Peer-to-peer Load Testing

Jorge Augusto Meira
UFPR, University of Luxembourg
Curitiba, Brazil - Luxembourg
Email: jmeira@inf.ufpr.br

Eduardo Cunha de Almeida
UFPR
Curitiba, Brazil
Email: eduardo@inf.ufpr.br

Yves Le Traon
University of Luxembourg
Luxembourg
yves.letraon@uni.lu

Gerson Sunye
INRIA - University of Nantes
Nantes, France
gerson.sunye@univ-nantes.fr

***Abstract*—Nowadays the large-scale systems are commonplace in any kind of applications. The popularity of the web created a new environment in which the applications need to be highly scalable due to the data tsunami generated by a huge load of requests (i.e., connections and business operations). In this context, the main question is to validate how far the web applications can deal with the load generated by the clients. Load testing is a technique to analyze the behavior of the system under test upon normal and heavy load conditions. In this work we present a peer-to-peer load testing approach to isolate bottleneck problems related to centralized testing drivers and to scale up the load. Our approach was tested in a DBMS as study case and presents satisfactory results.**

***Keywords*-peer-to-peer; large-scale systems; load testing;**

I. Introduction

Over the last decade, the web became omnipresent requiring to develop scalable systems due to the load of an increasing number of connected clients.

The fact is that the web is now portable and one can be connected all the time by using, for example, a smartphone or a tablet. Hence, the number of connected clients and the amount of data increased in orders of magnitude, which is called by the database community as "data tsunami" [1].

In this context, large-scale systems are designed to support the web load by gathering distributed components working along. A computer cluster is an example of a large-scale system. It gathers a set of machines (i.e., nodes) to form a single high performance computer, thus it is easy to scale up the system only by adding a new node. Failures can happen constantly due to the number of nodes running along combined with the load created by clients in terms of data and operations. For this reason, large-scale systems are designed to be fault-tolerant and scalable upon heavy load conditions.

As any kind of system, large-scale systems are tested to validate functionalities (e.g., data store and retrieve operations) and properties (e.g., fault-tolerance, scalability).

These systems, are commonly tested through a distributed test driver that coordinates the execution of test cases [2]. The distributed test driver typically consists of a test controller, or coordinator, that synchronizes the execution of test cases across distributed testers. Testers are responsible to invoke the inputs from a test case within the system under test (SUT) context and to compare the expected output with the observed output to assign a verdict.

However, the test drivers from the state of the art are not scalable to reproduce heavy load conditions, in particular two major load conditions, the cloud computing operations, such as the Amazon peak load during the Christmas season, and the internet-scale cyber attacks, such as the distributed denial of service (DDoS). In this paper, we leverage from a scalable peer-to-peer (P2P) test driver to generate these two major load conditions. First, the cloud computing load is important to expose defects that only appear upon peak load conditions. In [3], a number of defects exposed by this type of load is discussed in the context of GMail, Skype, Netflix, and Facebook. Second, a cyber attack load, like the DDoS produced by the hacker group "Anonymous" was responsible to take down several websites, such as the Bank of America, CIA, MasterCard, and VISA [1].

We based our test driver on two main hypotheses that are empirically questioned by reproducing a heavy load condition. First, large-scale systems can only be tested through a distributed test driver to avoid results polluted by test coordination bottlenecks (e.g. with current test drivers). Second, a P2P load testing approach enables to scale up a test load by several orders of magnitude and to coordinate it (distributing the test tasks and collecting results).

Our study case is an open source database management system (DBMS) that is commonly used as storage system in cloud computing solutions, such as Greenplum[2] and EnterpriseDB[3]. The results are discussed in the P2P load testing context.

The rest of paper is composed as follows. Next section present a overview about large-scale testing. Section III presents our testing methodology. Section IV describes the experiments. Section V concludes the paper.

II. Load Testing

Two main approaches can be used by load testing. The first approach is based on a point-to-point connection between the test driver and the SUT, where the driver submits

Work partially funded by Fundação Araucária.

[1] http://money.cnn.com/2012/01/20/technology/anonymous_hack/index.htm
[2] http://www.greenplum.com/
[3] http://www.enterprisedb.com/cloud-database

a certain load against the SUT interface. The objective is to validate the performance behavior of the SUT upon a given workload. Several tools, such as Hammerora[4], Oracle Application Testing Suit[5], and AppPerfect[6], provide a test driver to submit operations based on some type of load. However, they are limited to present performance results instead of the problems related to the load, which is typically presented by validation components used in software testing practices (ie., the test oracle). The testing tool Agenda [4] provides both a methodology and a test driver, however, it is tailored to database applications. Agenda presents a technique for checking database properties (e.g., ACID), and does not focus on load issues. In addition, all theses tools are based on a "single-party" test driver [5] that aims at testing the SUT in a point-to-point connection, with a single node. To reach large-scale load conditions it is required a "multi-party" test driver with distributed nodes (ie., tester components) communicating with the SUT interface.

Distributed test drivers are in general used along distributed testing, where a tester component is responsible to invoke the inputs from a test case within the SUT context and to compare the expected output with the observed output to assign a verdict [6], [7].

In this paper, we leverage the P2P [8] test driver approach, since a typical P2P application can easily reach from thousands to millions of users [9], as does Gnutella, Napster, Tapestry or Bittorrent. This scalability is possible, since there is no centralized component in a P2P system, where the peers are clients and servers at the same time. In the P2P approach, synchronization of test cases is done directly among the distributed testers without a centralized coordination. Figure 1 presents a UML deployment diagram that illustrates the deployment of PeerUnit. The diagram has two logical nodes, represented by rectangular cuboids. The coordinator and the tester run on different logical nodes and are connected by a communication path, which specifies that each tester is connected to exactly one coordinator and the coordinator can be connected to several testers. This approach only focuses on improving the performance of the test coordination by reducing the number of coordination messages and the time to exchange them [10], [8]. As such, it does not aim at generating different load conditions like those observed for the large-scale systems.

III. Load testing methodology

Large-scale testing is a hard task, if addressed without a rigorous and structured methodology. By analogy, it can be compared to integration testing, for which it is recognized that testing directly the whole system (big-bang strategy) is counter productive, while incrementally testing and integrating the system parts make it possible both to manage the system complexity and to understand and locate the detected faults. Where "divide and reign" methodology works for managing integration testing, there is no such strategy for testing large scale systems. Directly pushing a maximum load is likely to provoke failures, as a big-bang strategy usually does for integration, but it will not help understanding what are the causes of the failure, where does it come from. A test methodology is thus needed to cope with test scalability and to guide the testing campaign [11]: a test driver to handle such scale [12], [13], and a test load to allow exercising the SUT. In this paper, we present a P2P load testing approach based on two main hypotheses.

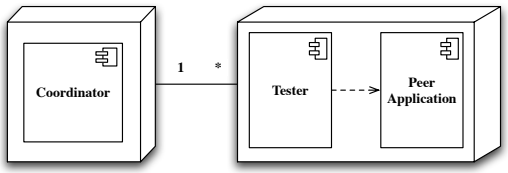

Figure 1. Peerunit Deployment

Hypothesis 1: For creating realistic load conditions, such as presented by [3], the test driver must scale up by distributing the test tasks and sending concurrent requests to the SUT.

Proof protocol 1: To compare the resource consumption of the SUT while testing through a centralized test coordination and a P2P test driver.

In fact, the empirical evidence will be obtained by showing the intrinsic limitations of a centralized test driver, compared to P2P scalability, that provides the mean to generate any peak load condition.

Hypothesis 2: The SUT does not use all of the allocated resources upon peak loads.

Proof protocol 2: To compare the inconsistencies between the declared SUT setup and the effective usage of the its resources.

This hypothesis is related to the defects exposed by the peak load generated by our P2P approach. We claim that these defects prevent the SUT from allocating all the available resources. Next, we discuss the P2P test architecture and the load generation.

A. P2P test architecture

The P2P test approach provides a solution for deploying the driver across a large number of machine nodes to generate peak load conditions. We take advantage of P2P inner scalability provided by the PeerUnit[7] testing framework.

The PeerUnit framework is based on 3 dimensions:

1) Functionality captured by the test sequence which enables a given behavior to be exercised;

[4]http://hammerora.sourceforge.net/

[5]http://www.oracle.com/technetwork/oem/app-test/index.html

[6]http://appperfect.com/products/load-testing/database-load-testing.html

[7]The PeerUnit Project: http://peerunit.gforge.inria.fr/

2) Scalability captured by the number of peers in the system;
3) Volatility captured by the number of peers which leave or join the system after its initialization during the test sequence.

In this paper, we are based on the scalability dimension to generate two major load conditions that are commonplace in large-scale systems. First, the cloud computing load that aims at reproduce peak load conditions experienced by major cloud computing players. In [3], a number of defects exposed by this type of load is discussed in the context of GMail, Skype, Netflix, and Facebook. Second, a cyber attack load that aims at reproduce DDoS attacks that take down major websites, such as those of the Bank of America, CIA, MasterCard, and VISA.

PeerUnit is implemented in Java and makes extensive use of dynamic reflection and annotations, using these features to select and execute the steps that compose a test case. Typically a test case are basically composed of a name, an objective and input/output data[14]. PeerUnit divides test cases into a sequence of test steps and coordinates the dispatch of the test case steps through the distributed testers. It ensures that a test step is completely finished by all testers before starting the next step. Then, testers are responsible to execute each test case steps against the SUT interface. The test case are implemented as Java classes and the test steps as Java methods. Table I illustrates the implementation of a load test.

Table I
TEST CASE

Step	Testers	Workload by tester
(a_1)	*	createConn400( );
(a_2)	*	createConn2000( );
(a_3)	*	createConn20000( );

The first test step (a_1) corresponds to the method *createConn400()*, in which a tester submits 400 operations to the SUT. For instance, if a_1 is performed by 10 testers that is equivalent to 4,000 operations. The following test steps increased the operations up to 200,000 (10*20,000) operations. The pseudocode can be observed in Listing 1

```
public class loadTest{
@TestStep(order=1, range = "*", timeout = 100000)
  public void createConn400( ){
   createThreads(400, connDBMS);
  }
@TestStep(order=2, range = "*", timeout = 100000)
  public void createConn2000( ){
   createThreads(2000, connDBMS);
  }
@TestStep(order=3, range = "*", timeout = 1000000)
  public void createConn20000( ){
   createThreads(20000, connDBMS);
  }
}
```

Listing 1. Test Case Class

The methods have 3 attributes: *order* that specifies the execution order of each step; *range* that identifies the testers responsible for the step ("*" means all possible testers will execute the test step); and *timeout* that is used to avoid deadlocks in case a tester did not finish the execution, in this case the step is aborted.

IV. EXPERIMENTS

In this section, we present the results of our experiments to validate the hypotheses and to generate peak load conditions for testing. The SUT used in the experiments was the PostgreSQL (version 8.3), a popular open source DBMS. The choice of PostgreSQL DBMS as our study case is due to two main reason: (i) the access to the source code helps us to understand the cause of the exposed defects; (ii) PostgreSQL is been used as the storage system of a number of cloud computing solutions where peak load conditions are expected. Some of these solutions include: Greenplum [15], EnterpriseDB Postgres Plus Cloud Database[8], the Heroku's Postgre-as-a-service solution[9], and the Cloud Foundry "vPostgres" service.

A. Database specification

To test a DBMS, we based our experiments on a popular database benchmark that provides database operations (i.e., transactions) and the database structure. The TPC-B benchmark is, actually, a load testing approach. It aims at testing the integrity of transactions upon significant disk processing (i.e., I/O operations)[10]. TPC-B provides four tables that implement a banking application (see Figure2) and reproduces the branches, tellers, accounts and accounts history.

The transactions reproduce deposits and withdraws performed by the customers (see Listning 2). Each transaction is composed by five actions: (i) update customer balance; (ii) select new balance; (iii) update teller balance; (iv) update branch balance; (v) commit deposit/withdraw in the history table.

Differently from TPC-B, which executes one transaction per DBMS client up to the commit state, we submit transactions concurrently up to the workload upper bound spread across the $|T|$ testers. The upper bound is defined by the test engineer with respect to the test objective. At each $t_i \in T$, the transactions are submitted using the "Weibull" distribution frequency. We use the "Weibull" distribution frequency, since it is commonly used by engineers in stress

[8]http://www.enterprisedb.com/cloud-database
[9]http://www.heroku.com/
[10]www.tpc.org/tpc-b

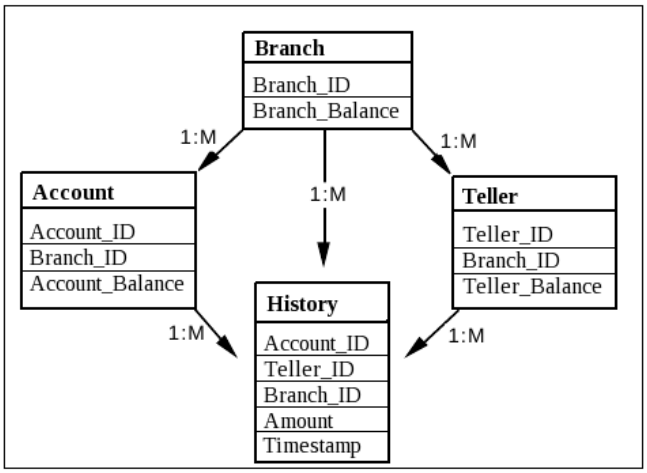

Figure 2. Database schema

testing [11] to search for a product failure rate [16], [17]. To guarantee the testers all submit requests at a specified time, we leverage the distributed PeerUnit coordination algorithm, that organize the testers in a tree-like manner where synchronization messages are exchanged with low coordination overhead [8].

```
BEGIN TRANSACTION
    Update Account where Account_ID = Aid:
        Read Account_Balance from Account
        Set Account_Balance = Account_Balance + Delta
        Write Account_Balance to Account
    Write to History:
        Aid, Tid, Bid, Delta, Time_stamp
    Update Teller where Teller_ID = Tid:
        Set Teller_Balance = Teller_Balance + Delta
        Write Teller_Balance to Teller
    Update Branch where Branch_ID = Bid:
        Set Branch_Balance = Branch_Balance + Delta
        Write Branch_Balance to Branch
COMMIT TRANSACTION
Return Account_Balance to driver;
```

Listing 2. Transaction pseudocode

B. Hardware Configuration

All of our experimentation is conducted on a cluster machine part of the Grid5000 platform[12]. We use 11 "Sun Fire X2200 M2" machine-nodes connected by Gigabit Ethernet, where each node is configured with 2 duo-core AMD Opteron 2218@2.613GHz and 8GB of main memory. We use one node to run exclusively the DBMS server and ten nodes to run clients, where each client is managed by a Peerunit tester. The clients and the server are executed on separate nodes, even for the smaller experimentations, to avoid performance interference. All nodes run with GNU/Linux Debian 5.0.9 (kernel 2.6). In all experiments reported in this paper, each tester is configured to run in its own Java Virtual Machine (JVM). The cost of test coordination is negligible and is not discussed in this paper (see [8] for a complete discussion on test coordination overhead).

[11]Stress testing is a type of load testing with the specific objective to degrade the performance of the SUT to find related defects.

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

C. DBMS Configuration

In all experiments only one setup parameter was modified, that is the maximum number of concurrent connections, `max_connections`, that by default is equal to 100. We set its value to 2,000, accordingly to the amount of available memory in the server node-machine [18]. This amount is controlled by the operating system parameter `SHMMAX`, which specifies the largest shared memory segment size. According to the PostgreSQL documentation, the following formula is recommended to calculate its value:

$$SHMMAX = (250kB + 8.2kB * shared\_buffers + 14.2kB * max\_conn)$$

D. Comparing the Test Driver Architectures

The first experiment highlights the need of a distributed testing architecture for simulating large-scale environments. The experiment compares the two test architectures, the first using the point-to-point test driver and the second using the P2P testing driver (distributing concurrent test requests). The point-to-point driver uses two machines, the first with the SUT and another simulating the clients. The second architecture is distributed and uses one machine to the SUT and five machines to simulate the clients.

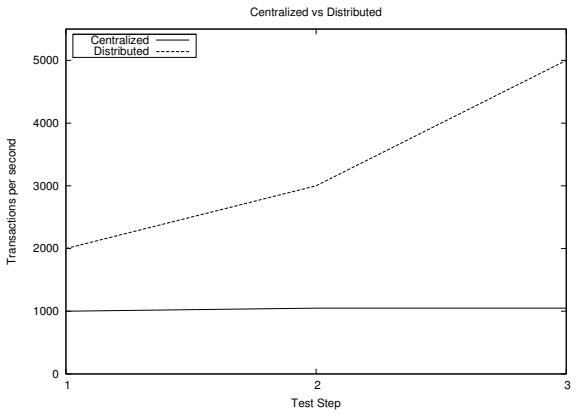

Figure 3. Hypothesis 1

Figure 3 shows the scalability problems of the centralized architecture where the number of created threads, that simulate the clients, is limited by the tester machine resources as

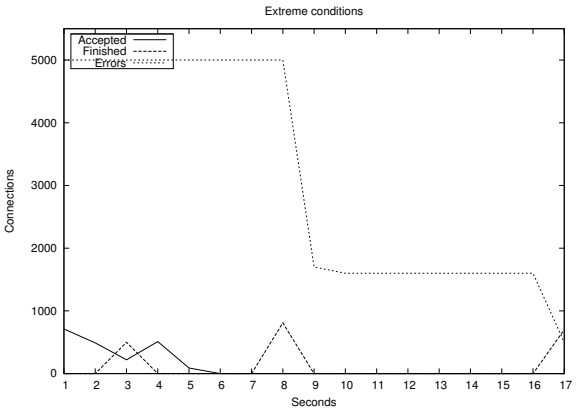

Figure 4. Load testing - Workload 100,000

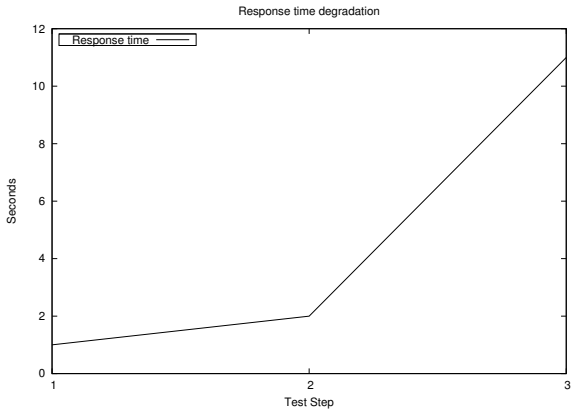

Figure 5. Response time degradation

long as we increase the load at each test step. It is possible to observe that the bottleneck is the tester machine, not the SUT. The number of concurrent transactions submitted by the point-to-point driver did not change due to the limitation of the tester machine, and the DBMS could accept all the submitted transactions as discussed in the next section.

E. Peak load conditions

This experiment is based on a distributed architecture and tries to reproduce a cloud computing load. This is an important step of the load testing to help analyzing the behavior of the DBMS upon peak conditions and also search for related defects. The expected behavior for this test, is the DBMS accepting 2,000 concurrent connections and finishing each transaction in the average time of the first step (we assume the first test step as the reference for the average response time). The exceeding connections shall be dropped by issuing a default error message[13].

Figure 4 shows the results where the DBMS presented an important number of error messages upon the peak conditions. From a functional test point of view, this is considered as a defect, since the observed connection value differs from the expected one.

In fact, we observed an increasing number of new backend DBMS processes created by each new transaction request. To create a backend process, besides getting a piece of shared-memory, the DBMS needs to manage the state of each process within an array. When a larger number of requests are taken by the DBMS due to the *Max_conn* setup, the number of backend processes fill out the array and eventually overtook its limit. The result: the new backend processes are dropped and the objective of $Max\_conn = 2,000$ is never reached (Hypothesis 2 proof).

The Figure 5 shows response time degradation, that in the worst case reach 11 seconds. It is related to peak conditions of a cloud computing environment or even a cyber DDoS attack.

[13]In PostgreSQL, when the number of requests exceeds the configured value, it issues the following message: *Sorry too many clients already*.

V. Conclusion

Large-scale systems are submitted to environments with huge variety of workloads. Load testing is becoming an insecapable technique to determine whether a system may face a huge amount of requests. Deploying a system that is supposed either to resist to load peaks or to DDoS should be done only after a testing stage targeting this aspect.

In this paper we presented a load distributed testing methodology based on P2P testing framework that shows the importance of the distributed testing methodology to reach the real problems in distributed systems. The experiments shown us the problems related with the centralized approaches where the bottleneck is not the SUT, but the centralized driver.

Our approach also shown the advantages of using P2P testing architecture to test large-scale systems. The scalability provided by this architecture is very important to scale up the workload and reach the performance limits of the SUT.

In future work, we plan to spread the load across a larger setup of node machines to scale up the test in orders of magnitude. This type of experiment will allow to reproduce the load of DDoS attacks upon any type of SUT. Furthermore, we plan to develop a proper stress testing methodology for cloud-computing systems to isolate the biases that may affect the test results (e.g., operating system, network) and to consider intrinsic aspects of large-scale systems (e.g., scalability, resilience, and heterogeneity of nodes).

References

[1] M. Stonebracker. (2012, Jan.) Voltdb. [Online]. Available: http://voltdb.com/

[2] T. Walter, I. Schieferdecker, and J. Grabowski, "Test architectures for distributed systems: State of the art and beyond," in *IWTCS*, 1998, pp. 149–174.

[3] H. S. Gunawi, T. Do, J. M. Hellerstein, I. Stoica, D. Borthakur, and J. Robbins, "Failure as a service (faas): A cloud service for large-scale, online failure drills," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-87, Jul 2011. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-87.html

[4] Y. Deng, P. G. Frankl, and D. Chays, "Testing database transactions with agenda," in *ICSE*, G.-C. Roman, W. G. Griswold, and B. Nuseibeh, Eds. ACM, 2005, pp. 78–87.

[5] ISO9646, "Open systems interconnection conformance testing methodology and framework," 1991.

[6] B. Long and P. A. Strooper, "A case study in testing distributed systems," in *DOA*, 2001, pp. 20–.

[7] A. Petrenko and A. Ulrich, "Verification and testing of concurrent systems with action races," in *TestCom*, ser. IFIP Conference Proceedings, H. Ural, R. L. Probert, and G. von Bochmann, Eds., vol. 176. Kluwer, 2000, pp. 261–280.

[8] E. C. de Almeida, J. E. Marynowski, G. Sunyé, Y. Le Traon, and P. Valduriez, "Efficient distributed test architectures for large-scale systems," in *ICTSS 2010: 22nd IFIP Int. Conf. on Testing Software and Systems*, Natal, Brazil, November 2010.

[9] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 335–371, 2004.

[10] E. C. de Almeida, G. Sunyé, and P. Valduriez, "Testing architectures for large scale systems," in *VECPAR*, 2008, pp. 555–566.

[11] E. C. de Almeida, G. Sunyé, Y. L. Traon, and P. Valduriez, "A framework for testing peer-to-peer systems," in *19th ISSRE, 11-14 November, Redmond, Seattle,USA*. IEEE Computer Society, 2008.

[12] E. C. de Almeida, G. Sunye, Y. L. Traon, and P. Valduriez, "Testing peer-to-peer systems," *Empirical Software Engineering*, vol. 15, no. 4, pp. 346–379, 2010.

[13] H. S. Gunawi, T. Do, J. M. Hellerstein, I. Stoica, D. Borthakur, and J. Robbins, "Failure as a service (faas): A cloud service for large-scale, online failure drills," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-87, Jul 2011. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-87.html

[14] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating test data from state-based specifications," *The Journal of Software Testing, Verification and Reliability*, vol. 13, pp. 25–53, 2003.

[15] F. M. Waas, "Beyond conventional data warehousing - massively parallel data processing with greenplum database," in *BIRTE (Informal Proceedings)*, 2008.

[16] A. Alhadeed and S.-S. Yang, "Optimal simple step-stress plan for khamis-higgins model," *IEEE Transactions on Reliability*, vol. 51, pp. 212 – 215, June 2002.

[17] C. Xiong, "Inferences on a simple step-stress model with type-ii censored exponential data," *IEEE Transactions on Reliability*, vol. 47, pp. 142 – 146, June 1998.

[18] P. G. D. Group. (2012, Jan.) Postgresql. [Online]. Available: http://www.postgresql.org/about/