Evaluating Data Freshness in Large Scale Replicated Databases⋆

Miguel Araújo and José Pereira

Universidade do Minho

Abstract. There is nowadays an increasing need for database replication, as the construction of high performance, highly available, and large-scale applications depends on it to maintain data synchronized across multiple servers. A particularly popular approach, used for instance by Facebook, is the MySQL open source database management system and its built-in asynchronous replication mechanism. The limitations imposed by MySQL on replication topologies mean that data has to go through a number of hops or each server has to handle a large number of slaves. This is particularly worrisome when updates are accepted by multiple replicas and in large systems.

It is however difficult to accurately evaluate the impact of replication in data freshness, since one has to compare observations at multiple servers while running a realistic workload and without disturbing the system under test. In this paper we address this problem by introducing a tool that can accurately measure replication delays for any workload and then apply it to the industry standard TPC-C benchmark. This allows us to draw interesting conclusions about the scalability properties of MySQL replication.

Keywords: Databases, Replication, MySQL, Data Freshness

1 Introduction

With the rapid growth of the Internet, availability has recently became critical due to large amounts of data being captured and used each day with the emerging online services. Large companies such as Google, eBay, or Amazon handle exabytes of data per year. Facebook claims to be one of the largest MySQL installations running thousands of servers handling millions of queries, complemented by its own Cassandra data store for some very specific queries. These Internet-based services have become a standard in our information society, supporting a wide range of economic, social, and public activities. And in this globalized era, since large organizations are present in different places all over the world, information must be always online and available. The loss of information or its unavailability can lead to serious economic damages. So, high-availability, performance, and reliability are all critical requirements in such systems.

Both of these challenges are commonly addressed by means of the same technique, namely data replication. Application components must be spread over a wide area network, providing solutions that enable high availability through network shared contents. Since databases are more and more deployed on clusters and over wide area networks, replication is a key component. Replicating data improves fault-tolerance since the failure of a site does not make a data item inaccessible. Available sites can take over the work of failed ones. And also improves performance since data access can be localized over the database network, i.e. transaction load is distributed across the replicas, achieving load balancing, and in the other hand it can be used to provide more computational resources, or allow data to be read from closer sites reducing the response time and increasing the throughput of the system.

There are however several different replication protocols, differing first and foremost whether propagation takes place within transaction boundaries [3]: Lazy schemes use separate transactions for execution and propagation, in contrast to eager schemes that distribute updates to replicas in the context of the original updating transaction. Thus, the eager method makes it easy to guarantee transaction properties, such as serializability but, since such transactions are distributed and relatively long-lived, the approach does not scale well [2]. On the other hand, lazy replication reduces response times as transactions can be executed and committed locally and only then propagated to other sites [4]. In detail, being replicated asynchronously, data is first written on the master server and then is propagated to slaves, and so, specially in the case of hundreds of servers, slaves will take some time to obtain the most recent data. Lazy propagation thus opens up the possibility of having stale data in replicas and makes data freshness a key issue for correctness and performance.

Most database management systems implement asynchronous master-slave replication. The systems provide mechanisms for master-slave replication that allows configuring one or more servers as slaves of another server, or even to behave as master for local updates. MySQL in particular allows almost any configuration of master and slaves, as long as each server has at most one master. As described in Section 2, this usually leads to a variety of hierarchical replication topologies, but includes also a ring which allows updates to be performed at any replica, as long as conflicts are avoided.

It is thus interesting to assess the impact of replication topology in MySQL, towards maximizing scalability and data freshness. This is not however easy to accomplish. First, it requires comparing samples obtained at different replicas and thus on different time referentials, or, when using a centralized probe, network round-trip has to be accounted for. Second, the number of samples that can be obtained has to be small in order not to introduce a probing overhead. Finally, the evaluation should be performed while the system is running a realistic workload, which makes it harder to assess the point-in-time at each replica with a simple operation. In this paper we address these challenges by making the following contributions:

– We describe a tool that obtains a small number of samples of log sizes using a centralized probe at different points in time. It then selects particularly interesting periods of time and computes a freshness value with the distance between lines fitted to these points.
– We apply the tool to two representative MySQL configurations with a varying number of replicas and increasingly large workloads using the industry standard TPC-C on-line transaction processing benchmark [1]. This allows us to derive conclusions on the scalability of MySQL replication.

The rest of the paper is structured as follows: Section 2 describes the MySQL replication architecture in detail. In Section 3, the method and tool used to achieve this goal is presented. Section 4 presents the results obtained and Section 5 concludes the paper.

2   Background

MySQL, as most database management systems do, implements asynchronous master-slave replication. It allows configuring each server as slave of any other server while simultaneously behaving as master for local updates. The configuration of replication allows an arrangement of masters and slaves in different topologies and it is possible to replicate the entire server, replicate only certain databases or to choose what tables to replicate.

2.1   Replication Mechanism

The replication mechanism of MySQL, works at a high level in a simple three-part process:

1. The master records changes to its data in its binary log (these records are called binary log events).
2. The slave copies the master's binary log events to its own log (relay log).
3. The slave replays the events in the relay log, applying the changes to its own data.

Briefly, after writing the events to the binary log, the master tells the storage engine to commit the transactions. The next step is for the slave to start an I/O thread to start the dump. This process reads events from the master's binary log. If there are events on the master, the thread writes them on the relay log. Finally, a thread in the slave called SQL thread reads and replay events from the relay log, thus updates slave's data to match the master's data. To notice that the relay log usually stays in the operating system's cache, having very low overhead.

This replication architecture decouples the processes of fetching and replaying events on the slave, which allows them to be asynchronous. That is, the I/O thread can work independently of the SQL thread. It also places constraints on the replication process, the most important of which is that replication is
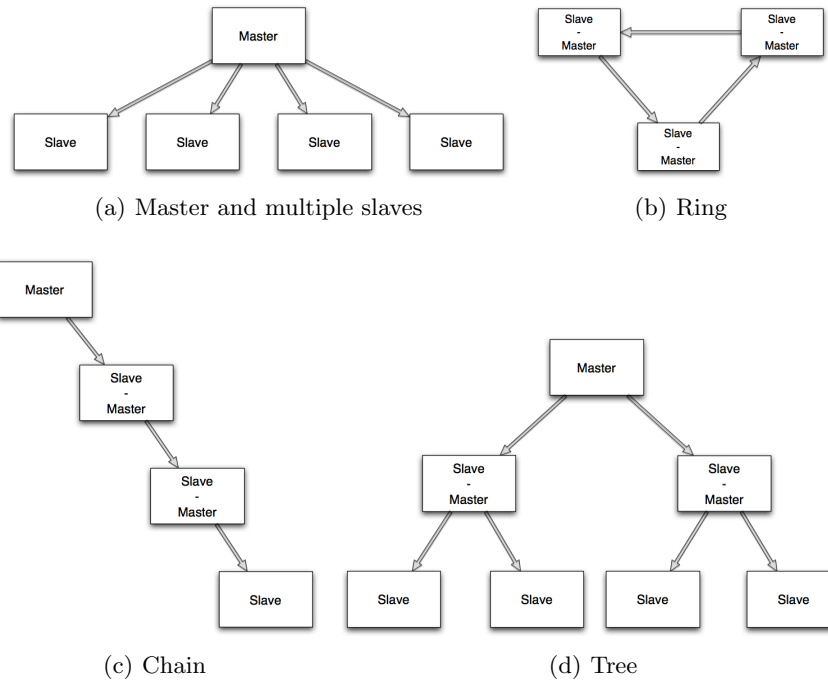
Fig. 1. Sample MySQL replication topologies.

serialized on the slave. This means updates that might have run in parallel (in different threads) on the master cannot be parallelized on the slave, which is a performance bottleneck for many workloads.

2.2 Replication Topologies

The simplest topology besides **Master-Slave** is **Master and Multiple Slaves** (Figure 1(a)). In this topology, slaves do not interact with each other at all, they all connect only to the master. This is a configuration useful for a system that has few writes and many reads. However, this configuration is scalable to the limit that the slaves put too much load on the master or network bandwidth from the master to the slaves becoming a problem.

Other possible configuration is **Master-Master in Active-Active Mode**. This topology involves two servers, each configured as both a master and slave of the other. The main bottleneck in this configuration resides on how to handle conflicting changes.

A variation on master-master replication that avoids the problems of the previous is the **Master-Master in Active-Passive** mode replication. The main difference is that one of the servers is a read-only "passive" server. This configuration permits swapping the active and passive server roles back and forth very

easily, because the servers configurations are symmetrical. This makes failover and failback easy.

The related topology of the previous ones is **Master-Master with Slaves**. The advantage of this configuration is extra redundancy. In a geographically distributed replication topology, it removes the single point of failure at each site.

One of the most common configuration in database replication, is the **Ring** topology (Figure 1(b)). A ring has three or more masters. Each server is a slave of the server before it in the ring, and a master of the server after it. This topology is also called circular replication. Rings do not have some of the key benefits of a master-master setup, such as symmetrical configuration and easy failover. They also depend completely on every node in the ring being available, which greatly increases the probability of the entire system failing. And if you remove one of the nodes from the ring, any replication events that originated at that node can go into an infinite loop. They will cycle forever through the topology, because the only server that will filter out an event based on its server ID is the server that created it. In general, rings are brittle and best avoided. Some of the risks of ring replication can be decreased by adding slaves to provide redundancy at each site. This merely protects against the risk of a server failing, though.

Another possibility, regarding some certain situations where having many machines replicating from a single server requires too much work for the master, or the replication is to spread across a large geographic area that chaining the closest ones together gives better replication speed, is the **Daisy Chain** (Figure 1(c)). In this configuration each server is set to be a slave server to one machine as as master to another in a chain. Again, like the ring topology the risk of losing a server can the decreased by adding slaves to provide redundancy at each site.

The other most common configuration is the **Tree or Pyramid** topology (Figure 1(d)). This is very useful in the case of replicating a master to a very large number of slaves. The advantage of this design is that it eases the load on the master, just as the distribution master did in the previous section. The disadvantage is that any failure in an intermediate level will affect multiple servers, which would not happen if the slaves were each attached to the master directly. Also, the more intermediate levels you have, the harder and more complicated it is to handle failures.

2.3 Data Freshness

Data replication must ensure ACID properties and copy consistency must be preserved through global isolation [6]. To ensure global isolation a transaction that modifies data must update all its copies before any other transaction can access the data. Property known as 1-copy serializability. This property can be ensured with synchronous replication, in which a transaction updates all replicas, enforcing the mutual consistency of all replicas. However, this replication model increases the transaction latency because extra messages are added to the transaction (distributed commit protocol).

On the other hand, lazy replication updates all the copies in separate transactions, so the latency is reduced in comparison with eager replication. A replica is updated only by one transaction and the remain replicas are updated later on by separate refresh transactions [7].

Although there are concurrency control techniques and consistency criterion which guarantee serializability in lazy replication systems, these techniques do not provide data freshness guarantees. Since transactions may see stale data, they may be serialized in an order different from the one in which they were submitted.

So, asynchronous replication leads to periods of time that copies of the same data diverge. Some of them have already the latest data introduced by the last transaction, and others have not. This divergence leads to the notion of data freshness: The lower the divergence of a copy in comparison with the other copies already updated, the fresher is the copy [5].

MySQL replication is commonly known as being very fast, as it depends strictly on the the speed that the engine copies and replays events, the network, the seize of the binary log, and time between logging and execution of a query [8]. However, there have not been many systematic efforts to precisely characterize the impact on data freshness.

One approach is based on the use of a User Defined Function returning the system time with microsecond precision [8]. Inserting this function's return value on the tables we want to measure and comparing it to the value on the respective slave's table we can obtain the time delay between them. But this measurements can only be achieved on MySQL instances running on the same server due to clock inaccuracies between different machines.

A more practical approach uses a Perl script and the Time::HiRes module to get the system time with seconds and microseconds precision.[1] The first step is to insert that time in a table on the master, including the time for the insertion. After this, the slave is queried to get the same record and immediately after the attainment of it the subtraction between system's date and time got from the slave's table is made, obtaining the replication time. As with the method described above this one lacks of accuracy due to the same clock inaccuracies.

3   Measuring Propagation Delay

3.1   Approach

Our approach is based on using a centralized probe to periodically query each of the replicas, thus discovering what has been the last update applied. By comparing such positions, it should be possible to discover the propagation delay. There are however several challenges that have to be tackled to obtain correct results, as follows.
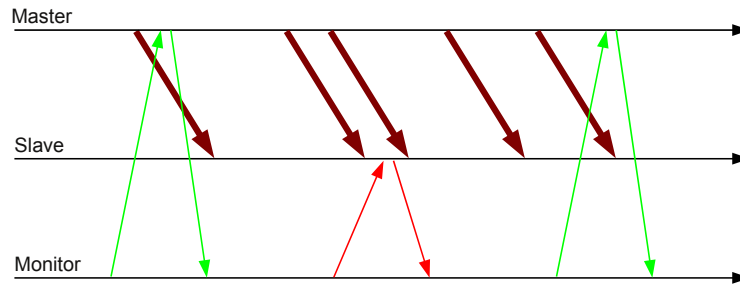
---

[1] http://datacharmer.blogspot.com/2006/04/measuring-replication-speed.html

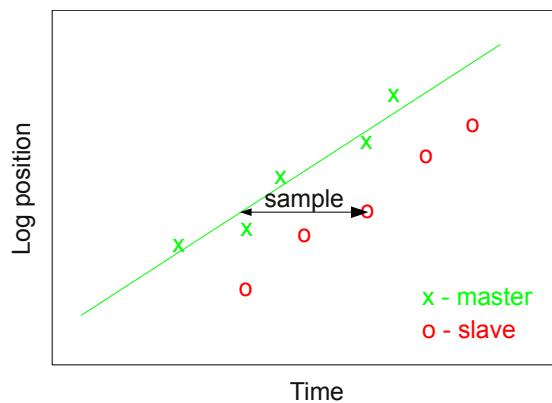**Fig. 2.** Impossibility to probe simultaneously master and slaves.

Fig. 3. Log position over the time

Measuring updates. The first challenge is to determine by how much two replicas differ and thus when two replicas have applied exactly the same amount of updates. Instead of trying to compare database content, which would introduce a large overhead, or using a simple database schema and workload that makes it easy, we use the size of the transactional log itself. Although this does not allow us to measure logical divergence, we can determine when two replicas are exactly with the same state.

Non-simultaneous probing. The second challenge is that, by using a single centralized probe one cannot be certain that several replicas are probed at exactly the same time. Actually, as shown in (Figure 2), if the same monitor periodically monitors several replicas it is unlikely that this happens at all. This makes it impossible to compare different samples directly.

Instead, as shown in (Figure 3) we consider time–log position pairs obtained by the monitor and fit a line to them (using the least-squares method). We can then compute the distance of each point obtained from other replicas to this line along the time axis. This measures how much time such replica was stale.
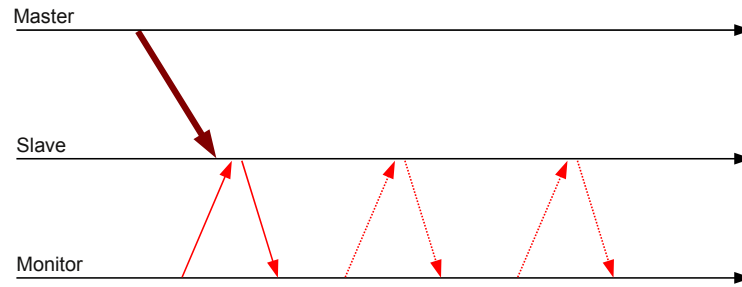
Fig. 4. Sampling twice without updates erroneously biases the estimate.

Eliminating quiet periods. Moreover, as replication traffic tends to be bursty. If one uses repeated samples of a replica that stands still at the same log position, the estimate is progressively biased towards a (falsely) higher propagation delay, as shown in (Figure 4). This was solved by selecting periods where line segments obtained from both replicas have a positive slope, indicating activity.

Dealing with variability. Finally, one has to deal with variability of replication itself and over the network used for probing. This is done by considering a sufficient amount of samples and assuming that each probe happens after half of the observed round-trip. Moreover, a small percentage of the highest round-trips observed is discarded, to remove outliers.

3.2 Implementation

An application to interrogate the master instance and several replicas of the distributed database scheme was developed. This tool stores the results in a file for each instance. To obtain the log position it uses the MySQL API in order to obtain the replication log position. The temporal series of observed log positions are then stored in separate files, one for each node of the distributed database.

Results are then evaluated off-line using the Python programming language and R statistics package. This script filters data as described and then adjusts a line to the values of the log files and compares them. This includes looking for periods of heavy activity and fitting line segments to those periods. With these line segments, the script compares each slave points with the corresponding segment on the master, if the segment does not exist for the selected point, the point is ignored. In the end, average is calculated based on the difference of values between slave points and corresponding segments on the master. A confidence interval can also be computed, using the variance computed from the same data.

4 Experiments

4.1 Workload

In order to assess the distributed database used in the case study, we have chosen the workload model defined by TPC-C benchmark [1], a standard on-line transaction processing (OLTP) benchmark which mimics a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. Specifically, we used the Open-Source Development Labs Database Test Suit 2 (DBT-2), a fair usage implementation of the specification.

Although TPC-C includes a small amount of read-only transactions, it is composed mostly by update intensive transactions. This choice makes the master server be almost entirely dedicated to update transactions even in a small scale experimental setting, mimicking what would happen in a very large scale MySQL setup in which all conflicting updates have to be directed at the master while read-only queries can be load-balanced across all remaining replicas.

Each client is attached to a database server and produces a stream of transaction requests. When a client issues a request it blocks until the server replies, thus modeling a single threaded client process. After receiving a reply, the client is then paused for some amount of time (think-time) before issuing the next transaction request. The TPC-C model scales the database according to the number of clients. An additional warehouse should be configured for each additional ten clients. The initial sizes of tables are also dependent on the number of configured clients.

During a simulation run, clients log the time at which a transaction is submitted, the time at which it terminates, the outcome (either abort or commit) and a transaction identifier. The latency, throughput and abort rate of the server can then be computed for one or multiple users, and for all or just a subclass of the transactions. The results of each DBT-2 run include also CPU utilization, I/O activity, and memory utilization.

4.2 Setting

Two replication schemes were installed and configured. A five machines topology of master and multiple slaves, and a five machine topology in daisy chain.

The hardware used included six HP Intel(R) Core(TM)2 CPU 6400 - 2.13GHz processor machines, each one with one GByte of RAM and SATA disk drive. The operating system used is Linux, kernel 2.6.31-14, from Ubuntu Server, and the database engine used is MySQL 5.1.39. All machines are connected through a LAN, and are named PD01 to PD06. Being PD01 the master instance, PD04 the remote machine in which the interrogation client executes, and the others the slave instances.

The following benchmarks were done using the workload TPC-C with the scale factor (warehouses) of two, number of database connections (clients) one hundred and the duration of twenty minutes.

Replica	PD02	PD03	PD05	PD06
Number of samples	15238	15121	15227	15050
Average delay (μs)	10133	10505	10249	10260
99% confidence interval ($\pm$)	363	373	412	378

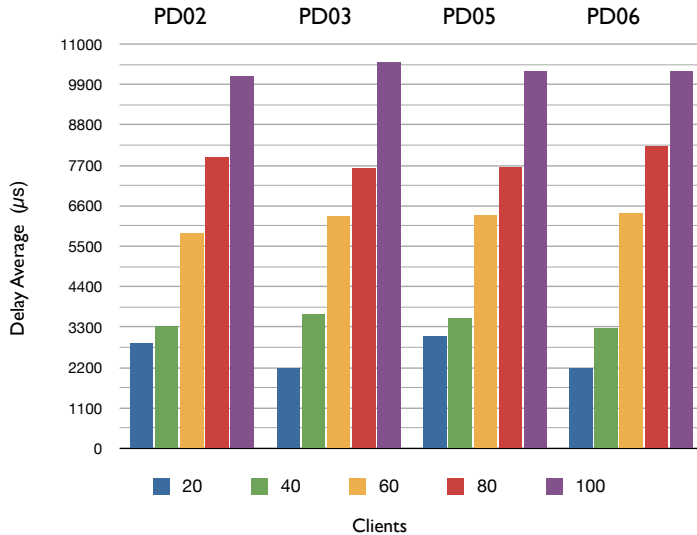**Table 1.** Results for master and multiple slaves topology with 100 clients.

Fig. 5. Scalability of master and multiple slaves topology.

4.3 Results

Results obtained with 100 TPC-C clients and the master and multiple slaves topology are presented in (Table 1). It can be observed that all replicas get similar results and that the propagation delay is consistently measured close to 10 ms with a small variability. This represents an upper bound on the worst case scenario staleness that a client can observe by reading from the master and any other replica if the replication connection is operational.

Results with an different numbers of TPC-C clients can be found in (Figure 5). They show that propagation delay grows substantially with the load imposed on the master. At the same time, as idle periods get less and less frequent due to the higher amount of information to transfer, the probability of a client being able to read stale data grows accordingly.

Results obtained with 100 TPC-C clients and the chain topology are presented in (Table 2). In contrast to master and multiple slaves, the delay now grows as the replica is farther away for the master. This configuration also gives an indication of how the ring topology would perform: As any replica would

Replica	PD02	PD03	PD05	PD06
Number of samples	12423	12819	12937	14004
Average delay (μs)	12353	19767	25698	30688
99% confidence interval ($\pm$)	557	700	864	984

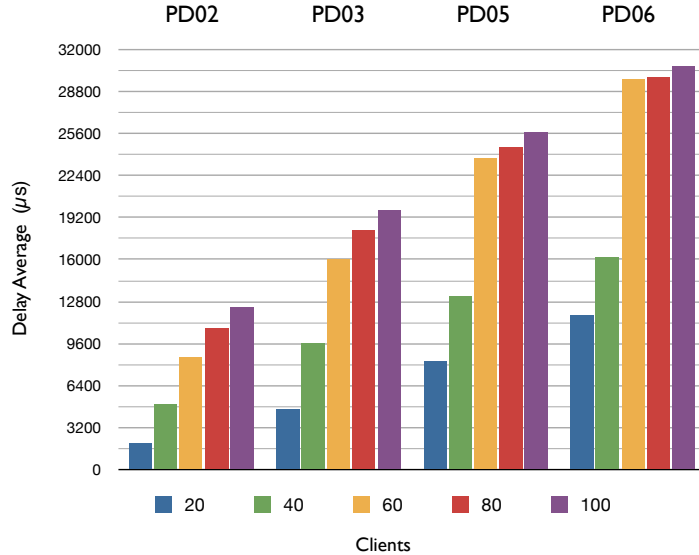**Table 2.** Results for chain topology with 100 clients.

Fig. 6. Scalability of the chain topology.

be, on average, half way to other masters, one should expect the same delay as observed here on replicas PD03 and PD05.

Results with an increasing number of TPC-C clients can also be found in (Figure 6), showing that propagation delay still grow substantially with the load imposed on the master. This means that using the ring configuration for write scalability with suffer the same problem, thus limiting its usefulness.

5   Conclusions

Asynchronous, or lazy, database replication is often the preferred approach for achieving large scale and highly available database management systems. In particular, the replication mechanism in MySQL is at the core of some of the largest databases in use today for Internet applications. In this paper we set out to evaluate the consequences on data freshness of the choice of replication topologies and of a growing workload.

In short, our approach measures freshness in terms of time required for updates performed at the master replica to reach each slave while using a realistic update-intensive workload, as the proposed tool can infer freshness from a small number of samples taken at different points in time at different replicas. Experimental results obtained with this tool show that, in both tested replication topologies, the delay grows with the workload which limits the amount of updates that can be handled by a single replica. Moreover, we can also conclude that in circular replication the delay grows as the number of replicas increases, which means that spreading updates across several replicas does not improve update scalability. Finally, the delay grows also with the number of slaves attached to each master, which means that read scalability can also be achieved only at the expense of data freshness.

The conclusion is that the apparently unlimited scalability of MySQL using a combination of different replication topologies can only be achieved at the expense of an increasing impact in data freshness. The application has thus to explicitly deal with stale data in order to minimize or prevent the user from observing inconsistent results.

References

1. Transaction Processing Performance Council. TPC Benchmark^{TM} C standard specification revision 5.11, February 2010.
2. K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. *Proceedings of the 20th International Conference on Data Engineering (ICDE 2004)*, 30:424–435, 2004.
3. J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, page 173, 1996.
4. B. Kemme. *Database replication for clusters of workstations*. PhD thesis, Technische Wissenschaften ETH Zürich, Zürich, 2000.
5. C. Le Pape, S. Gancarski, and P. Valduriez. Data quality management in a database cluster with lazy replication. *Journal of Digital Information Management (JDIM)*, 3(2), 2005.
6. M. Özsu and P. Valduriez. Distributed and parallel database systems. *ACM Computing Surveys (CSUR)*, 28(1):125–128, 1996.
7. E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. Research Report RR-3654, INRIA, 1999.
8. B. Schwartz, J. D. Zawodny, D. J. Balling, V. Tkachenko, and P. Zaitsev. *High Performance MySQL: Optimization, Backups, Replication, and More; 2nd ed.* O'Reilly, 2008.