Performance Trade-offs in Transactional Systems

Rafael Soares
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa
Lisboa, Portugal
joao.rafael.pinto.soares@tecnico.ulisboa.pt

Luís Rodrigues
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa
Lisboa, Portugal
ler@tecnico.ulisboa.pt

Abstract

During the last decade a number of systems supporting different forms of distributed transactions have been proposed, each implementing a different performance trade-off in the design space. In this paper we collect the performance features that have been identified by previous works and offer a systematic analysis of known results regarding the impossibility of achieving certain combinations of desirable properties along these dimensions. We also compare previous transactional systems in the light of this set of desirable performance aspects. Finally we discuss how certain combinations of features may be leverage to guide new transactional system designs.

CCS Concepts: • **Information systems** → **Database design and models**; **Distributed storage**.

Keywords: Distributed Transactions, Freshness, Consistency

ACM Reference Format:
Rafael Soares and Luís Rodrigues. 2023. Performance Trade-offs in Transactional Systems. In *10th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '23), May 8, 2023, Rome, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3578358.3591325

1 Introduction

Systems that offer transactional guarantees for partitioned storage are often designed to meet different performance goals. This makes it hard to compare existing work because, typically, one system does not outperform the other in absolute terms; different systems make different trade-offs among (sometimes conflicting) performance goals. Ideally, one would like to offer the strongest consistency level, namely strict serializability, with optimal performance. Although this goal is intuitive, defining "optimal" performance in this context may be ambiguous, in particular because there are multiple criteria that can be used to assess the performance of a given system. In practice, a given system may favour one criteria over another criteria and may also opt to relax consistency to achieve better performance. This makes it hard to compare existing systems and to understand which combinations of consistency properties and desirable performance goals can be achieved in practice.

In this paper we analyse previous works that support transactions in distributed partitioned storage systems and present the set of performance goals that have been addressed in their design. Several of these works have identified the impossibility of supporting specific combinations of desirable performance properties; we also aim at offering a systematic analysis of these known impossibility results. We then use the full set of performance goals and impossibility results to compare how previous works embody different performance trade-offs, identifying some combinations that are theoretically possible but never were implemented by any state-of-the-art system.

PaPoC '23, May 8, 2023, Rome, Italy

ACM ISBN 979-8-4007-0086-6/23/05.
https://doi.org/10.1145/3578358.3591325

2 System Model

We consider systems that implement a key-value store that is partitioned among multiple servers. Servers can be in the same or in different geographical locations. Servers can be replicated, but we do not address replication explicitly in our study. Instead, we simply assume that each partition is linearizable (this assumption is common among the works we cite in this study).

Clients read and/or write the key-value store using transactions that may access multiple keys, possibly stored by different partitions. We consider transactions that can be read-only or write-only. Note that some surveyed works support more general read-write transactions, which for some consistency criteria, can be modelled as a read-only transaction followed by a write-only transaction.

Transactions can be one-shot (when all keys are read or written in parallel in a single operation) or multi-shot (where a transaction can execute multiple read or write operations). Multi-shot write transactions can typically be converted in a single-shot write transaction by buffering all writes until commit time. Again, some of the systems surveyed in this paper only support one-shot transactions. Note that any impossibility results that applies to one-shot transactions are also valid for multi-short transactions.

Finally, it is worth mentioning that most systems require some form of inter-partition coordination to implement write

transactions. How this coordination is implemented is orthogonal to the trade-offs discussed in this paper but most systems use a 2-phase commit protocol.

3 Performance Goals

In this section, we enumerate the multiple performance goals that have been addressed by previous work.

3.1 Return Mutually Consistent Values

Transactions that read from multiple partitions should return mutually consistent values. In this paper we only consider systems that only return versions that have already been committed, a property known as *read committed*. Read committed does not impose any restriction on the possible combination of returned values. The following criteria can be used to further restrict the set of values that are admissible: i) *order-preserving visibility* ensures that snapshots preserve some order relation (monotonic writes, causal order, and/ or total order) and that no gaps are observed in that order relation (as defined in Tomsic et al [19]), and ii) *atomic visibility* prevents the "read skew" phenomenon [4] (if a transaction reads data written by another transaction it must observe all the updates written by that transaction).

3.2 Terminate in Two Communication Steps

Any distributed algorithm requires the exchange of messages among participants. The number of communications steps is a performance metric that captures the length of the sequence of message exchanges required to terminate the algorithm. For instance, a remote invocation takes two communication steps because it requires the transmission of a request followed by the transmission of a reply. In a setting where clients do not locally store copies of all keys, at least two communication steps are required to terminate a read or write transaction. Interestingly, it is impossible to achieve optimal number of communication steps while satisfying other performance goals, such as returning mutually consistent values that are fresh.

Several previous works use more than two communication steps. For instance, some system require the client to coordinate with the partitions to identify a consistent snapshot before performing the actual read operation [2, 14, 16–18], others execute a first round of read request without coordination but may be required to execute additional rounds if the results obtained are not mutually consistent [11].

In any case, in general terms, the lower the number of communication steps the better. First because each additional step adds latency to the operation, which is particularly relevant in geo-partitioned systems, where propagation latency is the main source of overhead [15]. Additionally, systems that rely on long multi-shot transactions like Facebook [1] suffer a cumulative performance penalty for each operation executed. Second, more steps are associated with additional CPU and bandwidth utilization, and may negatively affect the aggregated throughput of the system [8, 13].

3.3 One Version per Read

When reading from multiple partitions, one needs to ensure that the client observes versions that are mutually consistent. To increase the odds of obtaining mutually consistent results, some approaches require the client to read multiple versions of the same key in a single round. In the extreme case, if all past versions of an object are returned, it would always be possible to find mutually consistent versions. Ideally, one would like to transfer to the client a single version per object requested, to avoid spending CPU cycles and network bandwidth transferring redundant key versions.

An algorithm that is able to terminate reads in two communication steps (therefore, contacting each partition only once) and that reads one version per read is said to satisfy the *one response per read* property [12].

3.4 Supports Non-blocking Replies

We say that a system supports non-blocking replies if a partition can reply to a read or write request as soon as it is received, without waiting for some predicate to become true.

Systems that use some form of locking do not support non-blocking replies because the processing of a request may be blocked until the lock is released. Systems that require requests to be processed at a given logical (or physical) time fail to support non-blocking replies, because a request may be blocked until the time at the server advances. Systems that require requests to be totally ordered before being processed also prevent non-blocking replies. In several cases, the condition that blocks a request may take long to be overcome (for instance, a lock can take an arbitrarily long time to be released), adding a latency penalty to the system.

An algorithm that satisfies one response per read and supports non-blocking replies is said to be *latency optimal* [12].

3.5 Return Fresh Results

In systems that keep a single version of each key, partitions always return the most recent version known by the partition. Unfortunately, when reads are executed concurrently with write operations, the values returned by different partitions may not be mutually consistent. By keeping multiple version of each key it becomes easier to find (and return) a set of mutually consistent versions. However, this opens the door for returning arbitrarily old values. In the limit, by absurd, a system could simply discard all the updates and always return the initial values of each object. Freshness is an informal property that attempts to capture how "recent" are the values returned to clients.

The best freshness any system can aim for is to return the values written by the most recent write transaction that has committed before the read transaction was initiated (i.e., to ensure linearizable updates). Systems that do not attempt

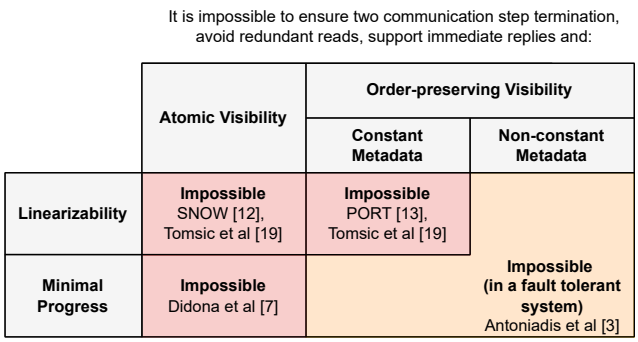

Figure 1. Impossibility results visualized.

to enforce linearizability often use the following metric to measure freshness: the length of the time interval between the commit time of the value returned and the commit time of the last update (the shorter, the better). An implementation may impose a maximum delay for updates to be become visible, i.e., state that if an update commits at time t, any read operation starting after time $t + \delta$ should not return versions older than t.

The weakest definition of freshness is denoted *minimal progress* [7], and states that any write must eventually become visible. Anything weaker, such as reading from indefinitely stale snapshots [13] is considered stale.

3.6 Use Constant-Size Metadata

Different techniques can be used to ensure that mutually consistent versions are returned. One is to use locks, to prevent versions that may yield inconsistent results to be returned. Another is to tag versions with metadata that can be used to check if two versions are mutually consistent. Metadata can assume many forms, including version numbers, logical or physical timestamps, vector clocks, or explicitly dependency lists. There is a well known trade-off between the size of the metadata and the accuracy of the information that can be extracted from the metadata [5]. Unfortunately, if more metadata provides more information, it also consumes more storage, network, and processing resources. Ultimately, if the metadata size increases linearly, such as with the number of partitions (such as when vector clock or explicit dependencies are used), the system becomes inherently non-scalable. Therefore, one would like to use metadata of constant size.

4 Impossibility Results

One would like to build a transactional system that could achieve all performance goals listed above. Unfortunately, some of these goals are conflicting with each other. This has been captured by a number of impossibility results that can be found in the literature, that are listed in this section. A view of the impossibility results is depicted in Figure 1.

It is impossible to design a system that terminates in two communication steps, returns one version per read, supports non-blocking replies, returns mutually consistent values that that respect atomic visibility, and ensures minimal progress.

This result was obtained by Didona et al [7] and it is a generalization of a previous result, known as the SNOW theorem [12], that considered linearizability instead of the minimal progress as the freshness guarantee.

If one relaxes the consistency guarantees, and opt to offer only order-preserving visibility, the impossibility result no longer applies. However, any system that attempts to combine these properties may require the use of metadata whose size is not constant. This is captured by the following result:

It is impossible to design a system that terminates in two communication steps, returns one version per read, supports non-blocking replies, returns mutually consistent values that respect order-preserving visibility, offers linearizability, and uses constant-size metadata.

This impossibility result was first captured in PORT [13]. Tomsic et al [19] obtained a similar result using a slightly different notion of freshness.

The impossibility results listed above are valid even if the state maintained by each partition is not replicated. However, to achieve fault-tolerance, implementations may be required to replicate the state of each partition, including transient state. In this case, if a read operation requires some transient state to be maintained at the server (for instance, a read-lock), the state needs to be replicated, which prevents termination from being achieved in two communication steps. This effect is captured by the following result:

It is impossible to design a system that terminates in two communication steps, returns one version per read, supports non-blocking replies, returns mutually consistent values that respect order-preserving visibility and offers minimal progress without updating partitions on read transactions.

This result was obtained by Antoniadis et al [3], and shows it is impossible to obtain all previously mentioned properties in a fault-tolerant system, as this combination of features requires clients to update partitions on read transactions and, in turn, replicating these updates involves additional communication steps.

As we will discuss later, systems such as COPS-SNOW [12], require a record of each read operation to be maintained by each partition. This allows to implement non-blocking reads in two communication steps if one disregards to costs associated with replicating those records. In any case, any system that requires read operations to change the state of the partitions may become prohibitively expensive in practice, given that most workloads are read dominated and, therefore, it is desirable to have reads be *invisible*[3].

System	Read Tx	Write Tx	Read/Write Tx	Transaction Type	Consistency
COPS-SNOW [12]	✓	✗	✗	Static	TCC
Eiger-PORT [13]	✓	✓	✗	Static	TCC
Eiger [11]	✓	✓	✗	Static	TCC
Cure [2]	✓	✓	✓	Dynamic	TCC
Wren [16]	✓	✓	✓	Dynamic	TCC
PaRiS [17]	✓	✓	✓	Dynamic	TCC
Clock-SI [9]	✓	✓	✓	Dynamic	SI
Spanner-RO [6]	✓	✗	✗	Dynamic	SS
Spanner-RW [6]	✓	✓	✓	Dynamic	SS
Calvin [18]	✓	✓	✓	Static	SS
Rococo [14]	✓	✓	✓	Static	SS

Table 1. Systems transactional models. SS represents Strict Serializability, SI represents Snapshot Isolation, TCC represents Transactional Causal Consistency. Static transactions need to know the entire read set and write set when the transaction starts.

5 Existing Systems

In this section, we provide a comparison of existing systems from the point of view of the performance goals enumerated early. Given the impossibility results, existing systems need to sacrifice some of the performance goals. This comparison highlights which sacrifices the designers have made in each case. This comparison is illustrated in Figure 2 as a directed acyclic graph (DAG), where each vertex represent a certain combination of performance goals (each performance goal is represented by a slice of a circle). Vertexes in the DAG capture combinations of performance properties that: i) have been implemented by previous work or, ii) that to the best of our knowledge, have never been implemented or, iii) that are impossible, according to the impossibility results discussed previously. Children of a vertex always add one or more additional performance goals to the parent vertex: the label on the edge clarifies which performance goal is added.

Note that in Figure 2 we only focus on the performance goals and do not consider other features of the systems, such as if they support read-write transactions, dynamic transactions, etc. Therefore, the reader must be aware that if a system A satisfies more performance goals than another system B, this does not make A necessarily strictly superior than B. The information in Figure 2 is complemented by Table 1, that includes some of of these additional features. Additionally, we consider that all systems should be fault-tolerant and, as such, are affected by the result presented in Antoniadis et al [3].

The root of the DAG is a system that only meets the performance goal of returning one version per read; all the systems we review have this property. From this vertex, we consider three different branches: systems that offer linerizability (i.e., where writes become immediately visible as soon as a the corresponding update transaction terminates), systems that offer only minimal progress, and a system that sacrifices progress to ensure all other properties. In the following we discuss each of these branches separately.

5.1 Systems that Offer Linearizability

Both Calvin [18] and Rococo [14] combine linearizability and atomic visibility, providing strict serializability, with constant metadata. Note that providing strict serializability requires update transactions to be totally ordered.

Calvin uses a distributed set of sequencers to batch and order transactions. Each sequencer batches transaction requests during a given interval of time. Once complete, each batch is first replicated, which can be achieved using some form of Paxos [10], and then each partition is sent their designated transactional inputs from each batch. Partitions define the global total order of transactions by merging in a deterministic manner the batches obtained from each sequencer. Then, in each partition, at the beginning of the execution, transactions issue requests for all the locks they need; these requests are scheduled by a single thread in the total order defined above. Finally, transactions can execute concurrently, waiting for the lock requests to be satisfied when needed and releasing all locks upon termination.

Rococo [14] only orders update transactions. Updates are performed in two steps: first all objects are locked and only after all locks have been acquired, the new values are written (and locks released). Read transactions are not totally ordered w.r.t write transactions. Thus, read transactions can read inconsistent versions when reading in a single round. To ensure that consistent versions are returned, Rococo performs multiple rounds of reads until two rounds return exactly the same values. If two consecutive rounds return the same values, this means that the read transaction was executed without observing the effects of other (concurrent) update transactions. Unfortunately, with this protocol, read transactions are only guaranteed to return in periods where there no concurrent conflicting update transactions.

Adding additional performance goals to Calvin/Rococo requires adding either termination in two steps or adding support for non-blocking replies. Note that adding these two additional goals is impossible according to the impossibility results of PORT [13]. Spanner [6] adds termination in two steps for read-only transactions by relying on synchronized clocks. A client can read its own clock and select a read snapshot that is guaranteed to include all update transactions that have terminated in the past. To the best of our knowledge, no system has added non-blocking replies to the performance goals achieved by Calvin/ Rococo, although such combination is not covered by the impossibility result.

5.2 Systems that Offer Minimal Progress

Among the systems that offer only minimal progress, we distinguish those that offer atomic visibility from those that offer the weaker order-preserving visibility.

Cure [2] ensures atomic visibility and minimal progress. Cure uses a vector clock, where each entry is a hybrid clock that captures the state of a different data center. This allows

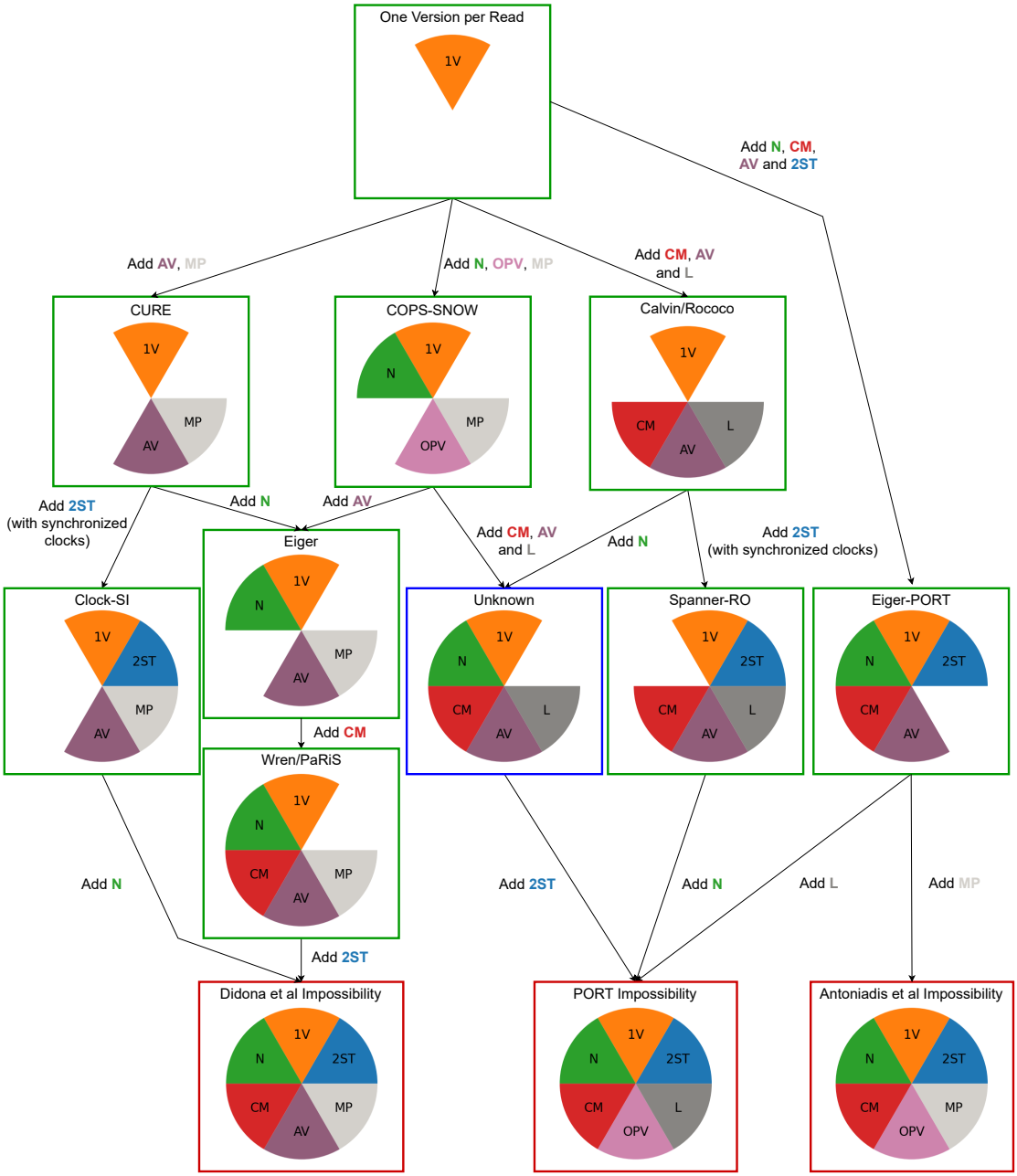

Figure 2. Systems comparison. Slices of the pie capture performance goals: one version per read (1V), non-blocking (N), constant metadata (CM), order-preserving visibility (OPV), atomic visibility (AV), minimal progress (MP), linearizability (L), and two-step termination (2ST). Color of the frame captures possibility/impossibility: green borders for existing systems, red borders for impossible combinations, and blue borders for combinations for which there is no system or impossibility result.

to represent a global snapshot that includes the state of different data centers at different points in time. When executing a read transaction, the client sets the read snapshot as follows. The entry associated with the local data center is set to the value of the local loosely synchronized clock. The entries associated to remote data centers are defined by a global stabilization procedure that keeps track of which updates have been applied to all partitions of each data center. Minimal progress is guaranteed because clients always observe the results of previous update transactions executed in the local data center. However, two-step termination is sacrificed, as clients must first contact a partition to obtain the read snapshot. Updates performed at remote data centers can be arbitrarily delayed if there is a network partition. Read

transactions may block due to clock skews, namely when the clock of the client is ahead of the clocks of some partitions.

COPS-SNOW [12] only supports simple writes and, as such, only provides order-preserving visibility. It implements efficient reads, that can be executed in just two steps and offer non-blocking replies. It achieves this by sacrificing the performance of write operations, that may need to contact multiple partitions to complete. Assume two different keys x and y, and two writes $x_1 \rightarrow y_2$. A read transaction that reads x before x_1 must read y before y_2 (to avoid violating causality). To ensure that the y partition can locally decide, without blocking, if a read transaction should observe the new version y_2 or a previous version instead, when y_2 is written it is tagged with the identifiers of all concurrent read transactions that must be ordered before y_2 This requires read transactions to update partitions with a record of the read operation (in a fault-tolerant setting, the replication of this state may involve additional communication steps). In COPS-SNOW, the set of transactions that must be ordered before y_2 is only computed when y_2 is written; this requires y's partition to contact all partitions that have writes in the causal past of y_2. Note that COPS-SNOW moves part of the coordination required to provided consistent reads off-path, embedding it in the writes. Although this makes reads more efficient, the penalty of the off-path coordination in the global system performance can still be significant[8].

If one considers to satisfy additional performance goals in a system similar to Cure, one may either add non-blocking replies or two step termination. Adding both is not possible, as it would result into a combination covered by the Didona et al [7] impossibility result. Adding two step termination could be achieved by resorting to loosely synchronized clocks, such as in Spanner. Clock-SI [9] uses the local clock to decide on the read snapshot; on one hand, this allows to achieve termination in two communication steps but, on the other hand, requests may be blocked due to the clock skew. While in Clock-SI only partitions have synchronized clocks, requiring clients to contact a partition to serve as transaction coordinator, the client operation could be extended with synchronized clocks to remove this additional communication step. Adding non-blocking replies could be achieved by letting the client read the most recent version of each object and subsequently check if it was able to read from a consistent snapshot; if not, the client chooses the freshest snapshot from the read objects and forces additional read rounds to obtain the missing mutually consistent values from that snapshot. This approach is implemented by Eiger [11]. In order to achieve termination in a limited number of steps, Eiger requires linearly sized metadata. One may further improve these optimal goals by adding constantly sized metadata. Following Cure design, instead of using a vector clock to define the global state of the system, one may use a single scalar provided by the global stabilization procedure, identifying the snapshot with the freshest updates applied by all partitions at all datacenters. Systems like Wren [16] and PaRiS [17] use this strategy to provide non-blocking replies and constant metadata.

5.3 Systems That Do Not Offer Freshness Guarantees

One may also sacrifice freshness to maintain all other optimal guarantees. Sacrificing freshness implies risking reading indefinitely stale values but, on the other hand, permits to read only from transactions that are known to have completed at all data-centers and/or partitions. Eiger-PORT [13] achieves this by running a global stabilization detection protocol, where each client keeps track of the last transactions that have been completed on each contacted server. It then uses this information to read only from update transactions that have terminated, i.e., whose values are already visible at all nodes. This can be achieved with constant metadata if transactions are tagged with a logical clock. Adding freshness and linearizability or minimal progress to a system such as Eiger-PORT, while preserving all other goals, was shown to be impossible.

6 Conclusion

In this work we have discussed some performance trade-offs in transactional systems. We have compiled a list of performance goals that have been addressed by related work and identified known impossibility results regarding the feasibility of combining these goals in a different systems. We have then compared several existing systems from the perspective of what combinations of performance goals they are able to achieve. Our analysis shows that there are some combinations of performance goals that have not been satisfied by previous work but that are also not covered by the known impossibility results.

7 Future Work

For future work, we will extend our survey to include other system characteristics like replication and it's impact on system design. We will also include a more detailed explanation of the costs associated with each performance goal, to better guide developers in their choice of system. Furthermore, we also plan to study if some combinations of properties that cannot be achieved deterministically may still possible to achieve with high probability in realistic scenarios of practical relevance.

Acknowledgments

The authors are grateful to Manuel Bravo and to the anonymous reviewers for their comments on a early version of this paper. This work was partially funded by Fundação para a Ciência e Tecnologia (FCT) under grant UI/BD/153590/2022 and via project UIDB/50021/2020 and DACOMICO (via OE withref. PTDC/CCI-COM/2156/2021).

References

[1] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. 2015. Challenges to Adopting Stronger Consistency at Scale. In *HOTOS'15* (Kartause Ittingen, Switzerland).

[2] Deepthi Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. 2016. Cure: Strong Semantics Meets High Availability and Low Latency. In *ICDCS '16* (Nara, Japan).

[3] Karolos Antoniadis, Diego Didona, Rachid Guerraoui, and Willy Zwaenepoel. 2020. The Impossibility of Fast Transactions. In *IPDPS '20* (New Orleans (LA), USA).

[4] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD '95* (San Jose (CA), USA).

[5] Manuel Bravo, Nuno Diegues, Jingna Zeng, Paolo Romano, and Luís E. T. Rodrigues. 2015. On the use of Clocks to Enforce Consistency in the Cloud. *IEEE Data Eng. Bull.* 38, 1 (2015), 18–31.

[6] James Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally Distributed Database. In *OSDI '12* (Hollywood (CA), USA).

[7] Diego Didona, Panagiota Fatourou, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. 2019. Distributed Transactional Systems Cannot Be Fast. In *SPAA '19* (Phoenix (AZ), USA).

[8] Diego Didona, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. 2018. Causal Consistency and Latency Optimality: Friend or Foe? *VLDB* 11, 11 (2018), 1618–1632.

[9] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. 2013. Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks. In *SRDS '13* (Braga, Portugal).

[10] Leslie Lamport. 1998. The Part-Time Parliament. *ACM TOCS* 16, 2 (may 1998), 133–169.

[11] Wyatt Lloyd, Michael Freedman, Michael Kaminsky, and David G Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *OSDI '13* (Lombard (IL), USA).

[12] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. 2016. The SNOW Theorem and Latency-Optimal Read-Only Transactions. In *OSDI '16* (Savannah (GA), USA).

[13] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. 2020. Performance-Optimal Read-Only Transactions. In *OSDI '20* (Virtual Event).

[14] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions. In *OSDI '14* (Broomfield (CO), USA).

[15] Srinivas Narayana, Joe Wenjie Jiang, Jennifer Rexford, and Mung Chiang. 2012. To coordinate or not to coordinate? wide-area traffic management for data centers. *Dept. Comput. Sci., Princeton Univ., Princeton, NJ, USA, Tech. Rep. TR-998-15* (2012).

[16] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. 2018. Wren: Nonblocking reads in a partitioned transactional causally consistent data store. In *DSN '18* (Luxembourg City, Luxembourg).

[17] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. 2019. PaRiS: Causally Consistent Transactions with Non-blocking Reads and Partial Replication. In *ICDCS '19* (Dallas (TX), USA).

[18] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *SIGMOD '12* (Scottsdale (AZ), USA).

[19] Alejandro Tomsic, Manuel Bravo, and Marc Shapiro. 2018. Distributed transactional reads: the strong, the quick, the fresh & the impossible. In *Middleware '18* (Rennes, France).