# TPC-C support in Lancet

Morales Gonzalez Mikael
*EFPL, Switzerland*
*December 20, 2019*

## Abstract

LANCET is a self-correcting tool designed to measure the open-loop tail latency of µs-scale datacenter applications with high fan-in connection patterns [2]. LANCET is self-correcting as it relies on online statistical tests to determine situations in which tail latency cannot be accurately measured from a statistical perspective, including situations where the workload configuration, the client infrastructure, or the application itself does not allow it.

LANCET currently supports Memcached, Redis and HTTP as application protocols. We present the implementation of another application protocol, namely TPC-C [7]. LANCET now completely supports TPC-C, allowing it to generate transactions following the TPC-C standards. A generic implementation of an R2P2 [3] server has also been implemented, which currently supports SQLite as RDBMS, but can be easily extended to support multiple database engines.

With the generic client and R2P2 server implementation of TPC-C, LANCET is now able to run a complete TPC-C benchmark remotely. In addition, LANCET is able to leverage the implementation of agents supporting NIC-based hardware timestamping, which when available, is used by LANCET to measure RPCs end-to-end latency on top of the TPC-C benchmark.

Our experiment shows LANCET accurately reports the latency distribution while running the TPC-C benchmark.

We also show the complementary CDF of each transaction using SQLite as RDBMS.

## 1 Introduction

Web-scale datacenters rely on the decomposition of extensive queries into smaller sub-queries that are processed individually in thousands of interconnected servers. Today's technology allows µs-scale interactions between these interconnected servers that internally communicate using RPCs.

The number of components involved in a single query has significantly increased and the extensive use of high fan-in, high fan-out patterns with the emergence of µs-scale interactions requires to focus on tail-latency.

Tail latency is harder to measure compared to others metrics such as throughput. Indeed, tail-latency depends on various number of factors beyond the workload itself. These factors, for instance, include, the overheads in the tool being used to measure the tail-latency and the experiment methodology.

LANCET is a self-correcting latency measurement tool designed to measure, in a statistically sound manner, the end-to-end tail latency of remote procedure calls in a testing environment [2]. LANCET is self-correcting as it relies on on-line statistical tests to determine situations in which tail latency cannot be accurately measured.

LANCET relies on state-of-the-art, hardware-based measurement techniques that combine NIC timestamping in hardware, using exclusively the standard Linux API [1], and user level matching of packets to RPCs allowing to measure end-to-end tail latency of remote procedure calls.

LANCET also support various application protocols, including Memcached, Redis, HTTP and more recently TPC-C.

TPC-C is an on-line transaction processing (OLTP) benchmark [7] representing any industry that must manage, sell, or distributed a product or service.

TPC-C simulates an environment where a population of terminal operators executes transactions against a database. Multiple transaction types are available, including entering and delivering orders, recording payments, checking the status of orders and monitoring the level of the stock at the warehouses.

This report focuses on the implementation of TPC-C on top of LANCET and on the implementation of a generic R2P2 [3] server running SQLite [5] to run the benchmark.

Our experiment shows LANCET accurately reports the latency distribution while running the TPC-C benchmark.

We also show the complementary CDF of each transaction, using SQLite as RDBMS, and evaluate current bottlenecks and possible improvements.

We discuss the necessary background (§2) and discuss the implementation of TPC-C (§3). We then analyze an execu-

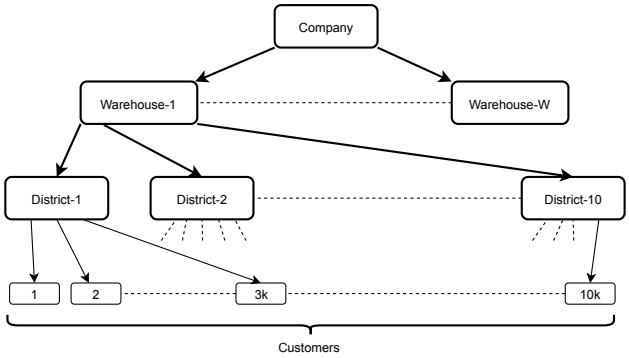


Figure 1: Illustration of the warehouse, district, and customer hierarchy of TPC-C's business environment. Each warehouse covers 10 districts. Each district serves 3'000 customers. All warehouses maintain stocks for the 100'000 items sold by the Company.

tion of the TPC-C benchmark on top of LANCET (§4), and conclude (§5).

## 2 Background

### 2.1 TPC-C

TPC-C is an on-line transaction processing (OLTP) benchmark. The benchmark is a mixture of read-only and update intensive transactions simulating the activities found in complex OLTP applications environments.

The performance metric reported by TPC-C is a "business throughput" measuring the number of orders processed per minute. Multiple transactions are used in TPC-C, more specifically five transactions are available: payment, order-status, delivery, stock-level and new-order transactions. In TPC-C, throughput is defined as how many new-order transactions per minute a system generates while the system is executing the other four transactions types. The performance metric is express in transactions-per-minute-C (tpmC).

It is important to note that the TPC-C benchmark is not bound to any database management system (DBMS). Any implementation of a database system that provides similar functionalities as any commercially available DBMS can be used to run the benchmark.

TPC-C portrays a company that is a wholesale supplier with a number of distributed sales districts and warehouses. The number of warehouses can be defined by the client before running the benchmark (default to 1). All the warehouses maintain stocks for the 100'000 items sold by the company. Figure 1 illustrates the warehouse, district and customer hierarchy of TPC-C's environment.

The benchmark emulates action from the customers such as placing a new order or requesting the status of an existing order. An order is composed, on average of 10 order lines (items). The company's system is also used to enter payments

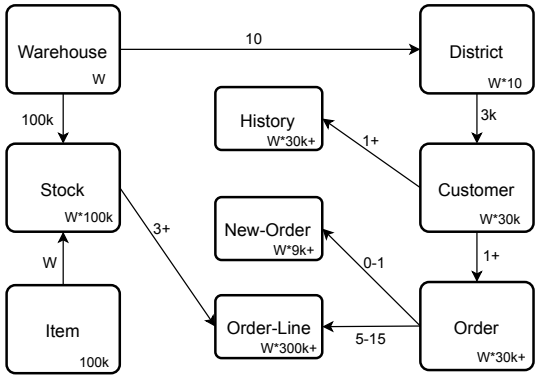


Figure 2: Entity-relationship diagram of the TPC-C database. The numbers in the entity blocks represent the cardinality of the table. The number next to the relationship arrows represent the average number of children per parent and the plus (+) symbol is used to illustrate that this number can vary as rows are added or deleted.

from customers, process orders for delivery and examine stock levels to identify potential supply shortage.

### 2.2 TPC-C Architecture

The database consist of nine tables. The characteristics and relationships among these tables are defined in Figure 2. The database is defined using tables and rows for simplicity, but any storage layout can be used.

More information about the tables layout and their attributes can be found in the TPC-C standard specification [8].

### 2.3 TPC-C Transactions

The TPC-C benchmark generates five different type of transactions, where each of them has a different frequency of execution. The ACID properties of transactions must never be violated when running the benchmark.

**New-Order transaction:** The new-order transaction is the backbone of the workload. It consists of entering a complete order through a single database transaction. This transaction represents 45% of the transactions executed in the benchmark and is a mid-weight, read-write transaction.

A fixed 1% of the new-order transactions are chosen to simulate user entry errors and require rolling back update transactions.

It is also used to compute the "business throughout", which is defined as how many new-order transactions per minute a system generates.

The input required to execute this transaction consists of: (i) The home warehouse number, (ii) the district number, (iii) the customer number chosen non-uniformly, (iv) the number of item in the order.

Then for each item in the order we must define: (i) the item number non-uniformly, (ii) the supplying warehouse number and (iii) the quantity.

**Payment transaction:** The payment transaction updates the customer's balance and reflects the payment on the district and warehouse sales statistics. This transaction represents 43% of the transactions executed in the benchmark and is a light-weight, read-write transaction.

In addition, this transaction includes non-primary key access to the CUSTOMER table in 60% of the times using the customer last name.

The input required to execute this transaction consists of: (i) The warehouse number, (ii) the district number, (iii) the payment amount, (iv) the payment date, (v) the customer selected 60% of the time by its last name and 40% by its number and (vi) the customer resident warehouse which is 85% of the time the home warehouse and 15% a remote warehouse.

**Order-Status transaction:** The order-status transaction queries the status of a customer's last order. This transaction represents 4% if the transactions executed in the benchmark and is a mid-weight, read-only transaction.

In addition, this transaction includes non-primary key access to the CUSTOMER table in 60% of the times using the customer last name.

The input required to execute this transaction consists of: (i) The warehouse number, (ii) the district number and (iii) the customer selected 60% of the time by its last name and 40% by its number.

**Delivery transaction:** The delivery transaction consists of processing a batch of 10 new orders, not yet delivered. Each order delivered in full by doing a read-write database transaction. The transaction comprised of one, up to 10 database transactions. This transaction represents 4% of the transactions executed in the benchmark.

The input required to execute this transaction consists of: (i) The warehouse number, (ii) the carrier number and (iii) the delivery date.

**Stock-Level transaction:** The stock-level transaction determines the number of recently sold items that have a stock level below a specified threshold. This transaction represents 4% of the transactions executed in the benchmark and is a heavy read-only transaction.

The input required to execute this transaction consists of: (i) The warehouse number, (ii) the district number and (iii) the threshold of minimum quantity in stock.

## 2.4 SQLite

SQLite is a relational database management system (RDBMS) contained in a C library [5]. In contrast with other database management system, SQLite is not a client-server database engine, rather it is small self-contained SQL database engine that ensures ACID properties and implements most of the SQL standards.

SQLite can be easily integrated into a project using the SQLite Amalgamation [6], which concatenate over 100 separate source files into a single large C file. The amalgamation contains everything an application needs to embed SQLite, making it easy to deploy. And because all code is in a single translation unit, compilers can do better optimizations resulting in machine code that is faster.

Typically, SQLite stores the entire database as a single cross platform file. For this project, we decided to store the entire database in memory given that SQLite offers this possibility and that the memory available in the machines used is high.

# 3 Implementation

## 3.1 TPC-C

TPC-C intensively relies on random generators. Random numbers and characters are constantly generated, either to define the attributes needed for a transaction or even for the creation of tuples needed to load the database during the startup phase.

Therefore, the implementation of these random generators plays an important role in the implementation of TPC-C. Before, going in more details about the implementation of TPC-C in LANCET, a quick definition of terms needs to be done.

**Uniform Random:** A simple random generator is typically defined as a uniform random generator that independently selects at random a number between a given range [x . . . y].

**Non-Uniform Random:** A non-uniform random generator is only used for generating customer numbers, last names and item numbers. It means that it selects independently and non-uniformly distributed, a random number over a specified range of values [x. . . y]. TPC-C standard specifications [8] defines how this number must be generated. The number selected by this non-uniform generator can then be converted to produce a name based on the needs.

```
NURand(A,x,y):
    return (((random(0,A) | random(x,y)) + C)
            % (y - x + 1)) + x;
```

Listing 1: Non-uniform generator

Listing 1 shows the implementation of the non-uniform random generator. A is a constant chosen according to the size of range [x. . . y] and C is a run-time constant randomly chosen within [0. . . A].

| Transaction | Frequency of execution |
|---|---|
| New Order | 45% |
| Payment | 43% |
| Order Status | 4% |
| Delivery | 4% |
| Stock Level | 4% |

Table 1: Frequency of execution of each transaction according to the TPC-C standards specification.

| Application Calls | |
|---|---|
| **Type** | **Description** |
| sqlite3_prepare | Compile SQL text into byte code |
| sqlite3_bind | Bind the values to the SQL query |
| sqlite3_step | Get the next result row, if possible |
| sqlite3_column | Column values in the current result row |

Table 2: The SQLite C interface

### 3.2 Lancet

When selecting TPC-C as the application protocol in LANCET, the only input necessary is the desired number of warehouses. This parameter is defined by default to 1.

LANCET will produce for each request generated, an uniformly distributed number between 1 and 100. This value will determine which type of transaction will be sent to the server according to the percentages defined in Table 1.

Based on the transaction selected, LANCET will generate the input necessary to execute it, as explained in §2.3, and then send the request to the server.

In order to represent a TPC-C request, a new data structure has been defined in LANCET, which represents the union of all possible parameters of the five existing transactions. This data structure also contains an extra parameter representing the transaction type as an enum, allowing the server to know which transaction is being sent and which parameters to extract.

### 3.3 R2P2 Server

A generic R2P2 server [3] has been implemented. The server is in charge of multiple steps before polling for requests.

First, on startup, the server determines the database engine being used and calls the corresponding methods in charge of creating the in-memory database and loading the tables for the TPC-C benchmark.

Then, in the second step, the server needs to load the tuples into the newly created tables. This part heavily relies on random generator, and is in charge of loading the warehouses with their corresponding districts, customers and items. The number of warehouses is given as an input to the server and should should correspond with the number of warehouses given by the client to LANCET.

In order to speed up this process, the creation and insertion of tuples is performed in batches of 2500 tuples allowing to complete the loading process in $\sim$10 seconds compared to $\sim$30 minutes if done individually.

Once the server is ready, it will start polling and is in charge of determining, for each request, which type of transaction is being sent by the client and to forward it to its corresponding method.

To implement each transaction, we heavily relied on the samples programs provided in the TPC-C standards specification [8], which provides a pseudo-code for each transactions enabling us to verify the correctness of the SQL queries. Furthermore, existing TPC-C implementations were used [4, 9] as sources of inspiration.

The TPC-C standards specification states that each transaction should print, on its completion, a certain number of values based on the transaction type. In order to respect the standards, the response to each request will contain the necessary values to fulfill theses requirements.

Given the difference between the responses of each transaction, a different approach has been taken to represent a TPC-C response packet. The union of all possible responses would generate too much unnecessary data and induce a waste of resources. Thus, we decided to represent a response as a simple block of bytes starting with the transaction type.

By having this constraint, that the first bytes of a response are reserved to the transaction type, we can leverage this in LANCET and parse the response differently based on the transaction type. This method allowed us to send only the precise number of bytes required and avoid possible waste of resources.

**SQLite:** We decided to use SQLite as a starting point given its portability and accessibility, but the server can be extended with any database system.

SQLite offers a simple API as shown in Table 2. Each transaction executed in the TPC-C benchmark starts with a `BEGIN TRANSACTION` SQL query and ends with either a `COMMIT TRANSACTION` or a `ROLLBACK TRANSACTION` SQL query. This ensures that each method is executed as a transaction respecting the ACID properties.

The workflow is really similar between each method handling a specific transaction type. To execute a specific query, the text version of the query is given to `sqlite3_prepare` which initialize a `sqlite3_stmt` object.

Then if the SQL query is parameterized, `sqlite3_bind` is used to fill the missing parameters necessary to run the query. This is typically the case when the query is based on the parameters passed in the request or when it is based on the result of a previous query.

Once the query is prepared, the next step is to call `sqlite3_step` which gives access to the next result row or informs the user that no more rows are available. Depending on the needs of the query, `sqlite3_step` can be call multi-

ple type depending on the number of rows that needs to be recovered.

Finally, to access the column of a row, `sqlite3_column` is called with the column index. Multiple versions of this method are available based on the column type. To save the result of a column of type `TEXT`, it is required that the user directly copies the returned sequence of characters into a local variable. Once a new `sqlite3_step` is performed, previous pointers returned should not be used and can lead to undefined behaviors.

Given the interface provided in C by SQLite, we can clearly see that exploring the results of a given query needs to be done by manually exploring row by row and column by column the rows returned by the query. This operation can become expensive, specifically when there is a non-negligible number of rows and when there are multiple columns with type `TEXT` requiring to be copied every single time.

Another small downside of using SQLite is that it is really verbose, leading to C files having ∼1000 lines.

## 4 Experiment

Our experiment aims to test the implementation of the TPC-C benchmark and to evaluate the performance of each transaction present in TPC-C. This experiment tests the transactions using exclusively SQLite as RDBMS and R2P2 as transport protocol. To evaluate precisely the end-to-end latency, LANCET is configured in a symmetric setup using NIC-based hardware timestamping to measure RPCs end-to-end latency without client bias.

### 4.1 Experimental Setup

The experiment uses one client and one R2P2 server. The client machine is configured with 15 threads. The R2P2 is a single threaded server running SQLite configured with an in-memory database to increase performance.

### 4.2 Transactions Evaluation

Figure 3 shows the complementary CDF of each transaction. The duration of each transaction was captured locally at the server to measure precisely the duration of each transaction without including the network latency.

As expected, the delivery and new_order transactions are the most heavy-weight, read-write, transactions followed by the stock_level transaction which is a heavy-weight, read_only, transaction.

### 4.3 R2P2 Experiment

Figure 4 shows the $99^{th}$-percentile tail latency as a function of the throughput. As we can see, the throughput cannot go

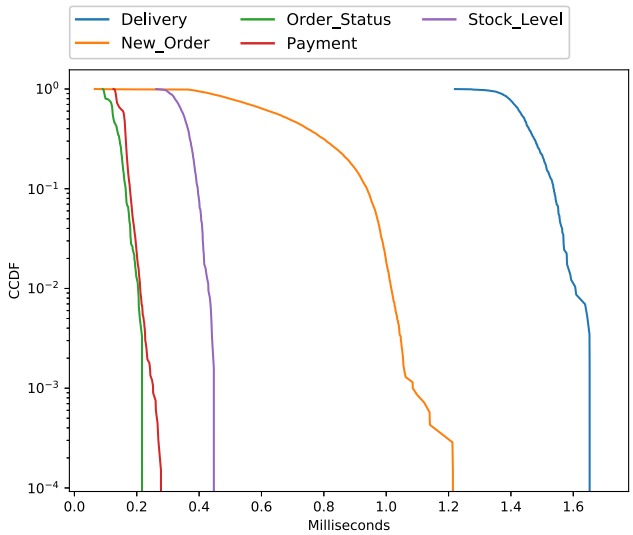


Figure 3: Complementary CDF of each transaction available in the TPC-C benchmark using SQLite as RDBMS.

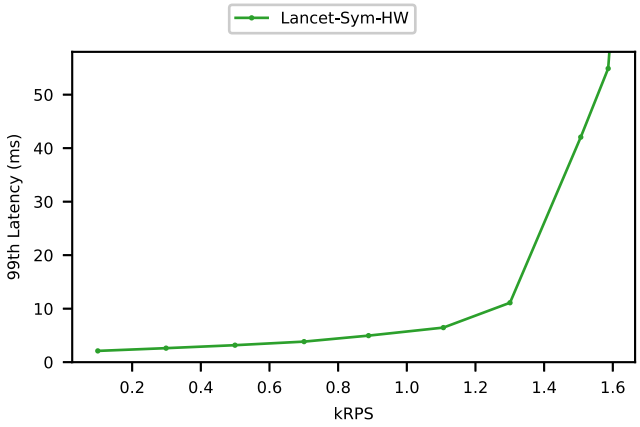


Figure 4: Latency vs throughput graph for a one client experiment using R2P2 as transport protocol.

past ∼1.6 kRPS. The reason behind this, is that we reached the maximal server capacity.

As we have seen in Figure 3, the transactions take a non-negligible amount of time to execute. But the main reason why latency increases with the number of requests per second, is that we are naturally experiencing some queuing at the server.

Performance could be improved by using another database engine. Indeed, SQLite supports an unlimited number of simultaneous readers, but it will only allow one writer at any instant time. Thus, using another database engine which provides better performance for concurrent execution would allow us to use is with a multi-threaded R2P2 server and improve performance.

## 5 Conclusion

LANCET has been extended with a full TPC-C benchmark. A generic R2P2 server is also provided allowing to run the benchmark remotely. While the server currently support only SQLite as a database system, it can be extended to support any database system which is able to execute transactions.

## References


[1] Kernel timestamping documentation. https://www.kernel.org/doc/Documentation/networking/timestamping.txt.

[2] KOGIAS, M., MALLON, S., AND BUGNION, E. Lancet: A self-correcting latency measuring tool. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 881–896.

[3] KOGIAS, M., PREKAS, G., GHOSN, A., FIETZ, J., AND BUGNION, E. R2p2: Making rpcs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Berkeley, CA, USA, 2019), USENIX ATC '19, USENIX Association, pp. 863–879.

[4] PAVLO, A. Python implementation of tpc-c. https://github.com/apavlo/py-tpcc.

[5] SQLite. https://www.sqlite.org/index.html.

[6] The SQLite Amalgamation. https://www.sqlite.org/amalgamation.html.

[7] TPC-C Benchmark. http://www.tpc.org/tpcc/.

[8] TPC-C Standards Specification. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[9] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 18–32.