Fall 2017

OASIS - Identifying the Core Attributes for RDBMS Alternatives

Benjamin P. McPherson

OASIS: IDENTIFYING THE CORE ATTRIBUTES FOR RDBMS SUBSTITUTES

by

BENJAMIN MCPHERSON

(Under the Direction of Vladan Jovanovic)

ABSTRACT

Since their introduction in the 1970s, relational database management systems have served as the dominate data storage technology. However, the demands of big data and Web 2.0 necessitated a change in the market, sparking the beginning of the NoSQL movement in the late 2000s. NoSQL databases exchanged the relational model and the guaranteed consistency of ACID transactions for improved performance and massive scalability [1]. While the benefits NoSQL proved useful, the lack of sufficient SQL functionality presented a major hurdle for organizations which require it to properly operate. It was clear that new RDBMS solutions which did not compromise functionality or scalability were necessary, which has led to the rise of a new class of modern relational database management systems, NewSQL [2].

This paper seeks to identify a consistent set of requirements necessary for an ideal RDBMS substitute. Among these requirements include possessing the features of a modern RDBMS, which include support of the relational data model and standard ANSI SQL, ACID transactions, and ODBC/JDBC drivers. Additionally, the substitute must address typical RDBMS' shortcomings in scalability by providing cost-effective scale-out capabilities. These requirements will then be used to filter out existing NoSQL and NewSQL database systems which could serve as viable substitutes to a typical RDBMS.

INDEX WORDS: ACID, Database, Hadoop, NewSQL, NoSQL, RDBMS

OASIS: IDENTIFYING THE CORE ATTRIBUTES FOR RDBMS SUBSTITUTES

by

BENJAMIN MCPHERSON

B.S. Valdosta State University, 2012

A Thesis Submitted to the Graduate Faculty of Georgia Southern University

in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

STATESBORO, GEORGIA

OASIS: IDENTIFYING THE CORE ATTRIBUTES FOR RDBMS SUBSTITUTES

by

BENJAMIN MCPHERSON

Major Professor:	Vladan Jovanovic
Committee:	Lixin Li
	Wen-Ran Zhang

Electronic Version Approved:
December 2017

ACKNOWLEDGMENTS

I would like to thank my family, my dog Beau, my loving girlfriend, Taylor, and friends for keeping me motivated and on the right track. Without them, I would not be where I am today. I would also like to thank Dr. Vladan Jovanovic, who has been an exceptional source of support and guidance throughout this research, as well as a professor who has inspired me to further my research into the field of databases and data warehousing. Finally, I have to give my thanks to Grassroots Coffee of Thomasville, GA for providing me with plenty of coffee and a wonderful atmosphere to get a majority of my work done.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

This paper presents OASIS, a consistent set of easily identifiable core attributes which are essential in viable substitutes to Relational Database Management Systems (RDBMSs). These substitutes must possess the capabilities of a traditional RDBMSs such as Oracle, MySQL, and Microsoft SQL Server, such as their ACID qualities, while seeking to overcome the issues of scaling and costs which have plagued the platform in the era of big data. By using OASIS, existing NoSQL and NewSQL systems can be classified as having the base attributes necessary to be labeled as RDBMS replacements. Organizations can then focus on identifying the systems which then possess features necessary for their operations rather than having to identify substitutes from square one.

Chapter 2 summarizes the background of database systems development from their inception to the present. Chapter 3 presents a detailed look at the currently available RDBMSs, NoSQL, and NewSQL database systems, their unique features, and disadvantages. Chapter 4 introduces OASIS and defines the attributes a system should possess to serve as a viable RDBMS substitute. Chapter 5 explores if those attributes are present in existing database systems and discusses how each goes about handling these attributes. Finally, Chapter 6 summarizes the results of the analysis from Chapter 5.

CHAPTER 2

BACKGROUND

Since their introduction in the 1970s, relational database management systems have served as the dominate data storage technology. Using the relational model first described in 1969 and again in 1970 by Edgar Codd [3, 4], their qualities of Atomicity, Concurrency, Isolation and Durability (ACID) ensure that data maintains a state of accuracy, making relational databases an invaluable tool for storing data which relies on consistency and precision. However, these databases possessed several limitations, namely with regards to scaling, that became more apparent through the years as larger quantities of data needed to be stored.

This came to an ahead with the introduction of "big data" and rise in the web applications in the early 2000s. The exploding size and scale of data pushed the limits of relational databases, which were no longer able to handle the influx of data and user traffic without having to make costly compromises such as data pruning or cost-prohibitive upward scaling on proprietary hardware [5]. To address these issues, a new kind of DBMS that differed significantly from the classic RDBMS model was introduced. Dubbed Not Only SQL (NoSQL), these open-source, distributed, non-relational databases were developed in response to the fact that the existing database products did not meet requirements with regards to scalability, performance, (relaxed) consistency, agility, and intricacy.

While NoSQL managed to solve many of the problems RDBMSs had with big data, they have been unable to completely take their place due to the need for their transactional and consistency requirements. NewSQL seeks to find a compromise between RDBMS and NoSQL, blurring the lines between the categories of database systems by providing scalability of NoSQL systems while maintaining the ACID qualities of relational database systems [6].

CHAPTER 3

CURRENT TECHNOLOGY LANDSCAPE

3.1 Relational Database Management Systems

Relational databases are the oldest and most widely used database technologies in the world. The foundation for their development began with E.F. Codd's 1970 paper "A Relational Model of Data for Large Shared Data Banks". In it, he describes the relational model, a structured method for storing data based upon principles of relational algebra. This model allows data to be split into normalized relations (tables) to reduce redundancy, saving storage space, as well as removing unnecessary dependencies among the data. This model also allows for these relations to be manipulated by an assortment operations including, but not limited to, permutations, projections, and joins [1]. The most popular and widely used relational database systems are Oracle, MySQL, Microsoft SQL Server, PostgreSQL and IBM DB2 [9].

3.1.1 ACID Properties

A key feature of modern relational database management systems is the adherence to the properties of Atomicity, Consistency, Isolation and Durability (ACID). These properties are important in maintaining reliable and accurate transactions. The concepts for this acronym came from two papers by computer scientist Jim Gray. The first of these papers, written in 1976, was titled "Granularity of locks and degrees of consistency in a large shared data base."

A major focus of this paper is consistency, with the second section of the paper being dedicated entirely to it. Consistency is described as a guarantee that the database remains in a consistent state at the end of any transaction by satisfying all of its assertions. It's here that Gray et al describes the need for varying degrees of consistency due to the need to cope with temporary inconsistencies required to move to another state of consistency.

Degree 0:

a. Does not overwrite dirty data of other transactions.
b. Commits writes before end of transaction.

Degree 1:

a. Does not overwrite dirty data of other transactions.
b. Does not commit any writes before end of transaction.

Degree 2:

a. Does not overwrite dirty data of other transactions.
b. Does not commit any writes before end of transaction.
c. Does not read dirty data of other transactions through use of isolation.

Degree 3:

a. Does not overwrite dirty data of other transactions.
b. Does not commit any writes before end of transaction.
c. Does not read dirty data of other transactions through use of isolation.
d. Other transactions do not dirty any data read by transaction before transaction is complete.

As can be seen in the figure above, the greater the degree of consistency, the greater the level of isolation a transaction is experiencing [12]. Isolation is most simply described as the visibility of a transaction to the system and to other users, although this definition would not appear for several years [13].

Gray's next paper came in 1981. “The Transaction Concept: Virtues and Limitations” served to create a concrete definition of a database transaction, deriving the concept from contract law. In drawing up a contract, a series of events must take place for the contract to be made. First, two or more parties negotiate before making a deal. Then, the deal is made binding

by an act that all parties agreed to, such as a signature, a handshake, or nod. While a contract is simply an agreement, their qualities and those of transactions are much the same. Below are these transaction properties:

Consistency: the transaction must obey the legal protocols, be completed and maintain consistency throughout the process

Atomicity: It either happens or it does not; either all are bound by the contact or none are

Durability: once the transaction is committed, it cannot be abrogated [11].

All transactions must have these qualities in order to maintain optimal effectiveness. Notice that atomicity and consistency once again are a key focus, with consistency of the data base being the result which atomicity and durability strive to insure.

Gray's two papers laid the groundwork for ACID, but the term would not appear until 1983, when Theo Haerder and Andreas Reuter published “Principles of Transaction-Oriented Database Recovery.” It is in this paper that the ACID acronym is used for the first time. Additionally, the properties of consistency, atomicity and durability from Gray’s 1981 paper are adapted and further emphasized. Isolation, an implicit topic in Gray et al's 1976 paper, is given a formal definition and declared a property of transactions. All together, they are referred to as the "ACID principle", which describe the transaction paradigm and state that a database is consistent if and only if it contains the results of successful transactions [13]. Adhering to this principle can allow for multiple users to access and manipulate data concurrently without damaging the integrity of the database and its relations.

3.1.2 Limitations of Relational Databases

Relational Databases were developed at a time where data was everywhere. Few organizations and individuals made use of databases and when they did, a standard computer hard drive or single server could serve their needs. Servers were not receiving requests to store and retrieve data for millions of users from multiple devices from all over the world. Today's standard enterprise-level data storage needs require multiple server stacks for the terabytes of data necessary for critical operations. According to a report by the McKinsey Global Institute in 2011, it was estimated that by 2009, nearly all sectors in the US economy had at least an average of 200 terabytes of stored data per company with more than 1,000 employees. This amount of data is twice that of US retailer Wal-Mart's data warehouse in 1999 [5, 14]. This increase in data reveals two major limitations of relational databases.

The first of these limitations involves one of the main features of RDBMS, its ACID properties. ACID's guaranteed consistency is achieved through isolating data, typically through the use of locks and serialized schedules, in order to prevent transactions from working with uncommitted data and producing dirty data [15]. However, employing these measures produces overhead which takes up the resources of the database server, ultimately limiting the parallel functionality of the database. While relaxing the isolation level of the database to improve performance is possible with some RDBMSs, it risks compromising the integrity of the data.

The second limitation is that a database must properly scale with the amount of data and user traffic. Unfortunately, this translates into a financial burden for organizations as they have to invest in costly software licenses and proprietary hardware in order to accommodate. Worse, this is come off as a Band-Aid solution that fails to address the core issue: relational databases weren't designed to support large-scale data, data throughput, and modern techniques, such as data partitioning [6].

3.2 Not Only SQL

In response to the surge in data and the necessity of high scalability in the late 2000s, NoSQL databases were developed. As discussed previously in Section 3.1.2, relational databases were not designed to tackle the challenges of storing and leveraging big data. This

problem is tied to its ACID guarantees, which consume resources in isolating a transaction, making it difficult for concurrency and distributed database set-ups. In contrast, Not Only SQL (NoSQL) database systems make use of eventual guarantees under the CAP theorem, allowing them to be fast and easily distributable systems, making them fit for being Big Data databases. There are five general classifications of NoSQL databases which, while all subscribing to the CAP theorem, all take their own approach. Among the most popular NoSQL databases are MongoDB, Apache Cassandra, and Redis [10].

3.2.1 BASE

BASE, or Basically Available Soft-state Eventual consistency, is a consistency model used in NoSQL database systems. The concept of BASE was introduced in a 1997 paper by Berkeley computer scientists Eric Brewer, Armando Fox and called "Cluster-Based Scalable Network Services" [41]. BASE serves as a contrast to ACID by trading in guaranteed consistency for eventual consistency, accepting the use of possibly out-of-date data in exchange for improved performance. While consistency is important, the goal is to maintain availability for cluster-based network systems. This is especially important in the era of Web 2.0, where web-based services rely on being available if possible [41].

3.2.2 CAP Theorem

CAP, an acronym which stands for Consistency, Availability, and Partition Tolerance, is seen as the driving principle behind the loosening of ACID guarantees and greatly impacted the development of NoSQL systems. CAP was developed by Brewer in 1998 and would first appear in published form in Brewer and Fox's 1999 journal article "Harvest, Yield, and Scalable Tolerant Systems" as the CAP principle. The article describes two variations of the CAP principle, a Strong CAP Principle and a Weak CAP Principle. The Strong CAP Principle is described as possessing the properties of Strong Consistency, High Availability and Partition Resilience [16]. The definitions of these properties according to Brewer and Fox are as follows. Strong Consistency means single-copy ACID consistency. High availability is provided through some form of redundancy, such as data replication, so that some of the data is available at any

given time. Partition resilience means the system can survive a partition between the replicas of data.

The Strong CAP principle recommends that a system should choose at most two of these properties to focus on, creating pairs of properties taken from the principle. The article does note, however, that while this works in theory, real world applications complicate the situation. This results in the Weak CAP principle, which was not precisely characterized by the article, which states that the stronger the guarantee that is made for any two principles, the weaker the guarantee of the third principle [Harvest]. Thus, it is an impossibility to have a database that is fully consistent, fully available, and fully partition tolerant. The continued emphasis of the Strong CAP principle, with an unclear focus on the Weak CAP principle, is cemented in Brewer's presentation made at the Principles in Distributed Systems Conference in 2000 "Towards Robust Distributed Systems", where Strong CAP principle is renamed the CAP Theorem [17].

3.2.3 Types of NoSQL Databases

NoSQL databases can be grouped into five different types: Key-value Stores, Column Family Stores, Document Stores, Graph Databases, and Multi-Model Databases. While they all have a foundation based on the CAP Principle and BASE, the methods they use to store data, the complexity of said data, and the particular jobs the different types excel at vary.

Key-value stores are considered the simplest type of NoSQL database [18]. Data is stored as a collection of objects without predefined fields and data types. A key-value is then used to refer to a particular data collection and is the only way data can be searched and retrieved. While this makes it unfit for being a substitute to a traditional relational database, it works best for storing large lists and catalogs of items which are normally searched by key-values (e.g. serial numbers) [19]. Column Family Stores, also known as Wide Column Stores, contrast relational databases in that their data is stored in columns rather than rows. These columns have three elements, a unique key-value name, the data itself, and a timestamp, with the data itself able to be stored without the use of nulls in order to save storage space.

Document Stores are defined by their ability to store their data as documents. These documents are non-restrictive and time-stamped and are accessed using a search function similar

to that of how Key-Value stores in addition to being able to search by the information found in the document. Graph databases are the most complex type of NoSQL database, as well as the most different. Based on graph theory, graph databases excel at tracking relationships, called edges, between entities, called nodes. Their strongest attribute beyond representing relationships is their ability to traverse large amounts of data quickly. The final type of NoSQL databases is multi-model. Unlike the other types previously discussed, multi-model databases are systems designed to handle multiple data models, granting them great flexibility and consolidating an organization's data into a single system [20].

3.2.4 Disadvantages

The main disadvantage of NoSQL is the very thing which gives it its performance and distribution advantages, which is its lack of ACID properties. The lack of relational capabilities, which are often engrained in an organizations data storage and analytics systems, means that being able to fully utilize a NoSQL database means incorporating other technologies to convert from relational to non-relational data or completely rebuilding the system around NoSQL. Both options are expensive and time consuming, often resulting in organizations maintaining their current set-up for as long as feasibly possible. Beyond the technology, one of the biggest hurdles in the adaptation of NoSQL is the scarcity of individuals with the necessary skill and know-how to properly utilize NoSQL interfaces to manipulate big data for the purpose of analysis. This, combined with the additional need for traditional data management skills places a huge burden on the limited resources that are the talent pool [7].

3.3 NewSQL

First coined by the 451 Group in their report titled "NoSQL, NewSQL and Beyond", NewSQL was a term used to categorize new scalable SQL database systems [21]. While the NoSQL movement brought about a monumental change in how organizations approached data and database technologies, it also brought about a major misconception. The NoSQL acronym was being taken to literally mean "No SQL" by some in the community who were quick to discard SQL in favor of performance [22]. However, the necessity of SQL and transactional

functionality was and still is imperative to organizations reliant on consistent data. Yet, the problems of traditional relational databases as described in Section 3.1.2 remained unresolved and the only available substitutes possessed great barriers of entry that put off many organizations. This demand for a modern relational database which possessed RDBMS functionality with the scalable capabilities of NoSQL would drive this development [23].

3.3.2 Types of NewSQL Systems

There are a growing number of NewSQL systems, with new ones popping up regularly. Each has its own underlying architectures and all support the relational data model and primarily use SQL to interact with applications, all being catered to work with existing applications originally designed around traditional RDBMS. Much like NoSQL systems, NewSQL systems can also be categorized based on their intended use: New Databases, New MySQL Storage Engines, and Transparent Clustering. It is worth noting that these are loose groupings that have yet to be defined in the same context that NoSQL systems have.

New Databases are simply defined as new database platforms. Built on entirely new platforms, these NewSQL systems are designed to operate as high-performance, large-scale databases. Most well-known NewSQL systems fall under this category, such as Google Spanner, VoltDB, and Splice Machine. New MySQL Storage Engines, also sometimes referred to as SQL Engines, are storage engines designed to improve the scalability of existing SQL systems, commonly MySQL due to its open-source nature. The final type of NewSQL system is a middleware layer that provides scalability to OLTP databases in the form of pluggable cluster transparency or transparent sharding functionalities [24].

3.3.3 Disadvantages

Unfortunately, NewSQL systems are still in their infancy. While many of them are already available in usable form, their use in large organizations pales in comparison to relational and NoSQL databases. Additionally, there is a lack of academic research for these systems, possibly because of their lack of large-scale use, which further perpetuates their lack of adoption.

This possible cycle means that RDBMS substitutes are failing to be acknowledged and implemented.

CHAPTER 4

OASIS: IDENTIFYING CORE ATTRIBUTES FOR RDBMS SUBSTITUTES

4.1 Determining the Core Attributes

In determining the attributes most important for a relational database substitute to possess, an infinite number of questions come to mind. Is it ACID compliant? How can it merge data from multiple tables? Can it be modified? Is it compatible with existing analytical software? While these questions are important to consider, many questions address specific needs of a user rather than general ones. In determining the core attributes a system must possess to serve as a RDBMS substitute requires filtering out the niche features and attributes to focus on the bare minimum; a baseline if you will. In looking through the feature sets of existing NoSQL and NewSQL system, a similar set of attributes emerged, leading to the development of OASIS.

4.2 What is OASIS?

OASIS (**O**pen-source, **A**CID, **S**QL, **I**nteroperable, **S**calable) is an acronym that represents a set of five core attributes that a relational database substitute is recommended to have. These attributes were chosen because they address three key issues relational database substitutes need to address to effectively take the place of traditional relational databases while also overcoming their shortcomings. These issues are as follows:

1. Support the relational data model, ACID, standard ANSI SQL, and existing personnel training.
2. Be compatible with existing and future RDBMS applications to reduce the need to migrate to another database system and supporting software.
3. Address the scalable performance shortcomings possessed by existing RDBMSs such as cost and heavy workloads [6].

Think of OASIS as a simplified evaluation guide that focuses first on determining whether a system can even be a RDBMS substitute. By identifying the presence of these attributes in a system, be it NoSQL, NewSQL, or so some other classification, a system can be referred to as being OASIS compliant and thus able to serve as a relational database substitute.

4.3 Defining the OASIS Attributes

4.3.1 Open-source

It must first be stated that being open-source is the only optional attribute found in OASIS. Should a system not be open-source, but possess the other four attributes, it may still qualify as being a RDBMS substitute. However, the benefits that come with being open-source greatly outweigh any perceived negatives [25]. The biggest negative is often the nature of being open to modification. Yet, as seen with successful projects such as Hadoop and MySQL, these projects are open-source while still having an official development team. The solution is to recognize and define the scope of outside contributors. This can range from performing bug reports and developing focused patches to fine-tuning and implementing features and creating forked projects based on the original.

This communal development greatly expands the reach and adoption of these systems, creating a vast community resource and user base. By being open-source, an organization can also implement the features a system lacks that are critical to their daily operations.

4.3.2 ACID

As discussed in Section 3.1.2, ACID is a set of properties which have come to define relational database transactions. As a result, a relational database substitute must also possess ACID properties in order to maintain the transactional and consistency requirements of traditional RDBMSs. There is no one way to go about achieving this, although it should be stressed that for the purpose of scalability, a lockless concurrency control is recommended (see Section 4.3.5 for more details).

4.3.3 SQL

The Structured Query Language (SQL) is the primary method a traditional relational database interacts with its data. Much like providing transactional guarantees, there are many ways which a RDBMS substitute can go about providing support for standard SQL. For example, Splice Machine utilizes Apache Derby, an open-source RDBMS, in its technology stack which processes SQL utilizing its embedded database engine. The major benefit of supporting standard SQL is that it enables organizations to tap into existing SQL-trained resources and technologies rather than have to invest heavily in training and other new technologies.

4.3.4 Interoperable

The primary goal of any substitute is to reliably stand in for the original. A relational database substitute needs to be able to not only replicate its key features, but it needs to be entirely interoperable with an organization's existing database systems and applications. This is achieved in three basic steps: ACID guarantees, standard SQL support, and compatibility with existing systems and applications. The former two are both necessary attributes of OASIS and have already been discussed at length, but what of the latter?

The answer to that lies in the Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC) drivers. These drivers are a major component in maintaining interoperability in that they exist to bridge applications and the DBMS [26, 27]. Most of the critical Business Intelligence applications on the market, such as Tableu, connect using these drivers. Given their importance, a RDBMS substitute would need to have these drivers available to allow for tools and software to interface with them, achieving interoperability. Furthermore, the ability to import and export data lends greatly to interoperability, allowing for easier migration of data to and from the system.

4.3.5 Scalability

The final attribute pertains to addressing the major flaw of traditional RDBMS. As was discussed in Section 3.1.2, the ever-increasing workloads being handled by traditional RDBMSs are overwhelming them. The sheer volume of incoming data means that these systems must be scaled not only to handle the currently influx, but projected future increases in data, too. This is an expensive endeavor which requires a cost-effective, scale out solution on commodity hardware. Additionally, many existing systems such as MySQL require massive amounts of specific coding to effectively shard, requiring additional time and manpower on top of hardware. To achieve this, several new and upcoming systems are making use of an HBase and Hadoop storage engine. Others, such as VoltDB, have achieved this by developing custom engines designed to achieve dynamic scaling.

CHAPTER 5

TEST CASES

In order to add provide a complete overview for the OASIS analysis, a series of benchmarks will be performed on Oracle 12C Enterprise Edition, VoltDB, and Splice Machine. These experiments will compare the performance of the systems with that of Oracle Database using the OLTP-Bench benchmarking software to attempt to perform a TPC-C benchmark. These experiments will test OASIS' functional attributes of ACID Compliance, SQL compatibility, Interoperability, and Scalability.

5.1 System Overview

All experiments will take place on the VMWare Player virtual machine software through the 64-bit version of CentOS 7 with access to 40 GB of hard disk space, 8 GB of memory, and 4 processor cores locked at 2.60GHz. CentOS7 was chosen for a few particular reasons. First, it is compatible with all three of the systems being evaluated in the test (although not officially, as is the case with Oracle). Second, its similarities to Red Hat Enterprise, a highly-used enterprise-level Linux operating system for server hosting, makes it a valuable platform for general compatibility. Finally, it is fairly low-impact, meaning more hardware resources can be dedicated to testing the systems.

While not optimal hardware for testing server loads, the specifications fall within the recommended settings for the standalone versions of VoltDB and Splice Machine, as well as the Standard Edition of Oracle Database 12c, which are designed for testing on a single, non-clustered computer. Unless otherwise specified, all software being used in these experiments are 64-bit versions.

5.2 OLTP-Bench

OLTP-Bench is an open-source benchmark framework developed using Java with the sole purpose of providing an extensible testbed for performing benchmarks on database management systems [43]. A key aspect of its extensibility is that any DBMS which has access to a JDBC driver has an ability to connect to the framework, which provides a means of testing certain aspects of interoperability with these systems.

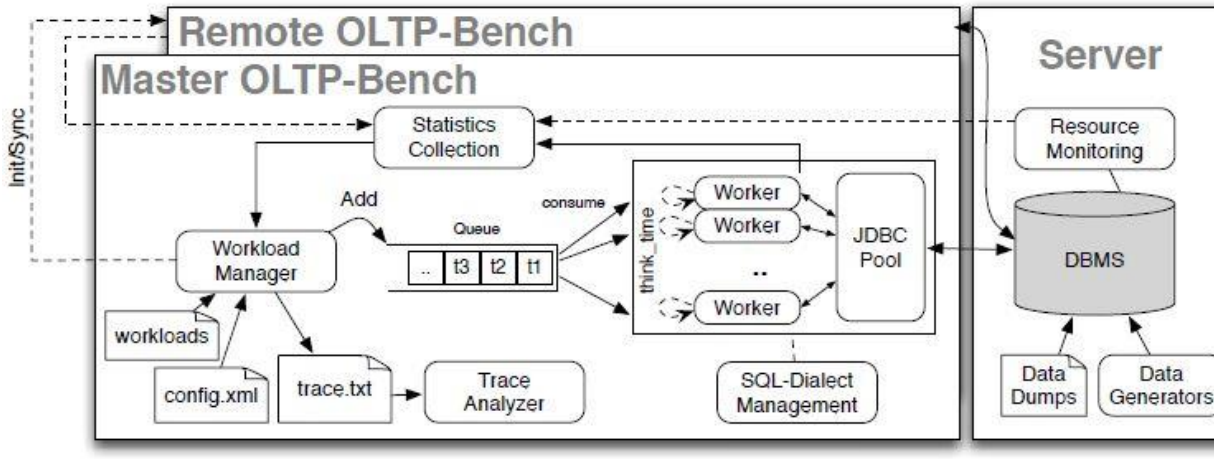

Figure 1: OLTP-Bench Architecture Diagram [43]

The goal of using OLTP-Bench is not to test the absolute performance of either VoltDB or Splice Machine. Instead, it will serve as a means to evaluate these systems and their ability to perform the kinds of ACID transactions that can be performed by traditional relational database systems. To that end, the specific benchmark which will be used for the purpose of these tests will be TPC-C, which is the current industry standard for evaluating the performance of OLTP systems [36].

The reason for this choice falls to the functionality required to perform OLTP, as seen in the following list [45]. It is immediately observable that some of these required functions align with the functionality described by the OASIS attributes of **A**CID, **S**QL, and **S**calability.

- ACID Transactions
- High concurrency
- Joins
- Stored procedures (SQL)
- DDL
- Constraints
- Foreign keys
- Sub-queries
- Views

5.2.1 TPC-C

TPC-C consists of nine tables and five procedures that simulate a data warehouse-centric order processing application. Its transactions are complex and write intensive, making transactional writes throughout much of the benchmark. Throughput for TPC-C is measured by the number of NewOrder transactions executed per second.

OLTP-Bench uses a modified version of TPC-C which omits the “thinking time” component of the benchmark [43]. This component is a simulated representation of the time spent by the worker to read the result of the transaction, as if a human is looking up and reading the data in order to “think” about what the data means. Because of this omission, each worker issues transactions without pausing, resulting in only a small number of parallel connections being needed to saturate the DBMS due to an increase in concurrent transactions. This makes it easier to test on less intensive hardware while getting similar results, although it also cannot be compared with the official published TPC-C benchmarks.

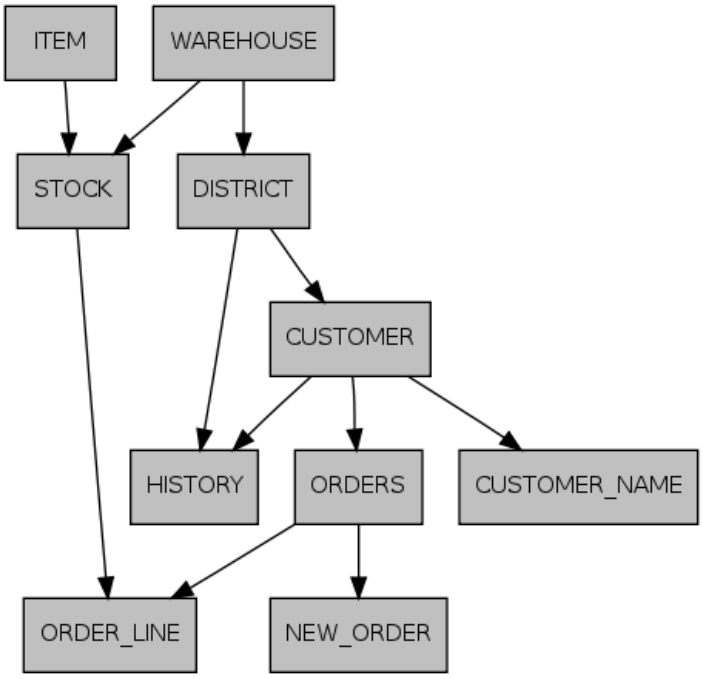

Figure 2: TPC-C Benchmark Schema [42]

The settings for the TPC-C benchmark will use the default values for time, rate, scale factor, and terminals provided for Oracle in the existing configuration files for OLTP-Bench. The ratio for rates will be the standard ratio described in the official TPC-C documentation [46]. These values are described in Table 1 below.

VARIABLE	**VALUE**
Time	60 Seconds
Rate	500
Scale Factor	1
Terminals	1
Weights	45,43,4,4,4

Table 1: Variable Values for TPC-C Testing

The variables are used to describe to OLTP-Bench how the specified benchmark is supposed to perform. Time is the length of time the benchmark is supposed to run in seconds. Rate represents the number of transactions. The Terminals attribute is the amount of concurrent users or "workers" making transactions. Scale Factor is a variable that increases the number of rows added to the tables by adjusting the cardinality of the warehouse table, creating larger user loads to provide more system stress. Any adjustment to the cardinality to the warehouse table affects the other tables in the benchmark, as their size is dependent on the number of warehouses. Finally, weights are the ratio of transactions that are run during the benchmark.

5.2.2 Code Modifications for OLTP-Bench

OLTP-Bench requires a few modifications in order to work with a new DBMS, assuming no major changes to the underlying code of OLTP-Bench are required to have compatible functionality. These changes include the addition of system-specific configuration XML files and DDL files, as well as changes to the dialects file for the target benchmark, and the inclusion of the database in the DatabaseType java file. Finally, the JDBC driver jar file for any new database system must be placed in the 'lib' folder of OLTP-Bench, which is referenced to in the configuration file.

5.3 Oracle Database

Oracle Database 12c Release 1, specifically version 12.1.0.2.0 – Enterprise Edition, will serve as the control to determine the validity of OLTP-Bench as a means of confirming whether or not OASIS attributes can be tested with it. As a fully-functional commercial relational database management system, it possesses the OASIS attributes of ACID, SQL, and Interoperability. It is for these reasons that it will serve as the base line for which the other systems will be compared in terms of functionality and performance.

5.3.1 Benchmark Evaluation

By default, OLTP-Bench supports performing benchmarks on Oracle systems. So, it should come as no surprise that performing the TPC-C benchmark is successful with minimal effort [43]. Utilizing the configuration file detailed in Figure A.2 and the DDL file described in Figure A.1, OLTP-Bench was able to successfully create and populate the database.

Using that same database, OLTP-Bench was then able to successfully perform the benchmark without error in repeated tests. The following tables illustrate a sample of the results obtained from executing the benchmark, bucketing the data in five second increments.

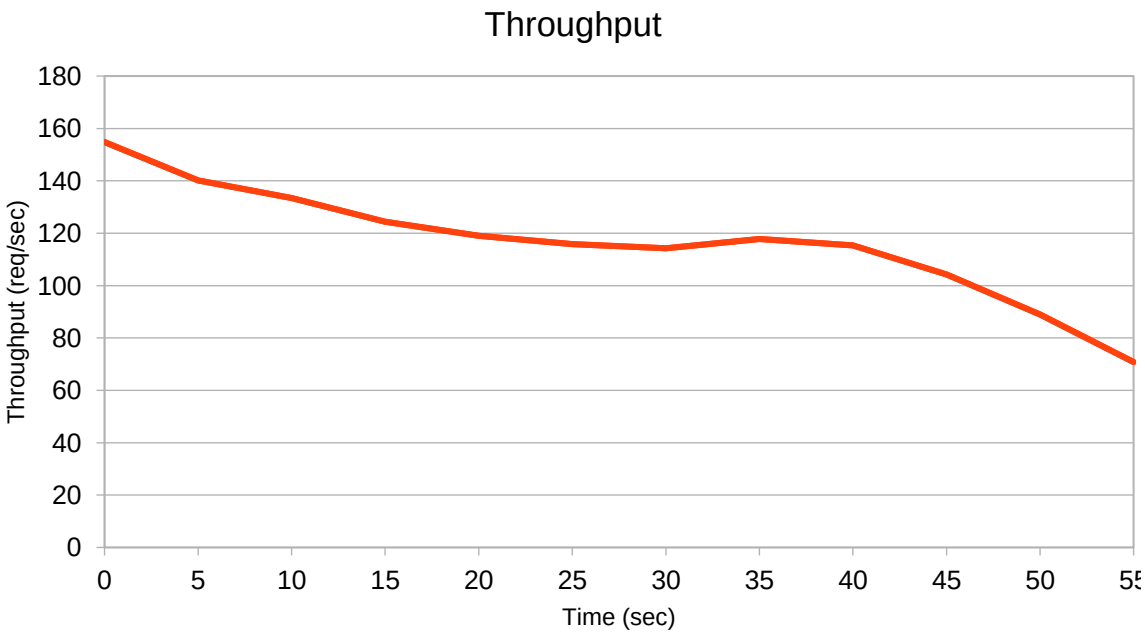

Figure 3: Oracle TPC-C Throughput Results

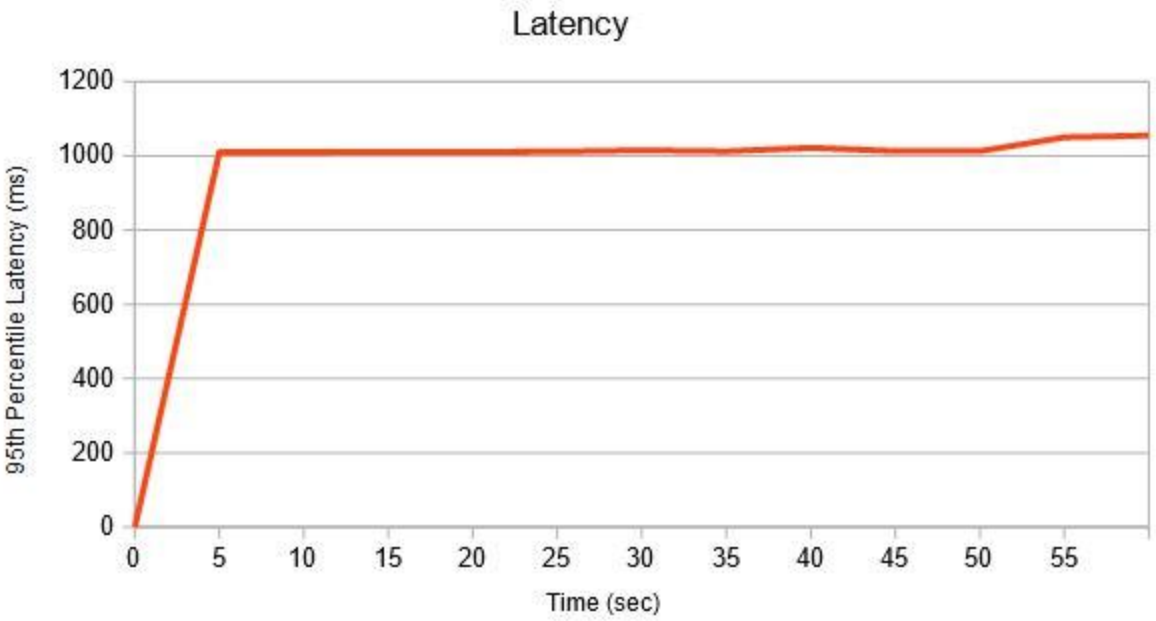

Figure 4: Oracle TPC-C Latency Results

5.3.2 Open-Source

Oracle RDBMS and the proprietary technologies which make up the system are not open source, including the Oracle SQL dialect (PL/SQL) compiler. However, this fact does not stop it from being one of the biggest RDBMS solutions on the market, as its quality and performance speak for themselves. The question becomes whether or not the rise of open-source and the need for custom solutions result in a change in policy in the future.

5.3.3 ACID

As can be surmised by the successful execution of TPC-C, a benchmark which is highly transactional and includes the use of rollbacks and intentional commits, it is clear Oracle exhibits the qualities required to perform ACID transactions.

5.3.4 SQL

OLTP-Bench creates all of their transactions through the use of predefined SQL procedures which are written in SQL-92. Oracle's specific dialect file, which is provided in the source code, is used to handle the Delivery and Order Status transactions, meaning that a majority of the benchmark is handled using standard SQL procedures.

5.3.5 Interoperable

The interoperability of Oracle is largely due to its JDBC driver, which allows for a plethora of programs to interface with it. As OLTP-Bench can connect with works with any DBMS system that supports SQL through JDBC [43], the successful connectivity with OLTP-Bench provides proof that there is interoperable capabilities of Oracle are present and functional.

5.3.6 Scalable

Scalability of RDBMS systems is generally their biggest flaw, which led database development down the very road it has taken. Oracle themselves have developed their own NoSQL system in order to provide a solution to the needs of Big Data and high-availability [44]. However, in order to test the scalability of Oracle, minor adjustments were made to the configuration file in order to increase the rate to 10,000, the number of terminals to 200, and the Scale Factor to 32. This exponential increase resulted in the results seen in Figures 5 and 6.

While extreme, the large increase in size and user count is to intentionally push the limits of the database. While there are limits to the hardware used in the benchmark, there's a stark difference in performance as the needs for a larger system with more concurrent user increases as compared to a small database with limited users.

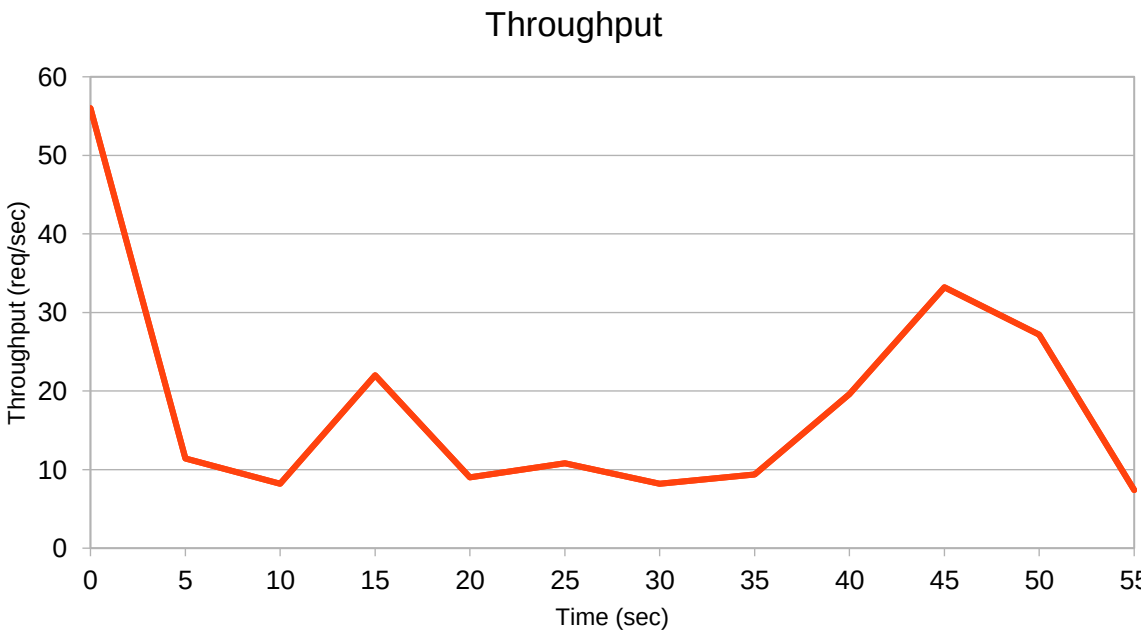

Figure 6: Oracle TPC-C Throughput with Large Dataset and Userbase

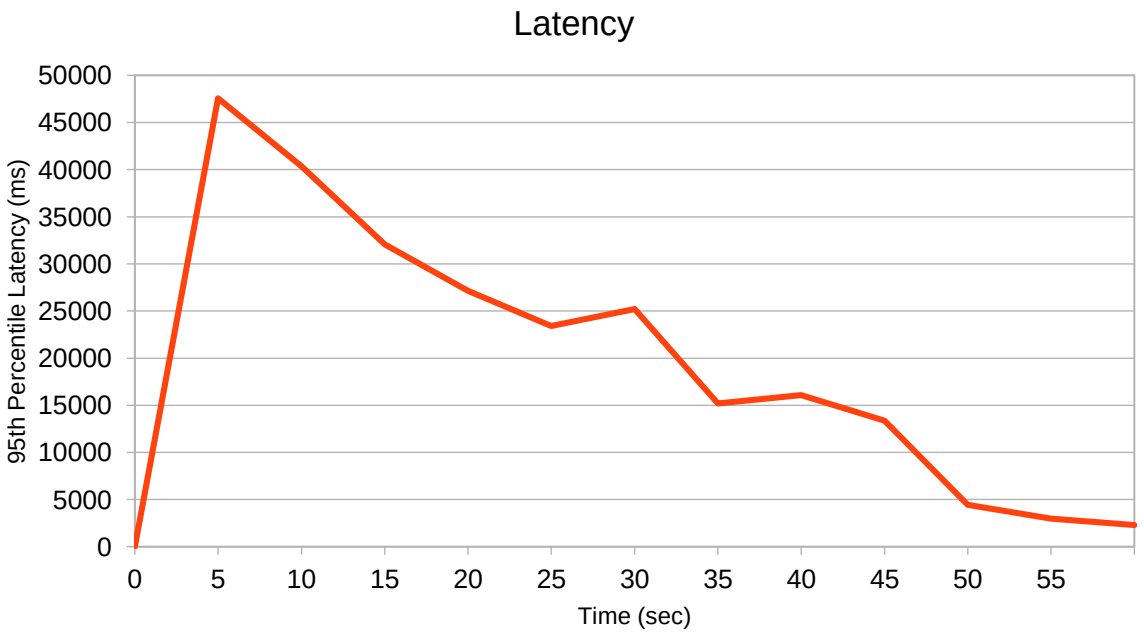

Figure 7: Orace TPC-C Latency with Large Dataset and Userbase

5.3.7 Results

Due to being a controlled experiment, it was expected that Oracle would be able to work with OLTP-Bench and TPC-C due to being a fully-functional RDBMS that had been confirmed to work with the software by the developers and further proven for this paper. Additionally, it proves that Oracle does possess some of the OASIS attributes, although clearly failing on the optional Open-Source attribute and struggling, if not failing, to demonstrate a clear case of providing scalability for very large data. This now provides a point of comparison which will then be used going forward for VoltDB and Splice Machine.

5.4 Volt DB

VoltDB is an open-source, commercial version of the H-Store distributed main memory database system developed as part of a collaboration between Brown University, Yale University, MIT, and Vertica Inc. with the goal of avoiding RDBMS scaling bottlenecks by building a system from the ground up that takes advantage of a shared-nothing distribution [28]. VoltDB was not designed to function well with large swaths of historical data sets, but rather data which needs to be streamed quickly. This makes it a system designed for processing first and foremost.

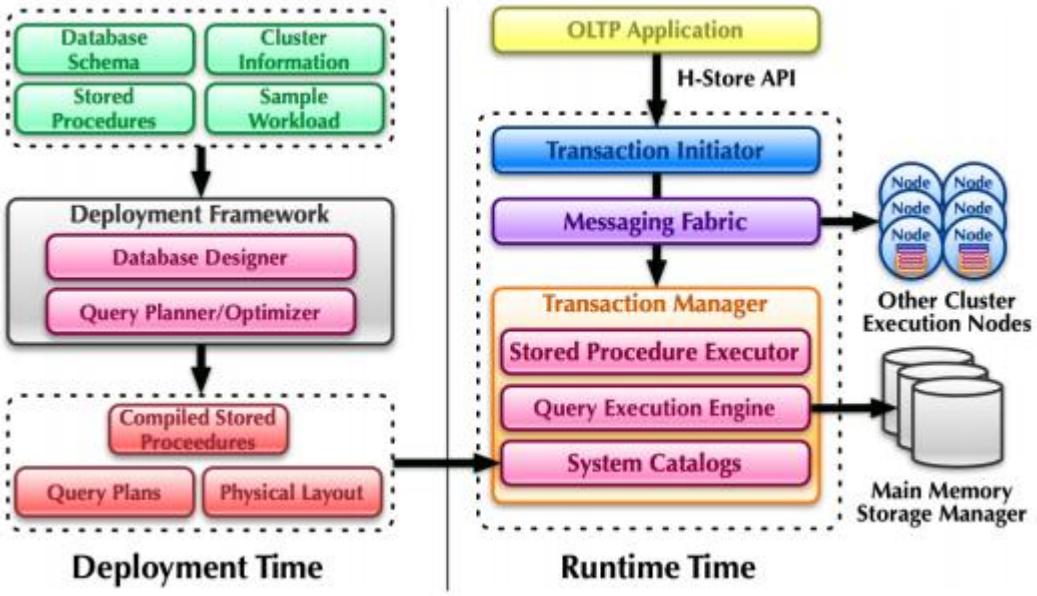

Figure 8: H-Store system architecture [28]

VoltDB requires that all persistence logic be coded as stored procedures. The obvious shortcomings with this approach are less code visibility and modularity, as well as higher maintenance needs to maximize effectiveness. However, it makes up for this by eliminating much of the overhead found in traditional relational databases due to being entirely in-memory and avoiding the use the locks.

5.4.1 Benchmark Evaluation

This experiment was performed using version 7.4 of the open-source code found on GitHub [30]. Utilizing OLTP-Bench, VoltDB was able to utilize the DDL file shown in Figure A.3 to create a database in preparation for performing the TPC-C benchmark. However, there were immediate problems when it came time to load data into the tables, a SQLFeatureNotSupportedException was encountered. Upon further research, the culprit would later be revealed to be the autocommit setting being turned off in the benchmark, as well as the manual commit commands, both of which VoltDB does not support [47]. Modifications were made to try and combat this, such as code to avoid reaching the commit-related commands which the database was "VOLT". While the modifications resolved the issue of the aforementioned error code, other issues would quickly take their place. It became clear that VoltDB and OLTP-Bench in their current incarnations simply will not work together. However, in order to have some kind of benchmark to work with, the only remaining available option is VoltDB's built-in TPC-C benchmark.

The reason VoltDB's own implementation of TPC-C was not initially used because of two important reasons. First, it features a code base different than that used in OLTP-Bench. While the two TPC-C implementations forego the worker wait time, the VoltDB version also does not include fulfillment of orders submitted to one warehouse, with items from another warehouse [48]. This change makes it a further derivative from the official benchmark. Second, it has been deprecated, with the code using deprecated libraries in order to allow for the benchmark to function. A modification to the VoltDB source code is required to allow for the benchmark to function [Figure B.1]. The results of this benchmark are as follows, which were run using the same settings used during the OLTP-Bench benchmark.

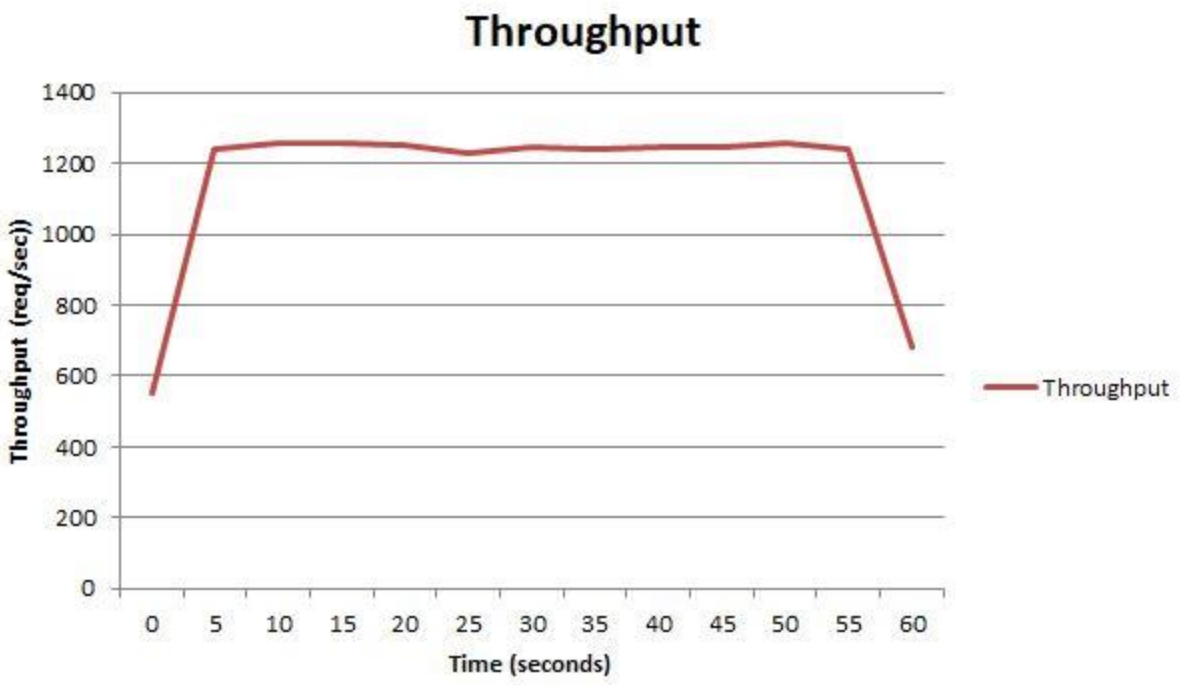

Figure 9: VoltDB Throughput Chart for TPC-C

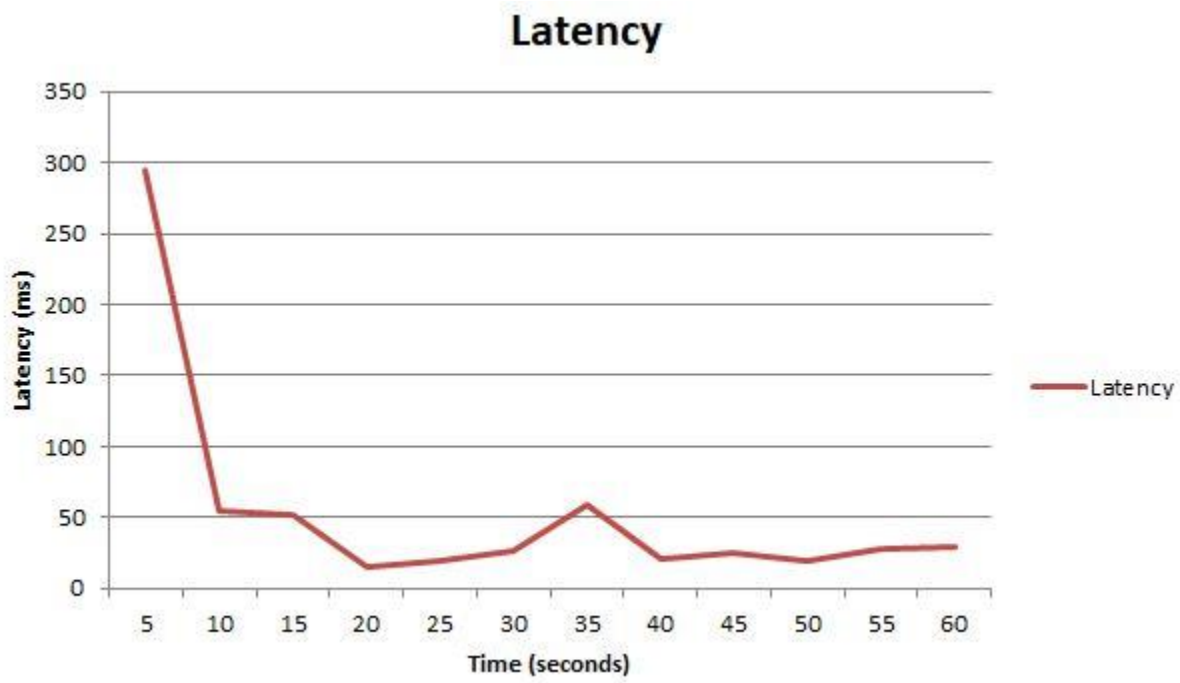

Figure 10: VoltDB Latency Chart for TPC-C

5.4.2 Open-Source

VoltDB is available in two editions. The first is an open-source Community Edition that is designed primarily for independent users, developers, and small companies. The source code for VoltDB is freely available on GitHub and must be manually built in order to run [30, 31]. While featuring much of the same core technologies that drive the system, there are very clear omissions such as the lack of transaction durability and disk persistency [32]. However, by being open to community modification, organizations are able to develop and make contributions to the source code of the software to fit their needs, which meets the standards OASIS defines for open-source. This opens the possibilities of dedicated software branches or eventually internalized features.

The second is the Enterprise Edition, which features greater support for enterprise-level solutions with elastic scalability, improved durability and replication, support for exporting, and around the clock customer support [32, 33]. While not important to the scope of this paper, the existence of an Enterprise Edition signifies that the system is enterprise ready rather than being in an alpha or beta state.

5.4.3 ACID

VoltDB provides serializable ACID consistency, which is the strongest level of ACID available [34]. This is done through the implementation of serializable isolation, in which all transactions are run one at a time without overlap. As each transaction is queued, it is appended to a “command log” which allows for the database to recover up to the last command logged. Because each command follows a serializable, and thus traceable, path, the exact changes made to the database can be redone. Any replicas are made durable before a transaction occurs [35].

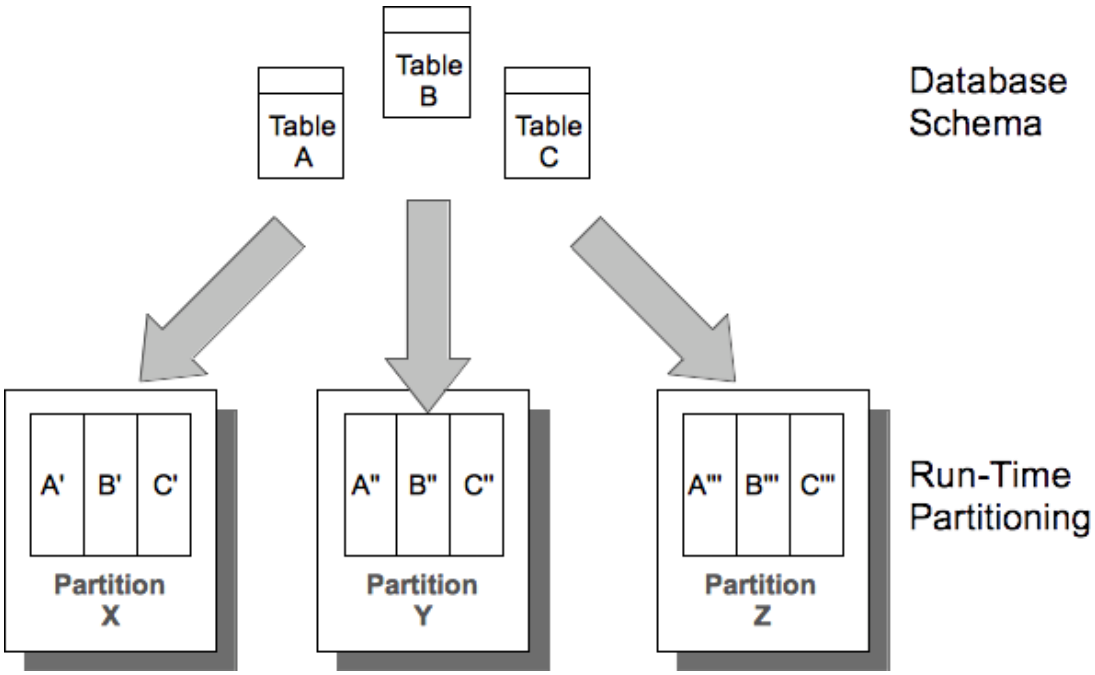

Figure 11: VoltDB's Table Partitioning [49]

While effective, it does have its drawbacks. VoltDB does not support external transaction control, as was demonstrated in the OLTP-Bench TPC-C benchmark where the commands to control how commits functioned resulted in errors during runtime. To maintain consistency, all calls to the database are essentially one ACID transaction each. Multi-statement transactions can be implemented as java stored procedures and are treated the same way, being seen by the system as a single ACID transaction.

5.4.4 SQL

VoltDB supports an OLTP-focused subset of SQL-99. SQL-based knowledge and techniques will carry over to VoltDB, but to maximize the capabilities of the system will require a major overhaul of pre-existing code [38]. We demonstrated in the benchmark that most general ADHOC SQL statements will work as expected, although the strict subset of SQL-99 and the limitations of the system to support certain common features such as commitment control will require an overhaul of existing code to accommodate this difference.

As the primary interaction model for transactional SQL statements in VoltDB are Java Stored Procedures (SQL/JRT), optimizing the performance of your SQL will require most existing code to be converted to that format. This results in additional work in converting an existing environment to be used with VoltDB. Additionally, the system performs optimally on queries designed to operate on single-partition stored procedures.

5.4.5 Interoperable

Interoperability's focus is on the primarily relies on supporting connectivity through JDBC/ODBC drivers. These drivers typically the means with which a database system imports or extracts data from a JDBC/ODBC compliant application. ODBC and JDBC connectivity are both supported by VoltDB [36]. Additionally, all ETL tools and programming languages which support JDBC or ODBC connectivity can also inter-operate with these systems. The issue, as demonstrated in our OLTP-Bench benchmark is that certain functionality which is commonly found in software which connects to RDBMS's, such as external commitment control, will not be interoperable.

5.4.6 Scalable

VoltDB's architecture is designed as a shared-nothing, cluster-native database built around high availability. As a cluster starts to reach its limits in terms of storage or throughput, VoltDB is able to elastically increase the number of nodes active in order to prevent downtime [35]. Combine this with the in-memory, high-performance processing, and the system is able to adjust to high demand as necessary. Once again, partitioning takes a key role in allowing for multiple transactions to occur simultaneously, even on a single thread, preventing the need for locking and increased processing overhead. Furthermore, the system is able to determine which small tables are read-only and replicate them across all partitions to allow for the partitions to remain as separated as possible, which helps maintain their individual performances.

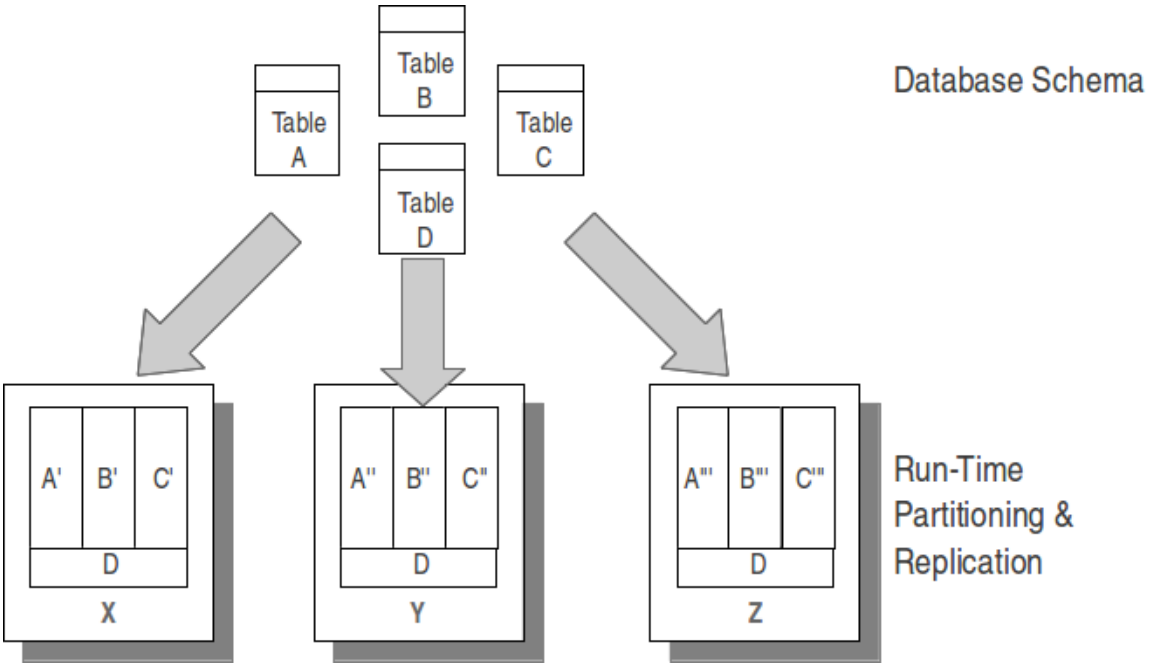

Figure 12 VoltDB in Replicating Small, Unchanging Data[49]

There are limits to VoltDB. According to the developers, 100 or less node clusters are seen as the optimal limit for VoltDB for the purpose of maintaining a solid balance between performance, fault-tolerance and manageability [39].

In order to present a show of the scalability capabilities, its own benchmark is run again using the same expanded test attributes used in the second round of Oracle benchmarks. Where Oracle struggled greatly on the same hardware to maintain throughput, the figures below show that VoltDB is experiencing quite the contrary, maintaining a steady stream of throughput for the entire duration of the benchmark

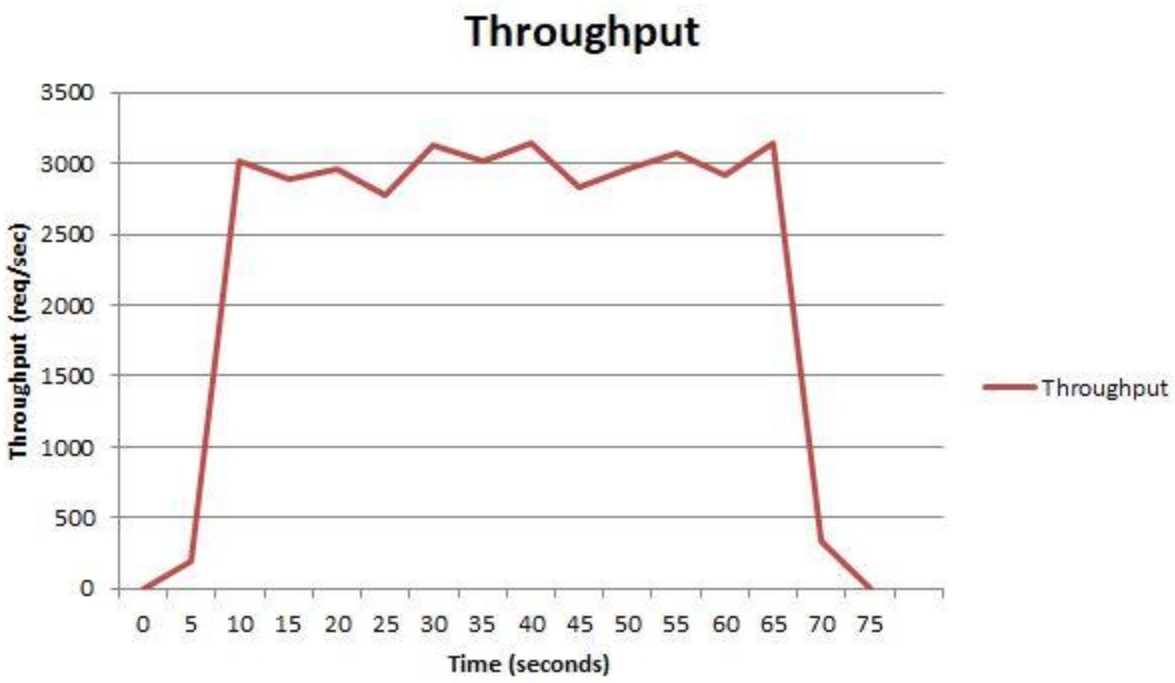

Figure 13: VoltDB Throughput Results with Large Data Set

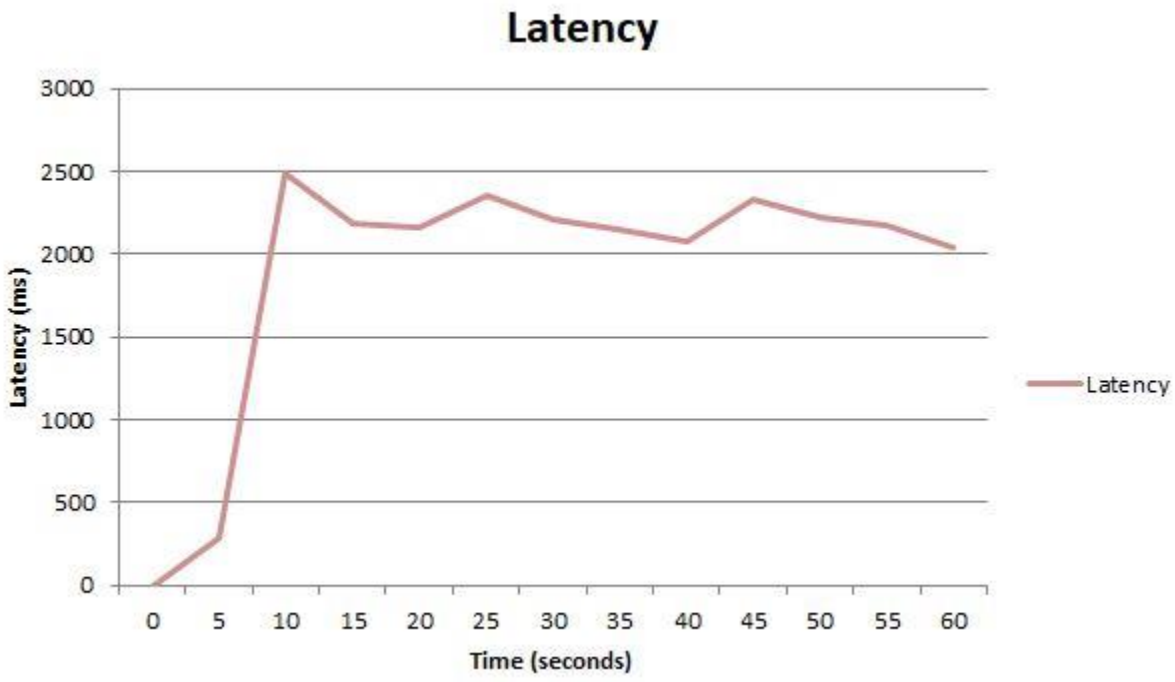

Figure 14: VoltDB Latency Results with Large Data Set

5.4.7 Results

VoltDB clearly demonstrates the attributes of Open-source and ACID, the latter of which is built around each call to a SQL Procedure, which the complete procedure is treated as a single transaction. While SQL compliant, certain key capabilities are not compatible such as not supporting external transaction control, which automatically makes it incompatible with applications which want to control when and how the system makes commits. Possessing a JDBC driver allows VoltDB to potentially interface with any application, leaving room for application development. However, the other lacking OASIS attributes means true interoperability is unlikely unless it is a specially-developed software rather than existing RDBMS software. VoltDB, based on the results of its own TPC-C benchmark, is shown to possess incredible throughput, easily processing 58x more requests at their peak.

Ultimately, VoltDB is a highly-capable and highly-scalable system, but would require an extensive overhaul of existing RDBMS infrastructure if it were to be used as a replacement.

5.5 Splice Machine

Splice Machine is an open-source, modern RDBMS which is powered by a three-part proven technology stack comprised of Apache HBase/Hadoop, Apache Spark, and Apache Derby [29]. By combining the storage of the Hadoop Distributed File System (HDFS), HBase's scaling and sharding technologies, the in-memory processing abilities of Spark, and the full-featured SQL database of Derby, Splice Machine is designed specifically to fulfill the need for a scale-out, high throughput SQL database.

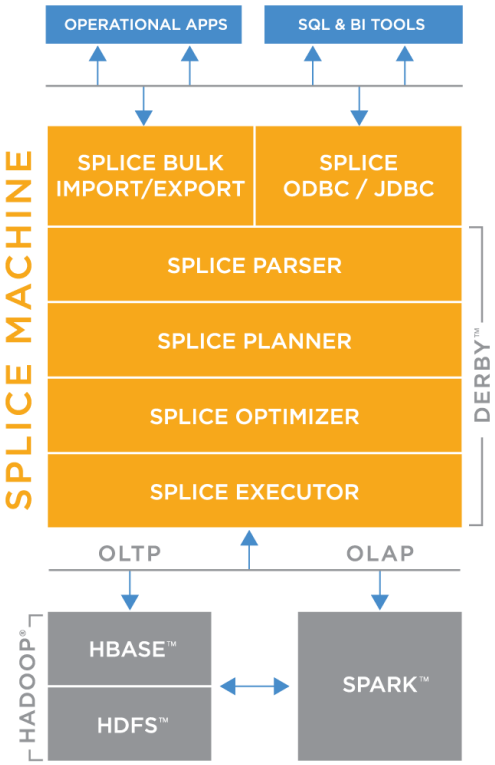

Figure 15: Splice Machine system architecture [29]

For the purposes of this paper, we will be using the Standalone Community Edition.

5.5.1 Benchmark Evaluation

During the OLTP-Bench TPC-C benchmark, a plethora of unexpected issues arose, which were contrary to the initial expectations had for the system, which were that the system would require only a few necessary tweaks to function properly. This was further reinforced by the fact that the company at one time used an older, modified version of OLTP-Bench to perform TPC-C tests on [50].

Unfortunately, OLTP-Bench's TPC-C benchmark was unable to run in its entirety on Splice Machine. During the database loading portion of the benchmark, a consistent error relating to the setNull Prepared Statement while loading Orders. This resulted in a cascading effect where the loading process was unable to complete successfully, resulting in an incomplete data set. Attempting to perform the benchmark would then result in errors as it failed to find data, some of which would exist in the database. Further research revealed nothing mentioned in the Splice Machine documentation pertaining to a setNull prepared statement, leading to the strong possibility that it simply is not supported or implemented as of this time. However, the findings of this are negative, as an incomplete benchmark fails to provide all of the information sought after in examining its OASIS attributes.

5.5.2 Open-Source

Splice Machine is available in two editions. The first is an open-source Community Edition that is designed primarily for independent users, developers, and small companies. The source code Splice Machine, much like VoltDB, is freely available on GitHub [31]. This open-source approach meets the criteria defined by OASIS. The second is the Enterprise Edition, which provides additional features not found in the open-source, Community Edition. This includes, but is not limited to - LDAP integration, PL/SQL Support, and Encryption Support [33].

5.5.3 ACID

Splice Machine describes itself as a full-featured Hadoop RDBMS. Through the use of MVCC (Multi-Version Concurrency Control), the system is able to utilize a lockless snapshot to create a new version of a record per each update [36]. While this is a storage-intensive approach, it allows for a high volume of concurrent users to execute their transactions without the risk of deadlocking while still maintaining ACID consistency.

This was demonstrated during the benchmark where attempts at loading and removing data would result in data conflicts based on dependencies. These conflicts would trigger a system rollback, demonstrating that the system maintains ACID transactions by means of atomicity.

5.5.4 SQL

Splice Machine also supports standard ANSI SQL, limited to the same subset of SQL syntax as Apache Derby, which originated on IBM's DB2 SQL system. By using Apache Derby as its database engine, on its technology stack to natively process SQL statements and queries before then converting it into a form useable by HBase/Hadoop [36]. All observations observed during the benchmark signal that the only notable limitations of SQL are the same ones imposed by Apache Derby.

Splice Machine also supports native PL/SQL, the proprietary server-based procedural extension of SQL used on Oracle database systems [37]. However, this is limited to the Enterprise Edition only, making it unavailable in our experiments.

5.5.5 Interoperable

Much like Oracle and VoltDB, Splice Machine provides a JDBC driver for interfacing with its data [36]. By successfully interfacing with OLTP-Bench directly through its JDBC driver, Splice Machine's interoperability is demonstrated to be possible.

5.5.6 Scalable

Splice Machine's scalability comes from Apache HBase, which serves as the system's storage layer [40]. In addition to HBase, each cluster node is embedded with Derby, namely its parser and a modified version of its planner, optimizer, and executor, and Spark computation engine. Each HBase co-processor, is then able to embed the system into each HBase region, providing massive parallel performance by giving each data shard computational ability rather than defaulting the processing to a single master server [29]. As mentioned prior, because the storage engine is built around HBase/Hadoop, it is extensible to commodity hardware.

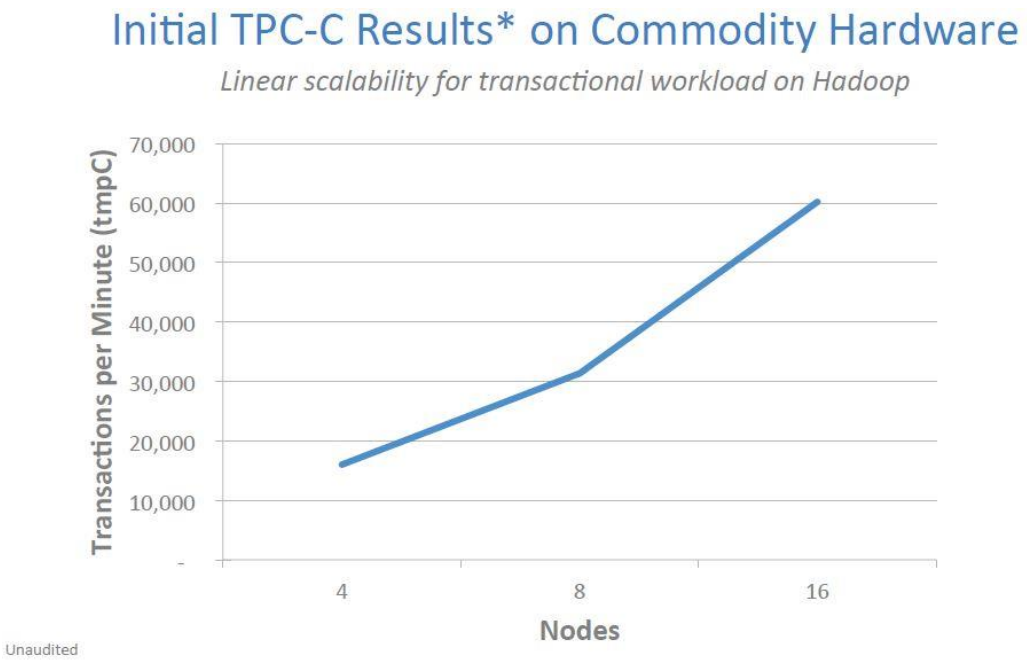

Figure 16: Splice Machine's TPC-C Evaluation Results

5.5.6 Conclusions

Splice Machine leverages the power and familiarity of numerous open-source projects to create a system which is not only familiar, but tried and true. Sadly, the OLTP-Bench failed to perfectly verify whether or not all of those claims are true as the results were inconclusive. What was revealed, however, is that Splice Machine meets four of the five OASIS attributes of open-source, ACID compliance, SQL compatibility, and interoperability. Scalability remains in question until the results published by Splice Machine can be verified by external sources.

CHAPTER 6

CONCLUSION

In this paper, a set of necessary qualities which must be possessed by a system designed to replace a RDBMS were defined in OASIS. These attributes were chosen because they meet the core functional capabilities of relational databases, such as ACID transactions, SQL compliance, and Interoperability through the use of JDBC/ODBC drivers, while also addressing the shortcomings plagued by existing RDBMS systems including the benefits of providing open-source components and a focus on scalability. In order to verify the validity of OASIS, an open-source benchmarking framework, OLTP-Bench, was chosen as a means to perform TPC-C benchmarks on two relatively new NewSQL systems, VoltDB and Splice Machine.

By using Oracle as a baseline, these two systems were run against the benchmark to see how compatible they were using the same base code which has been shown to work across all popular RDBMS's. In both instances, the results were inconclusive as the benchmarks ultimately failed to complete. However, their failures provided useful information about the systems which was used to determine if the attributes identified by OASIS were present and in what capacity. This information can provide a foundation for further research and development into examining the ways in which NoSQL and NewSQL can further serve as a substitute for relational database systems.

REFERENCES

[1] E. F. Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (June 1970), 377-387. DOI=10.1145/362384.362685 http://doi.acm.org/10.1145/362384.362685

[2] Hyde, J. (2014, October 20). *SQL On Everything, In Memory*. Lecture. Retrieved February 10, 2016, from http://www.slideshare.net/julianhyde/calcite-stratany2014

[3] Database. (n.d.). Retrieved February 10, 2016, from www.en.wikipedia.org/wiki/Database

[4] CODASYL. (n.d.). Retrieved February 10, 2016, from www.en.wikipedia.org/wiki/CODASYL

[5] Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C., & Byers, A. H. (2011, May). Big data: The next frontier for innovation, competition, and productivity. Retrieved February 10, 2016. <http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation >

[6] Leavitt, Neal. "Will NoSQL Databases Live Up to Their Promise?" *Computing in Science & Engineering* (2010): 12-14. Leavcom, Feb. 2010. Web. 2 Mar. 2016. <http://www.leavcom.com/pdf/NoSQL.pdf>.

[7] Dravenport, Thomas H., and Jill Dyché. *Big Data in Big Companies* (2013): n. pag. SAS Institute Inc. Web. <http://www.sas.com/content/dam/SAS/en_us/doc/whitepaper2/bigdata-bigcompanies-106461.pdf>.

[8] A Timeline of Database History. (n.d.). Retrieved February 10, 2016, from http://quickbase.intuit.com/articles/timeline-of-database-history

[9] "DB-Engines Ranking." - *Popularity Ranking of Relational DBMS*. SolidIT, n.d. Web. 16 June 2016. <http://db-engines.com/en/ranking/relational+dbms>.

[10] "DB-Engines Ranking." - *Popularity Ranking of Database Management Systems*. SolidIT, n.d. Web. 16 June 2016. <http://db-engines.com/en/ranking>.

[11] J. Gray. The Transaction Concept: Virtues and Limitations. 1981. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.59.5051 [Accessed 29 September 2015].

[12] Gray, J. N., R. A. Lorie, and G. R. Putzolu. "Granularity of Locks in a Shared Data Base." *Proceedings of the 1st International Conference on Very Large Data Bases - VLDB '75* (1975): n. pag. Web. 21 June 2016.

[13] Haerder, Theo, and Andreas Reuter. "Principles of Transaction-oriented Database Recovery." *ACM Computing Surveys* 15.4 (1983): 287-317. IBM Research Labratory. Web. 8 Sept. 2016. <http://dl.acm.org/citation.cfm?id=291>.

[14] Daytona Business Journal. "Wal-Mart More than Doubles the Size of Its NCR-based Data Warehouse." Dayton Business Journal, 17 Aug. 1999. Web. 11 Apr. 2015. <http://www.bizjournals.com/dayton/stories/1999/08/16/daily6.html>.

[15] S. K. Rahimi and F. S. Haug. *Distributed Database Management Systems: A Practical Approach*. JohnWiley & Sons, Inc, 2010.

[16] Armando Fox and Eric A. Brewer. 1999. Harvest, Yield, and Scalable Tolerant Systems. In *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems* (HOTOS '99). IEEE Computer Society, Washington, DC, USA, 174-.

[17] E. A. Brewer. Towards robust distributed systems. In *PODC*, page 7, 2000. https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[18] Rick Cattell. 2011. Scalable SQL and NoSQL data stores. *SIGMOD Rec.* 39, 4 (May 2011), 12-27. DOI: http://dx.doi.org/10.1145/1978915.1978919

[19] Kyle, Bruce D. "Getting Acquainted with NoSQL on Windows Azure." *Microsoft Developer Network*. Microsoft, 5 Apr. 2012. Web. 9 Sept. 2016. <https://blogs.msdn.microsoft.com/usisvde/2012/04/05/getting-acquainted-with-nosql-on-windows-azure/>.

[20] Aslett, Matthew. "Neither Fish nor Fowl: The Rise of Multi-model Databases." *Too Much Information - The 451 Take on Information Management*. The 451 Group, 8 Feb. 2013. Web. 12 July 2016. <https://blogs.the451group.com/information_management/2013/02/08/neither-fish-nor-fowl/>.

[21] Anslett, Matt. "NoSQL, NewSQL and Beyond: The Drivers and Use Cases for Database Alternatives." *NoSQL, NewSQL and Beyond: The Drivers and Use Cases for Database Alternatives*. The 451 Group, 15 Apr. 2011. Web. 12 July 2016. <https://451research.com/report-long?icid=1651>.

[22] Menegaz, Gery. "The NoSQL Community Threw out the Baby with the Bath Water | ZDNet." *ZDNet*. ZDNet, 28 Jan. 2013. Web. 14 July 2016. <http://www.zdnet.com/article/the-nosql-community-threw-out-the-baby-with-the-bath-water/>.

[23] Aslett, Matthew. "What We Talk about When We Talk about NewSQL." *Too Much Information - The 451 Take on Information Management*. The 451 Group, 6 Apr. 2011. Web. 6 Aug. 2016. <https://blogs.the451group.com/information_management/2011/04/06/what-we-talk-about-when-we-talk-about-newsql/>.

[24] Venkatesh, Prasanna. "NewSQL -- The New Way to Handle Big Data - Open Source For You." *Open Source For You*. Open Source For You, 30 Jan. 2012. Web. 8 Mar. 2016. <http://opensourceforu.com/2012/01/newsql-handle-big-data/>.

[25] VoltDB. "Open Source - Why Is It Important for VoltDB to Be Open Source?" *VoltDB - Open Source*. VoltDB, n.d. Web. 25 Sept. 2016. <https://www.voltdb.com/products/open-source>.

[26] Oracle. "JDBC Overview." *JDBC Overview*. Oracle, n.d. Web. 21 Sept. 2016. <http://www.oracle.com/technetwork/java/overview-141217.html>.

[27] Microsoft. "ODBC--Open Database Connectivity Overview." *Microsoft Support*. Microsoft, n.d. Web. 21 Sept. 2016. <https://support.microsoft.com/en-us/kb/110093>.

[28] Robert Kallman, Hideaki Kimura, Jonathan Natkins, *et al*. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.* 1, 2 (August 2008), 1496-1499. DOI=http://dx.doi.org/10.14778/1454159.1454211

[29] Splice Machine. "How It Works - Splice Machine." *Splice Machine How It Works Comments*. Splice Machine, n.d. Web. 1 Oct. 2016. <http://www.splicemachine.com/product/how-it-works/>.

[30] https://github.com/VoltDB/voltdb

[31] https://github.com/splicemachine/spliceengine

[32] VoltDB. "VoltDB Open Source & Enterprise Editions." *VoltDB Open Source & Enterprise Editions*. VoltDB, n.d. Web. 1 Oct. 2016. <https://www.voltdb.com/editions>.

[33] Splice Machine. "Pricing - Splice Machine." *Splice Machine Pricing Comments*. Splice Machine, n.d. Web. 1 Oct. 2016. <http://www.splicemachine.com/product/pricing/>.

[34] Hugg, John. "ACID: How to Screw It Up!" *ACID: How to Screw It Up!* VoltDB, 25 Feb. 2016. Web. 16 Oct. 2016. <https://www.voltdb.com/blog/acid-how-to-screw-it-up>.

[35] VoltDB. "How VoltDB Does Transactions." VoltDB, n.d. Web. 5 Nov. 2016. <https://cdn2.hubspot.net/hubfs/2180197/content/techincal_notes/lv_technical_notes/lv-technical-note-how-voltdb-does-transactions.pdf?t=1478549622517>. Technical Note

[36] Splice Machine. "White Paper: The First Hybrid, In-Memory RDBMS Powered by Hadoop and Spark." *The Open Source RDBMS Powered by Hadoop and Spark*. Splice Machine, n.d. Web. 28 Sept. 2016. <http://info.splicemachine.com/White-Paper.html>.

[37] Database Trends and Applications. "Splice Machine Introduces PL/SQL Support for Oracle-to-Hadoop Migration." *Database Trends and Applications*. Database Trends and Applications, 27 Sept. 2016. Web. 19 Oct. 2016. <http://www.dbta.com/Editorial/News-Flashes/Splice-Machine-Introduces-PL-SQL-Support-in-Support-Oracle-to-Hadoop-Migration-113780.aspx>.

[38] VoltDB. "Welcome to VoltDB - A Tutorial." (n.d.): n. pag. Web. 28 Oct. 2016. <http://downloads.voltdb.com/documentation/tutorial.pdf>.

[39] VoltDB. "Clustering & Scalability." *Clustering & Scalability*. VoltDB, n.d. Web. 7 Nov. 2016. <https://www.voltdb.com/products/clustering-scalability>.

[40] IBM. (n.d.). What is HBase? Retrieved January 30, 2016. <http://www-01.ibm.com/software/data/infosphere/hadoop/hbase/>

[41] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. 1997. Cluster-based scalable network services. *SIGOPS Oper. Syst. Rev.* 31, 5 (October 1997), 78-91. DOI=http://dx.doi.org/10.1145/269005.266662

[42] “Documentation > Deployment > Supported Benchmarks.” *HStore*, hstore.cs.brown.edu/documentation/deployment/benchmarks/.

[43] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: an extensible testbed for benchmarking relational databases. Proc. VLDB Endow. 7, 4 (December 2013), 277-288. DOI=http://dx.doi.org/10.14778/2732240.2732246

[44] “Oracle NoSQL Database.” Oracle NoSQL Database Technical Overview, Oracle, www.oracle.com/technetwork/database/database-technologies/nosqldb/overview/index.html.

[45] Zweben, Monte, and John Leach. “OLTP on Hadoop: Reviewing the First Hadoop--Based TPC--C Benchmarks.” Splice Machine, *Strata + Hadoop World*, 15 Sept. 2015.

[46] “TPC BENCHMARK C.” Feb. 2010, pp. 1–132., www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[47] *SQL Problems When Trying to Execute TPC-B. - VoltDB Forums*, forum.voltdb.com/forum/voltdb-discussions/other/756-sql-problems-when-trying-to-execute-tpc-b.

[48] *VoltDB Tpc-c-like Benchmark Comparison-Benchmark Description - VoltDB Forums*, forum.voltdb.com/forum/voltdb-discussions/other/30-voltdb-tpc-c-like-benchmark-comparison-benchmark-description.

[49] “How VoltDB Works.” *1.3. How VoltDB Works*, VoltDB, docs.voltdb.com/UsingVoltDB/IntroHowVoltDBWorks.php.

[50] Splicemachine. “Splicemachine/Tpc-Benchmark-Tools.” *GitHub*, Splice Machine, 7 Nov. 2016, github.com/splicemachine/tpc-benchmark-tools.

APPENDIX A - OLTP-BENCH COMPONENTS

```
BEGIN EXECUTE IMMEDIATE 'DROP TABLE ORDER_LINE CASCADE CONSTRAINTS PURGE';
EXCEPTION WHEN OTHERS THEN IF SQLCODE != -942 THEN RAISE; END IF; END;;
BEGIN EXECUTE IMMEDIATE 'DROP TABLE STOCK CASCADE CONSTRAINTS PURGE'; EXCEPTION
WHEN OTHERS THEN IF SQLCODE != -942 THEN RAISE; END IF; END;;
BEGIN EXECUTE IMMEDIATE 'DROP TABLE ITEM CASCADE CONSTRAINTS PURGE'; EXCEPTION
WHEN OTHERS THEN IF SQLCODE != -942 THEN RAISE; END IF; END;;
BEGIN EXECUTE IMMEDIATE 'DROP TABLE NEW_ORDER CASCADE CONSTRAINTS PURGE';
EXCEPTION WHEN OTHERS THEN IF SQLCODE != -942 THEN RAISE; END IF; END;;
BEGIN EXECUTE IMMEDIATE 'DROP TABLE OORDER CASCADE CONSTRAINTS PURGE'; EXCEPTION
WHEN OTHERS THEN IF SQLCODE != -942 THEN RAISE; END IF; END;;
BEGIN EXECUTE IMMEDIATE 'DROP TABLE HISTORY CASCADE CONSTRAINTS PURGE';
EXCEPTION WHEN OTHERS THEN IF SQLCODE != -942 THEN RAISE; END IF; END;;
BEGIN EXECUTE IMMEDIATE 'DROP TABLE CUSTOMER CASCADE CONSTRAINTS PURGE';
EXCEPTION WHEN OTHERS THEN IF SQLCODE != -942 THEN RAISE; END IF; END;;
BEGIN EXECUTE IMMEDIATE 'DROP TABLE DISTRICT CASCADE CONSTRAINTS PURGE';
EXCEPTION WHEN OTHERS THEN IF SQLCODE != -942 THEN RAISE; END IF; END;;
BEGIN EXECUTE IMMEDIATE 'DROP TABLE WAREHOUSE CASCADE CONSTRAINTS PURGE';
EXCEPTION WHEN OTHERS THEN IF SQLCODE != -942 THEN RAISE; END IF; END;;
CREATE TABLE CUSTOMER (
  C_W_ID NUMBER NOT NULL,
  C_D_ID NUMBER NOT NULL,
  C_ID NUMBER NOT NULL,
  C_DISCOUNT NUMBER(4,4) NOT NULL,
  C_CREDIT CHAR(2) NOT NULL,
  C_LAST VARCHAR2(16) NOT NULL,
  C_FIRST VARCHAR2(16) NOT NULL,
  C_CREDIT_LIM NUMBER(12,2) NOT NULL,
  C_BALANCE NUMBER(12,2) NOT NULL,
  C_YTD_PAYMENT FLOAT NOT NULL,
  C_PAYMENT_CNT NUMBER NOT NULL,
  C_DELIVERY_CNT NUMBER NOT NULL,
  C_STREET_1 VARCHAR2(20) NOT NULL,
  C_STREET_2 VARCHAR2(20) NOT NULL,
  C_CITY VARCHAR2(20) NOT NULL,
  C_STATE CHAR(2) NOT NULL,
```

```
  C_ZIP CHAR(9) NOT NULL,
  C_PHONE CHAR(16) NOT NULL,
  C_SINCE DATE NOT NULL,
  C_MIDDLE CHAR(2) NOT NULL,
  C_DATA VARCHAR2(500) NOT NULL,
  PRIMARY KEY (C_W_ID,C_D_ID,C_ID)
);
CREATE INDEX IDX_CUSTOMER_NAME ON CUSTOMER (C_W_ID,C_D_ID,C_LAST,C_FIRST);

CREATE TABLE DISTRICT (
  D_W_ID NUMBER NOT NULL,
  D_ID NUMBER NOT NULL,
  D_YTD NUMBER(12,2) NOT NULL,
  D_TAX NUMBER(4,4) NOT NULL,
  D_NEXT_O_ID NUMBER NOT NULL,
  D_NAME VARCHAR2(10) NOT NULL,
  D_STREET_1 VARCHAR2(20) NOT NULL,
  D_STREET_2 VARCHAR2(20) NOT NULL,
  D_CITY VARCHAR2(20) NOT NULL,
  D_STATE CHAR(2) NOT NULL,
  D_ZIP CHAR(9) NOT NULL,
  PRIMARY KEY (D_W_ID,D_ID)
);

CREATE TABLE HISTORY (
  H_C_ID NUMBER NOT NULL,
  H_C_D_ID NUMBER NOT NULL,
  H_C_W_ID NUMBER NOT NULL,
  H_D_ID NUMBER NOT NULL,
  H_W_ID NUMBER NOT NULL,
  H_DATE DATE NOT NULL,
  H_AMOUNT NUMBER(6,2) NOT NULL,
  H_DATA VARCHAR2(24) NOT NULL
);

CREATE TABLE ITEM (
  I_ID NUMBER NOT NULL,
```

```
  I_NAME VARCHAR2(24) NOT NULL,
  I_PRICE NUMBER(5,2) NOT NULL,
  I_DATA VARCHAR2(50) NOT NULL,
  I_IM_ID NUMBER NOT NULL,
  PRIMARY KEY (I_ID)
);

CREATE TABLE NEW_ORDER (
  NO_W_ID NUMBER NOT NULL,
  NO_D_ID NUMBER NOT NULL,
  NO_O_ID NUMBER NOT NULL,
  PRIMARY KEY (NO_W_ID,NO_D_ID,NO_O_ID)
);

CREATE TABLE OORDER (
  O_W_ID NUMBER NOT NULL,
  O_D_ID NUMBER NOT NULL,
  O_ID NUMBER NOT NULL,
  O_C_ID NUMBER NOT NULL,
  O_CARRIER_ID NUMBER DEFAULT NULL,
  O_OL_CNT NUMBER(2,0) NOT NULL,
  O_ALL_LOCAL NUMBER(1,0) NOT NULL,
  O_ENTRY_D DATE NOT NULL,
  PRIMARY KEY (O_W_ID,O_D_ID,O_ID),
  UNIQUE (O_W_ID,O_D_ID,O_C_ID,O_ID)
);

CREATE TABLE ORDER_LINE (
  OL_W_ID NUMBER NOT NULL,
  OL_D_ID NUMBER NOT NULL,
  OL_O_ID NUMBER NOT NULL,
  OL_NUMBER NUMBER NOT NULL,
  OL_I_ID NUMBER NOT NULL,
  OL_DELIVERY_D DATE,
  OL_AMOUNT NUMBER(6,2) NOT NULL,
  OL_SUPPLY_W_ID NUMBER NOT NULL,
  OL_QUANTITY NUMBER(2,0) NOT NULL,
```

```
  OL_DIST_INFO CHAR(24) NOT NULL,
  PRIMARY KEY (OL_W_ID,OL_D_ID,OL_O_ID,OL_NUMBER)
);

CREATE TABLE STOCK (
  S_W_ID NUMBER NOT NULL,
  S_I_ID NUMBER NOT NULL,
  S_QUANTITY NUMBER(4,0) NOT NULL,
  S_YTD NUMBER(8,2) NOT NULL,
  S_ORDER_CNT NUMBER NOT NULL,
  S_REMOTE_CNT NUMBER NOT NULL,
  S_DATA VARCHAR2(50) NOT NULL,
  S_DIST_01 CHAR(24) NOT NULL,
  S_DIST_02 CHAR(24) NOT NULL,
  S_DIST_03 CHAR(24) NOT NULL,
  S_DIST_04 CHAR(24) NOT NULL,
  S_DIST_05 CHAR(24) NOT NULL,
  S_DIST_06 CHAR(24) NOT NULL,
  S_DIST_07 CHAR(24) NOT NULL,
  S_DIST_08 CHAR(24) NOT NULL,
  S_DIST_09 CHAR(24) NOT NULL,
  S_DIST_10 CHAR(24) NOT NULL,
  PRIMARY KEY (S_W_ID,S_I_ID)
);
```

```
CREATE TABLE WAREHOUSE (
  W_ID NUMBER NOT NULL,
  W_YTD NUMBER(12,2) NOT NULL,
  W_TAX NUMBER(4,4) NOT NULL,
  W_NAME VARCHAR2(10) NOT NULL,
  W_STREET_1 VARCHAR2(20) NOT NULL,
  W_STREET_2 VARCHAR2(20) NOT NULL,
  W_CITY VARCHAR2(20) NOT NULL,
  W_STATE CHAR(2) NOT NULL,
  W_ZIP CHAR(9) NOT NULL,
  PRIMARY KEY (W_ID)
);
```

Figure A.1: DDL File to Create TPC-C Benchmark Tables in Oracle

```
<?xml version="1.0"?>
<parameters>
        <!-- General Workload configuration -->
   <dbtype>oracle</dbtype>
   <driver>oracle.jdbc.OracleDriver</driver>
   <DBUrl>jdbc:oracle:thin:@localhost:1521:orcl</DBUrl>
   <DBName>orcl</DBName>
   <username>system</username>
   <password>pass</password>

   <terminals>1</terminals>

   <works>
      <work>
        <time>60</time>
        <rate>500</rate>
        <weights>45,43,4,4,4</weights>
      </work>
```

```
        </works>

        <!-- TPCC specific -->
        <scalefactor>1</scalefactor>
        <dialect>config/dialects/tpcc_dialects.xml</dialect>
        <isolation>TRANSACTION_SERIALIZABLE</isolation>
        <transactiontypes>
        <transactiontype>
                <name>NewOrder</name>
                <id>1</id>
        </transactiontype>
        <transactiontype>
                <name>Payment</name>
                <id>2</id>
        </transactiontype>
        <transactiontype>
                <name>OrderStatus</name>
                <id>3</id>
        </transactiontype>
        <transactiontype>
                <name>Delivery</name>
                <id>4</id>
        </transactiontype>
        <transactiontype>
                <name>StockLevel</name>
                <id>5</id>
        </transactiontype>
        </transactiontypes>
</parameters>
```

Figure A.2: OLTP-Bench Configuration XML for Oracle

```
DROP TABLE WAREHOUSE IF EXISTS CASCADE;
CREATE TABLE WAREHOUSE (
  W_ID SMALLINT DEFAULT '0' NOT NULL,
  W_NAME VARCHAR(16) DEFAULT NULL,
  W_STREET_1 VARCHAR(32) DEFAULT NULL,
  W_STREET_2 VARCHAR(32) DEFAULT NULL,
  W_CITY VARCHAR(32) DEFAULT NULL,
  W_STATE VARCHAR(2) DEFAULT NULL,
  W_ZIP VARCHAR(9) DEFAULT NULL,
  W_TAX FLOAT DEFAULT NULL,
  W_YTD FLOAT DEFAULT NULL,
  CONSTRAINT W_PK_ARRAY PRIMARY KEY (W_ID)
);

DROP TABLE DISTRICT IF EXISTS CASCADE;
CREATE TABLE DISTRICT (
  D_ID TINYINT DEFAULT '0' NOT NULL,
  D_W_ID SMALLINT DEFAULT '0' NOT NULL REFERENCES WAREHOUSE (W_ID),
  D_NAME VARCHAR(16) DEFAULT NULL,
  D_STREET_1 VARCHAR(32) DEFAULT NULL,
  D_STREET_2 VARCHAR(32) DEFAULT NULL,
  D_CITY VARCHAR(32) DEFAULT NULL,
  D_STATE VARCHAR(2) DEFAULT NULL,
  D_ZIP VARCHAR(9) DEFAULT NULL,
  D_TAX FLOAT DEFAULT NULL,
  D_YTD FLOAT DEFAULT NULL,
  D_NEXT_O_ID INT DEFAULT NULL,
  PRIMARY KEY (D_W_ID,D_ID)
);

DROP TABLE ITEM IF EXISTS;
CREATE TABLE ITEM (
  I_ID INTEGER DEFAULT '0' NOT NULL,
  I_IM_ID INTEGER DEFAULT NULL,
  I_NAME VARCHAR(32) DEFAULT NULL,
  I_PRICE FLOAT DEFAULT NULL,
  I_DATA VARCHAR(64) DEFAULT NULL,
```

```
  CONSTRAINT I_PK_ARRAY PRIMARY KEY (I_ID)
);

DROP TABLE CUSTOMER IF EXISTS CASCADE;
CREATE TABLE CUSTOMER (
  C_ID INTEGER DEFAULT '0' NOT NULL,
  C_D_ID TINYINT DEFAULT '0' NOT NULL,
  C_W_ID SMALLINT DEFAULT '0' NOT NULL,
  C_FIRST VARCHAR(32) DEFAULT NULL,
  C_MIDDLE VARCHAR(2) DEFAULT NULL,
  C_LAST VARCHAR(32) DEFAULT NULL,
  C_STREET_1 VARCHAR(32) DEFAULT NULL,
  C_STREET_2 VARCHAR(32) DEFAULT NULL,
  C_CITY VARCHAR(32) DEFAULT NULL,
  C_STATE VARCHAR(2) DEFAULT NULL,
  C_ZIP VARCHAR(9) DEFAULT NULL,
  C_PHONE VARCHAR(32) DEFAULT NULL,
  C_SINCE TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
  C_CREDIT VARCHAR(2) DEFAULT NULL,
  C_CREDIT_LIM FLOAT DEFAULT NULL,
  C_DISCOUNT FLOAT DEFAULT NULL,
  C_BALANCE FLOAT DEFAULT NULL,
  C_YTD_PAYMENT FLOAT DEFAULT NULL,
  C_PAYMENT_CNT INTEGER DEFAULT NULL,
  C_DELIVERY_CNT INTEGER DEFAULT NULL,
  C_DATA VARCHAR(500),
  PRIMARY KEY (C_W_ID,C_D_ID,C_ID),
  UNIQUE (C_W_ID,C_D_ID,C_LAST,C_FIRST),
  CONSTRAINT C_FKEY_D FOREIGN KEY (C_D_ID, C_W_ID) REFERENCES DISTRICT (D_ID, D_W_ID)
);
CREATE INDEX IDX_CUSTOMER ON CUSTOMER (C_W_ID,C_D_ID,C_LAST);

DROP TABLE HISTORY IF EXISTS CASCADE;
CREATE TABLE HISTORY (
  H_C_ID INTEGER DEFAULT NULL,
  H_C_D_ID TINYINT DEFAULT NULL,
  H_C_W_ID SMALLINT DEFAULT NULL,
```

```
  H_D_ID TINYINT DEFAULT NULL,
  H_W_ID SMALLINT DEFAULT '0' NOT NULL,
  H_DATE TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
  H_AMOUNT FLOAT DEFAULT NULL,
  H_DATA VARCHAR(32) DEFAULT NULL,
  CONSTRAINT H_FKEY_C FOREIGN KEY (H_C_ID, H_C_D_ID, H_C_W_ID) REFERENCES CUSTOMER
(C_ID, C_D_ID, C_W_ID),
  CONSTRAINT H_FKEY_D FOREIGN KEY (H_D_ID, H_W_ID) REFERENCES DISTRICT (D_ID, D_W_ID)
);

DROP TABLE STOCK IF EXISTS CASCADE;
CREATE TABLE STOCK (
  S_I_ID INTEGER DEFAULT '0' NOT NULL REFERENCES ITEM (I_ID),
  S_W_ID SMALLINT DEFAULT '0 ' NOT NULL REFERENCES WAREHOUSE (W_ID),
  S_QUANTITY INTEGER DEFAULT '0' NOT NULL,
  S_DIST_01 VARCHAR(32) DEFAULT NULL,
  S_DIST_02 VARCHAR(32) DEFAULT NULL,
  S_DIST_03 VARCHAR(32) DEFAULT NULL,
  S_DIST_04 VARCHAR(32) DEFAULT NULL,
  S_DIST_05 VARCHAR(32) DEFAULT NULL,
  S_DIST_06 VARCHAR(32) DEFAULT NULL,
  S_DIST_07 VARCHAR(32) DEFAULT NULL,
  S_DIST_08 VARCHAR(32) DEFAULT NULL,
  S_DIST_09 VARCHAR(32) DEFAULT NULL,
  S_DIST_10 VARCHAR(32) DEFAULT NULL,
  S_YTD INTEGER DEFAULT NULL,
  S_ORDER_CNT INTEGER DEFAULT NULL,
  S_REMOTE_CNT INTEGER DEFAULT NULL,
  S_DATA VARCHAR(64) DEFAULT NULL,
  PRIMARY KEY (S_W_ID,S_I_ID)
);

DROP TABLE OORDER IF EXISTS CASCADE;
CREATE TABLE OORDER (
  O_ID INTEGER DEFAULT '0' NOT NULL,
  O_D_ID TINYINT DEFAULT '0' NOT NULL,
  O_W_ID SMALLINT DEFAULT '0' NOT NULL,
```

```
  O_C_ID INTEGER DEFAULT NULL,
  O_ENTRY_D TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
  O_CARRIER_ID INTEGER DEFAULT NULL,
  O_OL_CNT INTEGER DEFAULT NULL,
  O_ALL_LOCAL INTEGER DEFAULT NULL,
  PRIMARY KEY (O_W_ID,O_D_ID,O_ID),
  UNIQUE (O_W_ID,O_D_ID,O_C_ID,O_ID),
  CONSTRAINT O_FKEY_C FOREIGN KEY (O_D_ID, O_W_ID, O_C_ID) REFERENCES CUSTOMER
(C_D_ID, C_W_ID, C_ID)
);
CREATE INDEX IDX_OORDER ON OORDER (O_W_ID,O_D_ID,O_C_ID);

DROP TABLE NEW_ORDER IF EXISTS CASCADE;
CREATE TABLE NEW_ORDER (
  NO_O_ID INTEGER DEFAULT '0' NOT NULL,
  NO_D_ID TINYINT DEFAULT '0' NOT NULL,
  NO_W_ID SMALLINT DEFAULT '0' NOT NULL,
  CONSTRAINT NO_PK_TREE PRIMARY KEY (NO_D_ID,NO_W_ID,NO_O_ID),
  CONSTRAINT NO_FKEY_O FOREIGN KEY (NO_O_ID, NO_D_ID, NO_W_ID) REFERENCES OORDER
(O_ID, O_D_ID, O_W_ID)
);

DROP TABLE ORDER_LINE IF EXISTS CASCADE;
CREATE TABLE ORDER_LINE (
  OL_O_ID INTEGER DEFAULT '0' NOT NULL,
  OL_D_ID TINYINT DEFAULT '0' NOT NULL,
  OL_W_ID SMALLINT DEFAULT '0' NOT NULL,
  OL_NUMBER INTEGER DEFAULT '0' NOT NULL,
  OL_I_ID INTEGER DEFAULT NULL,
  OL_SUPPLY_W_ID SMALLINT DEFAULT NULL,
  OL_DELIVERY_D TIMESTAMP DEFAULT NULL,
  OL_QUANTITY INTEGER DEFAULT NULL,
  OL_AMOUNT FLOAT DEFAULT NULL,
  OL_DIST_INFO VARCHAR(32) DEFAULT NULL,
  PRIMARY KEY (OL_W_ID,OL_D_ID,OL_O_ID,OL_NUMBER),
  CONSTRAINT OL_FKEY_O FOREIGN KEY (OL_O_ID, OL_D_ID, OL_W_ID) REFERENCES OORDER
(O_ID, O_D_ID, O_W_ID),
```

```
  CONSTRAINT OL_FKEY_S FOREIGN KEY (OL_I_ID, OL_SUPPLY_W_ID) REFERENCES STOCK
(S_I_ID, S_W_ID)
);
--CREATE INDEX IDX_ORDER_LINE_3COL ON ORDER_LINE (OL_W_ID,OL_D_ID,OL_O_ID);
--CREATE INDEX IDX_ORDER_LINE_2COL ON ORDER_LINE (OL_W_ID,OL_D_ID);
CREATE INDEX IDX_ORDER_LINE_TREE ON ORDER_LINE (OL_W_ID,OL_D_ID,OL_O_ID);
```

Figure A.3: DDL File to Create TPC-C Benchmark Tables in VoltDB

```
<?xml version="1.0"?>
<parameters>
<!-- Connection details -->
   <dbtype>volt</dbtype>
   <driver>org.voltdb.jdbc.Driver</driver>
   <DBUrl>jdbc:voltdb://localhost:21212</DBUrl>
   <username>localhost</username>
   <password></password>
   <isolation>TRANSACTION_SERIALIZABLE</isolation>
   <works>
      <work>
        <time>60</time>
        <rate>500</rate>
        <weights>45,43,4,4,4</weights>
      </work>
          </works>
          <!-- TPCC specific -->
          <scalefactor>1</scalefactor>
          <dialect>config/dialects/tpcc_dialects.xml</dialect>
          <isolation>TRANSACTION_SERIALIZABLE</isolation>
          <transactiontypes>
          <transactiontype>
                    <name>NewOrder</name>
                    <id>1</id>
          </transactiontype>
          <transactiontype>
                    <name>Payment</name>
```

```
            <id>2</id>
        </transactiontype>
        <transactiontype>
            <name>OrderStatus</name>
            <id>3</id>
        </transactiontype>
        <transactiontype>
            <name>Delivery</name>
            <id>4</id>
        </transactiontype>
        <transactiontype>
            <name>StockLevel</name>
            <id>5</id>
        </transactiontype>
        </transactiontypes>
</parameters>
```

Figure A.4: OLTP-Bench Configuration XML for VoltDB

```
DROP TABLE IF EXISTS CUSTOMER;
DROP TABLE IF EXISTS DISTRICT;
DROP TABLE IF EXISTS HISTORY;
DROP TABLE IF EXISTS ITEM;
DROP TABLE IF EXISTS NEW_ORDER;
DROP TABLE IF EXISTS OORDER;
DROP TABLE IF EXISTS ORDER_LINE;
DROP TABLE IF EXISTS STOCK;
DROP TABLE IF EXISTS WAREHOUSE;

CREATE TABLE CUSTOMER (
   C_ID INTEGER NOT NULL,
   C_D_ID INTEGER NOT NULL,
   C_W_ID INTEGER NOT NULL,
   C_FIRST CHAR(16),
   C_MIDDLE CHAR(2),
   C_LAST CHAR(16),
   C_STREET_1 CHAR(20),
```

```
    C_STREET_2 CHAR(20),
    C_CITY CHAR(20),
    C_STATE CHAR(2),
    C_ZIP CHAR(9),
    C_PHONE CHAR(16),
    C_SINCE DATE,
    C_CREDIT CHAR(2),
    C_CREDIT_LIM DECIMAL(12,2),
    C_DISCOUNT DECIMAL(4,4),
    C_BALANCE DECIMAL(12,2),
    C_YTD_PAYMENT DECIMAL(12,2),
    C_PAYMENT_CNT INTEGER,
    C_DELIVERY_CNT INTEGER,
    C_DATA VARCHAR(500),
    PRIMARY KEY (C_W_ID,C_D_ID,C_ID)
);

CREATE TABLE DISTRICT (
    D_ID INTEGER NOT NULL,
    D_W_ID INTEGER NOT NULL,
    D_NAME CHAR(10),
    D_STREET_1 CHAR(20),
    D_STREET_2 CHAR(20),
    D_CITY CHAR(20),
    D_STATE CHAR(2),
    D_ZIP CHAR(9),
    D_TAX DECIMAL(4,4),
    D_YTD DECIMAL(12,2),
    D_NEXT_O_ID INTEGER,
    PRIMARY KEY (D_W_ID,D_ID)
);

CREATE TABLE HISTORY (
    H_C_ID INTEGER NOT NULL,
    H_C_D_ID INTEGER,
    H_C_W_ID INTEGER,
    H_D_ID INTEGER,
```

```
    H_W_ID INTEGER,
    H_DATE TIMESTAMP NOT NULL,
    H_AMOUNT DECIMAL(6,2),
    H_DATA CHAR(24),
    PRIMARY KEY (H_DATE, H_C_ID)
);

CREATE TABLE ITEM (
    I_ID INTEGER NOT NULL,
    I_IM_ID INTEGER,
    I_NAME CHAR(24),
    I_PRICE DECIMAL(5,2),
    I_DATA CHAR(50),
    PRIMARY KEY (I_ID)
);

CREATE TABLE NEW_ORDER (
    NO_O_ID INTEGER NOT NULL,
    NO_D_ID INTEGER NOT NULL,
    NO_W_ID INTEGER NOT NULL,
    PRIMARY KEY (NO_W_ID,NO_D_ID,NO_O_ID)
);

CREATE TABLE ORDER_LINE (
    OL_O_ID INTEGER NOT NULL,
    OL_D_ID INTEGER NOT NULL,
    OL_W_ID INTEGER NOT NULL,
    OL_NUMBER INTEGER NOT NULL,
    OL_I_ID INTEGER,
    OL_SUPPLY_W_ID INTEGER,
    OL_DELIVERY_D TIMESTAMP,
    OL_QUANTITY INTEGER,
    OL_AMOUNT DECIMAL(6,2),
    OL_DIST_INFO CHAR(24),
    PRIMARY KEY (OL_W_ID,OL_D_ID,OL_O_ID,OL_NUMBER)
);
```

```
CREATE TABLE OORDER (
   O_ID INTEGER NOT NULL,
   O_D_ID INTEGER NOT NULL,
   O_W_ID INTEGER NOT NULL,
   O_C_ID INTEGER,
   O_ENTRY_D TIMESTAMP,
   O_CARRIER_ID INTEGER,
   O_OL_CNT INTEGER,
   O_ALL_LOCAL INTEGER,
   PRIMARY KEY (O_W_ID,O_D_ID,O_ID)
);

CREATE TABLE STOCK (
   S_I_ID INTEGER NOT NULL,
   S_W_ID INTEGER NOT NULL,
   S_QUANTITY INTEGER,
   S_DIST_01 CHAR(24),
   S_DIST_02 CHAR(24),
   S_DIST_03 CHAR(24),
   S_DIST_04 CHAR(24),
   S_DIST_05 CHAR(24),
   S_DIST_06 CHAR(24),
   S_DIST_07 CHAR(24),
   S_DIST_08 CHAR(24),
   S_DIST_09 CHAR(24),
   S_DIST_10 CHAR(24),
   S_YTD INTEGER,
   S_ORDER_CNT INTEGER,
   S_REMOTE_CNT INTEGER,
   S_DATA CHAR(50),
   PRIMARY KEY (S_W_ID,S_I_ID)
);

CREATE TABLE WAREHOUSE (
   W_ID INTEGER NOT NULL,
   W_NAME CHAR(10),
   W_STREET_1 CHAR(20),
```

```
   W_STREET_2 CHAR(20),
   W_CITY CHAR(20),
   W_STATE CHAR(2),
   W_ZIP CHAR(9),
   W_TAX DECIMAL(4,4),
   W_YTD DECIMAL(12,2),
   PRIMARY KEY (W_ID)
);

CREATE INDEX IX_CUSTOMER ON CUSTOMER (C_W_ID, C_D_ID, C_LAST);
CREATE INDEX IX_OORDER ON OORDER (O_W_ID, O_D_ID, O_C_ID);
```

Figure A.5: DDL File to Create TPC-C Benchmark Tables in Splice Machine

```
<?xml version="1.0"?>
<parameters>
        <!-- General Workload configuration -->
   <dbtype>splice</dbtype>
   <driver>com.splicemachine.db.jdbc.ClientDriver</driver>
   <DBUrl>jdbc:splice://localhost:1527/splicedb</DBUrl>
   <DBName>tpcc</DBName>
   <username>splice</username>
   <password>admin</password>
  <terminals>1</terminals>

   <works>
      <work>
        <time>60</time>
        <rate>500</rate>
        <weights>45,43,4,4,4</weights>
      </work>
        </works>

        <!-- TPCC specific -->
        <scalefactor>1</scalefactor>
        <isolation>TRANSACTION_SERIALIZABLE</isolation>
        <transactiontypes>
```

```
        <transactiontype>
                <name>NewOrder</name>
                <id>1</id>
        </transactiontype>
        <transactiontype>
                <name>Payment</name>
                <id>2</id>
        </transactiontype>
        <transactiontype>
                <name>OrderStatus</name>
                <id>3</id>
        </transactiontype>
        <transactiontype>
                <name>Delivery</name>
                <id>4</id>
        </transactiontype>
        <transactiontype>
                <name>StockLevel</name>
                <id>5</id>
        </transactiontype>
        </transactiontypes>
</parameters>
```

Figure A.6: OLTP-Bench Configuration XML for Splice Machine

```
</dialect>
        <dialect type="SPLICE">
    <procedure name="Delivery">
      <statement name="delivGetOrderIdSQL">
        SELECT NO_O_ID
          FROM NEW_ORDER
          WHERE NO_D_ID = ? AND NO_W_ID = ? ORDER BY NO_O_ID ASC FETCH FIRST 1 ROWS
ONLY
      </statement>
```

```
    </procedure>
    <procedure name="OrderStatus">
      <statement name="ordStatGetNewestOrdSQL">
        SELECT O_ID, O_CARRIER_ID, O_ENTRY_D
            FROM OORDER
            WHERE O_W_ID = ? AND O_D_ID = ? AND O_C_ID = ? ORDER BY O_ID DESC FETCH
FIRST 1 ROWS ONLY
      </statement>
    </procedure>
  </dialect>
```

Figure A.7: Additional Entry to TPCC-Dialects.XML File for Splice Machine

```
  <dialect type="VOLT">
    <procedure name="Delivery">
      <statement name="delivGetOrderIdSQL">
        SELECT NO_O_ID
                    FROM NEW_ORDER
                    WHERE NO_D_ID = ? AND NO_W_ID = ? AND NO_O_ID > -1 LIMIT 1
      </statement>
    </procedure>
    <procedure name="OrderStatus">
      <statement name="ordStatGetNewestOrdSQL">
            SELECT O_ID, O_CARRIER_ID, O_ENTRY_D
                    FROM ORDERS
                    WHERE O_W_ID = ? AND O_D_ID = ? AND O_C_ID = ? ORDER BY O_ID DESC
LIMIT 1
      </statement>
    </procedure>
      <procedure name="NewOrder">
    <statement name="stmtGetDistSQL">
      SELECT D_TAX, D_NEXT_O_ID
                    FROM DISTRICT
                    WHERE D_ID = ? AND D_W_ID = ?
```

```
        </statement>
        <statement name="stmtGetStockSQL">
           SELECT S_QUANTITY, S_DATA, S_YTD, S_ORDER_CNT, S_REMOTE_CNT, S_DIST_01,
S_DIST_02, S_DIST_03, S_DIST_04, S_DIST_05,
              S_DIST_06, S_DIST_07, S_DIST_08, S_DIST_09, S_DIST_10
              FROM STOCK
              WHERE S_I_ID = ? AND S_W_ID = ?
        </statement>
      </procedure>
         <procedure name="StockLevel">
        <statement name="stockGetCountStockSQL">
                  SELECT COUNT(DISTINCT(OL_I_ID))
                        FROM ORDER_LINE, STOCK
                        WHERE OL_W_ID = ?
               AND OL_D_ID = ?
                        AND OL_O_ID &lt; ?
                        AND OL_O_ID >= ?
                        AND S_W_ID = ?
                        AND  S_I_ID = OL_I_ID
                        AND S_QUANTITY &lt; ?
        </statement>
      </procedure>
   </dialect>
```

Figure A.8: Additional Entry to TPCC-Dialects.XML File for VoltDB

```
VOLT("org.voltdb.jdbc.Driver", true, false)
SPLICE("com.splicemachine.db.jdbc.ClientDriver", true, false)
```

Figure A.9: Additional Entry to DatabaseType.java File

APPENDIX B - VOLTDB COMPONENTS

```
import com.Constants;

java -classpath $APPNAME-client.jar:$APPCLASSPATH:$APPNAME-procs.jar com.MyTPCC \
```

Figure B.1: Additions to MyTPCC.java File Necessary to Run Legacy TPC-C Benchmark