Volt GUI Console: A Graphical User Interface for Volt Distributed Transaction Processing Database

Rob Kallman

Brown University

rob.kallman@navy.mil

Submitted as partial fulfillment of a Master of Science (ScM) in Computer Science from Brown University, Spring 2009.

Approved By: _____ on 14 May 2009

Stanley Zdonik

Professor of Computer Science

Introduction

In this paper, I will describe my work on Volt DB and the graphical user interface I developed for it as my final project for a Master of Science in Computer Science at Brown University.

1.1 Summary

Volt is a high-performance, memory-resident, distributed, shared-nothing, transaction processing system built from scratch by a team comprised of students, professors, and professionals from Brown and Yale Universities, MIT, and Vertica Inc. The majority of the work thus far has been appropriately focused on the back-end and on the optimization of this benchmark bar-raising database system. The existence of a comprehensive graphical user interface has not been designed, though it is needed to extend the system beyond the command-line. We set out to develop a GUI that enhances demonstrations and allows developers and researchers to watch a 'playback' of a complete workload for study and exploration.

1.2 Organization of Paper

In the first section, I will give a brief overview of Volt and some of the more important characteristics. Next, I will describe my top-level design goals. Following that, I will dissect the design and the specifics of the GUI that I designed and built with the majority of the focus on my recent work on the Trace Simulator. I will also describe the existing visualizer package that I investigated before deciding to implement my own from scratch. Finally, I will describe some of the setbacks I encountered (with suggestions for completing the implementation), and some ideas for future improvement and enhancement of the entire interface.

2 Introduction to Volt

2.1 Background

Volt, previously called H-Store, is an open-source project built on the premise that the 'one-size fits all' legacy databases started in the 1970's are being significantly outperformed by specialized databases built from the ground up for their vertical market (e.g., data warehouses, OLAP, OLTP); thus, they are unburdened by unnecessary overhead. When many of these vintage databases were first developed and architected, hardware was much more expensive, yet not as robust as it is now. Current enterprise systems can store their entire database in main memory among clusters of multi-core machines; yet access time between main memory and secondary storage has not had the same performance increases as has the storage capacity. With such an abundance of primary storage, the database bottleneck can shift from being disk-bound to being network-traffic-bound or even CPU-bound. It is not inconceivable to imagine an

enterprise's computing grid having more than enough processor cores and main memory to process and hold even the largest on-line transactional data stores; ergo, the need for a DBMS designed specifically for short-lived, predictable, and fast transaction processing is obvious. To strengthen the argument, Stonebraker, et. al. showed that on the New Order transaction of TPC-C, only 6.8% of the execution time of the transaction using a legacy system is actually 'useful' work. The remaining time is spent in overhead such as locking (16.3%), latching (14.2%), logging (11.9%), and buffer management (34.6%) all of which are unnecessary in Volt. [3].

By using timestamps to maintain serializability and because all transactions are single-threaded, the majority of transactions have no danger of cascading rollbacks, and because procedures are stored, it is free of user stalls; therefore, the use of locks and the latching of data structures are largely unnecessary. For general transactions that could have concurrency problems, we are investigating variants of a two-phase locking mechanism for these special cases. We can avoid the need for logs by maintaining replica partitions on *k* sites (also known as *k*-safety). If a partition fails, the duplicate will immediately rollover without requiring any damage control or downtime. Furthermore, the expectation is that institutions will be able to add additional sites without restarting the system, so that when a site is restored, it can be seamlessly brought back online. Buffer management is needed by legacy systems in order to access pages through a buffer pool, while memory-resident databases are not burdened by this overhead [3]. Harizopoulos, et. al. showed that stripping the database of these unnecessary features resulted in a substantial speed increase from about 640 transactions per second up to an amazing 12700 transactions per second [3].

Because of the way Volt accesses workload, transactions must be written and finalized at compile time (except for run-time parameters). The optimal transactions (and the fastest) are executed in isolation on a single-node or partition. Such *single-site* transactions will guarantee serializability because the single-thread can execute such a query without relying on external reads and minimizes the amount of messaging. Also, because the stored procedure is accomplished on a single-site, there is no danger of cascading rollbacks should a transaction abort nor is any type of lock required. If the partitioning scheme of the database is such that it precludes single-site execution of a transaction, then the next preferable type is a *one-shot* transaction of a stored procedure. This type of transaction is sent to 1-*n* sites for execution, but each site can perform its portion of the workload without requiring intermediate results from other sites. The individual nodes can process the transaction as if it were single-sited and does not need knowledge of what the other sites are doing. If a transaction does not fit in the aforementioned categories, Volt can still execute the general transaction, but at much greater cost. Recently, developers have implemented ad-hoc queries to the capabilities, again, at the expense of performance.

2.2 Design Goals

My overarching design goal was to deliver a graphical user interface for the Volt that was useful, efficient and user-friendly. I wanted to use familiar interface objects such as buttons, combo boxes, and text fields so that the data entry and the operation of the

interface would be fairly intuitive. The majority of the users will be very comfortable with computers, though they won't want to be hindered by a clunky GUI.

3 Volt GUI Console Architecture

3.1 Introduction

At VLDB '08, our team demonstrated Volt against a version of the TPC-C benchmark, when the nascent program could only run on a server and on a single client. Volt is primarily ran through the command-line, but for the demo, we implemented a rudimentary graphical user interface which allowed conference attendees a chance to experiment with TPC-C after changing runtime parameters and settings. Then, the GUI had the capability to allow the user to select a stored procedure from the TPC-C catalog, insert parameters (or generate random ones), execute the procedure, and view the results once completed [1]. It also allowed the user to inspect the catalog as a tree structure, and had a panel for table results. Unfortunately, TPC-C does not have very interactive returns, so the table results were not very dramatic. The focus of the demonstration was on high-performance, not on user-friendliness. Once the 'Execute' button was pressed, the procedure would finish in mere seconds without much fanfare or visual excitement. An additional console displayed runtime measured in transactions per second. There is a need for better representation. As the system matures and as features are added, academics and professionals will want to see the benefits of various partitioning schemes and to ensure serializability is maintained due to the very optimistic concurrency control scheme and much-reduced logging implementation. Also, Volt has the potential for gaining much attention in academia and improved visuals will enhance our demonstrations at conferences.

In section 2, I outlined the basic architectural concept of Volt; a high-performance, specialized OLTP designed and built from scratch to excel in this domain. Since vital features are still being designed and implemented, the majority of software engineering focus has been on the execution engine, the catalog types, the query planner, supporting multiple sites, and the coding of the actual instructions making the system operate; hence, little attention has been focused on the user interface or the user experience.

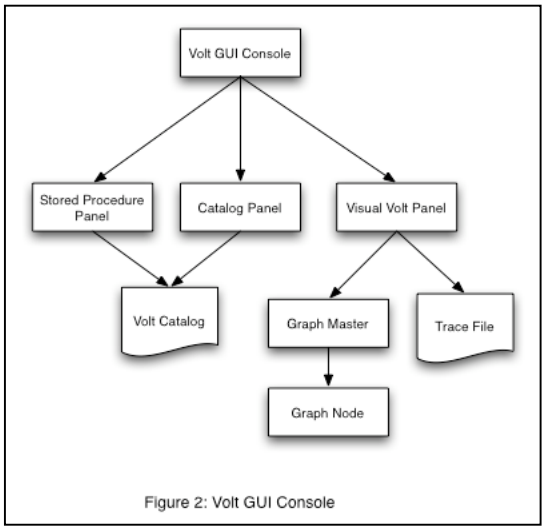

Figure 2: Volt GUI Console

The graphical user interface that we designed and built for VLDB '08 was focused on showing off our prototype and on demonstrating the execution of Stored Procedures from the industry-standard TPC-C benchmark with few additional functions. The main GUI, then known as ConsoleVLDB, consisted of three tabbed panels: Stored Procedures, Table Results, and Catalog Tree Data.

The Stored Procedure panel was the primary interface for VLDB '08. The user

could select any of the stored procedures that were listed in the catalog, enter parameters, and execute the chosen procedure. The Table results would display the return of the aforementioned transaction in tabular form (if applicable). Lastly, the Catalog Tree panel displayed the catalog hierarchy in a traditional interactive tree format. In the time since that conference, we have designed and implemented a fourth section: the Visual Volt panel.

3.2 Panel Specifics and Code

The GUI and associated classes were written in Java SE 6 and were tested on a MacBook Pro (2.6 GHz dual-core Intel processor with 4 GB RAM), running Mac OS X 10.5.7.

3.2.1 Volt GUI Console

The Volt GUI Console is the main program that runs the heavyweight Java Swing container that contains all of its lightweight subordinate panels. Prior to launching the GUI, a Volt server must be launched to accept query requests. The server code requires that the number of execution sites be entered at run time. The main console implements the program menu and initiates the instance of a Volt client and connects it to the aforementioned server. One necessary improvement is adding code that launches the server at runtime in addition to the client(s). Throughout this project, I used the older version of Volt that only supported one site, though multi-sites are now being tested as part of the latest build.

3.2.2 Stored Procedure Panel

The Stored Procedure panel (Figure 1 in the Appendix) loads the parameterized catalog (passed at run-time) and stores each statement in a Hash Map (keyed on the procedure name). These procedures populate a combo box from which a user can select any of the procedures that are in the catalog. The listener for the combo box loads the procedure and then updates the panel to display the specific parameter fields required for the query once a procedure is selected. This dynamic feature allows users to see exactly which data types are necessary. For demonstration purposes, the user can optionally select random values and the GUI controller will fill the fields with parameters of the correct type (e.g., long, string, timestamp). Once the procedure is loaded and the parameters entered, the transaction can be executed. Since the GUI is connected to an instance of the Volt server, the procedure and its parameters are packaged and sent to the execution engine provided the parameter values are of the correct Volt data type. After the procedure is finished, the return table is displayed in an imbedded table on the panel.

3.2.3 Catalog Tree Panel

The Catalog Tree panel (Figure 2 in Appendix) provides an interface to show the various entities contained in the Volt Catalog. After loading the current catalog (the same one throughout the entire console instance), the Catalog Tree controller piece iterates through each line in the catalog and organizes the entities into a common tree structure. The user can then quickly expand or contract the branches to inspect the

contents of the DDL objects (e.g., tables) and/or Stored Procedures (with their associated parameters, fragments, etc.). Though not as interesting as the other pieces of the GUI, the Catalog Tree Panel is a useful tool and can be used to understand the way the Volt catalog system is organized hierarchically in a graphical format. If the format of the Volt catalog changes as the system matures, this class will need to be modified because it uses loops that depend on specific names of the catalog types and may not recognize attributes in the event they change. The catalog is loaded upon creation and the GUI can only switch catalogs during a restart (provided the catalog path is revised in the 'Load Catalog' method).

3.2.4 Visual Volt Panel

The Trace Simulation Panel contains the workload trace visualizer. The next section addresses this portion of the GUI in much more detail.

4 Visual Volt Panel

4.1 Introduction

In short, the Visual Volt Panel (Figure 3 in Appendix) was the crux of my work over the past 5 months. Its purpose is to show a visual representation of a trace workload executing on a simulated multi-sited instance of Volt. I spent the majority of my time investigating and experimenting with different visualizer packages, developing the back-end trace mechanism, and figuring out how to translate plan fragments into a graphical object and then animating and synchronizing them as the workload comes in. This portion of the GUI currently does not require an instance of Volt; since it uses an external file (e.g., trace file), it is self-contained and runs in isolation.

4.2 Description

By far the most complex panel of the graphic user interface, the Visual Volt Panel contains features from the other panels but does much more. My goal was to add appealing graphics to complement the execution of the transactions in such a way that demonstrates the various fragments as they are dispatched by the coordinator to the appropriate sites for execution. The version of Volt that I used to develop this simulator only supported a single fragment for each query in a one-to-one relationship; herein laid the requirement for a trace backend to simulate them.

Should this be visualized as a single-site (one coordinator and one client), it would be neither worthy of investigation, nor very interesting; the fragment would be sent out by the coordinator and the site would execute it. When the site completes executing the plan fragment (which contains the entire plan), the coordinator will send out another one. Only when the distributed Volt that can execute Plan Fragments on 1- *n* sites will there be significant benefits of this system. In the interim, I modified a workload trace system developed by my colleague, Andy Pavlo, to introduce multiple plan fragments to each query statement.

After a trace file is loaded and conditioned for the simulator, the program can execute. Each query is divided into fragments, which are represented as colored circles

(hereafter referred to as fragment icons) grouped by their query, starting at the coordinator. At the starting timestamp, the coordinator dispatches the fragments to their respective nodes. If the node is available, the Plan Fragment is executed and the fragment icon will pass to the output of the graph and become part of the result. If the node is still executing an older fragment, the newly arrived fragment must wait for the other to complete (as per the serializable order of fragments). This sets up an execution queue that should be displayed on the GUI. On the side of the animation window is an output showing the current transaction and query whose fragments are on display. If one particular site is not partitioned correctly, it may be evident by consistently having such a queue.

As of now, the GUI back-end sets up four sites as the default. When the panel is created, the network is viewed as a rooted tree with the coordinator represented as the root. To get things started, the user must load a trace file from the system directory (or the default). After the file is loaded, the user can run the simulator. Each query creates a fragment thread that is 'dispatched' from the coordinator in serialized fashion as per the starting timestamp stored in the trace file. When the fragment arrives at its appropriate execution site, it will stay there until the stop timestamp.

4.3 Code

The network graph and the animation are built onto a JPanel that is part of the Visual Volt Panel. In order to ensure concurrency control of the GUI is maintained, I extended SwingWorker, a java class implemented with Java SE6. As the name implies, the SwingWorker creates a thread for execution that is not event-driven. This allows an event to be triggered, but the process does not block other processes during its routine. This worker class was used for the method that loads the catalog, the method that updates the textual information, and the animation controller, called 'GraphMaster.' Having never used SwingWorker, I gained a tremendous amount of experience implementing and tweaking this concurrency mechanism.

For the animation of the plan fragments, I considered using Sprite animation, which uses a controller to create and control discrete animated objects referred to as Sprites. After substantial effort, I did not have success with this animation paradigm. As an alternative, I extended the Thread class in my GraphMaster. Each instance of a plan fragment creates a thread that can be individually executed and disposed of. The efficiency of this structure, coupled with the aforementioned SwingWorkers, allow the GUI to smoothly stay updated. My intention was to synchronize the animation of the plan fragments with the update of the iterable trace data to simulate an actual workload. If this had worked out, a query would be simulated as follows: when the query is active, the coordinator will compare system time with the timestamp. When they are equal, the coordinator will 'execute' the query. The text data on the left of the screen will show which Transaction is current, which query is running, and which fragments are sent out. Then, animated ovals, representing the correct number of fragments, will 'travel' from the coordinator (root) node to its respective assigned site node. The node will switch its flag to 'isBusy' preventing other fragments from entering. It will then compare system time with the stop timestamp, and when they are equal, will destroy the fragment, update the text data in the GUI, and will set 'isBusy' to false so that another fragment can be

processed. The process repeats until the entire workload is finished. Due to time constraints, I was unable to coordinate the processes so that the trace iteration would match the fragment animation; I will come back to this issue later in the paper.

5 Workload Trace Back-end

5.1 Introduction

Earlier, I wrote that the version of Volt on which this GUI sits on only supported one single client. In order to make the simulator more interesting, I needed to simulate that the system had multiple sites running different types of transactions: single-sited, one-shot, and general. The Workload Trace mechanism is the interim measure that I used until an actual stream of multi-fragmented queries is supported and implemented.

5.2 Description

Workload trace files must be created as a separate event than the execution of the graphical user interface because this file is used to initialize the simulator. The trace file contains all of the Transaction Traces, Query Traces, and Plan Fragment Traces created from the given catalog. All three types of traces have the following attributes:

- Transaction ID
- Start Timestamp
- Stop Timestamp
- Parameters
- Catalog Guid

In addition to the attributes they inherit from the Abstract Trace Element, the Transaction Trace element has an Array List of Query Traces, the Query Trace element has a statement id and a list of plan fragment traces, and the Plan Fragment Trace hooks the actual Volt Plan Fragment out of the catalog.

When I created the Trace File used for the simulator, I duplicated the Plan Fragment Trace (and its Plan Fragment) a random number of times ($1 - n$) for each query to emulate a multi-node stream. In the current version, the start and stop timestamps are pre-contrived but the simulator should be able to easily handle actual timestamps. The plan fragments appear at the coordinator site at their start time, and exit their nodes at their stop times, however long they take. When I created the trace file, I randomly assigned an execution time between 10 and 200 milliseconds.

5.3 Code

When I started modifying the workload trace, I was given a trace file that had all of the catalog's Transaction Traces and its Query Traces. I had to implement a way to add the Plan Fragment traces to the existing trace file. The significant

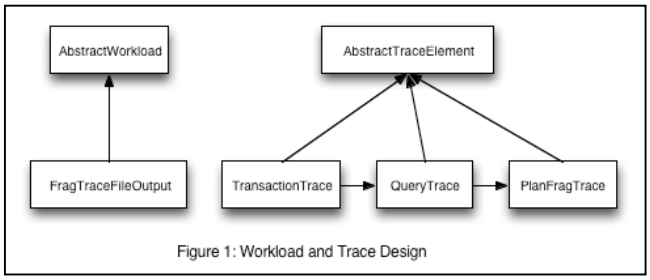

Figure 1: Workload and Trace Design

classes needed to create the trace files are: Abstract Workload and Frag Trace File Output (both of which are needed to load and create trace files), Abstract Trace Element, Transaction Trace, Query Trace, and Plan Frag Trace (all needed to populate the workload out of the trace file). The main method in Frag Trace File Output calls its 'Load' method, which opens a Buffered Reader to read in each line of the trace file. It then uses a series of nested loops to add the catalog elements to the trace elements. It loops through every transaction, then every query, and finally, every fragment. The entries in the trace files are stored as JSON entries, which are vastly more straightforward than straight text because each entry can be treated like an object and can have attributes associated with it. We use a parser called 'loadFromJSONObject' that converts the JSON script into Strings. The strings are used to create the trace elements. If the trace file needs to be written, then the 'writeToTraceFile must be called. This method translates the trace elements back into JSON objects and then writes them into the trace file. This conditioning only needs to be done once per catalog. This workload trace back-end is necessary until we can develop a way to stream real-time execution data.

6 Benefits of a Simulation

In addition to serving as a useful demonstration tool, the simulation panel could be used and even modified to keep current with the latest releases of Volt. At the time of this writing, this OLTP will now run transactions on multiple sites and will support one-shot and general transactions in addition to the single-site transactions it previously supported (though the multi-site feature was only recently added). Another benefit is that academics and professionals can study the console text after the routine runs for anomalies, inspection and investigation. By perusing some of the transaction print outs, researchers have an opportunity to verify the correct nodes are executing the fragments and that the execution time is consistent.

After the database designer is implemented, the administrator could review run-times to determine if the designer automatically partitioned the tables appropriately. Excessive runtimes could illuminate a mistake in the assignment of the transaction type or could illuminate a logical error in the partitioning scheme. In the meantime, the simulator gives researchers an opportunity to witness the execution at various speeds while implementing new modules and/or features. The most exciting benefit of the simulator is the visualization of the transaction execution. It is one thing for someone to watch command-line scrolling text as queries are run; it is another thing entirely to watch a simulation of the same process with graphics. It would be a trivial matter to loop the simulator for future conference participation. The appealing animation is sure to draw a crowd and could invite further interest in this ongoing research project.

7 Visualization Details

The primary data structure that I implemented for the animation is a class called Graph Node. As the name implies, it is the object that stores the positional data necessary for animation. It is represents the Plan Fragment and is only a temporary object. The node has a one-to-one relationship with a Plan Fragment Trace, which is created when the trace file is loaded. All of the fragment traces for a particular query are

stored in an array list. During the run-phase, the queries are iterated and the fragment icons are created and the nodes are released from the originating node that represents the workload coordinator. As of now, this structure is divided into two separate classes: PlanFragTrace, which holds the execution data, and GraphNode, which contains the spatial data for the animation. In the future, these two classes could be merged to streamline the code. Since the Graph nodes' (x,y) position is available, the fragment icons travel from their initial position (x0, y0) to their execution node. I calculate the path they take using the slope-intercept equation of a line.

The 'GraphMaster' class is responsible for drawing the network and for animating the execution of the fragments. This class extends java.lang.Thread so that the animation can play without blocking the other portions of the GUI. The JPanel that holds the GraphMaster uses the SwingWorker class for concurrency. These two classes solved several problems I had with keeping the animation running. Before extending them, I could not get the animation subroutine out of the event thread. Once the 'run' button fired, the animation would freeze the entire GUI because it would block the other procedures. Similarly, I also used the SwingWorker class for the workload iterator that updates the textual information in the panel. Again, without the use of concurrency control, the fields would not update because the iterator wouldn't trigger the paintComponent.

8 JUNG – An Alternate Visualizer Library

When I initially wrote the design of the visualizer, I anticipated using an open-source visualization package to implement the animation. After spending quite a bit of time weighing the different options and packages available, I found a library called JUNG 2.0. This module is very robust and mature and it is used to generate sophisticated network diagrams. It also implements several well-known graph algorithms (e.g., Djikstra's Shortest Path, Edmond-Karp Min-Cut/Max-Flow). Thought the documentation for the library left much to be desired, I was finally able to create the graph I wanted through experimentation. To create a graph in JUNG, the programmer needs to create vertices and edges. Both of these are completely customizable and can basically be anything. Once you add vertices to a graph and connect them with edges just by providing the to/from vertices.

Given that description, JUNG seemed to be the perfect fit as a base for my animation. I planned on using the JUNG-created graph as the background onto which I would overlay my transaction execution. Unfortunately, JUNG's strength is in its algorithms, not in its display. The user has NO control over HOW the graph is drawn. Vertices could be drawn in one order during one execution of the program and in another order during another. The way it works is that the graph sends vertex and edge data to a layout manager that is responsible for arranging the most efficient representation of the graph. It does not have access to the (x,y) coordinates of its vertices; consequently, I could not establish, with any degree of certainty, where to set my coordinator and where my sites would be located. Finally, it dawned on me that it would be much easier to create my own graph visualizer since I did not need the sophistication (and the complexity) of the JUNG library. I could not find a suitable package that would simplify

my task of animating my fragments from point A to B. At any rate, I abandoned JUNG in favor of a 'roll your own' approach, which ended up being a good decision.

9 Setbacks

In spite of dedicating months of work and research to this project, I was unable to complete all of my design goals. I spent so much time trying to force JUNG to work for me, I did not allow myself enough time to recover and fully implement my homegrown solution. At the point of this writing, the Visual Volt Panel does the following:

- A trace file can be loaded into the GUI (and ONLY .trace files)
- Once the file is loaded, the simulation can be started.
- The traces get created and are iterated in a series of nested loops.
- The GUI gets updated for every 1-*n* fragment that is contained in the trace file.
- A graph with *n* nodes is drawn on the GUI (Node count can only be changed at compile time, but it is a simple fix to pass it in as a runtime parameter).
- The correct number of fragment icons are emitted from the coordinator node and sends them to its execution node.

I was unable to implement the following:

- The fragments get iterated serially. In actuality, all fragments of a query should be dispatched simultaneously rather than one-at-a-time.
- The animation and the iterator are not synchronized so that the user can see which query is being executed.
- I didn't implement a collision-detection algorithm to notify an incoming fragment icon that its site is busy, therefore, it goes into the site regardless of availability.
- The simulator does not take either start or stop timestamps into consideration yet; it blindly animates the fragment icons.

10 Possibilities for Future Enhancement

Some of the enhancements that can be developed by students and researchers include the following unimplemented features (several of which were discussed in the previous sections):

- The application could allow the user to capture actual workload streams and funnel them directly into a trace file (in the interim) that can be visualized rather than using a 'canned' file of contrived data.
- It could allow the user to stream real-time workload execution into the Visualizer and be able to dial in a particular speed at which to animate.
- Instead of only accepting special 'trace' files, optionally allowing system logs to be the source for the Visualizer.
- The continuity and connectivity between the various GUI panels could be enhanced to simplify execution. Currently, the software has to duplicate much of the effort, as does the user; for example, the GUI loads the catalog several times when only once could be necessary.

- During the animation, if a fragment is delayed, the output could explain the reason for the queue (e.g., coordinator or site stalls).
- Provide the ability for the user to save the console output into an external file.

11 Conclusion

After spending 18-months on the H-Store/Volt project team, I feel like a contributing member. I am proud that my graphic user interface was displayed and used during our demonstration at VLDB '08; from what I heard, our booth was one of the 'must-see' exhibits at the conference. This project enabled me to greatly expand my understanding of databases in general, and Volt specifically. In order to capture the execution into a visual form, I had to delve into the way fragments are going to be executed. Also, my skills as a Java programmer were significantly improved because I used libraries and packages that were completely foreign to me beforehand. . Finally, the project served as an excellent capstone for my Master's program in Brown University's Computer Science Department. My membership in the Data Management group has revealed the importance of the field and the ubiquitous nature of its subject.

On my first day of classes over two years ago, Stan Zdonik said that Database technology is the best of all Computer Science disciplines. According to him, it involves aspects from every area of the field: algorithmic, theoretic, programmatic, and systematic. How right he was; every class I took during the degree program complemented my knowledge of data management holistically. From the introductory Database Management class, through Algorithm design, Combinatorial Optimization, Data Warehousing, and Complex Data Processing seminars, I cannot quantify the amount of knowledge that I gained. I am excited about the future possibilities that are inherent in the next generation of databases as we move into a specialty vertical market and away from the 'one-size' fits all paradigm of yesterday.

REFERENCES

[1] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, D. Abadi. "H-Store: A High Performance, Distributed Main Memory Transaction Processing System". In *VLDB '08*, pages 1496-1499, 2008.

[2] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. "The End Of An Architectural Era: (It's Time For A Complete Rewrite)". In *VLDB '07*, pages 1150–1160, 2007.

[3] S. Harizopoulos, D. Abadi, S. Madden, M. Stonebraker. "OLTP Through the Looking Glass, and What We Found There". In SIGMOD '08, pages 981-992, 2008.

Appendix

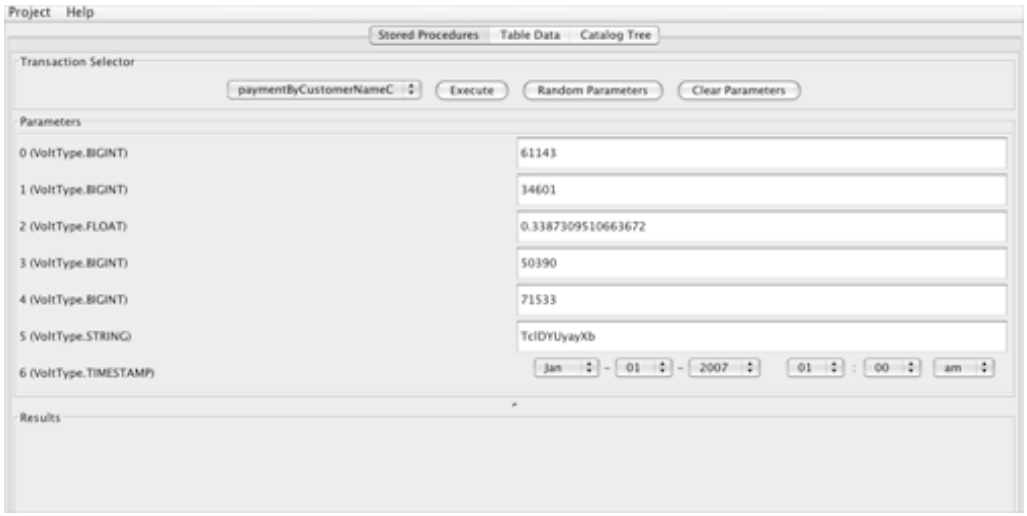

Figure 1: Stored Procedure Panel

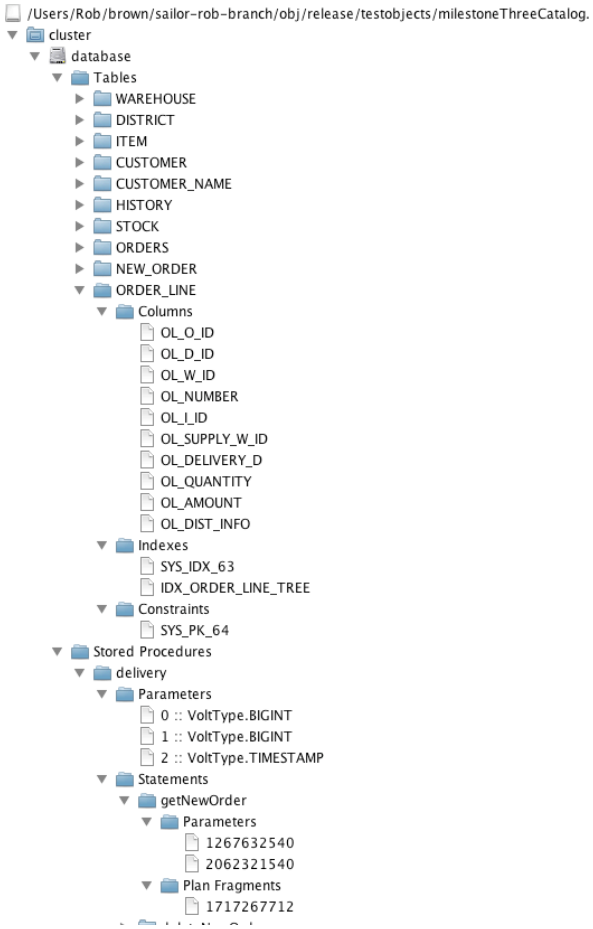

Figure 2: Catalog Panel

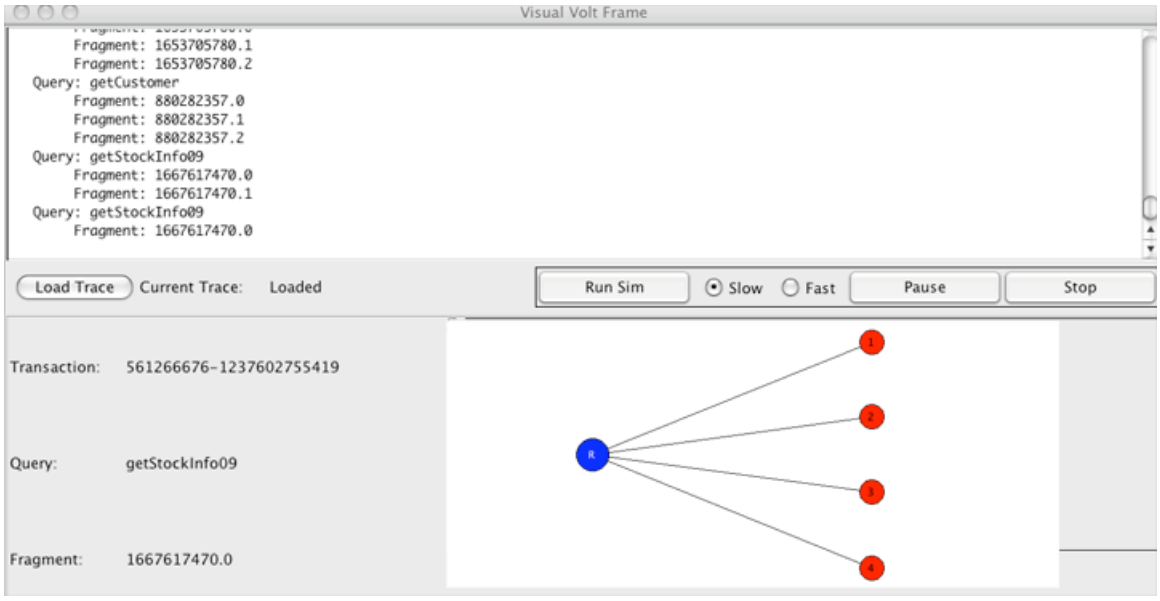

Figure 3: Visual Volt Panel