Examining TPC-C Characteristics on Modern E-Commerce Applications

Xueyuan Ren[0009–0006–0305–7565] and Yang Wang[0000–0002–9721–4923]

The Ohio State University, Columbus, OH, USA
{ren.450,wang.7564}@osu.edu

Abstract. TPC-C is widely used in evaluating transaction processing systems, due to its versatility and reasonable complexity to test different aspects of the database system. However, due to its age, it's questionable whether its workload still represents today's e-commerce applications. We compare the design of TPC-C with that of three popular online shopping applications. We observe a few key differences, including contention levels, use of complicated queries, imbalanced data and traffic, etc. Based on the study, we propose a few changes to TPC-C to better capture the performance characteristics of modern OLTP applications.

Keywords: TPC-C · OLTP · Benchmark.

1 Introduction

Since its introduction in 1992, TPC-C [4] has become the de facto benchmark for online transaction processing (OLTP) systems, probably due to its versatility and reasonable complexity to test different aspects of the system. Despite the introduction of later OLTP benchmarks, such as TPC-E [5], SmallBank [10], YCSB+T [14], etc., TPC-C remains one of the most popular benchmarks for OLTP systems. Examples of works using TPC-C for evaluation include both research prototypes [11, 16–19, 21, 22, 25] and industrial systems [23, 26, 27].

As a result, TPC-C is guiding research in this field to some extent, since a corresponding work has to address the bottlenecks in TPC-C to demonstrate its benefits. Such an important role is also a source of concern, especially considering the age of TPC-C: If the characteristics of TPC-C deviate from the applications it represents, it may misguide us to address problems that are not prevalent or miss real problems.

This paper compares TPC-C with three popular online shopping applications. On the one hand, we find that, at a high level, these applications provide similar functions as the ones simulated by TPC-C, including creating orders, making payments, browsing histories, etc. On the other hand, we observe that, compared to TPC-C, these applications have made a few different design choices, leading to different performance characteristics. These include using shorter transactions and relying on application-specific logic or administrator's effort to address atomicity and isolation issues, reducing contention levels with optimized implementations, using complicated OLAP-like queries in certain scenarios, etc.

Motivated by such observations, we have proposed changes to TPC-C so that it can better represent modern e-commerce applications.

2 Background of TPC-C

TPC-C simulates the database of a wholesale company, with a configurable number of warehouses. Each warehouse has 100,000 items to sell, and covers 10 districts each with 3,000 customers. It consists of nine tables (WAREHOUSE, DISTRICT, CUSTOMER, HISTORY, ORDER, NEW-ORDER, ORDER-LINE, STOCK, and ITEM) and five types of transactions:

```
TRANSACTION BEGIN
-- fetch customer data
1 SELECT c_discount,... FROM customer WHERE c_w_id=? AND c_d_id=? AND c_id=?;
-- fetch warehouse data
2 SELECT w_tax FROM warehouse WHERE w_id=?;
-- fetch district d_next_o_id for update
3 SELECT d_next_o_id,... FROM district WHERE d_w_id=? AND d_id=? FOR UPDATE;
-- increment d_next_o_id by one
4 UPDATE district SET d_next_o_id=d_next_o_id+1 WHERE d_w_id=? AND d_id=?;
-- insert order and new_order record
5 INSERT INTO oorder (o_w_id, o_d_id, o_id,...) VALUES (?,?,?,...);
6 INSERT INTO new_order (no_w_id, no_d_id, no_o_id) VALUES (?,?,?);
-- execute for each order_line item
for each order_line item:
    -- fetch item data
7   SELECT i_price, i_name, i_data FROM item WHERE i_id=?;
    -- fetch stock data
8   SELECT s_quantity,... FROM stock WHERE s_w_id=? AND s_i_id=? FOR UPDATE;
    -- insert order_line record
9   INSERT INTO order_line (ol_w_id,ol_d_id,ol_o_id,...) VALUES (?,?,?,...);
    -- update stock
10  UPDATE stock SET s_quantity = ?, ... WHERE s_w_id=? AND s_i_id=?;
TRANSACTION COMMIT
```

Fig. 1: TPC-C NEW-ORDER Transaction.

New-Order (Figure 1). It simulates the procedure of a customer creating an order. A NEW-ORDER transaction randomly selects a district from a warehouse, selects 5 to 15 items based on an uneven distribution (some items have a higher probability of being selected), and randomly selects a quantity between one and ten for each item. It creates an order that includes all the selected items.

To create a unique order ID for each order, TPC-C maintains a per-district D_NEXT_O_ID value (i.e., the next available order number) in the DISTRICT table. NEW-ORDER retrieves this D_NEXT_O_ID value, uses it as the ID of the new order, and then increments the D_NEXT_O_ID value so that the next order can have a different ID (lines 3-4). NEW-ORDER inserts a new order record in both the NEW-ORDER table and the ORDER table. Then for each selected item, it retrieves the price from the ITEM table, updates the item count in the STOCK table, and inserts a record in the ORDER-LINE table.

In terms of performance characteristics, the per-district D_NEXT_O_ID value is a major contention point, since all NEW-ORDER transactions of the same district need to retrieve and update the same D_NEXT_O_ID value. The second contention point is the item count in the STOCK table when multiple NEW-ORDER transactions try to order the same item. As discussed above, TPC-C uses an uneven distribution to select items to order, so the contention chance is not small despite that there are 100,000 items.

```
TRANSACTION BEGIN
1 UPDATE warehouse SET w_ytd = w_ytd + ? WHERE w_id=?;
-- fetch warehouse data
2 SELECT w_street_1,... FROM warehouse WHERE w_id=?;
-- update district d_ytd with payment amount
3 UPDATE district SET d_ytd = d_ytd + ? WHERE d_w_id=? AND d_id=?;
-- fetch district data
4 SELECT d_street_1,... FROM district WHERE d_w_id=? AND d_id=?;
-- for simplicity, we only show fetch customer by id
5 SELECT c_credit,... FROM customer WHERE c_w_id=? AND c_d_id=? AND c_id=?;
-- for bad credit customer, fetch c_data and update
if c_credit == "BC":
6   SELECT c_data FROM customer  WHERE c_w_id=? AND c_d_id=? AND c_id=?;
7   UPDATE customer SET c_balance = ?, c_data = ?,...
      WHERE c_w_id=? AND c_d_id=? AND c_id=?;
-- for good credit customer, update customer balance
else:
8   UPDATE customer SET c_balance = ?,...
      WHERE c_w_id=? AND c_d_id=? AND c_id=?;
-- insert history record
9 INSERT INTO history (...) VALUES (...);
TRANSACTION COMMIT
```

Fig. 2: TPC-C PAYMENT Transaction.

Payment (Figure 2). It simulates the procedure of a customer making a payment. It updates the warehouse and district sales statistics (i.e., year-to-date amount) (line 1 and line 3). It then updates the customer's balance and payment values in the corresponding tables.

In terms of performance characteristics, the per-warehouse sales statistics is a severe contention point, as all PAYMENT transactions on the same warehouse need to update the same value.

```
TRANSACTION BEGIN
-- fetch the customer, the last order, and order_line data respectively
1 SELECT c_first,... FROM customer WHERE c_w_id=? AND c_d_id=? AND c_id=?;
2 SELECT o_id, o_carrier_id, o_entry_d FROM oorder
    WHERE o_w_id=? AND o_d_id=? AND o_c_id=? ORDER BY o_id DESC LIMIT 1;
3 SELECT ol_i_id, ol_supply_w_id,... FROM order_line
    WHERE ol_w_id=? AND ol_d_id=? AND ol_o_id=?;
TRANSACTION COMMIT
```

Fig. 3: TPC-C ORDER-STATUS Transaction.

Order-Status (Figure 3). It simulates the procedure of a customer browsing past orders. It selects the customer's order with the largest order ID in the

ORDER table (line 4). Then it retrieves item information by selecting rows from the ORDER-LINE table with the same order ID.

```
TRANSACTION BEGIN
for each district to deliver:
  -- fetch oldest undelivered order
1 SELECT no_o_id FROM new_order WHERE no_w_id=? AND no_d_id=?
    ORDER BY no_o_id ASC LIMIT 1 FOR UPDATE;
  if no_o_id is not null:
    -- delete the new_order record
2   DELETE FROM new_order WHERE no_w_id=? AND no_d_id=? AND no_o_id=?;
    -- fetch the customer id for this order
3   SELECT o_c_id FROM oorder WHERE o_w_id=? AND o_d_id=? AND o_id=?;
    -- update the order carrier id
4   UPDATE oorder SET o_carrier_id=? WHERE o_w_id=? AND o_d_id=? AND o_id=?;
    -- update the order_line delivery date
5   UPDATE order_line SET ol_delivery_d = ?
      WHERE ol_w_id=? AND ol_d_id=? AND ol_o_id=?;
    -- fetch the total amount of this order
6   SELECT SUM(ol_amount) FROM order_line
      WHERE ol_w_id=? AND ol_d_id=? AND ol_o_id=?;
    -- update the customer balance
7   UPDATE customer SET c_balance = c_balance + ?, c_delivery_cnt =
      c_delivery_cnt + 1 WHERE c_w_id=? AND c_d_id=? AND c_id=?;
TRANSACTION COMMIT
```

Fig. 4: TPC-C DELIVERY Transaction.

Delivery (Figure 4). It simulates the procedure of the administrator making a delivery for an order. For each district, it selects the oldest order from the NEW-ORDER table, deletes this row, retrieves detailed information from the ORDER table, and then updates the ORDER-LINE table. It finally updates the balance in the CUSTOMER table.

In terms of performance characteristics, DELIVERY may contend with NEW-ORDER, as both may update the NEW-ORDER table. The actual contention rate may not be high, as NEW-ORDER adds new orders and DELIVERY retrieves and deletes the oldest order. Of course, this depends on how the database implements INSERT and SELECT.

```
TRANSACTION BEGIN
-- fetch district next order id
1 SELECT d_next_o_id FROM district WHERE d_w_id=? AND d_id=?;
-- retrieve low-stock items in recent orders
2 SELECT COUNT(DISTINCT(s_i_id)) FROM order_line, stock
    WHERE ol_w_id=? AND ol_d_id=? AND ol_o_id<? AND ol_o_id>=?
    AND s_w_id=? AND s_i_id=ol_i_id AND s_quantity<?;
TRANSACTION COMMIT
```

Fig. 5: TPC-C STOCK-LEVEL Transaction.

Stock-Level (Figure 5). It simulates the procedure of the administrator browsing the recently sold items whose stock level is under a threshold. Given a district, it first retrieves the D_NEXT_O_ID value from the DISTRICT table. It then selects order lines whose order IDs are greater than or equal to D_NEXT_O_ID - 20 from the ORDER-LINE table. For each item in these order lines, STOCK-LEVEL checks whether its quantity in the STOCK table is less than a threshold.

In terms of performance characteristics, STOCK-LEVEL contends with NEW-ORDER on D_NEXT_O_ID and maybe also on item quantity.

TPC-C further specifies the expected distribution of these five types of transactions: NEW-ORDER 45%, PAYMENT 43%, ORDER-STATUS 4%, DELIVERY 4%, and STOCK-LEVEL 4%. As a result, NEW-ORDER and PAYMENT are typically performance critical though others can get heavier in certain settings.

TPC-C also specifies a keying time and a think time before each transaction to simulate the customer's behaviors. However, most research prototypes remove such time in their evaluation for various reasons [24].

3 Methodology

We select three e-commerce applications to investigate: Spree [2], WooCommerce [7], and PrestaShop [1]. These applications are popular open-source e-commerce platforms with high Github stars (Spree: 15k, WooCommerce: 10k, PrestaShop: 8.8k as of writing). They are also widely used in the industry and have a large user base [8, 9], claiming thousands to millions of active stores, making them suitable candidates for our study. They also represent different approaches to e-commerce, with Spree built on Ruby on Rails, WooCommerce as a widely used plugin for WordPress that enables e-commerce functionality, and PrestaShop built on PHP.

For each application, we create a deployment including the web server running the corresponding application and a backend database (MySQL in our experiments). We first act as a seller to set up a sample store, then using a customer account to interact with the store through the webpages. We perform typical e-commerce operations, such as searching for items, creating orders, making payments, and browsing order histories, and then act as the seller again to confirm deliveries, browse item stocks, etc.

The development environments for these applications often output logs to the console, which include application transactions triggered by customer and seller operations. We can also enable database tracing during the procedure to record SQL statements issued by each of those actions. We analyze the recorded SQL traces to understand their intentions. We also cross-check with the application source code to understand application-level logic that is not captured by SQL traces. After analyzing the SQL traces and source code, we summarize the main functions and their corresponding transactions for each application. As a result, we collect a set of transactions for above typical operations in e-commerce applications.

4 Spree

Spree Commerce is an open-source e-commerce platform built with Ruby on Rails. It provides a flexible and modular architecture that allows developers to create custom online stores. Spree supports various features such as product management, order processing, payment integration, and shipping options.

A Spree deployment can include a number of "stores", which are similar to, but not exactly the same as, warehouses in TPC-C. A Spree store maps to a seller on online shopping websites like Amazon and Alibaba. As a result, different stores can have a vastly different number of items to sell and highly skewed traffic. In TPC-C, each warehouse has the same number of items and incoming traffic. The deployment has three roles: customer, seller (i.e., a store owner), and admin of the whole website. This study covers the customer and the seller, as their activities are similar to those in TPC-C.

Spree provides similar functions as TPC-C: A customer can create orders, make payments, browse past orders, etc. A seller can make deliveries, check stock levels, etc. However, Spree implements those functions in different ways.

4.1 Create Order

A customer in Spree can create orders with a NEW-ORDER and multiple ADD-ITEM transactions. In other words, a NEW-ORDER transaction in TPC-C maps to a NEW-ORDER and multiple ADD-ITEM transactions in Spree.

```
TRANSACTION BEGIN
-- validation and load information
SELECT 1 AS one FROM spree_orders
  WHERE spree_orders.number = 'R005887550' LIMIT 1;
SELECT 1 AS one FROM spree_orders
  WHERE spree_orders.number = CAST('R005887550' AS BINARY) LIMIT 1;
SELECT spree_users.* FROM spree_users WHERE spree_users.id = 2 LIMIT 1;
-- insert order record
INSERT INTO spree_orders (number, ......) VALUES ('R005887550', ......);
TRANSACTION COMMIT
```

Fig. 6: Spree NEW-ORDER Transaction.

Spree NEW-ORDER. This transaction creates a new order record for a customer when the customer adds her first item to her shopping cart. This order begins in the "cart" state and will hold all the items the customer adds until the customer either decides to checkout or to cancel the order.

Figure 6 shows the steps involved in creating a new order. Spree first generates a random order number (R005887550 in this example) and checks if an order with this exact number already exists in the spree_orders table. The second query performs a case-sensitive comparison to ensure the number is unique. If the order number is unique, it proceeds to the third query, which retrieves the customer information, particularly the customer's user ID, from the spree_users table. Spree will regenerate the order number until there is no same number existing. The final statement inserts a record for the new order in the spree_orders table. Compared to TPC-C NEW-ORDER, which generates a unique order ID by incrementing D_NEXT_O_ID, Spree's approach incurs almost no contention.

```
TRANSACTION BEGIN
-- load information
SELECT * FROM spree_prices WHERE ......;
SELECT * FROM spree_line_items WHERE ......;
......
-- compute item count
SELECT SUM(spree_stock_items.count_on_hand) FROM spree_stock_items
  INNER JOIN spree_stock_locations ON spree_stock_locations.deleted_at IS
       NULL AND spree_stock_locations.id = spree_stock_items.stock_location_id
  WHERE spree_stock_items.deleted_at IS NULL AND spree_stock_items.variant_id
        = 142 AND spree_stock_locations.deleted_at IS NULL AND
       spree_stock_locations.active = TRUE;
-- create line item
INSERT INTO spree_line_items (......);
-- set initial price
UPDATE spree_line_items SET pre_tax_amount = 71.99 WHERE id = 26;
-- compute adjusted price including tax, shipment, promo, etc.
SELECT SUM(quantity) FROM spree_line_items WHERE order_id = 4;
......
-- update order
UPDATE spree_orders SET ...... WHERE id = 4;
TRANSACTION COMMIT
```

Fig. 7: Spree ADD-ITEM Transaction.

Spree ADD-ITEM. The ADD-ITEM transaction in Spree adds an item to an existing order when a customer adds an item to her shopping cart.

Figure 7 shows the steps of ADD-ITEM. It first gathers the necessary information and performs a series of checks. Then, it retrieves the item information and runs a SUM query on spree_stock_items table to ensure the product is in stock and available for purchase before adding it to the cart. Here, Spree uses the SUM query to count items in all stock locations. If the product is available, it proceeds to create a new line item in the spree_line_items table thus adding the item to the cart. It then computes the order's total price by including each item's original price and adjusted cost such as tax, shipment, promo, etc. Finally, it updates the order information.

Compared to TPC-C NEW-ORDER, Spree ADD-ITEM does not update the item count, which is updated during CHECKOUT and discussed later. This means that different ADD-ITEM transactions do not contend, but ADD-ITEM may contend with CHECKOUT.

4.2 Checkout

The checkout process in Spree, which maps to PAYMENT in TPC-C, involves multiple transactions that handle various aspects of completing an order, such as addressing, delivery, payment, and order finalization. We will focus on the payment and order finalization transactions. These transactions include creating a payment record, updating the stock levels, and finalizing the order.

Separating them into multiple transactions may violate atomicity: A failure may cause payment to be made but stock level is not updated. Spree (or Ruby on Rails) has application-level logic to (help the seller) detect such incomplete checkout and resume the unfinished checkout, by logging necessary information. In other words, Ruby on Rails is implementing a redo log by itself. Prior studies have shown that real-world applications often use such "ad-hoc" transactions to improve performance at the cost of more complicated error handling [12, 20].

```
TRANSACTION BEGIN
-- load information and validate
SELECT * FROM spree_payments WHERE ......;
SELECT * FROM spree_users WHERE ......;
......
SELECT spree_stores.id FROM spree_stores INNER JOIN
     spree_payment_methods_stores ON spree_stores.id =
     spree_payment_methods_stores.store_id WHERE spree_stores.deleted_at IS
     NULL AND spree_payment_methods_stores.payment_method_id = 3 ORDER BY
     spree_stores.created_at ASC;
-- create payment records
INSERT INTO spree_payments (..., state, ...) VALUES (..., 'checkout', ...);
......
TRANSACTION COMMIT
```

Fig. 8: Spree PAYMENT Transaction.

Spree PAYMENT. This transaction creates a payment record for the order when a customer selects a payment method during checkout. Note that the payment is actually processed later.

Figure 8 shows the steps of the PAYMENT transaction. It begins with a series of SELECT queries to load and validate data, including the customer's account, her addresses, and the payment method she chose. Then it inserts a new payment record in the spree_payments table. The record's initial state is set to 'checkout', indicating that the payment has been created but not yet processed.

```
TRANSACTION BEGIN
-- load information and validate
SELECT 1 AS one FROM spree_variants LEFT OUTER JOIN spree_stock_items ON ...
     AND spree_stock_items.variant_id = spree_variants.id WHERE ......;
......
-- create movement record
INSERT INTO spree_stock_movements ......;
-- load and update stock item
SELECT * FROM spree_stock_items WHERE ...... FOR UPDATE;
UPDATE spree_stock_items SET count_on_hand = 99, ..... WHERE id = 142;
TRANSACTION COMMIT
```

Fig. 9: Spree UPDATE-STOCK Transaction.

Spree UPDATE-STOCK. It creates a stock movement record and decreases the item count for the purchased item.

Figure 9 shows the main steps in UPDATE-STOCK. This transaction includes several validation steps to ensure that the product variant is available for purchase. It checks if the variant exists, retrieves product information, and verifies stock availability (omitting due to space). Note that Spree's model of items is more complicated than that in TPC-C. In Spree, each item can have multiple "variants", often due to their differences in colors, sizes, etc., and variant information is stored in a separate table. As a result, related queries often need to JOIN the item and the variant tables. Then, UPDATE-STOCK creates an audit record in the spree_stock_movements table, which logs the quantity change applied to the stock item. Afterwards, it updates the item count (count_on_hand) in the spree_stock_items table (from 100 to 99 in this example). The update first uses SELECT FOR UPDATE to retrieve current item count and lock the item count to prevent concurrent updates.

Compared to TPC-C PAYMENT, Spree UPDATE-STOCK does not update sales statistics like year-to-date amount in TPC-C. As a result, Spree UPDATE-STOCK transactions do not have heavy contentions like TPC-C PAYMENT, unless they target the same item. If a Spree seller wants to view sales statistics, she will incur transactions to compute the sum of all orders, which will contend with UPDATE-STOCK. Considering UPDATE-STOCK is probably more frequent than viewing statistics, we believe this is a reasonable design choice to reduce contentions.

```
TRANSACTION BEGIN
-- load information
SELECT 1 AS one FROM spree_orders WHERE ......;
-- update order states
UPDATE spree_orders SET shipment_state = 'pending', payment_state = '
    balance_due', updated_at = '...' WHERE id = 4;
TRANSACTION COMMIT
```

Fig. 10: Spree Finalize-Order Transaction.

Spree FINALIZE-ORDER. It finalizes the order by updating its state related to shipment and payment.

This final simple transaction updates the overall status of the order, moving it to the next step in the checkout process. Figure 10 shows the statements involved in finalizing the order. The update statement sets the shipment state to 'pending' and the payment state to 'balance_due'. This indicates that the inventory has been allocated for the order, but the item is not yet shipped. A payment record exists, but the funds have not been captured yet by the seller. This transaction moves the order to the fulfillment workflow.

4.3 Order-Status Transactions

The Order-Status transactions in Spree are responsible for retrieving the current status of orders. There are two types of transactions: one for retrieving a list of orders for order history and another for retrieving the details of a specific order.
Spree ORDER-HISTORY. It retrieve a list of orders (the most recent 25 orders by default) for a specific customer. This transaction is triggered when a customer wants to view their order history. Because this process is read-only, each single

```
-- check if there are completed orders
SELECT 1 AS one FROM spree_orders WHERE user_id = 2 AND completed_at IS NOT
     NULL AND store_id = 1 LIMIT 1 OFFSET 0;
-- retrieve the last 25 completed orders
SELECT spree_orders.* FROM spree_orders WHERE user_id = 2 AND completed_at IS
      NOT NULL AND store_id = 1 ORDER BY created_at DESC LIMIT 25 OFFSET 0;
-- retrieve the last pending order
SELECT spree_orders.* FROM spree_orders WHERE user_id = 2 AND store_id = 1
     AND completed_at IS NULL AND state NOT IN ('canceled', '
     partially_canceled') ORDER BY created_at DESC LIMIT 1;
```

Fig. 11: Spree ORDER-HISTORY Transaction.

SELECT is executed as a transaction. We list the queries most relevant to the order status in Figure 11.

These queries fetch a paginated list of a user's completed orders to display on their account page. The first query checks if there are any completed orders for the user. Then, it selects all completed orders for the current customer and orders them by creation date. The last query checks if there are any pending orders for the customer.

```
-- retrieve order by order number and line items for the order
SELECT * FROM spree_orders WHERE ......;
SELECT * FROM spree_line_items WHERE ......;
```

Fig. 12: Spree ORDER-DETAILS Transaction.

Spree ORDER-DETAILS. It retrieves the details of a specific order. This transaction is triggered when a customer clicks an order to view the details of this order. It will retrieve all the information to display a complete, detailed view of a single order.

To build the order details page, Spree needs to reconstruct the entire order by fetching data from multiple tables. Figure 12 shows the queries involved in retrieving the order details. This includes the order itself, line item details, shipment and address information, and payment data.

Compared to TPC-C ORDER-STATUS, Spree's implementation is more complicated, but we do not see a fundamental difference in performance characteristics.

4.4 Shipment Transaction

After a seller verifies the payment information, she can start the delivery process and use the SHIPMENT transaction to mark an order as shipped. It has the similar role as DELIVERY in TPC-C. Figure 13 shows the details, which include some validation steps and a UPDATE statement to set the shipment status as 'shipped'.

```
TRANSACTION BEGIN
-- validation
SELECT 1 AS one FROM spree_shipments WHERE spree_shipments.number = ......;
......
-- update shipment status
UPDATE spree_shipments SET state = 'shipped', ... WHERE id = 3;
......
TRANSACTION COMMIT
```

Fig. 13: Spree SHIPMENT Transaction.

```
SELECT spree_stock_items.* FROM spree_stock_items INNER JOIN spree_variants
    ON spree_stock_items.variant_id = spree_variants.id INNER JOIN
    spree_products ON spree_variants.product_id = spree_products.id INNER
    JOIN spree_products_stores ON spree_products.id = spree_products_stores.
    product_id INNER JOIN spree_variants variants_spree_stock_items ON
    variants_spree_stock_items.id = spree_stock_items.variant_id WHERE ... IN
     (SELECT spree_variants.product_id FROM spree_variants WHERE
    spree_variants.deleted_at IS NULL GROUP BY spree_variants.product_id
    HAVING (COUNT(spree_variants.id) = 1))) ORDER BY spree_stock_items.
    created_at DESC, spree_variants.position ASC LIMIT 25 OFFSET 0;
```

Fig. 14: Spree STOCK-LEVEL Transaction.

4.5 Stock-Level Transaction

A seller can incur the STOCK-LEVEL transactions to view the stock levels of all product variants.

The first group of statements is executed when a seller navigates to the stock management page to view the stock levels of all products in the store. Figure 14 shows the main queries involved in this process. The query is a complex SELECT statement that incorporates multiple JOINs, a subquery, and keywords like GROUP BY, HAVING, ORDER BY, etc. It builds a list of all product variants for the store. Then, it executes a series of queries to load the associated products, variants, stock locations, and count on hand for each variant. Since these are quite straightforward, we do not list details here.

Compared to TPC-C STOCK-LEVEL, Spree STOCK-LEVEL looks more like an OLAP transaction, which tests the database's capability for complicated queries. We have observed similar transactions in other seller-side transactions, such as SALES-REPORT and ANALYTICS-DASHBOARD.

4.6 Summary of Comparison with TPC-C

As a mature product, Spree is obviously more complicated than TPC-C, represented by its more complicated table designs, various kinds of validations, etc. Apart from such implementation-level differences, we observe the following major differences:

– Unlike warehouses in TPC-C, Spree's stores are probably highly imbalanced, in terms of both item number and traffic.

- Compared to TPC-C, Spree tends to use multiple shorter transactions to implement a function, and relies on application-specific logic or seller's effort to achieve atomicity.
- The design of Spree has reduced contention level compared to TPC-C. This includes not relying on D_NEXT_ORDER_ID to generate a unique order ID, and not maintaining the sales statistics.
- TPC-C updates the item count when a customer adds the item to the order. Spree updates the item count when the customer checks out the order.
- Spree uses JOIN in multiple transactions due to its table design. Some complicated queries combine multiple JOINs, subqueries, GROUP BY keywords, etc, which make them more close to OLAP transactions.

5 WooCommerce and PrestaShop

We perform the same study for WooCommerce and PrestaShop. At a high level, their designs are quite similar to that of Spree, though the detailed implementations are different. This section summarizes the key comparison with TPC-C:

- (Same as Spree) Both have the concept of "store". Again, stores are probably highly load imbalanced.
- (Slightly different from Spree) Both use short transactions. In fact, both only use single-statement transactions, as shown in the traces we collected.
- (Different from Spree) Both use the "auto increment" feature of the database to generate a unique order ID. This approach still has less contention than retrieving and incrementing D_NEXT_O_ID in TPC-C.
- (Slightly different from Spree) Both maintain a per-item sales statistics for each store. Compared to TPC-C's per-warehouse sales statistics, maintaining a per-store per-item sales statistics incurs less contention.
- (Same as Spree) Both update item count when the customer checks out her order.
- (Same as Spree) Both use JOINs in multiple transactions. Both have complicated OLAP-like transactions to browse histories and/or statistics.

6 Suggestions and Results

Based on our study, we have tried to incorporate the difference into TPC-C. We find some are easy to incorporate, but some require a significant change to TPC-C. Therefore, our suggestion is to incorporate those "easy" changes into TPC-C and maybe build a new benchmark to incorporate those significant changes in the long term. In this section, we discuss the easy changes we suggested to TPC-C, their potential impact on evaluation of transaction processing systems, and why other changes require a significant effort.

Suggested changes to TPC-C. We find the following changes can be made to TPC-C without significant changes:

- We make warehouses imbalanced. Specifically, we use the zipf distribution to generate the number of items in each warehouse and the traffic to each warehouse. Data skew is also recommended by other works [15] and TPC-E [5].
- We split the NEW-ORDER transaction in TPC-C into one that creates an empty order, and multiple ADD-ITEM transactions, each adding one item to the order.
- We use the auto-increment feature of MySQL to generate a unique order ID.
- We remove the logic of updating the year-to-date sales statistics in PAYMENT.

Potential impact on evaluation. The suggested changes may affect the evaluation of transactional processing systems in the following ways:

- The imbalanced warehouses will test a database's capability to do load balancing, which is important for a distributed or multi-core database. Standard TPC-C does not have this capability.
- Due to using auto-increment to generate IDs and removing maintenance of sales statistics, modified TPC-C has less contention than standard TPC-C. Modified TPC-C contends only when customers purchase the same item from the same store. We can still tune this contention level by tuning the distribution of items to purchase.
- Shorter transactions and less contention make works that rely on code analysis more feasible.
- Shorter (and more) transactions may increase overhead when contention level is low, as the database system needs to pay more overhead for committing transactions and transferring messages. However, when contention level is high, shorter transactions can reduce contention duration.

To demonstrate some of such potential impact, we compare the results of standard TPC-C and our modified TPC-C on MySQL. We ran our experiments on CloudLab [13], using c220g5 machines. Each machine is equipped with two Intel Xeon Silver 4114 10-core CPUs (20 virtual cores with hyperthreading), 192 GB of RAM, and one Intel DC S3500 480 GB 6G SATA SSD. The machines are interconnected by a dual portIntel X520-DA2 10Gb NIC. We run MySQL server version 8.0.42 and configure it with 8 GB buffer pool size and 4 GB redo log.

Figure 15 shows the effects of using shorter transactions and removing contention points. When there is one warehouse, the throughput of modified TPC-C is significantly higher than that of standard TPC-C. This is because, with one warehouse, standard TPC-C has severe contention on per-warehouse sales statistics and per-district D_NEXT_O_ID, and modified TPC-C has removed such contention points. With more warehouses, the trend is the opposite: As the contention level gets lower with more warehouses, standard TPC-C gets higher throughput. Modified TPC-C suffers more from the overhead of processing more shorter transactions, and thus has lower throughput than standard TPC-C.

Effect of contention level. We tune the skewness of item hotness to change the contention level of modified TPC-C, since multiple NEW-ORDER transactions contend if they target the same item. Note that in standard TPC-C, the item

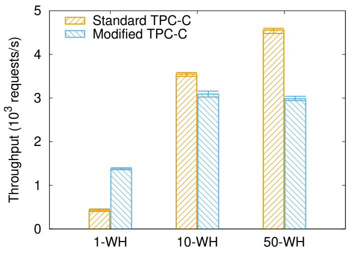
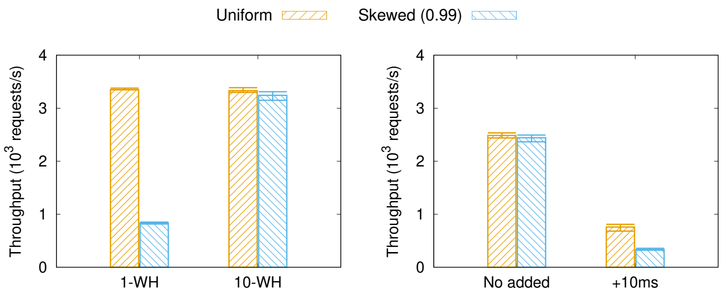


Fig. 15: Throughput of standard and modified TPC-C. For modified TPC-C, we count one NEW-ORDER and its corresponding ADD-ITEM as one transaction for a fair comparison with standard TPC-C.



(a) Single MySQL server with 1 or 10 warehouses

(b) 3-way replication with increased network latency

Fig. 16: Throughput of modified TPC-C for varying item skewness (uniform and skewed) under different settings.

access is non-uniform by default, but the contention level is different from that of our tuned item skewness. Figure 16(a) shows that, with a single MySQL server and one warehouse, tuning the skewness of item hotness can decrease throughput by up to 75%, but with 10 or more warehouses, such tuning does not have a significant impact on throughput. However, when running the 10-warehouse setting with three-way replication and with increased network latency, as shown in Figure 16(b), increasing such skewness can reduce overall throughput by 56%, since replication increases the duration of contention. This means, compared with standard TPC-C, modified TPC-C allows us to tune contention level under the same number of warehouses, enabling more flexible experiment settings.

Effect of traffic skewness. As discussed, in practice, the stores may be highly imbalanced. We simulate such an imbalance by introducing a skewness into store traffic. We run our experiments on a 4-node cluster where tables are partitioned by the warehouse ID. Our experiments show that, compared to the uniform distribution, a skewed distribution can reduce overall throughput by 11%. Of course, the throughput of a distributed setting may be affected by the number of nodes, the policy to partition tables, and load balancing strategies, which can be the targets of further optimization.

As shown in these results, modified TPC-C indeed shows quite different performance characteristics compared to standard TPC-C, which shows the importance of such a study.

Changes that require significant effort. We do not add more JOINs or complicated OLAP-like queries, since they require a significant re-design of TPC-C's table structure. We tried to move the logic of updating stock level from NEW-ORDER to PAYMENT, but that is harder than we thought. The reason is that in standard TPC-C, PAYMENT does not connect to any specific order and instead only generates some random payment. If we wanted to let PAYMENT update the stock level, we needed to let it connect to a specific order, retrieve the item and purchase count from the order, and then update the item count. Such changes require a significant re-design of PAYMENT. As a result, we do not incorporate these changes in modified TPC-C but recommend building a new benchmark to incorporate them.

7 Related work

In addition to TPC-C, the database community has developed several other benchmarks for evaluating OLTP systems. For example, the SmallBank benchmark models a banking application in which transactions operate customers' accounts [10]. The Voter Benchmark simulates a voting system in which users vote on their favorite contestant, updating the total number of votes [6]. The TATP benchmark simulates a mobile carrier database, which provides high-speed transactions to retrieve and update information for callers [3]. The YCSB+T benchmark extends YCSB, a key-value benchmark, to encapsulate multiple key-value operations in a transaction [14]. TPC has also introduced another OLTP benchmark TPC-E [5] in 2007.

However, we observe that despite its age, TPC-C remains one of the most popular OLTP benchmarks in both academia and industry, possibly due to its reasonable complexity (benchmarks like SmallBank and Voter only use simple SQL statements while TPC-E may be too complex) and the fact that online shopping demands high throughput under potential high contentions, which creates challenging problems.

A number of works have analyzed TPC-C and/or proposed changes to TPC-C [24, 28]. To the best of our knowledge, this is the first work to compare TPC-C with real-world online shopping applications.

8 Conclusion

This work compares TPC-C with three online shopping applications to understand the representativeness of TPC-C. We have identified a few key differences in the design choices of TPC-C and others, which can lead to different performance characteristics. Based on this study, we have proposed changes to TPC-C so that it can better represent modern e-commerce applications.

Acknowledgments. We thank the anonymous reviewers for their valuable feedback. This material is based in part upon work supported by the National Science Foundation under Grant Numbers CCF-2118745.

References

1. PrestaShop. https://github.com/PrestaShop/PrestaShop
2. Spree. https://github.com/spree/spree
3. TATP Benchmark. https://tatpbenchmark.sourceforge.net/
4. TPC-C Benchmark. https://www.tpc.org/tpcc/
5. TPC-E Benchmark. https://www.tpc.org/tpce/
6. Voter Benchmark. https://github.com/cmu-db/benchbase/wiki/Voter/
7. WooCommerce. https://github.com/woocommerce/woocommerce
8. Spree Commerce Documentation. https://spreecommerce.org/docs/user/what-is-spree-commerce (2024)
9. The State of Ecommerce in 2025. https://storeleads.app/reports (2025)
10. Alomari, M., Cahill, M., Fekete, A., Rohm, U.: The cost of serializability on platforms that use snapshot isolation. In: 2008 IEEE 24th International Conference on Data Engineering. pp. 576–585 (2008). https://doi.org/10.1109/ICDE.2008.4497466
11. Cai, Q., Guo, W., Zhang, H., Agrawal, D., Chen, G., Ooi, B.C., Tan, K.L., Teo, Y.M., Wang, S.: Efficient distributed memory management with rdma and caching. Proc. VLDB Endow. **11**(11), 1604–1617 (Jul 2018). https://doi.org/10.14778/3236187.3236209
12. Cheng, C., Han, M., Xu, N., Blanas, S., Bond, M.D., Wang, Y.: Developer's responsibility or database's responsibility? rethinking concurrency control in databases. In: 13th Annual Conference on Innovative Data Systems Research (CIDR'23). January 8-11, 2023, Amsterdam, The Netherlands. (2023)
13. CloudLab. https://www.cloudlab.us
14. Dey, A., Fekete, A., Nambiar, R., Röhm, U.: YCSB+T: Benchmarking web-scale transactional databases. In: 2014 IEEE 30th International Conference on Data Engineering Workshops. pp. 223–230 (2014). https://doi.org/10.1109/ICDEW.2014.6818330
15. He, H., Weng, S., Zeng, L., Zhang, H., Zhang, R., Cai, P., Zhou, X., Xu, Q.: Benchmarking distributed transactional database systems. In: International Symposium on Benchmarking, Measuring and Optimization. pp. 37–53. Springer (2024)
16. Lim, H., Kaminsky, M., Andersen, D.G.: Cicada: Dependably fast multi-core in-memory transactions. In: Proceedings of the 2017 ACM International Conference on Management of Data. pp. 21–35. SIGMOD '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3035918.3064015
17. Lu, Y., Yu, X., Cao, L., Madden, S.: Aria: a fast and practical deterministic oltp database. Proc. VLDB Endow. **13**(12), 2047–2060 (Jul 2020). https://doi.org/10.14778/3407790.3407808
18. Lu, Y., Yu, X., Madden, S.: Star: Scaling transactions through asymmetric replication. Proc. VLDB Endow. **12**(11), 1316–1329 (Jul 2019). https://doi.org/10.14778/3342263.3342270
19. Mu, S., Nelson, L., Lloyd, W., Li, J.: Consolidating Concurrency Control and Consensus for Commits under Conflicts. In: 12th USENIX Symposium on Operating

Systems Design and Implementation (OSDI 16). pp. 517–532. USENIX Association, Savannah, GA (2016), https://www.usenix.org/conference/osdi16/technical-sessions/presentation/mu

20. Tang, C., Wang, Z., Zhang, X., Yu, Q., Zang, B., Guan, H., Chen, H.: Ad hoc transactions in web applications: The good, the bad, and the ugly. In: Proceedings of the 2022 International Conference on Management of Data. p. 4–18. SIGMOD '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3514221.3526120
21. Thomson, A., Diamond, T., Weng, S.C., Ren, K., Shao, P., Abadi, D.J.: Calvin: fast distributed transactions for partitioned database systems. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. p. 1–12. SIGMOD '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2213836.2213838
22. Tu, S., Zheng, W., Kohler, E., Liskov, B., Madden, S.: Speedy transactions in multicore in-memory databases. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. p. 18–32. SOSP '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2517349.2522713
23. Wang, D., Chen, Y., Jiang, C., Pan, A., Jiang, W., Wang, S., Lei, H., Zhu, C., Zheng, L., Lu, W., Chai, Y., Zhang, F., Du, X.: Txsql: Lock optimizations towards high contented workloads. In: Companion of the 2025 International Conference on Management of Data. p. 675–688. SIGMOD/PODS '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3722212.3724457
24. Wang, Y., Yu, M., Hui, Y., Zhou, F., Huang, Y., Zhu, R., Ren, X., Li, T., Lu, X.: A study of database performance sensitivity to experiment settings. Proc. VLDB Endow. **15**(7), 1439–1452 (Mar 2022)
25. Wei, X., Shi, J., Chen, Y., Chen, R., Chen, H.: Fast in-memory transaction processing using rdma and htm. In: Proceedings of the 25th Symposium on Operating Systems Principles. pp. 87–104. SOSP '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2815400.2815419
26. Yang, X., Zhang, Y., Chen, H., Li, F., Wang, B., Fang, J., Sun, C., Wang, Y.: Polardb-mp: A multi-primary cloud-native database via disaggregated shared memory. In: Companion of the 2024 International Conference on Management of Data. p. 295–308. SIGMOD '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3626246.3653377
27. Yang, X., Zhang, Y., Chen, H., Sun, C., Li, F., Zhou, W.: Polardb-scc: A cloud-native database ensuring low latency for strongly consistent reads. Proc. VLDB Endow. **16**(12), 3754–3767 (Aug 2023). https://doi.org/10.14778/3611540.3611562
28. Zhang, H., Qu, L., Wang, Q., Zhang, R., Cai, P., Xu, Q., Yang, Z., Yang, C.: Dike: A benchmark suite for distributed transactional databases. In: Companion of the 2023 International Conference on Management of Data. p. 95–98. SIGMOD '23, Association for Computing Machinery, New York, NY, USA (2023)