Development and Evaluation of a Transactional Benchmark Driver for a Scalable SQL Database

A Thesis

submitted to the designated
by the General Assembly of Special Composition
of the Department of Computer Science and Engineering
Examination Committee

by

Michail Sotiriou

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WITH SPECIALIZATION
IN SOFTWARE

University of Ioannina
July 2019

Examining Committee:

- **Apostolos Zarras**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina (Supervisor)
- **Panos Vassiliadis**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina
- **Kostas Magoutis**, Associate Professor, Department of Computer Science, University of Crete

Acknowledgements

I would like to thank my advisor Associate Professor Apostolos Zarras for sharing his knowledge on the thesis scientific field with me. Through his lectures and guidance, I was given the opportunity to insist on the broader concepts of NoSQL DBs and model-driven applications across them. Moreover, I would like to thank Associate Professor Panos Vassiliadis and Associate Professor Kostas Magoutis for accepting to participate in my examination committee and for their comments on my thesis draft. I would also like to thank the whole community of the Department of Computer Science and Engineering, University of Ioannina for providing all scientific work and infrastructure.

Table of Contents

List of Figures

Abstract

Michail Sotiriou, M.Sc. in Computer Science, Department of Computer Science and Engineering, University of Ioannina, Greece, July 2019.
Development and Evaluation of a Transactional Benchmark Driver for a Scalable SQL Database.
Advisor: Apostolos Zarras, Associate Professor.

The process of benchmarking transactional database systems has a long history and provides important insight into the efficiency and scalability of such systems. Recent advances in scalable transactional databases have motivated the adaptation of traditional transactional benchmarks, such as TPC-C, to a variety of new database architectures. However, a challenge with the proliferation of database architectures today is the need to develop and evaluate new database drivers (ports of the benchmark-database interface to a new database architecture) for a wide range of systems. In this thesis, we develop and evaluate a driver for the Py-TPCC benchmark targeting the CockroachDB scalable SQL database. The driver lowers complexity by leveraging the psycopg2 PostgreSQL adapter for the Python programming language. We evaluate the Py-TPCC benchmark for CockroachDB and compare to an existing TPC-C benchmark implementation for CockroachDB written in Go, using the pq PostgreSQL adapter. Our results indicate that the use of a SQL interface (when offered by the scalable database) simplifies the development of TPC-C benchmark drivers. Despite similarities in the two implementations of the TPC-C benchmark, we observe performance differences that can be attributed to the variable efficiencies of use of CockroachDB resources across implementations of the PostgreSQL adapter.

Εκτεταμενη Περιληψη

Μιχαήλ Σωτηρίου, Μ.Δ.Ε. στην Πληροφορική, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιούλιος 2019.
Ανάπτυξη και Αξιολόγηση ενός Transactional Benchmark Driver για Scalable SQL Database.
Επιβλέπων: Απόστολος Ζάρρας, Αναπληρωτής Καθηγητής.

Η διαδικασία benchmarking δοσοληπτικών συστημάτων βάσεων δεδομένων έχει μακρά ιστορία και παρέχει σημαντική εικόνα για την αποτελεσματικότητα και την επεκτασιμότητα τέτοιων συστημάτων. Οι πρόσφατες εξελίξεις στις κλιμακούμενες δοσοληπτικές βάσεις δεδομένων έχουν παρακινήσει την προσαρμογή των παραδοσιακών benchmarks, όπως το native TPC-C, σε μια ποικιλία νέων αρχιτεκτονικών βάσεων δεδομένων. Ωστόσο, μία πρόκληση με τον πολλαπλασιασμό των αρχιτεκτονικών βάσεων δεδομένων σήμερα, είναι η ανάγκη να αναπτυχθούν και να αξιολογηθούν νέα προγράμματα οδήγησης βάσεων δεδομένων (φορητότητα των benchmark-database διεπαφών σε μια νέα αρχιτεκτονική βάσεων δεδομένων) για ένα ευρύ φάσμα συστημάτων. Σε αυτή τη διατριβή, διατυπώνουμε το επιστημονικό υπόβαθρο που σχετίζεται με δοσοληπτικές βάσεις δεδομένων και τα benchmarks που χρησιμοποιούν. Πιο συγκεκριμένα, αναπτύσσουμε και αξιολογούμε ένα πρόγραμμα οδήγησης για το Py-TPCC benchmark που στοχεύει στην κλιμακώσιμη βάση δεδομένων SQL της CockroachDB. Το πρόγραμα οδήγησης μειώνει την πολυπλοκότητα αξιοποιώντας τον psycopg2 PostgreSQL προσαρμογέα για τη γλώσσα προγραμματισμού Python. Θέτουμε το πλαίσιο πάνω στο οποίο κινηθήκαμε προκειμένου να φτάσουμε στην υλοποίηση του προγράμματος οδήγησης. Καταγράφουμε όλες τις παραμέτρους και απαιτήσεις που έπρεπε να λάβουμε υπόψη, ώστε να ορίσουμε τις διαδικασίες που απαιτούνται. Μέσω της πειραματικής διαδικασίας, αξιολογούμε το PY-TPCC benchmark για την CockroachDB και συγκρίνουμε με την υπάρχουσα υλοποίηση του native TPC-C benchmark για την CockroachDB γραμμένο σε γλώσσα προγραμματισμού Go, χρησιμοποιώντας τον pq PostgreSQL προσαρμογέα. Παρουσιάζουμε τα αποτελέσματα αυτής της σύγκρισης, σημειώνοντας διαφορές τόσο σε ποιοτικά χαρακτηριστικά, όσο και σε επίπεδο αρχιτεκτονικής κώδικα. Τέλος, οδηγούμενοι από τα αποτελέσματα, συμπεραίνουμε ότι τα σημεία στα οποία υπερτερεί το native TPC-C benchmark είναι η καλύτερη απόδοση του συστήματος σε επίπεδο δοσοληψιών, ενώ από την άλλη, το Py-TPCC benchmark υστερεί σε θέματα απόδοσης και κλιμάκωσης, ωστόσο ευνοεί την εύκολη υλοποίηση ενός νέου προγράμματος οδήγησης από τον προγραμματιστή.

Chapter 1

Introduction

Despite the fact that traditional relational database management systems (RDBMSs) provide powerful mechanisms to store and query structured data under strong consistency and transaction guarantees and have reached an unmatched level of reliability, stability and support through decades of development, the rapid and continuous evolution of computing systems and applications, as well as the exponential increase in the amount of useful data with its interdependence and complexity, which is generated and consumed at unprecedented scale, has become so vast that it cannot be stored or processed by traditional database solutions. Therefore, there was the need to manage Big Data, using novel data storage systems that could be able to deal with these requirements and respond to the needs of the time. It is about NoSQL databases, many of which offer horizontal scalability and higher availability than relational databases by sacrificing querying capabilities and consistency guarantees, like BigTable [12], Cassandra [13], MongoDB [14]. These trade-offs are pivotal for service-oriented computing and as-a-service models, since any stateful service can only be as scalable and fault-tolerant as its underlying data store. Nowadays, there are plenty of NoSQL database systems and it is hard to keep track of where they excel, where they fail or even where they differ, as implementation details change quickly and feature sets evolve over time [15].

From the other hand, modern web applications have evolved greatly in relation to previous years. The data they manage is larger and more complex. This results the algorithms required for data management as well as queries on them become more complex and the processing and response time of the system increases dramatically. The strictly relational schema usually constitutes a barrier to network applications, which consist of many different types of attributes. Text, comments, photos, videos, source code and other information should be stored in multiple tables. Since these web applications are very flexible, the underlying databases must be flexible enough to be able to support them. NoSQL DB systems have the ability to store and classify large datasets, while at the same time allow users requests. Cloud services make resources accessible from Internet to users presenting them through virtualization as a service (Infrastructure as a Service, IaaS), supply software platforms on virtualized infrastructure (Platform as a Service, PaaS) and software services can be executed on these distributed platforms (Software as a Service, SaaS). Some systems are offered only as cloud services, either directly in the case of Amazon SimpleDB and Microsoft Azure SQL Services, or as part of a programming environment like Google's AppEngine or Yahoo!'s YQL. Still other systems are used only within a particular company, such as Yahoo!'s PNUTS, Google's BigTable and Amazon's Dynamo [16]. The large variety has made it difficult for developers to choose the appropriate system. The most obvious differences are between the various data models. Comparing the performance of various systems is a harder problem. Some systems have made the decision to optimize for writes by using on-disk structures that can be maintained using sequential I/O, while others have optimized for random reads by using a more traditional buffer-pool architecture. Moreover, decisions about data partitioning and placement, replication, transactional consistency and so on all have an impact on performance.

Understanding the performance implications of these decisions for a given type of application is challenging. Developers of various systems report performance numbers for the ”sweet spot” workloads for their system, which may not match the workload of a target application. Moreover, an apples-to-apples comparison is hard, given numbers for different systems based on different workloads. Thus, developers often have to download and manually evaluate multiple systems. This process is time-consuming and expensive.

1.1 Objectives

In this thesis, we aim to develope and evaluate a transactional benchmark driver for a scalable SQL database and and define the specifications and actions required for the whole process:

- Qualitative and quantitative correlation between the software implementations of native TPC-C and Py-TPCC benchmarks
- Comparative experimental evaluation

1.2 Structure

The rest of this thesis is composed of the following chapters: In Chapter 2, we provide the scientific background on which this thesis is based. Describe precisely the relevant structures and their characteristics and make reference to similar systems, emphasizing what we are interested most. In Chapter 3, we clearly declare the system's prerequisites and analyze in detail all those steps that have been taken to achieve the implementation. We quote all the procedures followed step by step. In Chapter 4, we present particularly the experimental process with the relevant charts through which we validate the result of our research. Finally, in Chapter 5, we conclude and leave open for study further issues and future work.

Chapter 2

Background

In this chapter, we make an extensive reference at NoSQL Database systems, their categories and their characteristics in Section 2.1. We quote in detail the different types of NoSQL DBs, considering the requirements of performance, scaling and availability. After that, in Section 2.2, we introduce the terms of transactions and benchmarking serving systems that are connected with NoSQL DBs, as well as their evolution. Finally, in Section 2.3, we introduce CockroachDB, a key-value datastore, on which is based this thesis, referring to its features, characteristics, capabilities and analyzing its architecture .

2.1 NoSQL Database Systems

NoSQL databases provide the performance, scale, and flexibility required of modern applications. NoSQL are distributed systems, which are scalable, durable, adaptable, easily manageable, efficient, with high availability and allow parallel processing. The biggest worldwide IT companies such as Google, Amazon, Facebook, Twitter, Linkedin, etc. have turned their attention to NoSQL systems technology.

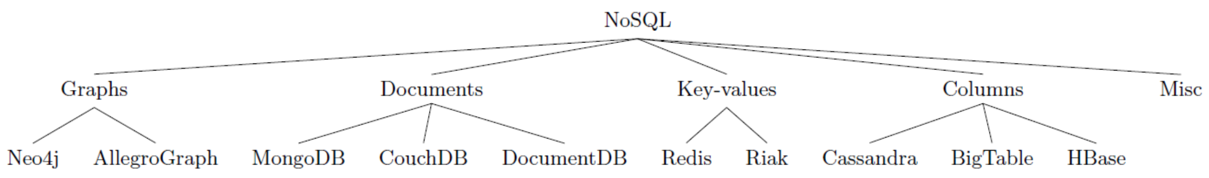


Figure 2.1: NoSQL databases categorised according to the supported data model

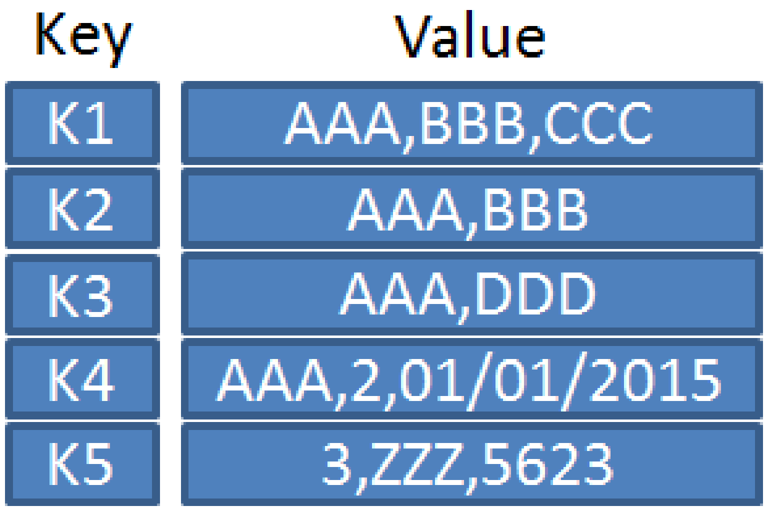
2.1.1 NoSQL DB Categories

NoSQL systems are distributed, non-relational bases, designed for large scale data storage and parallel processing data shared on a large number of servers. NoSQL databases focus on analyzing large data set (bigdata), offering increased scalability and high performance. They are very flexible in improving their performance, which is is due to their implementation, which is often done in cloud infrastructures or virtualized environments, in contrast with traditional RDBMS systems. The most commonly employed distinction between NoSQL databases is the way they store and allow access to data, as we can see in Figure 2.1. The only thing most NoSQL databases have in common is that they do not follow a relational data model. NoSQL databases typically fall into one of four categories:

- Key-value stores: A key-value store consists of a set of key-value pairs with unique keys, as shown in Figure 2.2. Due to this simple structure, it only supports get and put operations. As the nature of the stored value is transparent to the database, pure key-value stores do not support operations beyond simple CRUD (Create, Read, Update, Delete) [15]. Key-value stores are therefore often referred to as schemaless: any assumptions about the structure of stored data are implicitly encoded in the application logic and not explicitly defined through a data definition language. The obvious advantages of this data model lie in its simplicity. The very simple abstraction makes it easy to partition and query the data, so that the database system can achieve low latency as well as high throughput. However, if an application demands more complex operations, e.g. range queries, this data model is not powerful enough. Since queries more complex than simple lookups are not supported, data has to be analyzed inefficiently in application code to extract information.

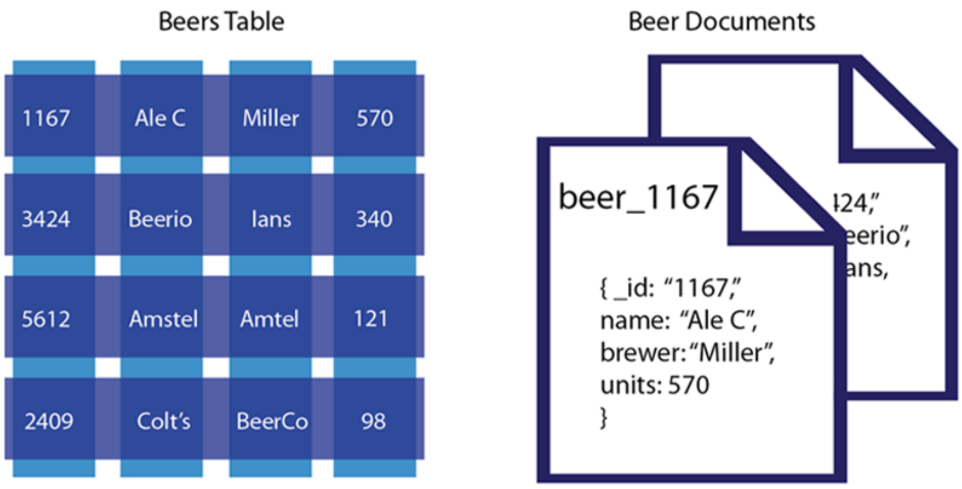
- Document stores: Follow the logic of key-value stores. The data in this case (collections from key-value pairs) is organized, more naturally and logically, without any restrictions of any schema. A document store is a key-value store that restricts values to semi-structured formats such as JSON documents (Figure 2.3). This restriction in comparison to key-value stores brings great flexibility in accessing the data. It is not only possible to fetch an entire document by its ID, but also to retrieve only parts of a document, e.g. the age of a customer, and to execute queries like aggregation, query-by-example or even full-text search.

- Wide-column stores: Wide-column stores inherit their name from the image that is often used to explain the underlying data model: a relational table with



Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Figure 2.2: Key-value store [1]

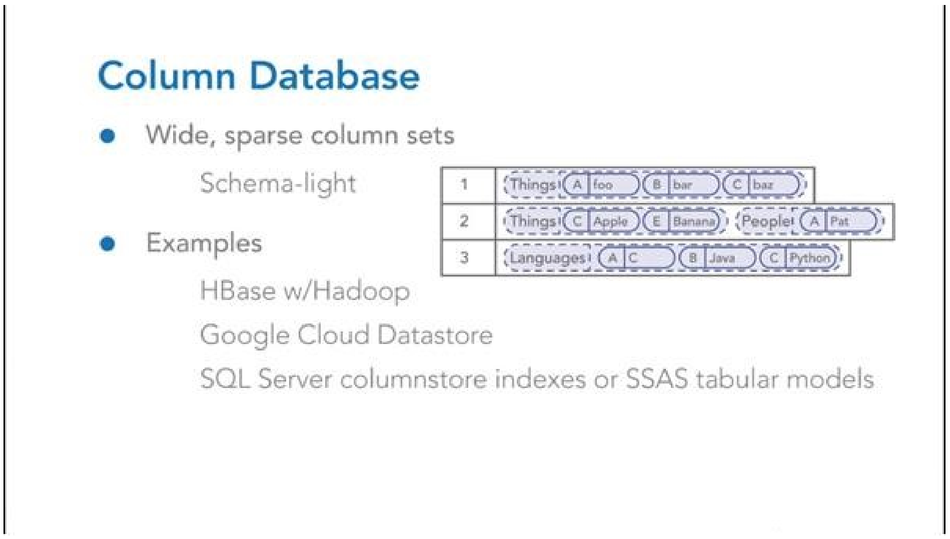


1167	Ale C	Miller	570
3424	Beerio	Ians	340
5612	Amstel	Amtel	121
2409	Colt's	BeerCo	98

```
{ _id: "1167,"
name: "Ale C",
brewer:"Miller",
units: 570
}
```

Figure 2.3: Document store [2]

many sparse columns. Technically, however, a wide-column store is closer to a distributed multi-level sorted map: the first-level keys identify rows which themselves consist of key-value pairs. The first-level keys are called row keys, the second-level keys are called column keys (Figure 2.4). This storage scheme makes tables with arbitrarily many columns feasible, because there is no column key without a corresponding value. Hence, null values can be stored without any space overhead. The set of all columns is partitioned into so-called column families to co-locate columns on disk that are usually accessed together. On disk, wide-column stores do not co-locate all data from each row, but instead values of the same column family and from the same row. Hence, an entity (a row) cannot be retrieved by one single lookup as in a document store, but has to be joined together from the columns of all column families. However, this storage layout usually enables highly efficient data compression and makes retrieving only a portion of an entity very efficient. The data are stored in lexicographic order of their keys, so that data that are accessed together are physically co-located, given a careful key design. As all rows are distributed into contiguous ranges (so-called tablets) among different tablet servers, row scans only involve few servers and thus are very efficient.



Column Database

- Wide, sparse column sets
  - Schema-light
- Examples
  - HBase w/Hadoop
  - Google Cloud Datastore
  - SQL Server columnstore indexes or SSAS tabular models

1
Things A foo B bar C baz
2
Things C Apple E Banana People A Pat
3
Languages A C B Java C Python

Figure 2.4: Wide-Column store [3]

- Graph: Graph database are designed for data whose relations are well represented as a graph consisting of elements interconnected with a finite number of relations between them. They use graph structures and is based on nodes, relationships between these nodes and their properties. Instead of columns and rows, here is a flexible graphical model (graphmodel) that can be used and deployed in parallel to many machines (servers - nodes), as shown in Figure 2.5. The type of data could be social relations, public transport links, road maps, network topologies, etc.

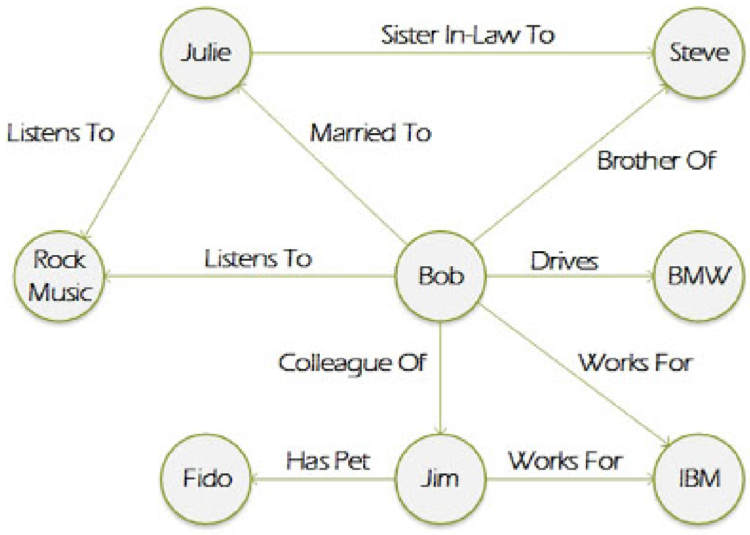


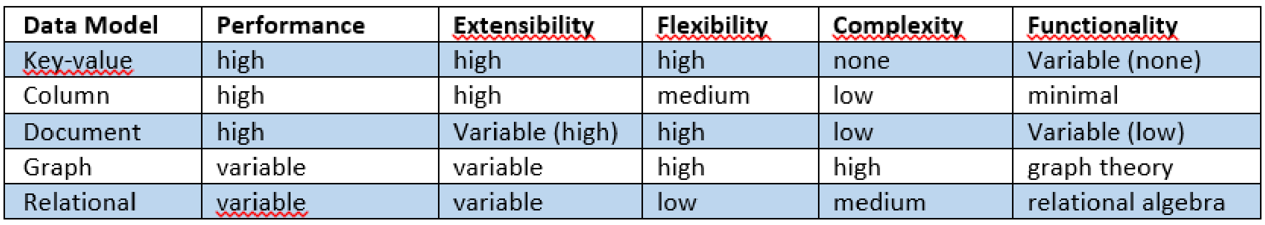
Figure 2.5: Graph store [4]

In Figure 2.6, we can see a distinction of some NoSQL DBs according to their qualitative characteristics.

2.1.2 ACID Transactions

A transaction symbolizes a unit of work performed within a database management system against a database and treated in a coherent and reliable way independent of other transactions. A transaction generally represents any change in a database. A transaction is a single unit of logic or work, sometimes made up of multiple operations. Any logical calculation done in a consistent mode in a database is known as a transaction. In order to ensure the integrity of the data, NoSQL database systems are based on transactions. This ensures cohesion of data during management [17]. The total of properties that guarantee that transactions on the database work reliably are known by acronym ACID (Atomicity, Consistency, Isolation, Durability), Figure 2.7:

- Atomicity: requires the modification to be made on the base to keeps the rule all or nothing. Every transaction that is individual is so called because if a part



Data Model	Performance	Extensibility	Flexibility	Complexity	Functionality
Key-value	high	high	high	none	Variable (none)
Column	high	high	medium	low	minimal
Document	high	Variable (high)	high	low	Variable (low)
Graph	variable	variable	high	high	graph theory
Relational	variable	variable	low	medium	relational algebra

Figure 2.6: NoSQL Data Models

of it fails, the whole transaction fails and the base stays the same was before the transaction was executed; you can never see "half a change"

- Consistency: the change can only happen if the new state of the system will be valid; any attempt to commit an invalid change will fail, leaving the system in its previous valid state
- Isolation: refers to the requirement that all actions do not can access or view data that is modified this moment from a transaction that has not yet been completed. Every transaction should not know if there are other transactions that are executed simultaneously, but to await the completion of a transaction to see / modify the data, which the other transaction needs; no-one else sees any part of the transaction until it's committed
- Durability: guarantees the user that if a transaction is successful then its results will not be lost. Changes made by transaction will not be lost even if it crashes the system and all the integrity requirements apply so that the system will not have to cancel this transaction. Once the change has happened - if the system says the transaction has been committed, the client doesn't need to worry about "flushing" the system to make the change "stick"

2.1.3 CAP Theorem

However, distortion from systems having ACID properties has been shown to cause various kinds of problems. Conflicts arise between the various aspects of high availability in distributed systems, which are not fully resolvable. Some databases are built to guarantee strong consistency and serializability, while others favour availability. This trade-off is inherent to every distributed database system. This situation describes the CAP theorem (Figure 2.8). In 2000, Eric A. Brewer [18] argued that each

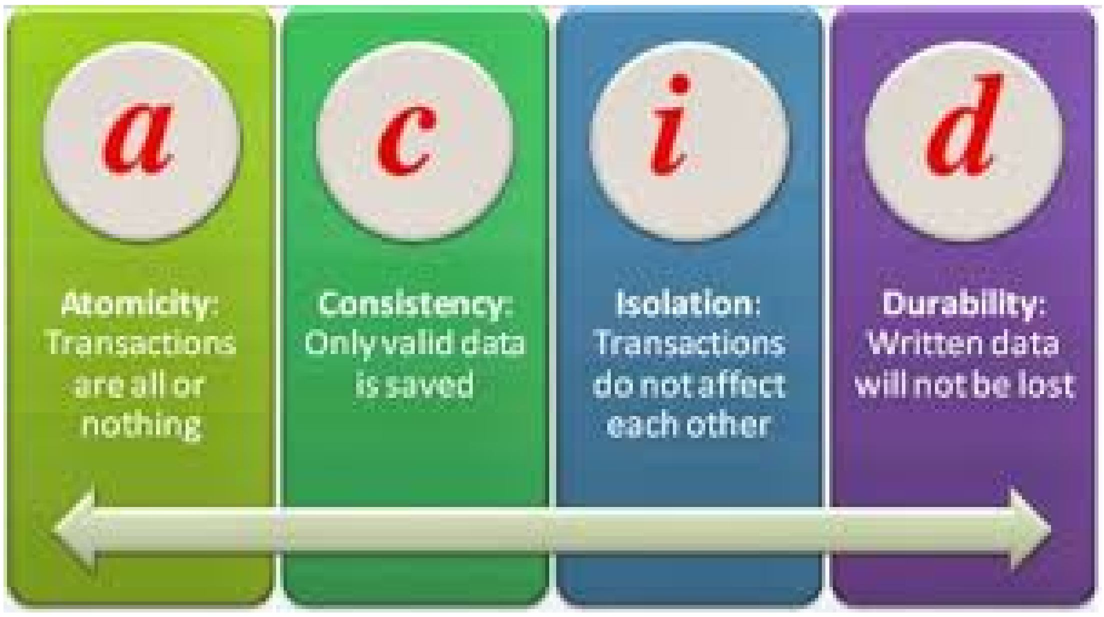

a
c
i
d
Atomicity: Transactions are all or nothing
Consistency: Only valid data is saved
Isolation: Transactions do not affect each other
Durability: Written data will not be lost

Figure 2.7: ACID Transactions [5]

partitioned system can guarantee at most two of the following three basic properties at the same time:

- Consistency (C): Reads and writes are always executed atomically and are strictly consistent. Put differently, all clients have the same view on the data at all times.
- Availability (A): Every non-failing node in the system can always accept read and write requests by clients and will eventually return with a meaningful response, i.e. not with an error message.
- Partition-tolerance (P): The system upholds the previously displayed consistency guarantees and availability in the presence of message loss between the nodes or partial system failure.

2.2 Benchmarks

The continuous proliferation of Big Data systems, and in particular NoSQL databases, coupled with the lack of apples-to-apples performance comparisons, makes it difficult to understand the tradeoffs between the systems and the workloads for which they are appropriate. It has become increasingly difficult to compare and choose the optimal combination of NoSQL storage technology and benchmarking system. Thus,

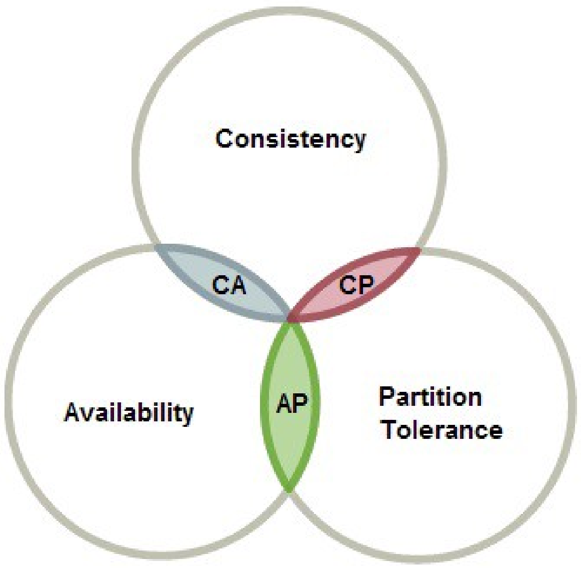


Figure 2.8: CAP Theorem [6]

there are benchmarks that need to be expanded, depending on the various data models, such as the column-group oriented BigTable model used in Cassandra and HBase versus the strongly-consistent key-value store CockroachDB, the performance of various systems, such as systems optimized for writes or reads, and the decisions about data partitioning and placement, replication, transactional consistency, and so on.

2.2.1 TPC-C Benchmark

TPC-C was developed and published by the Transaction Processing Performance Council (TPC). TPC-C benchmark is a descendant of TPC-A, measures the performance of Online Transaction Processing Systems (OLTP) and is based on a complex database and a number of different transaction types either executed on-line or queued for deferred execution. TPC-C is not only a hardware-independent but also a software-independent benchmark and defines a set of functional requirements that can be run on any transaction processing system, either ”proprietary” or ”open” systems. This is a property that allows the user not to be limited to specific machines that run on just one operating system or benchmarks that execute the same set of software instructions. An additional differentiation of the TPC-C with the rest of the benchmarks is that its modeling is done under actual production applications and environments rather than stand-alone computer tests in order not to evaluate performance factors like user interface, communications, disk I/Os, data storage, backup

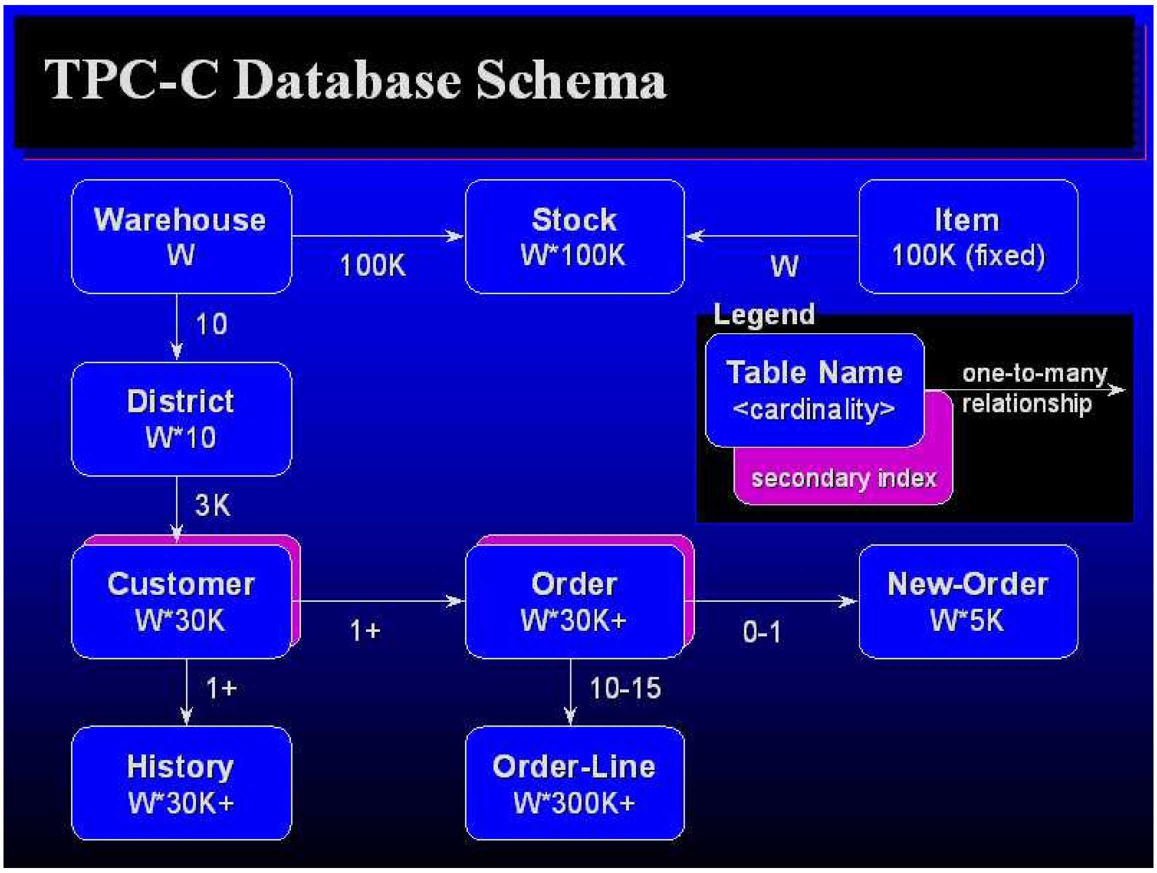


Figure 2.9: Native TPC-C Database Schema [7]

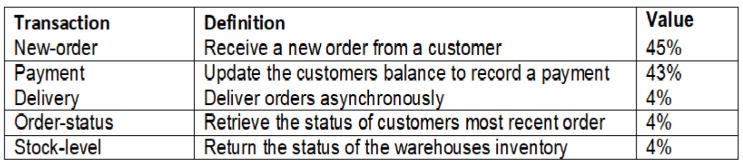
and recovery [19]. From now on and in the context of this thesis, we will refer to TPC-C benchmark as native TPC-C.

Native TPC-C simulates an environment in which the operator performs various transactions against a database. The benchmark describes the typical transactions of a wholesale company concerning order entries, including entering and delivering orders, recording payments, checking the status of orders and monitoring the level of stock at the warehouses. The simulated company operates out of a number of warehouses and their allocated districts. Native TPC-C is designed in such a way that the size of the company (i.e. the number of its warehouses) may vary and is able to scale both the number of terminals and the size of the database, as the company expands and new warehouses are created. As we can see in Figure 2.9, set parameters are the 100,000 items as well as ten sales districts per warehouse and 3,000 customers per district. Every operator can at any time implement one of five transactions on the company's goods ordering system. Both the transactions and their frequency are based on realistic scenarios.

The most frequent transaction is the new order, which on average comprises 10 different items. Each warehouse attemptsto maintain stock for the 100,000 items in the Company's catalog and fill orders from that stock. However, one warehouse will probably not have all the parts required to fill every order. Therefore, Native TPC-C requires that close to 10% of all cases is effected via the company's other warehouses. Another frequent transaction is the recording of a payment, received from a customer. Order status queries, the processing of delivery orders and checking of local stock levels for possible bottlenecks are less frequent. The entire business activity is modeled on the basis of these five transactions. The performance metric reported by native TPC-C measures the number of orders that can be fully processed per minute and is expressed in tpm-C.

The database consists of nine variously structured tables and so also nine types of data records. The size and number of the data records vary depending on the table. A mix of five concurrent transactions of varying type and complexity is executed on the database - largely online or queued for deferred batch processing. Due to their competing for the limited system resources many system components are stressed and data changes are executed in a variety of ways. The way in which data are entered by operators in native TPC-C is based on the most basic characteristics of reallife data-input situations. For example, it is possible for invalid item numbers to be entered, which then results in the cancellation of the transaction. In order to model as realistic a scenario as possible, the artificial simplifications used in many other benchmarks were largely omitted. Thus, for example, it must always be possible for all terminal input to be entered by real-life operators. To this end, all entry screens must include specified field definitions as well as labeled input and output fields and also have the common cursor motion and field correction mechanisms.

The throughput of native TPC-C is a result of the whole of activities at the terminals. Each warehouse has 10 terminals and all 5 transactions can be executed at each terminal. A remote terminal emulator (RTE) is used to simulate the terminal activities and to maintain the required combination of transactions while the performance is measured. The transaction combination represents the complete business processing of an order from its entry through its delivery. More specifically, the required combination is defined to produce an equal number of new order and payment transactions and one delivery transaction, one order-status transaction and one stock-level transaction for every 10 new-order transactions.



Transaction	Definition	Value
New-order	Receive a new order from a customer	45%
Payment	Update the customers balance to record a payment	43%
Delivery	Deliver orders asynchronously	4%
Order-status	Retrieve the status of customers most recent order	4%
Stock-level	Return the status of the warehouses inventory	4%

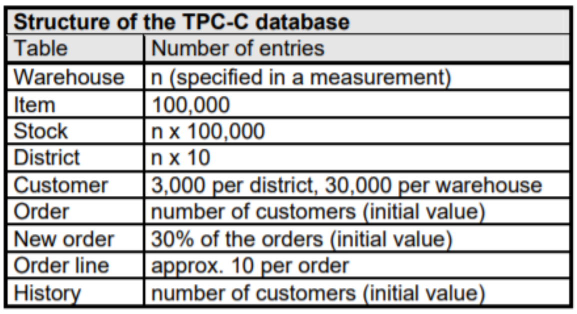
Figure 2.10: Native TPC-C Workload

The RTE is also used to measure transaction response time and to simulate keying and think times of the operator. Keying time is the time required to enter the data at the terminal. Think time is defined as the time an operator needs to read and evaluate the results of a transaction, before requesting another transaction. Each transaction has a minimum keying time and a minimum think time. In addition, the response times of each transaction must be below a defined threshold. For 90% of transactions this threshold is less than 5 seconds and less than 20 seconds for the stock-level transaction.

All possible performance-relevant database design techniques, e.g. partitioning or replication, are permitted in native TPC-C. Large performance advantages are not to be expected because of the way in which the use of data records through the transactions is defined. The performance, which measures the throughput of the system, is an approximation of the true cost of the system to the end-user and includes the cost of all hardware and software components, maintenance costs over 5 years and the storage capacity to hold the data generated over a period of 180 eight-hour days of operation at the reported throughput [20].

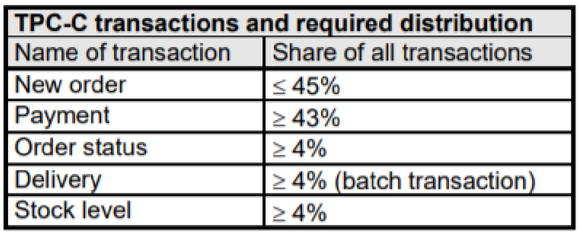
2.2.2 YCSB Benchmark

The Yahoo! Cloud Serving Benchmark (YCSB) is an open-source framework for evaluating and comparing maintenance capabilities of multiple types of data-serving systems. YCSB focus in testing the strength of key value operations. It is often used to compare relative performance of NoSQL database management systems, such as Apache HBase, Apache Cassandra, Redis, MongoDB and Voldemort. The original benchmark was developed by workers in the research division of Yahoo! who re-



Structure of the TPC-C database	
Table	Number of entries
Warehouse	n (specified in a measurement)
Item	100,000
Stock	n x 100,000
District	n x 10
Customer	3,000 per district, 30,000 per warehouse
Order	number of customers (initial value)
New order	30% of the orders (initial value)
Order line	approx. 10 per order
History	number of customers (initial value)

(a) Structure of the native TPC-C database



TPC-C transactions and required distribution	
Name of transaction	Share of all transactions
New order	≤ 45%
Payment	≥ 43%
Order status	≥ 4%
Delivery	≥ 4% (batch transaction)
Stock level	≥ 4%

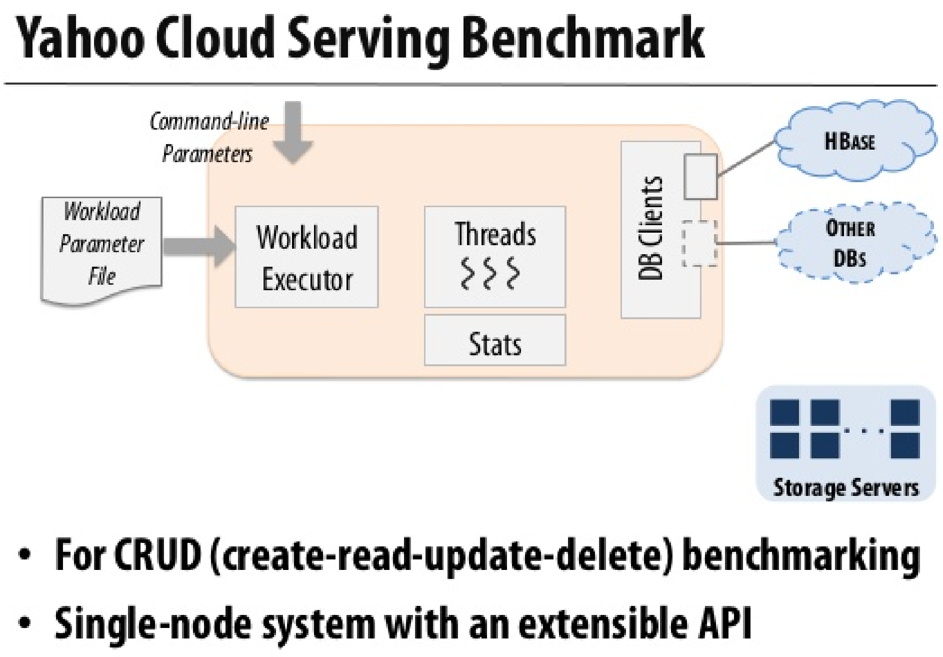
(b) Native TPC-C transactions and required distribution

leased it in 2010 [21] in order to facilitate performance comparisons for transaction-processing workloads of cloud data serving systems.

Firstly, YCSB was contrasted with the TPC-H benchmark from the Transaction Processing Performance Council, with YCSB being called a big data benchmark while TPC-H [22] is a decision support system benchmark. The goal of the YCSB project was to create a standard benchmark and benchmarking framework to assist in the evaluation of different cloud systems, focused on serving systems, which are systems that provide online read/write access to data. Usually a web user is waiting for a web page to load and reads and writes are carried out to the database as part of the page construction and delivery [23]. It consists of two parts:

- The YCSB Client: is a Java program for generating the data to be loaded to the database and generating the operations which make up the workload. The basic operation is that the workload executor drives multiple client threads. Each thread executes a sequential series of operations by making calls to the database interface layer, both to load the database (the load phase) and to execute the workload (the transaction phase). The threads throttle the rate at which they generate requests, so that we may directly control the offered load against the database. The threads also measure the latency and achieved throughput of their operations and report these measurements to the statistics module. Therefore, the most important aspect of the YCSB framework is its extensibility: the workload generator makes it easy to define new workload types and it is also straightforward to adapt the client to benchmark new data serving systems.

- The core workloads, a set of workload scenarios to be executed by the generator. The core workloads provide a picture of a system's performance and the client is



Yahoo Cloud Serving Benchmark

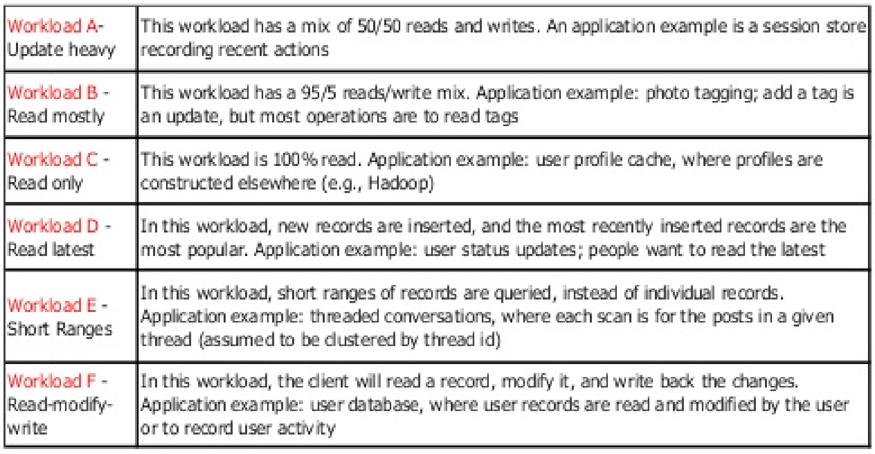
- For CRUD (create-read-update-delete) benchmarking
- Single-node system with an extensible API

Figure 2.12: YCSB Architecture [8]

extensible so that can be defined additional workloads to examine system aspects or application scenarios not covered by the core workload. The client can also be extended to benchmark different databases by an interface layer(Figure 2.12).

YCSB has 6 core workloads, as shown in Figure 2.13. There are 6 steps to running a workload:

- Set up the database system to test
- Create the table and load data into it, either using YCSB or manually
- Choose the appropriate workload
- Choose the appropriate runtime parameters (number of client threads, target throughput, etc.)
- Load the data
- Execute the workload



Workload A- Update heavy	This workload has a mix of 50/50 reads and writes. An application example is a session store recording recent actions
Workload B - Read mostly	This workload has a 95/5 reads/write mix. Application example: photo tagging; add a tag is an update, but most operations are to read tags
Workload C - Read only	This workload is 100% read. Application example: user profile cache, where profiles are constructed elsewhere (e.g., Hadoop)
Workload D - Read latest	In this workload, new records are inserted, and the most recently inserted records are the most popular. Application example: user status updates; people want to read the latest
Workload E - Short Ranges	In this workload, short ranges of records are queried, instead of individual records. Application example: threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id)
Workload F - Read-modify-write	In this workload, the client will read a record, modify it, and write back the changes. Application example: user database, where user records are read and modified by the user or to record user activity

Figure 2.13: YCSB Workloads

2.2.3 PyTPCC Benchmark

This project is a Python-based framework of the native TPC-C OLTP benchmark for NoSQL systems [24]. This code was originally written by Brown University students for a graduate seminar course on NoSQL Systems. The framework is designed to be modular to allow for new drivers to be written for different systems.

- Command-line Usage
- Implementing a Driver

The framework requires either Python 2.7 or Python 2.6 with the argparse package installed. Each driver may also require additional packages.

From now on, and in the context of this thesis, we will refer to the Python version of native TPC-C benchmark as Py-TPCC.

2.3 Cockroach DB

Cockroach Labs is a computer software company that develops commercial database management systems. It is best known for CockroachDB, which has been compared to Google Spanner, and more precisely has been built using a Google whitepaper on Spanner as an open-source database. Cockroach Labs was founded in 2015 by ex-Google employees Spencer Kimball, Peter Mattis, and Ben Darnell. CockroachDB is a project that is designed to store copies of data in multiple locations in order to deliver speed access. It is described as a scalable, consistently-replicated, transactional datastore. The database is scalable in that a single instance can scale from a single laptop to thousands of servers. CockroachDB is designed to run in the cloud and be resilient to failures. The result is a database that is described as ”almost impossible” to take down. Even if multiple servers or an entire datacenter were to go offline, CockroachDB would keep services online, ensuring applications are always available and correct, obtaining database automatically scales, rebalances and repairs itself, simplifing operations with orchestration tools like Kubernetes and Mesosphere DC/OS. With CockroachDB's distributed SQL engine and ACID transactions is possible the creation of the most IT flexibility work with a database that allows building across on-premise, cloud, hybrid cloud and multi cloud environments without sacrificing performance, SQL or scale and without ever manually sharding [25].

CockroachDB have been benchmarked on different parameters against PostgreSQL [26]. It is a distributed SQL database build on transactional and strongly-consistent key-value store. It can scale horizontally, survive hardware (disk, rack, machine and data center) failures without manual interference with interruption of minimum latency. It supports strongly consistent ACID transactions and provides a user-friendly SQL API to perform any function on the data. CockroachDB follows a standard of availability in which groups of symmetric nodes are used intelligently to agree on write requests. Once consensus is done, writes are available for read operation from any node in the cluster. Read / Write traffic is sent to any node because of their symmetric behaviour serving as client gateways. Load get balanced dynamically towards healthy nodes, no need to use any prone-to-error failover setups. That's how consistent replication with greater data redundancy and availability is achieved across the nodes. CockroachDB minimizes the operational overhead with self-organizing nodes that support built-in scaling, failover, replication and repair. Enterprise-grade infrastructure is maintained with zero downtime rolling upgrades. Cluster's health and performance can be managed and observed through the admin UI. Moreover, it can run on any cloud or on-premise. CockroachDB is only suitable for global, scalable cloud services. It's not suitable for low-latency use cases, unstructured data and Non-SQL analytics.

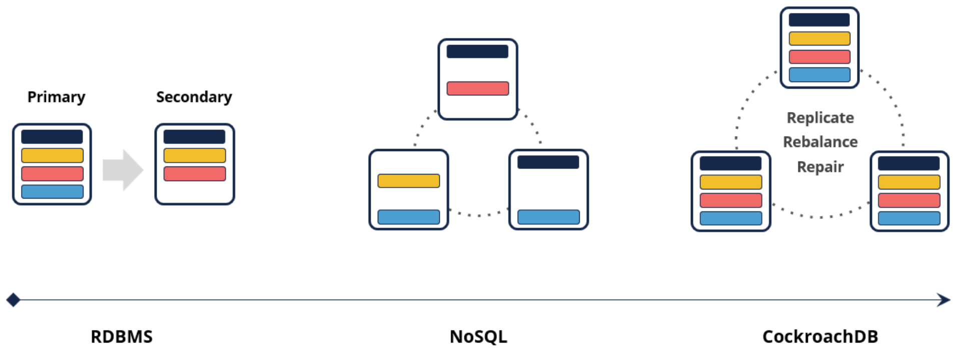
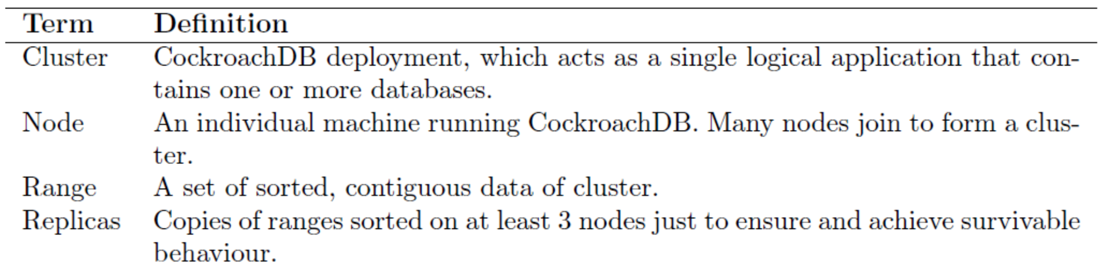


Figure 2.14: Difference between RDBMS, NoSQL and CockroachDB [9]

2.3.1 From SQL to NewSQL

NewSQL is a class of modern relational database management systems that seek to provide the same scalable performance of NoSQL systems for online transaction processing (OLTP) read-write workloads while still maintaining the ACID guarantees of a traditional database system. Many enterprise systems that handle high-profile data (e.g., financial and order processing systems) are too large for conventional relational databases. NewSQL combines availability and requirements of specific data models, which NoSQL databases offer, as well as transparent ACID transactions with primary, secondary indexes and industrial standards, that traditional SQL systems support [27]. NewSQL has the following characteristics:

- provides full ACID transactional support
- minimizes application complexity.
- increases consistency
- doesn't require new tool or language to learn
- provides clustering like NoSQL but in a traditional environment.



Term	Definition
Cluster	CockroachDB deployment, which acts as a single logical application that contains one or more databases.
Node	An individual machine running CockroachDB. Many nodes join to form a cluster.
Range	A set of sorted, contiguous data of cluster.
Replicas	Copies of ranges sorted on at least 3 nodes just to ensure and achieve survivable behaviour.

Figure 2.15: CockroachDB Glossary [10]

2.3.2 CockroachDB Architecture

CockroachDB has a layered architecture, designed to create an open-source database. Below, we provide an analysis of these layers CockroachDB can start running on machines with just two commands:

- cockroach start command with a -join flag for all the initial nodes in the cluster (to let the process know about other nodes it can communicate)
- cockroach init command to initialize the cluster for one time.

While cockroach process starts running, we can interact with CockroachDB through SQL API, which is modelled on PostgreSQL. All nodes of the cluster follow symmetrical behaviour, so that we can send request to any of them. This makes CockroachDB able to integrate with load balancers easily. CockroachDB converts SQL RPCs into operations that work with distributed key-value store. It starts data distribution across nodes by dividing the data into 64MiB chunks (ranges). Each range get replicated synchronously to at least 3 nodes (ensures survivability of DB). This way of handling the read / write request enables CockroachDB to make it highly available [28].

A node cannot serve any request directly. It finds the node that can handle it and communicates with it. Therefore, user doesn't need to know about locality of data, because CockroachDB is smart enough to track it for the user and enable symmetric behaviour for each node. CockroachDB ensures isolation to provide consistent reads, regardless of which node user is communicating with, through consensus algorithm. Finally, the data is written to or read from the disk using an efficient storage engine which keeps track of data's timestamp. User can find historical data for a specific period of time using SQL standard ”AS OF SYSTEM TIME” term. This high-level overview explains what CockroachDB does.

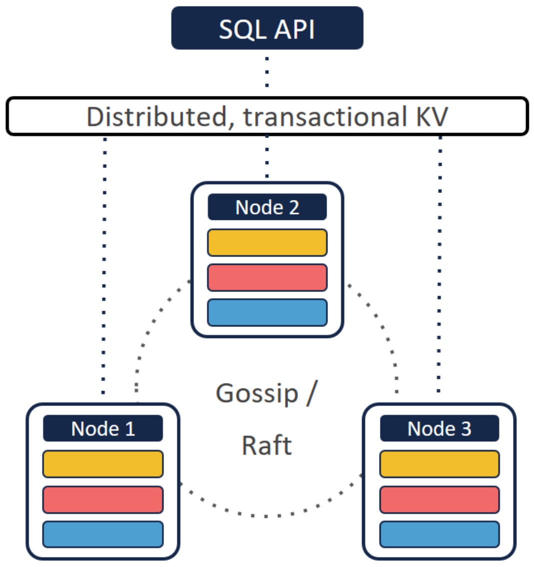


Figure 2.16: Architectural Overview [9]

At the highest level, CockroachDB converts clients' SQL statements into key-value data, which are distributed among nodes and written to disk by the architectural process, which is deployed in different layers communicating with each other as a service. The explanation of functions which are performed by each layer in the given order

- SQL Layer: Displays its SQL API to developers. After deployment, CockroachDB requires only a connection string to the cluster in order to start querying the data using SQL statements. As all nodes of the CockroachDB are symmetric, developers can send request to any node, ignoring locality. It's CockroachDB's job to handle load balancers, which is being done in very efficient manner. "Whichever node receives the request acts as the "gateway node", as other layers process the request". Every request arrives to the cluster as a SQL statement, but data is eventually written to and read from the storage layer as key-value pairs. To manipulate this, the SQL layer converts SQL statements into a plan of key-value operations, which it passes to the Transaction Layer.

- Transaction Layer: Implements support for ACID transactions by managing concurrent operations. CockroachDB encounters consistency of the database as the most important feature, avoiding potentially subtle and hard time to detect anomalies. However, all statements are handled as transactions, including single statements. The Transaction Layer receives key-value operations from planNodes of the SQL Layer. The TxnCoordSender sends its key-value requests to DistSender in the Distribution Layer.

- Distribution Layer: Provides a unified view of cluster's data. CockroachDB stores cluster's data in a monolithic sorted map of key-value pairs to make all data in a cluster to be accessible from any node. This keyspace is divided into 'ranges' i.e. contiguous chunks of the keyspace to make every key accessible from a single range. Moreover, it describes data and its location in the cluster. This sorted map of key-value pairs enables:

  - Simple lookups: Qqueries are able to quickly locate where to find the required data, because of the identification of nodes, which are responsible for certain portions of the data.
  - Efficient Scans: It's getting easy to find data within a particular range during a scan, because of the definition of the order of data.

  The Distribution Layer's DistSender receives BatchRequests from its own node's TxnCoordSender, keeped in the Transaction Layer. The Distribution Layer routes BatchRequests to nodes containing ranges of data, which is eventually routed to the Raft group leader or Leaseholder, which are handled in the Replication Layer.

- Replication Layer: Provides the replication of the data. It takes care of the data copying between nodes and ensures consistency between these copies by implementing consensus algorithm. As CockroachDB is highly available and distributed across nodes, it provides consistent service to the application even if some node goes offline. CockroachDB achieves this by replicating data between nodes to ensure that it remains accessible from any node. Ensuring such consistency when nodes go offline is a trivial task and many databases have failed in achieving this. CockroachDB solves this problem by using consensus algorithm (RAFT) to require that a quorum of replicas agrees on any changes to a range before COMMIT.
  At this point, we make a short reference to the RAFT protocol: A typical consensus algorithm accepts operations from a client, and puts them in an ordered log (which in turn is kept on each of the replicas), acknowledg-

ing an operation as successful to the client once it is known that the operation has been persisted in a majority of the replicas' logs. Each of the replicas in turn execute operations from that log in order, advancing their state. This means that at a fixed point in time, the replicas may not be identical, but they are advancing through the same log (meaning that if you give them time to all catch up, they will be in the same state).

CockroachDB high availability, which is also known as Multi-Active-Availability, requires 3 nodes in a cluster because 3 is the smallest number, which can achieve quorum (i.e. 2 out of 3). The number of failures that can be tolerated are calculated by following formula:

$$S_n = (Replicationfactor - 1)/2. \tag{2.1}$$

For example, if the cluster contains three nodes, it can afford failure of one node and if it contains 5 nodes, it can afford failures of two nodes and so on. CockroachDB automatically realizes nodes have stopped responding because of failure and works to redistribute data to maintain high availability along with survivable behaviour. also, this process works the other way around: when new nodes join the cluster, data automatically rebalances onto it, ensuring loading is evenly distributed. CockroachDB automatically realizes / detect when failure happens and starts redistributing data to continue maximizing survivability. Similarly, when a new node joins the cluster, data get automatically rebalanced to ensure even distribution of data load. In relationship to other layers in CockroachDB, the Replication Layer receives requests from and sends responses to the Distribution Layer and writes accepted requests to the Storage Layer.

- Storage Layer: Each CockroachDB node contains at least one store, designated when the node starts, which is where the cockroach process reads and writes its data on disk. Data is stored as key-value pairs on disk using RocksDB and it is treated primarily as a black-box API. Internally, each store contains three instances of RocksDB:
  - One for the Raft log
  - One for storing temporary Distributed SQL data
  - One for all other data on the node

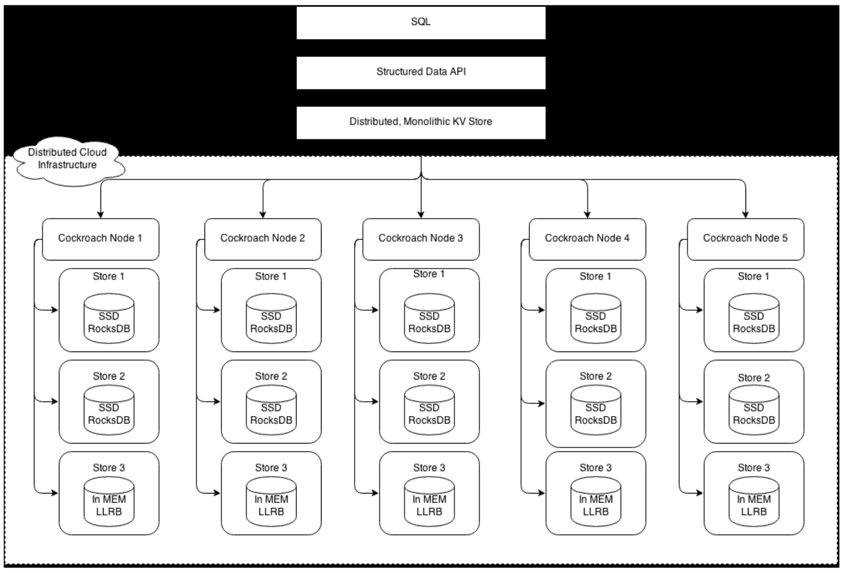


Figure 2.17: Layered Architecture Diagram [11]

In addition, there is also a block cache shared amongst all stores in a node. These stores have a collection of range replicas in turn. More than one replica for a range will never be placed on the same store or on the same node. In relationship to other layers in CockroachDB, the Storage Layer serves successful reads and writes from the Replication Layer.

CockroachDB guarantees consistency among replicas using replication layer of its architecture. It is achieved by offering serializability of SQL transaction, no downtime for server restarts, machine failures, or data center outages, server restart, machine failures supported in zero downtime, replication locally or in wide area, no stale reads when failure occurs and use of Raft Consensus algorithm. CockroachDB uses timestamp cache to cache the last value of members read by ongoing transaction. It provides isolation in read and write transactions. It provides all the relational concepts of SQL such as schemas, tables, columns, indexes etc. No need to learn a new language, SQL developers can structure, manipulate and query the data using well-established, time-proven tools and processes. It also supports PostgreSQL wire protocol which helps developers to connect their applications simply by plugging language specific drivers for PostgreSQL. CockroachDB supports traditional ACID semantics, which means that if you have few servers at single location or many servers distributed across different datacenters, it doesn't make any difference for CockroachDB transactions.

CockroachDb's transactions are distributed across the cluster without need to know the precise location of data. Simply, you just talk to the one node and get whatever you want. CockroachDB uses a peer-to-peer gossip protocol to communicate opportunities for rebalancing. This protocol provides exchange of information between nodes such as storage capacity, network address or other information, so that it ensures automated scaling and repair and high availability.

Chapter 3

Implementation

In this chapter, we introduce all the prerequisites, instructions and steps, which took place in order to integrate the implementation of our project. We refer to all technical details and suggestions, based on scientific background, occupation and experience, that we have recorded to contribute to the minimum in the specific scientific field. More specifically, at section 3.1, we mention all the prerequisites and preparations that need to be made in order to shape the environment required for our system to work. At section 3.2, we formulate in detail the steps of code analysis and production of CockroachDB driver.

3.1 Cockroachdbdriver Implementation

The factors that determine how easy is the process of creating a driver are:

- Existence of libraries: developers can use a library to make system calls instead of implementing those system calls over and over again. In addition, the behavior is provided for reuse by multiple independent programs

- Piece of code that could easily be readable and configurable for future work (i.e. abstractdriver)
- Programming language version that is compatible with the majority of applications

According to Andy Pavlo, there are some instructions for how to implement a new driver for a system. The source code for the new driver must be put in a single Python source file inside of the drivers directory. The file name must contain the name of the system in lowercase with no spaces, followed by the word "driver". This file must contain a class that extends the AbstractDriver class that define the driver interface and implements the required functions listed below:

- Configuration Functions, which configure the driver prior to loading data and executing transactions
  - makeDefaultConfig(): represents the default configuration of the driver
  - loadConfig(): the configuration parameters for the driver
- Data Loading Functions, which are used by the controller to load data into the target system using the driver
  - loadStart(): callback to indicate to the driver that the data loading phase is about to begin
  - loadTuples(tableName, *tuples): load the list of tuples into the underlying database
  - loadFinish(): callback to indicate to the driver that the data loading phase is finished
  - loadFinishWarehouse(w_id): callback to indicate to the driver that all of the data for a given logical WAREHOUSE has been given to the driver
  - loadFinishDistrict(w_id, d_id): callback to indicate to the driver that all of the data for a given logical DISTRICT has been given to the driver
  - loadFinishItem(): callback to indicate to the driver that all of the ITEM data has been given to the driver
- Execution Functions

  - executeStart(): callback to indicate to the driver that the transaction execution phase is about to begin
  - executeFinish(): callback to indicate to the driver that the transaction execution phase is finished

- Transaction Functions, which are used to implement the actual logic of the native TPC-C transactions
  - doDelivery(params): implements the DELIVERY transaction
  - doNewOrder(params): implements the NEW_ORDER transaction
  - doOrderStatus(params): implements the ORDER_STATUS transaction
  - doPayment(params): implements the PAYMENT transaction
  - doStockLevel(params): implements the STOCK_LEVEL transaction

3.1.1 Study of existing Py-TPCC drivers

During development, we needed to see some implementations from the already existing Py-TPCC drivers. We got involved with mongodbdriver, scalarisdriver and sqlitedriver. All the drivers have got the same code-architecture base as they implement AbstractDriver and the same basic functions that regard the workload production. Differences regard database individuals. Depending on the database, drivers need different packages in order to connect and produce workload to the database. Although the main functions are the same within the drivers, their implementation differentiate depending on the perquisites of each of them. For some databases function implementations look simpler that others, which is reasonable as soon as each database has got its own demands. Further information about the differences requires deeper examination of each database driver which is beyond the scope of this thesis. All of the drivers use the following functions:

- loadTuples()
- loadConfig()
- doDelivery()
- doNewOrder()

- doPayment()
- doStockLevel()

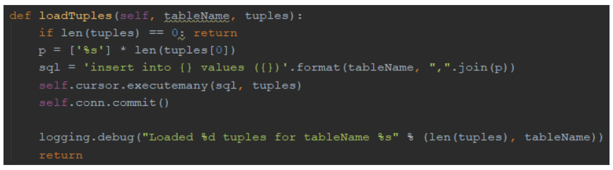
3.1.2 Psycopg2 and SQLite driver

Considering all of the above, we embarked on the core topic of this thesis. First of all, there was the need to use a library that would facilitate the simplification of code writing and would favor the interconnection and adaptation of the driver, and by extension the Py-TPCC benchmark, with the PostgreSQL database, used by CockroachDB. This is the library **Psycopg2** [29], which is the most popular PostgreSQL database adapter for the Python programming language. Its main features are the complete implementation of the Python DB API 2.0 specification and the thread safety (several threads can share the same connection). It was designed for heavily multi-threaded applications that create and destroy lots of cursors and make a large number of concurrent "INSERT"s or "UPDATE"s.

Psycopg2 is mostly implemented in C as a libpq wrapper, resulting in being both efficient and secure. It features client-side and server-side cursors, asynchronous communication and notifications, "COPY TO/COPY FROM" support. Many Python types are supported out-of-the-box and adapted to matching PostgreSQL data types; adaptation can be extended and customized thanks to a flexible objects adaptation system. Psycopg2 is both Unicode and Python 3 friendly. Observing the previously implemented drivers of Py-TPCC benchmark (Cassandra, CouchDB, CSV Output, HBase, HyperTable, Membase, MongoDB, Redis, Scalaris, SQLite), we recognised a big similarity between the basic functions (connect, fetchone, fetchall, execute, executemany) of Psycopg2 and SQLite3 library, which is getting import in SQLite driver [30]. Therefore, it was obvious to use SQLite driver as a template in order to build over there our CockroachDB driver.

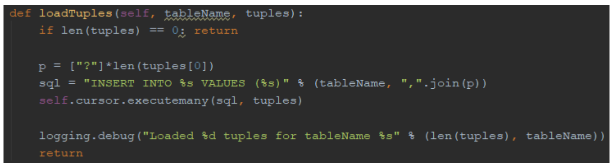
3.1.3 SQL compatibility

To the next level, the version of SQLite driver was old enough (Copyright 2011), and respectively the SQL version, so it was not recognizable by the Python 3.7.2 release. For this reason, we had to make changes and update the syntax of commands referred to the dictionary of "TXN_QUERIES" and method "loadTuples()", in order to be compatible with Python version. Moreover, there have to be modified the types



```
def loadTuples(self, tableName, tuples):
    if len(tuples) == 0: return
    p = ['%s'] * len(tuples[0])
    sql = 'insert into {} values ({})'.format(tableName, ",".join(p))
    self.cursor.executemany(sql, tuples)
    self.conn.commit()

    logging.debug("Loaded %d tuples for tableName %s" % (len(tuples), tableName))
    return
```

(a) CockroachDB loadTuples()



```
def loadTuples(self, tableName, tuples):
    if len(tuples) == 0: return

    p = ["?"]*len(tuples[0])
    sql = "INSERT INTO %s VALUES (%s)" % (tableName, ",".join(p))
    self.cursor.executemany(sql, tuples)

    logging.debug("Loaded %d tuples for tableName %s" % (len(tuples), tableName))
    return
```

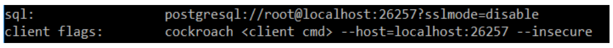
(b) SQLite loadTuples()

Figure 3.1: Function loadTuples()

below, so there is compatibility with CockroachDB SQL version in order to have CREATEs and DROPs of tables occasionally:

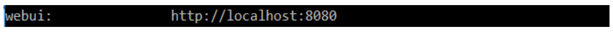
- Datetime instead of Timestamp
- Int instead of Tinyint
- Syntax of references: at the old SQL version, tables' primary keys were written in random order as a result there were not recognized the references between tables

3.1.4 CockroachDB Port



```
sql:             postgresql://root@localhost:26257?sslmode=disable
client flags:    cockroach <client cmd> --host=localhost:26257 --insecure
```

(a) CockroachDB localhost port



```
webui:           http://localhost:8080
```

(b) CockroachDB HTTP port

Figure 3.2: CockroachDB Ports

Furthermore, one more important thing that we have to consider was the port for communication between the two systems, CockroachDB and our implemented CockroachDB driver. For a local cluster, CockroachDB default port, used for internal and client traffic is 26257, which tells the node to listen only on localhost and for HTTP requests from the Admin UI the port is 8080.

3.1.5 Commit()

During the implementation and running of the driver, we have experienced the problem of the size of transaction from CockroachDB. The respective SQLite driver did

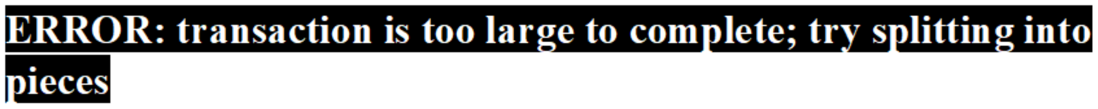

```
ERROR: transaction is too large to complete; try splitting into
pieces
```

Figure 3.3: Size Commit Error

not contain any limitation on the size of the transaction. CockroachDB uses a restriction on query size in order to protect the cluster memory usage. A single statement can perform at most 64MiB of combined updates. When a statement exceeds these limits, its transaction gets aborted. Currently, INSERT INTO ... SELECT FROM and CREATE TABLE AS SELECT queries may encounter these limits.

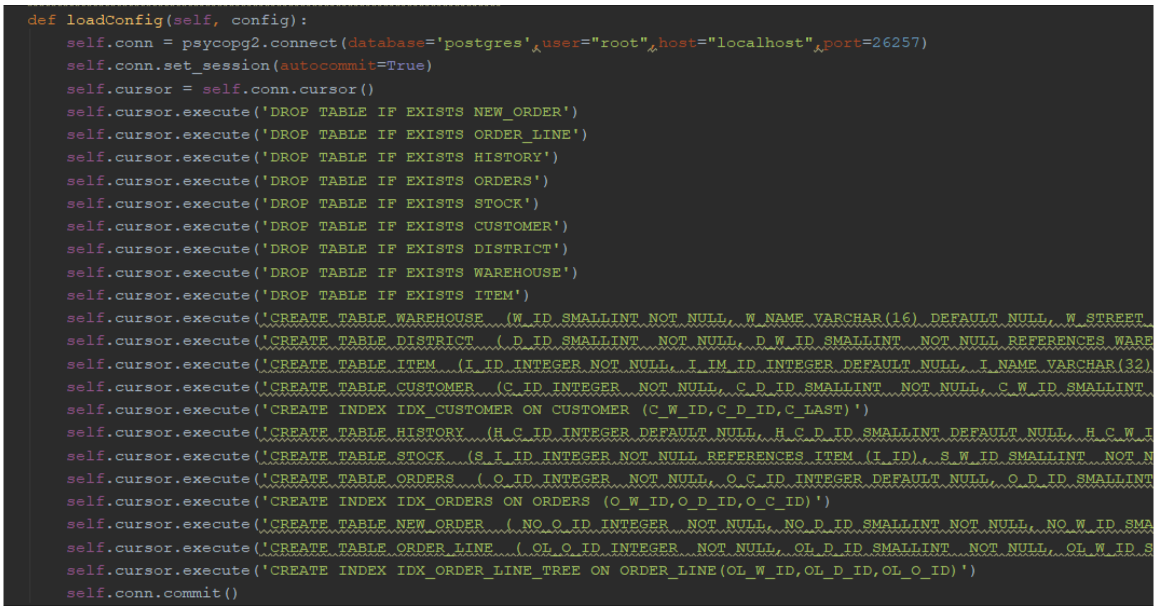
3.1.6 Function loadConfig()

Finally, we were concerned with the development of the loadConfig() function, which is the function responsible for the communication with the PostgreSQL database of CockroachDB. From here, the driver should clear out all of the records in the database prior to returning and load tuples or execute the transactional workload, respectively. More specifically, loadConfig() method accomplish two actions:

- Connection with PostgreSQL of CockroachDB
- Drops, if already exist, and Creates the Tables in PostgreSQL of CockroachDB

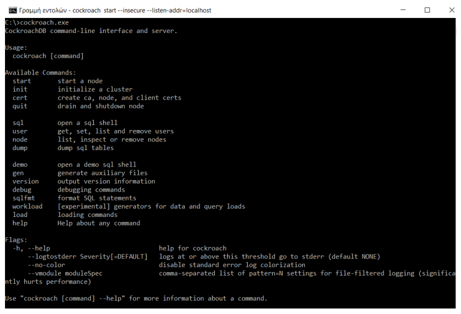
3.1.7 Batch_size

In native TPC-C benchmark, the number of queries required to complete a transaction in order to be ready for execute consists the batch_size. This constant variable is set at 10.000 queries by default in native TPC-C. So, we had to set the same batch size for our implementation of Py-TPCC cockroachdb driver.



```
def loadConfig(self, config):
    self.conn = psycopg2.connect(database='postgres',user="root",host="localhost",port=26257)
    self.conn.set_session(autocommit=True)
    self.cursor = self.conn.cursor()
    self.cursor.execute('DROP TABLE IF EXISTS NEW_ORDER')
    self.cursor.execute('DROP TABLE IF EXISTS ORDER_LINE')
    self.cursor.execute('DROP TABLE IF EXISTS HISTORY')
    self.cursor.execute('DROP TABLE IF EXISTS ORDERS')
    self.cursor.execute('DROP TABLE IF EXISTS STOCK')
    self.cursor.execute('DROP TABLE IF EXISTS CUSTOMER')
    self.cursor.execute('DROP TABLE IF EXISTS DISTRICT')
    self.cursor.execute('DROP TABLE IF EXISTS WAREHOUSE')
    self.cursor.execute('DROP TABLE IF EXISTS ITEM')
    self.cursor.execute('CREATE TABLE WAREHOUSE  (W_ID SMALLINT NOT NULL, W_NAME VARCHAR(16) DEFAULT NULL, W_STREET
    self.cursor.execute('CREATE TABLE DISTRICT  ( D_ID SMALLINT  NOT NULL, D_W_ID SMALLINT  NOT NULL REFERENCES WARE
    self.cursor.execute('CREATE TABLE ITEM  (I_ID INTEGER NOT NULL, I_IM_ID INTEGER DEFAULT NULL, I_NAME VARCHAR(32)
    self.cursor.execute('CREATE TABLE CUSTOMER  (C_ID INTEGER  NOT NULL, C_D_ID SMALLINT  NOT NULL, C_W_ID SMALLINT
    self.cursor.execute('CREATE INDEX IDX_CUSTOMER ON CUSTOMER (C_W_ID,C_D_ID,C_LAST)')
    self.cursor.execute('CREATE TABLE HISTORY  (H_C_ID INTEGER DEFAULT NULL, H_C_D_ID SMALLINT DEFAULT NULL, H_C_W_I
    self.cursor.execute('CREATE TABLE STOCK  (S_I_ID INTEGER NOT NULL REFERENCES ITEM (I_ID), S_W_ID SMALLINT  NOT N
    self.cursor.execute('CREATE TABLE ORDERS  ( O_ID INTEGER  NOT NULL, O_C_ID INTEGER DEFAULT NULL, O_D_ID SMALLINT
    self.cursor.execute('CREATE INDEX IDX_ORDERS ON ORDERS (O_W_ID,O_D_ID,O_C_ID)')
    self.cursor.execute('CREATE TABLE NEW_ORDER  ( NO_O_ID INTEGER  NOT NULL, NO_D_ID SMALLINT NOT NULL, NO_W_ID SMA
    self.cursor.execute('CREATE TABLE ORDER_LINE  ( OL_O_ID INTEGER  NOT NULL, OL_D_ID SMALLINT  NOT NULL, OL_W_ID S
    self.cursor.execute('CREATE INDEX IDX_ORDER_LINE_TREE ON ORDER_LINE(OL_W_ID,OL_D_ID,OL_O_ID)')
    self.conn.commit()
```

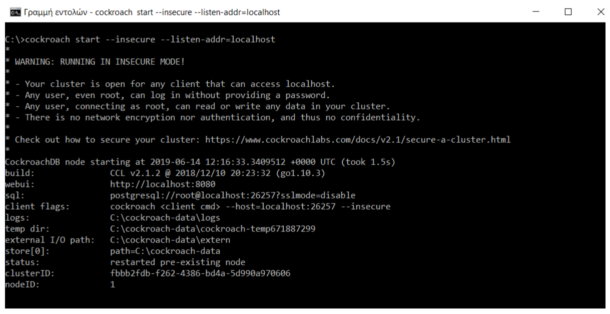
Figure 3.4: Function loadConfig()



Γραμμή εντολών - cockroach start --insecure --listen-addr=localhost

```
C:\>cockroach.exe
CockroachDB command-line interface and server.

Usage:
  cockroach [command]

Available Commands:
  start       start a node
  init        initialize a cluster
  cert        create ca, node, and client certs
  quit        drain and shutdown node

  sql         open a sql shell
  user        get, set, list and remove users
  node        list, inspect or remove nodes
  dump        dump sql tables

  demo        open a demo sql shell
  gen         generate auxiliary files
  version     output version information
  debug       debugging commands
  sqlfmt      format SQL statements
  workload    [experimental] generators for data and query loads
  load        loading commands
  help        Help about any command

Flags:
  -h, --help                           help for cockroach
      --logtostderr Severity[=DEFAULT]   logs at or above this threshold go to stderr (default NONE)
      --no-color                         disable standard error log colorization
      --vmodule moduleSpec               comma-separated list of pattern=N settings for file-filtered logging (significa
ntly hurts performance)

Use "cockroach [command] --help" for more information about a command.
```

(a) Cockroach.exe



Γραμμή εντολών - cockroach start --insecure --listen-addr=localhost

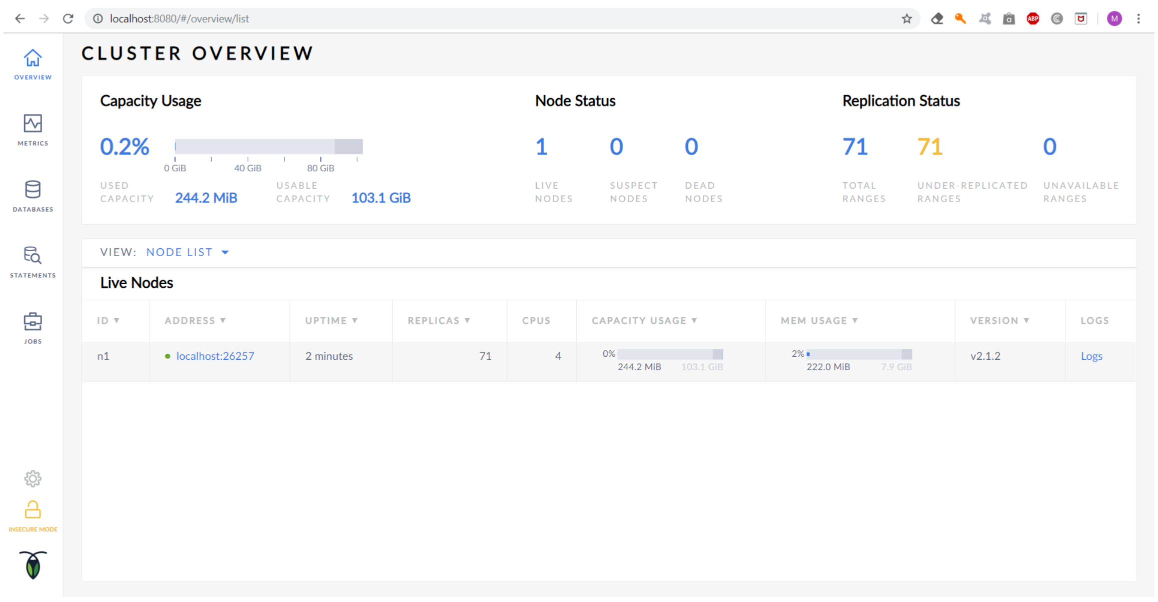
```
C:\>cockroach start --insecure --listen-addr=localhost
*
* WARNING: RUNNING IN INSECURE MODE!
*
* - Your cluster is open for any client that can access localhost.
* - Any user, even root, can log in without providing a password.
* - Any user, connecting as root, can read or write any data in your cluster.
* - There is no network encryption nor authentication, and thus no confidentiality.
*
* Check out how to secure your cluster: https://www.cockroachlabs.com/docs/v2.1/secure-a-cluster.html
*
CockroachDB node starting at 2019-06-14 12:16:33.3409512 +0000 UTC (took 1.5s)
build:               CCL v2.1.2 @ 2018/12/10 20:23:32 (go1.10.3)
webui:               http://localhost:8080
sql:                 postgresql://root@localhost:26257?sslmode=disable
client flags:        cockroach <client cmd> --host=localhost:26257 --insecure
logs:                C:\cockroach-data\logs
temp dir:            C:\cockroach-data\cockroach-temp671887299
external I/O path:   C:\cockroach-data\extern
store[0]:            path=C:\cockroach-data
status:              restarted pre-existing node
clusterID:           fbbb2fdb-f262-4386-bd4a-5d990a970606
nodeID:              1
```

(b) Cockroach.init

Figure 3.5: CockroachDB Installation

3.2 Cockroach DB and Prerequisites

For the implementation on this thesis, we assume that the installation is done on Microsoft Windows 10 64bit operating system locally on a desktop with CPU: Intel(R) Core(TM) i7-7500U @2.70GHz with 2 cores (2 logical cores per physical) and 8 GB RAM. Firstly, we downloaded the binary file and in a local host ran command ”C:/cockroach-v2.1.7.windows-6.2-amd64> ./cockroach.exe” from the directory of the file, where databases default, postgres & system are generated. After that, CockroachDB offers a web user interface for an optical representation of databases and the whole schema furthermore 3.6. Moreover, for the implementation of cockroachdb-driver.py, we installed Python 3.7.2 version.



CLUSTER OVERVIEW

Capacity Usage

0.2%

USED CAPACITY 244.2 MiB USABLE CAPACITY 103.1 GiB

Node Status

1	0	0
LIVE NODES	SUSPECT NODES	DEAD NODES

Replication Status

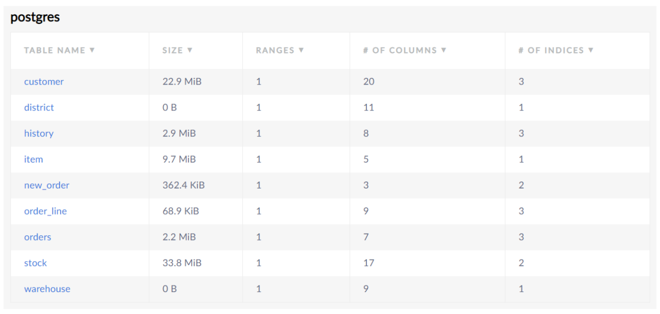
71	71	0
TOTAL RANGES	UNDER-REPLICATED RANGES	UNAVAILABLE RANGES

VIEW: NODE LIST

Live Nodes

ID	ADDRESS	UPTIME	REPLICAS	CPUS	CAPACITY USAGE	MEM USAGE	VERSION	LOGS
n1	localhost:26257	2 minutes	71	4	0% 244.2 MiB 103.1 GiB	2% 222.0 MiB 7.9 GiB	v2.1.2	Logs

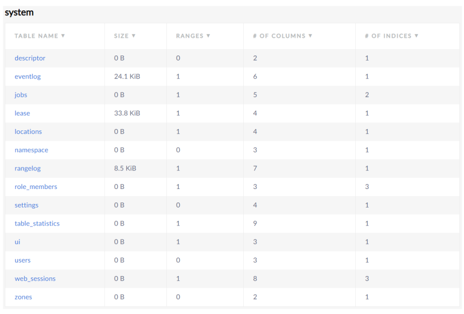
Figure 3.6: CockroachDB Web UI



postgres

TABLE NAME	SIZE	RANGES	# OF COLUMNS	# OF INDICES
customer	22.9 MiB	1	20	3
district	0 B	1	11	1
history	2.9 MiB	1	8	3
item	9.7 MiB	1	5	1
new_order	362.4 KiB	1	3	2
order_line	68.9 KiB	1	9	3
orders	2.2 MiB	1	7	3
stock	33.8 MiB	1	17	2
warehouse	0 B	1	9	1

(a) Postgres database



system

TABLE NAME	SIZE	RANGES	# OF COLUMNS	# OF INDICES
descriptor	0 B	0	2	1
eventlog	24.1 KiB	1	6	1
jobs	0 B	1	5	2
lease	33.8 KiB	1	4	1
locations	0 B	1	4	1
namespace	0 B	0	3	1
rangelog	8.5 KiB	1	7	1
role_members	0 B	1	3	3
settings	0 B	0	4	1
table_statistics	0 B	1	9	1
ui	0 B	1	3	1
users	0 B	0	3	1
web_sessions	0 B	1	8	3
zones	0 B	0	2	1

(b) System database

Figure 3.7: CockroachDB Tables

Chapter 4

Evaluation

In this chapter, we analyse the experimental evaluation of our work. More specifically, we compare the performane between the native TPC-C and Py-TPCC benchmark. In this case, we could derive useful information from the visual representation of the SQL database schema, offered by the CockroachDB Admin UI.

4.1 Py-TPCC vs Native TPC-C CLI differences

Initially, as a theoretical background, we lay down the differences that distinguish the Py-TPCC and native TPC-C benchmark. CockroachDB has built-in benchmarks and one of them is native TPC-C. All of the benchmarks use the same built-in SQL driver named Cockroachdriver, which is a wrapper around lib/pq (Figure 4.3). **lib/pq** is the C application programmer's interface to PostgreSQL. lib/pq is also the underlying engine for several other PostgreSQL application interfaces, including those written for C++, Perl, Python, Tcl and Go. lib/pq is a set of library functions that allow client programs to pass queries to the PostgreSQL backend server and to receive the results of these queries. In fact, this library acts as an emulation layer and provides round-robin load balancing amongst a list of Postgres URLs to connect to. In our case, round robin algorithm does not play any role, as our experiments are focused on the transactions themselves, and not the replication or sharding amongst several cluster nodes. Native TPC-C benchmark uses its own functions to generate transactions adjusted in Go language demands. Transaction values are generated randomly. Each transaction type has got inclusive creation and load function. The differences imply to code architectural style. For example, in native TPC-C load functions and related helper functions are separated to different code files while in Py-TPCC are all placed in one code file.

On the other hand, Py-TPCC benchmark consists of a driver that acts as a connection layer between the PostgreSQL of CockroachDB and the transaction generator using PostgreSQL adapter (Psycopg2 3.1.2). CockroachDB driver is used as link layer between client and cockroachdb. After connection is set, tables are created automatically. Values for table fills are generated in a random way, such as native TPC-C, and are written to the database. As soon as table filling finishes, Py-TPCC generates loads and executes transactions with corresponding functions as well. Each transaction type has got its own generate and load functions.

Below, there are some characteristics of native TPC-C benchmark, which are not contained at Py-TPCC:

- Choose generator seed (query values)
- Interleaved tables choice (–interleaved) (provides better query performance). Key-value ranges of closely related tables are optimized by attempting to be saved on the same key-value range. Behavior is not affected within SQL
- Table partitioning: user can define table partitions – choose how and where data is stored. Reduces latencies and costs.
  - Geo-partitioning (–partitions): location-based partitions; data must be saved close to the user (e.g if a district is located in Ioannina it would be wrong to be saved to a zone referred to Athens)
  - Archival-partitioning: set perquisites for data to be saved to nodes with specific features
  - In table creation:
    * List partitioning: enumerate all possible values for each

    * Range partitioning: specify range for each partition
    * Partition using primary key
  - Affinity partitions (–affinityPartition): run load generator against specific partition
  - Active warehouses (–activeWarehouses): run the load generator against a specific number of warehouses
  - Scatter ranges (–scatter)
  - Serializable (–serializable)
  - Split tables (–split)
  - Zones for partitioning (–zones): the number of zones should match the number of partitions and the zones used to start

We conclude that native TPC-C benchmark located in cockroach is well-suited and developed on the needs of CockroachDB.

In contrast, the similarities between the two benchmark are that both implementations are based on the same database schemas and workload concludes exactly the same insertion, update and delete queries.

With a first high level comparison, we notice that the native TPC-C benchmark is more monolithically written in relation with Py-TPCC, which explains the fact that it is not the same extensible. As we can see in figure 4.3, this becomes more understandable, taking into consideration that the Cockroach driver in native TPC-C constitutes a built in driver of the benchmark in contrast with Py-TPCC, where developer can choose which driver to implement, as we have already mentioned at 3.1.1.

Regarding the code architecture, Py-TPCC has much simpler code architecture style (Figure 4.1). Transaction value generation, transaction creation, transaction delivery and execution. In the file executor.py, parameters of the queries of each transaction are generated using random value generator. Also, there is a function called execute which is responsible for transaction execution. This function is strongly related with the benchmark run, as it caters for transaction start and execution during the benchmark execution interval. There is a function called inside named, executeTransaction. This function is always directed to corresponding executeTransaction of the abstract driver. In this way, execute transaction, refers to the actual transaction execu-

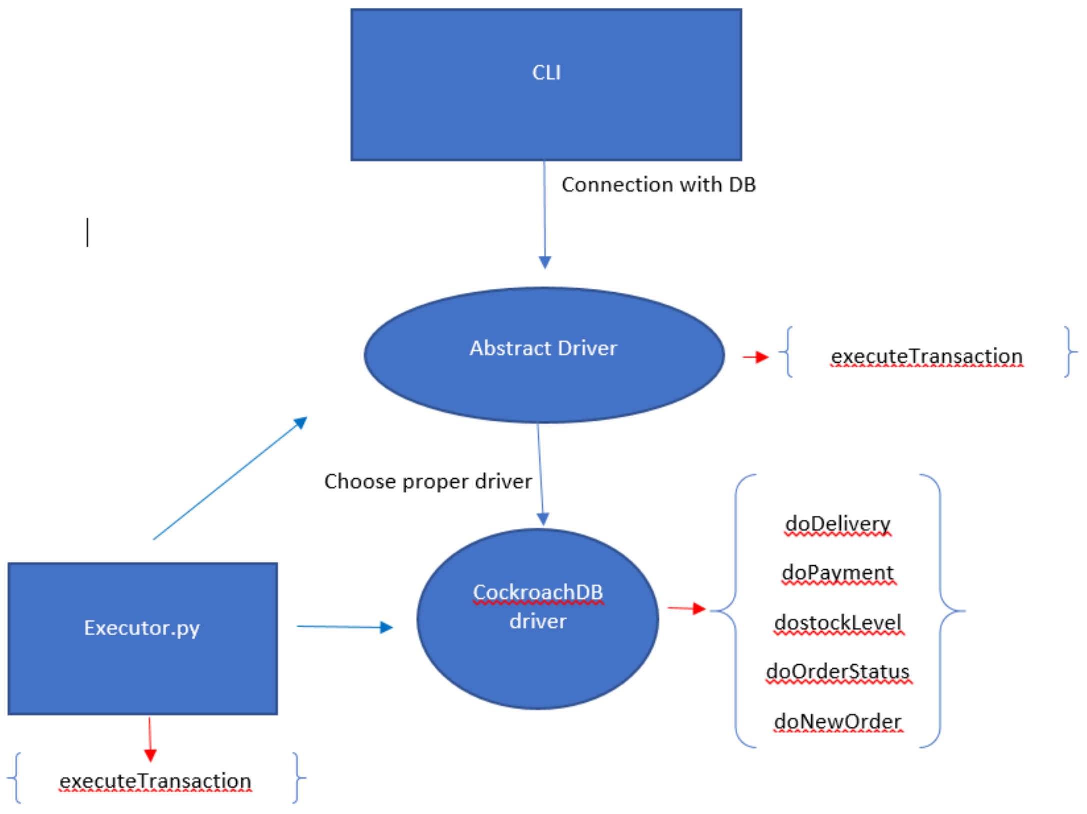


Figure 4.1: Py-TPCC Transaction process

tion functions written on the cockroachdb driver. The transaction execution functions of the cockroachdb driver are: doDelivery, doStockLevel, doPayment, doOrderStatus and doNewOrder, each for the synonym transaction type. In these functions, happens the actual execution of the transactions, using the execute function of the psycopg2 library 3.1.2. Functions that clearly regard transactions are implemented as we have already said within 3 main code files containing 500 lines of code for them.

In native TPC-C benchmark, things get more complex, as we can see in Figure 4.2. It is built in the original Cockroachdb code implementation provided in github [31]. There is a variety of benchmarks to choose containing native TPC-C. In native TPC-C transaction creation, execution and delivery happens within 5 files, each for every transaction type (delivery.go, payment.go, stocklevel.go, orderstatus.go, neworder.go). In each file, every transaction type has its own creation function (createStockLevel, createNewOrder, createDelivery, createOrderStatus, createPayment), where the corresponding queries of each one are created. First of all, in this functions there are filled in random way the values of the transaction queries. For transaction execution, each

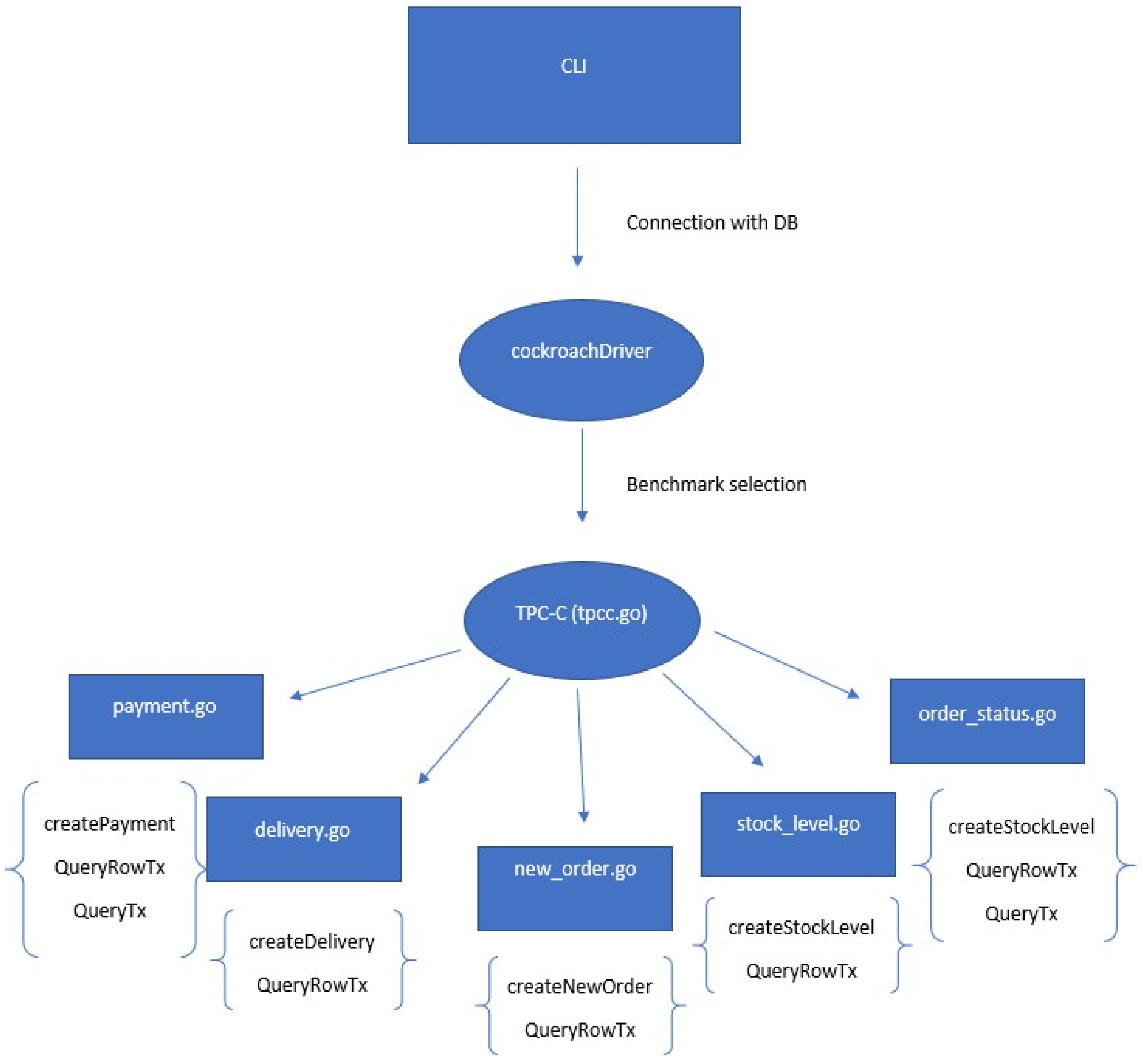


Figure 4.2: Native TPC-C Transaction process

file has got run fuction in which a sequence of several functions are called. The most basic of them are ExecuteInTx, which is an inline function, QueryRowTx or QueryTx, which is called for every single query execution in each transaction. Here, the size of the code including the above functions is 1.150 lines of code.

After completion of the implementation of our CockrochDB driver for Py-TPCC benchmark, we went through the process of running native TPC-C benchmark on CockroachDB in order to get some results to have some quantitative comparison between of them.

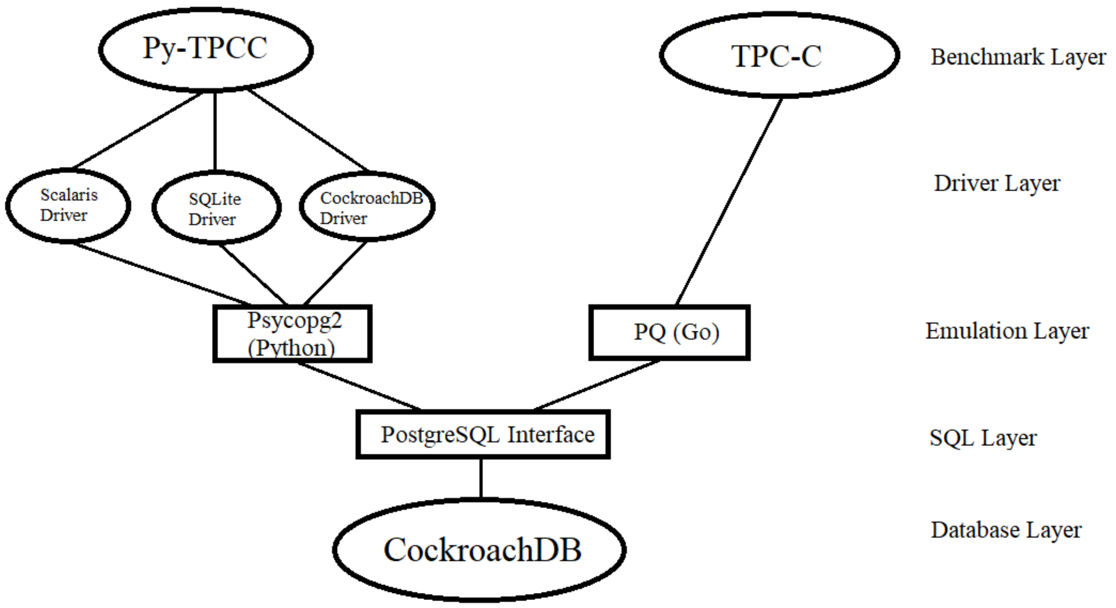
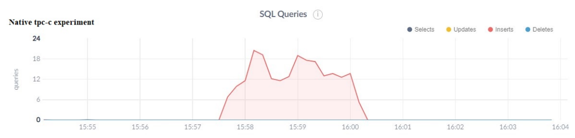


Figure 4.3: Native TPC-C vs Py-TPCC

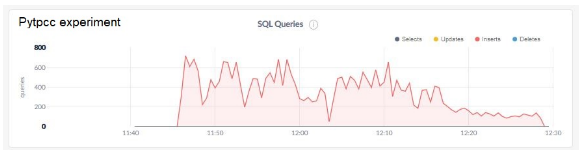
4.2 Py-TPCC vs Native TPC-C performance differences

In this section, we experimentally stress the CockroachDB both using Py-TPCC and native TPC-C benchmarks . In the context of this comparison, we executed in Py-TPCC the experiment for 10 clients and, respectively, we did the same in native TPC-C for 10 workers. We set the duration of our experiments at 60 seconds. So, we produced figures that describe in detail not only the quantitative differences between them, but the qualitative differences as well. All charts have been taken in real time from the CockroachDB Web UI and the X axis shows the time (measurement every 10 seconds). We depict the different ways in which the benchmarks affect the machine, as regards the CPU consumption and memory usage. We also check the costs of writing queries to prepare and fill the database for the transactions execution and the costs of performing transactions regarding the latency and the amount of queries that are finally performed. All of the experiments were hold multiple times in order to ensure the validity of our results. We distinguish 2 phases during experimentation:

- The phase of filling tables, where executed just insert queries
- The phase of experiment execution (performing transactions), where we have all types of queries (selects, updates, inserts, deletes).



(a) Queries native TPC-C



(b) Queries Py-TPCC

Figure 4.4: Queries during filling tables

4.2.1 Filling-in tables

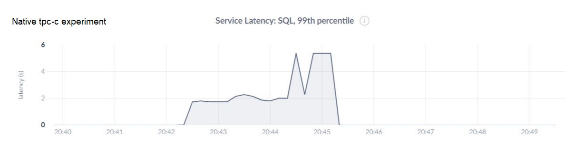
During the process of filling-in tables, each benchmark connects to the database, tables are created and their fill-in process starts.

During that time, we observe in Figure 4.4 that Py-TPCC performs a larger number of insert queries in a longer fill-in phase, compared to the native TPC-C. These differences in duration and number of SQL queries are caused by differences in the benchmark implementations. In native TPC-C table-creation and filling function, a bulk-insertion policy is used. Specifically, each worker issues set batches (bulks) of fill-in transactions, while the Py-TPCC benchmark executes insert transactions sequentially. Since tables are the same size in both benchmarks, we conclude that Py-TPCC performs a larger number of smaller insert transactions. The 99-th percentile latency[1] of native TPC-C is much higher than the corresponding latency of Py-TPCC (Figure 4.5).

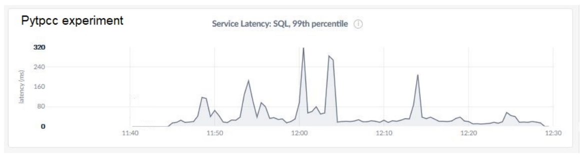
With increasingly stringent performance requirements of modern applications, the percentile (tail) latency has become an increasingly useful statistical measure of systems performance. This is the reason why in Figure 4.5 we depict the 99th percentile of the SQL queries. As we have already said, these figures are consisted of the phase of filling and we can easily spot some large differences. During the filling of tables, the duration lasts much longer in Py-TPCC comparing to native TPC-C benchmark. In native TPC-C experiment, we experience latency of order of seconds while in Py-TPCC of order of miliseconds. However, the time it takes to complete the process of filling tables in the native TPC-C is approximately 5 minutes, much less from the time it takes Py-TPCC to complete the same phase (approximately 50 minutes).

Concerning resource consumption during the fill-in phase, we can see in Figure

[1]With increasingly stringent performance requirements of modern applications, the percentile (tail) latency has become an increasingly useful statistical measure of systems performance.

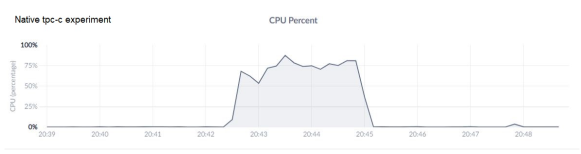


(a) 99th Percentile native TPC-C

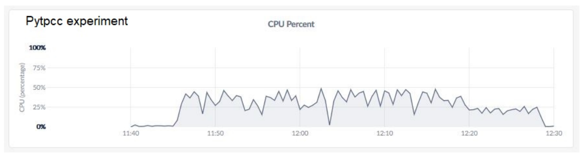


(b) 99th Percentile Py-TPCC

Figure 4.5: 99th Percentile during filling tables



(a) CPU Consumption native TPC-C



(b) CPU Consumption Py-TPCC

Figure 4.6: CPU Consumption during filling tables

4.6 and Figure 4.7 that both CPU consumption and memory usage are in agreement with the behavior described above. More specifically, requirements on computing power and memory are greater during native TPCC benchmark, as more resources are needed to serve the bigger queries in less time.

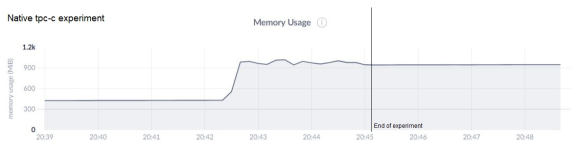
However, what really counts in benchmarking systems is the execution phase; being fast in the filling-phase is important, but not a critical factor for the evaluation process.

4.2.2 Experiment Execution

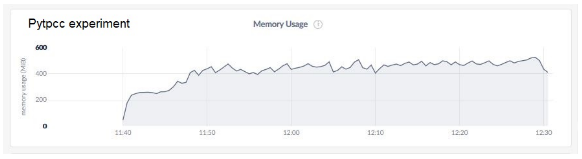
During the process of executing the core experimental phase of the TPC-C transaction mix, each benchmark executes transactions to the database for a duration of 60 seconds.

As we can see in Figure 4.8, the total number of transactions that take place is higher for the native benchmark (about 2x) compared to the Py-TPCC benchmark, and the increase (about 2x when increasing from 1 to 10 workers/clients, with a further 5-10% when increasing to 100 clients/workers) is nearly identical for the two benchmarks. The decreasing gains from an increasing level of load are due to saturation of resources in our testbed.

Similar to the number of transactions, the latency of native TPC-C benchmark is comparatively lower than the latency in the case of Py-TPCC, explaining the fact that native TPC-C can execute more transactions in the same time interval. More-

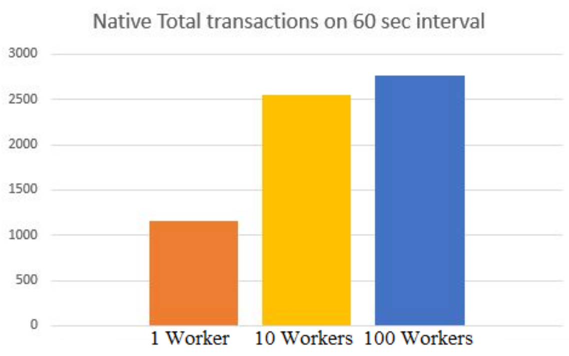


(a) Memory Usage native TPC-C

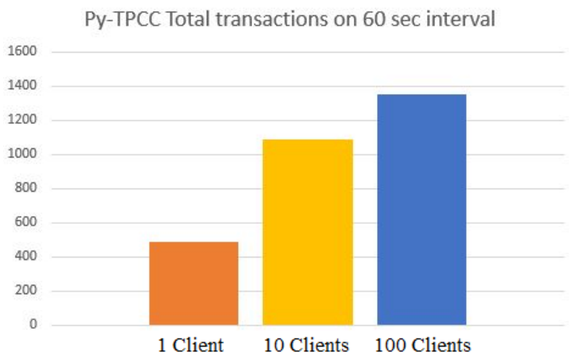


(b) Memory Usage Py-TPCC

Figure 4.7: Memory Usage during filling tables



(a) Native TPC-C total transactions

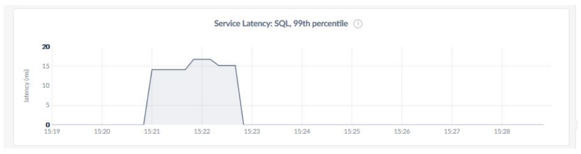


(b) Py-TPCC total transactions

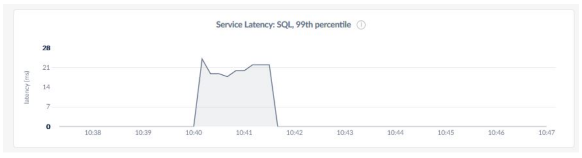
Figure 4.8: Total Transactions

over, we observe that as the number of workers/clients increases, the latency of both benchmarks gets higher; for the Py-TPCC benchmark this increase is greater than the native TPC-C for a larger number of clients, as shown in Figures 4.9, 4.10 and 4.11. With all else being equal, we believe that this performance difference between native TPC-C and Py-TPCC versions over CockroachDB is due to the performance of the PostgreSQL adaptation layer.

Finally, with regards to CPU consumption, we observe that the resources consumed by both benchmarks are similar. Figures 4.12, 4.13 and 4.14 depict that as the number of the workers/clients increases, CPU requirements increase as well, eventually reaching saturation.

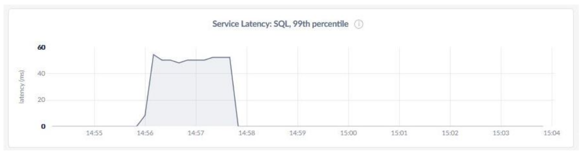


(a) Native TPC-C latency for 1 worker

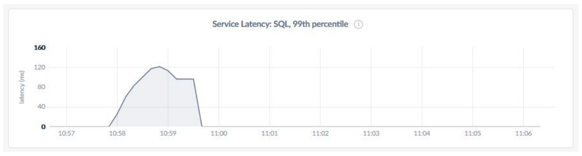


(b) Py-TPCC latency for 1 client

Figure 4.9: Latency for 1 worker/client

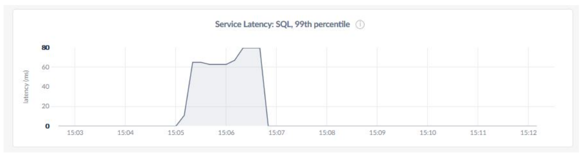


(a) Native TPC-C latency for 10 workers

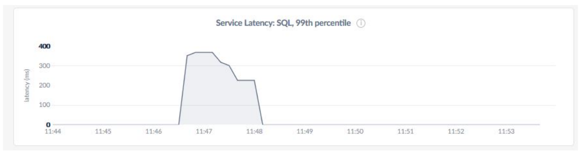


(b) Py-TPCC latency for 10 clients

Figure 4.10: Latency for 10 workers/clients

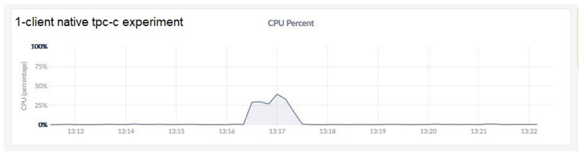


(a) Native TPC-C latency for 100 workers

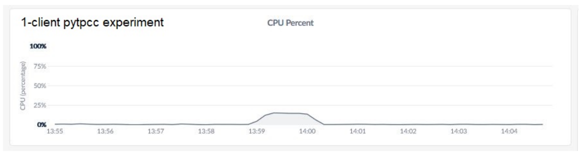


(b) Py-TPCC latency for 100 clients

Figure 4.11: Latency for 100 workers/clients

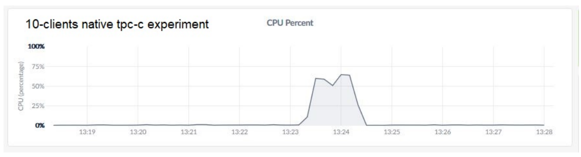


(a) Native TPC-C CPU consumption for 1 worker

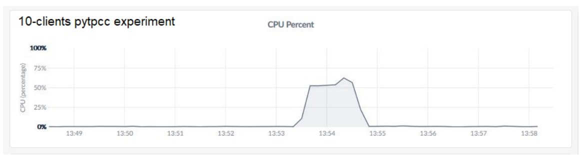


(b) Py-TPCC CPU consumption for 1 client

Figure 4.12: CPU Consumption for 1 worker/client

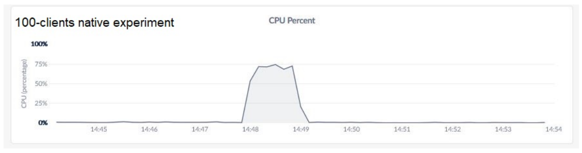


(a) Native TPC-C CPU consumption for 10 workers

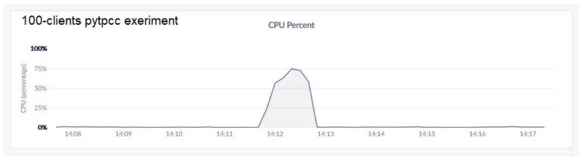


(b) Py-TPCC CPU consumption for 10 clients

Figure 4.13: CPU Consumption for 10 workers/clients



(a) Native TPC-C CPU consumption for 100 workers



(b) Py-TPCC CPU consumption for 100 clients

Figure 4.14: CPU Consumption for 100 workers/clients

Chapter 5

Conclusion and Future Work

In this chapter, we summarize our conclusions from the work carried out in the context of this thesis and describe possible future work. In Section 5.1, we report our conclusions and contributions through the study and evaluation of our implementation of a Cockroachdb driver for Py-TPCC. In Section 5.2, we suggest possible next steps.

5.1 Conclusions

In this thesis, we developed and evaluated a Cockroachdb driver for the Py-TPCC benchmark. We initially analyzed the theoretical background of different benchmarks and then studied the process of implementing a new driver for the Py-TPCC benchmark, differentiating with the a TPC-C benchmark written in Go and offered within the Cockroachdb distributions (referred to as native).

In Section 4.2.1 we observed that during the filling of the tables, the number of insert queries written in the database is much higher for the Py-TPCC than with native TPC-C. However, the time needed to complete the fill-in phase is shorter for native TPC-C compared to Py-TPCC benchmark. This behaviour is explained because of bulk transactions. During the transaction execution phase (Figure 4.2.2), set to run for 60 seconds, we observe that the number of transactions executed at the same time is higher for the native TPC-C benchmark than for Py-TPCC, with the number increasing with the increase of workers/clients. Since all other parameters of both systems (such as time of transaction execution phase, the database system, and the underlying platform) are equal, we believe that the difference in performance between the two benchmarks is due to the peformance capabilities of the SQL adaptation layer (psygopg2 vs. pg), which each benchmark uses to provide adaptation and connection with the underling database layer (Figure 4.3).

The code architecture and structure of the two TPC-C benchmark implementations (studied in Figures 4.2 and 4.3) have extensibility implications. The Py-TPCC architecture features one large code file (executor.py), which is easy to read and understand, and supports drivers in a database-independent manner; Py-TPCC drivers call a single function (executor.py) for the execution of transactions. In our experience, Py-TPCC is an easily extensible system. In native TPC-C, the creation of each different type of transaction is implemented by a specific function (payment.go, delivery.go, new_order.go, stock_level.go, order_status.go). Native TPC-C incorporates a built-in driver for CockroachDB and is tightly integrated to it, it is thus is not designed for extensibility. In general, a scalable database such as CockroachDB that exports an SQL interface drastically simplifies the process of porting a benchmark driver for it.

5.2 Future work

A research question that provided the initial motivation for this thesis, was whether an automated, or semi-automated process is possible for developing a new benchmark driver for an arbitrary data (key-value) store, for which a specification of its data-access API is available. While this thesis provides a first step into the study this research question, a full investigation extending our results would be an interesting future endeavor. Another possible future direction is to delve deeper into the complexities of different implementations of PostgreSQL adaptation layers and investigate scalability limits as well as evaluate benchmark performance in larger scale systems.

Bibliography

[1] “Key value store diagram.” https://en.wikipedia.org/wiki/Key-value_database#/media/File:KeyValue.PNG. Accessed: 2019-6-10.

[2] “Document store diagram.” https://developer.couchbase.com/documentation/server/3.x/developer/dev-guide-3.0/compare-docs-vs-relational.html. Accessed: 2019-6-10.

[3] “Wide column store diagram.” https://www.lynda.com/NoSQL-tutorials/Understanding-wide-column-stores/368756/387728-4.html. Accessed: 2019-6-10.

[4] “Graph store diagram.” https://whatis.techtarget.com/definition/graph-database. Accessed: 2019-6-10.

[5] “Acid transactions.” https://www.morpheusdata.com/blog/2015-01-29-when-do-you-need-acid-compliance. Accessed: 2019-6-10.

[6] “Cap theorem.” https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e. Accessed: 2019-6-10.

[7] “Native tpc-c database schema.” http://www.tpc.org/information/sessions/sigmod/sld009.htm. Accessed: 2019-6-10.

[8] “Ycsb architecture.” https://www.slideshare.net/sqrrl/ycsb-benchmarking. Accessed: 2019-6-10.

[9] “Difference between rdbms, nosql and cockroachdb.” https://callistaenterprise.se/blogg/teknik/2018/02/14/go-blog-series-part13/. Accessed: 2019-6-10.

[10] “Cockroachdb glossary.” https://www.cockroachlabs.com/docs/stable/architecture/overview.html. Accessed: 2019-6-10.

[11] "Cockroachdb layered architecture diagram." https://www.zcfy.cc/original/cockroach-design-md-at-master-cockroachdb-cockroach-github. Accessed: 2019-6-10.

[12] "Google cloud bigtable." https://cloud.google.com/bigtable/. Accessed: 2019-6-10.

[13] "Cassandra db." http://cassandra.apache.org/. Accessed: 2019-6-10.

[14] "Mongo db." https://www.mongodb.com/what-is-mongodb. Accessed: 2019-6-10.

[15] F. Gessert, W. Wingerath, S. Friedrich, and N. Ritter, "Nosql database systems: a survey and decision guidance," SIAM Review, vol. 45, no. 2, pp. 353–365, 2016.

[16] F. Moscato, R. Aversa, B. Di Martino, T. Fortiş, and V. Munteanu, "An analysis of mosaic ontology for cloud resources annotation," in 2011 Federated Conference on Computer Science and Information Systems (FedCSIS), pp. 973–980, Sep. 2011.

[17] S. Dalal, S. Temel, M. Little, M. Potts, and J. Webber, "Coordinating business transactions on the web," IEEE Internet Computing, vol. 7, pp. 30–39, Jan 2003.

[18] E. Brewer, "A certain freedom: Thoughts on the cap theorem," in Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC '10, (New York, NY, USA), pp. 335–335, ACM, 2010.

[19] "Tpc-c benchmark." http://www.tpc.org/default.asp. Accessed: 2019-7-1.

[20] V. Reniers, D. Van Landuyt, A. Rafique, and W. Joosen, "On the state of nosql benchmarks," in Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, ICPE '17 Companion, (New York, NY, USA), pp. 107–112, ACM, 2017.

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10, (New York, NY, USA), pp. 143–154, ACM, 2010.

[22] "Tpc-h benchmark." http://www.tpc.org/tpch/default.asp. Accessed: 2019-7-11.

[23] "Ycsb benchmark." `https://github.com/brianfrankcooper/YCSB`. Accessed: 2019-6-10.

[24] "Py-tpcc benchmark." `https://github.com/apavlo/py-tpcc`. Accessed: 2019-6-10.

[25] "Cockroachdb labs." `https://www.cockroachlabs.com/`. Accessed: 2019-6-10.

[26] "Postgresql." `https://www.postgresql.org/`. Accessed: 2019-6-10.

[27] A. Pavlo and M. Aslett, "What's really new with newsql?," SIGMOD Rec., vol. 45, pp. 45–55, Sept. 2016.

[28] K. Rabbani and I. Masli, "Cockroachdb: Newsql distributed, cloud native database," Universite Libre De Bruxelles, 2017.

[29] "Psycopg2." `https://pypi.org/project/psycopg2/`. Accessed: 2019-6-10.

[30] "Sqlite driver." `https://github.com/apavlo/py-tpcc/blob/master/pytpcc/drivers/sqlitedriver.py`. Accessed: 2019-6-10.

[31] "Tpcc transaction functions." `https://github.com/cockroachdb/cockroach/tree/master/pkg/workload/tpcc`. Accessed: 2019-6-10.

Short Biography

Michail Sotiriou is a M.Sc. graduate student at the Department of Computer Science and Engineering of University (CSE) of Ioannina, Greece. Michail received his B.Sc. degree from the School of Informatics of Aristotle University of Thessaloniki (AUTH). His research interests include NoSQL DBs on distributed systems and cloud serving systems.