Conflict Classes for Replicated Databases: A Case-Study

Ana Nunes
INESC TEC & U. Minho
Email: ananunes@di.uminho.pt

Rui Oliveira
INESC TEC & U. Minho
Email: rco@di.uminho.pt

José Pereira
INESC TEC & U. Minho
Email: jop@di.uminho.pt

***Abstract*—The major challenge in fault-tolerant replicated transactional databases is providing efficient distributed concurrency control that allows non-conflicting transactions to execute concurrently. A common approach is to partition the data according to the data access patterns of the workload, assuming that this will allow operations in each partition to be scheduled independently and run in parallel.**

The effectiveness of this approach hinges on the characteristics of the workload: (i) the ability to identify such partitions and (ii) the actual number of such partitions that arises. Performance results that have been presented to support such proposals are thus tightly linked to the simplistic synthetic benchmarks that have been used. This is worrisome, since these benchmarks have not been conceived for this purpose and the resulting definition of partitions might not be representative of real applications. In this paper we contrast a more complex synthetic benchmark (TPC-E) with a real application in the same area (financial brokerage), concluding that the real setting makes it much harder to determine a correct partition of the data and that sub-optimal partitioning severely constrains the performance of replication.

I. Introduction

Database replication has been a hot research topic for some time now, from single-tier architectures to multi-tier and cloud architectures. Currently, distributed transactions are a hot topic with an expanding audience, fostered by STM and cloud databases. Previous work on replication is thus being reused in new settings, widening its significance. The focus has been on how to enable highly available applications/services through fault-tolerant and scalable architectures [1], [2], [3], [4], [5], [6]. A key concern in the design of fault-tolerant database replication protocols is ensuring that sufficient transactions can be scheduled to execute concurrently such that the system performs adequately [7].

A common approach for designing concurrency control in database replication protocols relies on defining conflict classes. In short, the available data is partitioned according to some criteria, and a FIFO transaction queue is associated to each partition [1], [4], [5]. Disjoint data partitions constitute *basic* conflict classes. Compound conflict classes can be defined by grouping basic conflict classes.

Each transaction has an associated set of basic conflict classes according to the data partitions it accesses. Transactions that access disjoint sets of basic conflict classes are guaranteed not to conflict and thus can be concurrently executed. Conflicts may arise among transactions that access a common conflict class. In order to ensure correctness, transactions that conflict must be serialized.

Conservative concurrency control based on conflict classes requires transaction scheduling to adhere strictly to the order defined by the conflict class queues: transactions that access a common conflict class will always be serialized. This is a fundamental limitation of concurrency control mechanism.

In order to use replication protocols with conservative concurrency control efficiently, the data must be partitionable according to the particular data access patterns of the applied workload, which is a rather strong assumption. Moreover, even if possible, the concrete conflict class definition chosen influences the contention and maximum parallelism attainable. Therefore, the performance of conservative protocols hinges on a favorable definition of conflict classes, as the number of disjoint conflict classes defines the maximum number of transactions that can be executed concurrently.

Most of these protocols have only been tested in custom-tailored scenarios with very simple and unrealistic database schemas [2], [3], [5], [6]. Some have also been tested using benchmarks such as TPC-C [8] and TPC-W [9], for which straightforward table-based partitioning schemes can be easily derived [1]. The question remains whether the assumptions made regarding conflict class definition are still plausible when dealing with more complex benchmarks, for which partitioning is not straightforward at all or, more importantly, regarding real-world applications.

We answer this question by analysing the TPC-E [10] benchmark and a real-world application in the same domain focusing on partitioning and the suitability of database replication protocols with conservative concurrency control for these scenarios.

The rest of this paper is structured as follows. Section II provides some background on database replication protocols, with an emphasis on concurrency control based on conflict classes. Section III then introduces the synthetic TPC-E benchmark, described how conflict classes could easily be defined. Section IV-A then explains how conflict classes can be defined for a real brokerage application, showing how the result impacts concurrency in Section IV-B. Section V concludes the paper.

II. Background

The concept of isolation refers to which interactions are allowed among transactions executing in the system. A system that meets the *serializability* criterion guarantees that regardless of the actual interleaving of operations from different transactions during execution, the result is equivalent to a serial execution of the transactions.

The overwhelming majority of RDBMSs uses a different criterion, *snapshot isolation*, which differs from serializability by considering only write/write conflicts [11].

Concurrency control is a key issue in database replication. The idea is to guarantee that the concurrent execution of transactions does not adversely affect the correctness of the database system.

For example, in passive primary-copy scenarios, concurrency control occurs in three phases: first, in the primary, the database engine's native concurrency control decides on which transactions commit or abort and in which order; second, the primary forwards write operations or write sets to replicas; lastly, replicas apply writes. In order to prevent replica divergence, replicas must be guaranteed to decide on the same serialization order as the primary. Using a group communication that offers a FIFO multicast primitive provides that guarantee. In a multi-primary setting, having several primaries means that these must agree on a total order for transaction execution. If replicas apply updates according to that total order, strong consistency is guaranteed. Total order message delivery can be provided by group communication protocols.

Concurrency control comes in two flavours: conservative and optimistic. A conservative concurrency control prevents potentially conflicting transactions from executing concurrently. A common strategy for assessing potential conflicts is to partition the database, mapping each disjoint partition into a basic conflict class. Compound conflict classes are created by grouping basic conflict classes.

In some conservative approaches, such as the OTP protocol [6], a transaction queue is associated to each basic conflict class. In others, such as the NODO protocol [4], [5], transaction queues are associated to compound conflict classes. Transactions that access a given conflict class must traverse its queue.

In any case, the number of transactions allowed to execute concurrently is limited to the number of basic conflict classes defined over the database. Thus, the manner in which the database is partitioned is a determinant factor of the performance of a replication protocol using conservative concurrency control. In fact, protocols such as OTP further require that the application can be completely partitioned as well, since any transaction is restricted to accessing a single basic conflict class.

In replication protocols using optimistic concurrency control [1], [2], [3], potentially conflicting transactions are allowed to execute concurrently and a certification (conflict detection) phase takes place after transaction execution, but before the changes are applied to the database. While this mechanism allows more concurrency, transactions that are later found to conflict are aborted. Notice that the more transactions are allowed to execute concurrently, the more likely it is for conflicts to arise. Also, any transaction is vulnerable to being aborted by other transactions from the moment it starts to execute until is is certified: the longer it takes to execute and certify a given transaction, the more vulnerable it is. This is the caveat of most optimistic concurrency control strategies: when loaded, latency increases and fairness is compromised, particularly for long-running transactions, as exemplified with DBSM [1].

For example, the AKARA database replication protocol [1] mitigates this issue by introducing conservative re-execution for previously aborted transactions, using basic conflict classes to ensure conflict-free scheduling.

There have also been some proposals regarding automatic database partitioning, but these either: target data warehousing scenarios, where update transactions are ignored [12]; attempt to partition the application in an effort to shift some of the load to an application server [13]; or are only able to find a small set of partitions [14].

It is important to point out that this work is not only directly applicable when considering replication protocols with conservative concurrency control, but it also has a wider applicability to any proposal that assumes that real-world databases can be easily and efficiently partitioned into disjoint partitions [15]. Additionally, it provides some insight on the criteria that should be used to evaluate automatic database partitioning for replicated databases.

In the following sections, database and application partitioning, which lead to conflict class definition, are analysed for a synthetic benchmark that simulates the activity of a financial brokerage firm and a real-world application in the same domain.

III. TPC-E

TPC-E [10] is a benchmark that simulates the activities of a brokerage firm which handles customer account management, trade order execution on behalf of customers and the interaction with financial markets. This analysis was based on *tpce-mysql*,[1] an open-source implementation of the TPC-E benchmark. This benchmark defines 33 tables across four domains: customer, broker, market and dimension and 10 main transaction types that operate across the domains. TPC-E's read/write transactions are: Market Feed (MF), Trade Order (TO), Trade Result (TR), Trade Update (TU) and Data Maintenance (DM).[2]

Unlike TPC-C and TPC-W, TPC-E is an open benchmark suite [16]: new requests are received by the System Under Test regardless of the completion of previous requests. A closed benchmark suite does not suitably test replication protocols, since the inherent limit to the number of requests received by the system may obfuscate load/contention issues [1]. We argue that evaluating these protocols using current, more complex and more realistic benchmark suites will lead to significantly different conclusions about these protocols' performance and applicability, particularly in the financial brokerage domain.

TPC-E is clearly documented and conflict classes can be defined by inspection. In this analysis we considered the 1-copy-snapshot-isolation criterion [11]. By analysing the database footprint of each transaction type, we determined the

[1]https://code.launchpad.net/perconadev/perconatools/tpcemysql

[2]The Data Maintenance transaction type operates exclusively on a separate group of tables. As such, it is not relevant for this analysis and is essentially omitted from the discussion that follows.

TABLE I. BASIC CONFLICT CLASSES AND TRANSACTION TYPES

Conf. C.	Table	Transaction Type
C1	trade	MF, TO, TR, TU
C2	trade_history	MF, TO, TR
C3	trade_request	MF, TO
C4	cash_transaction	TR, TU
C5	settlement	TR, TU

TABLE II. COMPOUND CONFLICT CLASSES AND TRANSACTION TYPES (NAÏVE)

Conflict Class	Transaction Type
{C1, C2, C3}	MF, TO
{C1, C2, C4, C5}	TR
{C1, C4, C5}	TU

TABLE III. COMPOUND CONFLICT CLASSES AND TRANSACTION TYPES

Conflict Class	Transaction Type	Write mix
{C1, C2, C3}	MF, TO	48%
{C1, C2, C4, C5}	TR	43%
{C1}	TU_1	3%
{C5}	TU_2	3%
{C4}	TU_3	3%

TABLE IV. IN-DEPTH CONFLICT ANALYSIS: (I)NSERTS, (U)PDATES AND (D)ELETES

	C1	C2	C3	C4	C5
MF	U (PK)	I	D (PK)		
TO	I	I	I		
TR	U (PK)	I		I	I
TU_1	U (PK)				
TU_2					U (PK)
TU_3				U (PK)	

specific set of tables for which write/write conflicts (different transaction types) can occur.

A naïve conflict class definition results from inspecting reads and writes by transaction type and then grouping the tables in conflict classes. Table I depicts the basic conflict classes that can be defined in a table-based manner and the types of transaction that access them. Table II depicts the compound conflict classes can be defined based over the basic conflict classes, so that each transaction accesses a single conflict class, as required by NODO. Notice that because every transaction accesses C1, with a conservative concurrency control, all transactions must be serialized [17].

TPC-E transactions are composed of frames which makes it possible to define 3 sub-transaction types in lieu of TU. Table III depicts the compound conflict classes that can be defined considering TU's sub-transaction types TU_1, TU_2 and TU_3.[3] This would allow up to 3 transactions to execute concurrently using a conservative mechanism. However, after analysing the percentage of transactions of each type relatively to all write transactions in the benchmark's mix, displayed in Table III, we conclude that the majority of the load is concentrated in two non-disjoint compound conflict classes, which leads to same performance bottleneck that occurs for the naïve approach. This means that most of the time, transactions will execute serially.

Table IV details how each transaction type writes each table. For example, MF transactions update `trade` (C1) by primary key, insert one or more rows in `trade_history` and delete one or more rows from `trade_request` by primary key. Assuming a row-level locking model in the underlying database, concurrent inserts do not conflict, nor do inserts and concurrent updates by primary key, or inserts and concurrent deletes by primary key. Concurrent inserts on `trade_history` (C2) also do not conflict because the primary keys are provided as a part of the transaction's arguments: regardless of the order in which the inserts are executed, the end result is the same. Thus, only MF, TR and TU_1 transaction types conflict. Notice that there is no straightforward way to encode this information in a conflict class definition short of defining one conflict class per row, which is impractical. An alternative could be to explore workload-based approaches to database partitioning like Schism [14] which is not, however, directly applicable for this purpose.

The TPC-E recommended way of partitioning the database is to do so by customer identifier, which would effectively partition the trade table horizontally. But, for example, the MF transactions update the `trade` table ignoring the customer. Therefore, MF transactions would likely be distributed across partitions. In general, sharding the database would not prove helpful since it would either require: the exact transaction write set to be known before executing it (since conflict detection is done *a priori*); or the transaction to be added to queues of all conflict classes that match shards of accessed tables, rendering sharding useless.

Moreover, current automatic partitioning tools either do not target OLTP systems or produce a small number of partitions, thus being unsuitable for use with conservative concurrency control (as discussed in Section II).

Protocols based on conservative concurrency control offer inherently limited performance in these circumstances [7]. Still, the TPC-E benchmark can, in fact, be run with a high-level of parallelism, ie. a large number of transactions executing concurrently, with a low abort rate, using optimistic concurrency control [18].

IV. REAL-WORLD APPLICATION

Our case-study real-world application is a production system at a financial operator which provides brokerage and banking services to partners and clients. The main component of the system is an application that features an architectural pattern frequently employed by businesses, particularly SMEs, and showcases many of the challenges that these face: most features of the brokerage system can be traced to the business logic implemented within a RDBMS, using triggers and stored procedures. Globally, the application consists of hundreds of tables, and thousands of triggers and stored procedures. As a consequence of the development strategy, there is no documentation available, either regarding the business processes that govern operation, or the interactions and dependencies between them.

The complexity of this application and the lack of documentation exclude the possibility of defining conflict classes by simple inspection, as done in the previous section for TPC-E. A systematic, yet minimally invasive approach is required. We briefly introduce the technique used to determine appropriate conflict classes and then present an analysis of the conflict

[3]We assumed that each sub-transaction is executed a similar number of times, but a different distribution would lead to the same conclusion.

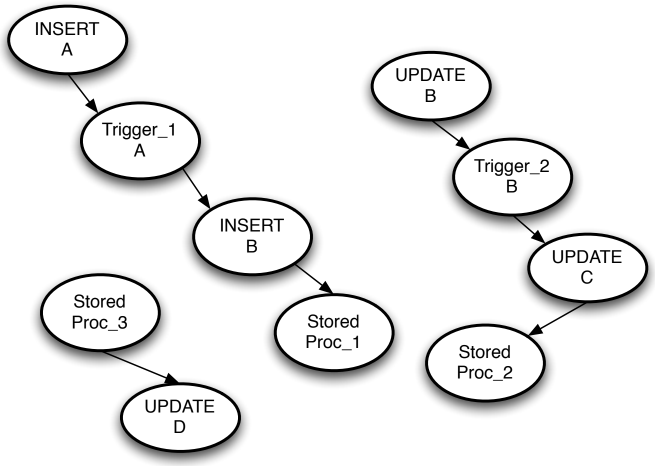

Fig. 1. An example of a write call-graph.

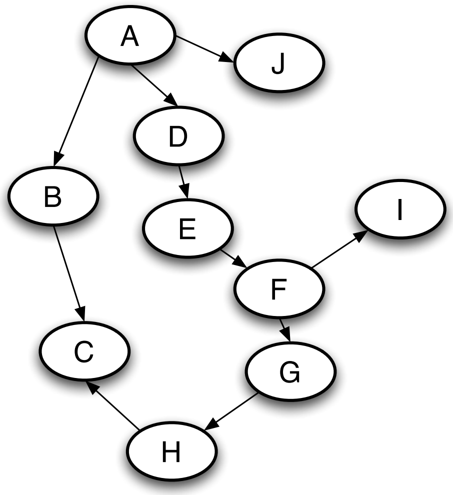

Fig. 2. An example subgraph.

class definition that results from applying this method to the real case-study application.

A. *Conflict class extraction method*

Conflict class extraction starts with the application's SQL source-code, collecting information about table write operations (INSERTs, UPDATEs and DELETEs) and about the structure of the application through the analysis of its triggers and stored procedure invocations. Using this information, the write call graph underlying the mesh of operations can be revealed. Both statistics and useful information about the application structure, which can be used for business process discovery, can be derived from the call graph. The tool generates a directed single-edged call graph. There is a vertex in the graph for each write operation applied to a given table. There is also a vertex for each trigger or stored procedure. An edge (a, b) implies that method (or operation) a calls or triggers method (or operation) b. Figure 1 shows an example of such a graph, featuring table write operations, triggers and stored procedures.

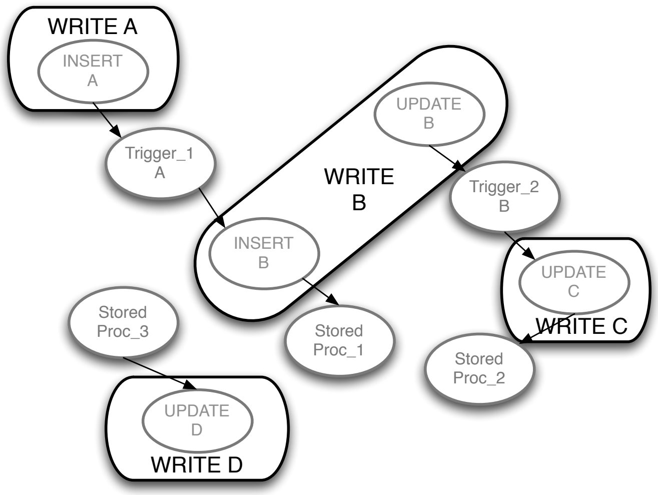

Fig. 3. Vertices representing write operations on the same table are aggregated into a single vertex, which simplifies the graph. Other vertices are folded into the edges, further highlighting the connections between tables.

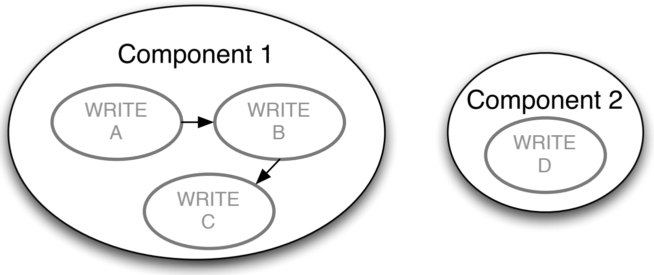

Fig. 4. Weakly-connected components that result from Figure 3. From this analysis, we could conclude that transactions that write on tables A, B or C, do not write on table D.

a) Possible executions: For any given vertex, the set of all of its successors matches the set of operations that might [4] be executed atomically with the operation represented in that vertex. For a given vertex A consider the subgraph, induced by the write call-graph on the vertex set consisting of all of A's successors. Consider the tree in Figure 2 as such a subgraph. In this case, as determined by a depth-first traversal starting at A, a possible execution would be:

- A, B, C, D, E, F, G, H, C, I, J

but, for example,

- A, D, E, F, G, H, C, I, J, B, C
- A, D, E, F, I, G, H, C, J, B, C

would also be possible, because no particular execution order among a vertex's successors can be assumed. Depth-first traversal mimics the nesting behaviour of calls.

By aggregating INSERT, UPDATE and DELETE operations on the same table in a single vertex, we get a simplified graph, which offers a more data-focused view of the application. Figure 3 depicts this process. This aggregated graph was then analysed in terms of connectivity.

[4]Conditional statements are ignored to simplify the approach.

b) Weak Connectivity: A directed graph (or component) is weakly-connected *iff* in the corresponding undirected version, for each pair of vertices, there is a path between them [19]. In terms of this specific analysis, each weakly-connected component corresponds to a self-contained set of triggers, stored procedures and INSERT, UPDATE or DELETE operations that can be executed within the same transaction. Notice that for any given transaction, the set of tables it writes is contained in a single component. Assume there are N components and that S_i is the set of tables for which there is a write vertex in component i. The following properties hold for weakly-connected components:

$$\forall i \neq j \in \{1..N\} \quad S_i \cap S_j = \emptyset \tag{1}$$

and

$$\forall i \in \{1..N\} \quad \bigcup_{i=1}^{N} S_i = \Omega \tag{2}$$

where Ω is the set of all tables. Assume that a conflict class C_i is defined as the set of tables in S_i. From (1) we can conclude that this method results in disjoint conflict classes, one per component. From (2) we can conclude that every read/write table is considered in a conflict class.

Figure 4 shows the weakly-connected components that result from the graph in Figure 3.

Regarding the real-world application, the graph is the best approximation that could be extracted from available data (i.e. source code for stored procedures, no complete traces available) considering only syntatic criteria.

B. Results

Applying the extraction method to the case-study application resulted in the identification of 130 weakly-connected components, which, as stated, correspond to the same number of disjoint conflict classes.

For this application, any replication protocol with conservative concurrency control based on conflict classes at most 130 transactions can be scheduled to execute concurrently.

However, upon examining database logs, we found that most transactions access the same conflict class (component/partition). Therefore, most transactions will be serialized, leading to higher contention that originally expected.

The obvious way to circumvent this issue is to partition the troublesome component. In an effort to do so, the component with the largest number of writes was analysed in search of *cut vertices*: vertices that, if removed from a graph, result in an increase of the number of components [19].

Of the 90 cut vertices found, the one that would lead to the largest number of new components (8) was selected. For each new component, Table V shows its size and the number of transactions that write on it.

Note that partitioning the graph like this would require that the table corresponding to the cut vertex could be added to each of the 8 new components, making these new partitions correspond to non-disjoint compound conflict classes. While some protocols with conservative concurrency control such as NODO are based on non-disjoint compound conflict classes, increasing the number of non-disjoint classes does not increase the level of concurrency allowed by the protocol.

An alternative would be to partition the table corresponding to the cut-vertex, creating 8 new disjoint conflict classes. This would, however, require refactoring the application and restructuring the database. Even assuming that this could be done, the vast majority of the writes remain concentrated in a single component (c5). The cut-vertex strategy could now be used to partition component c5 and so on. Still, the bulk of the writes targets a single table. The next step would be to partition the heavily-written table, which would necessarily lead to table partitioning based on filters over its attributes. In this case, matching accessed items to conflict classes would amount to a satisfiability problem, particularly considering that all "cut-vertex" tables must also be partitioned [7], [20]. In short, the same issues that made sharding unsuitable in Section III are also applicable for this scenario.

Although this technique is simple, it is exaustive (we tried removing all nodes and selected the ones that yield most partitions) and optimistic (we are not sure that these partitions could actually be realized by refactoring and, due to incomplete data, fail to acknowledge some edges), thus providing a very strong counter-argument. We can safely conclude that no easy refactoring exists such that effective conflict classes based on syntatic criteria can be defined.

Partitioning this application is much more complex than partitioning a TPC-E database. Moreover, while the number of disjoint conflict classes that can be defined for this application is much larger than what can be reasonably defined for TPC-E, it does not result in a pratical advantage when considering a conservative concurrency control mechanism. This scenario presents a significant hurdle for the performance of replication protocols with conservative concurrency control, which are thus unsuitable.

Notice that such a scenario would not, however, emcumber a replication protocol with optimistic concurrency control: even if most transactions write on a common table, but on different rows, no conflicts occur.

V. Conclusion

Replication is often used to achieve highly dependable database management services, however, if the result is unable to cope with the actual workloads it can be self-defeating, as the service grinds to a halt with peak loads. We have thus examined to what extent the assumptions of existing protocols hold in the real world: there is not, to the best of our knowledge, published work that provides a concrete counter-example that can be cited. This is precisely what makes our work significant.

First, we examined the TPC-E benchmark and found that despite being well-structured in terms of schema and transactions, the number of disjoint conflict classes that can be reasonably defined is very small, which implied that protocols based on conservative concurrency control are not suitable for this type of application.

Then, we analysed a real-world brokerage application. There was no documentation available regarding this appli-

TABLE V. NEW COMPONENTS, THEIR SIZE AND NUMBER OF WRITES.

Component	c1	c2	c3	c4	c5	c6	c7	c8
Size	1	3	28	12	135	2	10	4
Writes	0	50268	5172	4033	394738	359	4	474

cation, nor were transactions clearly defined. To enable our analysis we devised a general method for extracting a partitioning scheme based on a graph derived from the application's source code. In this case, the number of disjoint conflict classes that could be defined was significantly higher than in TPC-E. However, after looking at the distribution of write operations per tables, we found that the vast majority of the write load falls on a single partition, and at greater detail, on a single table. While this table could be horizontally partitioned, it is not clear how to do so to benefit a conservative concurrency control mechanism. We show that even if the application could hypothetically be refactored leading to a larger number of conflict classes, most transactions would still conflict.

ACKNOWLEDGMENT

This work is funded by the ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project Stratus/FCOMP-01-0124-FEDER-015020 and project FCOMP-01-0124-FEDER-022701.

REFERENCES

[1] A. Correia, J. Pereira, and R. Oliveira, "Akara: A flexible clustering protocol for demanding transactional workloads," *On the Move to Meaningful Internet Systems: OTM 2008*, pp. 691–708, 2008.

[2] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distributed and Parallel Databases*, vol. 14, pp. 71–98, 2003, 10.1023/A:1022887812188. [Online]. Available: http://dx.doi.org/10.1023/A:1022887812188

[3] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-r, a new way to implement database replication," in *Proceedings of the 26th International Conference on Very Large Data Bases*, ser. VLDB '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 134–143. [Online]. Available: http://dl.acm.org/citation.cfm?id=645926.671855

[4] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso, "Scalable replication in database clusters," *Distributed Computing*, pp. 147–160, 2000.

[5] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso, "Improving the scalability of fault-tolerant database clusters," in *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on.* IEEE, 2002, pp. 477–484.

[6] B. Kemme, F. Pedone, G. Alonso, and A. Schiper, "Processing transactions over optimistic atomic broadcast protocols," in *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on.* IEEE, 1999, pp. 424–431.

[7] A. Correia Jr, A. Sousa, L. Soares, J. Pereira, F. Moura, and R. Oliveira, "Group-based replication of on-line transaction processing servers," *Dependable Computing*, pp. 245–260, 2005.

[8] *TPC Benchmark C - Standard Specification*, Transaction Processing Performance Council (TPC) Std., 2001.

[9] *TPC Benchmark W - Standard Specification*, Transaction Processing Performance Council (TPC) Std., August 2001.

[10] *TPC Benchmark E - Standard Specification*, Transaction Processing Performance Council (TPC) Std., June 2010.

[11] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris, "Middleware based data replication providing snapshot isolation," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data.* ACM, 2005, pp. 419–430.

[12] J. Rao, C. Zhang, N. Megiddo, and G. Lohman, "Automating physical database design in a parallel database," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '02. New York, NY, USA: ACM, 2002, pp. 558–569. [Online]. Available: http://doi.acm.org/10.1145/564691.564757

[13] A. Cheung, S. Madden, O. Arden, and A. C. Myers, "Automatic partitioning of database applications," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1471–1482, 2012.

[14] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 48–57, 2010.

[15] D. Sciascia, F. Pedone, and F. Junqueira, "Scalable deferred update replication," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, june 2012, pp. 1 –12.

[16] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open versus closed: a cautionary tale," in *Proeceedings of the 3rd Symposium on Networked Systems Design and Implementation*, 2006.

[17] M. Patiño Martinez, R. Jiménez-Peris, K. Bettina, and G. Alonso, "Middle-r: Consistent database replication at the middleware level," *ACM Trans. Comput. Syst.*, vol. 23, pp. 375–423, November 2005. [Online]. Available: http://doi.acm.org/10.1145/1113574.1113576

[18] A. Nunes, R. Oliveira, and J. Pereira, "Ajitts: Adaptive just-in-time transaction scheduling," in *Distributed Applications and Interoperable Systems.* Springer, 2013, pp. 57–70.

[19] G. Chartrand and L. Lesniak, *Graphs & Digraphs.* Chapman & Hall, 1996.

[20] S. Guo, W. Sun, and M. Weiss, "Solving satisfiability and implication problems in database systems," *ACM Transactions on Database Systems (TODS)*, vol. 21, no. 2, pp. 270–293, 1996.