Spring 2022

Benchmarking NewSQL Database VoltDB

Kevin Schumacher

Benchmarking NewSQL Database VoltDB

A Project Report

Presented to

Department of Computer Science

San José State University

In Partial Fulfillment

Of the Requirements of the

Degree of Science

By

Kevin Schumacher

December 2022

The Designated Project Committee Approves the Project Titled

Benchmarking NewSQL Database VoltDB

by

Kevin Schumacher

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

December 2022

Dr. Suneuy Kim	Department of Computer Science
Dr. Ben Reed	Department of Computer Science
Dr. Thomas Austin	Department of Computer Science

Abstract

NewSQL is a type of relational database that is able to horizontally scale while retaining linearizable consistency. This is an improvement over a traditional SQL relational database because SQL databases cannot effectively scale across multiple machines. This is also an improvement over NoSQL databases because NewSQL databases are designed from the ground up to be consistent and have ACID guarantees. However, it should be noted that NewSQL databases are not a one size fits all type of database, each specific database is designed to perform well on specific workloads. This project will evaluate a NewSQL database, VoltDB, with a focus on its performance and consistency guarantees.

VoltDB is a NewSQL type database that has a share-nothing distributed architecture and a lock-free concurrency method of maintaining consistency within the database. The shared-nothing distributed architecture ideally allows transactions to take place on a single machine, eliminating the need for consensus algorithms between multiple machines. The lock-free concurrency method employed removes the computational overhead needed to process transactions increasing the latency.

To fully examine VoltDB, PyTPCC (a modified benchmark of TPC-C) was used to conduct a series of experiments. These experiments test VoltDB's performance with horizontal scaling, vertical scaling, various numbers of clients, partition keys, multiple replicas, batch transactions, and, finally, how consistent it is after each test. The results of these experiments showed that VoltDB had the best performance when the transactions touched a small section of the data, and that it was always consistent.

Table of Contents

List of Figures

Section 1

Introduction

In recent years, a new type of database management system has been developed, called NewSQL[1]. It is a type of relational database that horizontally scales. The goal of NewSQL is to combine the best parts of SQL and NoSQL. NewSQL has native ACID transactions like SQL and is able to horizontally scale like NoSQL. Essentially, NewSQL should always be consistent. To implement this goal a variety of strategies were implemented. Two common features that have come to define the term NewSQL are shared-nothing distributed architecture and a lock-free concurrency system[1]. The shared-nothing architecture allows for increased parallelism and sidesteps the need for extensive consensus algorithms. Therefore adding new machines to the NewSQL server, to increase storage capacity, should also increase the performance of the system. Having a lock-free concurrency system means there is significantly less overhead involved with processing transactions. While locks are powerful tools, they also come with complex challenges like deadlock detection/prevention. Removing the use of locks decreases the latency of transactions due to administrative overhead.

Most NewSQL[1] databases are designed for tasks like online transaction processing(OLTP), where transactions are repetitive, short and touch a small amount of data. Traditional SQL databases are capable of quickly and efficiently handling this type of workload when the amount of users and data is small, however, it quickly becomes expensive(by buying a more powerful machine) or slow (by using multiple machines with a consensus algorithm that guarantees high consistency) at higher amounts of users and data. NoSQL databases are not traditionally highly consistent[1], which is not ideal for online transaction processing(OLTP) since many OLTP workloads are handling data sensitive to consistency issues, e.g. ATM banking systems. While there have been new features coming out for NoSQL databases that increase the consistency guarantees, they tradeoff speed/availability for the increased consistency. NewSQL databases are better suited for OLTP workloads because they are actually designed to perform better over multiple machines and are highly consistent by default.

The NewSQL database that will be the focus of this project is VoltDB[2]. VoltDB is a popular NewSQL database and is directly based on the oldest example of a NewSQL database, H-store[3]. VoltDB is a NewSQL database that has a share-nothing distributed architecture and a single threaded access pattern that avoids the need for locks to maintain concurrency. The objective of this project is to benchmark VoltDB to measure and analyze the performance of scaling and multi-row transactions of a NewSQL type database under different factors using PyTPCC[4], a modified version of TPC-C[5]. This project covers a substantial number of experiments, testing VoltDB’s ability to horizontally scale, vertically scale, handle large numbers of clients, perform with different partitioning keys, perform with different numbers of replicas, perform with/without batch transactions, and stay consistent after each test.

TPC-C is a benchmark designed for traditional relational databases, i.e. SQL databases. It realistically simulates the workload of a wholesale supplier selling and delivering items to customers over the internet. In other words, the workload simulates an online transaction processing workload. TPC-C does this using the smallest set of transactions and tables possible while still maintaining the complexities found in an actual wholesale supplier's workload. The limited number of transactions guides testers to performance issues of the tested database without overburdening them with multiple similar transaction types commonly found in a real database implementation.

Section 2

Related Works

NewSQL, a subsection of database management systems, is essentially the next generation of NoSQL, a separate subset of database management systems known for their ability to scale. The goal of NewSQL is to have the scalability of NoSQL with ACID[1] guarantees for transactions. To implement this goal, a lock-free concurrency system is used along with a shared-nothing distributed architecture[1]. The shared-nothing architecture allows for increased parallelism and sidesteps the need for extensive consensus algorithms. This design does, however, need a special method to remain available.

C-store[6] is a column store database optimized for read heavy workloads. It was the first to define and implement K-safety, which is defined as a system that can tolerate k number of servers failing. Using K-safety, C-store was able to achieve high availability and improve the performance of the database. The variable k can be set by the user and is the replication factor of the database[6].

H-Store[3] is the precursor to VoltDB. It is a highly distributed row store with a shared-nothing architecture that stores data in main memory. The architecture of H-Store uses multiple single thread machines to create a serializable and distributed cluster. H-Store uses C-Store’s K-safety to increase its availability and durability. In order to optimize frequently reused transactions, H-Store implements a stored procedure. A stored procedure is a predefined function that contains SQL commands as well as regular programming code[3]. In a benchmark study by Abadi et al.[7] that used

TPC-C, H-Store was found to be faster than a standard relational database management system by a factor of 82 on a single machine. Abadi et al. attribute H-store's success to a number of features and optimizations. Firstly, H-store does its storage and processing all in main memory storage rather than having to go to disk storage which takes significantly longer. A standard relational database management system was created in an era where main memory was significantly more expensive and therefore a scarce resource. In comparison, main memory as a resource is now relatively cheap and plentiful. Secondly, H-store was designed for online transaction processing use cases, for which TPC-C is a widely recognized benchmark to test databases with an online transaction processing workload. One of the properties of online transaction processing workloads is that transactions are short[1,7,8]. Short transactions reduce the benefits of multithreaded databases because it takes more time and resources to handle the isolation guarantees of multiple statements being executed at once than it does to run each transaction to completion back to back. A single threaded database reduces the overall overhead by eliminating the need for algorithms designed to allow multiple threads to work at the same time like resource managers and concurrent B-trees. Finally, H-store was built from the ground up to have a high availability design by using a shared-nothing architecture and peer-to-peer network. A peer-to-peer network allows any node in the system to handle a client request coupled with the shared-nothing approach means that a single node can potentially handle a client's request without having to coordinate with multiple nodes[5]. NewSQL is a relatively new style of database management system so there are limited tools designed specifically to benchmark NewSQL databases.

Yahoo! Cloud Serving Benchmark (YCSB)[9] is a benchmarking tool designed to specifically test NoSQL's distributed database systems. Each NoSQL database may have a variety of workloads that they specialize in and traditional benchmarks designed for SQL databases are not designed to test those workloads. YCSB is designed for workloads that are done online with read/write access to the database. The main feature of YCSB is that it is open source and extendable. YCSB comes with five workloads already defined, each targeting a different type of workload. For example, one of the workloads is read-only while another is an update heavy workload[10].

Jacobsen et al.[10] did a benchmark analysis of VoltDB v2.1.3 and several NoSQL databases with a YCSB benchmark. They used the default configuration for the database, a single multi-partitioned transaction and the rest of the transactions being single partitioned. The lone multi partitioned transaction was a scanning operation. They found VoltDB did not scale on a workload with mostly read operations, a workload with mostly write operations, or on a workload with a mix of read/write operations. Jacobsen et al. came to the conclusion that the synchronous querying in the YCSB benchmark is not suitable for VoltDB's distributed database configuration and instead suggests that asynchronous communication with the servers is necessary to achieve scalability[3].

In Kamsky et al.[11] YCSB was considered for benchmarking MongoDB[11]. However, it was ultimately considered a bad fit for MongoDB because it was made for benchmarking key-value stores. This limited the benchmark to isolated read and write operations and did not fit the general use case of the database. The general use case for MongoDB took advantage of its ability to perform multi document transactions. Instead, TPC-C was used to benchmark because it better represented the complex use cases that MongoDB regularly handles. They did, however, have to modify TPC-C to use MongoDB's best practices while retaining the same workload and ACID guarantees[11]. In Panpaliya et al.[8] this benchmark experiment was further extended to test MongoDB with TPC-C on a shared cluster to determine how well MongoDB horizontally scaled.

This paper will expand upon the benchmarking of VoltDB in its modern form as opposed to the earlier version benchmarked by Jacobsen et al which treated VoltDB like a key value store. This paper will be using VoltDB version 11.0, a significantly updated version compared to VoltDB version 2.1.3 used by Jacobsen et al. [10]. To better emulate and understand VoltDB's typical workload, a modified version of TPC-C will be used similar to the modified version discussed in [8,11]. This is opposed to using YCSB which, while designed for distributed databases, uses simple statements, instead of transactions, that are too simplistic to cover the wider range of features that VoltDB offers as used inCooper et al. and Jacobsen et al.[9,10]. Using TPC-C this paper gets a more complete sense of VoltDB's abilities by doing a wide breadth of tests covering: horizontal scaling, vertical scaling, memory size, batch transactions, partitioning schemes, arrival rate, and replication.

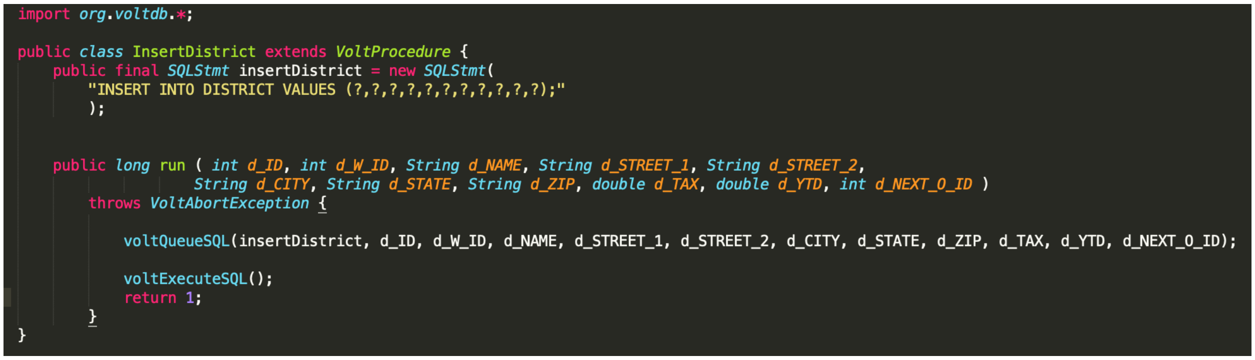
Section 3

Background

3.1 VoltDB Architecture

VoltDB is a distributed peer-to-peer row-store-based NewSQL database management system which uses a relational database system that runs on a cluster of computers that can efficiently scale horizontally because of a share-nothing architecture. VoltDB requires a schema that defines the indexes, partitioning columns, stored procedures, tables, and rows of the database. VoltDB supports a subset of the SQL query language that allows users to search, insert, and manipulate the data stored in the cluster.

A stored procedure is how VoltDB does an ACID transaction on the database. It uses predefined standard SQL language as well as Java code to evaluate and manipulate the database. Note that memory is not persistent across transactions, e.g. a dynamically set variable within the stored procedure will be reset once the transaction is complete



```java
import org.voltdb.*;

public class InsertDistrict extends VoltProcedure {
    public final SQLStmt insertDistrict = new SQLStmt(
        "INSERT INTO DISTRICT VALUES (?,?,?,?,?,?,?,?,?,?,?);"
        );


    public long run ( int d_ID, int d_W_ID, String d_NAME, String d_STREET_1, String d_STREET_2,
                String d_CITY, String d_STATE, String d_ZIP, double d_TAX, double d_YTD, int d_NEXT_O_ID )
        throws VoltAbortException {

            voltQueueSQL(insertDistrict, d_ID, d_W_ID, d_NAME, d_STREET_1, d_STREET_2, d_CITY, d_STATE, d_ZIP, d_TAX, d_YTD, d_NEXT_O_ID);

            voltExecuteSQL();
            return 1;
        }
}
```

Figure 1: Example of a simple stored procedure.

All transactions must be deterministic. In this specific context, a transaction is deterministic if all partitions are able to come to the same result. An example of a deterministic function would be every row in a partitioned table is incremented by the same random number. An example of a nondeterministic function would be every row in one partition is incremented by a different random number from every other partition. So instead of the entire table being incremented by one random number, each partition is incremented by a different number. If a nondeterministic function is needed, VoltDB has provided some functions like rand() and date() that simulate nondeterminism while also allowing for a deterministic transaction across multiple partitions by coordinating the resulting value such that all partitions get the same result when completing the same multi partitioned transaction. This is important because in a peer-to-peer system a nondeterministic function could cause inconsistency across partitions and replicas. While it is possible to perform ad hoc queries and manipulations to the database, a stored procedure should be used if it is frequent, performance sensitive, or complex. Each node has a copy of all stored procedures.

In VoltDB, a cluster is a group of multiple physical computers called hosts, and a node is a single CPU core on a physical computer. Since each node is a single core, there is no way to multi-thread on a node, therefore all reads and writes on the data happen one at a time on a node. This means that the consistency level of the database is always linearizable, the highest possible consistency level on a node.

Since VoltDB is peer-to-peer, every host has a client interface that can handle, or forward any client request. Each node in the system has all the information necessary to serve any user requests related to the partitions it stores; this includes indexes and stored procedures. If a node is not able to complete the request or is unable to do so alone, its host has all the routing information needed to send the request to the proper partition(s).

3.1.1 Indexing

Indexes are stored as B-trees in VoltDB and are created using the standard SQL language. Each partition has its own index as opposed to a single index over the entire database. Since the index is not global and partitions do not share data, there is no way to guarantee global uniqueness on any row except for the partitioning column which is global.

3.1.2 Partitioning

The partitioning column is a column in a table that is used to split the rows into different chunks. Only one column can be used as a partitioning column. If a user wants to use multiple columns as a partitioning then they must combine multiple columns into a single column. These chunks, which are non overlapping parts of the data split by the partitioning column, are used to horizontally scale the database by having a single machine be responsible for one or more chunks. This is called partitioning. Each node in the system is responsible for some number of the chunks, called partitions, which are nodes that contain a nonzero number of chunks of the data and handles all of the reads and writes to that partition. For optimal results, the cardinality for the partitioning column should not be too large or too small relative to the number of nodes in the system. If the cardinality is too small, then there are not enough partitions for each node, therefore limiting the throughput of the system. If there are too many partitions, the overhead of keeping track of each partition becomes burdensome. However, if a user wants to preserve the uniqueness of a column, they must choose that column as the partitioning column. This does mean that the cardinality will be high, however, the partitioning column is the only value that VoltDB keeps track of in every partition. If the partitioning column is unique then all the partitions can contain chunks of size one. In VoltDB because of the share-nothing architecture, each partition is stored on a single node and no other node has the same partition unless the entire node is replicated using K-safety.

3.1.3 Replication

VoltDB either partitions or replicates a table as a default setting. If a partitioning column is not stated, then a table is replicated. That is, without partitioning, a replicated table is, in its entirety, copied to every partition. Since the entire table is copied, a replicated table should ideally be small and read-only once it has been set up initially. If there is a change made to the replicated table, then all partitions must be updated with that information. Since all of the partitions have to coordinate together rather than a subset of them, this type of transaction has the highest possible latency in VoltDB. If a user wants to replicate a partition because they want to make the partition durable, then K-safety, which is a nonstandard feature, is used.

K-safety is the mechanism that VoltDB uses to replicate partitions of the database. This is not a default function of VoltDB; the user must specify how many copies of the partitions are needed in the terms of a K value. For example a K value of 1 means there are 2 replicas of a partition. Essentially the number of partition replicas is equal to the K value plus one. Each replica of the partition is a perfect synchronous copy and has no distinguishing features that separate them from each other. The server can handle up to k hosts failing before it is unable to service requests. Each command that contains a write operation is run separately on each partition replica; a read command only has to go to one partition replica. To avoid inconsistencies between partitions, all commands must be run in the same order. Read commands do not affect the consistency so they can be run on a single partition safely. All reads will be linearly consistent because stored procedures are ACID (more information about how they are ACID is in Section 3.1.4). It is recommended that the number of replicas be odd. In the event of a network failure or network partition, the block of partitions still connected to the majority of replicas continues to service user requests. If the K-safety is even and the split is also even, then one of the blocks will continue based on the internal numbering system of the nodes.

3.1.4 Read and Write Path

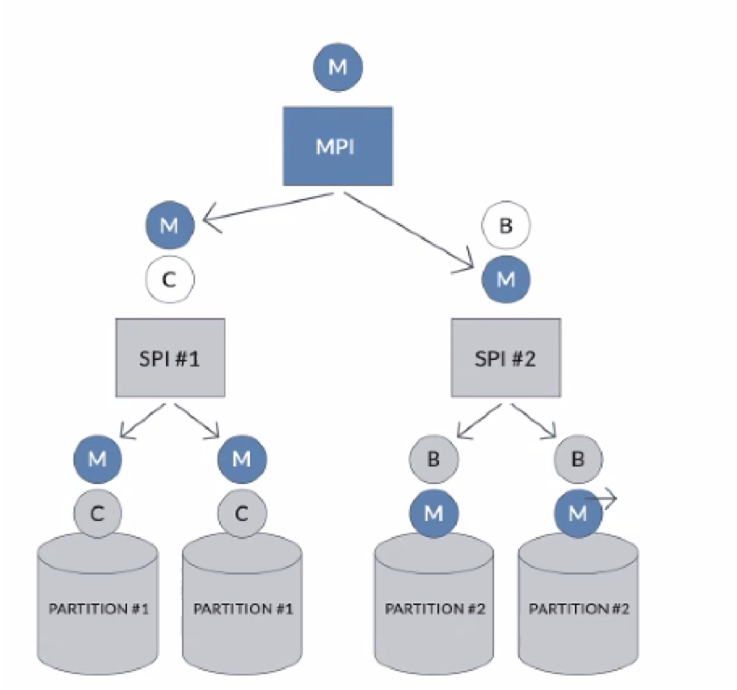


Figure 2: Single partition initiator and multi-partition initiator[12]

In VoltDB, there are three paths that a transaction, called a stored procedure, can take: single partition, multi-partition, and one-shot. A user can explicitly state in the definition of a stored procedure that it will always be a single partition transaction. However, if the user does not explicitly define that the transaction is a single partition transaction, the query planner will determine what type of transaction it is with the default being a multi-partitioned transaction. A single partition transaction is a transaction that is completely contained within a single partition and requires no information from any other partition. Since there can be more than one replica of a partition, single partition transactions are handled by the Single Partition Initiator(SPI). The SPI is a coordinator that determines the order of the transactions for every partition in the replica set. A multi-partition transaction is a transaction that requires information from multiple partitions in order to complete the transaction. The transaction is split into pieces that a single partition is able to complete on its own by the Multi-Partition Initiator(MPI) and sent to the SPI of that partition. To complete a write transaction, the MPI coordinates the transaction across the SPIs in a two phase commit procedure. Using a two phase commit is slow but it allows the system to stay linearly consistent because it is ACID. Multi-partition reads are allowed to proceed without waiting for a commit because they do not affect the consistency across partitions. One-shot transactions are transactions that can affect multiple partitions but do not require any additional information from another partition. For example, incrementing every column in a partitioned table by one does not require any information from any other partition and can affect many partitions, making it a one-shot transaction. This means once the MPI has split the transaction up, it can be treated as if it were a single partition transaction.

3.2 PyTPCC/TPC-C benchmark

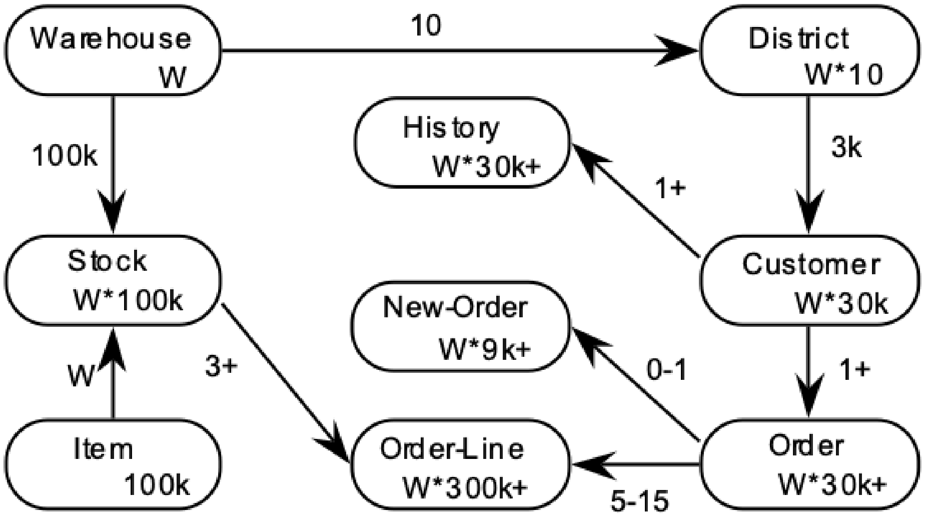


Figure 3:TPC-C/PyTPCC table relationship[5]

A project called PyTPCC, originally made by students of Brown University, implements TPC-C in python for the express purpose of benchmarking NoSQL databases. The project supports a number of NoSQL databases and uses a modular design that allows others to create drivers for any database of their choosing[10]. Since NewSQL is a fusion of SQL and NoSQL, PyTPCC is well designed to benchmark NewSQL databases.

TPC-C is a benchmark designed to simulate an online transaction processing (OLTP) workload on any type of machine regardless of the underlying hardware. Specifically, TPC-C simulates the activity of processing orders for a wholesale supplier without specifying what type of industry it is. The schema is made up of nine tables with five basic transactions manipulating and straining the database. With the exception of the Item table, all of the other tables increase with the number of warehouses in the

Warehouse table. Figure 3 shows the relationship between TPC-C tables where the arrows indicate the number of rows that are created per row of the parent table.

The names of the TPC-C tables are warehouse, district, customer, history, order, new-order, order-line, stock, and item. The five transactions are as follows:

1. The New-Order Transaction: This transaction is a mid-weight read-write transaction and introduces a variable load on the system. This transaction enters a complete order from a customer with variable items into the database. Seven different tables are used to complete the transaction: warehouse, district, customer, order, new-order, stock, and item. Depending on the number of items, let's call it *I*, there are 2 + *I* reads, 1 + *I* updates, and 2 + *I* inserts. This transaction is executed at a high frequency, being 45% of the workload.
2. The Payment Transaction: This transaction is a light-weight read-write transaction that uses non-primary key access a majority of the time. The transaction updates the customer’s balance and updates the warehouse and district tables’ year-to-date sales balance. Four different tables are used to complete the transaction: warehouse, district, customer, and history. The customer table can be accessed via the customer id or by the customer’s last name. In the first case, there are three updates and one insert. In the second case, there are an average of two reads, three updates, and one insert. This transaction is executed at a high frequency, being 43% of the workload.

3. The Order-Status Transaction: This transaction finds out the status of a customer’s previous order. It represents a mid-weight read only transaction with non-primary key access a majority of the time. Three different tables are used to complete the transaction: customer, order, and order-line. The customer table can be accessed via the customer id or by the customer’s last name. In the first case, there are two reads and one read the size of the number of items in the previous order. In the second case, there are an average of four reads and one read the size of the number of items in the previous order. This transaction is executed at low frequency, being only four percent of the workload.
4. The Delivery Transaction: This is a batch transaction that consists of up to ten transactions within the batch operation. Each transaction within the batch delivers/processes new orders created in the New-Order transaction. Four different tables are used to complete the transaction: customer, order, new-order, and order-line. There are one read, 2 + items-per-order updates, and one deletion. This transaction is executed at low frequency, being only four percent of the workload. It is not intended to be executed interactively as if a user was calling the transaction, it is instead executed using a queuing method.
5. The Stock-Level Transaction: This transaction finds recently purchased items whose stocks are lower than a specified value. This is a heavy read only transaction that reads from three different tables: district, order-line,

and stock. There is a minimum of 41 rows read and a maximum of 401 rows read. This transaction is executed at a low frequency, being only four percent of the workload.

One of the settings that can be set by the user of the benchmark is the number of clients accessing the database. Each client is a representation of an actual person using the database, they call transactions and get responses independent from other clients. The arrival rate of transactions to the database is increased by increasing the number of clients accessing the database.

The key difference between TPC-C and PyTPCC is that TPC-C has minimum workload percentages for all transactions except for New Order while PyTPCC has a specific percentage for each transaction. In TPCC the Payment transaction has a minimum of 43% of the workload and the Order-Status, Delivery, and Stock-Level transactions must each be at least 4% of the workload. PyTPCC explicitly requires that each transaction be a specific percentage of the workload. The workload percentages for PyTPCC are 45% New Order transactions, 43% Payment transactions, and 4% each for Delivery, Order-Status, Delivery, and Stock-Level transactions

Section 4

Experiment Setup

Unless specifically mentioned otherwise, all VoltDB servers for the experiments were done on a set of AWS c4.xlarge instances. These instances are optimized for compute heavy workloads. They have High frequency Intel Xeon E5-2666 v3 (Haswell) processors, four virtual CPUs, and 7.5 GiB of memory. The servers are placed in a cluster placement group; this strategy groups the instances within a single availability zone. This strategy has the benefit of increasing network performance between the instances in the group. Each instance had Java version 1.8.0, ant, Extra Packages for Enterprise Linux (EPEL) version 7, Python 2.7, and VoltDB version 11.0 installed. Unless explicitly stated otherwise the k factor is equal to one and the number of sites per host is equal to the number of virtual CPUs. The only settings that are changed from the default settings are explicitly mentioned. If a setting is not mentioned then it is kept at VoltDB's default setting.

The server instance that runs PyTPCC was run on a t3.2xlarge instance, which was not in the placement group of VoltDB instances but was in the same availability zone. This instance is a general purpose instance with an Intel Xeon Scalable processor (Skylake 8175M or Cascade Lake 8259CL), eight virtual CPUs, and 32 GiB of memory. This instance had Python 2.7, numpy, the VoltDB client driver, and PyTPCC installed.

The benchmark measures TPMC, the number of New Order transactions executed per minute, also called the business throughput. For all experiments the New

Order parameter generator was modified so that the customer’s warehouse and the ordering warehouse are always equal. This was done to reduce the number of factors for analysis. Unless stated otherwise, 300 clients were simulated for all experiments. 300 clients were chosen because it is the point when throughput does not increase by more than 3% per 100 clients added for all experiments.

Section 5

Results/Analysis

5.1 Impact of Partitioning Column Selection

By having different shard keys the system will split the data up in different ways potentially increasing or decreasing the efficiency of the system. Each table has its own shard key so different combinations of keys on different tables are possible. It is important to note that while VoltDB does not support foreign keys, it does place keys with the same value into the same partition. For example, say that the History table and the New Order table are partitioned by the H_C_W_ID column and the NO_W_ID column respectively. In the eyes of the database these two columns are not related, however, since both are of the same type, integer, if a row in their respective tables share the same partitioning value, let's say 1, then then the rows from both tables go to the same partition. This experiment is done to explore what happens when a poorly chosen partitioning strategy is used versus a well chosen strategy. For this experiment two general partitioning strategies were chosen: partitioning on the District ID and partitioning on the Warehouse ID.

For the District ID partition key, the Warehouse table and the District table are replicated across all partitions. If a table has a column associated with the District ID in the table, it is partitioned on that column. If a table does not have a column associated with the District ID, it is partitioned on a column associated with the Item ID.

For the Warehouse ID partition key, the Item table is replicated across all partitions. If a table has a column associated with the Warehouse ID in the table, it is partitioned on that column. This rule is sufficient to cover all tables except for the Item table which is already being replicated to all partitions and therefore has no partitioning value.

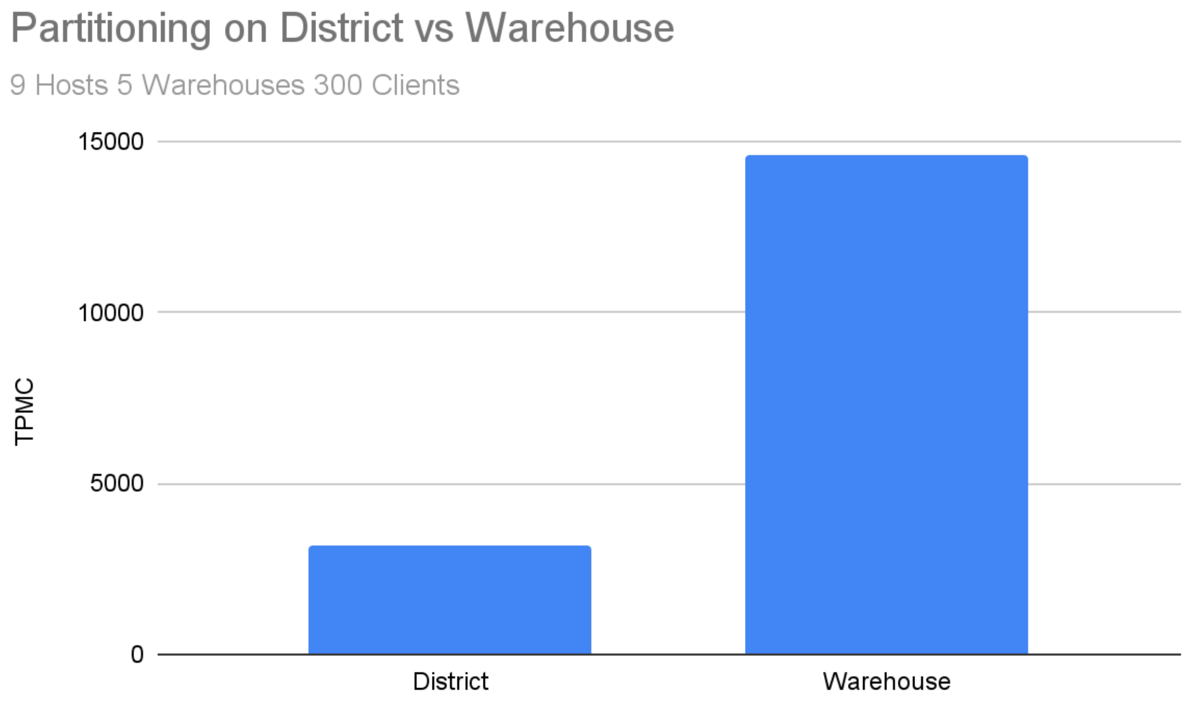


Figure 4: Impact of partitioning column selection

The database partition strategy where the tables were partitioned on the Warehouse ID had 3.5 times greater throughput than the partition strategy where the tables were partitioned on the District ID. The reason for this large difference in throughput is because the Warehouse ID strategy was designed to have all transactions be single partition, while the District ID had mostly multi partitioned transactions.

In the partitioning strategy that partitions mainly on the District ID, the New Order transaction interacts with all of the tables in the database, with the exception of the

History table. The Stock Level transaction interacts with three tables: the Order table, the Stock table, and the Item table. Both the Item table and the Stock table are partitioned on the columns associated with the Item ID, which can be any one of 200,000 unique integer ids. The District ID is in the range of 1 to 10 inclusive. The only Item IDs guaranteed to be in the same partition as any given District ID are the Item IDs that share the same value as that given District ID. In other words, 199,999 Item IDs are not guaranteed to be in the same partition as the District ID. This means that both the New Order transaction and the Stock Level transaction are not guaranteed, or likely to be, one shot transactions.

The Payment transaction updates the Warehouse table and District table. Both of these tables are replicated to every partition in the database and any update to those tables must be coordinated with every partition in the database. This transaction is guaranteed to be a multi partitioned transaction.

The Delivery transaction uses ten different District IDs in the transaction. This guarantees that this transaction will be a multi partitioned transaction.

The Order Status transaction is guaranteed to be a one shot transaction because it does not involve multiple District IDs, the Stock table, the Item table, updating the Warehouse table, or updating the District table.

The only transaction that is guaranteed to be a single partition transaction is the Order Status transaction. This transaction makes up 4% of the overall workload, which means 96% of the workload will most likely be made up of multi partitioned transactions. The New Order transaction, which makes up 45% of the workload, is not guaranteed to be a multi partitioned transaction the Multi-Partition Initiator(MPI) could detect that it can be done without any extra information. However, it is not guaranteed to detect when a transaction is a single partition transaction, and therefore even a transaction that could technically be a single partition transaction is not guaranteed to be treated like one.

All transactions with the Warehouse ID partition strategy are guaranteed to be a single partition transaction. None of the transactions use multiple Warehouse IDs, so with the sole exception of the Item table, all information associated with a single Warehouse ID is guaranteed to be in the same partition. The Item table is replicated across all partitions, however, all statements involving the Item table are read only so there is never a need to coordinate all of the partitions to update the table. The Delivery transaction updates all of the districts associated with a single Warehouse ID.

Since the Warehouse ID strategy had one hundred percent single partition transactions the database was able to do multiple transactions in parallel. This increase in parallelism means that partitions are not waiting idly while other partitions complete all of their transactions before coordinating with the waiting partition to complete the transaction it is waiting on. While the partition is waiting, reducing utilization, it does nothing because it must maintain the order that the transactions came to it to remain consistent.

5.2 Impact of Batch-like Transactions

VoltDB focuses mainly on online transaction processing(OLTP) which uses short transactions that only touch a small subset of the data. This means a batch-like transaction like the Delivery transaction, that takes longer to do and touches a wide spread of data, could have a negative impact on TMPC, i.e the throughput of VoltDB. This experiment is to evaluate how well the database performs with the original Delivery transaction, a modified version of the Delivery transaction, and finally without the Delivery transaction to see the impact of batch-like transactions on the throughput. The modified version of the Delivery transaction breaks the transaction up into ten separate transactions. Each of the ten transactions focuses on a single District ID within the Warehouse ID. Those ten transactions are still counted as a single transaction by PyTPCC because it covers the same functionality as the original transaction, this does not change the workload percentages explained in Section 3.2. For the case where the Delivery transaction is eliminated from the workload, the workload percentages explained in section 3.2 are adjusted to account for the Delivery transaction being removed. To account for this elimination the Order Status transaction is done 8% of the time instead of 4% of the time. The Order Status transaction was chosen to increase in frequency of execution because it is a read-only transaction that was unlikely to add a greater burden on the database by increasing its rate of execution. More importantly, the Order Status transaction is not the focus of the TPMC measurement. If the New Order transaction was chosen then the throughput would increase no matter what simply because it was being run more often.

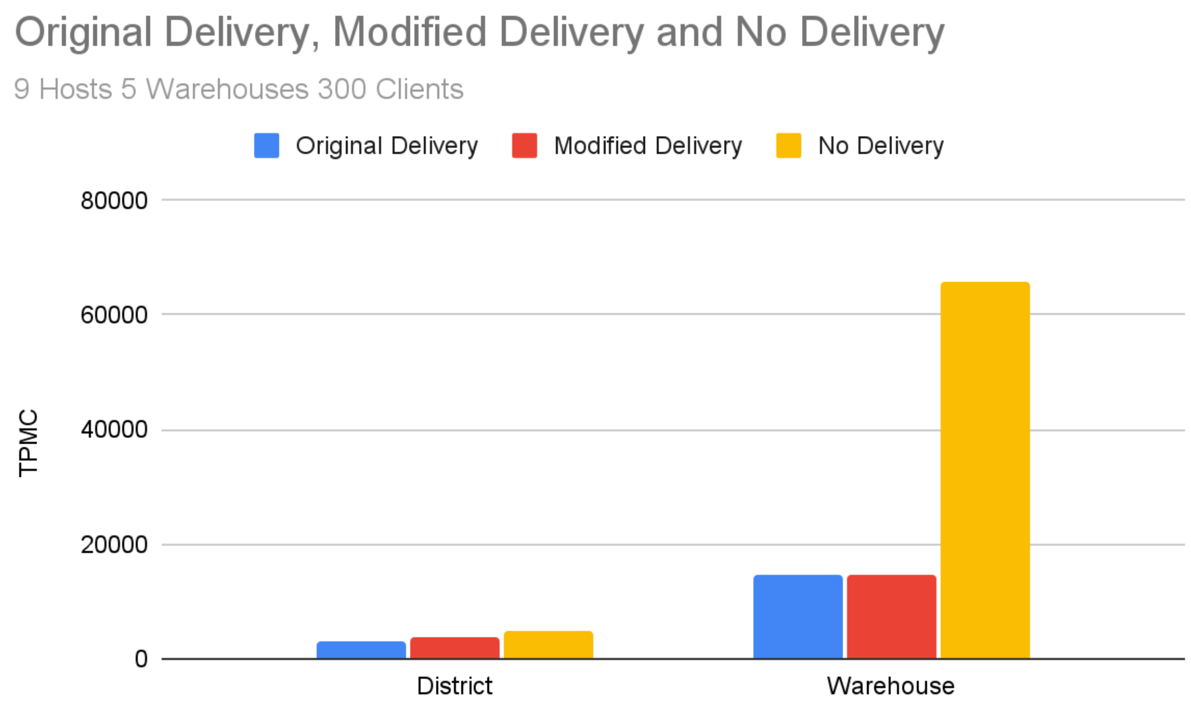


Figure 5: Impact of batch-like transactions

For the strategy that partitions on District ID the modified Delivery had an increased throughput of 21.93% and the maximum latency of the District transaction was roughly 8 times faster than the rest of the transactions. The reason that the throughput increased is because the modified Delivery transaction is guaranteed to be a single partition transaction. Single partition transactions increase parallelism and means that multiple unique partitions do not have to coordinate to complete the transaction. This is reflected in the latency, it goes down because the transaction requires no information that is not found on the partition which means it can immediately start and complete the transaction.

For the strategy that partitions on Warehouse ID the modified Delivery did not have a statistically significant change in throughput and the maximum latency of the District transaction was roughly 6 times slower than the rest of the transactions. The original Delivery transaction is already a single partition transaction; this means parallelism is not increased by dividing the transaction up in the modified transaction. Latency is increased because each of the ten transactions go to the server one at a time and wait until the previous transaction returns to send the next transaction. In other words, instead of a single round trip, the modified transaction has ten round trips to complete the same functionality as the original transaction. The only reason throughput does not suffer is because during the round trip time other transactions are being completed making up for the loss of efficiency by the modified Delivery transaction.

For the strategy that partitions on Warehouse ID where the Delivery transaction is eliminated from the workload, throughput increased, compared to original Delivery, by 3.5 times. For the strategy that partitions on District ID where the Delivery transaction is eliminated from the workload, throughput increased, compared to original Delivery, by 56.80%. The original Delivery transaction is a bulk transaction; VoltDB is not optimized for bulk transactions. At minimum, the Delivery transaction for the Warehouse ID partitioning strategy takes 17 times longer than any other transaction. At the 99th latency percentile, this gap is reduced but it still takes 5% longer than any other transaction. This is particularly significant because the other transactions do not have a gap that is statistically significant. By eliminating this bulk transaction throughput is increased beyond the four percent of the workload that the Delivery transaction makes up because in the time it takes to complete a single Delivery transaction multiple other transactions can be completed. There is a similar case for the District ID partitioning strategy, however, the benefit of removing the bulk transaction is lessened because most of the remaining transactions are not guaranteed to be single partition transactions. This means those multi partitioned transactions are still waiting on multiple transactions across multiple partitions which means eliminating a single long transaction does not confer the same benefits.

5.3 Vertical Scaling

This experiment features three different instances from the AWS c4 series: c4.large, c4.xlarge, and c4.2xlarge. The difference among these instance types are the number of virtual CPUs and the amount of memory. The c4.large instance has two virtual CPUs and 3.75 GiB of memory. The c4.xlarge instance has four virtual CPUs and 7.5 GiB of memory. Finally, the c4.2xlarge instance has eight virtual CPUs and 15 GiB of memory.

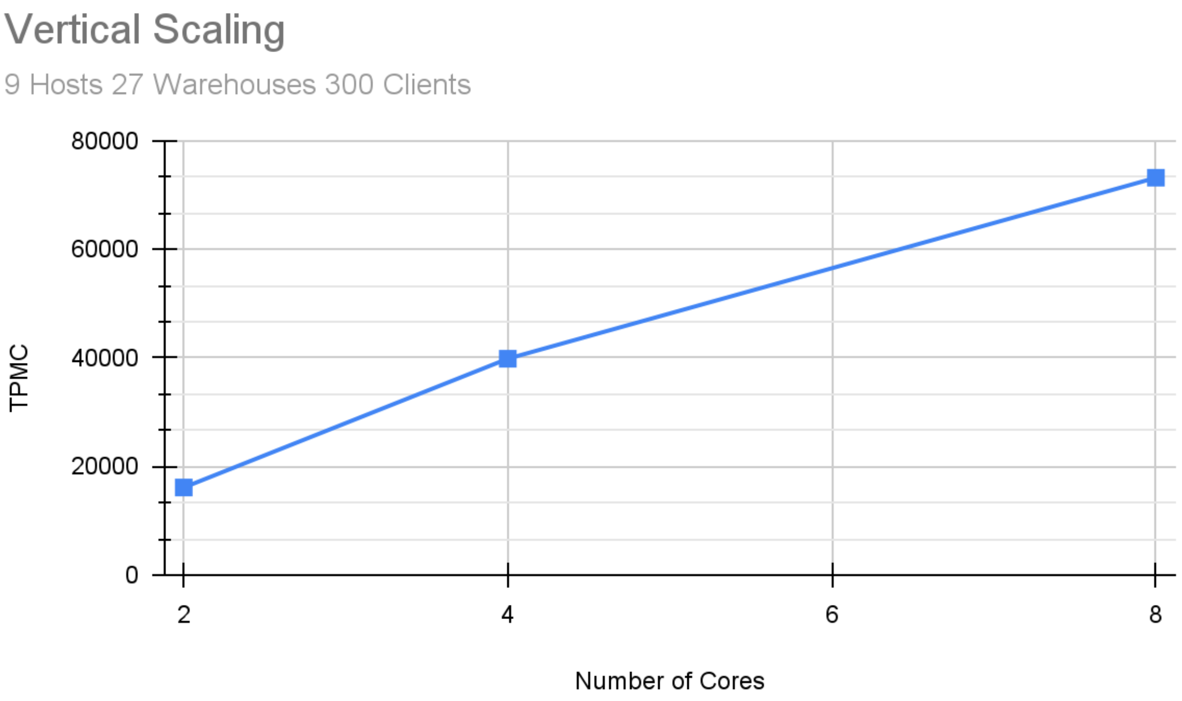


Figure 6: Impact of vertical scaling

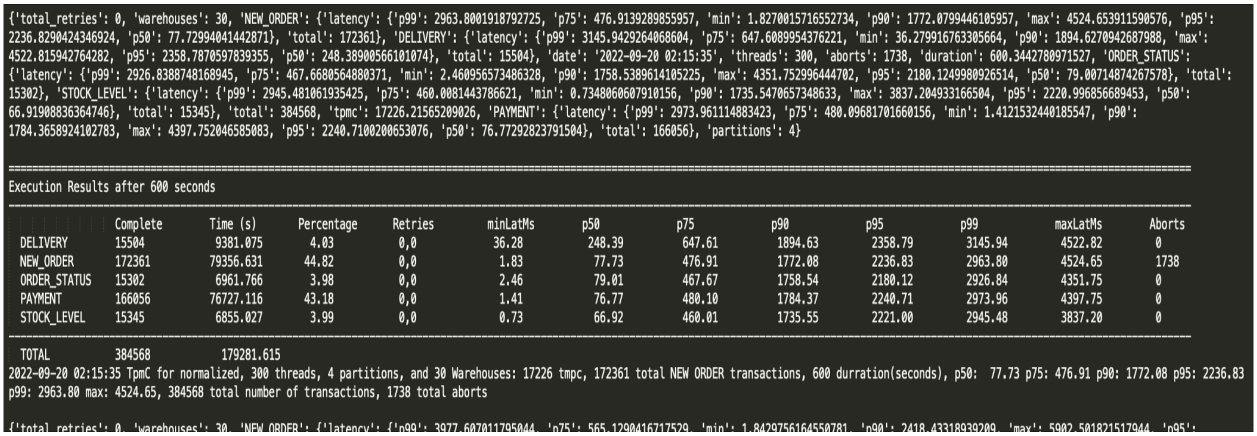
As the number of cores increases the throughput increases. From 2 virtual CPUs to 4 virtual CPUs throughput increased by around 145%. From 4 virtual CPUs to 8 virtual CPUs throughput increased by around 84%. This large increase in throughput is caused by increased parallelism. At 2 virtual CPUs and 9 hosts, there are 9 possible partitions. At 4 virtual CPUs and 9 hosts, there are 18 possible partitions. At 8 virtual CPUs and 9 hosts, there are 36 possible partitions. Each time the number of virtual CPUs doubled, so did the number of partitions. By increasing the number of partitions the load from the 300 clients is spread out across the cluster and each partition has to serve fewer requests. The throughput did not double when 36 partitions were reached because there are only 27 Warehouse IDs, which means only 27 total partitions can be utilized. This means that the 9 host cluster with 8 virtual CPUs is underutilized with 27 Warehouse IDs because 9 partitions are not being used.

5.4 Consistency Tests

Any database system is only as good as it is consistent with what has actually been committed to it. This experiment measured how often an inconsistency is found by running four tests after each experiment to determine if the database remained consistent. Each test is described in the TPC-C manual and tests the database by checking that values tracked across tables are in fact equal to each other. The tests are as follows:

1. The first test requires that the amount of money made by a warehouse is equal to the sum of the money made by the districts that warehouse is associated with.
2. The second test requires that 1 minus a district's next order id is equal to both the maximum order id and the maximum new order's order id associated with the district.
3. The third test requires that the number of rows in the New Order table associated with a district must be equal to the maximum new order's order id minus the minimum new order's order id plus 1.
4. The fourth test requires that the sum of the order's order line count associated with a district is equal to the number of rows in the Order Line tables associated with that district.

These consistency tests were run after every individual datapoint on every experiment done in this project. While these tests are not exhaustive, they are sufficient in TPC-C eyes to consider any database that passes them to be consistent.



```
{'total_retries': 0, 'warehouses': 30, 'NEW_ORDER': {'latency': {'p99': 2963.8001918792725, 'p75': 476.9139289855957, 'min': 1.8270015716552734, 'p90': 1772.0799446105957, 'max': 4524.653911590576, 'p95':
2236.8290424346924, 'p50': 77.72994041442871}, 'total': 172361}, 'DELIVERY': {'latency': {'p99': 3145.9429264068604, 'p75': 647.6089954376221, 'min': 36.279916763305664, 'p90': 1894.6270942687988, 'max':
4522.815942764282, 'p95': 2358.7870597839355, 'p50': 248.38900566101074}, 'total': 15504}, 'date': '2022-09-20 02:15:35', 'threads': 300, 'aborts': 1738, 'duration': 600.3442780971527, 'ORDER_STATUS':
{'latency': {'p99': 2926.8388748168945, 'p75': 467.6680564880371, 'min': 2.460956573486328, 'p90': 1758.5389614105225, 'max': 4351.752996444702, 'p95': 2180.1249980926514, 'p50': 79.00714874267578}, 'total':
15302}, 'STOCK_LEVEL': {'latency': {'p99': 2945.481061935425, 'p75': 460.0081443786621, 'min': 0.7348060607910156, 'p90': 1735.5470657348633, 'max': 3837.204933166504, 'p95': 2220.996856689453, 'p50':
66.91908836364746}, 'total': 15345}, 'total': 384568, 'tpmc': 17226.21565209026, 'PAYMENT': {'latency': {'p99': 2973.961114883423, 'p75': 480.09681701660156, 'min': 1.4121532440185547, 'p90':
1784.3658924102783, 'max': 4397.752046585083, 'p95': 2240.7100200653076, 'p50': 76.77292823791504}, 'total': 166056}, 'partitions': 4}


==========================================================================================================================================================================
Execution Results after 600 seconds
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------
                  Complete      Time (s)      Percentage     Retries       minLatMs       p50          p75          p90          p95          p99          maxLatMs      Aborts
  DELIVERY        15504          9381.075       4.03          0,0           36.28        248.39       647.61       1894.63      2358.79      3145.94       4522.82       0
  NEW_ORDER       172361        79356.631      44.82          0,0            1.83         77.73       476.91       1772.08      2236.83      2963.80       4524.65       1738
  ORDER_STATUS    15302          6961.766       3.98          0,0            2.46         79.01       467.67       1758.54      2180.12      2926.84       4351.75       0
  PAYMENT         166056        76727.116      43.18          0,0            1.41         76.77       480.10       1784.37      2240.71      2973.96       4397.75       0
  STOCK_LEVEL     15345          6855.027       3.99          0,0            0.73         66.92       460.01       1735.55      2221.00      2945.48       3837.20       0
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------

  TOTAL           384568          179281.615
2022-09-20 02:15:35 TpmC for normalized, 300 threads, 4 partitions, and 30 Warehouses: 17226 tmpc, 172361 total NEW ORDER transactions, 600 durration(seconds), p50:  77.73 p75: 476.91 p90: 1772.08 p95: 2236.83
p99: 2963.80 max: 4524.65, 384568 total number of transactions, 1738 total aborts

{'total_retries': 0, 'warehouses': 30, 'NEW_ORDER': {'latency': {'p99': 3977.607011795044, 'p75': 565.1290416717529, 'min': 1.8429756164550781, 'p90': 2418.43318939209, 'max': 5902.501821517944, 'p95':
```

Figure 7: Example of PyTPCC test that has passed its consistency tests

There were never any failures of the consistency tests. VoltDB was always consistent because every partition was only accessed by a single thread at a time. There was never any point in time where two transactions were running in the same partition at the same time. For another transaction to enter the partition the previous transaction must be committed. That means that for every partition there is an absolute order that transactions occurred in. This guarantees that no past transaction can interfere or change transactions happening in the future, leading to an inconsistent database.

5.5 Horizontal Scaling

In this experiment, a single host was added to the database server one at a time. The throughput at each total number of hosts in the system was captured.

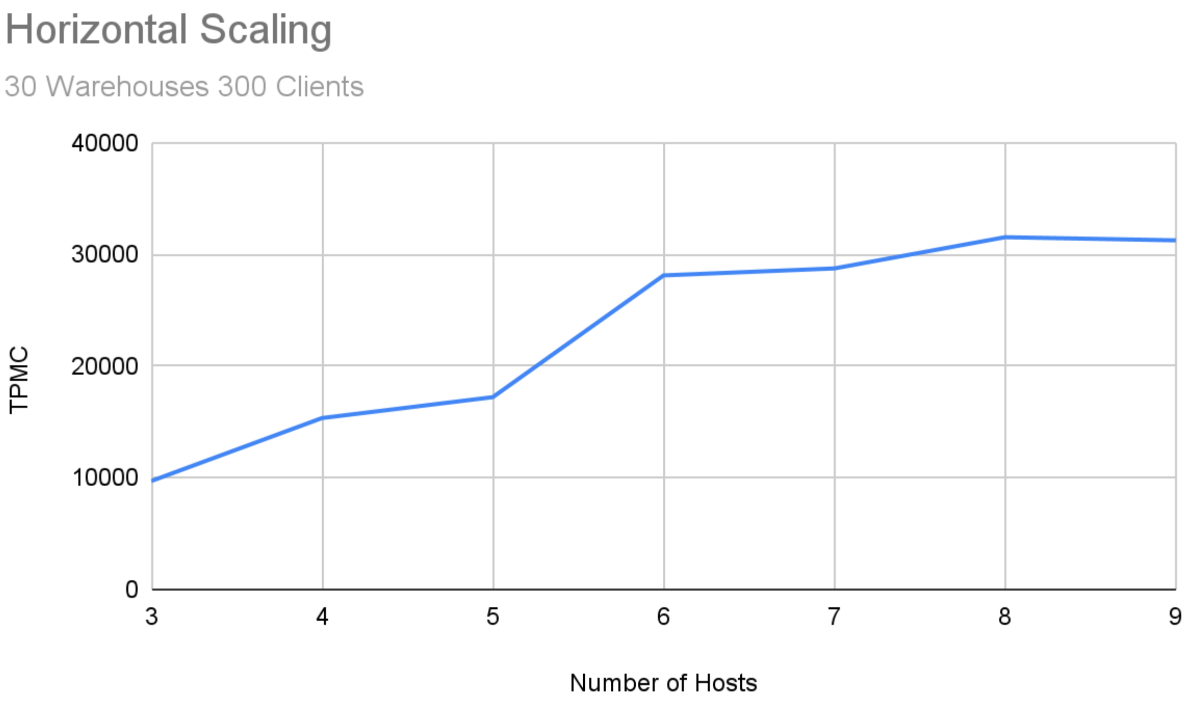


Figure 8: Impact of horizontal scaling

The throughput of the system went up with the number of hosts in the system. More specifically with each host added two unique partitions were created. By increasing the number of unique partitions each partition has less data associated with it. This spreads out the workload and increases parallelism which in turn increases throughput.

5.6 Memory Scaling

In this experiment, the number of warehouses, the unit of scaling for PyTPCC, is varied on four different instances with increasing amounts of memory. The purpose of this experiment is to determine the effect of memory on the database. Unfortunately, AWS does not allow users to change the amount of memory an instance has. The only way to get more memory is to choose a different instance. The instances featured in this experiment are c4.xlarge, c4.2xlarge, d2.xlarge, and r5a.xlarge, each instance type has respectively 7.5 GiB, 15 GiB, 30.5 GiB, and 32 GiB of memory.

Instances c4.xlarge, d2.xlarge, and r5a.xlarge all have 4 virtual CPUs while c4.2xlarge has 8 virtual CPUs. To make instance c4.2xlarge similar to the other instance types the number of sites per host was set to 4. By setting the sites per host to 4 it is guaranteed that only 4 virtual CPUs out of the 8 are used because each site, or partition, is single threaded.

The instance type d2.xlarge is storage optimized. The workload that gets the greatest benefit from this type of instance is one that requires a large number of sequential read and write operations on large amounts of data on local storage. This instance type uses a High-frequency Intel Xeon Scalable Processor (Haswell E5-2676 v3) which is from the same series of processors as the c4 instance types, High-frequency Intel Xeon Scalable Processor (Haswell E5-2666 v3).

The instance type r5a.xlarge is memory optimized. The workload that gets the greatest benefit from this type of instance is one that processes large amounts of data in memory. This instance type uses a completely different type of processor, the AMD EPYC 7000 series processor (AMD EPYC 7571).

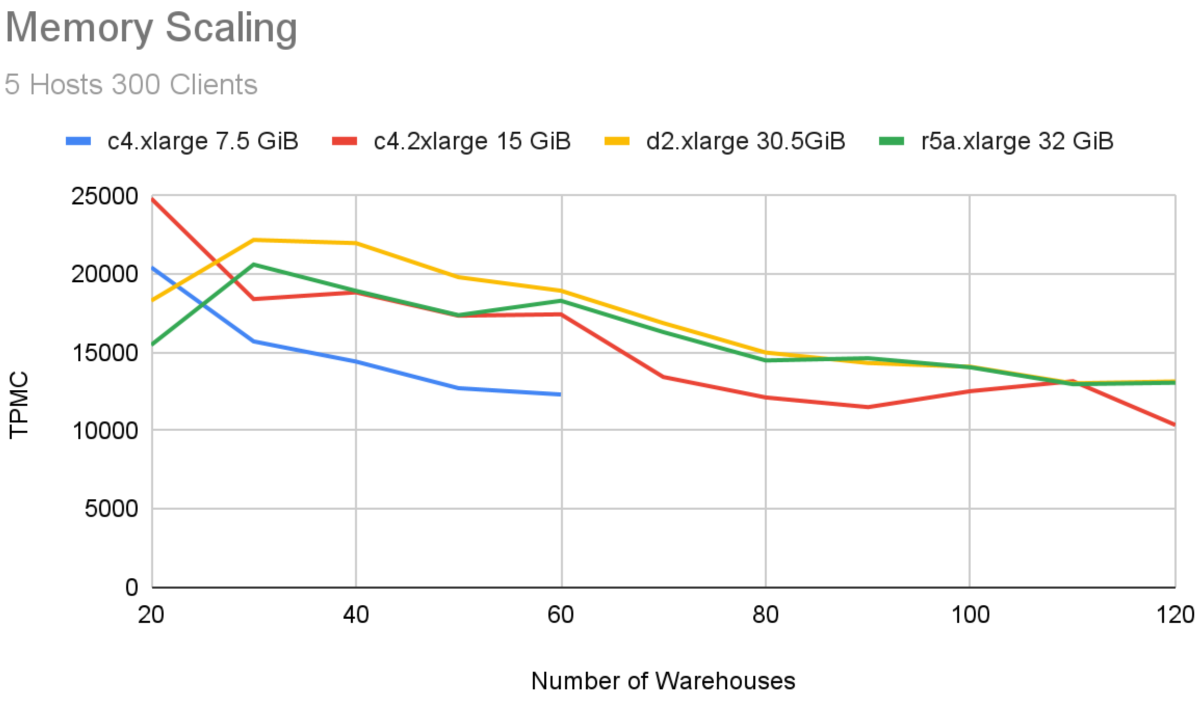


Figure 9: Impact of memory scaling

At a low amount of data in the database the c4 instance types have greater throughput than the d2 or r5a instance types. However, as the amount of data grows the inverse becomes true. The initial greater throughput by the c4 instances is likely because of how they are optimized compared to the r5a and d2 instance types. After 60 warehouses the c4.xlarge instance type is unable to store any more data and freezes all write operations to preserve the data within it. The other databases that have at minimum nearly twice the amount of memory are able to continue handling transactions past the endpoint of this experiment. Once the amount of data exceeds 30 warehouses the instances with a larger amount of memory have a greater throughput. The smaller memory instances have a lower throughput at higher amounts of data because at limited storage any change in the database causes how things are stored to be rearranged to accommodate the new data. The instances with higher memory capacity can add to the data stored without having to optimize how everything is stored because they still have lots of space. This means that the transactions have lower latency, and increase throughput. Increasing the amount of memory each host has increases the amount of data that can be stored, and keeps the throughput up while storing large amounts of data.

5.7 Impact of K factor

The k factor, or the replication factor, is an important factor in the durability and availability of database systems. The higher the k factor the greater the number of host failures that can be absorbed without shutting down the database. In this experiment, the number of unique partitions was held steady while the k factor increased. The equation to find the number of unique partitions is as follows:

(Number of hosts • Number of Sites per host) / (1 + K factor) = Number of Unique Partitions

To hold the number of unique partitions steady the number of hosts within the database cluster is changed when the k factor is changed. The number of sites per host was not changed because that would affect how much memory is available for each partition.

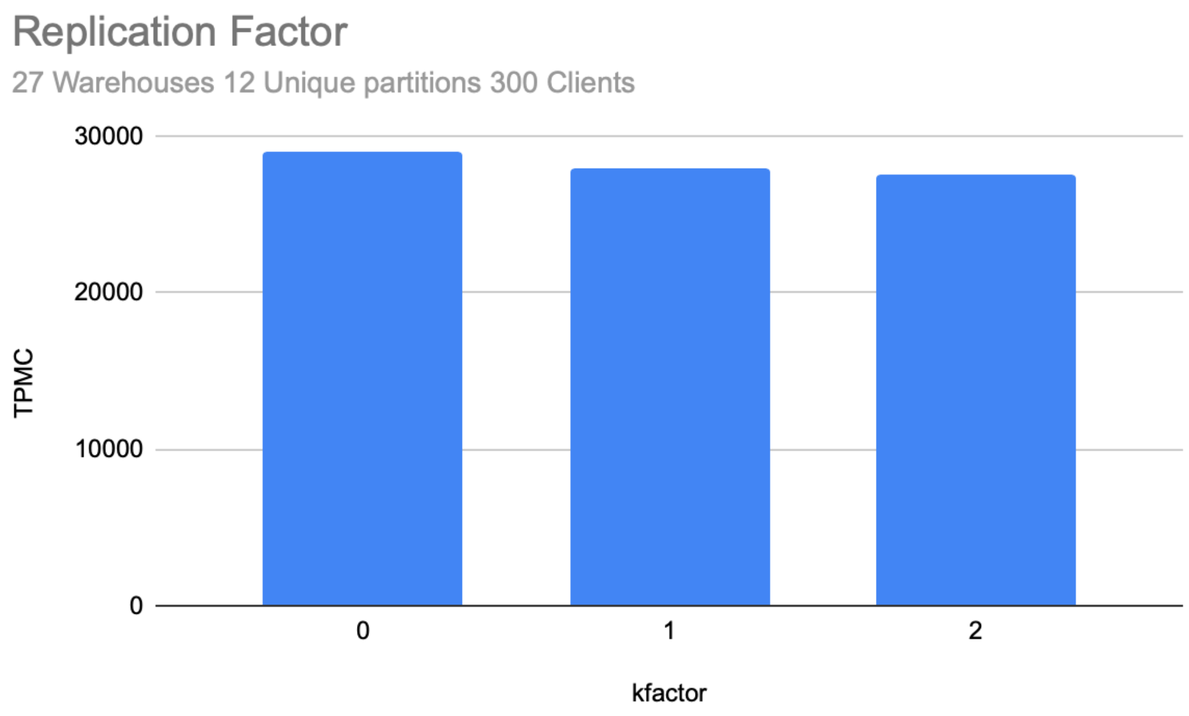


Figure 10: Impact of k factor

The difference between the throughput of k factor 1 and k factor 2 is not statistically significant: k factor 0 has a 4% greater throughput. K factor 0 had a slightly higher throughput because transactions immediately went to the relevant partition rather than going to the single partition initiator. By skipping the overhead from the single partition initiator, the latency of transactions was decreased. The difference between the throughput of k factor 1 and k factor 2 was not significant because the single partition initiator does not need to wait for all of the partitions, in the case of a read-write transaction, to complete before it returns the answer to the client. Once the transaction is in the partitions queue the order will not be lost because the database is single threaded and therefore serializable. Read-only transactions can be split across multiple partition replicas for increased parallelism and theoretically greater throughput because the read-only transactions are only sent to a single partition replica. This means that a single partition can handle multiple read-only transactions at the same time. However, in this workload, read-only transactions make up only 8% of the total transactions which is not enough to show a significant increase in throughput. Overall, increasing the k factor does not affect the throughput provided the number of unique partitions does not change.

5.8 Impact of Clients

By increasing the number of clients calling the database, the arrival rate of transactions entering the database is increased. Increasing the number of transactions being processed by the database increases the strain on it. The queues for entering the partitions increase and network traffic between hosts to direct transactions to the correct partition increases.

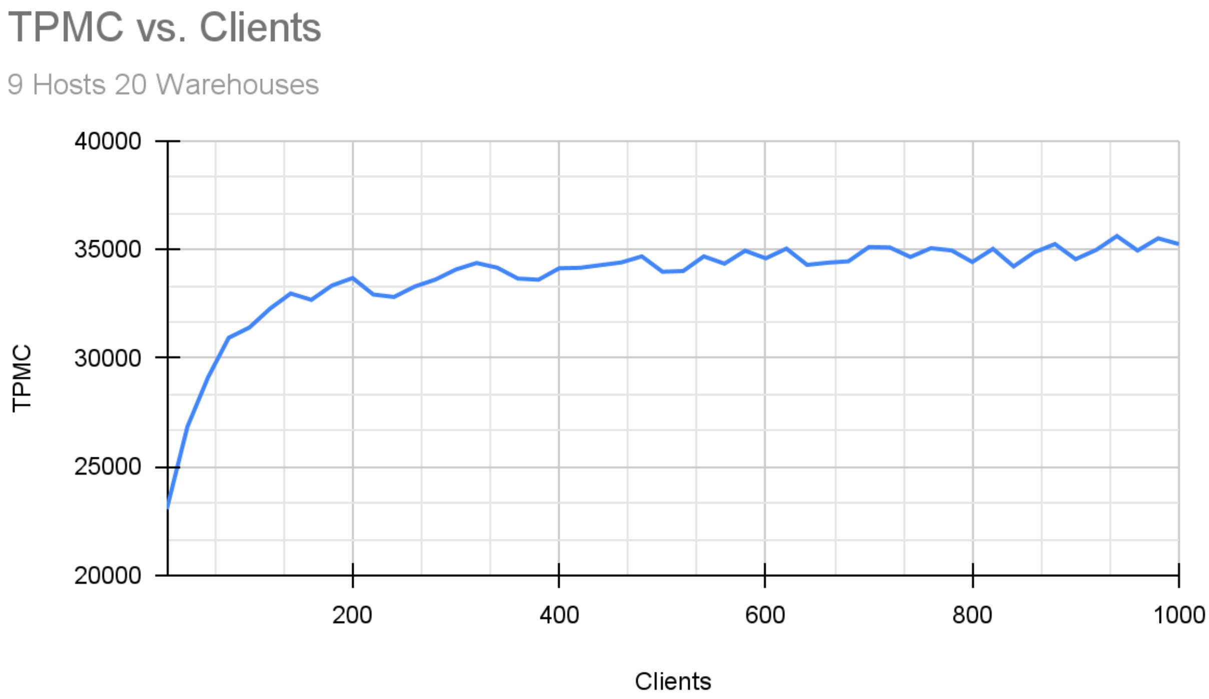


Figure 11: Impact of clients

The throughput increases significantly at first and then slowly increases with some fluctuations, however, it does not completely stop increasing. This experiment was unable to find a point where the throughput was not increasing over a sufficient period of time. From 20 to 300 clients, throughput grew by around 47.87% with most of the growth coming from 20 to 140 clients. From 300 clients to 1000 clients, the throughput only grew by around 5.07%. The increase in throughput is partially because of the parallelism achieved by having exclusively single partition transactions. Each transaction only has to be processed on a single partition, eliminating idle partitions waiting for busy partitions to process a single transaction. By increasing the arrival rate, the number of transactions to be processed by each partition increases, which decreases the chance that any partition is idle waiting for a transaction to process. Increasing the number of clients, i.e. the arrival rate, increases the throughput of the database.

Section 6

Conclusion

VoltDB, a NewSQL type database, represents a new approach to database management systems. It is always consistent and can horizontally scale given a proper partitioning scheme. PyTPCC demonstrates how VoltDB's performance is impacted by a number of situations and features. These experiments give insight into how to get the best performance out of VoltDB. Overall VoltDB performs best when transactions touch a small section of the database and the workload requires a high level of consistency. Additionally, VoltDB is horizontally scalable provided that the cardinality of the partitioning column is equal to or greater than the proposed number of machines in the cluster.

This project was focused on a single type of workload. Future works could explore how VoltDB performs on a variety of workloads to identify new use cases. They could also compare NewSQL to NoSQL and to SQL to highlight and contrast the strengths and weaknesses of the different types of databases under a variety of workloads. This would give insight into the problems and trade-offs employed by the currently available databases.

REFERENCES

[1] A. Pavlo and M. Aslett, "What's Really New with NewSQL?," 2016. [online] Available at: https://db.cs.cmu.edu/papers/2016/pavlo-newsql-sigmodrec2016.pdf

[2] “Using VoltDB” [Online]. Available: https://downloads.voltdb.com/documentation/UsingVoltDB.pdf

[3] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. Abadi, “H-Store: A High-Performance, Distributed Main Memory Transaction Processing System,” 2008. [online] Available at: https://www.vldb.org/pvldb/vol1/1454211.pdf

[4] apavlo. [Online]. Available: https://github.com/apavlo/py-tpcc/wiki.

[5] “TPC BENCHMARK.’’ [Online]. Available: https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf

[6] D. Abadi, A. Batkin, M. Cherniack, X. Chen, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O’Neil, P. O’Neil, A. Rasin, M. Stonebraker, N. Tran, and S. Zdonik, “C-Store: A Column-oriented DBMS,” 2005. [online] Available at: https://dl-acm-org.libaccess.sjlibrary.org/doi/pdf/10.5555/1083592.1083658

[7] D. Abadi, N. Hachem, S. Harizopoulos, P. Helland, M. Stonebraker, and S. Madden, “The End of an Architectural Era (It's Time for a Complete Rewrite)” 2007. [online] Available at: http://nms.csail.mit.edu/~stavros/pubs/hstore.pdf

[8] T. Panpaliya, “Benchmarking MongoDB multi-document transactions in a sharded cluster,” 2020, [online] Available at: https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1910&context=etd_projects

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, ‘‘Benchmarking cloud serving systems with ycsb,’’ Proceedings of the 1st ACM symposium on Cloud computing - SoCC ’10, 2010.

[10] H. Jacobsen, T. Rabl, M.Sadoghi, S. Gómez-Villamor, S. Mankovskii and V. Muntés-Mulero, “Solving Big Data Challenges for Enterprise Application Performance Management” 2012. [online] Available at: https://vldb.org/pvldb/vol5/p1724_tilmannrabl_vldb2012.pdf

[11] A. Kamsky, “Adapting TPC-C Benchmark to Measure Performance of Multi-document Transactions in MongoDB,” 2019. [online] Available at: http://www.vldb.org/pvldb/vol12/p2254-kamsky.pdf

[12] “How VoltDB does Transactions” [Online]. Available: https://voltactivedata.com/wp-content/uploads/2017/03/lv-technical-note-how-voltdb-does-transactions.pdf