NULL	Webster	NULL	

**Figure 3. Example of a game table.**

Each entry in one of these game tables holds information about the current state of the game at the moment each batter is making a plate appearance. This data includes the current inning and number of outs, the names of the batter, pitcher, and catcher, the batting and fielding teams, and the names of the runners on each base. Similar to what

was done in the Games table, each at-bat is assigned a unique identification number that will be used as a reference back to the information contained in its row. The need to disjoin all of this situational information into separate tables will be made clear shortly.

All outcomes that result from an individual plate appearance— such as hits, strikeouts, assists, put-outs, and base advancements —are then stored in the Game Events table which is presented in Figure 4.

GAME\_EVENTS Table

gid	ab	player	team	outcome
300	1	Miller	NWU	Fly Out
300	1	Sandberg	Dartmouth	put-out
300	2	Davis	NWU	1B
300	3	Lee	NWU	Sac
300	3	Davis	NWU	2:safe
300	3	Webster	Dartmouth	assist
300	3	Hill	Dartmouth	put-out

**Figure 4. Example of Game Events table.**

This table also stores all events that may occur during an at-bat, but are not a direct consequence of the at-bat. For example, events such as wild pitches, stolen bases, and hit batters are all recorded in the Game Events table. In addition to the event itself, each table entry also holds information about which player was directly involved in the event and what team they were on. All of the situational information relating to the event is cross referenced by a game identification number and an at-bat number. Since there can be many outcomes relating to the same plate appearance, a lot of space in the table can be saved using identification numbers instead of entering all the situational data directly into the Game Events table.

## **2.2 Calculating Statistics**

At any point in time, the user has the ability to query the database for statistics relating to batting, pitching, catching, fielding, and running. We present the user with a list of possible constraints they can place on their query, and then use their preferences to filter out the relevant data before performing the calculations. Construction of the actual query and calculation of all statistics is abstracted away from the user and done internally. The results are then presented to the user in graphical form. Statistics can be calculated for past games as well as current ones, and the combination of different queries that can be constructed is almost limitless. We discuss information retrieval and statistic calculations in greater detail in Section 3.3.

## **3 Implementation**

In this section, we discuss our actual implementation of the database query system. We first present the reasoning behind our database choice, and then discuss how information will actually be sent to and retrieved from the database.

### **3.1 Database Choice**

We have chosen to use MySQL [2], a database management system, in our implementation for several reasons. First, MySQL is a relational database system, which is what we had intended on using when we designed our program. The choice was also cost effective since the software is open source. The database is fairly simple to use, and has no limitations on table size that would be a hindrance to our project. The MySQL

database is also accessible through the Structured Query Language (SQL), which is ideal to use in constructing our queries.

One additional benefit of choosing to use MySQL was the existence of a Tcl Application Programming Interface (API) [1] that the program could use to interact with the database. The API package provides a series of Tcl commands to connect to and disconnect from the database, execute SQL commands, and send queries and fetch their results. Since the GUI for the electronic scorecard [4] was already written in Tcl/Tk, being able to use a Tcl API meant that we could avoid the extra step of having to extend the Tcl code to work with a C/C++ API. All of these factors seemed to indicate that MySQL was the best database choice to use in this program.

## **3.2 Information Storage**

Information gets recorded into the database at several different points throughout the program. All interactions directly involving the user of the electronic scorecard are handled by the programmer of the GUI. Relevant pieces of information gleaned from these interactions are then stored in global variables that our database program can access.

### **3.2.1 Team and Player Information**

Upon starting up the electronic scorecard program, the user has several options of how to proceed, including the ability to enter data for a new team into the database. Selecting this option brings up a series of entry boxes that allow the user to enter the team name and a player roster. This information is kept in global variables and arrays until the “OK”

button has been pressed. At that point, procedures are called to send that information to the database. We chose to use this method of having global variables serve as intermediaries between the user and the database in order to give the user the opportunity to change their mind or correct any errors they may have made before anything is actually sent to the database.

Once the user has elected to enter a new team, the first entry box to appear asks for the name of the team. Once the user clicks on “OK”, we have inserted into the GUI code a `query_team_list` procedure that creates an array filled with the names of every team currently listed in the Roster table. We then check to make sure that the new team name that has been entered is not identical to any other name that already exists, since the Roster table disallows duplicate entries. Any attempts to insert a duplicate name into the table will result in a MySQL error which we do not want the user to have to deal with. So, we circumvent this problem by calling the `insert_new_team` procedure, which issues the actual SQL command to insert a new entry into the Roster table, only if the new team name is unique. Additionally, this procedure also sends to MySQL the command to create a new table named after the new team, which is structured according to the description given in Section 2.1.

Once the `insert_new_team` procedure has finished, another entry box appears that allows the user to enter player information for the new team, which is subsequently stored in a global multidimensional array. After the user has pressed the “OK” button, the GUI code calls a new procedure we have written which loops through the array and sends SQL commands through the Tcl API to insert a new entry row into the team’s table

for every new name entered. Because the players will be referenced by name throughout much of the scorecard, we do not allow the user to enter any player information into the table with the name field left blank, although any of the other fields may remain empty at the user's discretion.

In addition to entering a new team, the user also has the option of editing a currently existing team. The user has the ability to add or remove player names from team rosters, or to remove an entire team completely. We can process these requests fairly simply by using the MySQL commands `insert INTO tbl_name`, `delete FROM tbl_name`, and `drop table tbl_name`. We also give the user the possibility of editing a player's profile as listed on the team roster by using MySQL's `update` command.

### **3.2.2 Creating a New Game**

Once the user has made any necessary changes or additions to any of the current team information, they can begin the process of scoring a new game and recording game events into the database. The user begins by selecting the teams playing, the location of the game, and the date and the time of day and their choices are stored in temporary variables. The database contains a table called Defaults, that serves more as a variable. It contains only one row and one column and is initially set to 0. When each new game is initialized, we query the database to get the number stored in Defaults. We set that number to be our identification number for the game, which we store in a global variable. Then, we update the Defaults table with the old number incremented by 1. It is necessary to use the database to store this number to ensure that the number will remain static in

between every new game, even if the scorecard program has been shut down. This way, we are able to guarantee that no two games will ever be assigned the same identification number. Once we have this ID number, we can create a new entry for the Games table in the database, using the number and all of the other situational information the user has entered.

### **3.2.3 Event Information**

Many of the events get sent to the database automatically through procedures inserted into the GUI code without any additional work done by the user other than the routine work needed to keep score. Other pieces of data rely on the user to specifically record the events in order for them to be correctly sent into the database.

Every new plate appearance is automatically recorded into the proper game table in the database. The GUI portion of the scorecard already contains code to display the current batter on the screen, so we simply add in our own procedure to update the database along with the interface. The GUI code contains global variables telling us the current inning and number of outs, who the current batter is along with what team he is on, and we maintain our own variables to hold the names of the current pitcher and catcher. Additionally, the GUI programmer has provided us with an array of the numbers of the players who are currently on base. We use this information to extract the players' names out of their team's table in the database. Finally, we keep a global counter variable that will serve as the at-bat identifier number and be incremented at each new plate appearance.

The GUI programmer has provided the user with numerous menu options to handle the most common outcomes that may result from an at-bat, in order to minimize the amount of work the user needs to perform. These built-in choices include all possible outcomes for the batter, as well as common fielding situations for fly-outs, ground-outs, and double plays. Since we are concerned with everything that happens as a result of an at-bat, the GUI also provides a means for the user to enter information about what happened to all the runners and which fielders were involved during each play. All of this information is kept in various arrays and variables that we will use to record information in the database. To keep the code efficient, we have written a general purpose procedure called `insert_event_record` that takes in the name of the player involved, his team, and the event name as parameters. We will call this procedure for all entries we make into the Game Events table, referencing each entry with the game identification number as well as the at-bat identification number.

We first need to send to the database information about what happened to the batter. The menu options help simplify this task because we can easily obtain the value of the event that was selected (e.g., BB, Fly Out, HR), and we already know that the batter was the player involved in the event. We also need to record what, if anything, the fielders did during the play. The GUI presents the user with the opportunity to select the order in which the fielders, referenced by position number, were involved in the play (e.g., 6-4-3). We can use this information to assign credit to each player as dictated by the rules of baseball. For example, in a 6-4-3 Ground Out, we credit the players in positions 6 and 4 with assists, and the player in position 3 with a put-out. The GUI

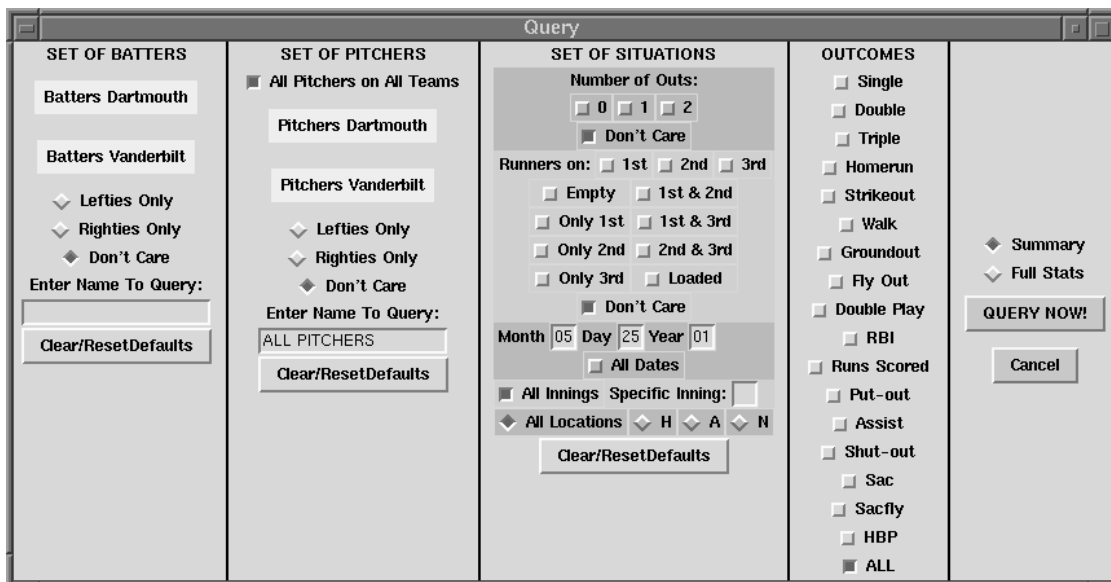
programmer has implemented an array that correlates each player with their position number so we are able to get the name of the fielder involved in the play. We can then call `insert_event_record` for each new event that we need to add to the database.

### 3.3 Information Retrieval

In this section, we discuss how queries will actually be constructed from the constraints selected by the user. We also discuss how statistics are calculated in more detail.

#### 3.3.1 Constructing Queries

Figure 5 below depicts an example of how the user will enter their query.



**Figure 5. Screenshot of Batter Query Window.**

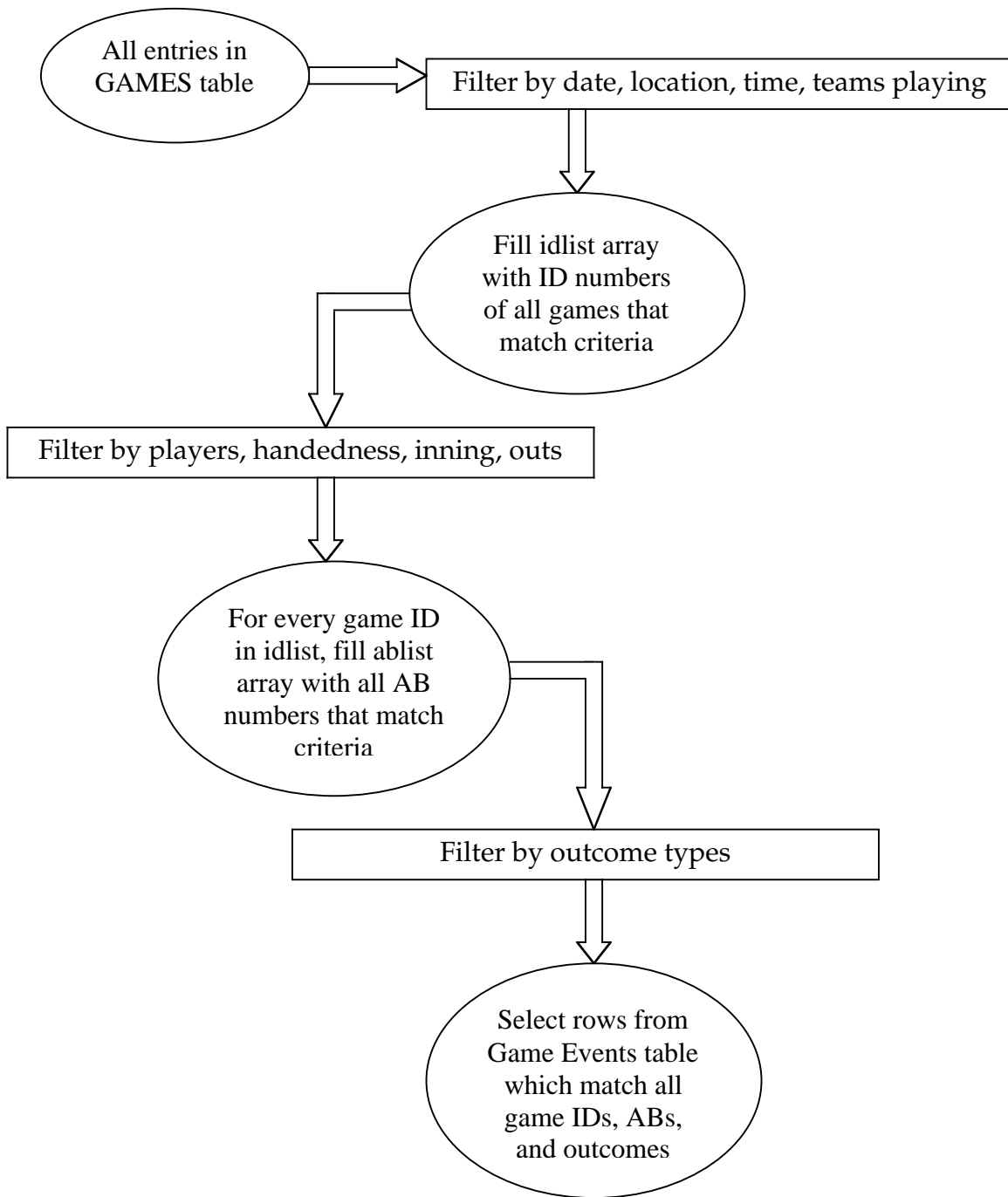
We will use their preferences to construct the actual query in SQL and retrieve the filtered results from the database. Once the user has made their selections, the values of which are temporarily stored in global variables, pressing the “Query Now” button will

initiate the call of a `send_query` procedure. This procedure begins by filtering data from all the available games to find the set of specific outcomes which match the user's criteria. It is only this set that will be used in the calculation of the statistics. The data filtering proceeds through three primary stages, which are represented graphically in Figure 6.

The first stage narrows down the possible list of games based on locational information. We query the database for the identification numbers of all entries in the Games table that match the user's criteria on date, time, and location of the games, as well as the two teams in the match-up. We use the `sql fetchrow` command from the Tcl API to have the ID numbers returned one by one, so we can store them in our *idlist* array.

The next stage filters out the relevant at-bats from each game in *idlist*. The user may narrow down their query to include either a specific batter, pitcher, catcher, or runner depending on the type of statistics they are querying for. If the user is querying for either all players on a specific team or all players on all teams, he may also choose to filter the players by their handedness. Additionally, the user can also ask for more situation-specific information, such as the inning, the number of outs, or the positions of the runners. The user has the opportunity to make the query as narrow or as broad as they want and they are not limited to choosing only one specific situation.

At this point in the `send_query` procedure, we want to loop through the *idlist* array and query each game table for a list of all at-bats that match the user's situational criteria. But before we perform the query, we must first join together each game table



**Figure 6. Data filtration procedure.**

with the tables of the two teams playing in the game so we can filter the players by handedness. Figure 7 below gives an example of such a joined table.

Game300 Table												
AB	I	Outs	Batter	hand	b_team	Pitcher	hand	Catcher	p_team	on_1st	on_2nd	on_3rd
1	1	0	Miller	r	NWU	Crawford	l	Wilkins	Dartmouth	NULL	NULL	NULL
2	1	1	Davis	r	NWU	Crawford	l	Wilkins	Dartmouth	NULL	NULL	NULL
3	1	1	Lee	l	NWU	Crawford	l	Wilkins	Dartmouth	Davis	NULL	NULL
4	1	2	Roberts	r	NWU	Crawford	l	Wilkins	Dartmouth	NULL	Davis	NULL
5	1	0	Webster	r	Dartmouth	Davis	r	Miller	NWU	NULL	NULL	NULL
6	1	0	Hill	l	Dartmouth	Davis	r	Miller	NWU	NULL	Webster	NULL

**Figure 7. Example of a game table joined with team tables.**

We can now query this newly joined table for all of the relevant at-bats matching the user's specifications. We fill in a new 2-dimensional array, *ablist*, where the row index will correspond to the game identification number. Each entry in the row will contain an at-bat identification number that has been returned by the query for that game.

We now have every (game ID, at-bat ID) combination that is relevant to the user's query criteria. At this stage, we can now query the Game Events table for all of the outcomes that the user has chosen. We loop through every entry in the *ablist* array, and for each index, we append to our query string an `or` clause that filters out rows matching the game ID number and the at-bat ID number (e.g., `select from Game Events where (gid=1 and ab=2) or (gid=3 and ab=5)...`). Finally, we append to our query string the set of outcome types that the user has selected. We can now issue an `sql query` command that will return every row in the Game Events table which matches all of the criteria the user entered in the query box. We store the results of this final query command in a temporary table which will be used in our statistic calculations.

### **3.3.2 Calculating Statistics**

To help standardize the results, we return the same set of statistics to the user for each query, regardless of what outcomes they selected. Therefore, depending on how narrow the constraints were, it is possible that many of the statistics that are returned will have values of 0. At this point, most of the statistic calculations are done either by querying the temporary table of results for a specific outcome and then obtaining the number of entries returned, or by performing mathematical operations on two or more different statistics. The Tcl API provides a nice `sql numrows` command that will return the number of rows in the query results, which in our case is also identical to the number of times a specific outcome occurred. Keeping in mind this convenient means of compiling statistics, we make sure that when we enter events into the database, we enter each one separately so as to have an accurate count. For example, when storing the number of RBIs that resulted from one at-bat, we create a new entry in the Game Events table for each RBI that the batter was credited with. This way, we are able to very simply count the number of RBIs that occurred in a given situation without having to do any string parsing.

### **3.3.3 Performance**

We conducted several tests to measure how fast queries are performed on tables with large record sizes. Tables were created and filled with a large number of entries, and were then queried for different numbers of records as a proportion of table size. The results are given below in Table 1:

**Table 1. Performance Testing**

<b>No. of table records</b>	<b>Time to query all records</b>	<b>Time to query 50% of records</b>	<b>Time to query 20% of records</b>	<b>Time to query 10% of records</b>
3000	0.13s	0.09s	0.06s	0.02s
8000	0.32s	0.19s	0.08s	0.04s
15000	0.65s	0.36s	0.15s	0.10s

(Note: Tests were conducted using MySQL on a UNIX platform.)

The results indicate that MySQL continually performs well on large tables, and that the time needed to execute a query varies directly with the number of records returned by the query. The largest query tested was for 15,000 records, and the execution time was under 1 second. We therefore expect that the overall time of query execution for our program will depend more on the amount of work done through Tcl code and less on the amount of work done through the database.

#### **4 Current Limitations and Future Work**

There are several areas of limitation contained in this version of the program that have the potential to be improved for later versions. One severe limitation is that the program is not yet fully comprehensive at this point. It is extremely important that the program is able to record data for all situations that may arise or else the statistics returned will be inaccurate. The more information we are able to store, the more statistics we are also able to return. Many of the at-bat event category menus list “other” as an option for telling the scorecard what happened during the at-bat. However, selecting this option

currently does not record anything in the database. One reason for this is that the “other” option gives the user a text box in which to enter the event. Since the user may enter anything he wishes, there is no way to either interpret his input into standard form or to offer him the option of later querying for the event he entered.

Another limitation of the current version of the scorecard is that there is still a lot of user responsibility for entering events. One of the proposed benefits of an electronic scorecard versus a paper one was the automation of information storage. Unfortunately, the complex rules of baseball make it impossible to ever make very many definite assumptions about what may have happened during the course of the game. For example, although it is very improbable, it is not completely impossible for a batter to reach first base after getting three strikes. However, there is still room for improvement in making the program as intuitive as possible.

An alternative method of query construction that needs to be explored is to join together the information contained in the Games table, the individual game tables, the team tables, and the Game Events table into one large table. Then, we send only one query containing all of the attributes selected by the user to this joined table. However, in order to use this method, we would need to restructure the way we information is stored in our tables. Rather than have individual tables for teams and game at-bats, all of the information stored in those table types must be conglomerated into two large tables—one to hold player information for all teams, and one to hold at-bat information for all games. Using this breakdown of tables, we would then be able to easily join together all the information we have stored into one large table from which we can select our query.

This method needs to be implemented and compared against the algorithm we presented in Section 3.3 so that we may determine which would be a more effective method to use in our program.

## **5 Summary**

This paper presented the basic infrastructure for a database query system that will interact with an electronic baseball scorecard GUI. Although the current implementation is still an early version of the program, we have designed a functional algorithm that allows us store a lot of important game information in the database and later retrieve that information in the form of basic player and team statistics. Through the interaction of Tcl and the MySQL database, our program allows the user to utilize an intuitive user interface to submit an almost limitless number of potential queries. With the groundwork we have laid, we should be able to adapt the query system to be even more comprehensive and useful in future versions.

## **Acknowledgments**

I would like to thank my advisor, Tom Cormen, for all his help and support throughout this project. Also, Lea Wittie for all her help with debugging program, Jessica Webster for giving me the opportunity to join in with her baseball program and for answering all my Tcl questions, and John Konkle for all his help in setting up the software we needed.

I would like to thank my family for all the support they have given me, both morally and financially. And finally, thanks to my friends Brad, David, Elizabeth, Rob, and Terrence for all of the great times we've had in the past four years.

## References

- [1] Tony Darugar. *The Tcl Generic Database Interface*. Available at <http://www.binevolve.com/~tdarugar/tcl-sql/docs/api.html> (27 September 1998).
- [2] MySQL AB. *MySQL database version 3.23*. Available at <http://www.mysql.com/downloads/mysql-3.23.html> (22 January 2001.)
- [3] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 3<sup>rd</sup> ed.* Boston: WCB McGraw-Hill, 1997.
- [4] Jessica Webster. *Stats and Bats: Covering Your Bases from Interface to Database*. Senior Honors Thesis, Department of Mathematics and Social Science, Dartmouth College.