Viewing and Analyzing Multimodal Human-computer Tutorial Dialogue: A Database Approach

Jack Mostow, Joseph Beck, Raghu Chalasani, Andrew Cuneo, and Peng Jia

Project LISTEN[1], 4213 RI-NSH, 5000 Forbes Ave, Carnegie Mellon University, Pittsburgh, USA
Telephone: 412-268-1330 (voice) / 268-6436 (FAX); email: Mostow@cs.cmu.edu

Abstract. It is easier to record logs of multimodal human-computer tutorial dialogue than to make sense of them. In the 2000-2001 school year, we logged the interactions of approximately 400 students who used the Reading Tutor and who read aloud over 2.4 million words. This paper discusses some of the problems in extracting useful information from such logs and the difficulties we encountered in converting the logs into a more easily understandable database. Once log files are parsed into a database, it is possible to write SQL queries to answer research questions faster than analyzing complex log files each time. The database permits us to construct a viewer to examine individual Reading Tutor-student interactions. This combination of queries and viewable data has turned out to be very powerful. We provide examples of questions that can be answered by each technique as well as how to use them together.

Keywords: viewing and analyzing tutorial dialogue, session logs, evaluation tools, Project LISTEN, Reading Tutor, multimodal dialogue, speech I/O, session browser, database queries, perl, MySQL

1 Introduction

It is easier to record logs of multimodal human-computer tutorial dialogue than to make sense of them. We discuss this problem in the context of Project LISTEN's Reading Tutor, which listens to children read, and helps them. The resulting multimodal dialogue includes mouse clicks and speech as student input, and spoken and graphical assistance as tutor output. In the 2000-2001 school year, hundreds of students used the Reading Tutor daily at three elementary schools, reading over 2.4 million words. How can we analyze so much data? The Reading Tutors logged thousands of sessions, but the logs are too detailed to see the forest for the trees.

An alternate way to capture detailed tutorial interactions in human-viewable form is to videotape them, as we have done in a number of studies [7]. Video has obvious advantages, but many drawbacks. Video is laborious to record well at schools, invades privacy, can distort student behavior, captures only one level of externally observable detail, omits internal events and tutorial decision processes, and is tedious to analyze or search. To avoid these drawbacks, we describe a database approach we developed to view and analyze logs of children's interactions with Project LISTEN's Reading Tutor.

1.1 Project LISTEN's Reading Tutor

The aspects of the Reading Tutor relevant to this paper are the channels of communication between Reading Tutor and student, and the forms in which the Reading Tutor logs them. For publications on successive versions and evaluations of the Reading Tutor [4], please see www.cs.cmu.edu/~listen.

[1] This work was supported by the National Science Foundation under Grant Number REC-9979894. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation or the official policies, either expressed or implied, of the sponsors or of the United States Government. We thank other members of Project LISTEN who contributed to this work; mySQL's developers; the students and educators at the schools where Reading Tutors recorded data; and the anonymous reviewers for making this paper better than we intended.

A Reading Tutor session consists of logging in followed by a series of stories, which the student and Reading Tutor take turns picking. Each story is a sequence of a few types of steps: assisted reading, writing, listening, and picking. The Reading Tutor may insert a preview activity beforehand, and/or a review afterward, also built out of the same types of steps.

The Reading Tutor's output is textual, graphical, and audio. It displays text on the computer screen, sentence by sentence, for the child to read aloud. It gives various sorts of help on a word or sentence by playing human or synthesized speech, with graphical cues such as highlighting words as it speaks them.

The Reading Tutor inputs the student's speech, mouse clicks, and keyboard presses. It uses a less than perfectly accurate speech recognizer to produce a time-aligned transcript of each student utterance.

1.2 Data recorded by the Reading Tutor

The Reading Tutor logs information in various forms. Although other tutorial dialogue systems may be organized very differently, we suspect that they may encounter some of the same problems we identify, such as proliferation of log entry types, the need to identify a few key control points, and the need to identify the appropriate level(s) of detail and aggregation at which to represent and analyze tutorial interactions.

The .wav file for each utterance contains the digitized speech. Its "hypothesis" file contains the speech recognizer's time-aligned transcript, one word per line, showing its start and end time as offsets in centiseconds from the beginning of the utterance:

```
BY 227 249
11 250 278
```

Utterance files are a simple representation, both human-readable and conducive to automated analysis, especially since this representation has remained stable for years. However, they are incomplete, capturing only what students said and what the recognizer "heard," not what else the student or Reading Tutor did. The "sentence" file shows the alignment of this hypothesis against the sentence text, using –1 to indicate that a sentence word had no hypothesis word aligned against it:

```
121     -1      -1
divided -1      -1
by      227     249
11      250     278
is      -1      -1
11      -1      -1
```

The Reading Tutor records tutorial interactions in detailed logs, starting a new log each time it is launched. Log files are tens of megabytes in size and many thousands of lines long. Each line is generated by a "logprint" utility function that records a sequential event number, a severity level, a timestamp, the number of digitized speech samples recorded so far, the name of the function that logged this event, and an event description whose form and content depend on the type of event logged and is often much longer than in the examples, which were chosen for their brevity. For example, this pair of lines in the log indicates that an utterance has ended and been captured:

```
16466, Notice,  "Tue Apr 10 12:30:20.387 2001", 10763200,
"CListener::FinalizeUtterance", "EndUtterance"

16467, Notice,  "Tue Apr 10 12:30:20.417 2001", 10763200,
"CCapture::WriteWaveFile(int)", "Wrote File:
d:\\listen\\cd\\Tue-Sep-19-23-44-58.093-2000\\Capture\\fAT6-6-1994-08-
01\\dec-fAT6-6-1994-08-01-Apr10-01-12-30-14-902.wav"
```

Over years of development, the Reading Tutor code has accumulated over a thousand calls to logprint for various purposes, including debugging and performance tuning. They record not only externally observable events, but also internal decisions at various levels of control. For example, suppose the student clicks on the word "cat." The Reading Tutor first computes the types of help it can give on this word, such as speaking the word, sounding

it out phoneme by phoneme, and so forth. From this set it chooses to give, say, a rhyming hint. From its table of rhymes, it randomly chooses the word "bat," and verifies that it has a recording of "bat." Then it queues a sequence of audio and graphical actions to say "rhymes with," display the word "bat" beneath the word "cat," and say "bat." The log file records this sequence of actions as separate events, but does not explicitly link them together as a single abstract event of the form "give help of type *h* on word *w* from time *t1* to *t2*." One reason for this is the difficulty in logging intervals rather than single events. Obviously a log message cannot be generated at time *t1* that also provides the endtime. Logging the event at time *t2* would destroy the chronological ordering of the log files. Since the log files were designed to be human readable, this ordering is an important property.

Although a detailed record can be useful for debugging, it is impractical to write scripts to parse and analyze a thousand different types of log entries. Over time we have reorganized the code to instrument key control points. For example, all graphical and audio output goes through a single "player" object, and key tutorial events are logged by a single function named CUserHistory::RecordEvent.

1.3 Previous ways to view and analyze Reading Tutor data

The Reading Tutor's logs are hard for humans to read and for programs to analyze. We first tried to solve the readability problem by making the Reading Tutor generate summaries, whose level and purpose varied.

Reading Tutors connected to the Internet via each school's local area network with an automated script every night to ftp summary data back to Carnegie Mellon to be mirrored and archived. These summaries provided a human-readable view of what the student was doing in the session – mostly for us researchers, since teachers have little time to peruse such detail.

Between student sessions, each Reading Tutor displayed a summary to let us monitor its overall usage and reliability objectively, both for a given day and historically, rather than rely on reported teacher impressions. For example, the following summary was displayed on a Reading Tutor in a school lab:

> Today : 1 readers, median session 24 minutes, total 26 minutes, 0 program restarts
>
> Average (in 154 days used): 2 readers, median session 18 minutes, total 39.3 minutes, 2.9 program restarts.
>
> Last... start: 4/05/2001 5:03 AM; login: 4/05/2001 12:24 PM; story: 4/05/2001 12:24 PM; utterance: 4/05/2001 12:34 PM; logout: 4/05/2001 1:00 PM

The Reading Tutor displayed a student roster to monitor individual usage, facilitate scheduling, and indicate student performance. The roster showed how long each student had read that day, the student's name (altered in this paper to protect anonymity), his or her current level, number of distinct stories and words seen to date, and usage, e.g.:

24 min.	Sarah Reader	(level B) has read 145 stories and 1460 words, in 88 17.1 minute sessions from 10/23/2000 to 4/05/2001

Clicking on the student's name brought up the "Reading Journal" summary generated for that day's session, listing the sequence of activity steps in the session. For example, the April 5 session comprised 6 activities, totalling 12 steps, one of them described as follows:

> Reading Tutor chose "How To Make Cookies By Emily Mostow." (level B). Sarah has not finished this activity before.
>
> April 5, 2001 At 12:24:28 PM: Started reading "First, Get The Batter."
>
> New words: Put Ingredients Batter Oven
>
> April 5, 2001 At 12:26:52 PM: Finished this step after 2 minutes, with 5.8 words accepted per minute.

Summaries can help expose bugs concealed by more detailed descriptions. For example, browsing the first (1999) version of the Reading Journal quickly exposed a logic error that sometimes caused the Reading Tutor to pick stories at too high a level – a bug that appeared only after prolonged use, since it occurred only when the student had finished all the stories at a given level.

The summaries were generated as plaintext or HTML, and aimed at human viewing. For automated analysis, we wrote special-purpose perl scripts specific to particular research questions. For example, by analyzing hypothesis and sentence files we were able to measure total usage and the amount of rereading [6]. Other questions required scripts to analyze log files. These scripts were complex enough for us to doubt their correctness. We allayed some of these doubts (and found bugs to fix) by hand-analyzing randomly selected examples, but still did not entirely trust the scripts.

The Reading Tutor bases some of its decisions on the student's history. For example, it needs to keep track of which words and stories the student has seen before. For this purpose it maintains a database of previous events. Some of our previous analyses used this type of database. The database for a single Reading Tutor was considerably more selective than the detailed log files, but over the course of a year often grew to over 100 megabytes. Nevertheless, it lacked some information needed to view or analyze tutorial interactions in detail.

2 A database approach

A July 2000 talk by Dr. Alex Rudnicky on his session browser for the Communicator system [1], coupled with our desire to understand how students were using the Reading Tutor at schools, inspired the vision of a log viewer to display Reading Tutor interactions at multiple levels of detail. We report on work in progress toward this vision – bugs and all.

Our approach parses utterance and log files into a database, enabling us to answer research questions by writing concise database queries instead of complex perl scripts. In contrast to the summaries recorded by the Reading Tutor, the log viewer uses stored queries to generate views at different levels dynamically, allowing us to modify the view form and content. We now explain how we represent, populate, query, and view the database.

2.1 Database schema for Reading Tutor data

Figure 1 summarizes the entity relationship schema we developed to model data from the 33 Reading Tutors used during the 2000-2001 school year by hundreds of students at three elementary schools. Database schemas are important to get right, and this one took weeks to finalize. To edit the schema, we used Microsoft Visio™, which keeps track of fields and dependencies while providing a graphical view that lists each table's fields (not shown here) but suppresses other details such as the data type of each field.

The entities modeled in the schema range in grain size from schools down to individual word encounters. Arrows encode many-to-one relations. For example, each Reading Tutor was launched once a day, more if it crashed and was restarted; a school had multiple students; a launch had some number of student sessions.

We faced a challenge in figuring out how to model the elements of a session. One choice was to model a session as a sequence of story encounters, a story as a sequence of sentence encounters, and a sentence encounter as a sequence of utterances. This model matched a useful simplified view of a session as consisting of picking a series of stories to read, and it used appropriately different fields to describe stories, sentences, and utterances. For example, story tables had a field for the difficulty level of the story, sentence encounter tables had a field for when the student began reading this sentence, and utterance tables had a field for name of the wav file that contained the student's speech. However, this structure did not match the reality that stories were multi-step activities, that the Reading Tutor often inserted a teaching activity before a story, and/or a review afterwards, and that the Reading Tutor architecture treated each session as a uniform tree structure of such steps. Such a step tree would be easy to represent recursively in the schema, but would omit bona fide distinctions among stories, sentences, and utterances. How to reconcile these two models? We decided to encode both models in parallel, relating them by linking each sentence encounter to the step during which it occurred.

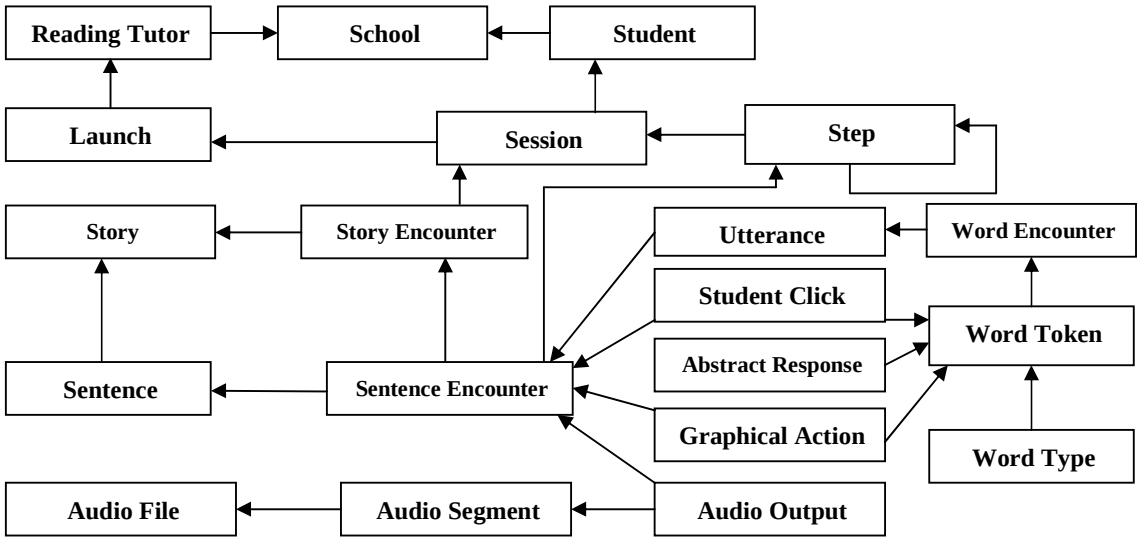

Figure 1: Summary of schema for database of Reading Tutor interactions

We used separate tables to represent mouse clicks, graphical actions, and Reading Tutor audio output, because they have different parameters. However, it is not very useful to view them separately, so we wound up generating merged tables that combined them into a unified sequential view.

We did not attempt to model the Reading Tutor's decision processes, only its interactions with students. We did include "Abstract Response" to model feedback as occurring either before the student began to read the sentence, after the Reading Tutor detected a mistake, as backchanneling (active listening such as "uh huh") or as praise. But just because the schema could represent abstract responses did not mean that the parser could always extract or infer them from the logs, or figure out which words they involved, so this table was incompletely populated. It was not always feasible to determine what word received this help, because help actions were not logged as abstract "molecules" such as "sound out CAT," but instead as individual "atomic" graphical actions and audio segments, such as "highlight the letter C" and "play phoneme_K.wav."

2.2 Populating the database

To populate the database, a perl script parsed through individual log files in a single forward pass. Each line was either disregarded as unimportant, used to maintain state that may be relevant for future records (e.g., user name, session start time), or was used to generate a particular record. The parser scanned for 33 different types of lines in a log – 13 different kinds of CUserHistory::RecordEvent lines, plus 20 other types of lines.

When the parser generated a record, it called a filer function to handle the communication with the database and create the necessary records in dependent tables, such as Sentence for a sentence encounter record. The filer returned an internal id from the database so that future related records, such as audio output within a sentence encounter, could refer to it.

For example, this line marked the start of a sentence encounter with the text "Dividing By 11":

```
15023,    Notice, "Tue    Apr    10    12:27:13.648    2001", 7776000,
"CParagraph::SetSentence", "Dividing By 11"
```

Accordingly, the script created a Sentence Encounter record with 12:27:13.648 as its start time, "Dividing By 11" as its text, and (based on state maintained from parsing previous lines) the internal (database) ids of the launch, session, story encounter, and step. 740 lines later, the script found a "coach_goes_forward" event, which enabled it to fill in the end time of the Sentence Encounter as 12:28:35.386.

Similarly, the parser started an Utterance record when it encountered a "user_begins_turn" event, and completed it when it encountered a pair of lines in the log showing that an utterance ended and was captured. The parser extracted from these lines the start time, end time, event number, and wav file for the utterance. From the state it maintained, it filled in the sentence encounter ID and its number of utterances so far. For each word in the sentence file, the parser created a Word Encounter record to specify which word, if any, was aligned against the sentence word, and computed its start and end times from the time offsets.

2.3 Implementation of the multi-level log viewer

Given the database, the viewer was straightforward to implement. We used MySQL Database-Server (see www.mysql.com) to serve the database, perl DBI packages to interface to the database, perl and CGI scripts to generate the views, and Apache Web-Server [8] to serve the views. All of these packages are free to download. To save time, we adopted a uniform tabular style rather than craft more specialized, aesthetic views, as we would if they were intended for teachers and not just us.

A link to a view is encoded as a call to the script that generates that view. For example, clicking the link http://logviewer.cmu.edu:9876/cgi-bin/storyList.pl?session_id=8562 calls the script storyList.pl to list the story encounters for session number 8562. The script executes a database query to retrieve them:

```
select story.story_id, title, level, file_path, story_encounter_id,
start_time, end_time, type_desc, student_level, byte_offset,
event_number, initiative, sms, ems
   from type_description ,story, story_encounter
   where story_encounter.story_id = story.story_id and
      story_encounter.session_id = $sessionid and
      type_description.type_id = story_encounter.exit_through
   order by start_time, sms
```

The script outputs an HTML table with a row for each record returned, embedding any links to other views.

2.4 Views: what the browser shows

In general, the log viewer generates a view as a list of records in an HTML table, with one row per value, and column headers at the top. One or more fields in each row may contain clickable links to more detailed views. To protect student anonymity, we restrict access and omit or alter names in our examples.

The highest-level view lists the (three in this database) values of School, with columns for school name, location, number of Reading Tutors, and number of students. Clicking on a school name brings up a table of Reading Tutors at that school, with columns for machine name, description, and number of launches. Clicking on a machine name brings up a chronologically ordered table with one row for each launch of the Reading Tutor on that machine. Each row includes the date and time of the launch, which version of the Reading Tutor was launched, the number of sessions during that launch, and a link to the log file for the launch. Clicking on a session count links to the list of sessions, and so on.

Clicking on a session's story count, step count, or student name brings up a table of stories read, session steps, or student information, respectively. For example, Figure 2 lists stories read in one session. Each row shows the story encounter's start time, end time (omitted here to save space), duration, number of sentences read, total number of sentences in the story, title, reader level, how the encounter ended, story level, and who picked the story.

This example revealed two instructive problems. A bug in the populating script inflated the number of sentences in the story to include previews and reviews. Second, the "select_response" value for how the last story encounter ended indicates that the log was missing expected information. From the log, it appears that the user timed out, but no Pause event was recorded. As these examples illustrate, a database populated from log files not designed for that purpose can be informative but buggy or incomplete.

Start Time	Dura tion	Num Sent Encount	Num Sen tences	Title	Level	Exit Through	Story Level	Initiativ e
04-05-2001 12:24:25.673	00:02:26.641	6	40	How to Make Cookies by Emily Mostow.	B	user_reaches_ end_of _activity	B	tutor_ initiative
04-05-2001 12:28:14.892	00:03:14.961	14	56	One, two,	A	user_reaches_ end_of _activity	A	student_ initiative
04-05-2001 12:31:34.159	00:02:44.817	5	112	Pretty Mouse by Maud Keary	B	select _response	B	tutor_ initiative

Figure 2: Table of activies for a session

Figure 3 shows the table of sentence encounters for the story "How to Make Cookies by Emily Mostow" (written by the first author's then-8-year-old daughter). The first two records come from preview activities that introduced the words "oven" and "batter," which the student was encountering for the first time in the Reading Tutor. The fifth record shows that the sentence "Then put it in the oven" took 33 seconds, with 2 utterances and 3 other actions, totaling 5 – few enough to list here, which is why we chose this example.

SentenceEncounter List					
StartTime	Duration	Num Actions	Num Utterances	Total Action	SentenceStr
04-05-2001 12:24:25.693	00:00:01.412	3	0	0	OVEN
04-05-2001 12:24:27.105	00:00:01.542	3	0	0	BATTER
04-05-2001 12:24:28.677	00:00:44.23	47	4	51	First get the batter
04-05-2001 12:25:12.700	00:00:24.886	20	4	24	Next put all the ingredients in
04-05-2001 12:25:37.857	00:00:33.908	3	2	5	Then put it in the oven
04-05-2001 12:26:11.765	00:00:40.539	3	3	6	Last eat them

Figure 3: Table of sentence encounters for the story "How to Make Cookies by Emily Mostow"

Figure 4 shows what happened during this encounter. First the Reading Tutor decided to give some preemptive assistance, though exactly what is not specified. 9 seconds later it prompted the student by saying "to get help click, on a word." The two utterances turned out to consist of the words "then put," followed by off-task speech. Finally the student clicked the *Go* button to go on to the next sentence.

Besides starting at the list of schools and browsing downward to more detailed views, we wanted to view specific entities found by queries. We therefore provided a more direct form of access by inputting the entity's database id. The same integer represents different ids in different tables, so the user must also select the type of table – of schools, Reading Tutors, launches, sessions, story encounters, sentence encounters, or utterances.

Start Time	End Time	Action	Description	AudioFilePath
04-05-2001 12:25:38.87	00-00-0000 00:00:00.0	Abstract Response	preemptive help	NONE
04-05-2001 12:25:47.681	00-00-0000 00:00:00.-1	Audio	NONE	d:\listen\data\sounds\phrases_to_get_ help_click_on_a_word.wav
04-05-2001 12:25:48.302	04-05-2001 12:26:00.129	Utterance	NONE	Click
04-05-2001 12:26:01.481	04-05-2001 12:26:11.575	Utterance	NONE	Click
04-05-2001 12:26:11.565	00-00-0000 00:00:00.-1	Click	user_goes_forward	NONE

Figure 4: Table of actions and utterances for the sentence encounter "Then put it in the oven"

3 Using the database to answer research questions

We are using the database both to replicate previous studies [3] rapidly and to answer new research questions [5], thanks to the (relative) ease of constructing queries and validating their correctness. For statistical analysis we use SPSS's ability to import data from the database, and our SQL client (urSQL)'s ability to export results into Excel.

3.1 Constructing queries

To answer a research question, we formulate it as an SQL query. As a simple example, were students likelier to back out of a story if the Reading Tutor chose it than if they did? This query counts how often students backed out of stories, disaggregated by who chose:

```
select se.initiative, count(*)
from student_click sc, story_encounter se, sentence_encounter sen
where
   sc.start_time = se.end_time and
   sc.type_id=8 and
   sc.sentence_encounter_id = sen.sentence_encounter_id and
   sen.story_encounter_id = se.story_encounter_id
group by se.initiative
```

The basic logic (in the `where` condition) is to find story encounters that ended when the student clicked *Back* (click type 8) out of a sentence and this click occurs at the same time a story ended. One problem could arise if a student clicked back at the same time someone else finished a story on a different computer. This case should not count as backing out of a story. To avoid this miscount, the `where` clause also specifies that the click must occur in a particular sentence in a story that ends at the same time as the click. The first line specifies what data to collect – the initiative (who chose the story) and how many items met the condition.

Item	Initiative	count(*)
1	(null)	140
2	student_initiative	703
3	tutor_initiative	2457

Table 1: Results of query to determine when students backed out of stories

The query results in Table 1 (based on about 400 students) support a Yes answer to the research question: the Reading Tutor chose the story in 2457 cases where the student backed out, the student chose in 703, and who chose was not specified in 140 cases. Null values may be deliberate for introductory tutorial "stories" presented automatically to newly enrolled students, or may be caused by bugs in generating or parsing the logs. They seem inevitable in a large database, but if rare enough do not preclude informative analysis, as this example illustrates.

3.2 Using the viewer to debug and improve queries

Queries are excellent tools for generating summary results, but are less powerful for examining individual cases. For example, to discover that not all the *null* initiatives were due to introductory tutorials, we used a query to find instances of stories with null initiatives. Then we used the viewer to examine their text and tutorial interactions. This combination of techniques was useful on several occasions.

For example, to analyze how much time students spent waiting for the Reading Tutor to respond, we developed a query to compute the delay from the last word a student says in a sentence to the first word s/he says in the next sentence. This query is 75 lines long because it requires several steps: finding the last word of a sentence and the time it was uttered, finding the time the student uttered the first word of the next sentence, ensuring that both sentences are in the same story, etc. After debugging each step, we ran the full query. By sorting the resulting delays, we found some too long to be believable. The table of results identified the story encounters where they occurred. By using the viewer to browse these story encounters, we found the bug: between some sentence encounters were activities where students were supposed to write, which the query erroneously included as part of the delay. It is important to note that it is not necessary to create a new view for each research question. For example, the view shown in Figure 3 was used to find the flaws in the query to compute how long students were waiting. However, this view was designed before we knew that we would be conducting this analysis, and could be used to verify other queries. The same tactic of viewing outliers traced supposedly hours-long sessions to a bug in filling in missing end times.

An additional benefit of the log viewer is that by presenting student-computer interactions in a more understandable form, it makes it easier for people with incomplete knowledge of the project to take part in data analysis. Our project had a near perfect split between those people who understood how the Reading Tutor worked and those people who could write SQL queries. The viewer allowed those unfamiliar with the tutor to perform sanity checks on their queries (as in the case of student writing activities, mentioned above). People who were less familiar with SQL used the viewer to examine unlikely query results and to find glitches in the database. This dichotomy of project members' knowledge is not unique to Project LISTEN; finding some means to work around this gap is very helpful.

3.3 Benefits of using a database

One benefit of using the database is the ease of getting summary information from the database. We have had project meetings where questions were raised and immediately addressed by a quickly written query (e.g. "How many times did the Reading Tutor provide each type of help?"). Although is it slower to get detailed information about each student, rather than a summary, from the database, it is comparable to or better than the prior technique of using perl scripts. Using queries, a complex analysis of how long students delayed before beginning to pronounce a word took 3 weeks to go from an idea to writing up results. This is comparable to the amount of time it took to write the perl scripts. However, the perl scripts were developed by someone who had been working on Project LISTEN for significantly longer (2 years) than the person who was developing the queries (6 months). Also, the work using queries analyzed how these delays related to students' performance on paper tests [2], so the investigation was somewhat more complex than the original analysis [3].

The comparison of SQL vs. perl is not quite a fair one, as the difference in ease of use has less to do with the languages than with the data each of them processes. SQL queries manipulate a structured database that we took time to set up, while the perl scripts had to work with low-level log files. To create the database, we had to debug a set of perl scripts. This task was time intensive, but only had to be done once. Its constant cost is amortized over all of the analyses performed. So for investigating a small set of research questions a database might not be worth the cost of setting up, but for more open-ended investigations it is.

4 Conclusions

We detailed a database approach to view and analyze multimodal tutorial dialogue, and how we applied it to Reading Tutor data. We now summarize its caveats, then its benefits.

It was hard to develop a good schema and useful views, especially for pre-existing logs that lack some desired information, at least in easy-to-extract form. We plan to make the Reading Tutor generate the database records in real time, or at least more parseable logs.

The database must be robust to tutor crashes and bugs. For example, when a crash ends a log prematurely, the end time of events in progress must be filled in. Parsing the logs exposed some Reading Tutor bugs, such as assigning the same filename to two utterances, which must not be allowed to corrupt the database.

Populating the database took weeks for our data, with 2.4 million word encounters. In the 2000-01 school year, 33 Reading Tutors each recorded hundreds of logs, typically tens of megabytes long, with thousands of utterance files. To avoid duplicating data, a long populating process must be robust to stops and restarts.

Although the database takes long to design and construct, it pays off in queries much shorter than perl scripts, because they are expressed more declaratively. Database technology absorbs much of the complexity of searching and assembling data. When necessary, we speed up queries by adding appropriate indices, but that type of optimization is easier and less bug-prone than rewriting conventional procedures to speed them up.

Views package certain types of queries in a reusable, understandable form, easier to use than querying the database directly – especially for views that integrate multiple tables, and for users more fluent at clicking on links than at formulating SQL queries. It is hard to design views both concise and detailed enough to be useful. Some of our views list hundreds of records – too many to fit on one screen or peruse easily. Views should summarize lower-level details in informative aggregate form, for example, durations and counts of utterances and actions in a sentence encounter. Queries make such aggregation easier, less bug-prone, and more flexible than in procedural tutor code.

We use queries both to answer statistical questions by aggregating over lots of data, and to find examples of particular phenomena, such as outlier values. We use the viewer to inspect such examples in detail, finding bugs or unexpected cases that refine the question. Finally, simply browsing our data at multiple levels often exposes interesting phenomena.

References (also see www.cs.cmu.edu/~listen)

1. Bennett, C. and Rudnicky, A.I.. The Carnegie Mellon Communicator Corpus. Proceedings of the *7th International Conference on Spoken Language Processing (ICSLP2002)*. 2002 (submitted)
2. Jia, P., Beck, J.E. and Mostow, J.. Can a Reading Tutor that Listens use Inter-word Latency to (better) Assess a Student's Reading Ability? Proceedings of the *ITS 2002 Workshop on Creating Valid Diagnostic Assessments*. 2002
3. Mostow, J. and Aist, G.. The Sounds of Silence: Towards Automated Evaluation of Student Learning in a Reading Tutor that Listens. Proceedings of the *Fourteenth National Conference on Artificial Intelligence (AAAI-97)*. p. 355-361. 1997
4. Mostow, J. and Aist, G., Evaluating tutors that listen: An overview of Project LISTEN, in *Smart Machines in Education*, K. Forbus and P. Feltovich, Editors. 2001.
5. Mostow, J., Aist, G., Beck, J., Chalasani, R., Cuneo, A., Jia, P. and Kadaru, K.. A La Recherche du Temps Perdu, or As Time Goes By: Where does the time go in a Reading Tutor that listens? Proceedings of the *Sixth International Conference on Intelligent Tutoring Systems (ITS'2002)*. 2002
6. Mostow, J., Aist, G., Huang, C., Junker, B., Kennedy, R., Lan, H., IV, D.L., O'Connor, R., Tassone, R., Tobin, B., and Wierman, A., 4-Month Evaluation of a Learner-controlled Reading Tutor that Listens, in *Speech Technology for Language Learning*, P. DeCloque and M. Holland, Editors. in press.
7. Mostow, J., Huang, C. and Tobin, B., Pause the Video: Quick but quantitative expert evaluation of tutorial choices in a Reading Tutor that listens, in *Artificial Intelligence in Education: AI-ED in the Wired and Wireless Future*, J.D. Moore, C.L. Redfield, and W.L. Johnson, Editors. 2001. p. 343-353.
8. Wrightson, K., *Apache Server 2.0: A Beginner's Guide*. Network Professional's Library. 2001, Berkeley: Osborne.