Under Pressure Benchmark for DDBMS Availability

Alessandro Gustavo Fior[1], Jorge Augusto Meira[1,2], Eduardo Cunha de Almeida[1]
Ricardo Gonçalves Coelho[3], Marcos Didonet Del Fabro[1], Yves Le Traon[2]

[1] Universidade Federal do Paraná, Brazil
{alessandrog, eduardo, marcos.ddf}@inf.ufpr.br
[2] University of Luxembourg, Luxembourg
{jorge.meira, yves.letraon}@uni.lu
[3] Universidade Estadual do Norte do Paraná, Brazil
rgcoelho@uenp.edu.br

Abstract. The availability of Distributed Database Management Systems (DDBMS) is related to the probability of being up and running at a given point in time and to the management of failures. One well-known and widely used mechanism to ensure availability is replication, which includes performance impact on maintaining data replicas across the DDBMS's machine nodes. Benchmarking can be used to measure such impact. In this article, we present a benchmark that evaluates the performance of DDBMS, considering availability through replication, called Under Pressure Benchmark (UPB). The UPB measures performance with different degrees of replication upon a high-throughput distributed workload, combined with failures. The UPB methodology increases the evaluation complexity from a stable system scenario to a complex one with different load sizes and replicas. We validate our benchmark with three high-throughput in-memory DDBMS: VoltDB, NuoDB and Dbms-X.

Categories and Subject Descriptors: H.2 [**Database Management**]: Miscellaneous

Keywords: availability, benchmark, distributed databases, failure, replication

1. INTRODUCTION

The availability of a distributed system is related to the probability of its being up and running at a given point in time. In the context of Distributed Database Management Systems (DDBMS), the definition of availability may be extended to the capacity to manage failures [Özsu and Valduriez 2011] with low impact. There are different capabilities related to failures management, such as keeping the system operational during failures, recovering with efficiency and ensuring that no data is lost.

The large majority of applications have availability requirements (i.e, fully operational) around 99,99% [Gray and Siewiorek 1991]. In some critical systems, such as telecommunication systems, the availability rate grows up to 99,999%. To provide this rate, it is necessary to use robust mechanisms and to consider their performance impact. Two important mechanisms used by DDBMSs to ensure availability are replication and recovery. However, the use of these mechanisms has a direct impact on the performance, which may be more or less important depending on the rate of availability.

The performance of DDBMS is analyzed through the execution of a benchmark, in which different workloads are submitted and the execution time is measured. To provide basis to compare different systems, the benchmark must follow well-defined evaluation methods [Jain 1991]. Several benchmarks have been proposed focusing on comparing different DBMSs. However, none of them are suitable

Supported by the Fonds National de la Recherche, Luxembourg 902399, CNPQ grant 481715/2011-8, Fundacao Araucaria grant 20874 and SERPRO.

for DDBMS, since they do not include availability requirements. In one hand, the benchmarks that evaluate availability [Vieira and Madeira 2003; Li and Levine 2012] focus on centralized DBMS. In other the hand, the Yahoo Cloud Serving Benchmark (YCSB) [Cooper et al. 2010] evaluates performance and scalability issues of cloud-based systems. However, it does not take into account availability in the current specification.

In this article, we present the Under Pressure Benchmark (UPB) that evaluates the performance of DDBMSs considering availability through synchronous replication[1]. The goal of UPB is to measure how the performance of DDBMSs is affected by applying different degrees of replication and failures. The UPB methodology increases the evaluation complexity from a stable system scenario up to a faulty system scenario upon different load sizes and replicas. The scenarios are designed to evaluate the following different settings: (1) no fault tolerance, no failures; (2) fault tolerance, no failures; (3) fault tolerance with failures. For each scenario, the benchmark generates a set of partial metrics measuring the DDBMS performance. These partial metrics are summarized to provide a unique availability index. We conducted experiments applying our benchmark on three different high-throughput in-memory DDBMS: VoltDB, NuoDb and Dbms-X.

This article is structured as follows. Sections 2 and 3 present our benchmark and methodology, respectively. Section 4 describes our validation through experimentation. Section 5 discusses related work. Section 6 concludes the article.

2. THE ENVIRONMENT OUTLINE

We present in this article a benchmark called Under Pressure Benchmark (UPB). It evaluates one important mechanism related to DDBMS availability [Özsu and Valduriez 2011]: replication. While data replication is up and running, the impact on performance may vary depending both on the evaluation environment and the proper implementation that differ from DDBMS (e.g., asynchronous or synchronous). The UPB seeks to assess the efficiency and performance impact of replication with different configurations and workloads. The workload varies up to extreme situations in terms of faults and bulk load.

A set of partial metrics computes performance along the execution of the UPB, for instance, the number of transactions per second with or without node failures. At the end, a final metric sums up the partial metrics to present the overall performance of the DDBMS under pressure. We present the main characteristics of UPB as follows:

—Portability: UPB can be applied to any DDBMS with minor adaptations.
—Adaptability: The UPB methodology allows using different workloads.
—Flexibility: Since UPB provides a set of partial metrics, one may use subsets for specific proposes, such as, analyzing different configurations or applications.
—Extensibility: The methodology supports to be extended to consider different DDBMS features, by incorporating new metrics.

2.1 Architecture

The UPB architecture is designed to fit a classical DDBMS architecture. It is composed by client machines and a cluster of distributed machine nodes to run the DDBMS(see Figure 1).

2.1.1 *Clients.* The connection between the DDBMS and the clients depends on the DDBMS. The clients connect and submit transactions to any cluster node or all the cluster nodes at the same time with the DDBMS connector (i.e. driver) taking care of load balance.

[1]Since the evaluated DDBMSs ensure the ACID properties, we assume a synchronous replication.

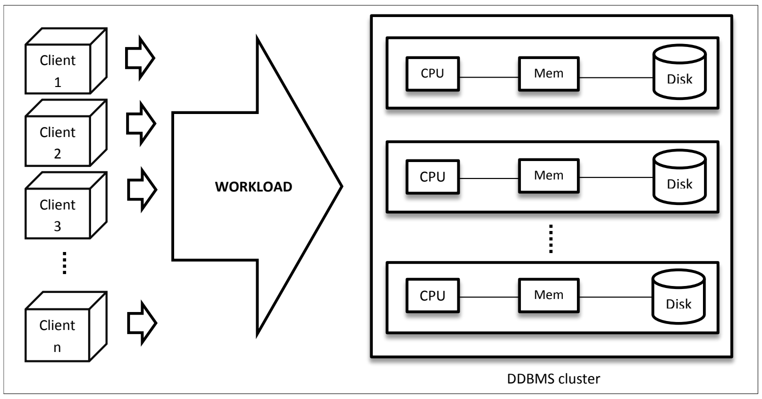

Fig. 1. UPB architecture

UPB has a stepwise methodology to drive its execution, in which the number of clients submitting transactions grows at each step. Two configuration knobs are required to limit the number of clients and their throughput (in transactions per seconds). The number of clients may vary from DDBMS, even if they are running in the same environment, and a tuning procedure must be taken to draw the best fit number and avoid bungles to spoil the final result. The tuning procedure includes leveraging the results of the partial metrics to figure out the proper configuration.

For an Online Transaction Processing (OLTP) application running in a real environment, there is no throughput limitation for clients, since they are distributed all over the Wide Area Network (WAN) and do not run in the same machine. Otherwise, mimicking a large number of clients may saturate the client machine increasing latency of the transaction requests. As consequence, the DDBMS may never reach its performance boundary and the final performance result may be spoiled as well. To scale out the benchmark and avoid any client contention, we implemented our benchmark to run across distributed machines [Meira et al. 2012].

2.1.2 *Fault Tolerance.* Data replication is the main mechanism for fault tolerance in DDBMS with two main strategies implemented by the DDBMS: asynchronous updating of replicas (i.e., optimistic strategy) and synchronous updating (i.e., pessimistic strategy). For those strategies, a replication factor defines the number of nodes that could be out of operation without data loss or service outage.

The replication factor is configured following the availability requirements given by the application. According to the rules-of-thumb [2], to size the hardware based on data replication, the number of cluster nodes must be a multiple of the number of copies.

The UPB goal is to evaluate the performance impact on the DDBMS while a number of its nodes is unavailable. It does not evaluate the impact of different failures. In this context, our understanding follows the definition of *fail-fast* presented by Gray and Siewiorek [1991] in which any system module should operate perfectly or stop immediately. It means that any failure in a node, whether hardware or the DDBMS itself, is enough to immediately disrupt the node.

2.1.3 *Workload.* In this work we leverage the YCSB [Cooper et al. 2010] workload generator, but the UPB accepts any workload based on the transaction concept. Transactions could be composed by simple database operations (e.g., reads, writes, deletes) or complex business operations, such as, the transactions of the TPC-C and the TPC-E [3]. The only constraint is that at least 90% of the transactions must complete in less than 1 second. This requisite is normally used by well-known

[2]http://www.voltdb.com/
[3]http://www.tpc.org/

Table I. *Scenarios*

Combination	K	F	Relevance	Comment
1	0	0	Yes	Basis to scenarios comparison
2	0	Y	Unfeasible	$F > K$
3	X	0	Yes	Performance impact of fault tolerance mechanism
4	X	Y	Yes	Performance impact of failures

benchmarks to cope with real applications [4]. From now, we denote this constraint as ϕ.

3. METHODOLOGY

In this section we present the UPB methodology. It is divided in three tasks: (1) defining the availability scenarios, (2) specifying a steady state and (3) executing the scenarios. These tasks are explained in the following.

3.1 Defining the Availability Scenarios

We define scenarios that represent the set of possible states wrt. availability. The scenarios are chosen based on the combination of values of two variables, as described below:

(1) Fault tolerance index (K): quantity of "failed nodes" supported by the cluster without service outage. The possible K values are:
   - —$K = 0$ (no fault tolerance): the service stops in presence of any node failure.
   - —$K = 1, 2, ..., \frac{N}{2}$: N represents the nodes that compose the DDBMS cluster. In this case, the DDBMS supports failures in K nodes. The values vary between 1 to $\frac{N}{2}$.

(2) Number of failed nodes (F).
   - —$F = 0$: cluster without failures.
   - —$F = 1, 2, ..., K$: cluster with K "failed nodes". The failed values are between 1 to K.

However, it is not necessary to have one scenario for all combinations of variable values, since some scenarios cannot occur in practice. Table I shows possible values of the variables, a short explanation and the relevance of each combination. Since some values depend on the scenario, X represents the values assumed by K ($1 \leq X \leq \frac{N}{2}$) and Y represents the values assumed by F ($1 \leq X \leq K$) in different scenarios.

Following the combination analysis, there are three interesting scenarios that will be applied by UPB:

—Scenario (1) - combination 1: No fault tolerance ($K = 0$), no failures ($F = 0$)
—Scenario (2) - combination 3: Fault tolerance ($K > 0$), no failures ($F = 0$)
—Scenario (3) - combination 4: Fault tolerance ($K > 0$), failures ($0 < F \leq K$)

Since the values of K and F vary, each scenario has one or several steps to evaluate all the possibilities.

3.2 Specifying the Steady State

In this section we explain how to define a steady state, which represents a sustainable execution state of the DDBMS. The overall performance of the DDBMS varies about 2 percent during the steady

[4]http://www.tpc.org/tpce/

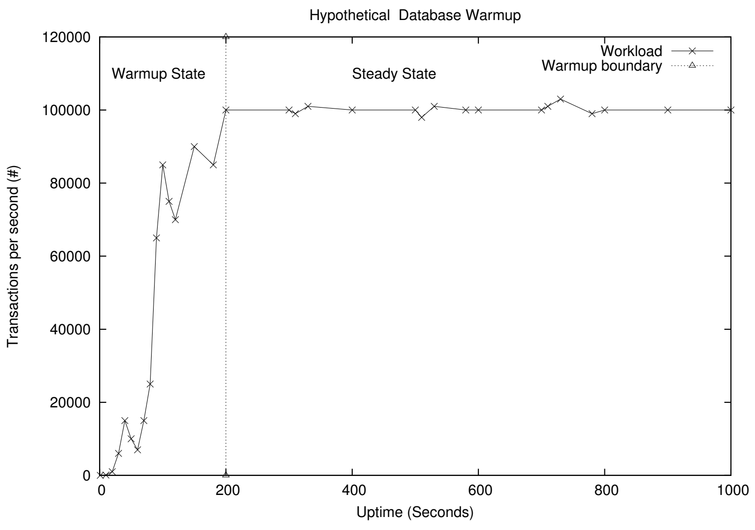

Fig. 2. Warming period and steady state

state [5]. Before reaching such state, it is necessary to pass through a warming period, preventing any unstable behavior (see Figure 2).

Similarly to the TPC-C benchmark, the warming period is set up by the responsible for applying the benchmark. The UPB performance measures must be the transactions per second (*tps*) average during a period of 2 minutes in a steady state. This period is defined empirically, with the objective to measure the system behavior and to normalize possible punctual performance instabilities. The performance is measured by monitoring the number of *tps*, collected during the steady state.

To guarantee an accurate measurement, each client has a Q configuration to limit their workload throughput (i.e., transactions per second). It prevents from having a stressed environment on the client side (i.e., client contention) that could generate an unstable performance. The maximum workload a client is able to submit with no contention on the client side (i.e., the maximum value allowed for Q) is represented by L_c. To define L_c we apply an iterative method:

(1) Define the low load (Q) respecting the latency ϕ;
(2) Measure the performance (*tps*):
IF $Q * 0.95 \leq tps$ *THEN* $Q = Q * 1.10$, repeat step 2 *ELSE* $L_c = Q * 0.90$;

To exemplify this method, let's assume that an hypothetic client is set to submit initially 10 tps ($Q = 10$). The cluster is evaluated with this load and it processes all the transactions. The load of the client is increased by 10% until the difference between the cluster performance and the Q configuration of the client is less than 5%. Consider that the load is increased until $Q = 174$ tps, but the cluster processes just 150 tps. In this case, the difference between the performance expected and achieved is higher than 5%. The L_c is this limit achieved decreased by 10% – in this illustration, 156 tps.

3.3 Executing the Scenarios

UPB is executed iteratively, in three steps, one per scenario. Each step may be executed more than one time depending on the max value of the fault tolerance index(K). The partial metrics are denoted by $T_{K,F}$ (F represents the "failed nodes").

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

Step 1:
In this step the fault tolerance and failures are not taken into account. In real environments this configuration is not recommended, but in our methodology it is important to set a basis for comparison. This configuration is used to define the max performance reached by the cluster, since there is no additional costs related to replication and/or failures (i.e., K=0, F=0). The max performance is represented by $T_{K,F}$ (see Algorithm 1).

```
input  : Q, a set of client workloads; $L_c$, maximum workload per client
output: $T_{K,F}$
foreach $q \in Q$ do
    $q = L_c * 0.10$;
    while $(Qtotal + q) * 0.95 \leq tps$ and $q \leq L_c$ do
        $lastQ = q$;
        $q = q * 1.10$;
    end
    if $(Qtotal + q) * 0.95 \leq tps$ then
        $Qtotal = Qtotal + q$;
    else
        $T_{K,F} = Q_{total} + lastQ$;
        return $T_{K,F}$
    end
end
```

Algorithm 1: Baseline

Step 2:
The fault tolerance mechanisms have an impact on the system performance even without node failures. This impact varies depending on the implementation and strategies used by each DDBMS. The step 2 aims to verify and to measure these impacts.

The fault tolerance index is configured with different values($K > 0$). The node failure index(F) is not used. The goal is to measure the data replication impact on performance. For each K value, the max performance is calculated, using the Algorithm 1, and represented by:

$$D_{K,0} = (1 - \frac{T_{K,0}}{T_{0,0}}) * 100$$

Step 3:
This step measures the performance during a failure state, with a fault tolerance mechanism activated. The DDBMS must be operational during node failures. In this case, the performance may be impacted by read workloads (the replicated data is used to maximize the throughput) and write workloads (the replicated data must be synchronized, implicating in additional costs depending on the K value).

The number of failures vary up to the number of replicated nodes: $1 \leq F \leq K$

For each F value, the performance is calculated, using the Algorithm 1, and related to $T_{K,0}$ obtained in the step 2. Thus, performance degradation is defined by:

$$D_{K,F} = (1 - (\frac{T_{K,F}}{T_{K,0}})) * 100$$

The final degradation metric(DF_K) is given by summarization of the metric $D_{K,F}$

Table II. Overall Metrics

Metric	Step	Description
$D_T = \frac{\sum_{i=1}^{K} D_{i,0}}{K}$	2	Performance degradation with fault tolerance
$D_F = \frac{\sum_{i=1}^{K} DF_i}{K}$	3	Performance degradation during failures

$$DF_K = \frac{\sum_{F=1}^{K} (\frac{1}{F} * D_{K,F})}{\sum_{F=1}^{K} (\frac{1}{F})}$$

We used a weighted average to calculate the global degradation considering that simultaneous failures are less common than unique failures. Thus, the weight of simultaneous failures on the metric are lower.

3.4 Overall Metrics

We define two overall metrics that summarize the partial metrics calculated. Each metric is the average value obtained for one kind of metric, over the K index, as shown in Table II. We do not use step 1 to calculate an overall metric, since it is used only as basis for comparison.

These two metrics are the final results of the performance analysis of UPB.

4. EXPERIMENTS

In this section we present experiments performed by applying the UPB in three different DDBMSs. Our goal is to validate the proposed benchmark and its methodology as a robust approach to compare DDBMSs's availability. The experiments are conducted by following the three steps of UPB.

4.1 Experimental Setup

The experimental evaluation was performed in three high-throughput DDBMSs: VoltDB, NuoDB and Dbms-X. In order to provide a fair comparison, all the experiments were performed in the same environment, the Grid'5000 testbed [6]. The experimental setup is described bellow:

—Intel Xeon E5440 QC (2.83 GHz / 4 MB), 2 sockets, 4 cores per socket

—Memory 8 GB

—Network Myri-10G (10G-PCIE-8A-C)

—Debian GNU/Linux Lenny x64

—Java$^{\text{TM}}$SE Development Kit 7, Update 17

—YCSB-0.1.4

—DDBMS: VoltDB v2.8.4.1 Community Edition, NuoDB Starlings Release 1.1, Dbms-X

[6]http://www.grid5000.fr

```
CREATE TABLE usertable (

       YCSB_KEY VARCHAR(1000) NOT NULL,
        FIELD1 VARCHAR(1000), FIELD2   VARCHAR(1000),
        FIELD3 VARCHAR(1000), FIELD4   VARCHAR(1000),
        FIELD5 VARCHAR(1000), FIELD6   VARCHAR(1000),
        FIELD7 VARCHAR(1000), FIELD8   VARCHAR(1000),
        FIELD9 VARCHAR(1000), FIELD10 VARCHAR(1000),
       PRIMARY KEY (YCSB_KEY)

);
```

Fig. 3. Database schema

```
SELECT  FIELD1, FIELD2, FIELD3, FIELD4, FIELD5,
        FIELD6, FIELD7, FIELD8, FIELD9, FIELD10
FROM    USERTABLE
WHERE   YCSB_KEY = :1
```

Fig. 4. Read operation

We used 6 server-class machines running the DDBMS and three to run the clients. In order to avoid any interference, the clients and servers were run in separate machines.

To load the database and generate the workload, we used the YCSB framework [Cooper et al. 2010]. The database schema is composed by a single table with 11 columns (see Figure 3).

Each column stores 1,000 alphanumeric characters, totalizing 1,100 bytes per record. In our experiments, the database was loaded with 1 million tuples. The workload was based on read operations that perform a select using the primary key and columns projection (see Figure 4). The YCSB workload engine has been configured to use a Zipfian query distribution [Gray et al. 1994].

NuoDb and Dbms-x send SQL synchronous requests through JDBC calls. VoltDb, however, needs to have an asynchronous connection to achieve better performance [7]. For this reason, we developed a specific asynchronous driver on top of YCSB.

Each client machine runs 32 threads to generate and to submit the workload. For the DDBMS cluster, both VoltDB and Dbms-X use similar configuration. The cluster has 6 nodes running 6 sites, totaling 36 partitions. However, the NuoDB implementation doesn't support data partitioning. Every NuoDB node can play three different roles: Broker (i.e., manages access and control of transaction engines and storage managers), Transaction Engine (TE)(i.e., provides access to a single database. It handles requests from clients, caches data, and coordinates transactions) and Storage Manager (SM)(i.e., each storage manager is associated with exactly one database). Thus, each layer can scale independently.

[7]http://www.voltdb.com/

Table III. NuoDB configuration

Configuration	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6
K=1	Broker	TE	TE	TE	TE	TE & SM
K=2	Broker	TE	TE	TE	TE & SM	TE & SM
K=3	Broker	TE	TE	TE & SM	TE & SM	TE & SM
K=4	Broker	TE	TE & SM	TE & SM	TE & SM	TE & SM

Table IV. *Defining the workload limit per client (L_c)*

Client Configuration (Q)	VoltDB Performance (tps)	Variation
70.000	69.999	0,0%
80.000	80.005	0,0%
90.000	90.039	0,0%
100.000	94.543	5,5%
110.000	94.266	14,3%

The NuoDB architecture supports one or more TEs, on a single node or across the cluster. The NuoDB Documentation [8] recommends to add TEs to improve performance. Thus, during the experiments, we maximize the number of TEs to improve the requests management. For each database, one SM is required (i.e., depending the number of replicas the number of SMs is increased). The node Broker is fixed in one node. Therefore, the NuoDB cluster assumes different configurations based on the fault tolerance index(K), as shown in Table III.

4.2 Step 1

UPB has two parameters that determine how the experiments should be executed: Warm-up time and maximum workload per client (L_c). These parameters are related to the environment and must be defined in the first run. The way to obtain these parameters is quite similar for any DDBMS. For this reason, we only present the results from VoltDB. After that it is possible to obtain the maximum performance of the cluster without fails ($F = 0$) and configured with $K = 0$.

4.2.1 *Parameters Definition.* To get the workload limit per client (L_c), we evaluate the cluster performance by setting a client with different Q values. The Table IV presents the sustained performance of the VoltDB cluster and indicates the variation between the measured performance and Q configuration of the clients (maximum throughput a client could submit).

When Q parameter is less than 90.000 tps, the client is capable to submit the workload determined by Q. In these situations, the difference between Q and the real performance is pretty close (less than 0,04%). However when Q is higher than 90.000, the clients can't submit more than 95.000 tps. In that situation the difference between the performance and Q is higher than the 5% (threshold defined in methodology). Because that, the maximum Q configuration acceptable by a VoltDB client (L_c) is 90.000 tps.

4.2.2 *Cluster Performance with K=0 and F=0.* To get $T_{0,0}$ it is necessary to run the workload several times, increasing the Q until the difference between Q total and the cluster performance is higher than 5%. The results of the VoltDB runs are presented on Table V. The column Q_{total} indicates the sum of Q of all clients on the environment, considering that the Q configuration of each client must be equal or less than L_c parameter determined previously.

The VoltDB results indicate that the cluster performance increase continuously until 238.740 tps. After that, the difference between Q_{total} and Cluster Performance is higher than 5%. This behavior

[8]http://www.nuodb.com/

Table V. *VoltDB runs to determine $T_{0,0}$ (No fault tolerance ($K = 0$), no failures ($F = 0$))*

Q_{total}	**VoltDB Performance (tps)**	**Variation**
180.000	180.052	0,0%
220.000	218.839	0,5%
240.000	237.564	1,0%
250.000	238.740	4,5%
255.000	237.378	6,9%
260.000	239.018	8,1%

Table VI. *Parameters defined for each DBMS*

DBMS	$T_{0,0}$
VoltDB	238.740 tps
NuoDB	112.692 tps
Dbms-X	49.116 tps

Table VII. *The performance degradation using fault tolerance mechanism.*

	VoltDB		**NuoDB**		**Dbms-X**	
Environment	$T_{K,0}$	$D_{K,0}$	$T_{K,0}$	$D_{K,0}$	$T_{K,0}$	$D_{K,0}$
K=0	238.740	-	112.692	-	49.116	-
K=1	156.659	34.38%	103.340	8.30%	70.663	-43.86%*
K=2	112.388	52.92%	103.669	8.00%	74.875	-52.44%*
K=3	101.899	57.32%	101.166	10.22%	80.506	-63.90%*

* The negative value means a performance increasing, no degradation.

of performance stabilization around 238.000 tps is due a backpressure situation. In all of the VoltDB runs the ϕ requirement has been met.

The $T_{0,0}$ of the others DDBMS is obtained in a similar way than VoltDB. The results are presented on Table VI.

4.3 Step 2

On Step 2 we evaluated the DDBMS with different K configurations. The results of $T_{K,0}$ and the performance degradation of the fault tolerance systems ($D_{K,0}$) are presented on Table VII.

The VoltDB results indicate that the backpressure state was achieved in the experiments. This means that the maximum performance has been reached.

The performance degradation on VoltDB happens because when we increase the replication level, we are dividing the available partitions among the duplicate copies. For example, on our environment when $K = 1$ there are 18 partitions to process transactions and the other 18 partitions store copies of the data. In the VoltDB architecture, the performance will be proportionally decreased as replication is increased.

While NuoDB and VoltDB present the same behavior with performance decreasing as long as K increases, Dbms-X behaves differently with its performance increasing. This happens because Dbms-X is a key-value store with eventual consistency to maintain the replicas (similar to a *Distributed Hash Table*). In this context, the read requests are routed to any node without looking up the most-up-to-date value.

4.4 Step 3

The performance of the DBMS evaluated with faults ($F > 0$) and the performance degradation when compared to a non-fault cluster are indicated on Table VIII.

The results on a faulty environment indicate that VoltDB performance degradation is less than 3% when K is configured as 1 or 2. But when $K = 3$, performance degradation increases considerably, varying between 8.9% and 25.4%. Similarly to VoltDB, Dbms-X also presented a performance

Table VIII. *DDBMS performance in a faulty environment - The degradation is based on non-fault cluster.*

	VoltDB		**NuoDB**		**Dbms-X**	
Environment	$T_{K,F}$	$D_{K,F}$	$T_{K,F}$	$D_{K,F}$	$T_{K,F}$	$D_{K,F}$
K=1 and F=1	152.072	2.928%	111.660	−8.05%*	40.472	42.72%
K=2 and F=1	110.491	1.687%	106.244	−2.48%*	57.227	23.56%
K=2 and F=2	109.781	2.319%	109.729	−5.84%*	26.760	64.26%
K=3 and F=1	92.8194	8.910%	103.469	−2.27%*	70.273	12.76%
K=3 and F=2	86.837	14.781%	103.922	−2.72%*	55.525	31.02%
K=3 and F=3	76.003	25.413%	104.304	−3.10%*	35.017	56.50%

* The negative value means a performance increasing, no degradation.

Table IX. *This summarizes the performance degradation results in a faulty environment.*

Metric	**VoltDB**	**NuoDB**	**Dbms-X**
DF_1	2.928	-8.05*	42.72
DF_2	1.897	-3.6*	37.13
DF_3	13.511	-2.475*	27.05

* The negative value means a performance increasing, no degradation.

Table X. *Overall metrics - This summarizes the partial metrics. D_T is the average of performance degradation metric (with fault tolerance), over the K index. D_F is the average of performance degradation metric (during failures), over the K index.*

Metric	**VoltDB**	**NuoDB**	**Dbms-X**
D_T	48.21	8.84	-53.4*
D_F	6.11	-4.70*	35.63

* The negative value means an increase on performance, no degradation.

degradation during step 3. However, it is much more evident when reaching 64% in the worst case.

Due to its particular implementation, NuoDB behaves differently, increasing performance during the fault injection. This is due to the peer-to-peer messaging infrastructure used to route tasks to nodes. Therefore, the fewer node replicas are running the fewer tasks and messages are routed. In addition, we observe that NuoDB has the stablest performance degradation upon faults, which may be also inherited from the resilience of the P2P backend.

Table IX summarizes the performance degradation results in a faulty environment. They corroborate our observations that VoltDB and Dbms-X suffer with faults, while NuoDB actually improves performance due to its P2P nature almost reaching the baseline results ($K = 0$, $F = 0$).

4.5 Final Comparison and Discussion

Based on the partial metrics presented above, it is possible to calculate the final metrics for each DDBMS and to compare the availability of them based on two different aspects. The final metrics are presented on Table X.

On one hand, Dbms-X did not get the best overall throughput. In the other hand, it had the best D_T result, but only in situations where fewer faults are expected, despite being a DHT-like system that processes transactions (but no ACID compliant). While VoltDB had the best overall performance throughput in a faultless scenario, the same was not observed while faults are injected (reflected by the D_F metric). In contrast, NuoDB had some performance impact to maintain the replicas, but presented a surprisingly D_F that is a direct result from its P2P core. Therefore, NuoDB presented the best results in environments with faults.

One may argue that one DDBMS is better than the other based on the presented results. While this may be true considering the execution scenarios to test replication and resilience, the DDBMS makes a different set of tradeoffs to improve availability, which may lead to situations where one of them will be more appropriate than the other. In this context, we claim that the UPB can be an important tool to help choosing the more appropriate DDBMS while heating the debate on availability solutions.

5. RELATED WORK

Over the last years, the DDBMS popularization raised the interests of the database community to design new benchmarks that satisfy new evaluation needs, including high-availability and high-throughput. In 2010, the YSCB was presented by Yahoo [Cooper et al. 2010] to benchmark four main features of distributed database stores: (1) Performance; (2) Scalability; (3) Availability; and (4) Replication. The YSCB includes a load generator to allow benchmarking different database stores, including relational and key-value ones. However, the load generator does not include fault-injection and assessing availability and replication are put off to future work.

The R-cubed benchmark was designed to assess availability of computational systems in general that was further extended to a benchmark suite called the System Recovery Benchmark (SRB) [Mauro et al. 2004]. The suite bundles five different benchmarks: (1) Cluster - SRB-X; (2) Application - SRB-D; (3) Databases - SRB-C; (4) Hard disk (RAID) - SRB-B; (5) Operating system - SRB-A. However, just two of them were implemented (i.e., SRB-X and SRB-A).

The DBench-OLTP [Vieira and Madeira 2003] benchmark was also designed to assess availability of transactional systems. It broadly uses the TPC-C specification with two main extensions in order to mimic real system problems: fault-injection and measures related to system dependability (e.g., mean time to detect errors). The UPB measures availability from another angle compared to the related work measuring the performance impact to maintain replicas upon faults.

An important recent work is the extension of TPC-E [Li and Levine 2012]. The authors focus on validation mechanisms to ensure the high-availability for DBMSs. The high-availability is supported by using two servers to run the database: 1- Main server; 2 - Standby server (i.e., backup server). The methodology is an extension of TPC-E [9] with metrics to evaluate the impact of failures. To validate the availability, the authors proposed two metrics: 1 - Impact of using a standby server; 2 - Recovery impacts. This benchmark aims to measure the impact of mirroring of centralized DBMS.

UPB *versus* Related Work
The main aspect that make the existing benchmarks not suitable for DDBMS is related to availability requirements. The related work presents two approaches. The distributed one misses any type of fault-injection mechanism preventing any attempt to benchmark availability. In fact, they point out benchmarking availability only as future work (e.g., YCSB). The second approach provides fault-injection, but for centralized environments. In contrast, we present a benchmark that aims to exercise the mechanisms developed to support high availability claimed by DDBMS.

Thus, our experiments are focused on the comparison of DDBMS performance. The comparison of our results with related benchmarks is an open issue, and particularly hard to do since they don't deal with availability aspects supported by our benchmark.

6. CONCLUSION

We presented the Under Pressure Benchmark (UPB) for evaluating DDBMS that support availability through replication. The UPB methodology increases the evaluation complexity from a stable system scenario up to a faulty system scenario, including (1) no fault tolerance, no failures; (2) fault tolerance,

[9]http://www.tpc.org/tpce/

no failures; (3) fault tolerance with failures. To the best of our knowledge, we are the first benchmark with such nature.

The UPB provides a focused benchmark to deal with a central issue related to DDBMS availability. We believe that the UPB fits the requirements for evaluating DDBMS upon critical situations, such as heavy loads and failures. Moreover, the UPB provides a good basis for database administrators to take decision about replication indexes, based on performance impact.

We validated our benchmark through experimentation of high-throughput in-memory DDBMS: VoltDB, NuoDB, and Dbms-X. We have verified that data replication has a large impact on performance, as a side-effect of availability. The impact could be considered negative or positive, depending on the DDBMS. This is more evident while the DDBMS is under high-throughput load, which is expected in Cloud-based applications.

There are different paths for future work. It is necessary to define metrics for other availability mechanisms, including data partitioning, database mirroring and recovery. However, in the latter, the database could be considered in a non-steady state because machine nodes need to be dropped along execution. In the context of high-throughput in-memory DDBMS, next steps include measuring the overhead of keeping snapshots and checkpoints in disk while the system is upon different workload scenarios, like those explored in this article. We also intend to improve the fault-injection evaluation by measuring the impact of different failures by simulating specific faults (e.g., disk/memory failure, network outage).

REFERENCES

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of ACM symposium on Cloud computing*. New York, NY, USA, pp. 143–154, 2010.

Gray, J. and Siewiorek, D. P. High-Availability Computer Systems. *IEEE Computer* 24 (9): 39–48, 1991.

Gray, J., Sundaresan, P., Englert, S., Baclawski, K., and Weinberger, P. J. Quickly Generating Billion-Record Synthetic Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data Conference*. Minneapolis, Minnesota, pp. 243–252, 1994.

Jain, R. *The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling.* Wiley, 1991.

Li, Y. and Levine, C. Extending TPC-E to Measure Availability in Database Systems. In *Proceedings of TPC Technology conference on Topics in Performance Evaluation, Measurement and Characterization*. Seattle, WA, USA, pp. 111–122, 2012.

Mauro, J., Zhu, J., and Pramanick, I. The System Recovery Benchmark. In *Proceedings of the IEEE Pacific Rim International Symposium on Dependable Computing*. Washington, DC, USA, pp. 271–280, 2004.

Meira, J. A., de Almeida, E. C., Traon, Y. L., and Sunyé, G. Peer-to-Peer Load Testing. In *The International Workshop on Load Testing of Large Software Systems*. Montreal, Canada, pp. 642–647, 2012.

Özsu, M. T. and Valduriez, P. *Principles of Distributed Database Systems, Third Edition.* Springer, 2011.

Vieira, M. and Madeira, H. A Dependability Benchmark for OLTP Application Environments. In *Proceedings of the International Conference on Very Large Data Bases*. Berlin, Germany, pp. 742–753, 2003.