An Automatic Source Code Generation Tool for OLTP Database Benchmarks

Zhe Zhang
zhe@cs.brown.edu

Department of Computer Science, Brown University

1 Introduction

H-Store [3] is a research OLTP DBMS project being developed in collaboration by MIT, Brown University, Yale University and HP Labs. Based on assumptions that long-running transactions are not needed for high-end OLTP applications and OLTP DBMS can do without disks [11] , H-Store has some features that do not exist in current market-leading DMBS products. For example, H-Store is designed without multi-threading support in the execution engine. In order to test how well H-Store performs under various OLTP workload environments and therefore explore performance bottlenecks in both physical design and system implementation, H-Store researchers use benchmarks modeled after real-world OLTP environments. These benchmarks run on top of H-Store and collect statistics about the system's performance. A laborious part of benchmark implementation for H-Store is that the researcher needs to spend a lot of time in writing redundant code. In this paper, we present a novel OLTP benchmark source code generation tool that simplifies the process of creating a new OLTP benchmark. Using the new system, the researcher only needs to specify the key benchmark characteristics using an intuitive GUI. The reduction of the time spent on redundant work saves time and allows researchers to deploy new benchmarks more rapidly.

This paper is organized as followed: Section 2 introduces the high-level characteristics that are common in OLTP benchmarks and describes two well-known benchmarks. Section 3 demonstrates the process of creating an example benchmark using the source code generation tool and gives a high-level description of how the tool is implemented. Section 4 describes future work of the source code generation tool.

2 OLTP Benchmark

A database benchmark is a special kind of application that is used to evaluate a database system's performance. Due to the fact that there are two types of databases (OLTP database and OLAP database), database benchmarks are divided into OLTP benchmarks and OLAP benchmarks correspondingly. OLTP (On-Line Transaction Processing) databases are designed to run real-time business operations, whose execution unit is a transaction that can be invoked by thousands of concurrent users to insert, update and delete rows. OLTP benchmarks are designed to evaluate the performance of databases by measuring their *transactional throughput*, usually expressed as the number of transactions executed per second. OLAP (On-Line Analytical Processing) databases are designed to support large multi-dimensional analytical queries of read-only data from a data warehouse. OLAP benchmarks are designed to evaluate the performance of databases by measuring their *analytical throughput* [4], usually expressed as the total processing time divided by the number of queries processed.

Next, we delve deeper into OLTP benchmarks from the perspectives of benchmark specification and benchmark execution stages. We then study two well-known benchmarks that have been implemented for H-Store. Finally we summarize the redundant work in H-Store benchmark implementation, from which our motivation of designing an automatic source code generation tool for OLTP benchmarks originates.

2.1 Benchmark Specification

In general, the specification of an OLTP benchmark includes: (1) a database schema, (2) the database population constraints, (3) the transaction workloads, (4) the transaction execution frequencies and (5) the transaction parameter generation rules. These components are defined as follows and are referred to throughout the rest of the paper.

- **Database schema**: it describes the structure of a database using a formal language such as SQL. The elements defined in the schema include: tables, table columns, indexes, primary keys, foreign key dependencies and so on.
- **Database population constraints**: they describe the constraints to apply when populating (Stage 1 in Section 2.2) the tables of the database. A common constraint is the *table cardinality*, which specifies the number of rows to populate into each table. For every column in every table, the data used for population is generated according to its *column generation rule*, usually expressed in terms of the *data range* and the *probability distribution type*. For example, for an integer-typed column, its data range can be from 1 to 10000, and its probability distribution type can be uniform distribution.
- **Transaction workloads**: a *transaction workload* is a sequence of SQL operations (query, add, update and delete) that are applied to a set of tables (in the schema).
- **Transaction execution constraints**: they describe the constraints to apply when running the transactions. One common transaction execution constraint is the *transaction execution frequency*. For each transaction, the higher its transaction execution frequency is, the more times it will be selected by the transaction coordinator in the statistics collection stage (Section 2.2). A *transaction coordinator* is a program that randomly selects transactions to execute, one transaction at a time. Another common transaction execution constraint is the *transaction parameter generation rules*, which determine how the parameters for running the transactions are generated. The rules are usually expressed in the *data range* and the *probability distribution type*.

2.2 Benchmark Execution Stages

Typically, running an OLTP benchmark goes through the following stages:

1. **Database creation and population**: A data loader program is launched at this stage to create a database instance reflecting the database schema and populates each table in the database based on the database population constraints.
2. **Ramp-up**: Transactions are run against the database to warm up its cache, with no performance statistics recorded. The reason to discard the statistics at this stage is that a database with an empty cache runs slower than the normal cases where its cache is fully-loaded. Thus the statistics at this stage are not representative.
3. **Statistics collection**: After the cache is warmed up, the transaction coordinator begins to collect the statistics of running the transactions, such as the amount of the time it takes to execute each transaction and the number of times each transaction has been executed. The transaction coordinator also maintains the statistics of per-transaction OLTP throughput and overall OLTP throughput. After a specified length of time elapses, the transaction coordinator halts the system and displays the final throughput results.

2.3 TM1

The TM1 (Telecom One) benchmark is designed for telecommunication applications [9] and is widely used by network equipment providers as well as software and hardware vendors to compare the performance of different products [1]. The TM1 benchmark simulates a typical Home Location Register (HLR) database used by a mobile carrier.

The TM1 database schema is a simplified version of a real-world HLR application schema [9]. It stores

information about subscribers, such as their mobile phone numbers, current geographical locations of subscribers' mobile devices within the network, the services that they are subscribing and the privileges of accessing the services. There are four tables in the schema: Subscriber, Access_Info, Special_Facility and Call_Forwarding. The default size of the Subscriber table can range from 100,000 to 5,000,000 tuples. For the other three tables, their default sizes are proportional to that of the Subscriber table. For every column of every table, its probability distribution is uniform, within its own range.

There are seven pre-defined transactions run in TM1. 80% of them are read transactions and 20% are write transactions. For a given subscriber id, GET_SUBSCRIBER_DATA retrieves its profile; GET_NEW_DESTINATION retrieves its current call forwarding destination; GET_ACCESS_DATA retrieves its access validation data; UPDATE_SUBSCRIBER_DATA updates its service profile data; UPDATE_LOCATION updates subscriber location; INSERT_CALL_FORWARDING adds a new call forwarding record; DELETE_CALL_FORWARDING removes a call forwarding record.

Implementation of TM1 for H-Store includes TM1Loader, TM1Client (corresponding to the transaction coordinator in Section 2.1), seven transactions, and a schema definition file. Except for the schema definition file that comprises table-creation SQL statements, all components are written in Java and are built on top of the *H-Store Benchmark Framework* (a platform written in Java that supports the development and execution of H-Store benchmarks). Similar to the benchmark execution stages described in Section 2.2, TM1Loader creates an in-memory database instance (since H-Store is an in-memory database) and populates it with the generated data, and TM1Client selects transactions to execute, one at a time. Each transaction extends from VoltProcedure, the abstraction of a running H-Store transaction, and invokes SQL operation methods. The statistics of transaction throughput in TM1 is maintained by the H-Store Benchmark Framework.

2.4 TPC-E

The TPC-E benchmark [12] is introduced in 2007 by the Transaction Processing Performance Council, the organization that introduced the de facto industry-standard TPC-C [13] benchmark in 1992. Since its inception, TPC-E is gradually being accepted as the new industry-standard OLTP benchmark due to its benefits over TPC-C. For example, TPC-E is more close to today's business and academic model, more representative of today's schema and transaction complexity (TPC-C has 9 tables, 92 columns and 5 transactions while TPC-E has 33 tables, 188 columns and 10 transactions), and has significantly higher price-performance of benchmark-running [7, 6].

The 33 tables in the TPC-E schema are divided into four categories: Customer-related tables, Broker-related tables, Market-related tables and Dimension-related tables. The sizing rules of the tables are more complex than TM1: while in TM1 the cardinalities of all tables are proportional to the same table, in TPC-E, for some tables, their cardinalities are fixed; for some tables, their cardinalities are proportional to CUSTOMER table; for the other tables, their cardinalities are initially proportional to CUSTOMER table but will increase during benchmark run at a rate proportional to the transaction throughput rate.

The activities simulated in TPC-E transactions include: managers of brokerage houses generating reports on the current performance potential of various brokers; customers looking up their customer profiles and summaries of their overall standing based on current market values for all assets; customers tracking the daily trend of securities; customers conducting research on security prior to making trade-execution decisions; customers looking up the summaries of recent activities; brokerage houses processing the ticker-tape from the market exchange; trades looking up by customers or brokers; trades update by customers or brokers; security exchange by customers, brokers or authorized third-party; completing of stock market trades [14].

The implementation of TPC-E in H-Store is similar to TM1 in code structure, except for how the generation of data for table populating and parameters for transaction execution are implemented. While in TM1, the data generation algorithms are implemented directly in the loader, for TPC-E, the TPC provides official data generation code which is called via the Java Native Interface by the loader and the transaction coordinator respectively.

2.5 Redundant Work in OLTP Benchmark Implementations

Based on the descriptions of the benchmark specification, the benchmark execution stages and two example OLTP benchmarks, we summarize the redundant work that exist in (most) H-Store OLTP benchmark implementations as follows:

- **Creating the table instances**: an instance is created for each table in the database schema according to the benchmark specification. The table instances created will be populated by the data loader.
- **Implementing the data loader**: the data loader consumes a set of table instances and populates each table according to the its population constraint.
- **Describing the database population constraints**: the database population constraints should be described either in programs (a population constraint can be abstracted using a class) or data files (the loader can read the constraints by parsing the files).
- **Implementing the transaction workloads**: the user needs to implement the transaction workloads on top of H-Store SQL methods. A H-Store SQL method is a Java method that abstracts a SQL operation in H-Store.
- **Describing the transaction execution constraints**: like the database population constraints, the researcher needs to describe the transaction execution constraints either in programs or data files.
- **Implementing the transaction coordinator**: the transaction coordinator consumes a set of transaction workloads and executes them according to the transaction execution constraints.

3 Automating the generation of OLTP benchmarks

3.1 Motivation

Under the current benchmark implementation paradigm in H-Store, whenever a researcher wants to implement a new H-Store benchmark, he/she has to do the redundant work described in Section 2.5. Typically, a database researcher is more concerned about the design of benchmarks and the analysis of running the benchmarks. However, due to the overhead of the redundant work, it is possible that the researcher has to spend far more time in benchmark implementation than in benchmark design and analysis. For the researchers who need to frequently implement new benchmarks, it can be very painful to do the redundant work over and over again. Thus, the current benchmark implementation paradigm in H-Store impedes researchers' productivities in benchmark design and analysis.

Researchers' productivities can be improved if there exists a tool that enables them to generate benchmark implementations with less time spent on the redundant work. Thus, we have developed the H-Store OLTP Benchmark Source Code Generation Tool . Via the tool's GUI, the researcher specifies (1) the database schema, (2) the workload execution file and (3) the constraints of database population and transaction execution. Then the tool outputs the corresponded benchmark source code. The researcher then can use H-Store's built-in deploying tool to compile the generated benchmark source code and eventually run the benchmark.

3.2 Creating an example benchmark

This section provides an overview of the process of creating an example H-Store benchmark using the tool. In general there are three steps that the user must complete: (1) Set up project properties such as the project name and the project path. (2) Specify the database schema and the database population constraints. (3) Specify the workload trace file and the transaction execution constraints. We describe these steps in details as follows.

3.2.1 Initial configuration

In the initial configuration pane (see Figure 1), the user specifies the benchmark name, the benchmark package name and the project path. The benchmark name and the benchmark package name are needed by the H-Store Benchmark Framework to perform Java runtime reflection. The benchmark package name also

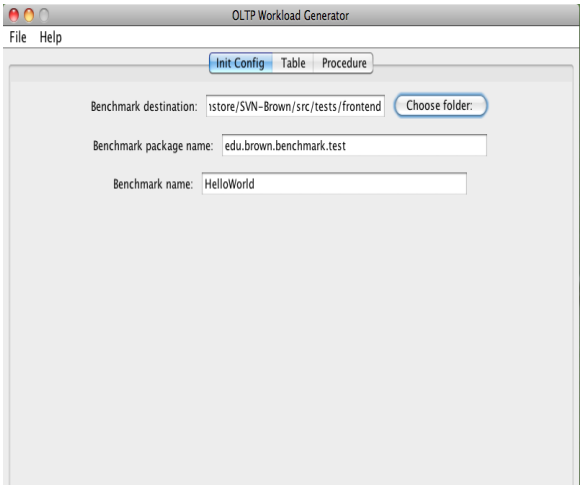

Figure 1: Initial benchmark configuration

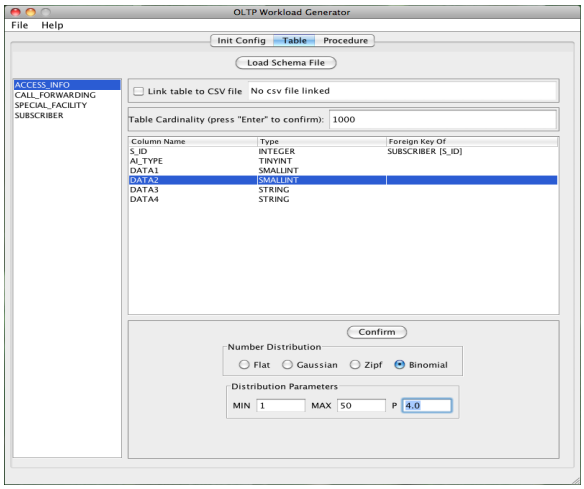

Figure 2: Database configuration

determines the declaration of the package names for all the generated Java classes in the benchmark. The project path specifies where the generated Java source files will be located in the local file system.

3.2.2 Database configuration

In the database configuration pane (see Figure 2), the user specifies the database schema by clicking the *Load Schema file* button and selecting the database schema file. After the schema file is chosen and parsed into its in-memory data structure, the list of table names in the schema are displayed on the left hand side of the pane. When clicking a table name in the list, the user can specify its table cardinality on the right hand side pane. Also shown on the right hand side pane are the list of columns in the table. For each column, the user needs to specify its column generation rule, in terms of the probability distribution type and the data range.

There are in total three types of column data: *number*, *string* and *timestamp*. Each data type has its own column generation rule and the tool takes care of the diversities by providing the correspondent input fields for each type. For numbers, the probability distribution type can be one of Flat distribution, Gaussian distribution, Zipfan distribution and Binomial distribution. Each distribution is expressed in terms of its data range and any additional parameters. For strings and timestamps, the probability distribution type is uniform distribution. The data range of strings is its length. The data range of timestamps is its date range.

Besides specifying the table cardinality and the column generation rules, another way of specifying how a table is to be loaded is by linking it to a csv file (see Figure 3) so that the data loader will read the data from the csv file to populate the table. Once a table is linked to csv file, the specification of its table cardinality and column generation rules are overriden.

Compared with the redundant work described in Section 2.5, the tool removes researchers' burden of (1) creating the table instances, (2) implementing the data loader and (3) describing the database population constraints.

3.2.3 Transaction configuration

In the transaction configuration pane (see Figure 4), the user specifies the transaction workload trace file by clicking the *Load workload trace file* button. A workload trace file (see Figure 6) logs the information of a set of executed transactions, such as the transaction names, the timestamps and the execution results. The H-Store infrastructure has the feature to generate such trace files when running OLTP workloads. After the work load trace file is chosen and parsed into its in-memory data structure, the list of transaction names are displayed on the left hand side pane. When clicking a transaction name in the list, the user can specify its transaction execution frequency on the right hand side pane. Also shown on the right hand side pane are the

Figure 3: Link table to csv file

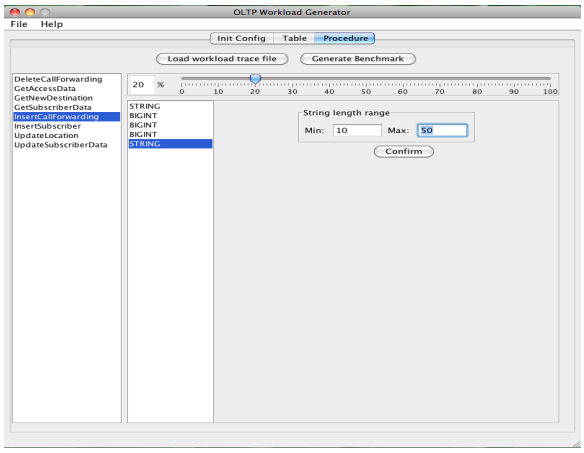

Figure 4: Transaction configuration

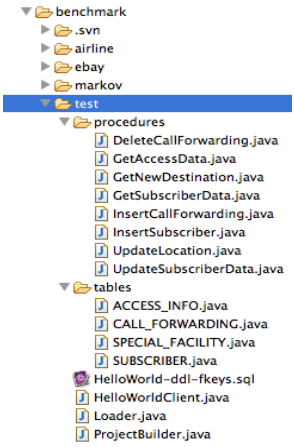

Figure 5: Generated benchmark code

list of parameters of the selected transaction. For each parameter, the user needs to specify its transaction parameter generation rules.

Compared with the redundant work described in Section 2.5, the tool removes researchers' burden of (1) implementing the transaction workloads, (2) implementing the transaction execution constraints and (3) implementing the transaction coordinator.

3.2.4 Benchmark generation

After the user completes specifying the initial project properties, the database configuration and the transaction configuration, the final step for the user is to click the *Generate Benchmark* button in the transaction configuration pane. Then the benchmark source code will be generated and written to the specified location in the local file system (see Figure 5). As can be seen from the figure, the generated source code includes: the data loader (see its source code in Appendix B), the transaction coordinator (with the file name HelloWorldClient.java), the project builder (providing book-keeping data to H-Store's built-in deployment tool), transactions (encoding the transaction execution constraints) and tables (encoding the table-creating statements and the database population constraints).

Then the user can use H-Store's built-in deploying tool to compile and run this benchmark. During the whole benchmark creation process, the researcher has done none of the redundant work specified in Section 2.5. No Java code needs to be written by the researcher.


```
{"ID":8,"CATALOG_NAME":"GetSubscriberData","START_TIMESTAMP":19070091645712,"STOP_TIMESTAMP":19070181627182,"ABORTED":false,"PARAMS":
[82485],"XACT_ID":"995684858-1249930612244","QUERIES":[{"ID":9,"CATALOG_NAME":"GetData","START_TIMESTAMP":19070179598782,"STOP_TIMESTAMP":
19070181186414,"ABORTED":false,"PARAMS":[82485],"BATCH_ID":0,"PROC_NAME":"GetSubscriberData"}]}
{"ID":10,"CATALOG_NAME":"GetAccessData","START_TIMESTAMP":19070186389100,"STOP_TIMESTAMP":19070186799487,"ABORTED":false,"PARAMS":
[46457,2],"XACT_ID":"17649447-1249930612339","QUERIES":[{"ID":11,"CATALOG_NAME":"GetData","START_TIMESTAMP":19070186444623,"STOP_TIMESTAMP":
19070186779163,"ABORTED":false,"PARAMS":[46457,2],"BATCH_ID":0,"PROC_NAME":"GetAccessData"}]}
{"ID":12,"CATALOG_NAME":"GetAccessData","START_TIMESTAMP":19070187040439,"STOP_TIMESTAMP":19070187306954,"ABORTED":false,"PARAMS":
[17622,4],"XACT_ID":"17649447-1249930612340","QUERIES":[{"ID":13,"CATALOG_NAME":"GetData","START_TIMESTAMP":19070187069354,"STOP_TIMESTAMP":
19070187291309,"ABORTED":false,"PARAMS":[17622,4],"BATCH_ID":0,"PROC_NAME":"GetAccessData"}]}
{"ID":14,"CATALOG_NAME":"InsertCallForwarding","START_TIMESTAMP":19070187536871,"STOP_TIMESTAMP":19070212195941,"ABORTED":false,"PARAMS":["000000000040321",
1,16,17,"000000000040321"],"XACT_ID":"1190265908-1249930612340","QUERIES":[{"ID":15,"CATALOG_NAME":"query1","START_TIMESTAMP":19070187574865,"STOP_TIMESTAMP":
19070203806746,"ABORTED":false,"PARAMS":["000000000040321"],"BATCH_ID":0,"PROC_NAME":"InsertCallForwarding"},{"ID":16,"CATALOG_NAME":"query2","START_TIMESTAMP":
19070203851584,"STOP_TIMESTAMP":19070211725141,"ABORTED":false,"PARAMS":[40321],"BATCH_ID":1,"PROC_NAME":"InsertCallForwarding"},{"ID":
17,"CATALOG_NAME":"check","START_TIMESTAMP":19070211758944,"STOP_TIMESTAMP":19070212017846,"ABORTED":false,"PARAMS":[40321,1,16],"BATCH_ID":
2,"PROC_NAME":"InsertCallForwarding"}]}
```


Figure 6: Part of a workload trace

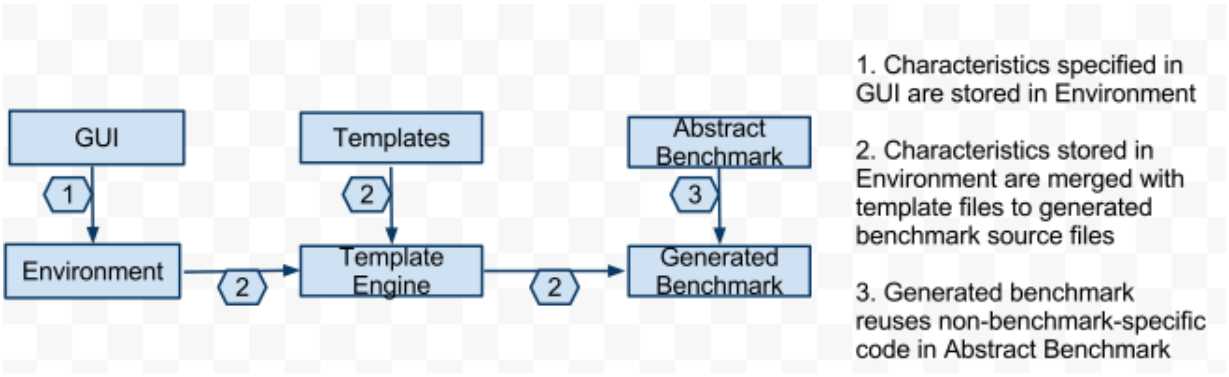

Figure 7: Archiecture of H-Store OLTP Benchmark Source Code Generation Tool

3.3 Implementation

The implementation of the tool is straightforward. As shown in Figure 7, it is composed of a front-end module and four back-end modules: *Environment*, *Template Engine*, *Templates* and *Abstract Benchmark*. The front-end module implements the GUI on top of the Java Swing/AWT API. When the user specifies the benchmark constraints in the GUI, the constraints get stored in the Environment module, which is essentially name-value pairs. When the user clicks the *Generate Benchmark* button, the constraints are read from the Environment and fed to the Template Engine. The Template Engine, built on top of the Apache Velocity Engine (a Java-based template engine) [2], binds the variables in the pre-defined velocity template files to the constraints and generates the corresponded Java source files (see sample code in Appendix). Every generated source file extends from a class in Abstract Benchmark: it reuses (by invoking) its non-constraint-related methods such as table loading algorithm and overrides (by hardcoding into) its constraint-related methods such as getTableCardinality(). See in Appendix B for an example of how benchmark constraints are hardcoded in constraint-related methods.

4 Conclusions and Future Work

We have introduced a GUI-based source code generation tool that enables benchmark designers to implement H-Store benchmarks without needing to write Java code. More work can be done to change the fact that the user cannot save a benchmark specification to a file (and later on load it to GUI). The functionality of saving/loading benchmark specification can be useful if the user cannot complete specifying all benchmark characteristics at one time (for example, the benchmark to be implemented is too complex so the user specifies a new part in the benchmark whenever he/she has figured it out).

References

[1] AMD64 Multi-Core Computing Platforms and solidDB enable consoli- dated approach to subscriber databases and related applications. `http://www.amd.com/us/Documents/43237-A_AMD64_and_solidDB_White_paper_PDF.pdf`.

[2] Apache Velcity Engine. `http://velocity.apache.org/engine/releases/velocity-1.6.2/`.

[3] H-Store Project. `http://db.cs.yale.edu/hstore/`.

[4] OLAP Benchmark Study . `http://www.olapcouncil.org/research/spec1.htm`.

[5] TM1 Benchmark: measure performance of database in critical Telco applications! `http://virtual.vtt.fi/virtual/proj1/projects/merlin/expsolutions/tm1_benchmark_flyer.pdf`.

[6] Top Ten TPC-C by Price/Performance. `http://www.tpc.org/tpcc/results/tpcc_price_perf_results.asp`.

[7] Top Ten TPC-E by Price/Performance. `http://www.tpc.org/tpce/results/tpce_price_perf_results.asp`.

[8] Velocity Template Language(VTL). http://velocity.apache.org/engine/releases/velocity-1.5/user-guide.html#velocity_template_language_vtl:_an_introduction.

[9] N. Gupta. Enabling High Performance HLR Solutions. http://www.sun.com/products-n-solutions/hw/networking/atca/HLR-on-ATCA-v2-Final.pdf, April 2006.

[10] Solid Information Technology. Telecom One (TM1) Benchmark Description, March 2005.

[11] 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 *Proceedings of the 33rd international conference on Very large data bases (VLDB '07)*, pages 1150–1160. VLDB Endowment, 2007.

[12] The Transaction Processing Council. TPC-E Benchmark (Draft Revision 0.32.2g). http://www.tpc.org/tpce/, July 2006.

[13] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/, June 2007.

[14] T.-E. S. C. I. S. Trish Hogan and T. Group. Overview of TPC Benchmark E: The Next Generation of OLTP Benchmarks. ftp://ftp.software.ibm.com/eserver/benchmarks/wp_TPC-E_Benchmark_022307.pdf.

A Loader template file

```
public class Loader extends AbstractLoader
{

    private AbstractTable[] m_tables;

    public Loader(String[] args)
    {
        super(args);
        m_tables = new AbstractTable[$tblNames.size()];
        #set( $idx = 0 )
        #foreach( $tblName in $tblNames )
          m_tables[$idx] = new $tblName();
        #set( $idx = $idx + 1 )
        #end
    }

    @Override
    protected AbstractTable[] getAllTables()
    {
        return m_tables;
    }

    public static void main(String[] args)
    {
        org.voltdb.benchmark.ClientMain.main(Loader.class, args, true);
    }

    @Override
    protected String getSchemaFileName()
    {
        return $schemaFileName;
    }
}
```

B Sample source code generated from Loader template file

```
public class Loader extends AbstractLoader
{

    private AbstractTable[] m_tables;

    public Loader(String[] args)
    {
        super(args);
        m_tables = new AbstractTable[4];
        m_tables[0] = new SUBSCRIBER();
        m_tables[1] = new SPECIAL_FACILITY();
        m_tables[2] = new ACCESS_INFO();
        m_tables[3] = new CALL_FORWARDING();

    }

    @Override
```

```
    protected AbstractTable[] getAllTables()
    {
        return m_tables;
    }

    public static void main(String[] args)
    {
        org.voltdb.benchmark.ClientMain.main(Loader.class, args, true);
    }

    @Override
    protected String getSchemaFileName()
    {
        return "HelloWorld-ddl.sql";
    }
}
```