CoherentPaaS

Coherent and Rich PaaS with a Common Programming Model

ICT FP7-611068

Coherent PaaS Evaluation

D10.4

September 2016

Document Information

Scheduled delivery	30.09.2016
Actual delivery	30.09.2016
Version	1.3
Responsible Partner	UPM

Dissemination Level:

PU Public
PP Restricted to other programme participants (including the Commission)
RE Restricted to a group specified by the consortium (including the Commission)
CO Confidential, only for members of the consortium (including the Commission)

Revision History

Date	Editor	Status	Version	Changes
31.08.2016	Iván Brondino	Draft	0.1	TOC
29.09.2016	François Savary	Draft	0.2	Active Pivot contribution
29.09.2016	Giorgos Saloustros	Draft	0.3	H-Eutropia contribution
29.09.2016	Pavlos Kranas	Draft	0.4	MongoDB contribution
29.09.2016	Raquel Pau	Draft	0.5	Sparksee contribution
29.09.2016	Iván Brondino	Draft	0.6	LeanXcale contribution
29.09.2016	Ying Zhang	Draft	0.7	MonetDB contribution
29.09.2016	Valerio Vianello	Draft	0.8	CEP contribution
29.09.2016	Boyan Kolev	Draft	0.9	Common Query Language contribution
29.09.2016	Iván Brondino	Draft	1.0	Initial version
29.09.2016	Miguel Matos	Draft	1.1	Internal Review
29.09.2016	Pavlos Kranas	ICCS	1.2	Internal Review
29.01.2016	Marta Patiño	UPM	1.3	Final Version

Contributors

Iván Brondino (LeanXcale), Marta Patiño-Martínez (UPM), Ricardo Jiménez-Peris (LeanXcale), François Savary (QuartetFS), Giorgos Saloustros (FORTH), Pavlos Kranas (ICCS), Raquel Pau (Sparsity Technologies), Miguel Matos (LeanXcale), Ying Zhang (MonetDB), Ricardo Vilaça (LeanXcale), Boyan Kolev (INRIA)

Internal Reviewers

Miguel Matos (LeanXcale)
Pavlos Kranas (ICCS)

Acknowledgements

Research partially funded by EC 7th Framework Programme FP7/2007-2013 under grant agreement n° 611068.

More information

Additional information and public deliverables of CoherentPaaS can be found at: http://coherentpaas.eu

Glossary of Acronyms

Acronym	Definition
D	Deliverable
DoW	Description of Work
EC	European Commission
PM	Project Manager
PO	Project Officer
WP	Work Package

Table of Contents

List of Figures

1. Executive Summary

The CoherentPaaS project aims at providing with a cloud data stores ecosystem gifted with full ACID transactional semantics and common query language that enables to query heterogeneous data stores such as document, graph, SQL and columnar databases. This deliverable describes and discuss the results of the performance evaluation of the platform.

2. Introduction

The data management world has been evolving towards a large diversity of data management technologies. The main motivation is the increasing demand for flexibility, efficiency and scalability. This blooming of data management technologies, where each technology is specialized and optimal for specific processing, has led to a "no one size fits all" situation. On one hand, traditional SQL databases have diverged into operational and analytical databases and then an evolution of them has led to a specialization for particular challenges and workloads. On the other hand, a new world has appeared NoSQL data stores, where new data models, query languages and APIs appropriate for those data models have been proposed. NoSQL data stores already targeted scalability, but with the cost of renouncing to the data consistency provided by transactions, because at the time of their design transactional processing was the bottleneck.

This trend has resulted in a large proliferation of query languages and APIs leading to a number of isolated silos that create increasing pains at enterprises due to the difficulties in updating these independent silos and joining data across them.

CoherentPaaS addresses all these issues by providing a rich PaaS with a wide diversity of data stores and data management technologies optimized for particular tasks, data, and workloads. CoherentPaaS integrates NoSQL, SQL data stores, and complex event processing data management systems, providing them with holistic transactional coherence and enabling correlation of data across data stores by means of a common query language.

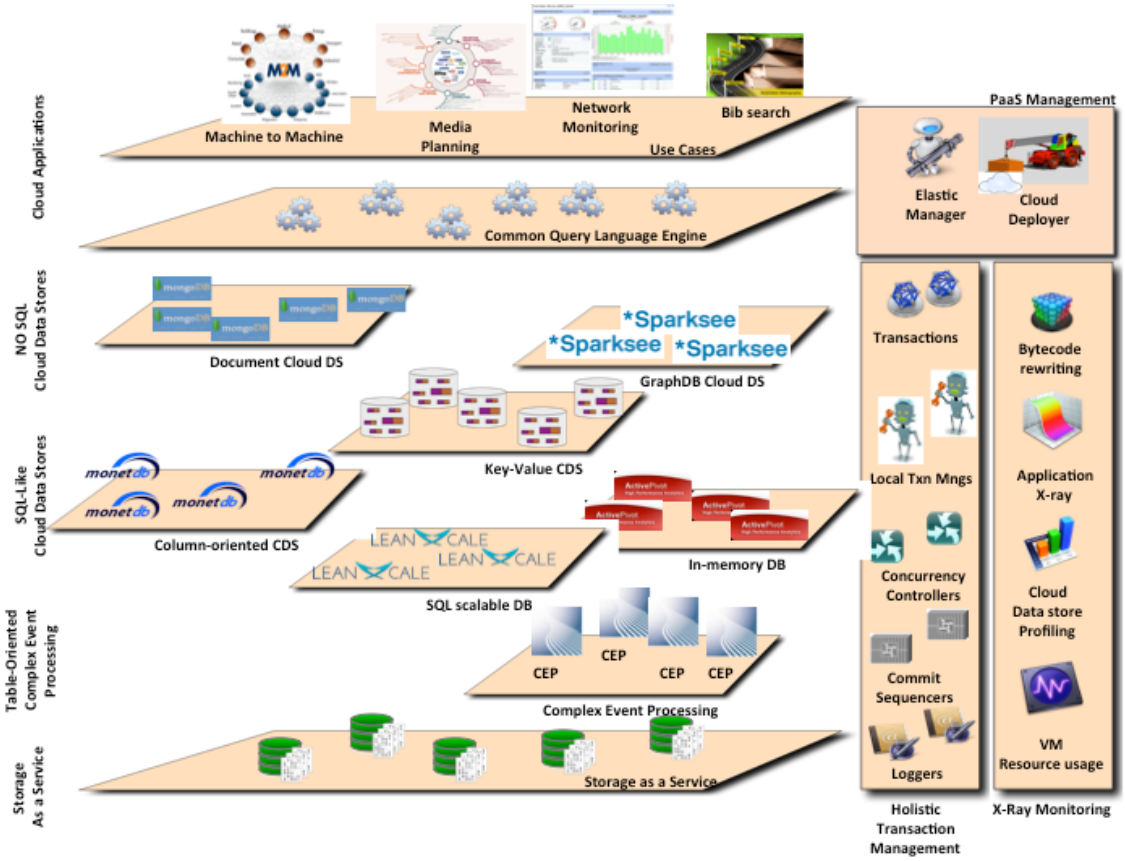

Figure 1 CoherentPaaS Global Architecture

The goal of this deliverable is to present the final performance evaluation of the Coherent PaaS platform. Due to the richness of CoherentPaaS, a single evaluation does not cover all the features provided by the platform. Therefore, we decided to exercise every single aspect of the solution in a variety of evaluations using different benchmarks. The CoherentPaaS platform presents two main innovations that needed to be thoroughly tested: a) Full ACID global transactions across heterogeneous data stores through the Holistic Transaction Manager (HTM); and b) the possibility to query different data stores through a unified SQL like query language, the CloudMdsQL.

We start by measuring the cost of integrating the HTM in every data stores. In order to support holistic transactions, data stores had to implement multi-versioning (HTM offers Snapshot Isolation) and to enrich the data store clients to communicate with the HTM client (the Local Transaction Manager). More details can be found in restricted deliverables [1][1][2][2][3][3][4][4]. The evaluation consists in running a well-known benchmark for cloud data stores, Yahoo! Cloud Serving Benchmark (YCSB)[5]. The benchmark provides with a variety of operation types (reads, inserts, updates, scans, among others) and allow to configure custom workloads selecting the proportion of each operation type. In order to stress the SQL like capabilities of the Common Query Engine (CQE) we have designed a benchmark based on the well-known TPC-H benchmark. The performed evaluation of the Common Query Language (CQE) aims at showing the ability to run optimized queries across heterogeneous data stores, as well as to evaluate the functional and performance benefits of the usage of native queries in combination with SQL statements. Throughout the performed experiments, we measure the execution times of different queries and operators, supported by the common query language, in the context of two major groups: performance and functional evaluations. Within the performance evaluation, we assess the different optimization techniques, enabled by the language and engine, by comparing the execution times with and without applying optimization. For the functional benefits, we stress on the extended set of operators the CoherentPaaS platform provides to data stores that do not natively support them. Since the CloudMdsQL language is SQL-like, we have designed our benchmark on top of the well-known TPC-H[6] benchmark.

This deliverable presents three scalability evaluations. We study the scalability of a system in terms of maximum sustainable throughput under a response time constraint. CoherentPaaS comes with a scalable brand new key-value data store, H-Eutropia; a scalable SQL database, LeanXcale; and a scalable complex event processing system (CEP) based on Storm[7]. We study the scalability of LeanXcale running the *de facto* database

[1] Restricted deliverable D4.2 – Local Transaction Manager API describes the API exposed by the LTM to the Applications, Common Query Engine and to Data Stores. The API describes how to start and commit transactions (for Apps and CQE) and the general contract that needs to be implemented by the data stores in order to provide full ACID transactional semantics.

[2] Restricted deliverable D4.3 – Local Transaction Manager for all cloud data stores describes the implementation of all data stores of the general contract described in restricted deliverable D4.2.

[3] Restricted deliverable D4.4 – Recovery Management for all cloud data stores describes the recovery protocol of the HTM. It describes as well the implementation of the common contract described in D4.2 with regard the recovery path in the HTM.

[4] Restricted deliverable D4.6 – Holistic Transaction Manager provides a deep dive in the HTM architecture and protocols. The HTM is protocols are generic and provide a common interface, described in D4.2, which allows to enhance any data store with transactional semantics.

5 https://github.com/brianfrankcooper/YCSB

6 http://www.tpc.org/tpch/

7 http://storm.apache.org

benchmark TPC-C[8], the scalability of H-Eutropia running the YCSB benchmark and a joint scalability evaluation of the CEP with H-Eutropia.
The remainder of this deliverable is structured as follows: Section 2 presents the HTM overhead evaluation in cloud data stores. Section 3 the evaluation of the CQE and CloudMdSQL. Section 4 presents the H-Eutropia scalability evaluation. Section 5 presents the LeanXcale scalability evaluation results running TPC-C. Section 6 presents the scalability of the platform in terms of number of supported data stores. Finally, Section 7 presents the scalability of the CEP with H-Eutropia.

2.1. Relation with other deliverables

This deliverable references previous work done in the context of the project. The outcome of this work is detailed in the following restricted deliverables:

- **D4.6 – Holistic Transaction Manager** provides a deep dive in the HTM architecture and protocols. The HTM is a generic layer with a common interface, described in D4.2, which allows to enhance any data store with transactional semantics. The deliverable focus in how we scale the transactional semantics.
- **D4.2 – Local Transaction Manager API** describes the API exposed by the LTM to the Applications, Common Query Engine and cloud data stores. The API describes how to start and commit transactions (for Apps and CQE) and the general contract that needs to be implemented by the data stores in order to provide full ACID transactional semantics across cloud data stores.
- **D4.3 – Local Transaction Manager for all cloud data stores** describes the implementation of all data stores of the general contract described in restricted deliverable D4.2.
- **D4.4 – Recovery Management for all cloud data stores** describes the recovery protocols of the HTM. It describes as well the implementation details of the common contract described in D4.2 with regard the recovery path by the cloud data stores. The recovery is important to ensure the Durability and Atomicity of the ACID semantics.
- **D5.2 No-SQL data store implementation (initial version)** presents the technical design of the No-SQL data stores. The deliverable focus in the multi-versioning implementation on the persistent storage, required for the integration with the HTM to support transactional semantics.
- **D5.3 No-SQL data store implementation (final version)** presents a technical description and the technical implementation details of the data store. The deliverable focus in the multi-versioning implementation on the persistent storage, required for the integration with the HTM to support transactional semantics
- **D6.2 Cloud SQL-like data store implementation (initial version)** presents a technical description of the SQL data stores. The deliverable focus in the multi-versioning implementation on the persistent storage of the data stores, required for the integration with the HTM to support transactional semantics.
- **D6.3 Cloud SQL-like data store implementation (final version)** presents a technical description and the technical implementation details of the SQL data stores. The deliverable focus in the multi-versioning implementation on the

8 http://www.tpc.org/tpcc/

persistent storage of the data stores, required for the integration with the HTM to support transactional semantics.

3. Holistic Transaction Manager and MVCC overhead evaluation

In this section we present a performance evaluation of each data store that has been enriched with the CoherentPaaS transactions. For comparison purposes we present the same evaluation with and without transactions. The evaluation has been conducted using the Yahoo Cloud Serving Benchmark (YCSB), an up-to-date benchmark for data stores that has been enriched to support CoherentPaaS transactions. YCSB client implement basic data store row operations such as READ, INSERT, UPDATE, SCAN, READ-MODIFY operations. Each data store implemented its own YCSB client with the corresponding invocation to specific data store operations.
YCSB operations access a single table, *usertable*. *Usertable* is populated with synthetic data generated by the benchmark. The benchmark allows to customize the *usertable* by defining the key and columns length and the number of columns. For this evaluation we use the standard configuration of user table with a key of 100 bytes and 10 columns of 100 bytes each.
For each data store we carefully designed appropriate workloads that resembles a typical workload given the data store nature. The workload comprises some of the YCSB operations and for each operation it is defined a proportion of occurrence. For example, a Read Only Workload could comprise 50 % of READs and 50 % of Scans.
Having defined the workloads, for each workload we run the with different number of clients against the data store, with and without transactional support. We group a fixed number of operations, a batch, if running against the vanilla version of the data store; or a transaction if running with transactional support, and we measure the latency of executing the batch/transaction and the throughput in terms of batches/transactions per second.
Some data stores like Sparksee, LeanXcale and MonetDB already provide with transactional support. We measure the overhead of providing global coordination support for this type of data stores.
For each data store we choose a small and a big dataset. The size of the datasets used varies depending the data store.
We present the results in terms of percentage of latency increase and throughput loss in order to measure the actual overhead of providing transactional support.

3.1. H-Eutropia

3.1.1. Setup

Our experimental platform consists of two machines equipped with a 16 core Intel(R) Xeon(R) E5-2630 CPU running at 2.4 GHz. Server is equipped with 256 GB DDR-IV DRAM. Both nodes are connected with a 1 Gbits/s network link. As storage device, the server uses a Samsung SSD PRO 950 NVMe of size 256GB. H-Eutropia is implemented in C and can be accessed from applications as a shared library. H-Eutropia is cross-linked between the Java code of HBase and the C code of H-Eutropia. We use the open-source Yahoo Cloud Serving Benchmark YCSB to generate workloads in both transactional and non-transactional cases. We deploy Holistic Transaction Manager (HTM) in a separate machine which is connected via 1 Gbits/s network link.

We use the open-source Yahoo Cloud Serving Benchmark (YCSB) to generate synthetic workloads. In YCSB we have added support for integration with HTM. Each transaction consists of a set of operations (get, scan, put). In all case transaction size is set to 120. When transactions are disabled we instead send a batch of the same operations to the server.

3.1.2. Database

For our experiments we use rows where each contains 10 key/value pairs each. Each key is of size 31 bytes and each value is of size 128. We run our experiments for two database sizes: small and large. Small database consists of 10 million rows consisting of 100 million key value pairs of total size approximately 16GB. Large database consists of 65 million rows which map to 650 million key value pairs of size 100GB. Databases are preloaded in H-Eutropia. We use 4 regions in all cases in a single region server.

3.1.3. Evaluation

We initially evaluate read only workloads which dominate data analytics. Subsequently we run a mix workload which contains also updates to fully evaluate the HTM overhead. Each get operation fetches a row of 10 key value pairs. Each scan operation fetches on average 16 rows or 160 key value pairs each. Mix workload consists of 70% scans, 25% gets, and 5% updates. Each update fully updates a row. All mix transactions contain a set of update operations since batch size is set to 120 and update percentage is 5%. Below we depict our results depicting the relative percentage of throughput loss and latency increase when using the CPAAS transactional stack. We run the benchmark with increasing number of load until system reaches peak performance. The following figures present the evaluation results.

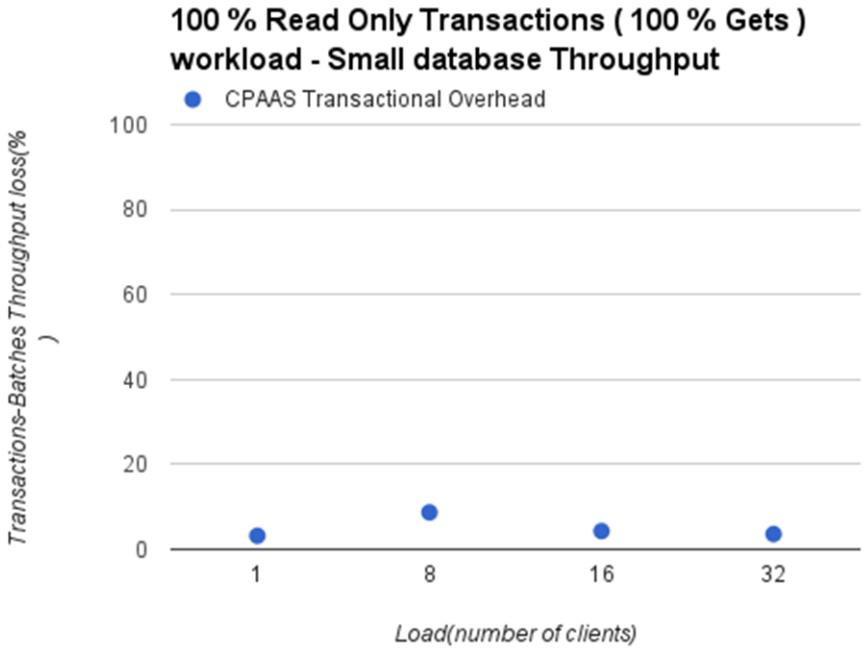

Figure 2 H-Eutropia - 100 % Read Only Transactions (100 % Gets) workload - Small database -Transactions-Batches Throughput.

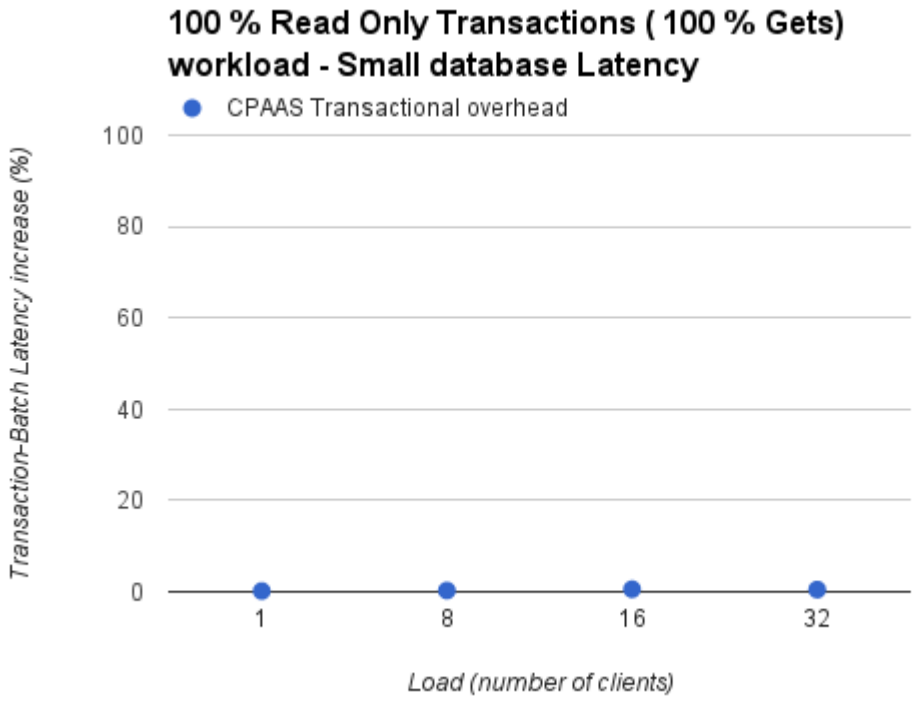

Figure 3 H-Eutropia - 100 % Read Only Transactions (100 % Gets) workload - Small database -Transactions-Batches Latency.

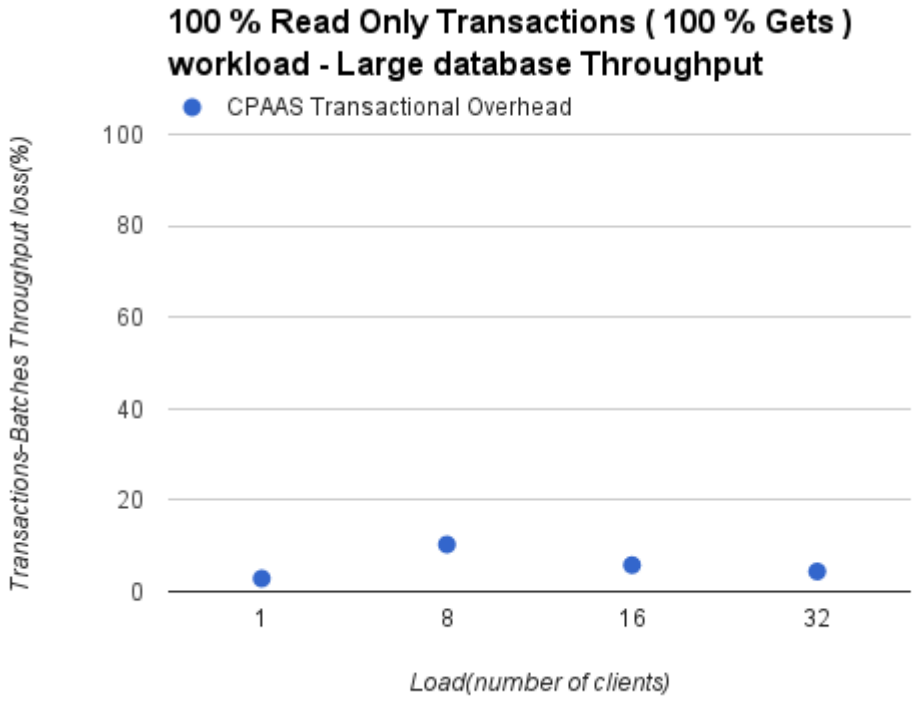

Figure 4 H-Eutropia -100 % Read Only Transactions (100 % Gets) workload - Large database -Transactions-Batches Throughput.

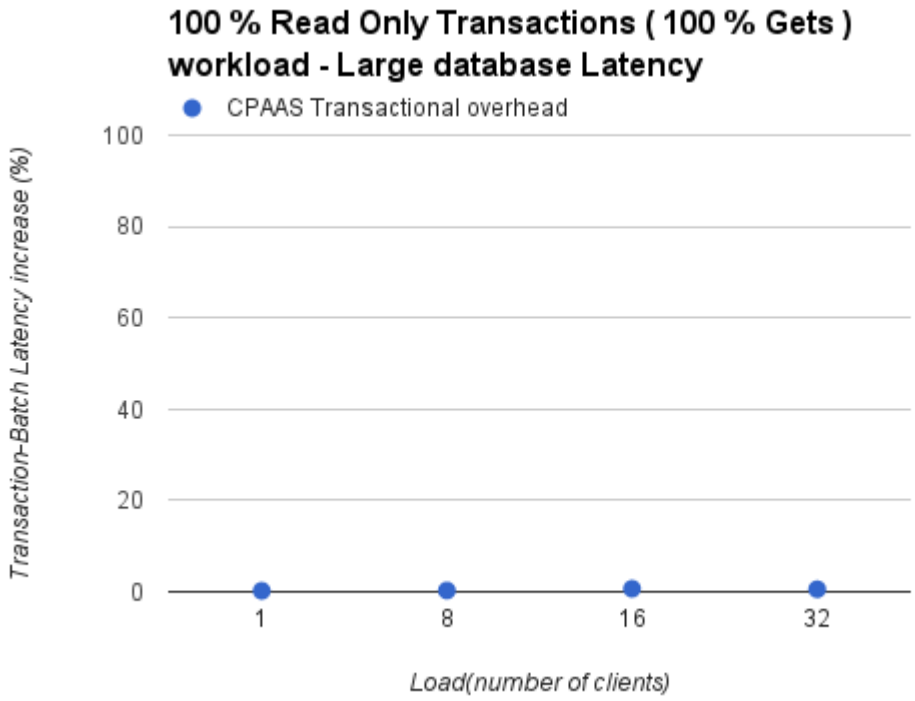

Figure 5 H-Eutropia -100 % Read Only Transactions (100 % Gets) workload - Large database -Transactions-Batches Latency.

.

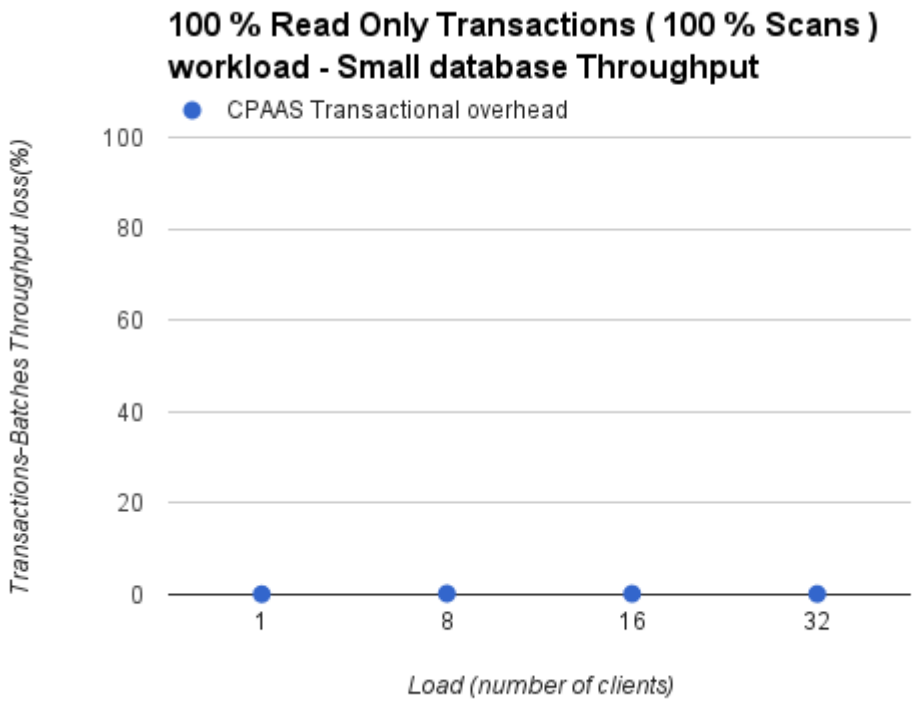

Figure 6 H-Eutropia -100 % Read Only Transactions (100 % Scan) workload - Small database -Transactions-Batches Throughput.

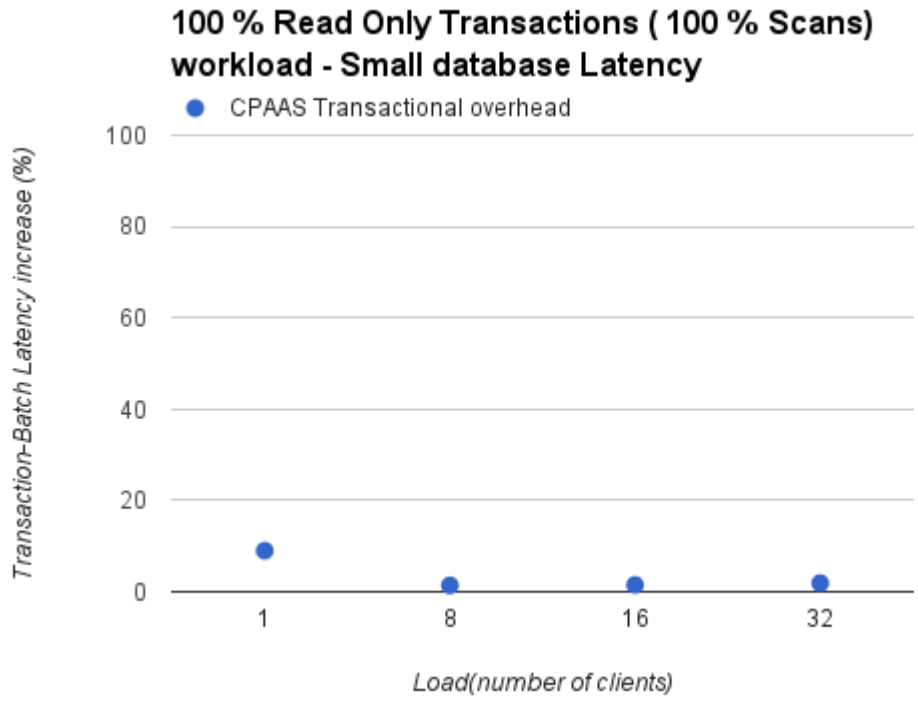

Figure 7 H-Eutropia -100 % Read Only Transactions (100 % Scan) workload - Small database -Transactions-Batches Latency.

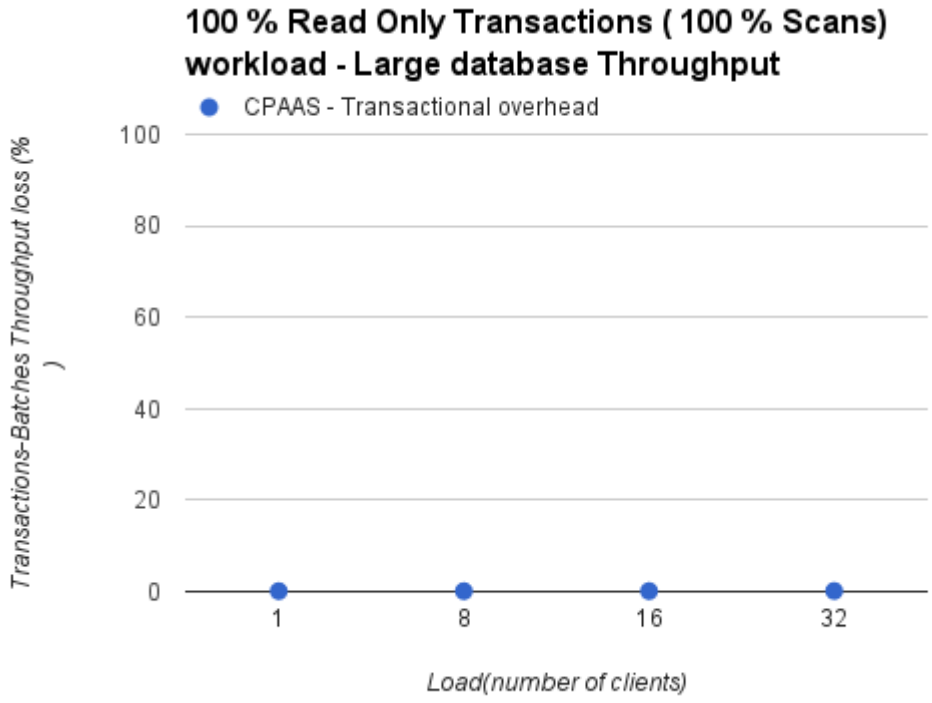

Figure 8 H-Eutropia - 100 % Read Only Transactions (100 % Scan) workload - Large database -Transactions-Batches Throughput.

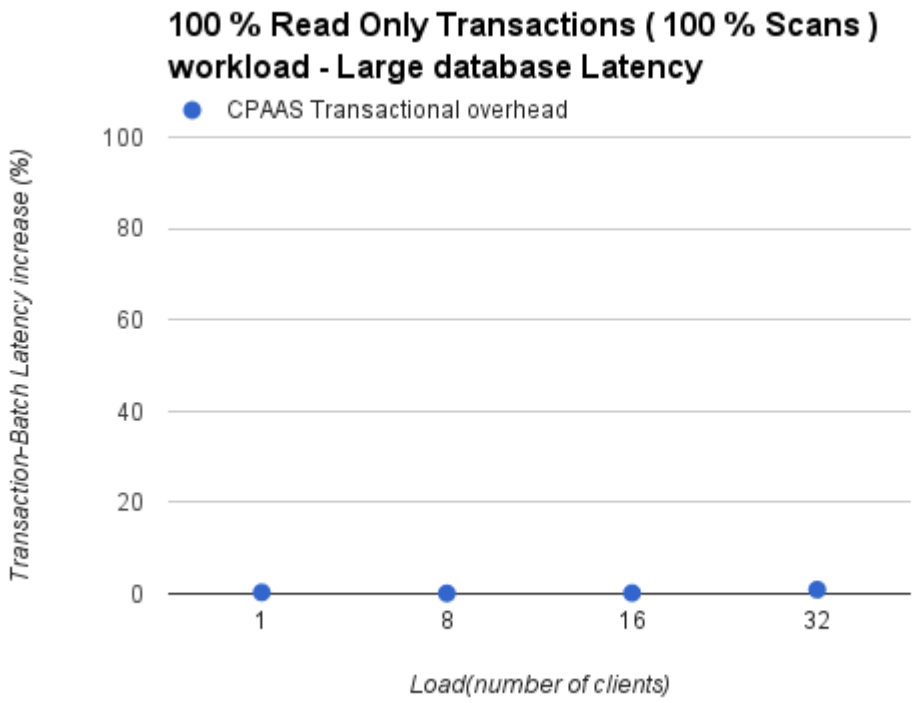

Figure 9 H-Eutropia - 100 % Read Only Transactions (100 % Scan) workload - Large database -Transactions-Batches Latency.

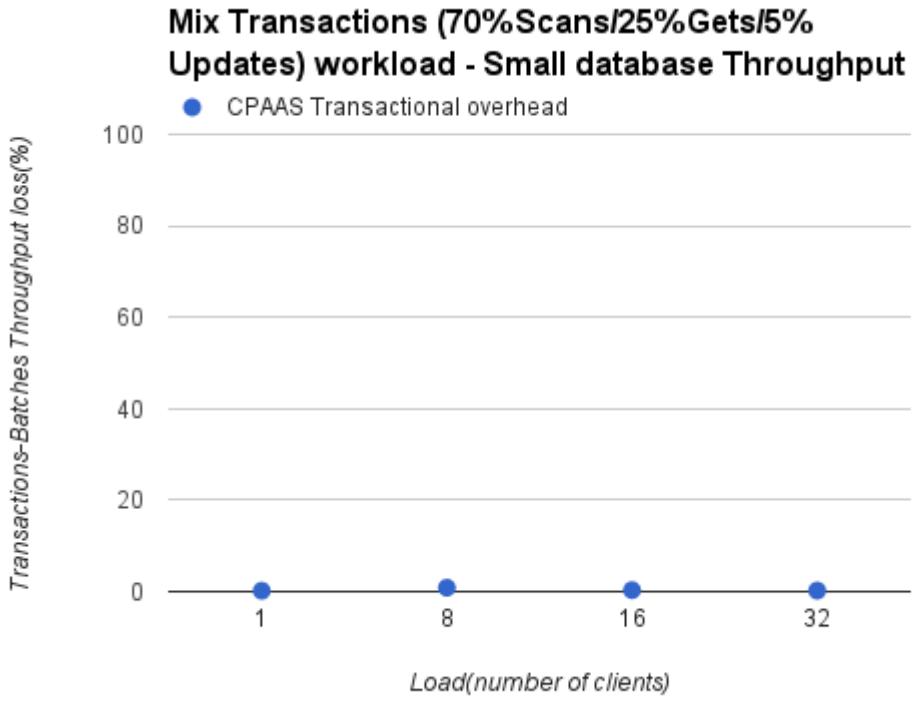

Figure 10 H-Eutropia - 100 % Update Transactions (70% scan/25% Gets/ 5% Updates) workload - Large database - Transactions-Batches Throughput.

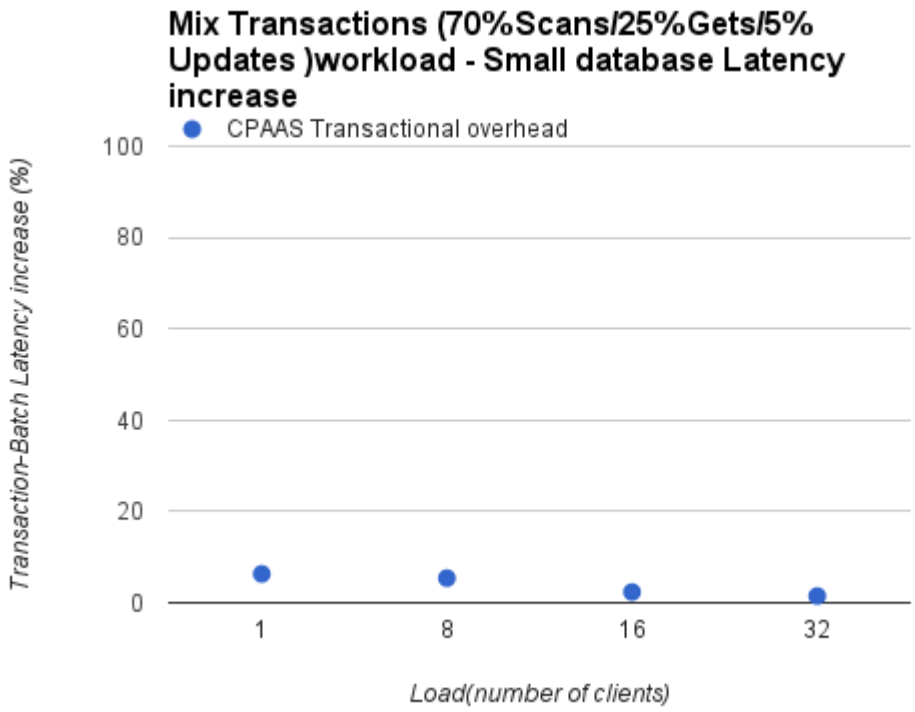

Figure 11 H-Eutropia -100 % Update Transactions (70% scan/25% Gets/ 5% Updates) workload - Small database - Transactions-Batches Latency.

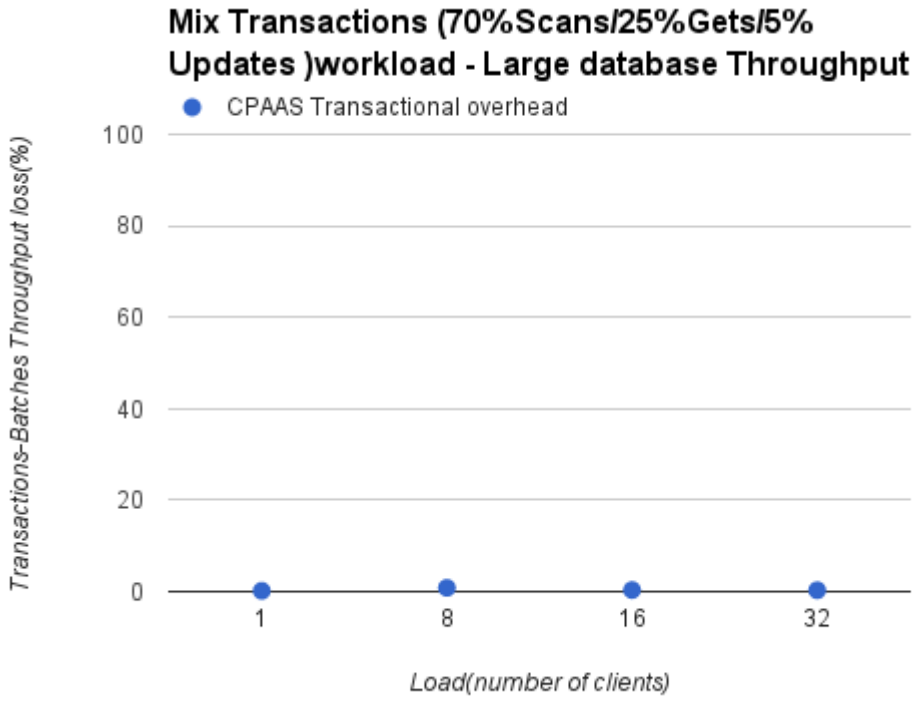

Figure 12 H-Eutropia - 100 % Update Transactions (70% scan/25% Gets/ 5% Updates) workload - Large database - Transactions-Batches Throughput.

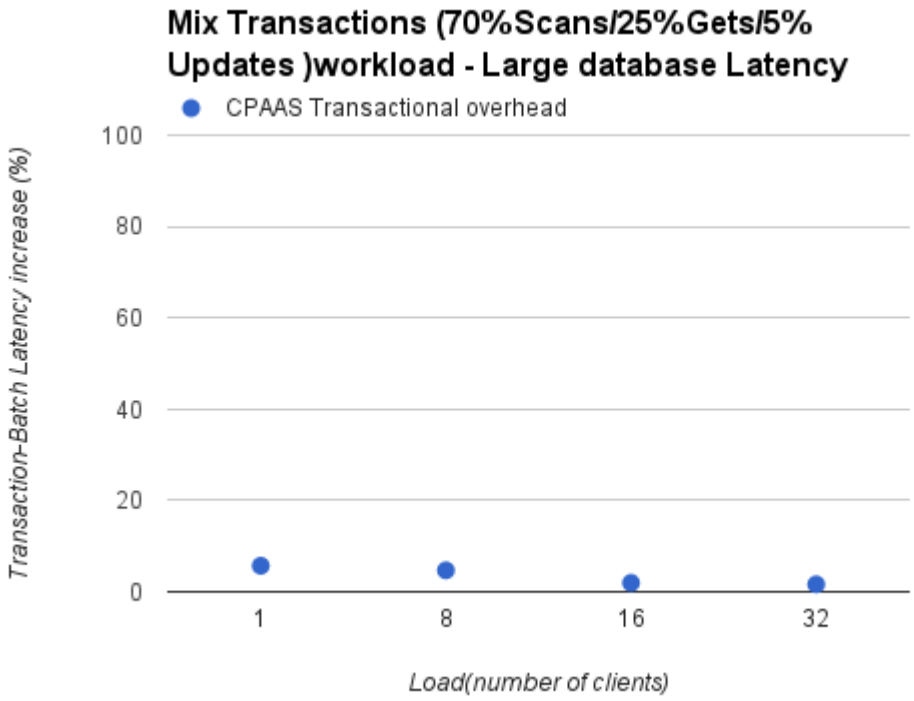

Figure 13 H-Eutropia - 100 % Update Transactions (70% scan/25% Gets/ 5% Updates) workload - Large database - Transactions-Batches Latency.

3.1.4. Conclusions

As we notice from the above figures the impact in performance drop and latency increase of CPAAS overhead is below 5% in all cases. For the read only queries the interaction with the Holistic Transaction Manager(HTM) is minimal. Read only queries interact with the HTM to read a timestamp and this takes place out of the data path.

In the case of mix workload where are updates are present the overhead is less than 10% in all cases. This impact is low due to following reasons. Conflict management takes place asynchronously while the transaction is being executed so the conflict management cost is amortized for a large number of operations. Also the write set is sent to the logger service. Loggers write and persist data in a log which is a fast operation for various kind of devices (hard disks, ssd). CPAAS defers the actual transfer of the write set to the actual store to increase performance.

3.2. LeanXcale

3.2.1. Setup

The LeanXcale data store has been deployed in two machines and we run the YCSB benchmark client in another machine. All machines are equipped with 12-core Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz, 128GB of RAM and a directly attached 0.5 TB SSD hard disk. The machines are connected through a standard gigabit ethernet network. All machines are using Ubuntu 12.04 and Oracle JRE 1.7.

In one node of LeanXcale we deploy the master components and HTM servers: Zookeeper, HBase Master, HDFS Namenode, Snapshot Server, 2 Conflict Managers and

Commit Sequencer Manager. In the other machine we run 1 query engine instance, 4 HBase RegionServer instances, 1 HDFS DataNode instance, 1 Logger instance, 2 Conflict Managers and 1 Local Transaction Manager.

3.2.2. Database

The source data is the default dataset of this benchmark, with different numbers of rows where each row has one key column with a length of 100 bytes and 10 value columns with 100 bytes each. We used the following different dataset sizes:
A small dataset with 1 million rows resulting in a 1.0 GB file if the data is stored as a CSV file and a 1.1 GB LeanXcale database.
A big dataset with 20 million rows resulting in a 20 GB file if the data is stored as a CSV file and a 22 GB LeanXcale database.

As LeanXcale is a SQL database with a regular JDBC interface to create the data sets for the experiments, the default YCSB schema was used, namely 1 table with 11 columns of the data type STRING, where each column has a variable data size of maximum 255 characters. In addition, the first column is a primary key.

3.2.3. Evaluation

We ran the read-only workload using both data sets (i.e. 1M and 20M rows), and with increasing number of clients (i.e. from 1 to 200). A fixed batch size of 100 (i.e. 100 operations per transaction) is used.
The results are shown in the graphs below. For each set of experiments, we present i) increase of latency (in %) of CPAAS vs. Native; ii) throughput loss (in %) of CPAAS vs. Native.

3.2.3.1. Read-Only Workload

Our read-only workload contains 100% READ operations. Rows are selected according to the uniform distribution. For web applications, this workload represents use cases, such as user profile selection.

Small Size Dataset

Figure 14 and Figure 15 show the observed relative overhead for throughput and latency of read only workload using a small size dataset. The throughput increases until 150 concurrent clients, so we stopped increasing the number of clients at 200.

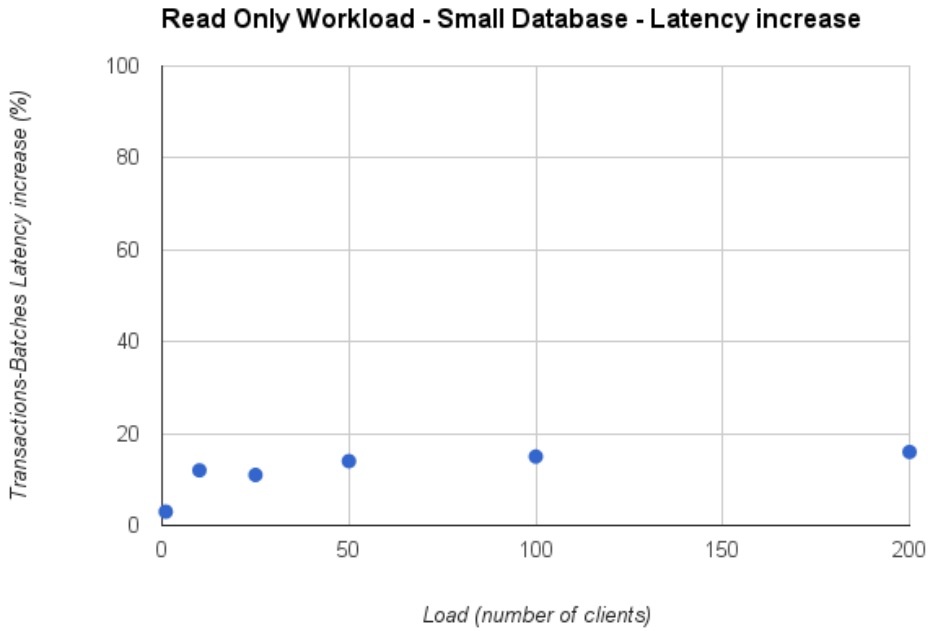

Figure 14 LeanXcale – Read Only workload – Small database -Transactions-Batches Latency.

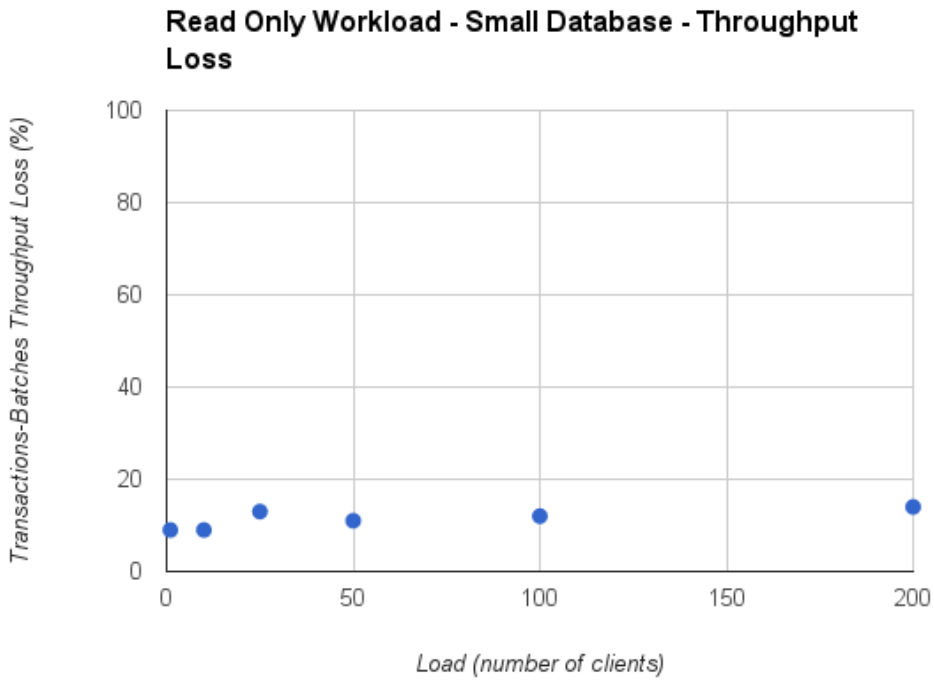

Figure 15 LeanXcale – Read Only workload – Small database -Transactions-Batches Throughput

Big Size Dataset

Figure 16 and Figure 17 show the observed relative overhead for throughput and latency of read only workload using a big size dataset. The throughput increases until 50 concurrent clients, so we stopped increasing the number of clients at 100.

Figure 16 LeanXcale – Read Only workload –Big database -Transactions-Batches Latency

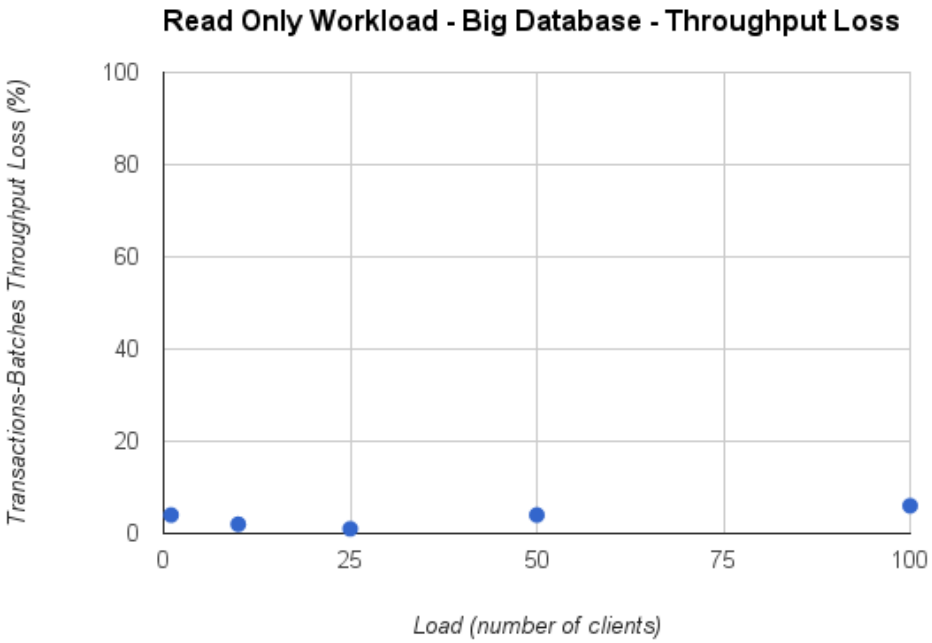

Figure 17 LeanXcale – Read Only workload – Big database -Transactions-Batches Throughput

3.2.3.2. Mix Workload

Our mix workload contains 30% READ, 20% UPDATE,10% READ and MODIFY, and 40% SCAN operations. Rows are selected according to the uniform distribution. On a single scan the maximum number of records to access is 300 and the distribution used to choose the number of records to access on a scan is uniform. This is a workload with 30% of update operations representative of several OLTP applications.

Small Size Dataset

Figure 18 and Figure 19 show the observed relative overhead for throughput and latency of mix workload using a small size dataset. The throughput increases until 50 concurrent clients, so we stopped increasing the number of clients at 100.

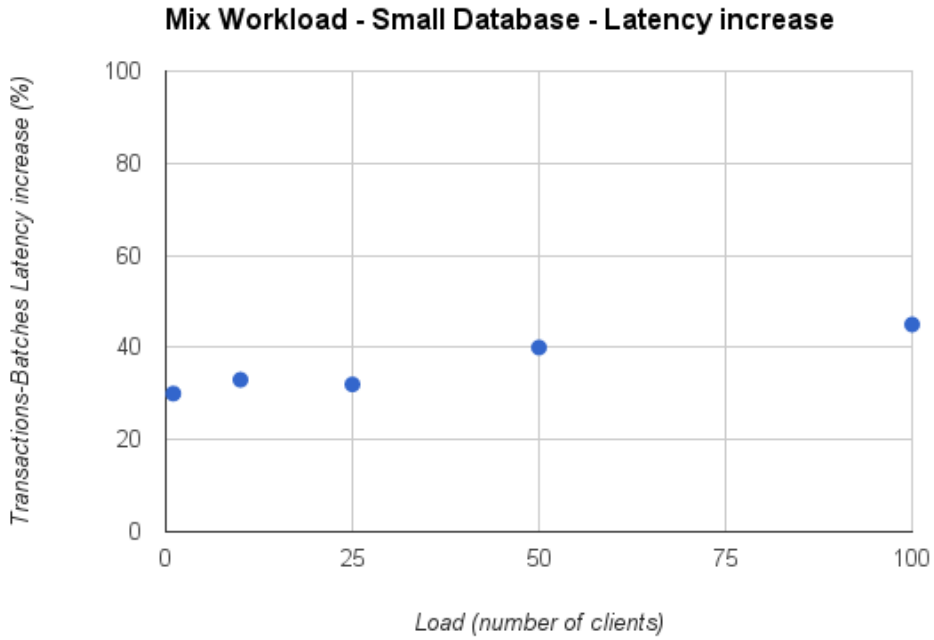

Figure 18 LeanXcale – Mix workload – Small database -Transactions-Batches Latency

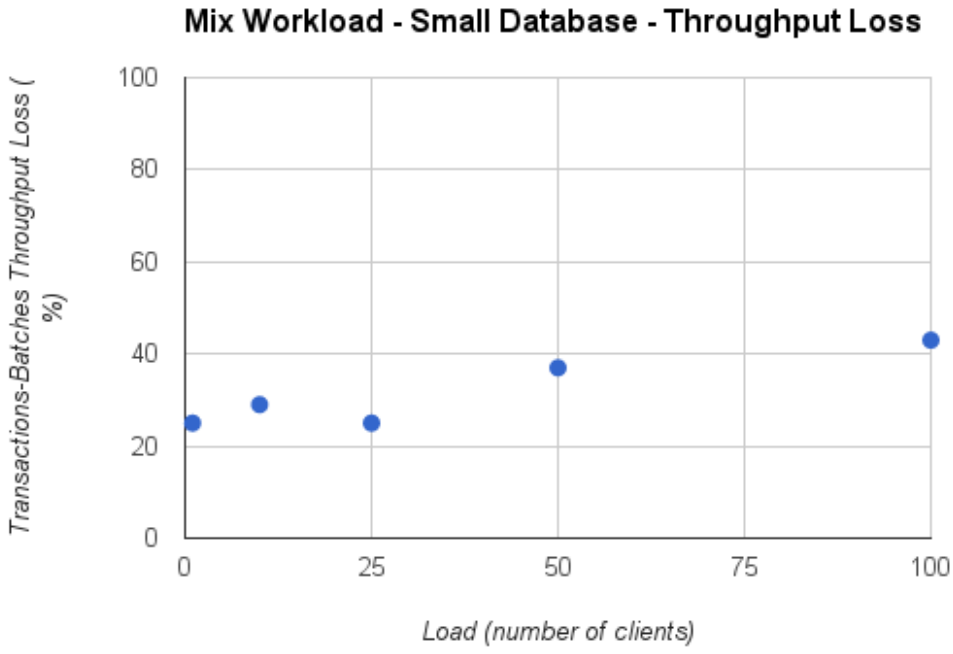

Figure 19 LeanXcale – Mix workload – Small database -Transactions-Batches Throughput

Big Size Dataset

Figure 20 and Figure 21 show the observed relative overhead for throughput and latency of mix workload using a small size dataset. The throughput increases until 25 concurrent clients, so we stopped increasing the number of clients at 50.

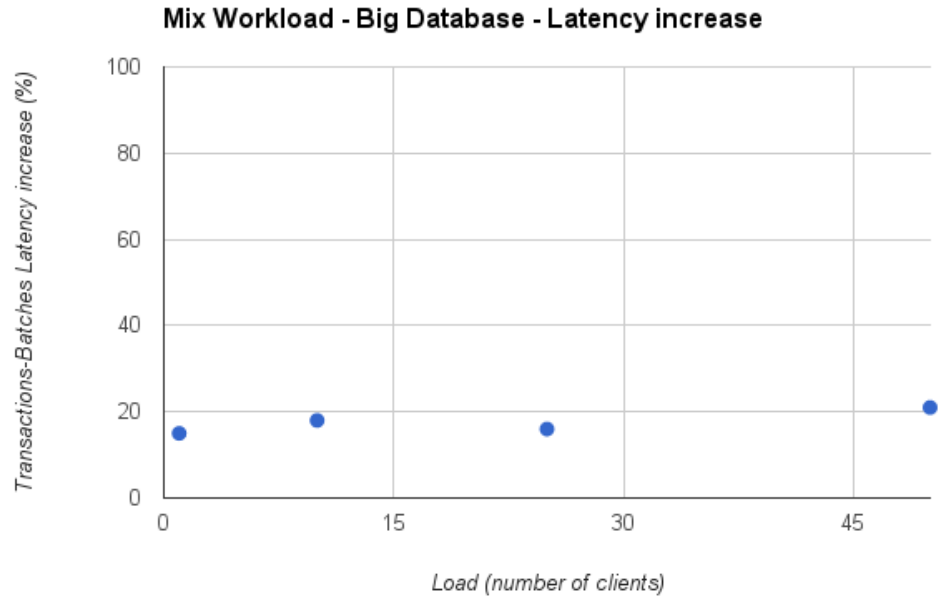

Figure 20 LeanXcale – Mix workload – Big database -Transactions-Batches Latency

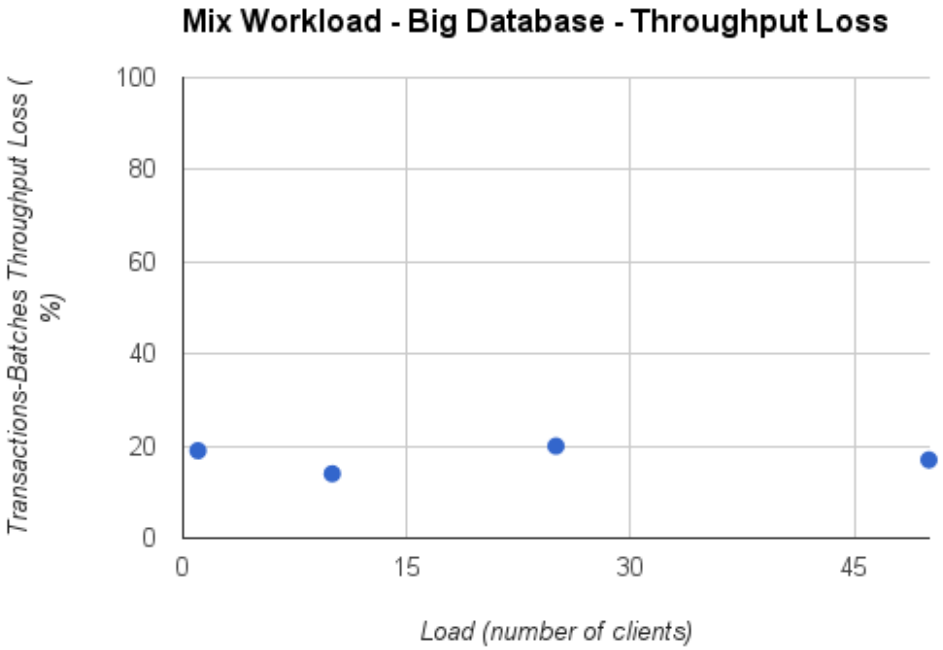

Figure 21 LeanXcale – Mix workload – Big database -Transactions-Batches Throughput

3.2.4. Conclusions

During the evaluation, we have observed that the overall performance of both CPAAS and native experiments scales nicely with increasing number of clients.

As expected, when CPAAS transaction management is used, there is some noticeable overhead. Moreover, we can see that the relative overhead (latency increase/throughput loss) on big database is lower as overhead absolute value is independent of the database size and absolute value of native latencies of operations increase due to database not fitting in memory.
Furthermore, as expected the overhead is higher for mix workload than for read only workload. For read only workload there is still some overhead as, contrary to some data stores without transactional support and that implement it on client side, LeanXcale already has built-in transactional support and thus the LeanXcale server has still to be contacted when transaction finishes and it then checks that is a local read only transaction. For update transactions, the overhead is higher and more than regular native transactions as it implies additional work on server side, to allow the transaction already guaranteed to commit to be rollback (for example due to rollback of another data store). Moreover, it also implies additional work on the local HTM which is absent in native transactions.

3.3. MongoDB

3.3.1. Setup

For the YCSB benchmark of MongoDB we have use the following setup:

- A node for MongoDB datastore, version 3.2.8, with 4 Intel processors @2,1GHz, 16GB of RAM, 40GB of storage disk, using Debian 8.3
- A node for the YCSB benchmark client with 4 Intel processors @2,1GHz, 8GB of RAM, 20GB of storage disk, using Ubuntu 14.04.4
- A node for the zookeeper, the monitoring and the configuration management components, required by the holistic transactional manager with 4 Intel processors @2,1GHz, 8GB of RAM, 20GB of storage disk, using Ubuntu 14.04.4
- A node for the snapshot manager and commit sequencer components, required by the holistic transactional manager with 4 Intel processors @2,1GHz, 8GB of RAM, 20GB of storage disk, using Ubuntu 14.04.4
- A node with 4 conflict manager components, required by the holistic transactional manager with 4 Intel processors @2,1GHz, 8GB of RAM, 20GB of storage disk, using Ubuntu 14.04.4

All nodes have been deployed under a private network, connected with a standard gigabit network

3.3.2. Database

The source data for the YCSB benchmark uses the default schema, where all data is stored in a single collection. However, YCSB uses a randomly generated 8 bytes *long* value for the key of each record. Even if MongoDB can support *long* values for primary keys, it is recommended to use its internal *ObjectId* class for storing primary key values, which require 12 bytes. For each auto-generated *long* value provided by YCSB, we serialize it to a byte array and append a randomly generated 4 bytes' value. Then, we construct the *ObjectId* object which is used for storing the primary key of each document. Moreover, as MongoDB is a document-based data store, in real-life environments it is expected that documents will not only contain fields used for storing several attributes

of an entity (i.e name, address, etc.) which usually require just a couple of hundred of bytes per each fields, but will also store information that includes other bigger-size documents (i.e json files containing configuration parameters, bytes of a picture, etc) which may require several Kbytes of storing space. Due to this and to the fact that in YCSB all fields must contain the same size, we also increased the column size to 1500 bytes, with 10 columns per record, thus simulating the scenario where a document will contain several small size fields, and one field which contains information which require several Kbytes of space.
Regarding indexing, regarding the benchmark of vanilla MongoDB which will be used as a reference to evaluate our transactional solution, we only used the default indexing on the documents primary key, which is the default behaviour of the data store. The Transactional MongoDB implementation uses instead additional indexes, required for the MVCC provision, as explained in the relevant D5.2[9] [5] and D5.3[10] [6] restricted deliverables. These indexes involve the primary key of each document and the additional metadata that uniquely characterize each version (i.e commit timestamp and transaction identifier).

We generated the data source with the YCSB benchmarking tool. For the transactional scenario, we additionally implemented a YCSB bulk load method on the *MongoClient* in order to quickly import data with initial '0' value timestamps for all documents. We evaluated vanilla MongoDB against transactional MongoDB with two scenarios: using a small-size database that will entirely fit in the data store server memory, and using a big-size database that cannot fit in the data store server memory, thus I/O will be exercised on the server-side. For the big-size setup, we initially inserted 1.500.000 documents which resulted into 20,971GBs of data and 24,047GBs of storage space in the server node. For the small-size setup, we initially inserted 200.000 documents which resulted into 1,398GBs of data. However, after receiving the benchmark results, we identified that the bandwidth of the network was saturated very early when gradually incrementing the number of threads, due to the big size of each document and the very low latencies still observed in higher throughputs. As a result, regarding the small size datasets, we decreased the column size to 750Kbytes, and increased the number of rows to 400.000, which resulted into 2,798GBs of data and 3,203GBs of storage space. The results of the evaluation are explained in the following subsection.

3.3.3. Evaluation

Each benchmark is configured to execute 10 consecutive operations per each transaction. We evaluated two different scenarios for each data source size: read-only transactions that contain 100% of read operations, and transactions that contain a mix of 90% read and 10% write operations. The read-only scenario was used to evaluate the performance overhead of the MVCC provision, as there are no data modification operations and the HTM overhead remains minimum. The second scenario was used to evaluate real-life use of the transactional MongoDB, where the dominant read operations occur simultaneously with operations that alter the state of the data store, thus requiring

[9] Restricted D5.2 No-SQL data store implementation (initial version) presents a the technical design of the data store . The deliverable focus in the multi-versioning implementation on the persistent storage, required for the integration with the HTM to support transactional semantics.
[10] Restricted D5.3 No-SQL data store implementation (final version) presents a technical description and the technical implementation details of the data store. The deliverable focus in the multi-versioning implementation on the persistent storage, required for the integration with the HTM to support transactional semantics.

the HTM to execute the commit protocol, log the modified data to ensure the durability property and safely commit to the data store.

For each benchmark, we observed the average latency and overall throughput of the batch of operations (even if they are bracketed into a transaction or not), and the average latency of the operations involved in the benchmark (i.e latency of reads, latency of writes etc.). We first loaded the data store with the corresponding data, as described in the previous subsection, and then launched the benchmark to generate traffic using 1 thread and unlimited throughput. By doing so, we were able to identify the maximum throughput that each thread can reach and to note the corresponding latencies. Then, we gradually increased the number of threads by adding 5 more until we reached 30 threads. For each benchmark test we used a single node for generating traffic via the YCSB client, as described in the previous subsection. The results of the benchmarks can be found below.

3.3.3.1. Read-Only Operations -- Small Size Dataset

The following diagrams show the observed relative overhead for throughput and latency regarding transactions per second, using a small size dataset of 3,2GBs, as described in the previous subsection.

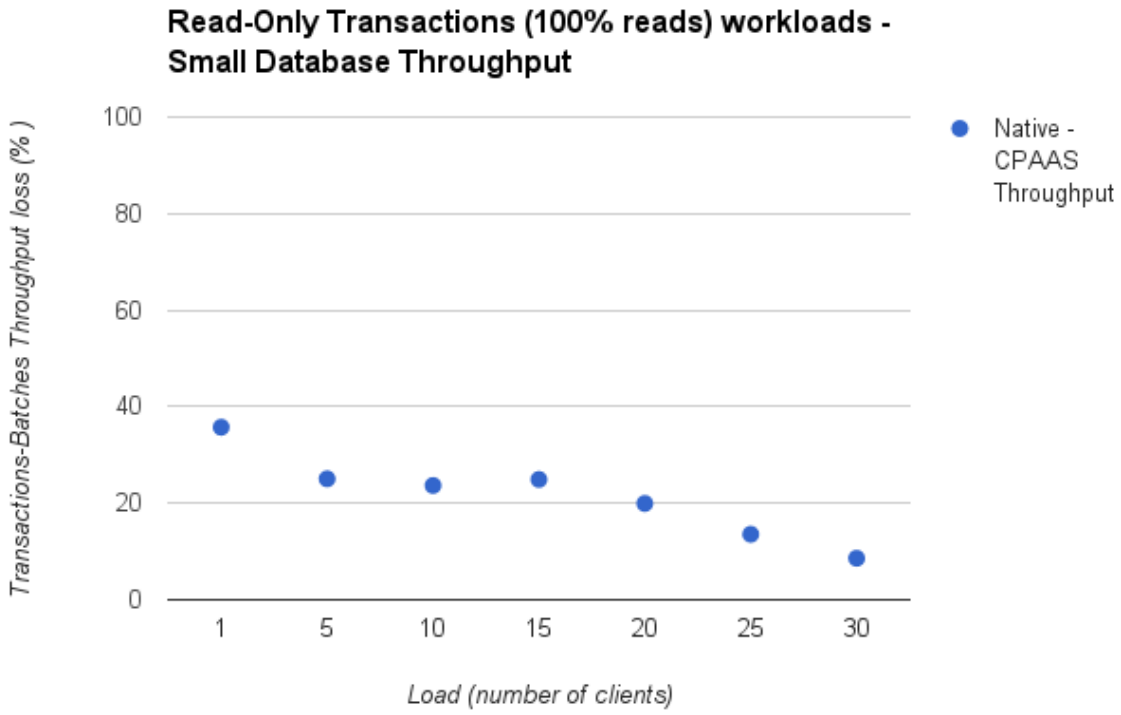

Figure 22 MongoDB - Read-Only Transactions (100% reads) workloads- Small Database Throughput

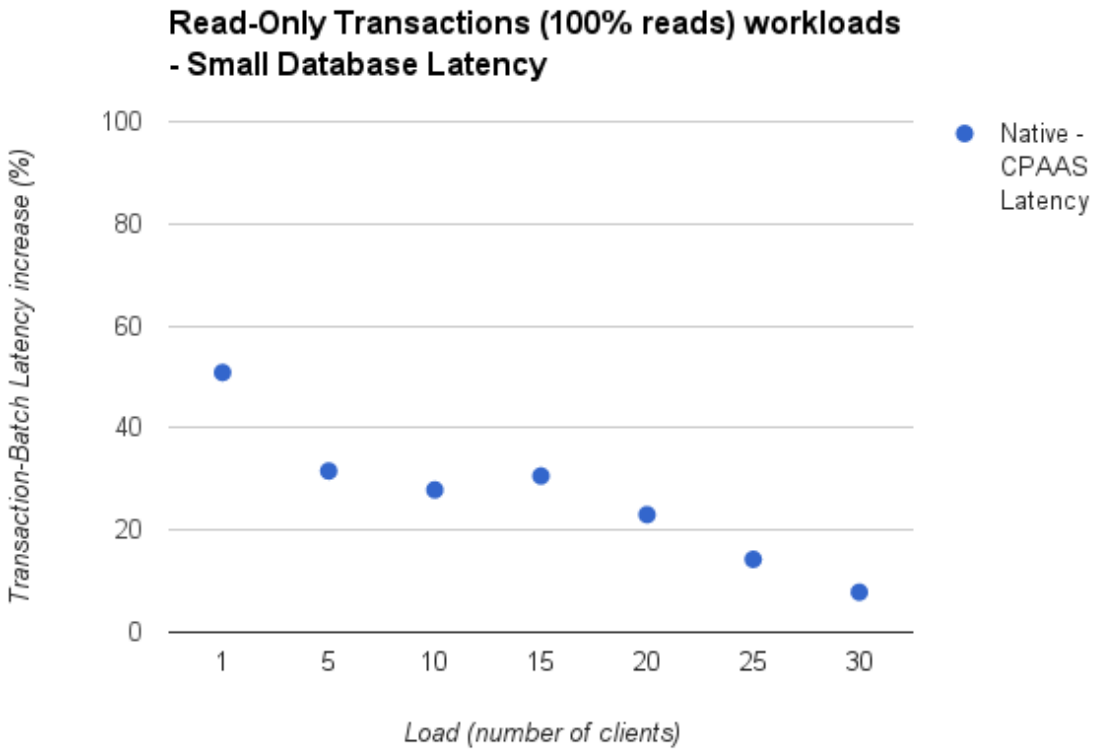

Figure 23 MongoDB - Read-Only Transactions (100% reads) workloads- Small Database Latency

Regarding throughput, Transactional MongoDB gives a smaller throughput, due to the increased overall latency which is caused by the MVCC provision. However, when MongoDB reaches an increased number transactions per second, the throughput remain constant, and this is the maximum throughput that can be achieved by a single client node. This is caused by the saturation of the network bandwidth. As explained in the previous subsection, initial tests with bigger size documents (1500 Bytes per column) caused the saturation of the network to occur with relatively small workload. We decreased the size of columns to 750 Bytes, and the saturation is now observed in bigger workloads, when it reaches its upper bounder. Due to this, we conclude that when using a single node for the ycsb client, the resources are saturated with more than 20-25 concurrent threads, both in vanila and transactional MongoDB. As a result, no safe results can be obtained for these numbers.

Regarding the latency, we observe a constant overhead in all cases, which is the cost of the MVCC provision. We remind that the MVCC feature is implemented in the client side for MongoDB and the latter is unaware of this implementation, as it simply retrieves documents accordingly. In small size datasets like this, where all data can fit in memory, MongoDB operates very fast, as there's no need for any I/O operation in the server-side, and the overall latency costs a few milliseconds. In these cases, even if the overhead added in the client side for the MVCC is small, the result is an significant increase of the relative value of the difference. However, in larger data sets, it is expected that the relative increase will be minimal.

3.3.3.2. Read-Only Operations -- Big Size Dataset

After the benchmarking with a small size dataset, which can easily fit in memory of the data store, we continued our evaluation process using a big size dataset of 24GBs. The following figures shows the observed relative overhead in terms of throughput and latency.

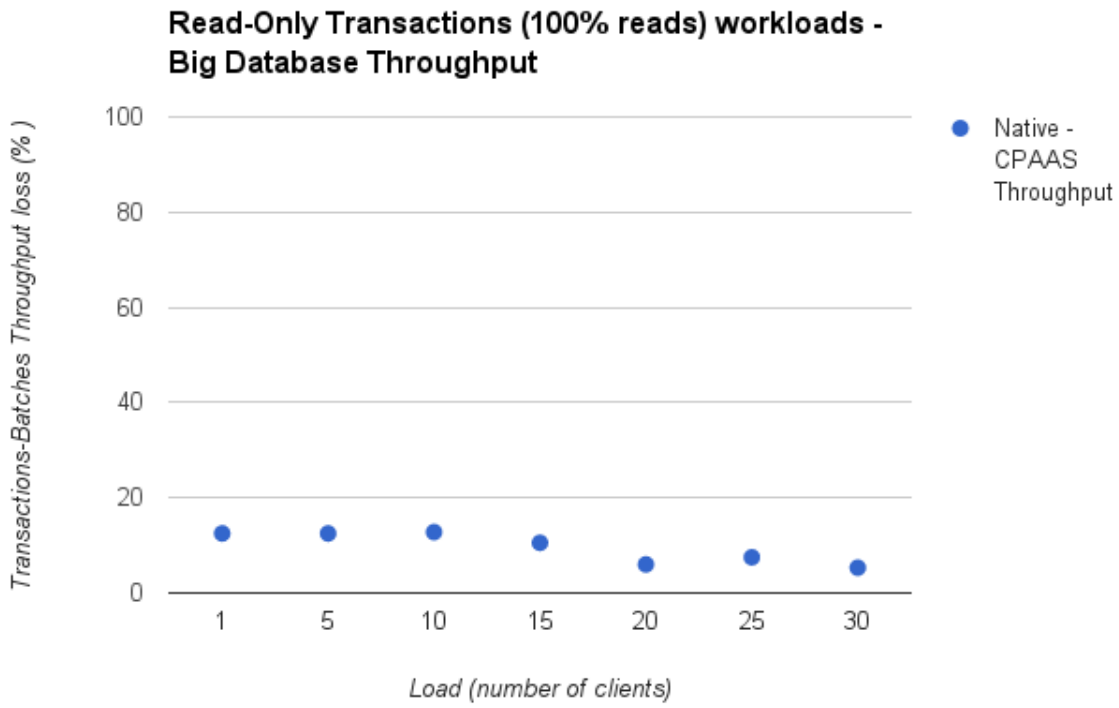

Figure 24 MongoDB Read-Only Transactions (100% reads) workloads- Big Database Throughput

Figure 25 MongoDB - Read-Only Transactions (100% reads) workloads- Big Database Latency

Regarding the observed latency of the transactions, we observe a constant difference of a slightly increased overhear. As expected, by increasing the size of the data set, MongoDB has to use intensive I/O operations on the server side in order to retrieve the result set, which increases the overall latency of the operations significantly. The additional constant overhead occurred by the MVCC provision now results in a low increase of relative value of the difference of the latency. This can be also concluded by observing the throughput of the transactions, with the curve of the throughput achieved by the transactional MongoDB to be very close to the curve of the vanilla MongoDB.

3.3.3.3. Mixed Transactions -- Small Size Dataset

After evaluation the MVCC provision using read-only operations, we then run the tests using the same datasets, but changing the proportion of operations. We used a new workload which consists of dominant (90%) read operations, and 10% insert operations, which is usually used in real-life scenarios. The following figures shows the relative overhead of throughput and latency of the transactions.

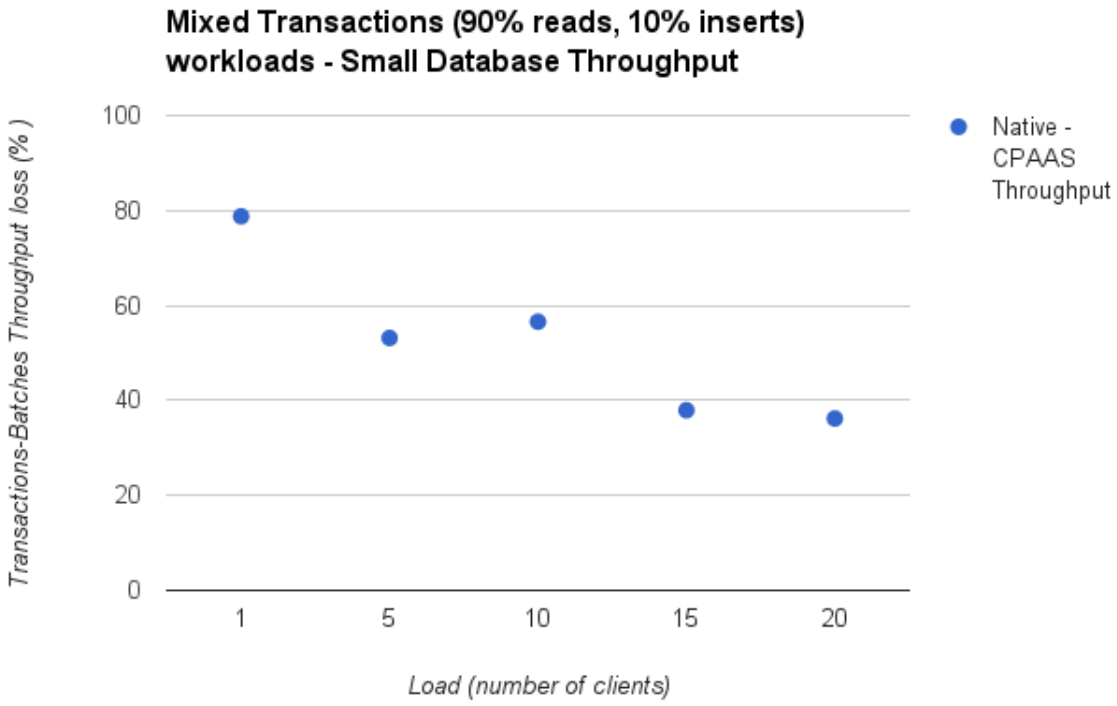

Figure 26 MongoDB - Mixed Transactions (90% reads, 10% inserts) workloads- Small Database Throughput

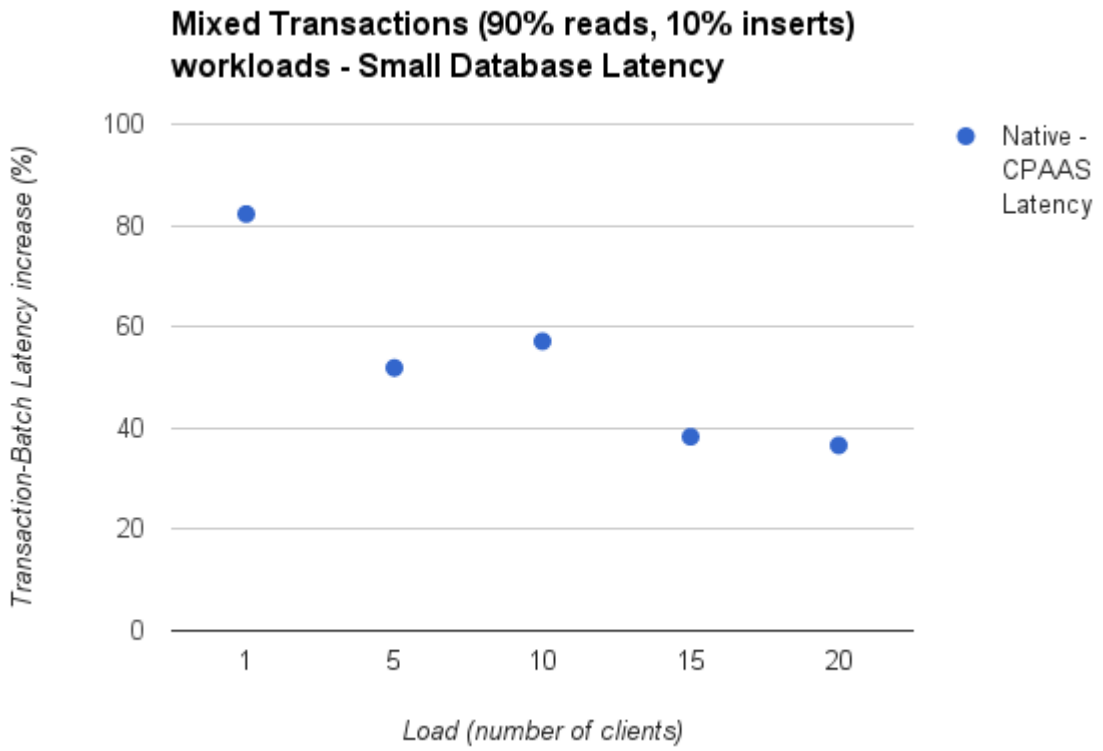

Figure 27 MongoDB - Mixed Transactions (90% reads, 10% inserts) workloads- Small Database Latency

Using workloads which also contain insert operations, we now observe a constant, but higher overhead of the latency. Regarding the throughput, vanilla MongoDB increases its throughput proportionally as the number of threads increases, until it reaches the saturation of the client resources, as explained previously. Transactional MongoDB starts saturating the resources earlier, before reaching the same throughput, as it involves additional operations during the commit phase, compared with vanilla MongoDB which simply inserts a record.

3.3.3.4. Mixed Transactions -- Big Size Dataset

The next benchmark uses the same workload (90% read operations, 10% write operations) with a big size dataset of 24GBs that cannot fit in memory. The following figures show the relative overhead of throughput and latency of the transactions.

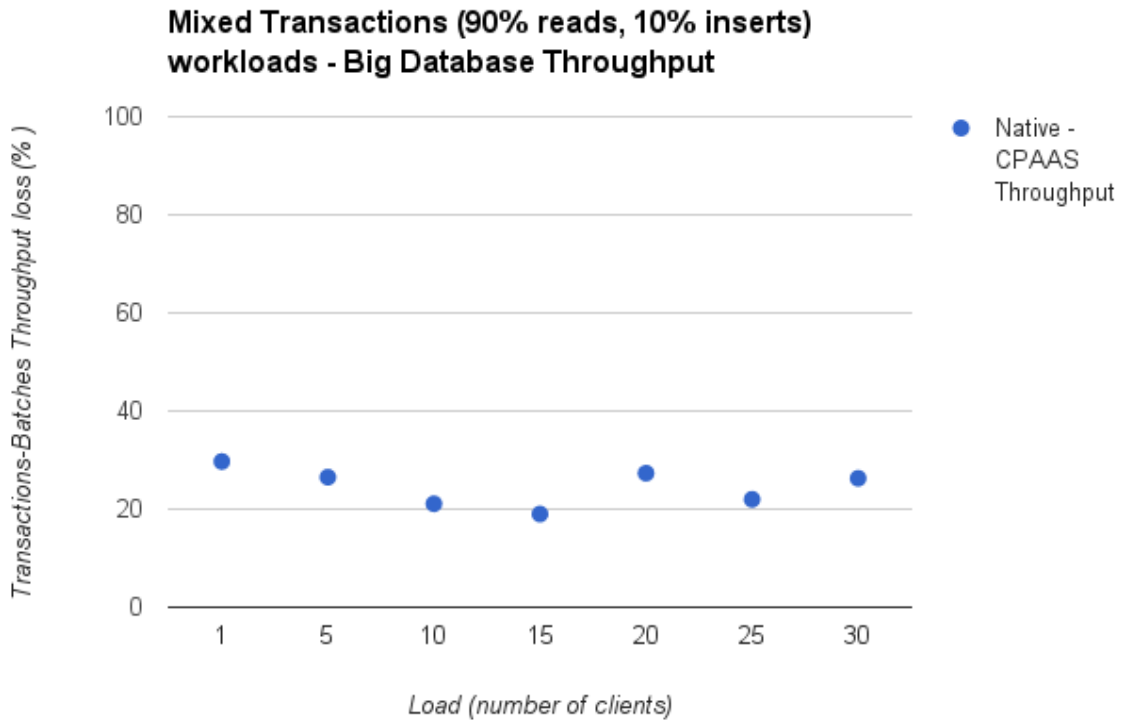

Figure 28 MongoDB - Mixed Transactions (90% reads, 10% inserts) workloads- Big Database Throughput

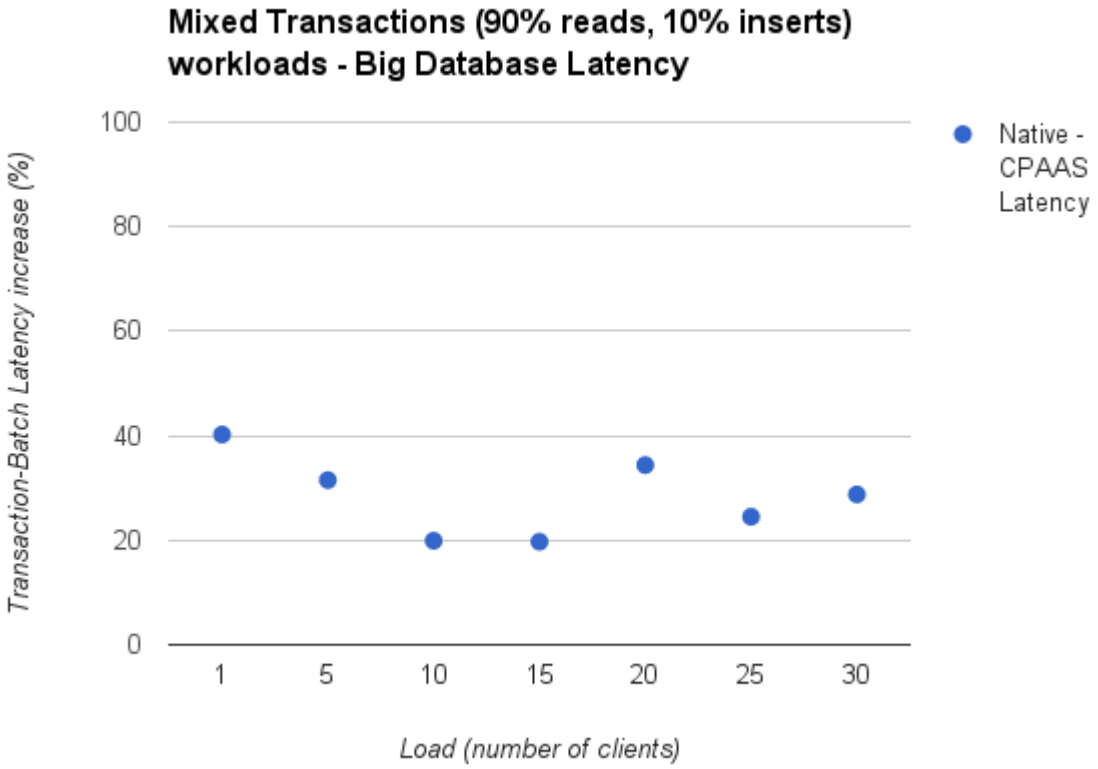

Figure 29 MongoDB - Mixed Transactions (90% reads, 10% inserts) workloads- Big Database Latency

From the figures, we observe a constant overhead of the latency, which is the cost of the transactional semantics that are provided by our implementation. This overhead is

relatively significant when having workloads that produce low traffic (i.e having 1 or 5 threads in the ycsb client), however, as the traffic increases, so does the observed latency proportionally, which causes the relative difference to drop. We also observe that the throughput increases proportionally as the number of thread increases, however the resources on the client side are saturated sooner than vanilla MongoDB. This is caused by the additional operations that our implementation uses in order to provide transactional semantics. Using a second YCSB client for producing traffic, thus, achieving the same overall throughput, it is expected that the latency will continue to increase proportionally, thus the relative overhead will be dropped even more.

3.4. MonetDB

3.4.1. Setup

For the evaluation of the YCSB benchmark on MonetDB, we have used three machines with identical hardware. Each machine is equipped with:

- One Intel Xeon E5-2650 v2[11]
- 256GB RAM
- 5.4 TB HDD (3x SW RAID0)
- 1GB/s Ethernet
- 40GB/s Infiniband

The experiments machines are connected to each other in a LAN cluster, through both the Ethernet and the Infiniband connections. All three machines run Fedora 24.
One machine runs the HTM branch of the MonetDB server code. The HTM branch of MonetDB has been forked off the main MonetDB development branch (called "default"), and it contains CoherentPaaS specific feature extensions.
One machine runs all servers needed by HTM, include zookeeper (version 3.4.5), and configuration manager, conflict manager, snapshot server and commit sequencer (HTM servers). Finally, one machine runs the YCSB client.

3.4.2. Database

To create the data sets for the experiments, the default YCSB schema was used, namely 1 table with 11 columns of the data type STRING, where each column has a fixed data size of 100 characters. In addition, the first column is a primary key.

Two data sets have been created

- The smaller data set has 20 million rows, which results in 20GB data in the CSV format, or a 7.6GB database once loaded into MonetDB.
- The bigger data set has 100 million rows, which results in 100GB data in the CSV format, or a 37GB database once loaded into MonetDB.

MonetDB only temporarily creates indices at runtime to accelerate query execution, hence, no additional storage is needed for indices.

3.4.3. Evaluation

Because MonetDB does not allow concurrent updates on the same table, we have separated read-only and updating transactions into two different workloads. With the

[11] http://ark.intel.com/products/75269/

read-only workload, we vary the size of the database and the number of clients. With the updating workload, we vary the batch size (i.e. number of operations in one transaction).

3.4.3.1. *Read-only workload on small and big data sets*

Our read-only workload has the same definition as Workload-C of the YCSB benchmark specification[12], which contains 100% READ operations. Data are selected according to the Zipfian distribution. For web applications, this workload represents use cases, such as user profile selection. Among the five standard workloads defined by the YCSB benchmark, the read-only workload most closely resembles the target applications of MonetDB.

We ran the read-only workload using both data sets (i.e. 20M and 100M rows), and with increasing number of clients (i.e. from 1 to 10). A fixed batch size of 10 (i.e. 10 READ operations per transaction) is used. The results are shown in the graphs below. For each set of experiments, we present i) latency of MonetDB+CPAAS and Native MonetDB; ii) increase of latency (in %) of CPAAS vs. Native; iii) throughput of MonetDB+CPAAS and Native MonetDB; iv) throughput loss (in %) of CPAAS vs. Native.

3.4.3.2. *Results read-only workload on small database*

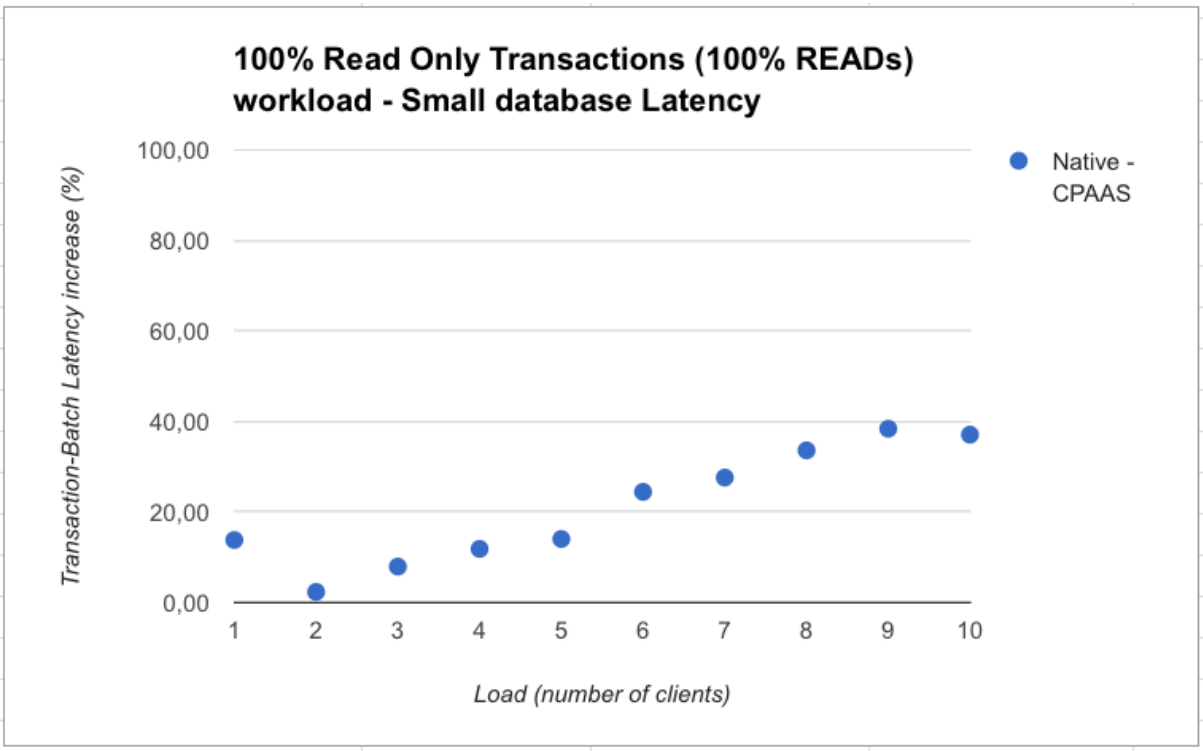

Figure 30 MonetDB - 100% Read Only Transactions (100% READs) workload - Small database -Transactions-Batches Latency increase (%).

[12] https://www.cs.duke.edu/courses/fall13/cps296.4/838-CloudPapers/ycsb.pdf

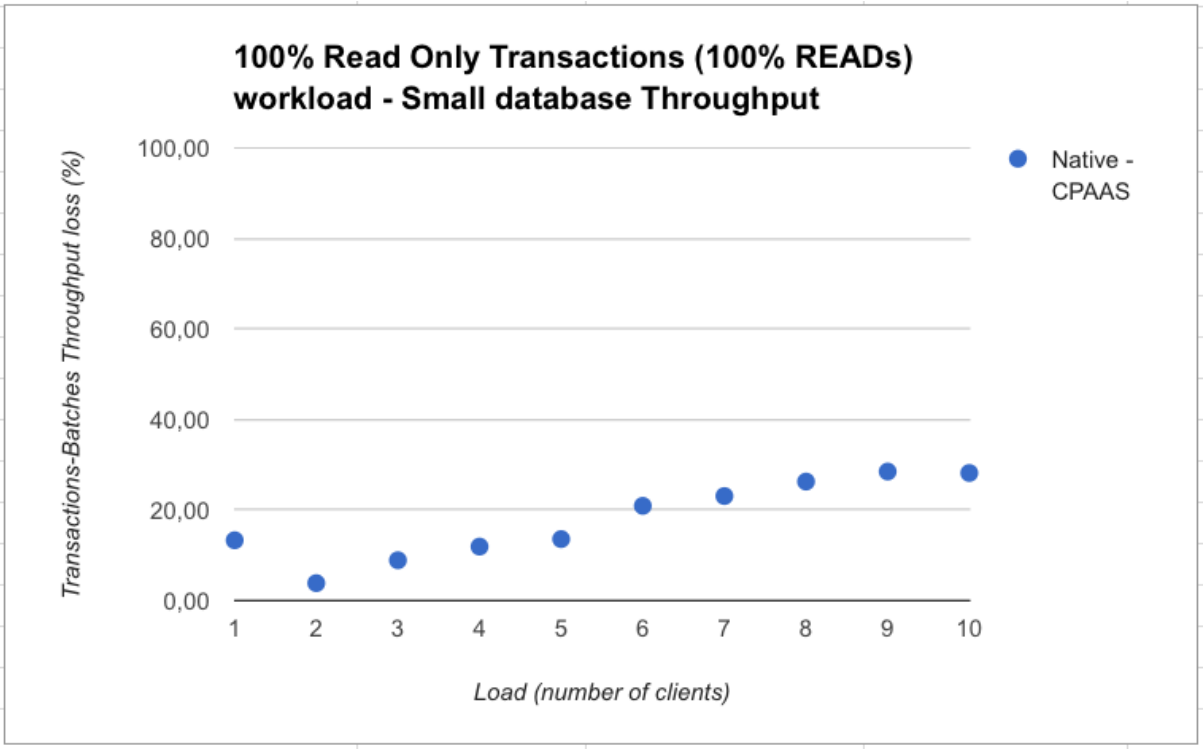

Figure 31 MonetDB - 100% Read Only Transactions (100% READs) workload - Small database -Transactions-Batches Throughput loss (%).

3.4.3.3. Evaluation read-only workload on small database

During the evaluation, we have observed that the overall performance of both CPAAS and Native scales nicely with increasing number of clients. The average latency decreases while the throughput increases. The throughput flattens with 8 or more concurrent clients, so we stopped increasing the number of clients at 10.

Figure 30 and Figure 31 compare the performance of CPAAS again the native executions. As expected, when CPAAS transaction management is used, there is some noticeable overhead. In general, the CPAAS overhead increases constantly with growing number of clients, until it flattens around 20% with 8 or more clients. With only 1 client, the CPAAS overhead is relatively large. This is most probably because the workload of 1 client is too light to keep the system resources busy. Moreover, the light workload is more prone to any fluctuation in the system than heavier workloads.

3.4.3.4. Results read-only workload on large database

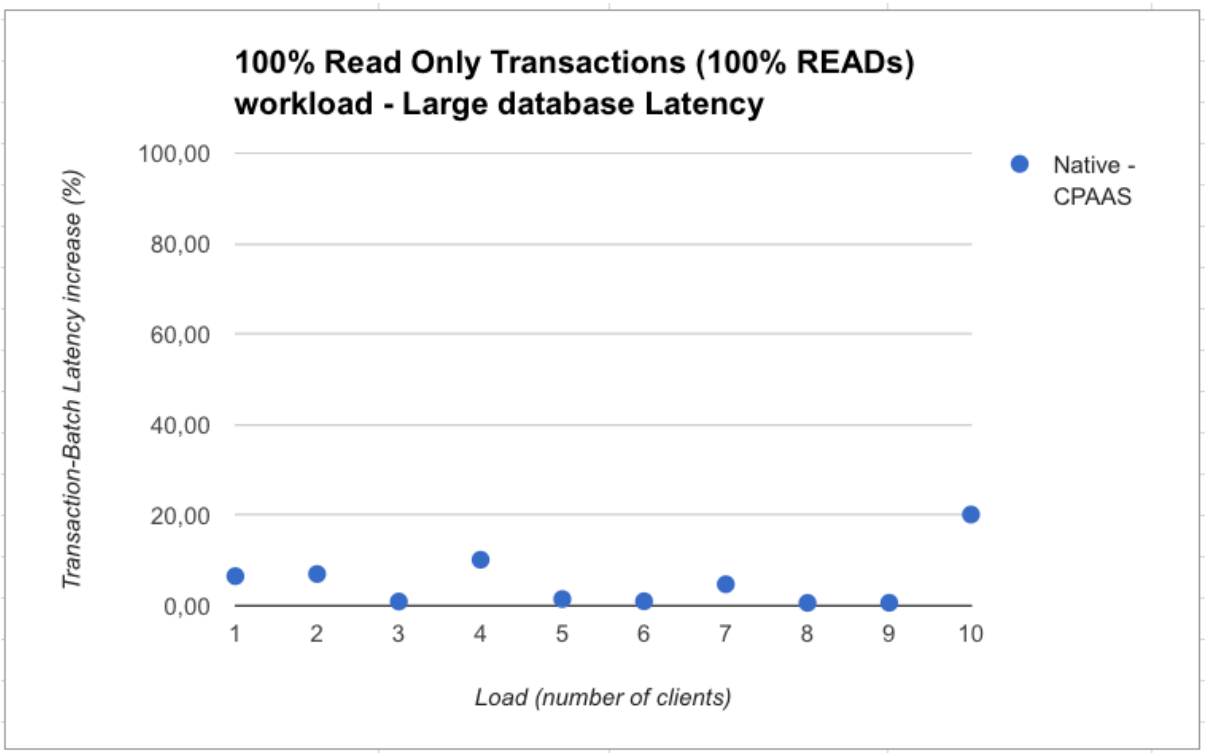

Figure 32 MonetDB - 100% Read Only Transactions (100% READs) workload - Large database -Transactions-Batches Latency increase (%).

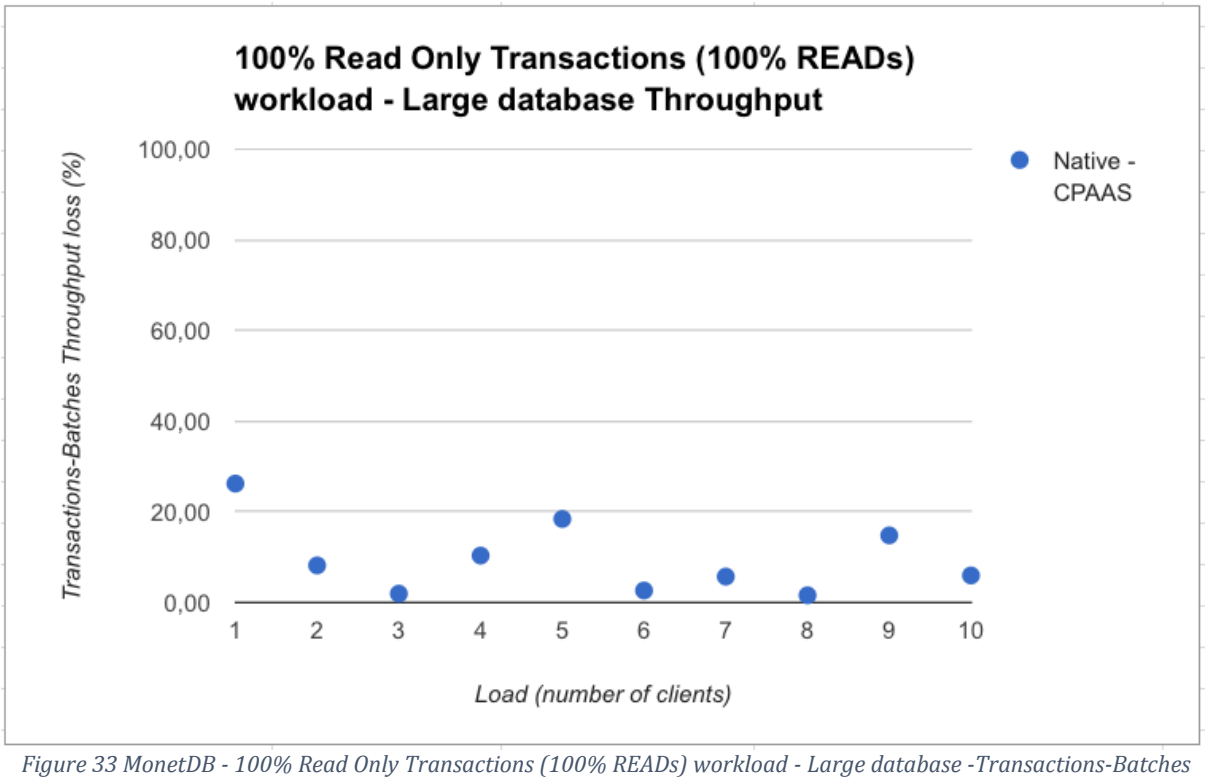

Figure 33 MonetDB - 100% Read Only Transactions (100% READs) workload - Large database -Transactions-Batches Throughput loss (%).

3.4.3.5. Evaluation read-only workload on big database

Figure 32 and Figure 33 show that the overhead of CPAAS is mostly less than 10%. This is because the CPAAS overhead is fairly constant when the data size increases, e.g. from 20M rows to 100M rows. Consequently, for the large database (i.e. 100M), the percentage of the CPAAS overhead becomes smaller.

*3.4.3.6. **Updates workload on small data set***

To study the overhead of the CPAAS transaction manager on the updating transactions, we have defined a updating workload, which contains 100% INSERT operation. This straightforward definition is chosen, because MonetDB is not optimised for the type of updates used in the YCSB benchmark, i.e. single row UPDATE or INSERT[13]. By not mixing updating operations with read-only operations, it makes it easier to understand the experiment results.

We ran the updating workload using the data set containing 20M rows, and with increasing batch size (i.e. from 1 to 100,000 INSERT operations per transaction) is used. For all updating experiments, only 1 client is used. The throughputs and latencies of the YCSB benchmark with versus without the CPAAS transaction manager are shown in the graphs below.

3.4.3.7. Results updating workload on small database

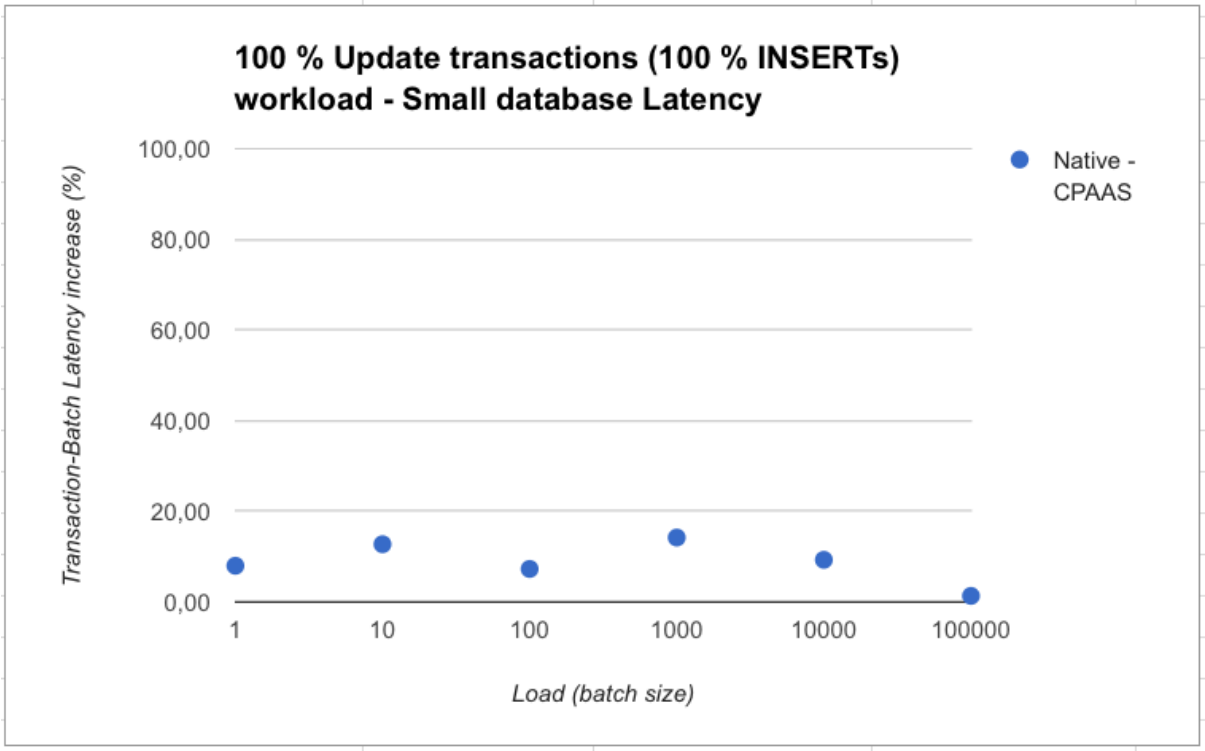

Figure 34 MonetDB - 100% Update Transactions (100% INSERTs) workload - Small database -Transactions-Batches Latency increase (%).

[13] MonetDB is only optimised for bulk updates, such as loading a small number of big CSV files, or even directly attaching binary files to the database. Even though each group of the YCSB INSERT operations are wrapped in a single transaction, they still incur a fairly big amount of overhead in MonetDB to handle the individual INSERT within the transaction.

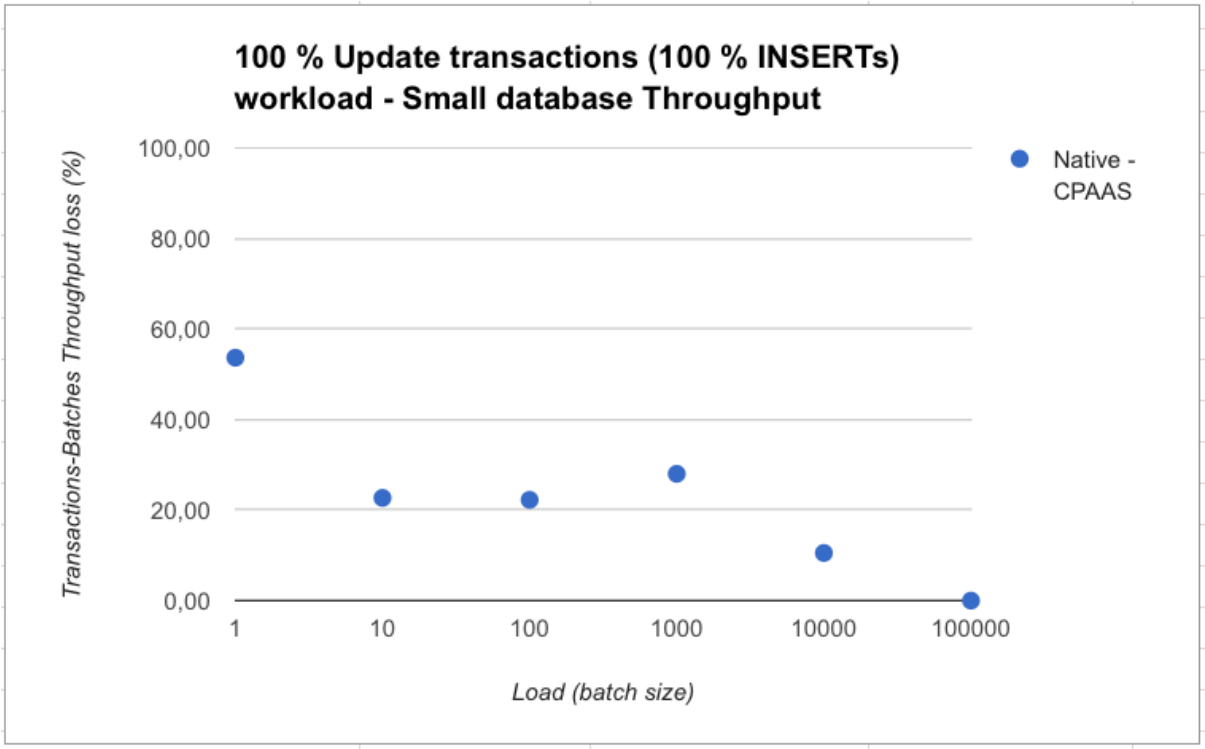

Figure 35 MonetDB - 100% Update Transactions (100% INSERTs) workload - Small database -Transactions-Batches Throughput loss (%).

3.4.3.8. Evaluation updating workload on small database

Figure 34 again shows that the latency introduced by the CPAAS transaction is fairly constant for different batch sizes. No matter the size of a transaction, the same amount of communication is needed with the CPAAS transaction managers.

Figure 35 shows that the impact of the CPAAS overhead on the throughput decreases with larger transaction sizes. This is because the CPAAS overhead per transaction is constant, as shown by Figure 26. If the transaction size increases, the execution time of the transaction will increase accordingly, while the percentage of the CPAAS overhead in the total execution time decreases.

3.5. Sparksee

3.5.1. Setup

The Sparksee implementation of the YCSB benchmark has been executed using three different servers connected through a standard gigabit ethernet network:

- A server for the Sparksee datastore with two Intel Xeon E5-2609 processors at 2.4 GHz (4 cores / 4 threads), 128GB of RAM, two 1TB hard disks and Debian 8.4.
- A server for the YCSB benchmark client with one Intel Xeon E5530 processor at 2.40GHz (4 cores / 8 threads), 32GB RAM, one 4TB hard disk and Debian 7.10.
- A server for the rest of the components: ZooKeeper, the Conflict Managers, the Snapshot Server, the Commit Sequencer and the Configuration Manager. The hardware of this server is the same as the one for the client with one Intel Xeon E5530

processor at 2.40GHz (4 cores / 8 threads), 32GB RAM, one 4TB hard disk and Debian 7.4.

3.5.2. Database

The source data is the default dataset of this benchmark, with different numbers of rows where each row has one key column with a length of 100 bytes and 10 value columns with 100 bytes each. We used the following different dataset sizes:

- A small dataset with 2 million rows resulting in a 2.1GB file if the data is stored as a CSV file and a 2.7GB Sparksee database file once loaded into Sparksee.
- A medium dataset with 20 million rows resulting in a 21GB file if the data is stored as a CSV file and a 27GB Sparksee database file once loaded into Sparksee.
- A big dataset with 35 million rows resulting in a 35GB file if the data is stored as a CSV file and a 46GB Sparksee database file once loaded into Sparksee.

Sparksee is a graph database for labeled and attributed multigraphs based on vertical partitioning and collections of objects identifiers stored as bitmaps. So, the data of this benchmark can easily be defined in Sparksee as a single node type with an unique string attribute (the key) and 10 indexed string attributes more (for the remaining 10 fields). That is how the schema for this benchmark is defined in Sparksee to have a common benchmark for all the data stores issuing a lot of transactions to be able to check the HTM overhead.

3.5.3. Evaluation

The benchmark is configured to run a transaction size of 10 random operations with the following proportions:

- 20% Insert operations: Addition of a full node with the key and the other 10 attributes.
- 20% Update operations: Updating some of the attributes of a node.
- 20% Scan operations: Reading up to 300 nodes with all the attributes from a starting node key.
- 20% Read operations: Reading a node with all the attributes.
- 20% ReadModify operations: It's a combination of a Read operation followed by an Update of the same node.

All the benchmark operations are written in the Sparksee Algebra by mapping the benchmark table names to Sparksee Object Type names, which could be node types (as in this database) or edge types, and mapping the benchmark columns of the table to Sparksee attributes.
The benchmark is run with a variable number of threads ranging from 1 to 32 and the following figures show the results for the different dataset sizes.

3.5.3.1. Evaluation on the small dataset

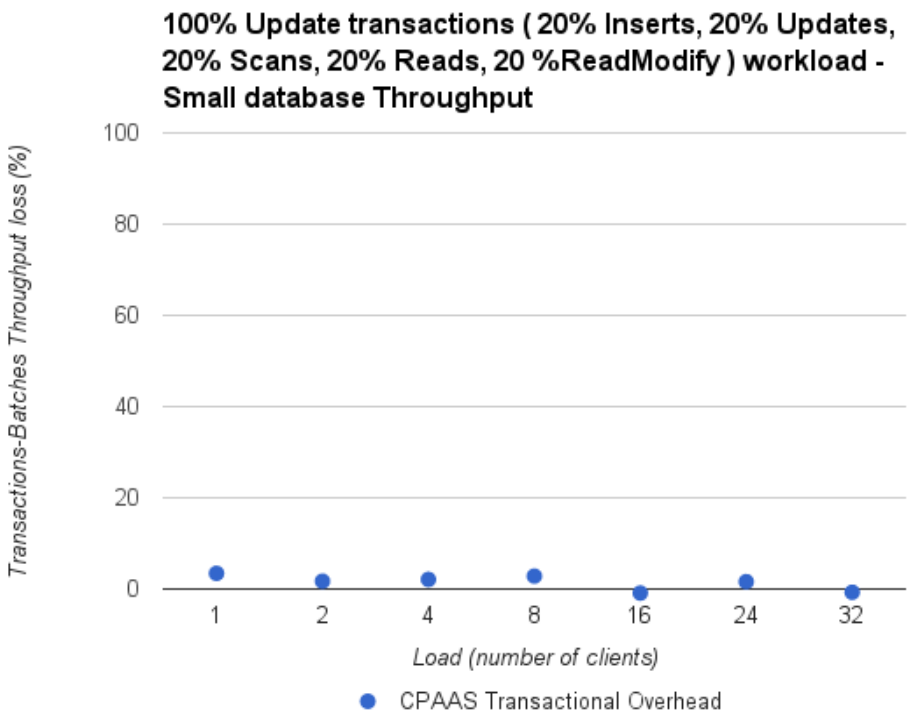

Figure 36 Sparksee - 100% Update transactions (20% Inserts, 20% Updates, 20% Scans, 20% Reads, 20% ReadModify) workload - Small database Transactions-Batches Throughput loss

Using the smallest database, the throughput loss with the HTM transactions is very small, between 1.7% and 3.4% up to 8 client threads, which is the number of cores in the Sparksee data store server. If we increase the number of client threads even further than the number of cpu cores the performance ceases to increase and the execution results become a lot more erratic. That can produce even negative throughput loss numbers because the system is saturated and the small HTM overhead can't really be measured anymore.

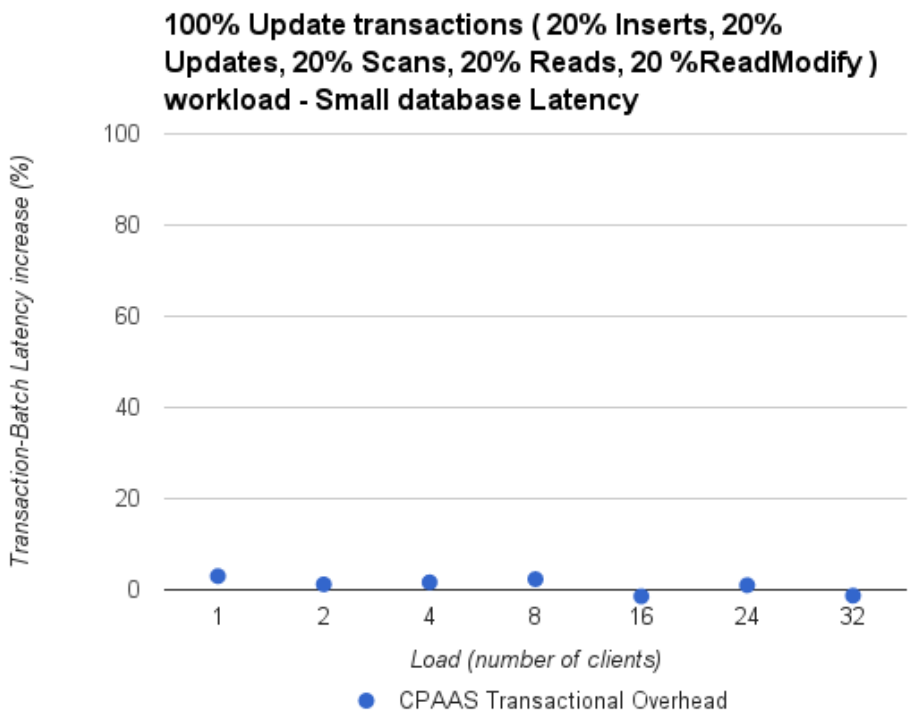

Figure 37 Sparksee - 100% Update transactions (20% Inserts, 20% Updates, 20% Scans, 20% Reads, 20% ReadModify) workload - Small database Transactions-Batches Latency Increase

The transaction latency increase in percentage for the smallest database follows the exact same pattern as the throughput loss and is less than 3%. The latency increase also becomes unpredictable once the number of client threads exceeds the number of cpu cores, which is also the point where the performance ceases to increase because the system is entering in saturation. As a result, we get a lot more variability in the results.

3.5.3.2. Evaluation on the medium dataset

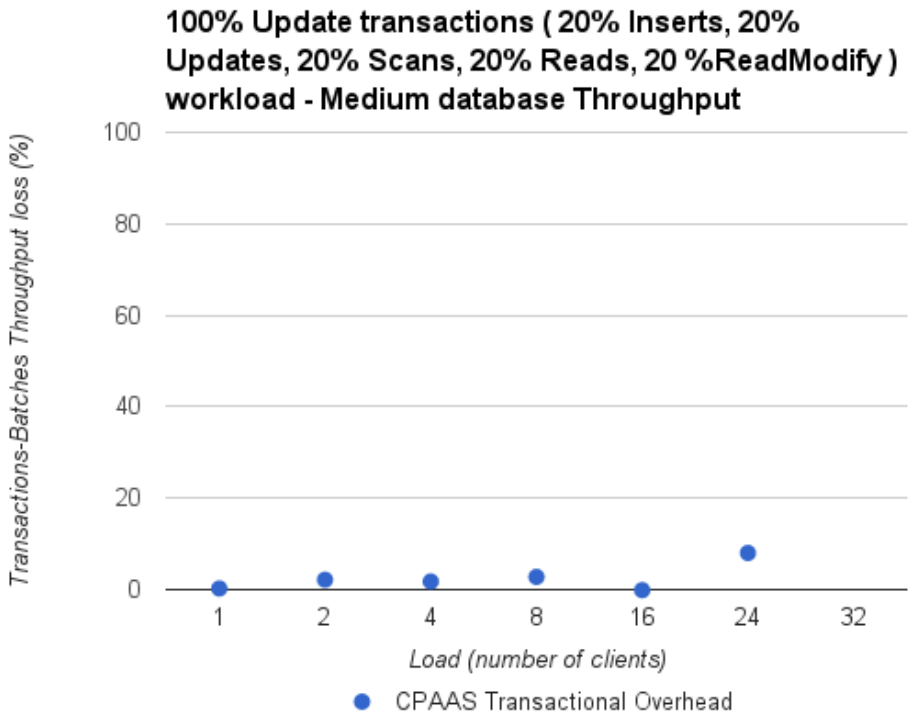

Figure 38 Sparksee - 100% Update transactions (20% Inserts, 20% Updates, 20% Scans, 20% Reads, 20% ReadModify) workload - Medium Database Transactions-Batches Throughput loss

Using a medium database, the throughput performance loss is similar, with even more constant results of at most 2.7% up to 8 thread clients. Again, as the load is increased and the number of threads become bigger than the number of cores in the server, the results can be unpredictable because it's not the HTM overhead what's causing the differences.

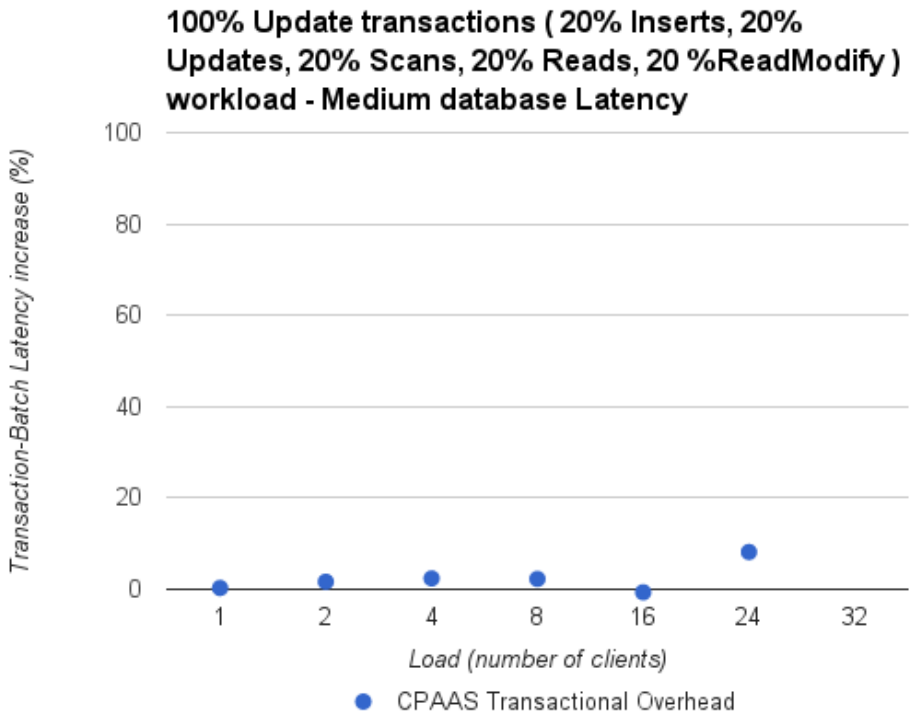

Figure 39 Sparksee - 100% Update transactions (20% Inserts, 20% Updates, 20% Scans, 20% Reads, 20% ReadModify) workload - Medium Database Transactions-Batches Latency Increase

Using the medium database size, the latency increase pattern looks almost exactly as the corresponding throughput loss in Figure 2.5(c) too, with a maximum of 2.4% latency increase for the optimal range of client threads.

3.5.3.3. Evaluation on the big dataset

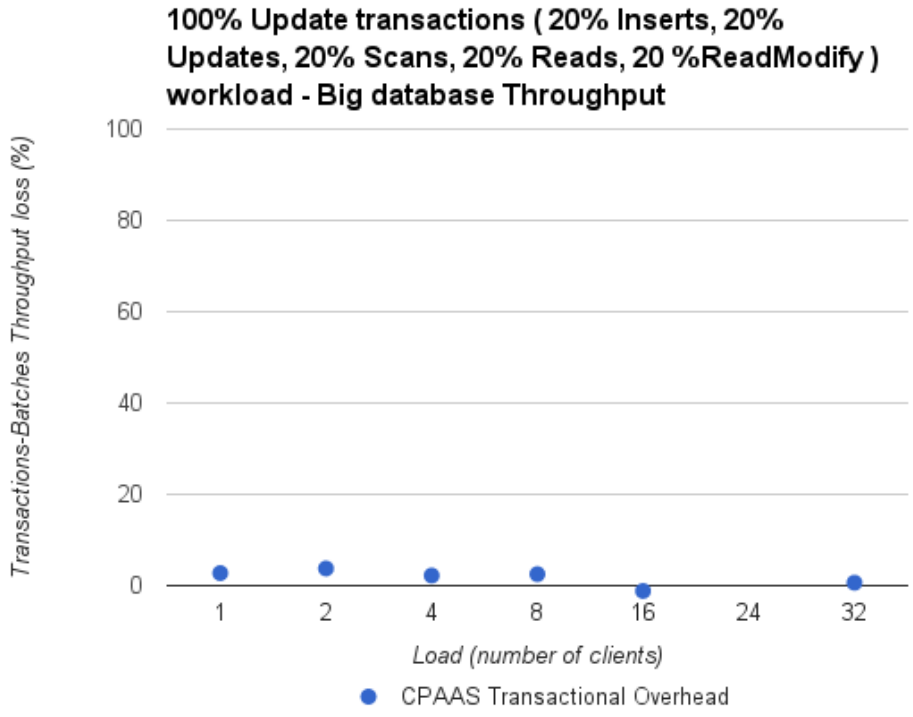

Figure 40 Sparksee - 100% Update transactions (20% Inserts, 20% Updates, 20% Scans, 20% Reads, 20% ReadModify) workload - Big database Transactions-Batches Throughput loss

Using the biggest database tested, a very similar pattern appears again, with the result throughput loss percentage still less than 3.8% until the number of client threads exceeds the number of cpu cores of the server.

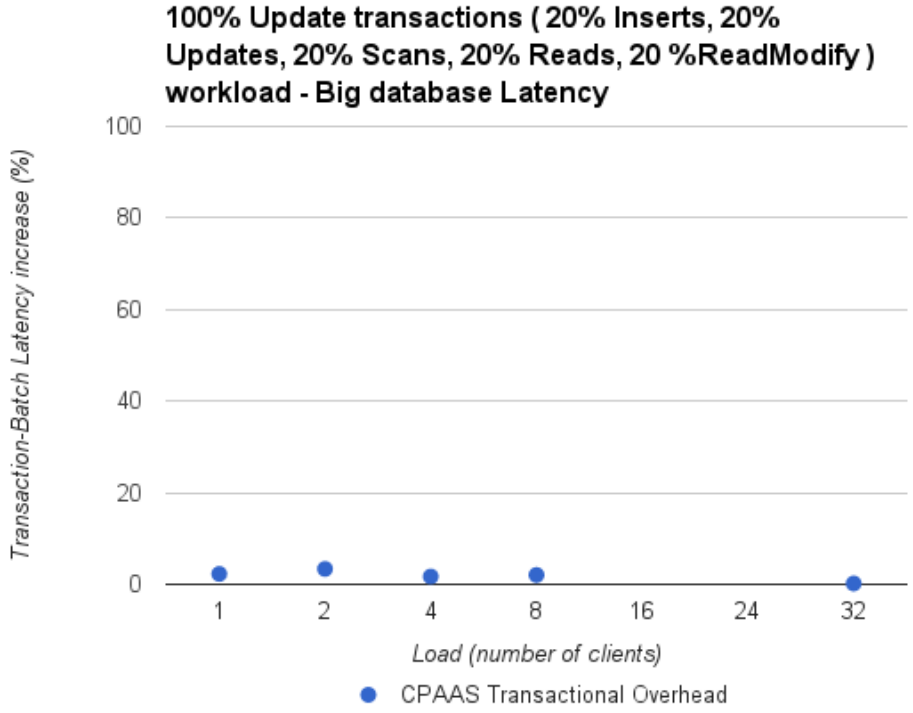

Figure 41 Sparksee - 100% Update transactions (20% Inserts, 20% Updates, 20% Scans, 20% Reads, 20% ReadModify) workload - Big database Transactions-Batches Latency Increase

Using the big database once more we get the same pattern with less than 3.4% latency increase.

3.5.3.4. Conclusions

The most significant information that we can see in both the throughput and latency figures is that the overhead introduced by the Holistic Transaction Manager is almost irrelevant, as expected. The reason is that Sparksee has its own transactions that are active even when it's used as a standalone database. When it's used in conjunction with the HTM, the amount of work in the HTM is minimized because Sparksee already does most of the job and the extra work required in the Sparksee server to handle the HTM transactions is also very small. As a result, we can see that both the throughput and the latency difference, using the CoherentPaaS HTM transactions and without using them, is very small and almost the same for all the different number of client threads and database sizes tested.

In the same way both the throughput increase percentage and the latency loss percentage using the CPAAS HTM transactions remains very low and predictable as long as the number of clients does not exceed the number of cpu threads. As a result, using a number of clients equal or lower to the number of cpu cores of the server is highly recommended to obtain the maximum performance possible. But the results show that if the number of client threads is increased even further the throughput loss and the latency increase will still be kept small anyway, sometimes even smaller because the difference in this situation is the server saturation and not the small overhead of the

transactions. So the HTM overhead will not have any significant impact in the performance independently of the number of client threads or database size.
In conclusion, Sparksee integration with the Holistic Transaction Manager gives the advantage of having transactions across data stores without losing performance.

3.6. ActivePivot

3.6.1. Setup

For ActivePivot, the benchmark has been performed on the following hardware: 1 server: 32 cores, 64 threads, 512GB RAM, centOS 7.1.15.03, Intel Xeon E5-4650 processors, 2.70 Ghz.
In addition to the ActivePivot server, the whole HTM stack was deployed on this server, namely zookeeper, one conflict manager, the snapshot server, the commit sequencer and the configuration manager.

3.6.2. Database

The ActivePivot server used the following datasets:

- A small one, with 1000000 rows
- A large one, with 10000000 rows

Each row contains 10 columns and one key. The size of a column is 100 bytes, which is also the size of the key. As a result, datasets size is around 1Gb and 10GB respectively.

The ActivePivot server is then designed to store all those rows in a table partitioned by record key, and an OLAP cube whose hierarchies are matching each of the column of the table is built on top of it. Usually, a hierarchy contains several attributes which are grouped and sorted, but as this benchmark expects to retrieve a single row and all of its attributes in its native format, each hierarchy is mapped to a single attribute only.

3.6.3. Evaluation

The workload for the evaluation has the following characteristic:

- 70% of read operations
- 10% of insert operations
- 20% of update operations.

Moreover, the operations of batched by 10. In the context of the CoherentPaaS environment, it implies that the 10 operations are made in the context of one transaction.
This workload represents a realistic usage of ActivePivot: we expect to load a big dataset at start-up, then the server is fed with real-time updates while users issue a lot of read-only transactions to compute their KPIs on ‘fresh’ data or investigate their business related metrics.
The evaluation has been performed with and without the CoherentPaaS transactional engine, using 1, 10, 20, 30, 40, 50 and 60 concurrent clients. The server having only 64 threads, we expect conflicts on hardware resources when using a high number of clients.

3.6.3.1. Evaluation results

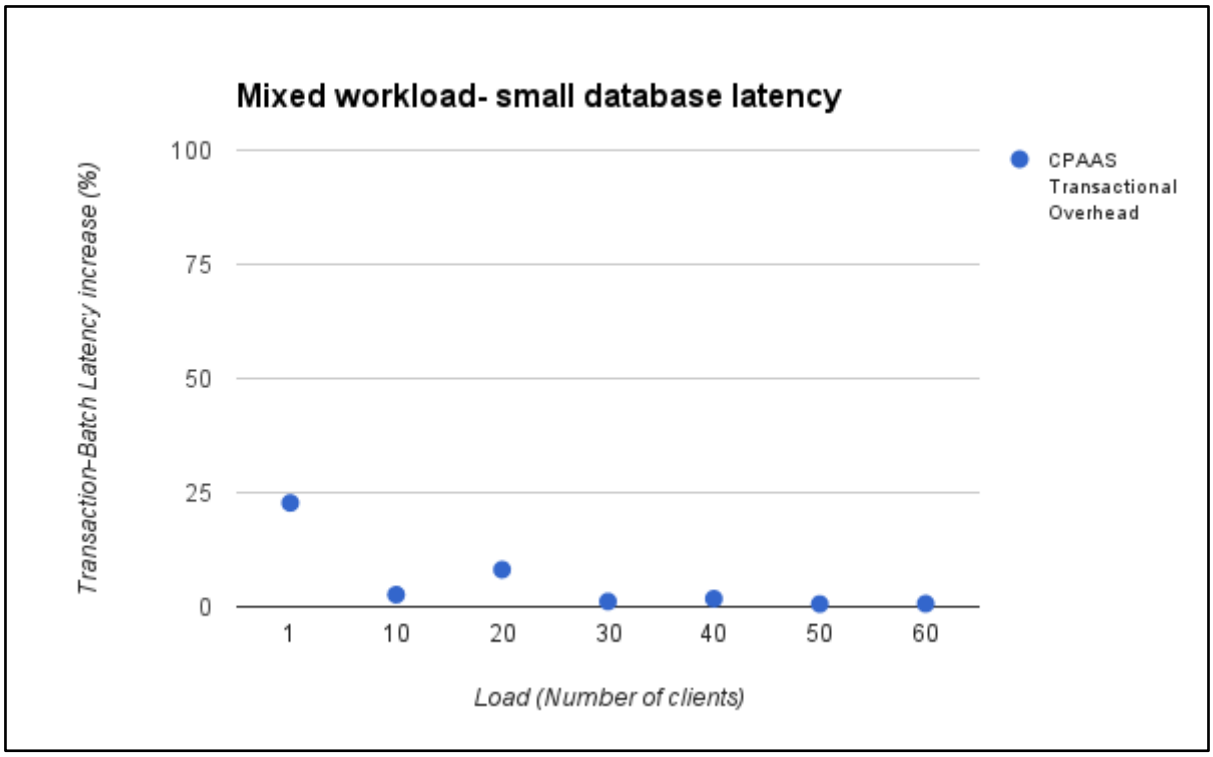

Figure 42 ActivePivot- Mixed workload-small database latency

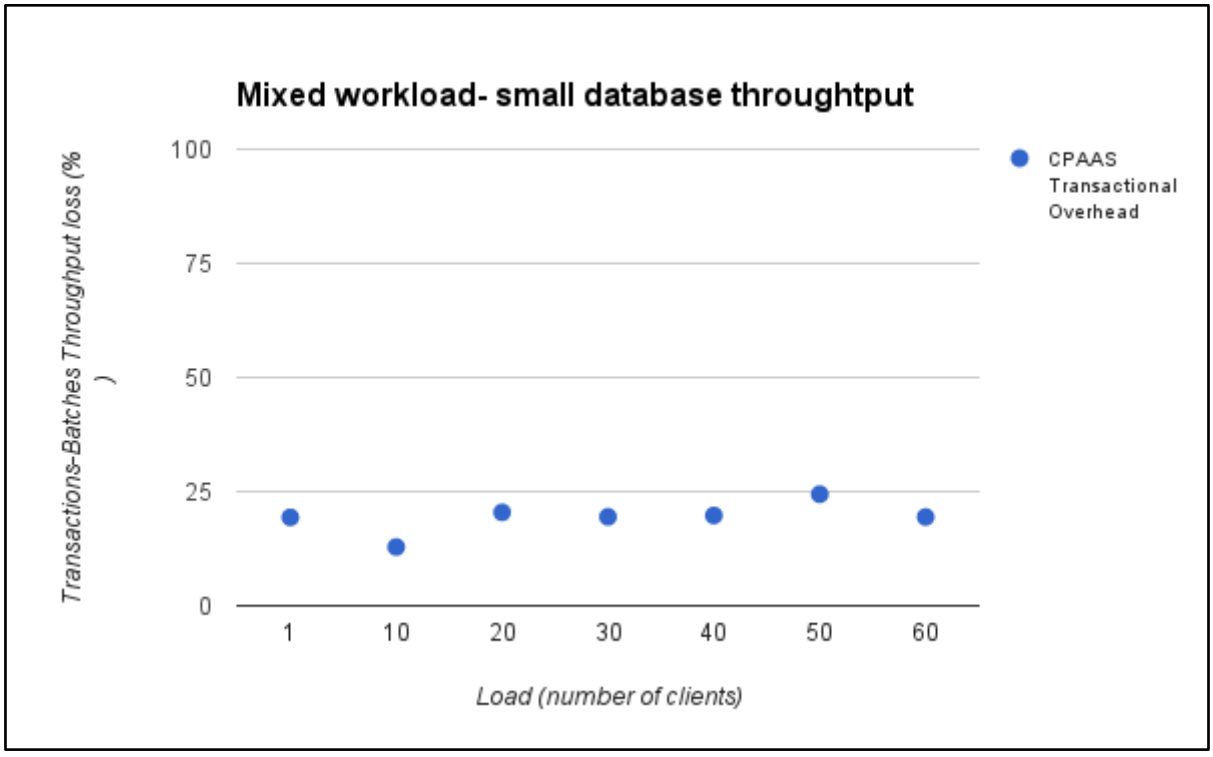

Figure 43 ActivePivot-Mixed workload-small database throughput

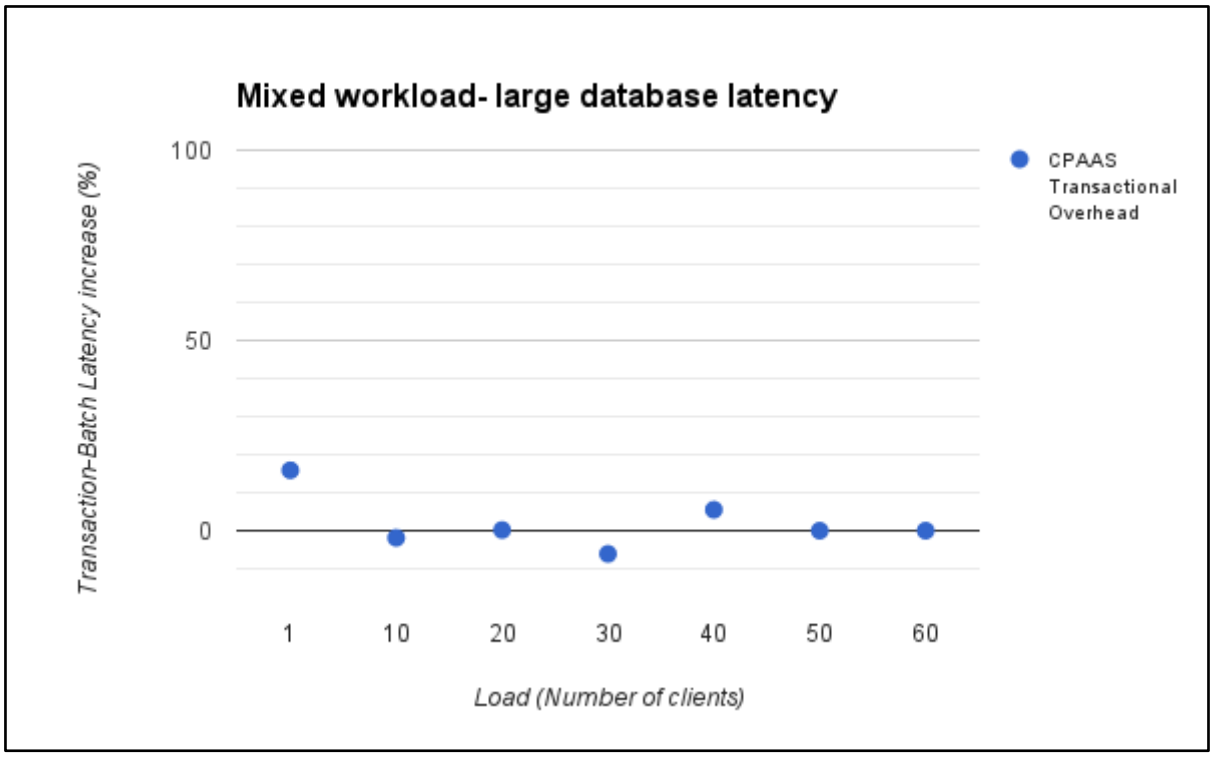

Figure 44 ActivePivot-Mixed workload- large database latency

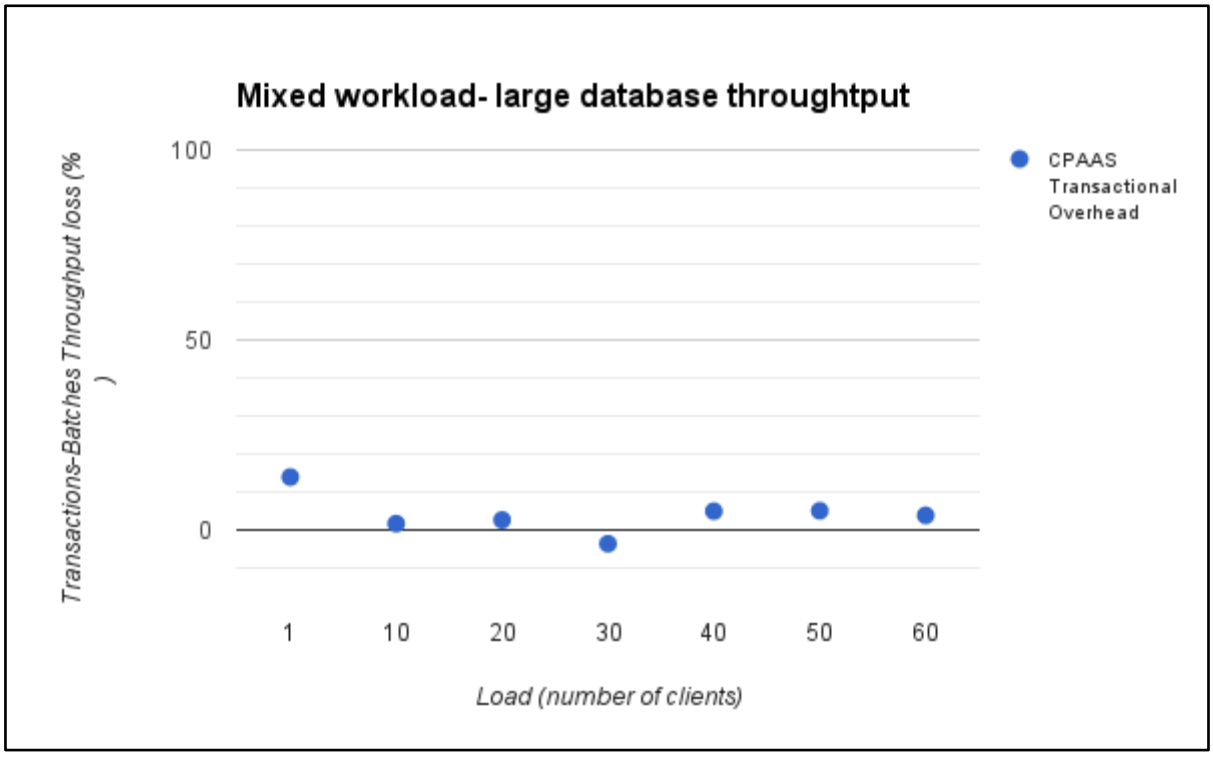

Figure 45 ActivePivot-Mixed workload-large database throughput

3.6.4. Conclusions

Considering Figure 42 and Figure 44 about latencies, we observe that the usage of the HTM introduces a small overhead, by increasing the latency around 5-10%. However, using the HTM provides the following features:

- Persistence of transactions (as an in-memory database, ActivePivot does not provide persistence natively)
- Conflict checking: ActivePivot does not natively checks for conflicts. When several rows are inserted with the same key into the database, those are considered as updates, and the last row to be inserted is considered as the valid one.
- Synchronization with other data stores.

The overhead can thus be considered negligible compared to the features it provides.

Figure 43 shows the relative impact on throughput with a small dataset: it is around 20%.

This can be explained by the read-only transactions (70% of the batches) being faster with small databases and thus having less impact on the global throughput: they are indeed almost not affected by the presence of the HTM, which has a stronger impact on updates and inserts. As the overhead is a fixed value (for a specific operation), it is more visible when operations are faster.

Figure 45 show the relative impact on throughput with a large dataset. The impact is stable, around 5%, except for the case where only one client is used: the throughput with one client being however really low (less than one transaction per second), this value is not significant.

The goal of CoherentPaaS being to handle a 'big data' environment, the figures concerning the large dataset are the most relevant. Figure 44 and Figure 45 both show a small overhead, and considering the features that the usage of the HTM provides to ActivePivot, this overhead is perfectly acceptable.

4. Common Query Language/Engine

The goal of this section is to provide a performance evaluation of the CoherentPaaS platform for queries executed from the common query engine and using the common query language. This evaluation shows the ability of the platform to perform optimized queries across data stores. Furthermore, queries/operators that are not provided by the data store itself can be done with the common query language, for instance Eutropia and ActivePivot do not provide join operations; however, this operation can be done at the common query language. This section will also report on the optimizations the query compiler does for executing queries. We will also show how the performance improves when native queries are used. For this purpose, we will use the data and some of the queries based on the TPC-H benchmark schema[14]. Each data store contains the same 8 datasets, generated by the TPC-H generator, stored in tables or in the corresponding for the data store structures.

For the experiments, we use the following environment:

- A cluster of identical machines: 8GB RAM, 4 CPU cores @2.4GHz;
- One node for the common query engine;
- One node for each of the data stores (6 in total);
- Each data store node contains data generated as per the TPC-H schema, with scale factor 1;
- Although the experiments involve only read-only queries, they were performed in the context of holistic transactional management.

We will explore the benefits of the CoherentPaaS platform, classified in two major groups:

- performance benefits, which includes optimizations, automatically performed by the components of the system, as well as performance benefits thanks to the support of native queries;
- functional benefits, which includes extended functionality for operators that are not natively supported by data stores, as well as SQL support for non-SQL data stores.

4.1. Performance benefits

In this section we will evaluate the optimization techniques enabled by the common query language/engine and will compare the execution times (with and without optimization) of different queries that involve each of the data stores. We divide the performed test cases in three categories: **automatic optimizations**, **automatic query transformations**, and **native query support performance benefits**. Furthermore, we evaluate a number of test case queries in each category against data stores each query is applicable to. Finally, we summarize the results of each test case in a comparison table, showing the benefits of the performed optimizations.

[14] http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf

4.1.1. Automatic optimizations

These are performed by the query compiler following heuristic approaches for rewriting the query execution plan, so that the execution time of expensive operators is significantly reduced; these include selection pushdowns and semi-joins across data stores, which the compiler does by moving selection operations to be handled by the wrappers (hence data stores).

4.1.1.1. Selection pushdown

We consider the following query, assuming that the column L_PARTKEY at the data store is indexed. The non-optimized execution of this query implies applying the selection predicate L_PARTKEY = 10 at the common query engine level, which requires a large number of rows from the LINEITEM table (5k rows) to be retrieved. In the optimized variant, the selection is pushed down and performed efficiently at the data store, which results in retrieving only the selected rows (2 rows) from the data store. This test case is applicable to SQL capable data stores, i.e. LeanXcale, MonetDB, and MongoDB.

```
T1( L_ORDERKEY long, L_PARTKEY long, L_SUPPKEY long, L_LINENUMBER int,
    L_QUANTITY float, L_EXTENDEDPRICE float, L_DISCOUNT float, L_TAX float,
    L_RETURNFLAG string, L_LINESTATUS string, L_SHIPDATE date, L_COMMITDATE date,
    L_RECEIPTDATE date, L_SHIPINSTRUCT string, L_SHIPMODE string,
    L_COMMENT string)@datastore =
(
    SELECT L_ORDERKEY, L_PARTKEY, L_SUPPKEY, L_LINENUMBER,
           L_QUANTITY, L_EXTENDEDPRICE, L_DISCOUNT, L_TAX,
           L_RETURNFLAG, L_LINESTATUS, L_SHIPDATE, L_COMMITDATE,
           L_RECEIPTDATE, L_SHIPINSTRUCT, L_SHIPMODE, L_COMMENT
    FROM LINEITEM
    WHERE L_QUANTITY < 5
)

SELECT * FROM T1 WHERE L_PARTKEY = 10
```

Table 1 Execution times in milliseconds for the PushDown test case

TestID	datastore	non-optimized	optimized
PD1	LeanXcale	355214	80
PD2	MonetDB	13895	120
PD3	MongoDB	22074	9

4.1.1.2. Bind joins and Scalar lookups

One of the major challenges to the CloudMdsQL language/engine is to allow integration operations (mostly joins) across heterogeneous data stores and to also be able to perform them in an efficient way. That is why we pay a special attention to the use of **bind joins** (see deliverable D3.4[15], Section 5.4.3), which is an efficient technique to apply semi-joins between datasets from different data stores by allowing the output of one of the subqueries to be used by the other subquery to filter out the unnecessary for the join rows. Moreover, we apply this technique to any pair of CoherentPaaS data stores, even if they do not support SQL querying.

[15] CoherentPaaS Deliverable D3.4: Query Engine for the Common Query Language v2 (Full Functionality)

We consider the following query, assuming that the column O_ORDERKEY is indexed. The selection predicate L_PARTKEY = 10 results in retrieving only 24 rows from the LINEITEM table which are then supposed to be joined with the ORDERS table, which contains 1.5mln rows. The non-optimized execution of this query implies that the entire ORDERS table is retrieved at the common query engine before being joined with the small table. In the optimized variant, the T2 subquery is rewritten by adding the predicate O_ORDERKEY IN (k_1, k_2, …, k_{24}) , where k_i are the values of the column L_ORDERKEY, taken from the small table T1. Thus, only the rows that match the join condition (24 instead of 1.5mln) will be retrieved from the ORDERS table.

```
T1( L_ORDERKEY long, L_PARTKEY long, L_SUPPKEY long, L_LINENUMBER int,
    L_QUANTITY float, L_EXTENDEDPRICE float, L_DISCOUNT float, L_TAX float,
    L_RETURNFLAG string, L_LINESTATUS string, L_SHIPDATE date, L_COMMITDATE date,
    L_RECEIPTDATE date, L_SHIPINSTRUCT string, L_SHIPMODE string,
    L_COMMENT string)@datastore1 =
(
    SELECT L_ORDERKEY, L_PARTKEY, L_SUPPKEY, L_LINENUMBER,
           L_QUANTITY, L_EXTENDEDPRICE, L_DISCOUNT, L_TAX,
           L_RETURNFLAG, L_LINESTATUS, L_SHIPDATE, L_COMMITDATE,
           L_RECEIPTDATE, L_SHIPINSTRUCT, L_SHIPMODE, L_COMMENT
    FROM LINEITEM
    WHERE L_PARTKEY = 10
)

T2( O_ORDERKEY long, O_CUSTKEY long, O_ORDERSTATUS string, O_TOTALPRICE float,
    O_ORDERDATE date, O_ORDERPRIORITY string, O_CLERK string, O_SHIPPRIORITY int,
    O_COMMENT string)@datastore2 =
(
    SELECT O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS, O_TOTALPRICE,
           O_ORDERDATE, O_ORDERPRIORITY, O_CLERK, O_SHIPPRIORITY,
           O_COMMENT
    FROM ORDERS
)

SELECT T1.*,
    T2.O_CUSTKEY, T2.O_ORDERSTATUS, T2.O_TOTALPRICE,
    T2.O_ORDERDATE, T2.O_ORDERPRIORITY, T2.O_CLERK,
    T2.O_SHIPPRIORITY, T2.O_COMMENT
FROM T1 JOIN BIND T2 ON L_ORDERKEY = O_ORDERKEY
```

The above query, as specified, is applicable only if both datastore1 and datastore2 support SQL subquerying (i.e. LeanXcale, MonetDB, and MongoDB). The programmer specifies which side of the join to be rewritten (T2 in our case) using the keyword BIND; and the compiler then takes care of doing the rewriting by pushing down to the subquery for T2 a predicate condition containing an IN operator that takes the values from an intermediate named table.

Next, we will explore the performance of this approach when using native subqueries, especially when T2 is native, as it requires the native query to access intermediate data from the table storage of the common query engine. This is done with the help of the programmer by adding to the signature of T2 the following clause:

```
JOINED ON O_ORDERKEY REFERENCING OUTER AS t1_keys
```

Thus, whenever T2 is used for a bind join, the join key values of the other side of the join (T1) are provided in the intermediate table t1_keys and the native query for T2 needs to use the corresponding mechanism that its wrapper provides to access these join keys and use them to return only rows that match the join criteria. Sparksee and ActivePivot

wrappers provide different mechanisms for expressing this, as listed in the corresponding implementations of T2 below:

```
T2( O_ORDERKEY long, O_CUSTKEY long, O_ORDERSTATUS string, O_TOTALPRICE float,
    O_ORDERDATE date, O_ORDERPRIORITY string, O_CLERK string, O_SHIPPRIORITY int,
    O_COMMENT string JOINED ON O_ORDERKEY REFERENCING OUTER AS t1_keys)@sparksee
= {*
  rs = CloudMdsQL.t1_keys()
  while rs.next():
       rs2 = graph.compute("""PROJECT(GRAPH::GET(
          GRAPH::SELECT( 'ORDERS'.'O_ORDERKEY' = """ + str(rs.getInt(1)) + """), 0,
          [ 'ORDERS'.'O_ORDERKEY', 'ORDERS'.'O_CUSTKEY',
            'ORDERS'.'O_ORDERSTATUS', 'ORDERS'.'O_TOTALPRICE',
            'ORDERS'.'O_ORDERDATE', 'ORDERS'.'O_ORDERPRIORITY',
            'ORDERS'.'O_CLERK', 'ORDERS'.'O_SHIPPRIORITY', 'ORDERS'.'O_COMMENT']),
            [1,2,3,4,5,6,7,8,9])""")
       while rs2.next():
             yield( rs2.getLong(1), rs2.getLong(2), rs2.getString(3), \
                 rs2.getFloat(4), rs2.getDate(5), rs2.getString(6), \
                 rs2.getString(7),rs2.getInt(8), rs2.getString(9) )
       rs2.close();
  rs.close()
*}
```

```
T2( O_ORDERKEY long, O_CUSTKEY long, O_ORDERSTATUS string, O_TOTALPRICE float,
    O_ORDERDATE date, O_ORDERPRIORITY string, O_CLERK string, O_SHIPPRIORITY int,
    O_COMMENT string JOINED ON O_ORDERKEY REFERENCING OUTER AS t1_keys)@activepivot
= {*
  SELECT NON EMPTY CROSSJOIN([Orders].[orderkey].[O_ORDERKEY].Members,
    [Orders].[ordercust].[O_CUSTKEY].Members,
    [Orders].[orderstatus].[O_ORDERSTATUS].Members,
    [Orders].[orderprice].[O_TOTALPRICE].Members,
    [Orders].[orderDate].[O_ORDERDATE].Members,
    [Orders].[orderpriority].[O_ORDERPRIORITY].Members,
    [Orders].[orderclerk].[O_CLERK].Members,
    [Orders].[ordershipprio].[O_SHIPPRIORITY].Members,
    [Orders].[ordercomment].[O_COMMENT].Members)
  ON ROWS FROM (SELECT {$$£[Orders].[orderkey].[O_ORDERKEY].£t1_keys$$ }
  ON COLUMNS FROM [TPCHCube])
*}
```

The above approaches allow for the native query to filter out rows by applying an analogue of the IN operator. This is possible, because Sparksee and ActivePivot provide powerful query mechanisms. However, for simpler query languages, such as the one for the key-value data store Eutropia, this approach is not applicable, as Eutropia has only primitives either to retrieve a whole table (scan) or to make a lookup for the value for a particular key (get). That is why, to achieve the same benefits of bind join as above, when T2 is a Eutropia query, we take advantage of **scalar lookups** (see deliverable D3.4, Section 5.4.1), that allow a parameterized named table (T2) to be used as a scalar function and evaluated for every value of a column from another table (T1). So, T2 is defined as a parameterized function that gets all the values of ORDERS for a particular key. Then, T2 is called in the SELECT list of the main SELECT statement of the query, instead of being joined with T1. The implementation of T2 for Eutropia and the main SELECT statement for this case are as follows:

```
T2( O_ORDERKEY long, O_CUSTKEY long, O_ORDERSTATUS string, O_TOTALPRICE float,
    O_ORDERDATE date, O_ORDERPRIORITY string, O_CLERK string, O_SHIPPRIORITY int,
    O_COMMENT string WITHPARAMS orderkey long)@eutropia =
{*
    get 'orders', orderkey, {SQLFormat => true}
*}
```

```
SELECT T1.*,
    T2(L_ORDERKEY).O_CUSTKEY, T2(L_ORDERKEY).O_ORDERSTATUS,
    T2(L_ORDERKEY).O_TOTALPRICE, T2(L_ORDERKEY).O_ORDERDATE,
    T2(L_ORDERKEY).O_ORDERPRIORITY, T2(L_ORDERKEY).O_CLERK,
    T2(L_ORDERKEY).O_SHIPPRIORITY, T2(L_ORDERKEY).O_COMMENT
FROM T1
```

Table 2 Execution times in milliseconds for the BindJoin test case

TestID	T1	T2	non-optimized	optimized
BJ1	MonetDB	MonetDB	56582	101
BJ2	MongoDB	MongoDB	63783	44
BJ3	LeanXcale	LeanXcale	109664	36689
BJ4	Sparksee	Sparksee	69393	627
BJ5	ActivePivot	ActivePivot	193963	1776
BJ6	MonetDB	MongoDB	65292	49
BJ7	MonetDB	Sparksee	69307	612
BJ8	MonetDB	ActivePivot	203272	1508
BJ9	MonetDB	Eutropia	97592	178
BJ10	Sparksee	ActivePivot	180256	1615
BJ11	ActivePivot	Eutropia	96430	280
BJ12	MongoDB	Eutropia	99141	212
BJ13	MongoDB	Sparksee	68259	572
BJ14	MongoDB	ActivePivot	205071	1664

4.1.2. Automatic query transformations

Automatic transformations of SQL query plans to native queries are performed by the wrapper thanks to its awareness of the SQL capability of its data store; this implies SQL operations, such as aggregate, group by, order by, and selection, to be pushed down by the wrapper to be efficiently executed by the data store instead of at the common query engine level.

Let us consider the following queries, which request different operations to a data store, expressed in SQL. The queries focus on selection, aggregation, grouping, sorting and join operations to a table in the data store. Regardless of the underlying data store, if it can be queried by SQL subqueries with CloudMdsQL, those operations are expressed with the same statements. The non-optimized executions imply that the whole table LINEITEM is retrieved from the data store and the requested operation is done at the common query engine. However, the data store wrapper is aware of the SQL capabilities of its data store, so in the optimized variant, those operations are translated by the wrapper to their corresponding native statements and pushed down to be executed efficiently at the data store, which may take advantage of any available indexes on the aggregated, grouped and ordered columns. This is especially useful for NoSQL data stores such as MongoDB, which natively do not support SQL, but now are given the possibility to be abstracted with the same querying mechanism as all relational databases, while still performing the execution in the most efficient for the data store way. However, the join test case is only applicable to relational data stores (LeanXcale and MonetDB in our case).

Query_SEL:

```
T1(L_ORDERKEY long)@datastore =
```

```
(
  SELECT L_ORDERKEY FROM LINEITEM
  WHERE L_QUANTITY < 5
    AND L_PARTKEY = 10
)
SELECT * FROM T1
```

Query_AGG:

```
T1(TOTAL_COUNT long)@datastore =
(
  SELECT COUNT(*) FROM LINEITEM WHERE L_QUANTITY < 5
)
SELECT * FROM T1
```

Query_GRP:

```
T1(O_ORDERSTATUS string, O_ORDERPRIORITY string, AVG float)@datastore =
(
  SELECT O_ORDERSTATUS, O_ORDERPRIORITY, AVG (O_TOTALPRICE)
  FROM ORDERS
  WHERE O_ORDERSTATUS = 'F'
  GROUP BY O_ORDERSTATUS, O_ORDERPRIORITY
)
```

Query_ORD:

```
T1(O_ORDERSTATUS string, O_ORDERPRIORITY string, AVG float)@datastore =
(
  SELECT O_ORDERSTATUS, O_ORDERPRIORITY, AVG (O_TOTALPRICE)
  FROM ORDERS
  WHERE O_ORDERSTATUS = 'F'
  GROUP BY O_ORDERSTATUS, O_ORDERPRIORITY
  ORDER BY O_ORDERSTATUS, O_ORDERPRIORITY
)
```

Query_JOIN:

```
T1(ORDERKEY long, CUSTKEY long, ORDERSTATUS string, QUANTITY float)@datastore =
(
  SELECT O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS, L_QUANTITY
  FROM LINEITEM JOIN ORDERS
    ON L_ORDERKEY = O_ORDERKEY
  WHERE L_PARTKEY = 10
)
SELECT * FROM T1
```

Table 3 Execution times in milliseconds for the automatic transformations test cases

TestID	Query	datastore	non-optimized	optimized
QT1	Query_SEL	LeanXcale	355214	80
QT2	Query_SEL	MonetDB	13895	120
QT3	Query_SEL	MongoDB	22074	9
QT4	Query_AGG	LeanXcale	345809	1375
QT5	Query_AGG	MonetDB	5354	38
QT6	Query_AGG	MongoDB	19981	191
QT7	Query_GRP	LeanXcale	148270	1508
QT8	Query_GRP	MonetDB	6310	110
QT9	Query_GRP	MongoDB	6348	1965
QT10	Query_ORD	LeanXcale	145389	1390
QT11	Query_ORD	MonetDB	6266	110
QT12	Query_ORD	MongoDB	5877	1675
QT13	Query_JOIN	LeanXcale	109664	108
QT14	Query_JOIN	MonetDB	56582	108

4.1.3. Native query support performance benefits

The support of native queries in CloudMdsQL allows for the user to write powerful subqueries to the data stores that efficiently perform specific for the data store operations with native statements, rather than to express them in SQL and handle at the common query engine. This requires deeper expertise and knowledge about the specifics of the queried data stores; however, the added value is that the programmer can still request efficient operations at the data store, thanks to the support of native queries, and combine the results with other data store queries.

Let us consider the same queries from the previous subsection: Query_SEL, Query_AGG, Query_GRP, Query_ORD and Query_JOIN. Let us assume that the programmer needs to request the same operations to non-SQL data stores (such as Sparksee and ActivePivot) that can natively support them. The naïve way to achieve this is for the wrapper developer or DBA to provide a static mapping of data store structures to relational tables and let the programmer express her query in SQL at the common query engine level. But now the programmer can take advantage of the native queries support and express the operations in native for the data store statements, which will result in an optimal execution. Note that the generated TPC-H data for ActivePivot is stored in one single cube; so although ActivePivot does not natively support joins between cubes, we categorize it as a join-capable data store only for this test case, because in fact a conceptual join between LINEITEM and ORDERS is handled by ActivePivot by dealing with dimensions from the same cube.

With this group of test cases, we compare the execution times:

- when the requested operations are expressed in native queries and done by the data store (optimized), and
- when the operations are expressed in SQL and executed at the common query engine, after the wrapper simply delivers the entire table (non-optimized).

Table 4 . Execution times in milliseconds for test cases focusing on native query support benefits

TestID	Query	datastore	non-optimized	optimized
NQ1	Query_SEL	Sparksee	19068	76
NQ2	Query_SEL	ActivePivot	20009	261
NQ3	Query_AGG	Sparksee	26893	26
NQ4	Query_AGG	ActivePivot	12592	112
NQ5	Query_GRP	Sparksee	36870	2654
NQ6	Query_GRP	ActivePivot	10067	16
NQ7	Query_ORD	Sparksee	51075	2640
NQ8	Query_ORD	ActivePivot	8907	77
NQ9	Query_JOIN	Sparksee	69393	785
NQ10	Query_JOIN	ActivePivot	193963	189

4.2. Functional benefits

In this section, we focus on the added value of the CoherentPaaS platform with respect to the functionality of the underlying data stores, both as individual subsystems and all of them as a whole. For instance, in the previous subsection, with the evaluation of the bind join technique, we already discussed one major functionality benefit – the ability to

perform joins across different data stores, even involving data stores that do not natively support joins (Sparksee, MongoDB, Hbase, ActivePivot). In this section, we elaborate more the functional enhancements for some data stores. We will benchmark different queries that involve the affected data stores. We divide the performed test cases in two categories: **extended functionality for data stores** and **SQL support for non-SQL data stores**. Furthermore, we evaluate a number of test case queries in each category against data stores each query adds functional value to. Finally we summarize the results of all test cases of each category, showing the execution times of the performed tests.

4.2.1. Extended Functionality for Data Stores

This group of queries focuses on the ability of the CoherentPaaS platform to provide useful functionality for querying a particular data store, even though the data store does not natively support it. For example, non-relational data stores do not have support for joins and key-value data stores do not support operations like aggregate, group by, order by value, filter by value, etc. But now the programmer has the tools to involve such data stores in the operations the data store lacks.

Within this group of test cases, we assume that the programmer or wrapper developer has implemented beforehand the CloudMdsQL named tables that correspond to the datasets in the data store and we evaluate the functional benefits of applying SQL operations on top of these datasets, at the common query engine level, thus extending the data store functionality with features it did not previously have. With respect to this, we focus on NoSQL data stores and the SQL operators they are missing, in particular, joins for MongoDB, as well as selections (not on the primary key), aggregates, grouping and sorting for Eutropia.

We consider the following SQL queries assuming that CloudMdsQL named tables have the same names as the tables in the TPC-H schema.

Query_SEL:
```
SELECT L_ORDERKEY FROM LINEITEM
WHERE L_QUANTITY < 5 AND L_PARTKEY = 10
```

Query_AGG:
```
SELECT COUNT(*) FROM LINEITEM WHERE L_QUANTITY < 5
```

Query_GRP:
```
SELECT O_ORDERSTATUS, O_ORDERPRIORITY, AVG (O_TOTALPRICE)
FROM ORDERS
WHERE O_ORDERSTATUS = 'F'
GROUP BY O_ORDERSTATUS, O_ORDERPRIORITY
```

Query_ORD:
```
SELECT O_ORDERSTATUS, O_ORDERPRIORITY, AVG (O_TOTALPRICE)
FROM ORDERS
WHERE O_ORDERSTATUS = 'F'
GROUP BY O_ORDERSTATUS, O_ORDERPRIORITY
ORDER BY O_ORDERSTATUS, O_ORDERPRIORITY
```

Query_JOIN:
```
SELECT O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS, L_QUANTITY
FROM LINEITEM JOIN ORDERS
  ON L_ORDERKEY = O_ORDERKEY
WHERE L_PARTKEY = 10
```

Table 5 Execution times in milliseconds for test cases focusing on extended functionality

TestID	Query	datastore	Exec. time
EF1	Query_SEL	Eutropia	6068
EF2	Query_AGG	Eutropia	3137
EF3	Query_GRP	Eutropia	6678
EF4	Query_ORD	Eutropia	6194
EF5	Query_JOIN	MongoDB	63783

4.2.2. SQL support for non-SQL data stores

Although some non-SQL data stores support relational operators that can be used within a native query, the programmer may still be given the possibility to query them using SQL statements, assuming that a wrapper developer or DBA has implemented beforehand the CloudMdsQL named tables that correspond to the datasets in the data store. The queries in this category share the same concept with the ones from the previous subsection, but we put them in a different category, because the value they add is less significant, as they do not provide missing functionality, but simply allow non-SQL data stores to be queried by SQL, thus exposing them to a wider community of programmers.

We consider the queries Query_SEL, Query_AGG, Query_GRP and Query_ORD from the previous subsection, but this time addressing the data stores Sparksee and ActivePivot.

Table 6 Execution times in milliseconds for test cases focusing on SQL support on non-SQL stores

TestID	Query	datastore	Exec. time
SQ1	Query_SEL	Sparksee	19068
SQ2	Query_SEL	ActivePivot	20009
SQ3	Query_AGG	Sparksee	26893
SQ4	Query_AGG	ActivePivot	12592
SQ5	Query_GRP	Sparksee	36870
SQ6	Query_GRP	ActivePivot	10067
SQ7	Query_ORD	Sparksee	51075
SQ8	Query_ORD	ActivePivot	8907
SQ9	Query_JOIN	Sparksee	69393
SQ10	Query_JOIN	ActivePivot	193963

5. Scalability evaluations

In this section we present three scalability evaluations. We performed scalability evaluation of H-Eutropia, LeanXcale and a joint scalability evaluation of the CEP and H-Eutropia. We also measure the scalability of the platform in terms of number of data stores.

5.1. Evaluation of H-Eutropia

In section 5.1.1, we study the efficiency of H-Eutropia in standalone setups and compare it with HBase and Cassandra NoSQL stores. In section **Error! Reference source not found.**, we evaluate H-Eutropia scalability performance.

5.1.1. H-Eutropia standalone evaluation

Our experimental platform consists of two systems (client and server) each with two quad-core Intel(R) Xeon(R) E5520 CPUs running at 2.7 GHz. The server is equipped with 48 GB DDR-III DRAM, and the client with 12 GB. Both nodes are connected via a 10 Gbits/s network link. As storage devices, the server uses four Intel X25-E SSDs (32 GB) and we make a RAID-0 with them using the standard md Linux driver. Eutropia is implemented in C and is accessed from applications as a shared library. H-Eutropia is cross-linked between the Java code of HBase and the C code of Eutropia. We use the open-source Yahoo Cloud Serving Benchmark (YCSB) to generate synthetic workloads. The default YCSB implementation executes gets as range queries and therefore, exercises only scan operations. For this reason, we modify YCSB to use point queries for get operations. We still exercise range queries in Workload E, which uses scan operations.

	Workload
A	50% reads, 50% updates
B	95% reads, 5% updates
C	100% reads
D	95% reads, 5% inserts
E	95% scans, 5% inserts
F	50% reads, 50% read-modify-write

Figure 46: YCSB workloads

In all cases, we run the standard workloads proposed by YCSB with the default values as shown in Figure 46. We run the following sequence proposed by the YCSB author: Load the database using workload A's configuration file, run workloads A, B, C, F, and D in a row, delete the whole database, reload the database with workload E's configuration file, and run workload E.

We use a small dataset that fits in memory and a large dataset that does not. The small dataset is composed of 100M key/value pairs whereas the large dataset has 500M key/value pairs. Each key is of approximate size of 32 bytes whereas the value is of size 100 bytes.

We analyze the efficiency and performance of H-Eutropia, compared to HBase and Cassandra. We measure efficiency as cycles/op, which shows the cycles needed to complete an operation on average. We calculate efficiency with the equation shown in Figure 47

$$cycles/op = \frac{\frac{CPU\_utilization}{100} \times \frac{cycles}{s} \times cores}{\frac{average\_ops}{s}},$$

Figure 47: Efficiency metric equation

Figure 48 depicts the speedup in efficiency (cycles/op) achieved by H-*Eutropia* over HBase and Cassandra. We see that H-*Eutropia* significantly outperforms both HBase and Cassandra. H-*Eutropia* uses fewer cycles/op by up to 2.9*X*, 8.4*X*, and 5.6*X* for write intensive, read intensive and mixed workloads compared to HBase. Compared to Cassandra, the improvement depends on the size of the dataset. With the small dataset H-*Eutropia* outperforms Cassandra by up to 5.8*X*, 16.1*X*, and 13.5*X* for the write, read intensive, and mix workloads, respectively. With the large dataset, H-*Eutropia* improves cycles/op over Cassandra by up 3.9*X*, 61.4*X*, and 37.2*X* write, read-intensive and mixed workloads respectively.

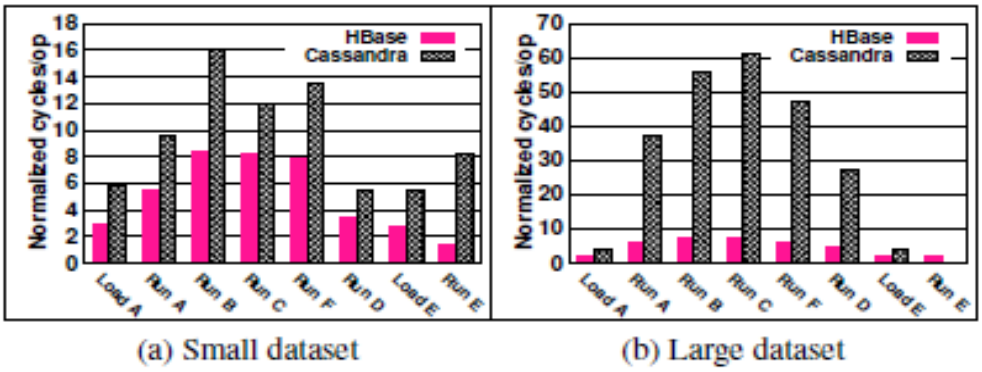

Figure 48: Improvement in efficiency (cycles/op) achieved by H-Eutropia over HBase and Cassandra.

We see that H-Eutropia significantly outperforms both HBase and Cassandra. Compared to HBase, H-Eutropia uses fewer cycles/op by up to 2.9×, 8.4×, and 5.6× for write intensive, read intensive, and mixed workloads. Compared to Cassandra, the improvement depends on the size of the dataset. With the small dataset, H-Eutropia outperforms Cassandra by up to 5.8×, 16.1×, and 13.5× for the write, read intensive, and mix workloads, respectively. With the large dataset, H-Eutropia improves cycles/op over Cassandra by up 3.9×, 61.4×, and 37.2× write, read-intensive and mixed workloads respectively. Next, we examine throughput in terms of ops/s. Figure 49 shows performance in kops/s whereas Figure 50 depicts the amount of data read and written by each workload. For the small dataset, H-Eutropia has up to 5.4× higher throughput compared to HBase and up to 10.7× compared to Cassandra. In addition, H-Eutropia does not perform any reads during the run phases. Cassandra does not read any data either, whereas HBase reads 5.1 GB and 5.2 GB when running workloads A and E,

respectively. H-Eutropia significantly reduces the amount of data written to the device by 38% and 17% compared to HBase and Cassandra.

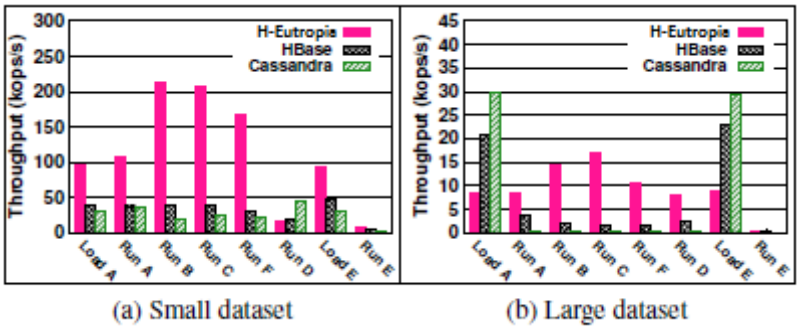

Figure 49: Throughput (kops/s) achieved by H-Eutropia, HBase and Cassandra

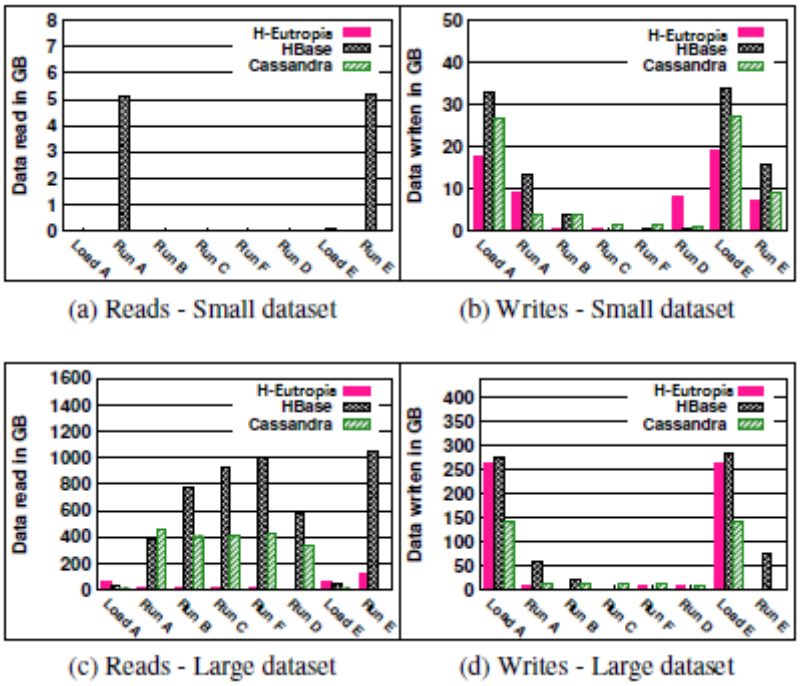

Figure 50: Amount of data, in GB, read/written by H-Eutropia, HBase, and Cassandra

For the large dataset, during the run phase, H-Eutropia outperforms HBase and Cassandra by up to 10.7× and 153.3×, respectively. This improvement is reflected in a significant reduction of the amount of data read from the storage device, by up to 16× and 6.9× compared to HBase and Cassandra, respectively. For read-intensive and mixed workloads, H-Eutropia is more lightweight not only in CPU utilization but also in the amount of data read. Our modified Bε–tree performs faster lookups than the LSM-trees used by HBase and Cassandra, obtaining significant improvement in throughput.

During the load phase (write intensive workloads) for the large dataset H-Tucana exhibits up 2.5× and 3.7× worse throughput than HBase and Cassandra, as shown in Figure 11b). Figure 12 shows that during the load phase, H-Eutropia writes 264 GB and reads 69 GB, although the size of the dataset including metadata is 77.2 GB. This is not inherent to the design of Eutropia, as shown by the results in Section 4.2, but rather due

to mmap, as follows. With mmap modified disk blocks are written to the device not only during Tucana's commit operations, but also periodically, by the flush kernel threads when they are older than a threshold or when free memory shrinks below a threshold, using an LRU policy and madvise hints. We believe that due to the increased memory pressure in H-Eutropia is due to the Java HBase front-end, mmap evicts not only log pages, but also leaf pages. This reduces the amount of I/Os that can be amortized for inserts due to the limited buffering in our Bε–tree. To solve this problem, we need to (a) control better which pages are evicted by mmap, which will be effective up to roughly the 10-15% ratio of memory to SSD capacity (see Section 3.1), and (b) add buffering one level higher in the Bε–tree. In the same figure, we notice that in run D phase, using the small dataset, we write more data than the other systems. This is because workload D inserts new key-value pairs and then searches for them. YCSB always searches for keys that exist in the database. In Eutropia newly inserted keys appear in searches only after a commit operation. If a key is not found, we issue a commit operation to read it. These commit operations causes increased traffic to/from the device. However, the other systems retrieve the new values directly from memory. In the large dataset case, all systems write them to devices and all of them write about the same amount of data.

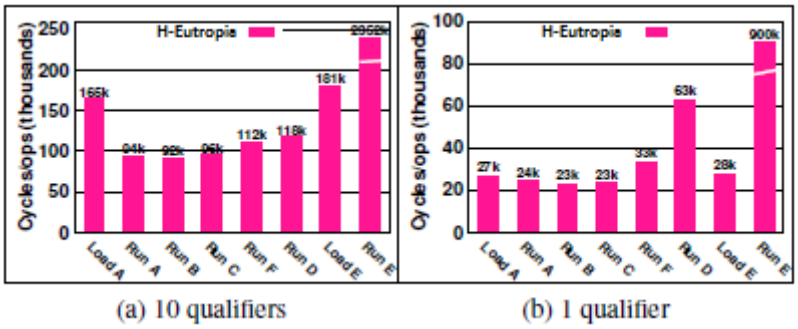

Figure 51: Number of cycles needed by H-Tucana for YCSB workloads

Figure 51 shows the cycles/op in H-Eutropia to execute all the workloads with ten (default configuration) and with one qualifier. With ten qualifiers, write intensive workloads require on average 172K cycles/op and read intensive and mix workloads require on average 115K cycles/op. Workload E that performs scans uses more than 2.3M cycles/op (for retrieving 50 key-value pairs). Figure 13b shows that with a single key, all write intensive, read-intensive, and mixed workloads require on average 27K cycles/op, whereas workload E requires 900K cycles/op. In more detail, we see that on average 40% of the time is used by the HBase component in H-Eutropia, 23% by Eutropia, 33% by the system, and 5% by other processes

5.2. Scalability of H-Eutropia

5.2.1. Experiment design

H-Eutropia alters HBase storage engine keeping its management, failure recovery, and scaling mechanisms. H-Eutropia splits tables into shards named regions. Each region

contains a consecutive range of the key space. It later distributes regions to HRegion servers for scale out purposes. Clients contact H-Eutropia Master server. HMaster keeps the mapping between tables and their corresponding regions. Clients contact HMaster to get this mapping information and contact the appropriate HRegion server for the corresponding key range. In the following experiments we evaluate the scaling capabilities of H-Eutropia.

The machines used for these experiments consists of quad core Intel(R) Xeon(R) CPUs X3220 at 2.40GHz with 8GB of DRAM. As storage device they use an INTEL SSD model SC2BB160G4 of capacity 160GB. SSD can perform up to 7500 random 4KB write operations and up to 75000 4KB random read operations. All machines are connected via a 1GBps switch.

Initially we choose an initial database size A for 1 node and we increase the database size as follows. For each region server added we increase the database size by A. We first measure the single node throughput to set the appropriate number of clients C. Continuing, we increase the load by adding C clients per region server. Finally, for all cases we presplit the table into a fixed number of regions. The number of regions is analogous to the number of regions servers. For each region server we add 4 regions to the table. These regions are assigned to the region's server by HMaster in a round robin fashion.

We run YCSB benchmark in three sets of experiments. First we evaluate a write only workload, second a read only workload, and finally a mix workload which consists of 50% gets and 50% updates. In each case we scale the system from 1 to 3 and 5 servers respectively.

In the write workload we run the benchmark against an empty table. As more load is added the final size of the table increases. In the read workload, we populate the table using the following rule. For each region server we insert 40 million key value pairs per region server. Finally, for the mix workload case we populate each server with 20 million key value pairs with 4 regions per server. YCSB generate requests using uniform distribution, by default.

5.2.2. Results

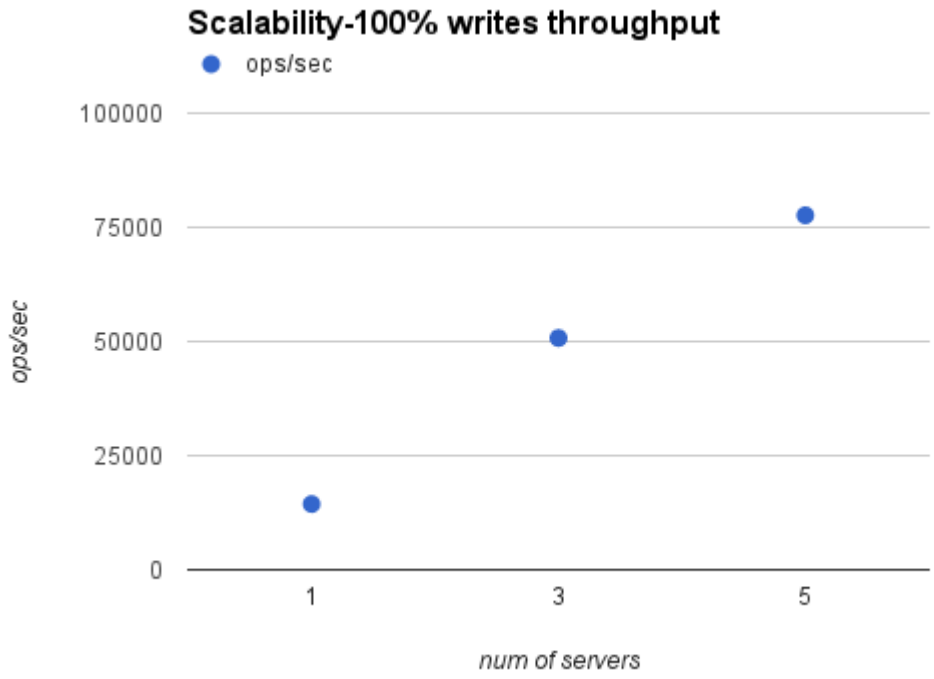

Figure 52 H-Eutropia: Scalability - 100% writes throughput

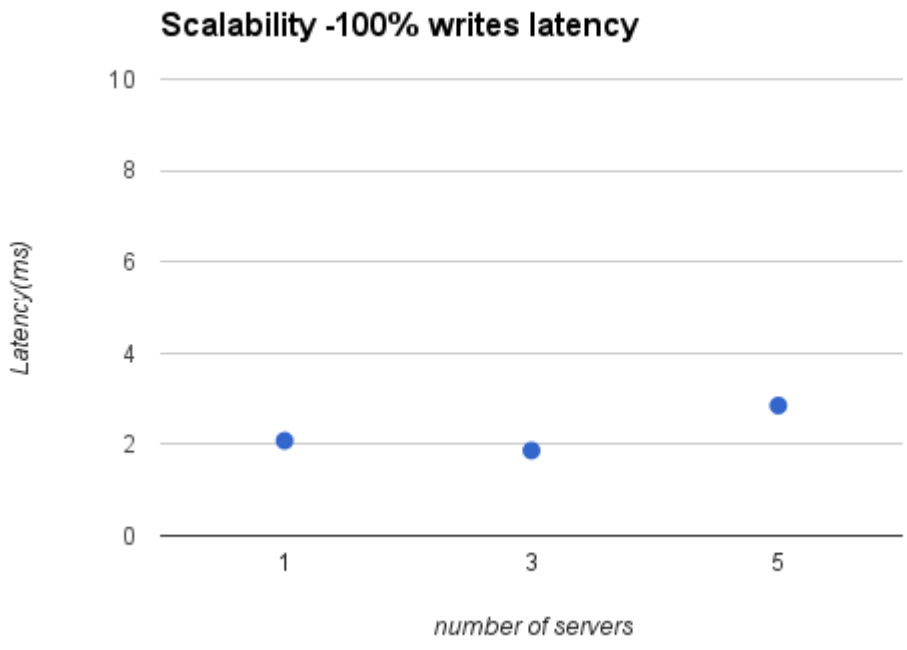

Figure 53 H-Eutropia: Scalability - 100% writes latency

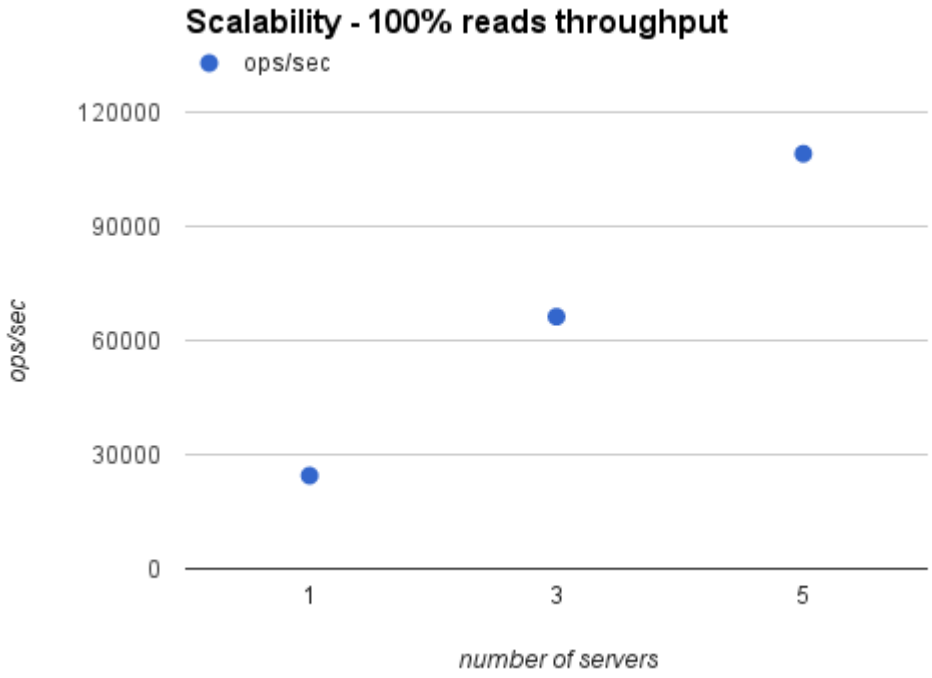

Figure 54 H-Eutropia: Scalability - 100% reads throughput

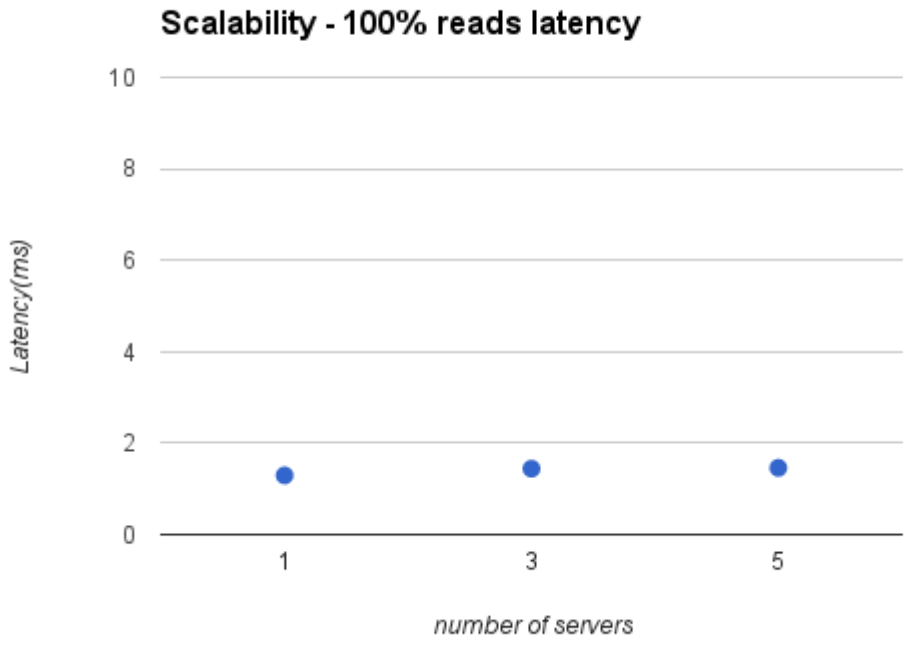

Figure 55 H-Eutropia: Scalability - 100% reads latency

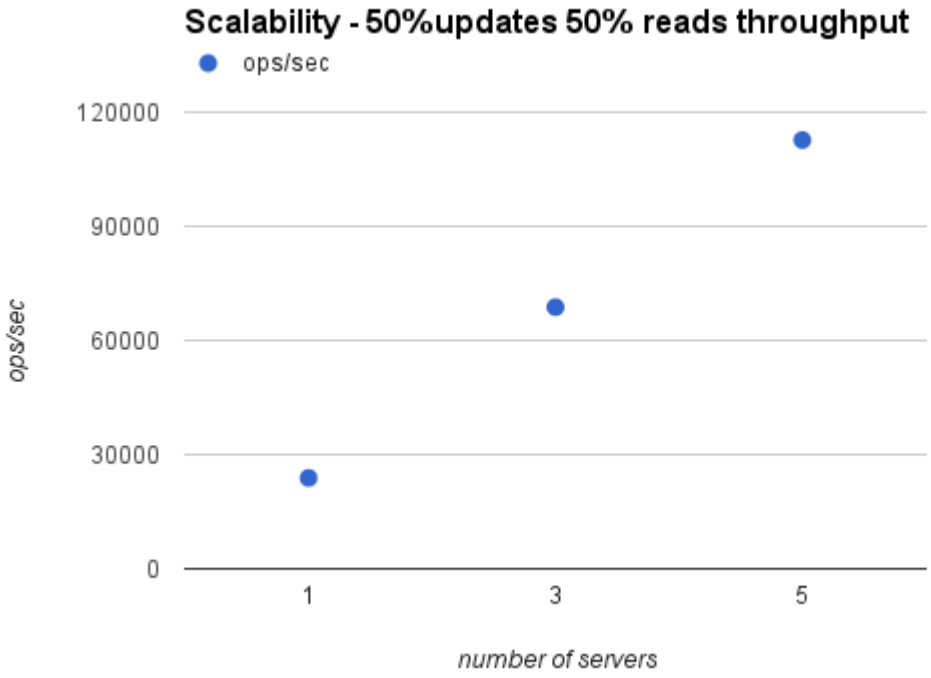

Figure 56 H-Eutropia: Scalability 50% updates - 50% reads throughput

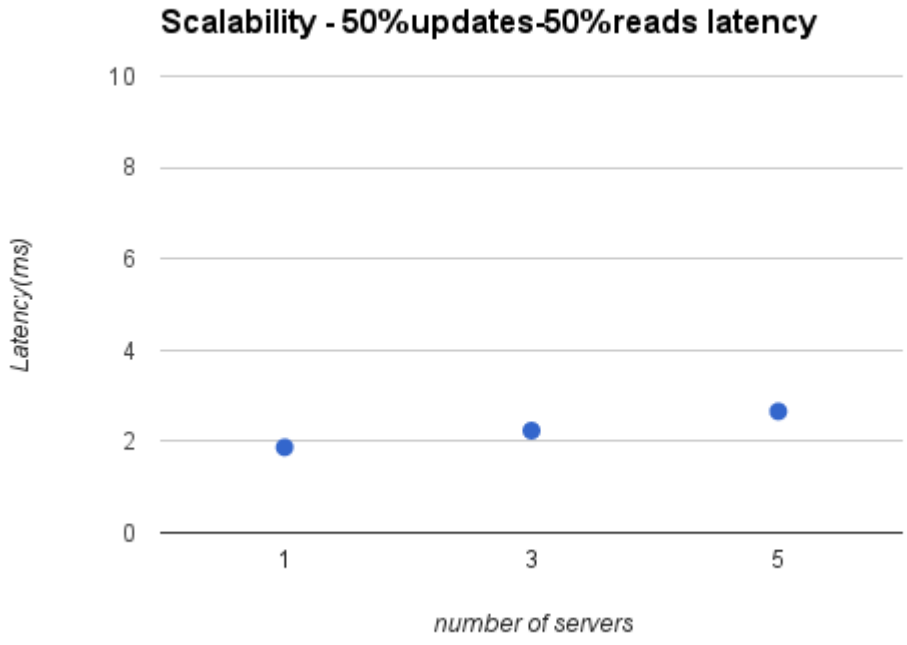

Figure 57 H-Eutropia: Scalability 50% updates - 50% reads latency

5.2.3. Conclusions

In the write workload we are IO bound whereas in the read and mix case we are CPU bound.
As we see from the figures above as more servers are added the system is able to scale its throughput almost linear. The latency observed by the clients remains almost constant as more servers are added and the load is linearly increased. This due to the

fact that the table is splitted into more shards that are distributed in more servers. Clients requests follow a uniform distribution so they take advantage of all the available servers. In this way, the system is able to utilize the extra resources added.

5.3. Scalability of LeanXcale

The evaluation consists in studying the scalability of the LeanXcale database. We study the scalability of the platform in terms of maximum sustainable throughput under a response time constraint. To evaluate the database, we use the industrial benchmark TPC-C™ (TPC-C)[16], the reference benchmark for OLTP databases. TPC-C benchmark stresses key hardware components in database systems like I/O, CPU, memory, and for distributed databases, also network. We have implemented the benchmark application according to the TPC-C specification. The implementation is available at GitHub[17].

We have used a shared-nothing cluster with 464 cores. The cluster is composed of two different types of machines. Machines type A are equipped with 4-core Intel(R) Xeon(R) CPU X3220 @ 2.40GHz, 8 GB of RAM, 1 Gbit Ethernet and a directly attached 150 GB SSD hard drive.

Machines type B are equipped with 12-core Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz,128GB of RAM, 1Gbit Ethernet and a directly attached 0.5 TB SSD hard disk. There are in total 52 machines, 20 type A machines and 32 type B machines. All machines run Ubuntu 12.04 LTS.

5.3.1. TPC-C Benchmark

TPC-C database model is defined in the following figure.

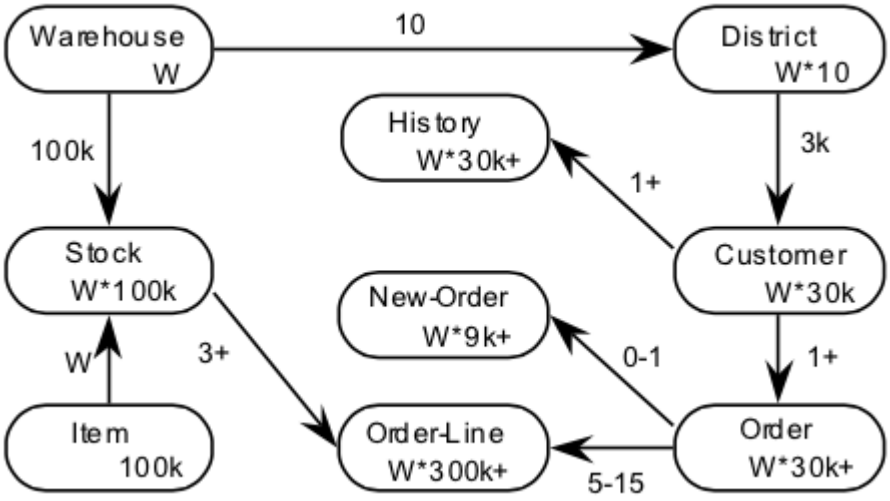

Figure 58 TPC-C schema design

[16] http://www.tpc.org/tpcc/

[17] https://github.com/rmpvilaca/EscadaTPC-C

The database' size and load scale unit is the warehouse. For every warehouse, there are 10 clients accessing concurrently to data in the same warehouse. A 1 warehouse database size (in plain CSV text) is 63.15 MB and it spans 599011 rows.
Scaling the database in terms of size is linear to the number of warehouses.

TPC-C defines five types of transactions:

- **NewOrder**: Inserts a new order with a variable number of items. The transaction performs 2 row selections with data retrieval, 1 row selection with data retrieval and update and 2 row insertions. Then, for a variable number of items (in average 10), performs (1 * number of items) row selections with data retrieval, (1 * number of items) row selections with data retrieval and update and (1 * number of items) row insertions.

- **Payment**: Updates the customer's balance and reflects the payment on the district and warehouse sales statistics. This transaction presents two cases: In the first case, a customer is retrieved based the customer id, then the transactions performs 3 row selections with data retrieval and update; and 1 row insertion. In the second case, the customer is retrived based on the last name. The tansaction performs 2 row selections (on average) with data retrieval, 3 row selections with data retrieval and update; and 1 row insertion.

- **OrderStatus**: Checks the status of a given order. The transaction first picks a customer and his last order. To do so, the transaction defines two cases. In the first case, the customer is retrieved based on the customer id, then the transaction performs 2 row selections with data retrieval. In the second case, the customer is retrieved based on the last name, and then the transaction performs 4 row selections (on average) with data retrieval. Finally, in both cases, the transactions checks the status (delivery date) of each item on the order (on average there are 10 items per order). This operation performs: (1 * number of items) row selections with data retrieval.

- **Delivery**: Process a batch of 10 new (not yet delivered) orders. The transaction performs 1 row selection with data retrieval, (1 + number of items per order) row selections with data retrieval and update, 1 row selections with data update and 1 row deletion.

- **StockLevel**: Determines the number of recently sold items that have a stock level below a specified threshold. The transaction performs 1 row selection with data retrieval, (20 * number of items per order) row selections with data retrieval and at most (20 * number of items per order) row selections with data retrieval.

TPC-C workload states that for every 100 submitted transactions, 45 are NewOrder, 43 are Payment, 4 are StockLevel, 4 are OrderStatus and 4 are Delivery.

5.3.2. Experiment design

We have configured the platform as follows: Zookeeper, HBase Master, HDFS Namenode, Snapshot Server and Commit Sequencer Manager run each one in a type A machine. This number of machines is fixed for all configurations.
We run, 1 Query Engine instance, 2 Conflict Managers instances, 1 Logger instance, 4 HBase RegionServer instances and 1 HDFS DataNode instance (from now on, computing node) per type B machine. TPC-C clients run in type A machines.
We started to measure the capacity of one computing one in terms of database size and number of concurrent clients. We stress the computing node until we reach almost resources (i.e: CPU, IO) saturation while the benchmark SLAs[18] are still met.

In order to measure the capacity of the computing node we populate 50, 100, 200, 300, 400 and 500 warehouses and we inject load from 500 to 5000 clients. We show that a single computing node is able to handle 3000 clients with a database populated with 300 warehouses. After 300 warehouses, the latency grows exponentially because the system is almost saturated. The scarce resource is CPU, reaching around 80 % of utilization with 400 warehouses.

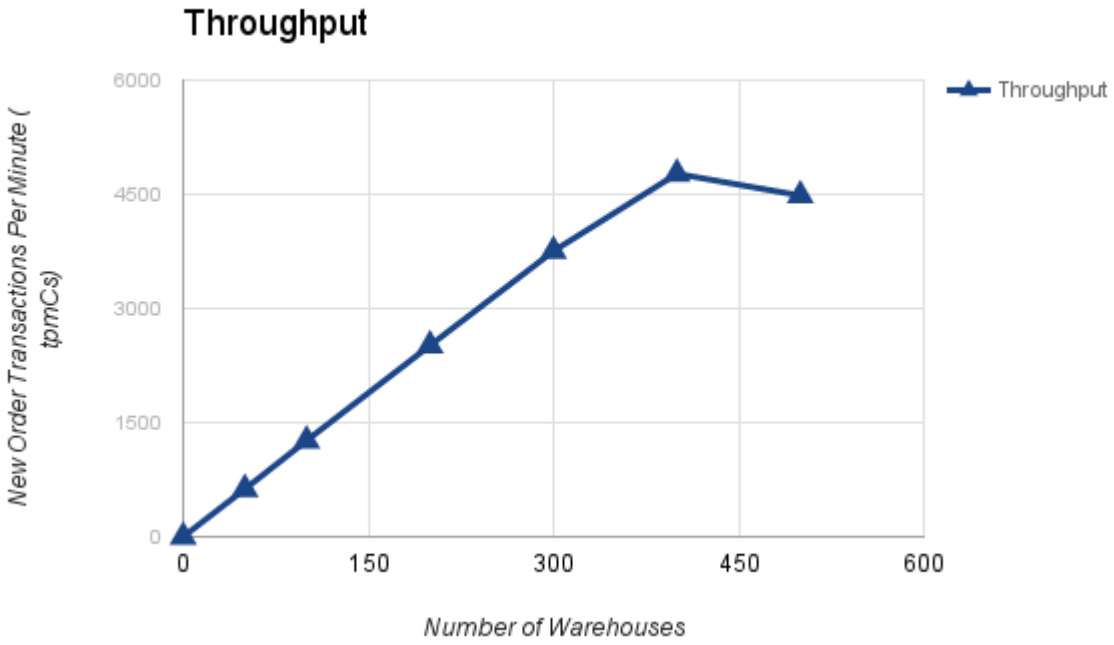

Figure 59 Evolution of the throughput using a single computing node.

[18] http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf

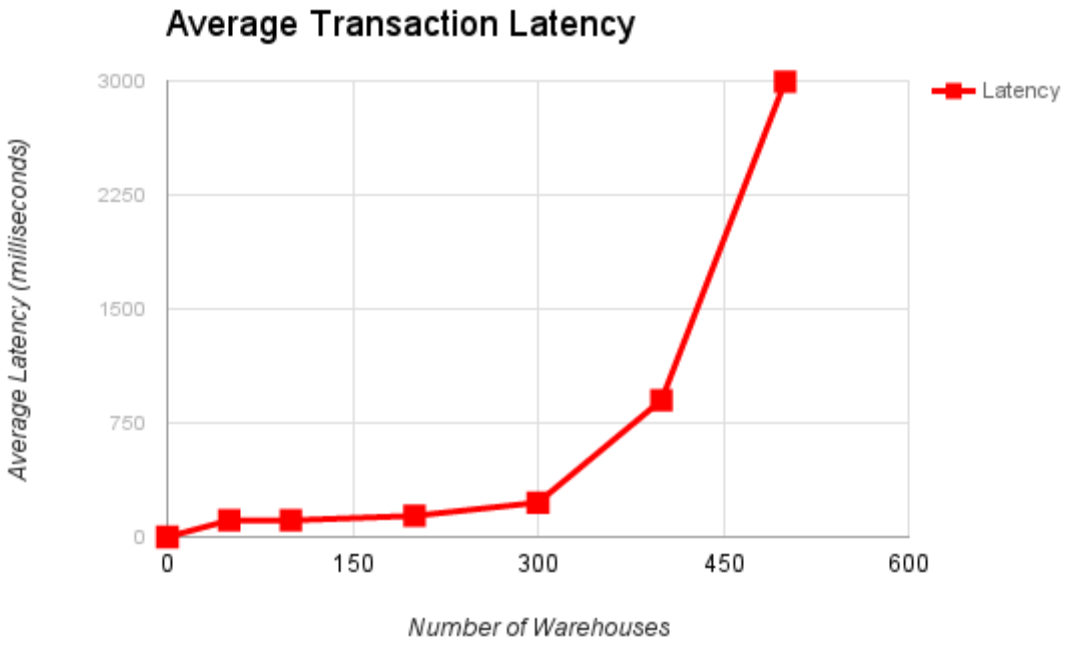

Figure 60 Evolution of the latency using a single computing node.

The next step in our evaluation consists in scaling the number of nodes with the database size and the number of clients. We scale the system to up to 32 computing nodes. We use up to 16 client machines to inject TPC-C workload. Each machine runs up to 2 TPC-C client instances. The evaluation consists in scaling the computing nodes as we scale the database size and the number of concurrent clients in the system. The goal of this evaluation is to prove that LeanXcale scales linearly. That is, increasing the number of computing nodes while we increase the database size and the load the system each computing node is able to process the same load while the latency is not affected. Table 7 shows the different configurations adopted for the selected database sizes and load.

Table 7 TPC-C benchmark scalability

Warehouses	Clients	DBSize (#rows)	DBSize (GigaBytes)	Number of Computing nodes
300	3000	149803300	19	1
600	6000	299606600	38	2
1200	12000	598913200	76	4
2400	24000	1197726400	152	8
4800	48000	2245649500	304	16
9600	96000	4491299000	608	32

5.3.3. Results

We have run from 3000 to 9600 clients in settings with 1 to 32 computing nodes.

Figure 61 and Figure 62 show the evolution of the throughput and latency for the given run. From the figures we can observe that the solution scales linearly. Throughput scales linearly and the latency observed by the clients does not increase as more load is injected. In this evaluation, Snapshot Server and Commit Sequencer machines (type A) presented less than 10% of CPU of utilization in the largest configuration.
For the size of the machine where the Snapshot Server and Commit Sequencer run, the ultimate bottlenecks of the system, LeanXcale could potentially can handle ten times more load. Running these two services in a bigger machine with more powerful hardware this limit could be increased even more.

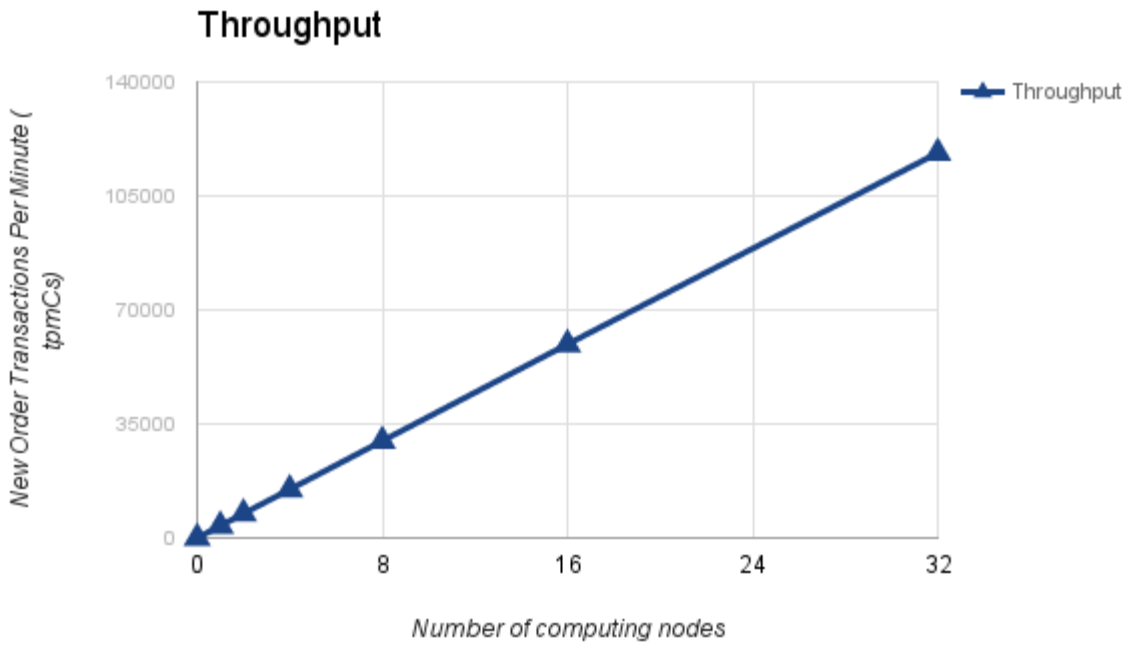

Figure 61 Scalability of LeanXcale – Evolution of throughput at different scales.

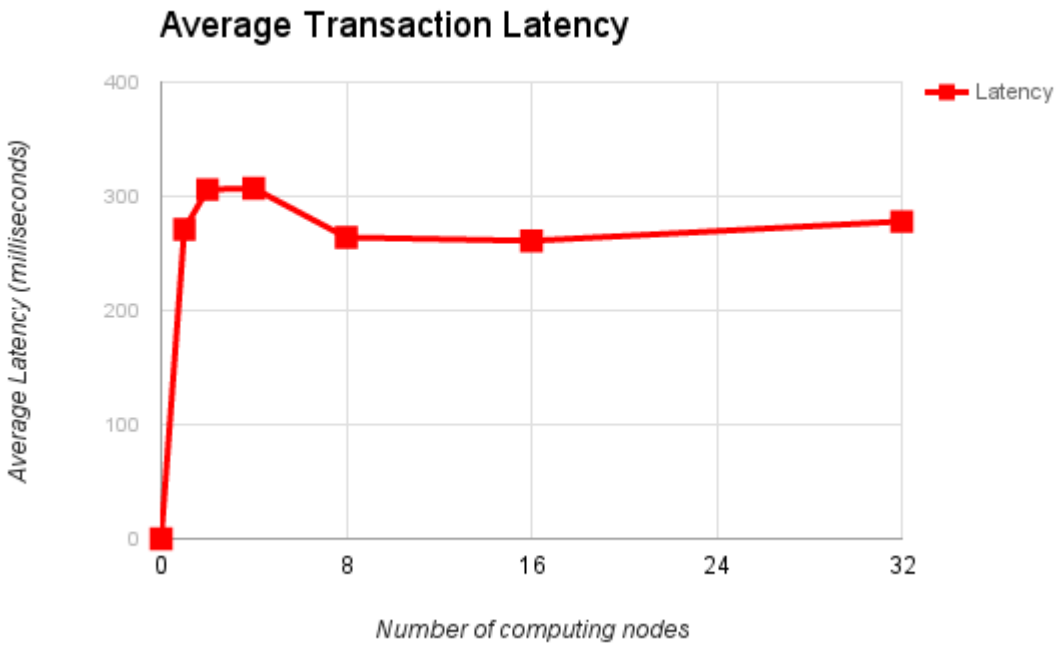

Figure 62 Scalability of LeanXcale - Latency of the system at different scales.

5.4. Scalability of the Holistic Transaction Manager

To evaluate the scalability of the transaction engine, we have developed a micro-benchmark to emulates the TPC-C workload on the transaction manager. We have

implemented all TPC-C update transactions (NewOrder, Delivery and Payment). For each transaction we invoke the functions to start transaction, check for conflicts and commit in the Transaction Manager API. Read-only transactions (OrderStatus and StockLevel) do not exercise the transaction manager, therefore we have skipped them for this evaluation. The goal is to measure the overhead added by the holistic transaction manager to provide transactional semantics on top of the non-transactional storage.
The workload uses the same transaction probabilities of occurrence defined in TPC-C. We have removed thinking and keying times in order to obtain the maximum possible throughput.

5.5. Experiment design

We study the scalability of every server of the transaction manager. For the Local Transaction Manager, we measure how the server scales in terms sustainable throughput measured in transactions per second. For the Conflict Managers, we measure how the server scales in terms of number of conflicts checked per second. Snapshot Server and Commit Sequencer are standalone servers and thus they don't scale. The work performed of these two servers depend on the number of Local Transaction Managers. The work they do per Local Transaction Manager does not depend on the actual load of the Local Transaction Managers. Therefore, we measure how many concurrent Local Transaction Managers they can handle. Since the both the Commit Sequencer and Snapshot Server process batches of timestamps (Commit Sequencer produces them, Snapshot Server processes consumed batches, more details in [4]) we measure the number of batches processed per Local Transaction Manager.
In order to perform this evaluation, we have used a cluster composed of 12-core Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz,128GB of RAM, 1Gbit Ethernet and a directly attached 0.5 TB SSD hard disk. There are in total 16 machines all run Ubuntu 12.04 LTS.
We deploy 1 Local Transaction Manager instance with 1 co-located Logger instance and up to 12 Conflict Manager Instances per machine. Local Transaction Manager/Logger pair and Conflict Managers are deployed in different machines.
For the Snapshot Server and Commit Sequencer evaluation, we have used a single 4-core Intel(R) Xeon(R) CPU X3220 @ 2.40GHz, 8 GB of RAM, 1 Gbit Ethernet and a directly attached 150 GB SSD hard drive.

5.5.1. Local Transaction Manager Evaluation

In order to study the scalability of the Local Transaction Manager, we provision with a Snapshot Server and Commit Sequencer node and as many Conflict Managers are needed in order to determine the capacity of a single Local Transaction Manager. We deploy a single Logger per Local Transaction Manager. Figure 63 presents the obtained throughput for different configurations varying the number of Local Transaction Managers. A single Local Transaction Manager is able to handle 40000 transactions per second. In our experiment, we deployed up to 10 Local Transaction Managers. The resulted throughput in the biggest configuration is close to 360000 transactions per second. Which prove that the Local Transaction Manager is able to scale linearly. Regarding the latency, Figure 64 shows that latency remains constant despite of the load and the number of Local Transaction Managers which suggests that the limit of the number of Local Transaction Manager is far away from the figures we present here.

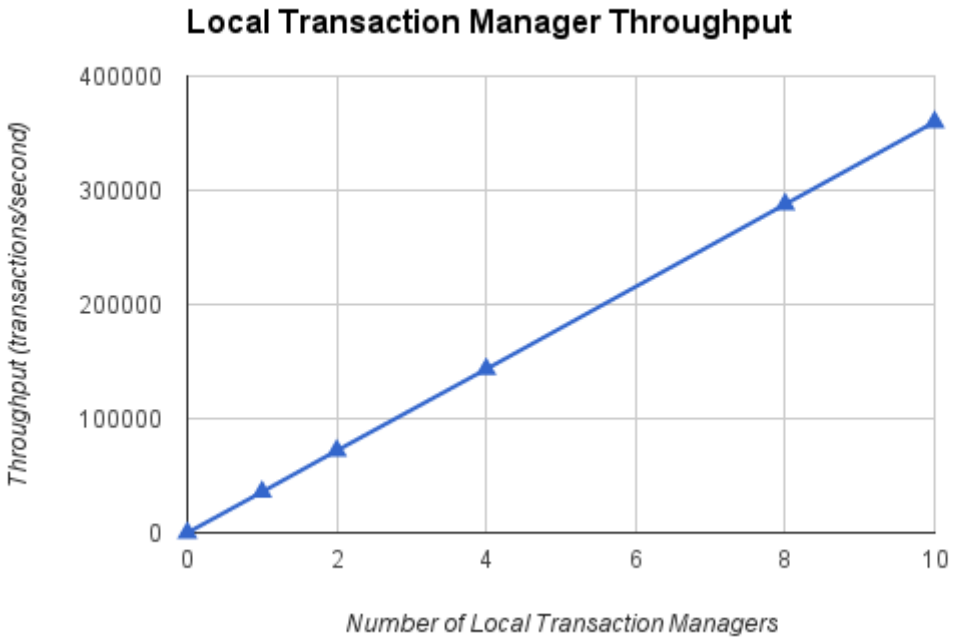

Figure 63 Local Transaction Manager - Evolution of the throughput with different number of Local Transaction Managers

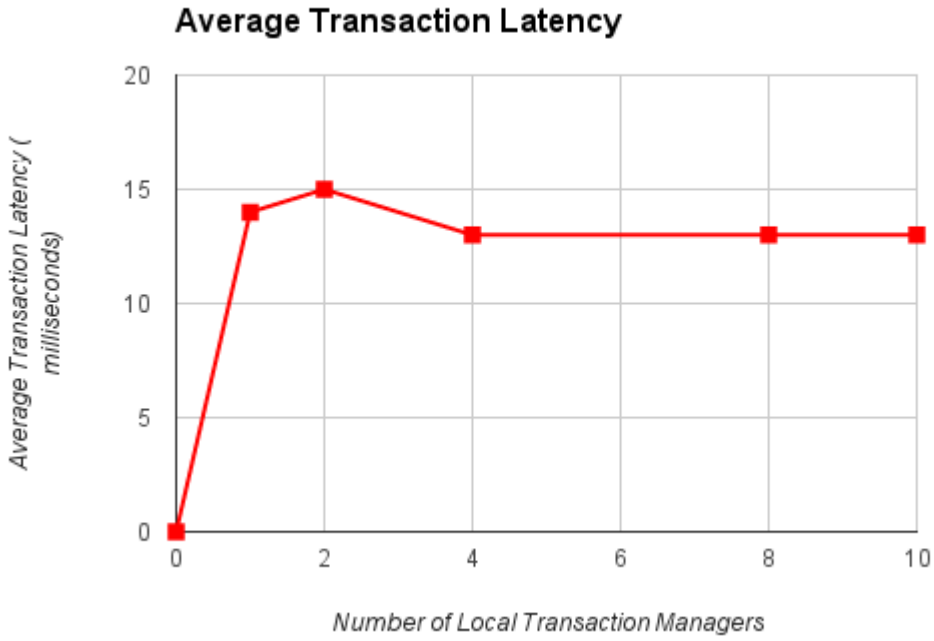

Figure 64 Local Transaction Manager -Evolution of the latency with different number of Local Transaction Managers

5.5.2. Conflict Manager evaluation

In order to study the scalability of the Conflict Manager, we provision with a Snapshot Server and Commit Sequencer node and as many Local Transaction Managers/Loggers are needed in order to determine the capacity of a single Conflict Manager Node. Then we scale the number of Conflict Managers as much as possible in order to find the limitation of our system.

Figure 65 presents the obtained throughput for different configurations varying the number of Conflict Managers. A single Conflict Manager is able to handle 72433 conflicts per second. In our experiment, we deployed up to 120 Conflict Managers. The resulted throughput in the biggest configuration is close to 8728203 conflicts per second. Which prove that the Conflict Manager is able to scale linearly. When it comes to latency, Figure 66 shows that it remains constant despite of the load and the number of Conflict Managers which suggests that the limit of the number of Conflict Manager service is far away from the figures we present here.

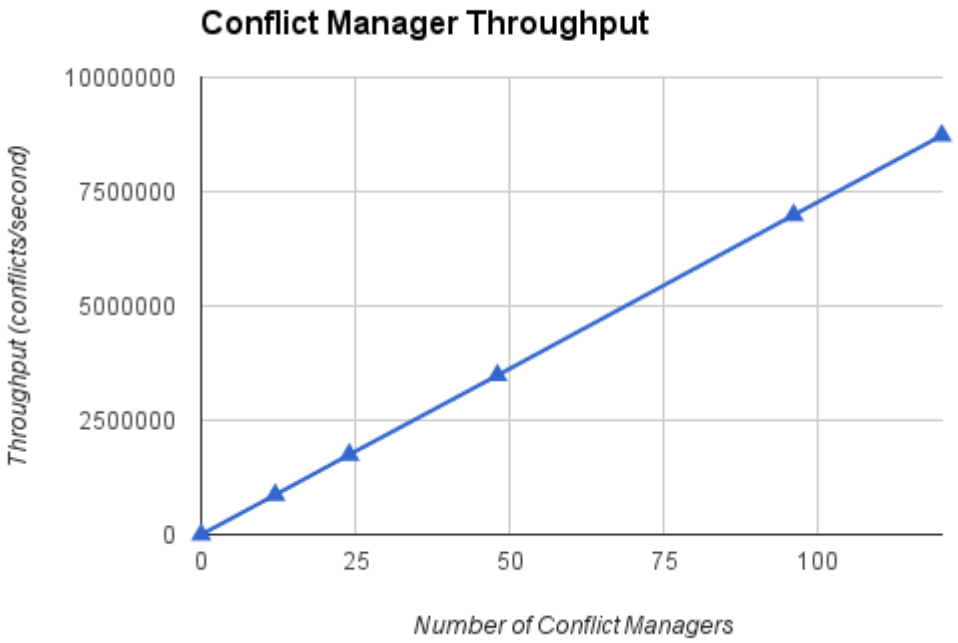

Figure 65 Conflict Manager - Evolution of throughput with different number of Conflict Managers

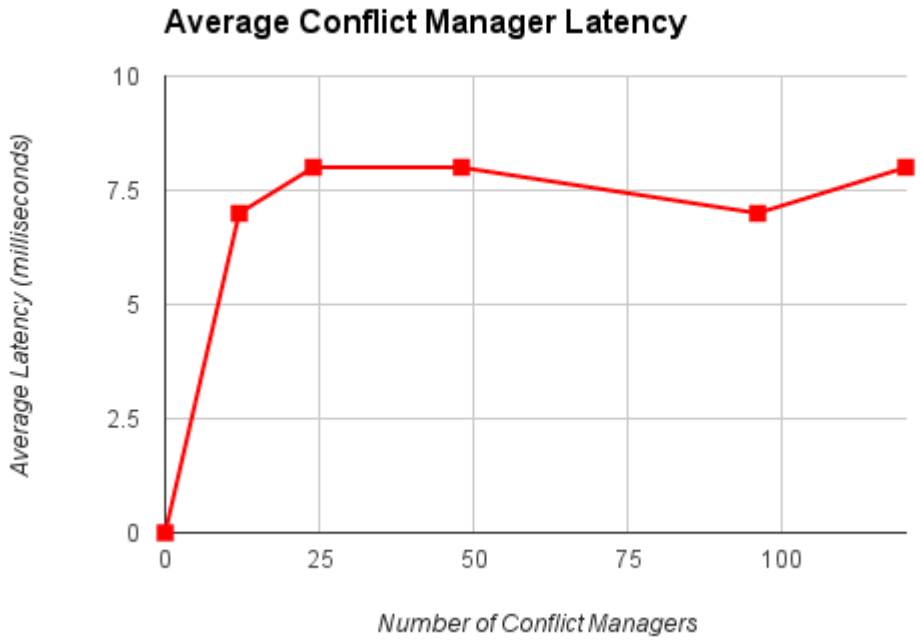

Figure 66 Conflict Manager - Evolution of the latency with different number of Conflict Managers

5.5.3. Snapshot Server and Commit Sequencer evaluation

The study of the capacity of the Snapshot Server and the Commit Sequencer is done through a simulation. As previously mentioned, the capacity of these two services depend on the number of Local Transaction Managers being served. This is because the job done is constant per Local Transaction Manager and it does not depend on the actual load of the Local Transaction Manager. That is, for the Snapshot Server and the Commit Sequencer it represents the same amount of work if the Local Transaction Manager is serving 1 or 1 Million transactions per second. This is because of our fancy design to scale the transaction management. More details can be found in restricted deliverable D4.6 [4].

In order to saturate these two services, the number of machines needed to generate such load would be extremely high (as we will proof later in this section). Our simulation consists in deploying these two components and deploy as many client threads (Snapshot Server and Commit Sequencer spawn a thread for each Local Transaction Manager, plus the server thread which is in charge to process/compute the batches).

Each client thread is orchestrated by the benchmark generating the exact amount of work that a single Local Transaction Manager would generate in the system.

We measure the throughput of the system in terms of computed batches per second. We measure the process batches latency of the servers. The Snapshot Server produces, when the system is under loaded, a new snapshot every 10 milliseconds. We consider the Snapshot Server to be saturated when this latency restriction is no longer met.

Figure 67 shows the evolution of the simulation with an increasing number of client threads (Local Transaction Managers). The Snapshot Server is able to serve linearly up to 800 Local Transaction Managers. When it comes to latency, it is fairly constant (below 10 milliseconds) until 800 clients, as shown in Figure 68.

These results suggest the following deduction, if the Snapshot Server and the Commit Sequencer are able to handle up to 800 clients without trespassing our latency restriction of 10 milliseconds, it means that the Holistic Transaction Manager, when

properly scaled, could be able to serve up to 800 times the throughput of a single Local Transaction Manager. If we take into account the results of our evaluation, where a Local Transaction Manager handles up to 40000 transactions per second, we can safely state that our Holistic Transaction Manager could potentially handle up to **32 Million transactions per second** as we would have reached the limit of the ultimate bottlenecks of the system.

Another interesting conclusion is that the latency increase is up to 15 milliseconds (Figure 64 shows the whole transactional processing latency). As we have seen in Section 5.3.3, the average TPC-C transaction latency is around 300 milliseconds. This means that the overhead in a typical database workload is only around 5 % in latency increase.

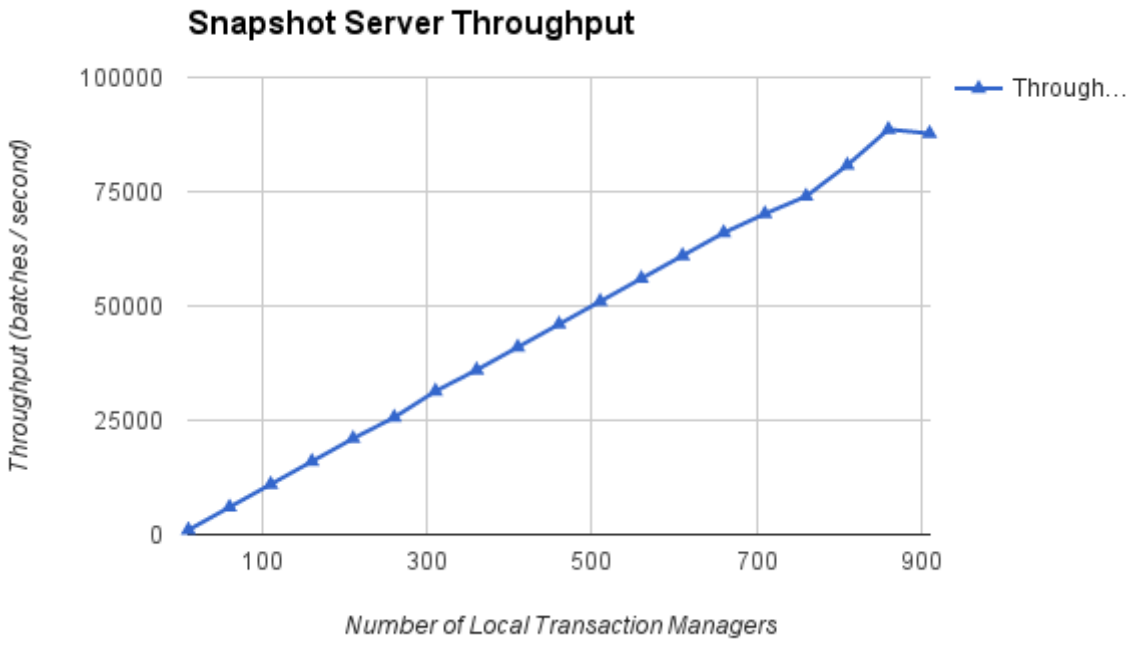

Figure 67 Snapshot Server - Evolution of the throughput with different number of Local Transaction Managers

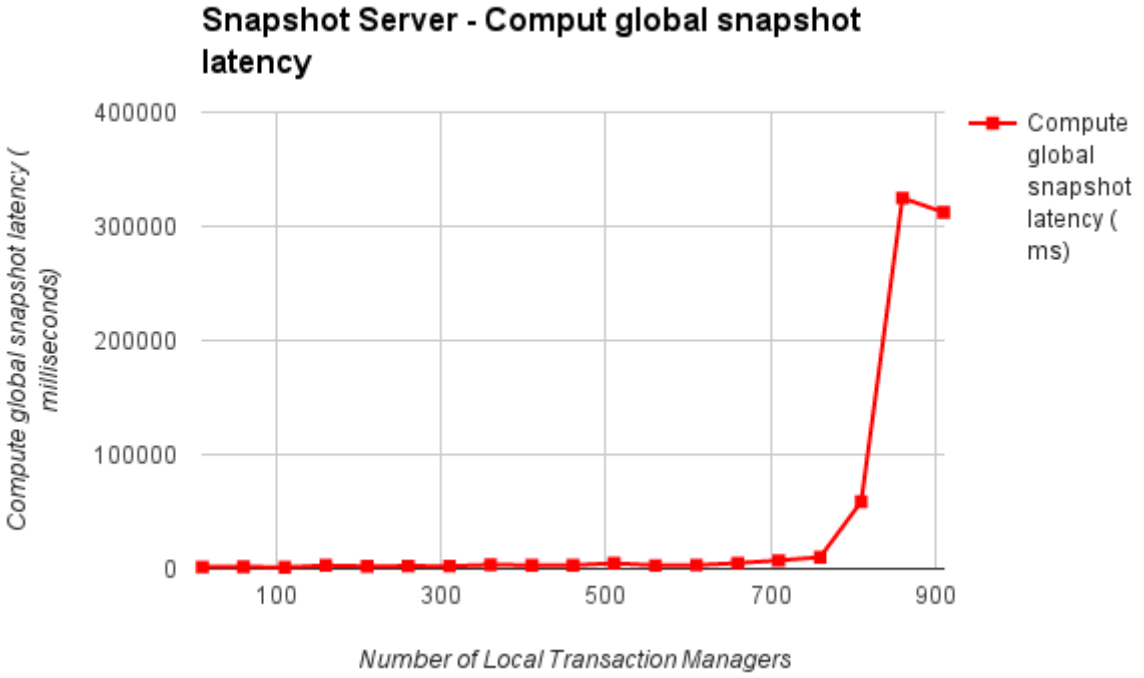

Figure 68 Snapshot Server - Evolution of the latency with different numbers of Local Transaction Managers

5.6. CEP and Eutropia

In this Section we evaluate the performance of the integration between two CoherentPaaS components, the Complex Event Processor and the H-Eutropia key-value data store. In this evaluation we use a synthetic dataset containing tuples with a simplified CDR schema composed by 4 fields labelled idcaller, idreceiver, duration and timestamp. The first two fields identify the caller and the receiver number of a phone-call. The field duration reports the duration of the phone-call in seconds and the field timestamp marks the time the phone-call was made.

In order to process very high input loads, CEP queries can be parallelized. When a CEP query includes operators accessing an external data store parallelize the query in not enough to process these high input loads because the data store access latency could slow down the overall throughput of the CEP. Indeed, in order to be accessed from CEP query, data stores must be able to scale according with the number of CEP operator instances accessing them. With CEP query using only one data store operator, the data store would see only one client connection, instead when the CEP query is parallelized the data store is accessed by concurrent clients that are the different instances of parallel CEP operators.

The goal of these experiments is to show how the CEP can take advantage of the high performance and scalable H-Eutropia data store in order to process massive input load.

We defined two workloads to test the most common interaction between CEP H-Eutropia:

- WORKLOAD-W: 100% PUTs operations: a parallelized CEP query is used to store in H-Eutropia all the incoming CDR records.

- WORKLOAD-R: 100% GETs operations: a parallelized CEP query is used to fetch from H-Eutropia all the records corresponding with keys currying on the stream processed in the CEP.

The evaluation was made at UPM cluster. We used several blades Supermicro SYS-5015M-MF+, each equipped with a quad-core Intel Xeon X3220@2.40GHz, 8GB of RAM and 1Gbit Ethernet and a directly attached 160GB SDD disk. To monitor the nodes in the experiments we used Ganglia Monitor System that is a scalable distributed monitoring system for high-performance computing systems. In particular we used the following deployment:

- 2 nodes for running load generator.
- 5 nodes to run the CEP.
- 1, 3 or 6 nodes to run H-Eutropia.

Load generators are applications able to create tuples representing the CDRs. The load generator is organized in stages. In each stage, it produces tuples with a constant rate. At the end of a stage, the load generator either starts a new stage with an increased rate or stops the experiment if the CoherentPaaS CEP was not able to process in time more than 90% of the tuples sent in the previous stage.

In Figure 69 we see that with WORKLOAD-W the total throughput (number of tuples per second processed in the CEP) when using 6 H-Eutropia nodes (120.000) is almost double than the throughput obtained with 3 H-Eutropia nodes (80.000). Using the same configuration for the CEP we see that the throughput increases because H-Eutropia scales and allows the CEP to process much more tuples using the same resources.

We can also observe that moving from 1 to 3 H-Eutropia nodes the system does not scale linearly. This happens because the distribution of input data is not uniform and some H-Eutropia nodes will receive more load then the others. So throughput stops rising when the first H-Eutropia node is saturated even if the other nodes can receive more load.

In the experiment with WORKLOAD-R, the CEP was able to process a maximum load of 48.400 tuples per second when H-Eutropia was deployed into a single node. In the experiments with H-Eutropia deployed in 3 and 6 nodes, the CEP reached a maximum throughput of around 70.000 tuples per second. In those cases, the limitation was given by the data set and the resources available for the data generator application that were not able to generate a higher load needed for the CEP to take full advantages of the distributed H-Eutropia deployments.

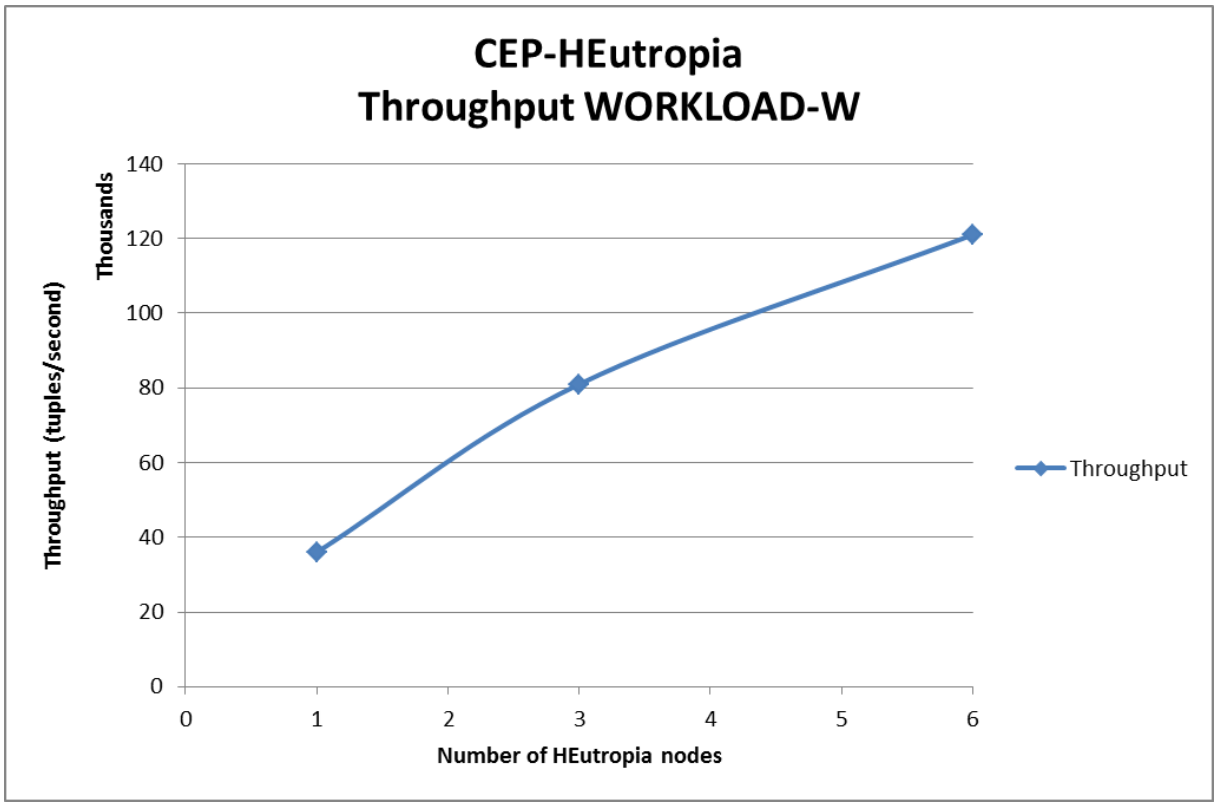

Figure 69 – CEP-HEUTROPIA experiments with WORKLOAD-W

5.7. HTM scalability on number of data stores

The goal to this benchmark is to assess the CoherentPaas HTM scalability with respect the number of data stores. For this experiment we use YCSB and present the same evaluation with each data store individually and with a subset of CoherentPaaS data stores.

5.7.1. Setup

We use 6 machines for this evaluation. All machines are equipped with 4-core Intel(R) Xeon(R) CPU X3220 @ 2.40GHz, 8 GB of RAM and a directly attached 150 GB SSD hard drive. The machines are connected through a standard gigabit Ethernet network. All machines have Ubuntu 12.04 and Oracle JRE 1.7. The components running on each machine were:

- YCSB
- HTM servers
- LeanXcale
- H-Eutropia
- MongoDB
- ActivePivot

5.7.2. Database

The source data is the default dataset of this benchmark, with different numbers of rows where each row has one key column with a length of 100 bytes and 10 value columns

with 100 bytes each. As we want to compare the scalability of CoherentPaaS HTM in the number of data stores we used a fixed dataset size of 100 thousand rows. We populate the data to be evenly split according to the number of data stores to be used in that experiment. So with a single data store experiment all data is populated to that single data store and with 4 data stores experiment each data store have ¼ of the total rows.

5.7.3. Evaluation

We ran a fixed workload using the fixed data set and a single client. To evaluate the HTM overhead an update workload was more suited and thus we used a workload with 90% READS and 10% UPDATE (update all columns). Rows are selected according to the uniform distribution with a fixed batch size of 20 (i.e. 20 operations per transaction, following the proportions given before). As a given experiment may contain several data stores we have modified YCSB to round robin operations between the data stores in use for a given experiment. So in an experiment with a single data store it receives all operations while for an experiment with two data stores, D1 and D2, a set of sequential operations O1 to O10: D1 will receive the odd operations while D2 will receive the even operations.

For this experiment we needed to define a common workload and data set for all data stores. Therefore, only read and full-updates are considered. For this experiment we selected H-Eutropia, and Active Pivot

We present the latency and throughput for each individual data store and with all selected data stores, in order to evaluate the overhead of having HTM with a single data store or with several. The results are shown in the graphs below. Figure 70 and Figure 71 demonstrate that the latency and throughput do not get affected. In the experiment with only H-Eutropia, latency is around 30 ms. With ActivePivot latency is around 320 ms. When H-Eutropia and ActivePivot run together, half of the operations is handled by H-Eutropia and the other half is handled by ActivePivot. It is expected that latency for the mixed scenario is an average between the latencies obtained for each data store individually. As Figure 70 illustrates, the transaction latency for the mixed scenario is around 180 ms which is expected.

5.7.4. Conclusions

This experiments proves that HTM is able to handle different type of data stores in the same setting without interfering with the performance of each of them.

This evaluation complements the HTM overhead evaluation described in Section 3, HTM overhead evaluation proved that the overhead induced by the HTM to the data stores is negligible. The results presented in this evaluation asses that HTM does not impact neither in the individual performance of the data stores neither in the global performance of the system when more than one data store is present.

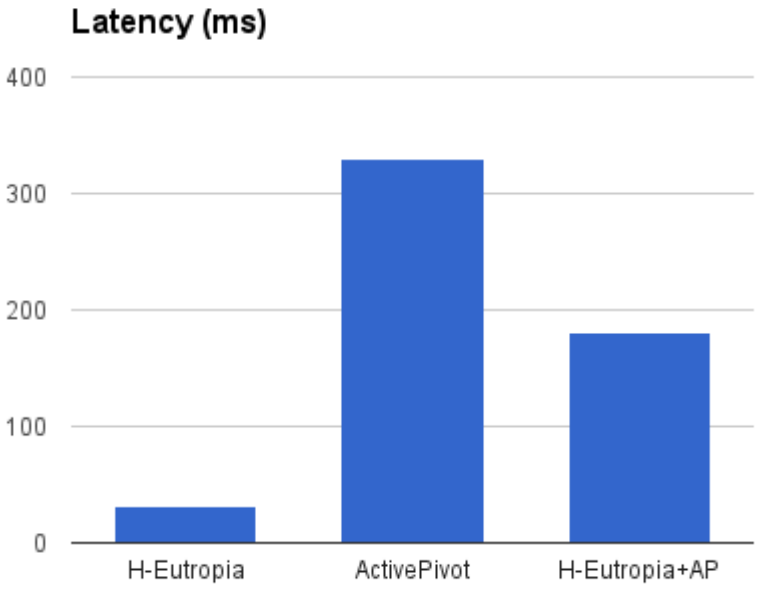

Figure 70 HTM Scalability -Latency

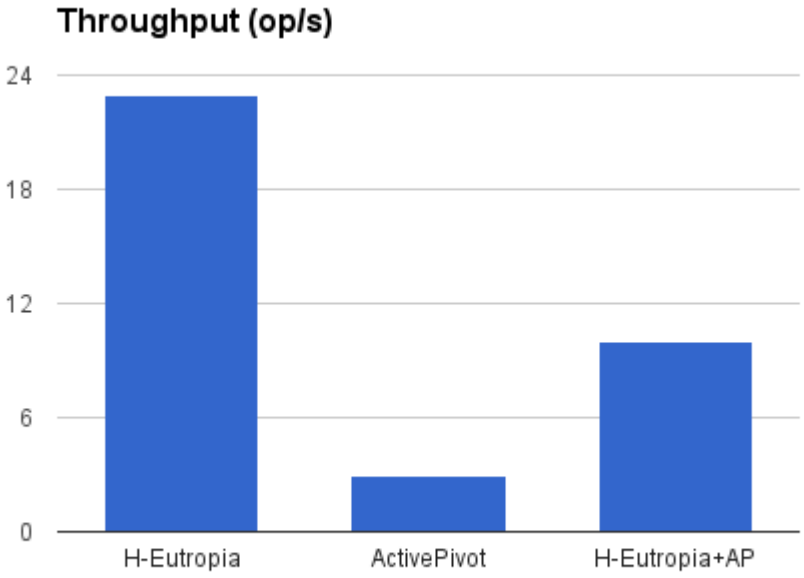

Figure 71 HTM Scalability Throughput

References

[1]. Marta Patiño Martínez et al. D4.2 Local Transaction Manager API. 2014.
[2]. Marta Patiño Martínez et al. D4.3 Local Transaction Manager for all cloud data stores. 2015.
[3]. Marta Patiño Martínez et al. D4.4 Recovery Management for all cloud data stores. 2015.
[4]. Marta Patiño Martínez et al. D4.6 Holistic Transaction Manager. 2014.
[5]. Pavlos Kranas et al. D5.2 No-SQL Data Stores Implementation v1. 2014.
[6]. Pavlos Kranas et al. D5.3 No-SQL Data Stores Implementation v2 2015