MySQL and The Trouble with Temporal Data

Chad Vicknair
Department of Computer and Information Sciences
University of Mississippi
University, MS 38655
cmvickna@cs.olemiss.edu

Dawn Wilkins
Department of Computer and Information Sciences
University of Mississippi
University, MS 38655
dwilkins@cs.olemiss.edu

Yixin Chen
Department of Computer and Information Sciences
University of Mississippi
University, MS 38655
ychen@cs.olemiss.edu

ABSTRACT
Storing historical data is not new. Data warehouses are filled with it. However, querying temporal data has only, in the past few years, become a realistic possibility.

Yet the SQL standard has not had significant updates in temporal data support since 1992. The demands of the modern world are no longer satisfied by decades-old solutions. MySQL, an open-source database extremely popular in the sciences, has still not yet fully implemented SQL-92. Its approach to temporal needs is lacking. Too many queries are not answerable, and those that are answerable are often answered inconsistently.

1. INTRODUCTION AND MOTIVATION
A “temporal data type” is a data type specifically intended to handle time-related data. Generally, this refers to dates and times. Probably the most common use for temporal data is to represent an ordered series of events. Event 1 happened at time A. Event 2 happened at time B, such that B happens subsequent to A. Event 3 happened at time C, such that C happens subsequent to B. This facilitates a query like “Find all events that happened after Event 1.”

A second use for temporal data is in representing indeterminate portions of time. For example, a database of cooking recipes could list the estimated time of each step in the process. This facilitates a query like “Find all recipes, the sum of whose steps are less than 2 hours.” In other words, “Find all recipes that take less than 2 hours to prepare.”

Another use for temporal data is to maintain a clearly defined start and end point. For example, in a university, students generally enter the university in some default major such as Undecided. Eventually, most students change majors to specialize in some field. Some students change majors multiple times. A full history could be maintained on each student to reflect his or her major at various points in their tenure at the university. This would facilitate a query like “Find the average length of time a student waited before first choosing a major.”

While the Structured Query Language (SQL) standard and individual database vendors have recognized the virtue of supporting temporal data, they have not supported temporal data as fully as they should have. For any database management system (DBMS), it would be nice to support all domains fully and completely, but attempting this goal would result in a dramatic reduction in efficiency. As temporal data becomes of greater interest and as it becomes more practical to measure and store temporal data at various levels of precision, some significant problems become apparent. It is clear that MySQL has difficulty handling many temporal tasks. In attempting to efficiently support more mainstream domains, MySQL does not support domains that were less significant in the past, but which are gaining popularity.

This paper is structured as follows: Section 2 gives background information about MySQL. Section 3 contains an explanation of how the SQL standard and MySQL implement temporal data support. Section 4 is a discussion of the problems, contradictions, and open issues associated with the SQL standard and MySQL’s implementation of the SQL standard. Existing approaches to adding more complete temporal data support are discussed in Section 5. Section 6 suggests directions for the future.

It should be noted that as a convention, capital letters are used to denote a specific data type used in the SQL standard or MySQL, while standard capitalization is used to denote generic references to some type of data. For example, a DATE is a MySQL data type that stores a date. “DATE” is the data type. “Date” is a generic data type. Similarly, an INTERVAL is an SQL representation of an interval of time.

Also, “table” is used in SQL statements in this paper as a generic table name. The word “table” is typically a reserved word and is understood to not be a valid table name.

2. ABOUT MYSQL
MySQL is an open-source relational database management system. It is owned by Oracle Corporation and can be used under either the GNU General Public License1 or a standard commercial license purchased from Oracle. MySQL is a robust, multi-threaded, transactional DBMS. It is highly scalable and can be distributed across multiple servers. Because it can be used free of charge, it holds significant market share within the scientific community. While often considered inappropriate for domains of high security like financial institutions or certain areas of the government, MySQL

1http://www.gnu.org/licenses/gpl.html
has become the leading relational database in many areas of academia, including scientific research and educating students.

MySQL does have some limitations. For example, it does not fully implement the SQL standard. According to MySQL, “one of our main goals with the product is to continue to work toward compliance with the SQL standard” [5]. As such, there are certain aspects of SQL that MySQL does not implement, especially in the area of temporal data.

All testing and examples were performed on a computer running Ubuntu Linux version 11.10. Except where otherwise noted, the version of MySQL used in this paper is 5.1, as it is the latest version readily available for Ubuntu Linux.

### 3. TEMPORAL DATA

#### 3.1 SQL and Temporal Data Types

The SQL standard introduced temporal data types in the 1992 revision, so-called “SQL-92.” SQL-92 describes a distinction between intervals and what it terms “datetimes”. A datetime specifies some fixed point in time. Datetime can be subdivided into specific data types: DATE, TIME, and TIMESTAMP [3].

A DATE contains a year, month, and day. A TIME contains an hour, minute, and second. A DATETIME contains a year, month, day, hour, minute, and second. According to the SQL standard, “[i]tems of type datetime are mutually comparable only if they have the same datetime fields” [3].

An INTERVAL is subdivided into two classes: year-month intervals and day-time intervals. Semantically, an interval represents some length of time with indeterminate endpoints [3]. For example, “5 months,” “4 days,” and “30 hours” are all intervals. Year-month intervals represent some number of years and months. Day-time intervals represent some number of days, hours, minutes, seconds, and fractions of a second.

SQL-92 defines operations that can be performed on intervals and datetimes. Adding, subtracting, multiplying, and dividing datetimes and intervals will result in different types. For example, a subtraction of two years, 2001 - 2002, produces an interval of 1 year. Operations are not commutative. Some basic operations and their result types are illustrated in Table 1.

<table>
<thead>
<tr>
<th>Operand 1</th>
<th>Operator</th>
<th>Operand 2</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datetime</td>
<td>+ or -</td>
<td>Datetime</td>
<td>Interval</td>
</tr>
<tr>
<td>Interval</td>
<td>+</td>
<td>Datetime</td>
<td>Datetime</td>
</tr>
<tr>
<td>Interval</td>
<td>+ or -</td>
<td>Interval</td>
<td>Interval</td>
</tr>
<tr>
<td>Interval</td>
<td>* or /</td>
<td>Numeric</td>
<td>Interval</td>
</tr>
</tbody>
</table>

Subsequent SQL revisions have done little to further storage, manipulation, and querying of temporal data.

#### 3.2 MySQL and Temporal Data

MySQL implements part of the SQL-92 standard with respect to temporal data. It supports the following temporal data types: DATE, TIME, DATETIME, and TIMESTAMP.

DATE works just as described in the SQL standard [6]. Supported DATE values range from 1000-01-01 to 9999-12-31. In this case, “supported” refers to the ability to guarantee correctness for dates within the range. Earlier dates may work but are not guaranteed [6].

Both the MySQL DATETIME and TIMESTAMP function as the TIMESTAMP data type specified in SQL-92. DATETIME values are stored as dates paired with times. They contain the following information: year, month, day, hour, minute, and second. Supported DATETIME values range from 1000-01-01 00:00:00 to 9999-12-31 23:59:59 [6]. As above, “supported” means that the database can guarantee that the data will perform as expected, though earlier dates may work.

TIMESTAMP values also store information about dates and times, but they are stored based on Unix epoch time. They are inserted and retrieved as DATETIME in the local timezone (as set on the server), but they are stored as Coordinated Universal Time (UTC) [7]. TIMESTAMPs can only range from 1970-01-01 00:00:00 UTC to 2038-01-19 03:14:07 UTC, the current range of supported Unix time. Any datetime outside of this range is invalid.

The TIME type works a bit differently. TIME data can be either a representation of time of day or a representation of a temporal interval [8]. The semantics of a TIME column are context-sensitive. As such, TIME values can range from -838:59:59 to 838:59:59. Values outside the acceptable range are rounded to the nearest acceptable value.

The interval data type is not directly supported in MySQL. While it can be simulated using the TIME data type, the range is limited to the range of the TIME data type. Intervals outside this range are not supported as a data type for a column. There is currently no support for storing intervals per se in a table.

MySQL has several built-in temporal functions [9]. They allow for simple datetime manipulation. They include the ability to create datetimes from their component parts as well as to perform several of the datetime calculations described in Table 1. Intervals can be used as arguments in datetime functions. For example, the ADDDATE() function, which takes as arguments a datetime and an interval and returns the datetime plus the interval, can be called as follows:

```
SELECT ADDDATE('2004-02-25', INTERVAL 10 DAY);
```

It will return '2004-03-06', which accounts for the month changeover and leap year. MySQL makes best guesses when converting strings to temporal data. In almost every case in which a datetime would be accepted, MySQL can also accept a string representation of that datetime. It looks at the string to determine whether it can convert the string to a datetime using an “educated guess” based on the layout of the string. There is a function, STR_TO_DATE(), which allows the user to specify a DATE or DATETIME as a string and a string format that specifies the exact position of the various fields within the string.

### 4. PROBLEM DESCRIPTION

Both the SQL standard for temporal data and MySQL’s implementation of that standard have several unfortunate limitations, discussed below.


### 4.1 Null Data

Storing incomplete or unknown temporal data in SQL is typically done with a NULL. DATE and TIME data types, as described by SQL-92, are considered each to be composed of three separate integers of various acceptable ranges. For example, DATE is the single data type assigned to a table attribute that stores a date (and not a time). SQL only allows for an all-or-nothing nullability. That is, the data as a whole can be null, but parts of a date cannot.

As a workaround, within a datetime, MySQL does not use NULLs for individual date and time fields. Instead, it uses null markers, which are always zero. A null year, month, day, hour, minute, or second is zero [6].

This works well for dates because there is no year 0000, nor month 00, nor day 00, 00:00:00, however, is a valid time; it is exactly midnight. This can lead to ambiguity. If a table is to have a single column that sometimes contains a date and time and sometimes contains just one or the other, then there is no real difference between date X with an unknown (i.e., NULL) time and date X at midnight.

Because of the way null fields within a temporal column are marked, it is possible to compare two temporal data types with different component fields. According to the SQL standard, however, this should be impossible [3]. The fact that this happens is far less distressing then how it happens. MySQL uses zero as a null marker to denote that part of a datetime value is null. This allows for calculations to be performed on datetime values without having to make special allowances or checks for null data.

This has the interesting advantage of naturally providing intuitive search results. With most SQL queries, it is often assumed that, unless some query condition that is incompatible with a NULL is specified, a query should return fields with both null and non-null values. Therefore, the query

```
SELECT d FROM table ORDER by d;
```

where d is a DATETIME column, should return a list of all rows in table, including NULLs. Requesting all the years in d with the query

```
SELECT YEAR(d) FROM table ORDER BY YEAR(d);
```

would return a list of all the years with 0000, listed first. When requesting a full list of all dates ordered by the column d, a DATETIME column, MySQL would return NULLs first, followed by a series of dates with the null years first, followed by all non-null (i.e., non-zero) years listed below. Within each year, the null months would be listed first, followed by all non-null months.

Using null markers that, when ordered, precede all valid values makes sense, but because 0 is a valid hour, minute, and second, the system breaks down when dealing with time. Time will not behave the way dates do, and this inconsistency can potentially lead to significant confusion.

One of the major disadvantages of using the zero as a null marker is that one might not want the null markers to behave the same way valid data does in all circumstances. For example, the query

```
SELECT ADDDATE('0000-02-15', INTERVAL 15 MONTH);
```

returns 0001-05-15, which is outside the range of valid DATE values and no longer guaranteed to work. If the year 0000 is truly representative of NULL, then it should not increment. The above query should return instead 0000-05-15.

---

### 4.2 Granularity

Related to storing null data is storing data in multiple granularities. Because zero is meaningful in every time field and because MySQL also uses zero as a null marker for each field in a time, storing TIMES or DATETIMES of various granularities in a single column is not possible. Record 1 may be assigned an unknown time at some point during the 5:00 AM hour. Record 2 may be assigned an unknown time at some point during the first minute of the 5:00 AM hour. Record 3 may be assigned a time of exactly 5:00 AM and zero seconds as its time. All three times will be stored as 05:00:00. In an equality comparison, all three times will be considered equal, yet they are not equal.

SQL-92 neither requires nor suggests any mechanism for storing a time with a precision greater than second [3]. In addition, MySQL 5.5 and previous versions provide no direct support for storing milliseconds in a table column. Much like the way MySQL handles intervals, MySQL 5.5 recognizes that fractional seconds exist. They are just ignored when storing fractional seconds in the database [11]. Consider the following query:

```
SELECT MICROSECOND('2005-10-31 16:17:18.987654321');
```

The query will return 987654, the microsecond, portion of the datetime string. However in MySQL 5.5, inserting the same datetime string into a table column declared to be DATETIME will only store 2005-10-31 16:17:18. As of MySQL 5.6.4, MySQL allows for the storage of fractional seconds in database table columns [12]. Version 5.6.4, however, is, at the time of this writing, still in development, and version 5.5 is the latest generally available release.

---

### 4.3 Overflow

Within MySQL, a DATE can fall within one of three basic ranges: supported, legal, and illegal. “Supported” means accepted by the system and guaranteed to work. “Legal” and “illegal” are terms not explicitly defined but which were extrapolated from other terminology used in the reference manual. “Legal” means possibly accepted by the system but not guaranteed to work. “Illegal” means not accepted by the system. The supported range for DATETIMes is 1000-01-01 to 9999-12-31. Also supported is a zero in any field or fields. The legal, but unsupported, range is 0001-01-01 to 9999-12-31. The illegal range is any date before 0000-00-00 or after 9999-12-31. While in practice, dates that are legal but not supported have held up as well as supported dates, it is unclear if and when the use of unsupported dates would break down.

In practice, relatively few dates before the year 1000 are stored with the precision of an exact day. If only the year is known, an integer type can be used to handle years. The larger issue, however, is one of granularity. If various historical dates are known, some of which are before 1000 and some of which are after, and if granularity finer than year is required, storing only the year in an integer column is not an option.

For most domains, the inability to store exact dates before 1000 is inconsequential. Occasionally, however, this information may be necessary. Consider, for example, genealogy. A genealogy database would require, on occasion, the storage of exact dates prior to the year 1000. MySQL does not guarantee dates before 1000. Though those dates tend to
work in practice, this is likely not enough of an assurance to trust with important business information.

This problem extends to dates in the BC era. BC dates are not "legal." Usually, this is not an issue, as exact BC dates are rarely known, and years can be stored in an integer column. AD years could be stored as positive integers and BC as negative integers, but this is a workaround, not actual support for BC dates. As before, though, if the table column is to hold dates across a broad spectrum of years, integers may no longer work. Storing even a single exact date would mean the data type for the column could no longer be INTEGER and would have to be DATE or DATETIME. Null markers (in the case of MySQL) would be needed for the unknown days and months. DATE, however, has no mechanism to support BC dates. Attempting to insert any date before year 0000 results in an error.

MySQL has only limited support for intervals. It supports them as the data types for a table attribute, but it forces them into the TIME data type. Therefore, interval data types are limited by the maximum and minimum values of TIME, \(-838:59:59\) to \(838:59:59\). This translates to \(\pm 35\) days. Intervals beyond 35 days are not supported.

For virtually every data type in virtually every system, there is the possibility of data overflow. How the system handles overflow, however, should be consistent.

The function \texttt{DATE()} takes, among other data types, a string or integer that represents a date and returns a DATE object. \texttt{DATE()} is not always necessary, as MySQL does have the ability to parse a string submitted to an SQL statement or query.

The query

\begin{verbatim}
SELECT DATE('2001-13-20');
\end{verbatim}

returns NULL. 2001-13-20 is an illegal date, as there is not 13th month. An attempt to insert this date into a table with a single DATE or DATETIME column, \(d\), with the statement

\begin{verbatim}
INSERT INTO table VALUES(DATE('2001-13-20'));
\end{verbatim}

will add NULL to the table. Consider the table from the previous \texttt{INSERT} statement filled with a wide spectrum of supported \texttt{DATE}s, including NULL and \texttt{DATE}s with null markers (i.e., 0000-00-00). Issuing a query with the illegal date 2001-13-20 within the \texttt{DATE()} function in the WHERE clause, such as

\begin{verbatim}
SELECT d FROM table WHERE d = DATE('2001-13-20');
\end{verbatim}

will return an empty set without errors or warnings. In three separate contexts, \texttt{DATE('2011-13-20')} is twice evaluated to be NULL and once evaluated to be something else entirely, though it is unclear exactly what.

Without the function \texttt{DATE()} wrapping the date string '2001-13-20', MySQL interprets the invalid date differently. The query

\begin{verbatim}
SELECT '2001-13-20';
\end{verbatim}

returns simply 2001-13-20, presumably still as a string. Used within an \texttt{INSERT} statement into the same table as above, the query

\begin{verbatim}
INSERT INTO table VALUES('2001-13-20');
\end{verbatim}

inserts a DATE filled with null markers (i.e., zeros). Using the illegal date in the \texttt{WHERE} clause when querying the table from the above example without the \texttt{DATE()} function such as

\begin{verbatim}
SELECT d FROM table WHERE d = '2001-13-20';
\end{verbatim}

results in an empty set with warnings that state \texttt{Incorrect date values}.

These examples of illegal dates have an illegal month, but the same is true of an illegal value in any field (e.g., day 39 or year 10000).

\texttt{TIME} overflows work differently still. Legal \texttt{TIME} ranges from \(-838:59:59\) to \(838:59:59\). A time before or after this range is rounded to the nearest valid time. Therefore, the query

\begin{verbatim}
SELECT TIME('1000000:50:30')
\end{verbatim}

returns \(838:59:59\). As with dates, the same SELECT statement without the \texttt{TIME()} wrapper function echoes back the submitted string, in this case \(1000000:50:30\).

Given a table with a single \texttt{TIME} column, \(t\), the statement

\begin{verbatim}
INSERT INTO table VALUES(TIME('1000000:50:30'));
\end{verbatim}

inserts the time \(838:59:59\) into the table. This holds true for an \texttt{INSERT} statement without the \texttt{TIME()} wrapper function.

When used in a \texttt{WHERE} clause of a SELECT statement, the presence of the wrapper function alters the behavior. Given the above table filled with a wide range of supported times, times filled with null markers, and NULL, the query

\begin{verbatim}
SELECT t FROM table WHERE t = TIME('1000000:50:30');
\end{verbatim}

returns the set of rows where \(t\) is equal to \(838:59:59\). However, the query

\begin{verbatim}
SELECT t FROM table WHERE t = '1000000:50:30';
\end{verbatim}

returns an empty set with a warning that \(1000000:50:30\) is not a valid \texttt{TIME}.

Minute or second overflow, on the other hand, behaves more like \texttt{TIME} overflow. The query

\begin{verbatim}
SELECT TIME('100:70:30');
\end{verbatim}

returns NULL. Without the wrapper function, the SELECT echoes the string back. Attempting to insert a time with a minute and/or second overflow will add NULL to the table if the \texttt{TIME()} function is used and a \texttt{TIME} filled with null markers otherwise. Issuing a query with a minute and/or second overflow in the \texttt{WHERE} clause will return an empty set and warnings for an incorrect time value regardless of whether the wrapper function is used.

4.4 Non-Gregorian Calendars

The Gregorian calendar is the dominant calendar in the Western world and takes into account the fact that the Earth's revolution around the Sun is not a whole number of days. This results in occasional slight adjustments the calendar. Leap year is an example of this. Prior to the Gregorian calendar, the Julian calendar was the dominant Western calendar. The Julian calendar did not account for leap year. Adjustments had to be made in the transition from the Julian calendar to the Gregorian calendar.
MySQL uses the proleptic Gregorian calendar, meaning that all dates are fixed around the Gregorian calendar and that the Gregorian calendar is used to represent even those dates that took place during the time when the Julian calendar was in use [10]. The same mechanism can be seen in play in the annual dates of Hanukkah. Hanukkah moves around the Gregorian calendar because it is based on the Jewish calendar. Within the Jewish calendar, Hanukkah holds a fixed position, but non-Jews tend to relate to Hanukkah in terms of the Gregorian calendar, which is why it appears to move around from year to year.

There are many other calendars used around the world. There is the Islamic calendar, the Chinese calendar, the Buddhist calendar, and the Hindu calendar just to name a few. MySQL supports none of them.

4.5 Interval vs. Duration

In 1988, the International Organization for Standardization published ISO 8601, which contained several recommendations for effectively utilizing temporal data. It defined the term “duration” to be any non-negative number of the smallest time granularity recognized by the system (i.e., any amount of time) [4]. SQL-92 calls this an “interval.”

ISO 8601 defines an “interval” to be any duration with fixed endpoints [4]. SQL-92 has no concept of this per se. Of course, it can be represented using more primitive data types, but it does not exist as a single data type.

Intervals, in the ISO 8601 sense, can be used to represent, in a single column, any time that starts at one datetime and ends at another. They can be used to record any clearly defined start and end time. A patient entered a hospital at datetime 1 and was discharged at datetime 2. A television news program airs every day from time 1 to time 2.

The concept of the interval, as defined by ISO 8601, has implications in other areas as well. It does not have to remain in the realm of temporal data. They are particularly useful for handling estimates or inexact data or data for which storing all data points would be impractical or unnecessary. During the next holiday season, each store in a database table consists of a single DATE column, the SQL statement

```
SELECT DATE('2010-11-12');
```

will insert the value 2010-11-12 into the column. Yet, in both the previous SQL statements, an argument could be made that the system was simply making the best of bad input. If the above table, rather than having a DATE column, had had a DATETIME column, a data type with both a date component and a time component, the query

```
INSERT INTO table VALUES('10:11:12');
```

would insert the value 2010-11-12 00:00:00 into the table column. Surely, this best guess is not reasonable. Surely what was meant was 0000-00-00 10:11:12, but because of this unorthodox string parsing, data will probably be stored in an incorrect field. To make matters worse, when executing, the INSERT statement produces no errors nor warnings, so the inaccurate data storage is silent.

The `TIME()` function has a similar problem. `TIME()` accepts a number of data types as an argument. Of note here are string and decimal numbers. In MySQL, at coarser granularity, `TIMEs` consist of hours, minutes, and seconds. At finer granularity, `TIMEs` consist of hours, minutes, seconds, and fractions of a second. In MySQL 5.5 and before, `TIMEs` with precision of less than a second can only be passed as arguments to and returned from functions. The finer granularity cannot be stored in a table. MySQL does not require the user to pass a full and complete time as an argument to `TIME()`. One can use common shorthand. The query

```
SELECT TIME('11:12:00');
```

is interpreted to be 11:12:00, which is 11 hours and 12 minutes. Representing a time as 11:12 is an inherently ambiguous notation. It could also be interpreted as 11 minutes and 12 seconds. At some point, a convention was established that defined the above query as returning 11:12:00.

However, the query

```
SELECT TIME('1112');
```

returns 00:11:12, 11 minutes and 12 seconds. Removing the colon alters the semantics in an unexpected way. Note that the string argument was maintained for consistency with the previous example. In the previous example, using a string was necessary as submitting 11:12 without the single quotes returns an error. In this example, the ‘1112’ could be represented as an integer without changing the return value of `TIME()`.

Three logical interpretations of `TIME('1112')` are

1. as 11 hours and 12 minutes
2. as 11 minutes and 12 seconds
3. as 1,112 seconds, which is 18 minutes and 32 seconds

The first interpretation is more consistent with `TIME('11:12')`. The second and third interpretations are more consistent with the subsequent examples. Consider the following query:

```
SELECT TIME('111.2');
```

The result will be 00:01:11.2, which is 1 minute and 11.2 seconds. Consider, too, the following query:

```
SELECT TIME('1:11.2');
```
The results will be 01:11:00.200000, which is 1 hour, 11 minutes, and 0.2 seconds. This is probably one of the stranger potential interpretations of '1:11.2'. Furthermore, given a table consisting of a single DATETIME column, the statement

```
INSERT INTO table VALUES('1:11.2');
```

will insert 0001-11-02 00:00:00, which is a date that exists outside the range of supported dates in MySQL.

5. THEORETICAL WORK

Most work in the area of expanding the ability of relational databases to handle temporal data takes a theoretical approach. They tend to take an approach that would add to relational logic or involve editing the source code of a database, solutions which are largely out of the question for a database administrator.

C. J. Date, et al., present a thorough theoretical implementation of interval (as defined by ISO 8601) support for a relational database [2]. Date, et al., codify a method of reasoning about intervals in a relational sense and help define a set of operations on intervals. They also show how complicated interval queries can become. The authors write all examples in Tutorial D, a language they created that very few, if any, full-scale databases actually support. For a relational theoretician or database designer, this may work very well, but for a database administrator or database student, writing all examples in an obscure language is not helpful to them.

A book by Claudio Bettini, et al., also addresses the issue of temporal databases, but the authors specifically concentrate on the issue of multiple granularities [1]. This book contains an incredible amount of theory, presenting several definitions, theorems, propositions, lemmas, and algorithms. It does address intervals, as well, but it focuses more on supporting multiple temporal granularities. This book would be an extremely valuable tool for a designer of a temporal database, but it is a bit too theoretical for an average database administrator or database student.

6. CONCLUSIONS

There is a growing need to create temporal databases, that is, databases capable of effectively storing, manipulating, and querying temporal data. While this is a noble goal and may, in the future prove, immensely helpful, the solid theories of handling temporal data have not yet been implemented in the SQL standard or MySQL. For most purposes today, what is needed is the ability to efficiently and understandably extend existing databases to create more usable, consistent temporal data support.

While this paper has, in general, addressed temporal data from the perspective of a database administrator or user, there is another issue. In part because it is free, MySQL is commonly used in academia. Students who are still beginners at databases will often be learning database theory and practice using MySQL. Often, even in beginning database classes, temporal data types are covered. The database students of today are the database administrators of tomorrow. The pitfalls of MySQL and its handling of temporal data are often not taught in database classes. Students should be able to learn about MySQL’s idiosyncrasies in an isolated educational environment that encourages experimentation and

allows for failure before they stumble across the obstacles of temporal data in SQL and MySQL in a real-world environment that is much less tolerant of failure.

7. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under award number EPS-0903787. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect those of the National Science Foundation.

8. REFERENCES


