Consistency Without Borders

Peter Alvaro, Peter Bailis, Neil Conway, Joseph M. Hellerstein

Abstract

Distributed consistency is a perennial research topic; in recent years it has become an urgent practical matter as well. The research literature has focused on enforcing various flavors of consistency at the I/O layer, such as linearizability of read/write registers. For practitioners, strong I/O consistency is often impractical at scale, while looser forms of I/O consistency are difficult to map to application-level concerns. Instead, it is common for developers to take matters of distributed consistency into their own hands, leading to application-specific solutions that are tricky to write, test and maintain.

In this paper, we agitate for the technical community to shift its attention to approaches that lie between the extremes of I/O-level and application-level consistency. We ground our discussion in early work in the area, including our own experiences building programmer tools and languages that help developers guarantee distributed consistency at the application level. Much remains to be done, and we highlight some of the challenges that we feel deserve more attention.

1 Introduction

Cloud computing infrastructure and mobile devices have become widespread in a relatively short period of time. For many programmers, this means that distributed systems have quickly become their primary model of computation [14]. A growing proportion of developers must address the challenges of distribution, ensuring correct application behavior despite asynchrony, concurrency, and partial failure. Over the past several decades, developers of distributed systems have traditionally relied on *I/O-level* techniques [7, 10, 25, 27, 34, 44, 45, 57] such as serializable transactions [12], linearizable data stores [40], and atomic broadcast [29] to ensure correct behavior. These techniques encapsulate much of the complexity of distributed programming in the storage or messaging layers, simplifying application development.

Despite the historical success of these low-level approaches, there is mounting evidence that an I/O-level approach to consistency guarantees becomes increasingly problematic as systems grow to global scale. Strong consistency protocols require coordination, which has significant latency and availability costs, and can lead to unpredictable behavior under load [14]. Perhaps most importantly, I/O-level interfaces divorce operations from their applications' semantic context: while programmers reason in terms of application-level correctness properties, storage systems provide guarantees about low-level operations such as reads and writes to opaque registers. This forces developers to manually translate between application-level concepts and low-level storage and messaging operations, a task that is error-prone and requires extensive knowledge of the underlying system. In turn, the storage or messaging infrastructure cannot leverage application semantics [8, 27, 42], leading to conservative protocols with needlessly high latency and reduced availability. As a result, many practitioners choose to avoid I/O-level consistency mechanisms whenever possible [14, 18, 59], instead relying on informal application-level design patterns to achieve correct behavior [36, 37]. These patterns are insightful, but challenging to correctly implement, test and maintain in each particular application scenario.

The bulk of the research on consistency has focused on the subtleties of various consistency techniques—both strong and relaxed—at the I/O layer. That work does little to alleviate the application developer's end-to-end consistency conundrum: whether to delegate to a strongly consistent I/O infrastructure, or to shoulder the software engineering burden of designing, testing and maintaining application-specific consistency code.

We believe it is imperative for the technical community to re-frame this discussion by offering consistency solutions that inhabit the design space in between these two extremes. We envision a range of consistency techniques *across the stack*—including at the *object*, *dataflow*

SoCC'13, October 1–3 2013, Santa Clara, CA, USA.

http://dx.doi.org/10.1145/2523616.2523632

and *language* levels—with various tradeoffs between efficiency, generality, and engineering complexity.

In this paper, we motivate the further (and, in some cases, renewed) study of these alternative approaches to distributed consistency. We offer an informal taxonomy of strategies and associated insights both from our own work as well as recent developments from other researchers. We identify opportunities for further exploration and highlight several areas in which non-I/O-level mechanisms have already begun to succeed in the wild. As a community, we have an opportunity to demonstrate that correctness at scale is not in conflict with availability, performance, and programmer productivity.

2 Case Study

To illustrate how consistent outcomes can be achieved at several different places in the software stack, consider a scenario in which several programs manipulate a directed graph. This system can be divided into (at least) two tiers: the *storage tier* (e.g., a DBMS or key-value store) manages the persistence of the graph data structure, while the *application tier* accesses the graph by submitting read and write operations against the graph store. To improve fault tolerance and scalability, we assume the graph is replicated and partitioned.

We consider two applications that use this graph store:

1. The *deadlock detector* queries a "waits-for" graph that records dependencies between processes. The task is to check whether the graph contains a cycle, which indicates a deadlock [21].

2. The *garbage collector* uses a "refers-to" graph to record references between a collection of distributed objects. The objective is to detect strongly connected components that are not reachable from a distinguished "root" object; such components can safely be reclaimed [1].

Both programs have similar correctness requirements. For the deadlock detector, all deadlocks should eventually be reported, with no false positives. Similarly, the garbage collector should ensure that all unreachable components are eventually discovered, and no "live" objects are returned as suitable for garbage collection.

How should we map these application-level semantics down to the low-level storage abstraction? In the remainder of the paper, we will consider these semantics at the traditional extremes (Section 3), and via convergent objects (Section 4), distributed dataflows (Section 5), and whole program analysis (Section 6).

Before we do so, we note that *neither* example application requires a strong consistency guarantee such as linearizability or serializability to maintain correctness. Both deadlock and unreferenced memory are *stable* properties [20]: once such a property holds, it will persist (until a corrective action is taken, such as aborting one of the participants in a deadlock). However, deadlock is also a *strong stable* property [52]: it can be detected given a subset of the global graph. This implies that the deadlock detector only requires very weak semantic guarantees from the graph store: as long as all waits-for edges are eventually observed (regardless of order), all deadlocked transactions will be detected. In contrast, the garbage collector requires global knowledge: just because one partition of the graph store contains no references to an object does not imply there are no references globally. Hence, garbage collection requires stronger consistency guarantees than deadlock detection. However, neither requires "strong" consistency—and its concomitant costs in decreased availability and increased latency—to achieve correct behavior.

3 Consistency at the Extremes

To guarantee that application-level invariants are never violated, programmers are often forced to choose between one of two "extreme" strategies: *generic* I/O-level interfaces that control the order of events such as messages or reads and writes, and *custom*, typically ad hoc solutions that force application logic to assume all responsibility for ensuring that correctness invariants are preserved. Both approaches have significant limitations.

3.1 I/O-Level Consistency

Database systems have long provided guarantees about the interleaving of "conflicting" operations on shared state [13]. These guarantees are defined in terms of storage operations like reads and writes: for example, conflict serializability defines a conflict as two operations on the same data item submitted by different transactions, in which at least one of the operations is a write [50]. Although originally defined for centralized systems, these consistency models have subsequently been applied to distributed data management [12]. A wide variety of consistency models have been proposed that make different tradeoffs between latency, availability, and the space of permissible operation interleavings (e.g., [7, 25, 44, 45, 57]). Similarly, distributed systems often rely on ordering guarantees on *messages* that reference shared state. Techniques such as state machine replication [53] ensure consistency among replicas of a distributed service by guaranteeing that messages are delivered in the same order to all replicas. Group communication systems [15] provide a variety of ordering guarantees for broadcast messages.

Encapsulating consistency in an I/O-level abstraction allows for a clean separation between application logic and low-level storage/messaging concerns. However, there are two major problems with this approach. First, applications typically interact with such infrastructure using a narrow API based on read and write operations or uninterpreted messages. While this encourages loose coupling, it means that application-level semantic knowledge is lost and hence the underlying system must make conservative assumptions (e.g., that two writes to the same object, or two messages to the same agent, cannot be safely reordered). These conservative assumptions lead to inefficiency: for example, recent designs for scalable serializable data stores offer per-object throughput limited to 10's to 100's of updates per second [26, 56]. Second, developers who wish to relax I/O-level consistency guarantees must translate application-level invariants to properties of execution traces. This is difficult to do and often requires expert knowledge of both the particular storage/messaging systems in use, and the subtleties of the application requirements.

Returning to the case studies introduced in Section 2, we see that I/O-level consistency mechanisms are unsatisfactory. The graph store has no knowledge of application semantics, so any correctness guarantees must be specified in terms of read/write traces: for example, the store might promise that all reads and writes will be executed in a linearizable manner [40]. While such a strong ordering guarantee will ensure consistent outcomes, applications like those presented in the case study will "overpay" in performance and availability penalties for stronger guarantees than they require. In fact, as we observed in Section 2, deadlock detection places no restrictions on the order in which new waits-for edges are recorded, while garbage collection requires only the coarse-grained constraint that the entire reachability relation is computed before concluding that an object is unreferenced.

3.2 Application-Level Consistency

A perennial alternative to storage-level consistency is to delegate responsibility for maintaining correctness invariants to application logic. With richer semantic knowledge it is often possible to avoid the costs of conservative global coordination in favor of custom, application-level solutions [23, 28, 36, 37]. For example, the deadlock detector could be implemented in a pipelined fashion, emitting strongly connected graph components as soon as they are discovered, while the garbage collector could be implemented in terms of *mark* and *sweep* phases that are strictly synchronized.

Unfortunately, such ad hoc solutions push complexity to application developers: programmers must ensure correct behavior for any possible event schedule, rather than, say, only serializable schedules. Moreover, application-level consistency makes code reuse difficult, forcing programmers to design, implement, and validate consistency mechanisms from scratch for each new module. Intuitively, the two case study programs are quite similar—both check properties of the transitive closure of a graph—but in practice, application-level consistency mechanisms from one program would not be reusable for the other.

4 Object-Level Consistency

The two approaches described in Section 3—I/O guarantees and custom application logic—represent the status quo in ensuring consistent program outcomes. Enforcing consistency at the extremes leads to solutions that are either ill-fitted or over-fitted to the applications whose invariants they protect. As a result, programmers must sacrifice either efficiency or generality/reusability. In the remainder of this paper, we advocate an aggressive exploration of intermediate solutions between these two extremes. We begin closest to the storage layer, studying proposals for how opaque storage APIs can be enriched with semantic knowledge in a "piecemeal" fashion; subsequent sections move closer to the application layer.

Systems that provide guarantees about uninterpreted read-write traces are forced to make conservative assumptions about application semantics. To address this, several researchers have explored concurrency control techniques for objects or abstract data types [33, 39, 47, 51, 54, 60]. Recently, Conflict-Free Replicated Data Types (CRDTs) have been proposed as a way to encode a common class of useful semantic properties: data values that change in an associative, commutative, and idempotent fashion, which guarantees that replicas of such values will eventually converge [55]. Encoding additional semantic knowledge is clearly useful: the significant latency, throughput, and availability improvements delivered by these efforts supports our thesis that opaque I/O-level consistency is not a complete solution. The object-based approach also addresses some of our concerns about the poor reusability of application-level consistency, since common data types like counters, lists, and graphs can be implemented once and then shared by multiple applications. Returning to the case study, we could use a Set CRDT to encode the adjacency list of each vertex in the graph—this would allow concurrent writes to the adjacency list to be safely reordered at a substantially lower cost than, say, using a linearizable test-and-set operation to update each adjacency list.

In general, such strategies offer only limited guarantees for developers. While object-level approaches can encode semantic knowledge about individual objects or values, system-wide semantics about the *composition* of

objects cannot be represented. Object-level consistency often focuses on achieving storage-level properties like replica convergence [55], leaving the developer responsible for mapping from high-level application properties to invariants over individual objects. For example, a guarantee that all replicas will converge to the same graph state is not strong enough to ensure that the garbage collection application presented in Section 2—which requires global knowledge of the refers-to graph—is correctly synchronized. While it might be possible to represent the entire application as a monolithic convergent object, this approach leads to complex objects that are difficult to reason about and reuse. In the limit, this requires an entire application to be a single convergent object: this reduces to the previously discussed extreme approach of application-specific consistency logic (Section 3.2).

A similar line of research in the database community has explored using semantic knowledge to improve concurrency control and query processing (e.g., [8, 10, 11, 30, 32, 42, 49]). Like the above work on object-level consistency, these proposals pursue an incremental approach in which semantic knowledge must be added to individual storage APIs and manually annotated by application developers. In general, these techniques complicate the interface between applications and storage systems and do not provide mechanisms to allow individual semantic properties to be composed to achieve application-level correctness invariants. Semantics-based concurrency control has seen little adoption in practice.

In short, object-level approaches present a middle ground suitable for programmers who are willing and able to reason about distributed consistency at a more application-specific level than is possible with I/O-level consistency. When applicable, convergent distributed objects allow programmers to focus their attention on cross-object consistency concerns. Outside of academia this approach has garnered some interest in certain NoSQL developer communities in recent years [19].

5 Flow-Level Consistency

For those who wish to raise the abstraction above single object consistency, a natural next step is to consider the semantics associated with data as it transits through application modules, across process boundaries, and between services. For example, a request for a user's timeline on a social network interacts with multiple services that aggregate related data, transform that data, and eventually render a response. To achieve correct behavior, application developers need to reason about the correctness properties that hold over the output of this cross-object, cross-service dataflow.

Reasoning about the consistency properties of applications composed from a variety of services requires reasoning both about the semantic properties of components and how these properties are preserved across compositions with other components. Hence it requires a model that captures both component semantics and the dependencies between interacting components. One approach is to view the distributed system as an *asynchronous dataflow*, in which streams of inputs pass through a graph of components that filter, transform, and combine them into streams of outputs. Component properties can be captured with a language of *annotations* that allow programmers to assert semantic properties; given annotations for individual components, determining whole-stack semantic properties for the entire dataflow graph becomes an inference problem.

Order insensitivity is a particularly important semantic property. Certain components are insensitive to message delivery order—they produce a unique output set for all orderings and batchings of their inputs. We call such components *confluent* [48]. For example, the deadlock detector presented in Section 2 is confluent and can safely process waits-for edges as soon as they become available. It is instructive to compare confluence with the goal of CRDT-level replica convergence (Section 4). Convergence applies to individual objects, while confluence is a property of dataflow components, and—by composition—of larger dataflow graphs. Compositions of confluent components simplify reasoning about higher-level application-level correctness properties, allowing developers to ignore asynchronous network behavior and concurrency across potentially complex services.

Dataflow consistency is not a well-studied topic. Our own ideas are fairly recent, embodied in a tool called BLAZES that helps programmers assess and enforce flow-level consistency properties like order insensitivity [3]. Based on component annotations provided by the programmer, Blazes determines if consistent outcomes are guaranteed without any coordination. When components are not confluent, Blazes synthesizes additional synchronization logic to ensure unique outputs. When possible, it exploits application-specific strategies based on data *sealing* to avoid the latency and availability costs of global coordination. For example, the garbage collector is not confluent: it can incorrectly conclude that memory is unreferenced if it observes only a subset of the global graph. However, in order to make progress, it does not require a total order over the refers-to records, but instead requires only an indication that the graph (or a particular partition of the graph) is complete ("sealed"), which can be achieved via producer-consumer punctuations [58] in the dataflow.

Given the case study applications, a developer would annotate the graph store and deadlock detection components as confluent. Blazes recognizes that the compo-

sition of a confluent replicated graph store and a confluent deadlock detector yields a confluent composite dataflow, and allows the system to execute without synchronization. The garbage collection component would be annotated as non-confluent but—as is common practice [41]—partitioned into generations or "epochs." If the dataflow is enhanced to produce sealing punctuations that indicate when individual allocators will produce no more edges within a given epoch, Blazes can synthesize a simple, barrier-based coordination strategy that prevents the garbage collector from executing until the graph partition is sealed—that is, the mark phase has ended for a given epoch. This strategy is much less expensive than a general coordination protocol: rather than waiting for coordination on every message, only a single coordination round is required per epoch.

The principal drawback of the dataflow approach is the need for manual component annotations: annotating modules can be burdensome and error-prone, especially for complex components. Incorrect annotations corrupt the analysis and can result in unsafe optimizations. For reusable modules (like the CRDTs discussed in Section 4), it may be possible to have an expert supply annotations. This amortizes the cost of annotation and reduces the risk of errors, but is only applicable for commonly used components. This drawback aside, flow-level approaches to consistency occupy an interesting middle ground: they are more broadly applicable than language- or application-level approaches, and more powerful than object-level approaches, which cannot capture composition across services.

6 Language-Level Consistency

Flow-level consistency only requires an abstract dataflow graph depicting the system architecture, and hence can be used with existing programs and off-the-shelf stream processors such as Storm [43]. However, it also requires that users manually add semantic annotations, which is burdensome and error-prone. These concerns are exacerbated as the complexity of the system increases. In this section, we consider a more radical approach: if the entire system is written in a high-level language that directly encodes both dependencies and appropriate semantic properties, the compiler can automatically analyze the consistency properties of entire applications.

6.1 Dependency Analysis

Database systems are a prominent example of the power of automatic dependency analysis. Because all data has a uniform representation (relations) and declarative rules are used to compute derived data (e.g., views), the system can easily observe how base data is used to compute derived data. This allows powerful capabilities like automatic materialized view maintenance [35], constraint inference [17, 46], and provenance analysis [22].

To enable similarly powerful lineage analysis for large-scale distributed systems, several technical challenges must be addressed. First, we need a uniform representation for all system state, including process-local knowledge, system events like timers and interrupts, and network messages. Second, we need a notion of dependencies that accounts for both synchronous, process-local dependencies (local computation) and asynchronous, cross-process dependencies (communication). We call the combination of these ideas *data-centric programming* [2]: all system state is represented in a uniform manner (as relations), which enables the system logic to be written as declarative queries over that state. An extended language that admits asynchronous queries can capture communication within the same declarative framework [5]. The most recent data-centric language designed by our group is called *Bloom* [4, 16].

6.2 Semantics

Dependency analysis reveals how inputs, outputs, and intermediate state are related; in addition, we need knowledge of *semantics*—that is, how these data values change over time and which invariants are preserved. Semantic properties and coordination requirements are closely related: if a program's semantics allow a situation in which a correctness invariant might be violated, then we might use a coordination protocol to prevent such a scenario from arising.

An important semantic property is *monotonicity*: intuitively, a monotonic operator is one that processes its inputs in an order-insensitive manner and never retracts a previous output in the face of new information. Typical examples of monotonic operators include set union, join, projection, and selection [4], as well as CRDT-like lattices with algebraic composition [24]. The *CALM Theorem* states that, if a program can be expressed entirely using monotonic logic, it is guaranteed to be confluent—that is, deterministic—despite the effects of network nondeterminism [6, 38]. Hence, monotonic operations form a "safe" vocabulary for distributed programming: because the program's output is a deterministic function of its input, it is much easier to check that correctness invariants are preserved.

For data-centric languages such as Bloom, there is a simple conservative test to determine the monotonicity of individual rules or entire programs—essentially, monotonicity is part of the language's type system [4]. Because monotonicity implies confluence, this test can identify a program's consistency requirements. For example, con-

sider the two programs from Section 2; in first-order logic, the semantic difference between the programs surfaces immediately. The deadlock detector enumerates the set of processes $\{t|\text{waits-for}^*(t,t)\}$. The garbage collector instead checks for *non-existence* in the set of references:

$$\{o|\neg\exists(r \in \text{refers-to}^*(Root,r), r = o)\}$$

Since negation is non-monotonic, static analysis can flag the garbage collection program as possibly requiring (partial) order guarantees to ensure deterministic results.

The language-centric approach to whole-program consistency refines rather than replaces the flow-centric strategy: the programmer is freed from the burden of choosing the correct annotations and of understanding intra- and inter-module dependencies.

7 Challenges and Opportunities

Providing flexible tools for managing distributed consistency throughout the software stack will require solving several challenges, both technical and sociological. In this section, we outline several areas for future investigation, with more questions than answers.

Software architecture and adoption. In this paper, we have examined how consistency can be achieved at various places between application logic and storage infrastructure. A general trend can be observed: as consistency mechanisms move closer to the application, they lose generality but gain efficiency. Which points in this spectrum will be the most compelling, and how can they be realized in concrete software architectures? I/O-level consistency mechanisms can be delivered and deployed as prepackaged services (e.g., a coordination service such as Zookeeper or a linearizable data store). In contrast, ad hoc application-level consistency schemes are closely integrated with program logic. Neither approach seems like an attractive way to deploy reusable consistency mechanisms that are placed throughout the software stack.

Our approach with Bloom has been to develop a new language that is well-suited to consistency analysis. While this "revolutionary" strategy may bear fruit in the long term, the need to adopt new programming models can inhibit short-term adoption. Is a brand-new language truly necessary, or can a variant of consistency analysis be adapted for mainstream languages, perhaps with the help of annotations or domain-specific constructs? In contrast to Bloom, CRDTs are more limited in scope but have proven easier to integrate into existing systems, as illustrated by recent work on productizing CRDT support in the Riak key-value store [19]. The dataflow-annotation approach we have pursued with Blazes can be seen as a middle ground. Which of these avenues (if any) will see widespread adoption remains unclear.

Program analysis and language design. One way to help programmers enforce consistency invariants is to provide them with domain-specific tools for distributed program analysis. New languages are one example we have explored; limited annotations are another. There is plenty of room for further work in that vein, and other approaches deserve exploration as well. Ideally, for example, one can imagine new programming language tools that extract properties like monotonicity guarantees from legacy programs and imperative languages. Short of that, perhaps familiar but restricted subsets of traditional languages could be amenable to analysis and more easily popularized (e.g., Java with immutable or fully-versioned objects). There may also be ways to exploit the recent progress in software synthesis, model checkers and theorem provers. The field here seems wide open.

Goldilocks and the many I/O consistency models. Many distributed consistency models have been proposed at the storage level, ranging from weak models like eventual consistency to strong guarantees like linearizability. Recently, a plethora of interesting new models have been proposed [7, 25, 44, 45, 57], but an important question remains unaddressed: how do the developers of an application choose the right consistency model? If they forego linearizability, they must consider that weaker consistency models allow more undesirable phenomena (e.g., causality violations or lost updates). Most applications can tolerate some of these phenomena but not others; conversely, the cost of preventing different phenomena at the I/O layer varies widely [9]. For a given application, when is the cost of preventing a particular phenomenon warranted? And how does the developer know that the resulting application is correct?

As we have argued above, I/O-level consistency guarantees are specified in terms of properties of read-write traces, so it is difficult to understand which low-level consistency guarantees are both sufficient and necessary to maintain application-level invariants. For example, a shopping cart application might want to guarantee that all of a user's purchases have been reflected in the cart before her "checkout" operation is processed. This can be achieved using causal consistency, by having the purchases "happen-before" the checkout (the purchases themselves need no causal dependencies). Instead of forcing programmers to understand the intricacies of the many available I/O-level consistency models, can we build tools to automatically map high-level application invariants down to low-level consistency mechanisms? For example, the CALM Theorem allows Blazes to prove that certain programs will be consistent over an eventually consistent I/O layer. Can this kind of analysis be adapted to a larger class of programs that exploit richer forms of I/O consistency? The related problem of determin-

ing whether a given application can produce serializable outcomes when run at a lower isolation level has been studied in the database literature [31].

Beyond determinism. Work on eventual consistency often tries to guarantee deterministic behavior. For example, confluence analysis identifies program fragments that produce deterministic outcomes despite non-deterministic network behavior. Similarly, CRDTs ensure that all replicas of an object converge to the same state, regardless of duplicated or reordered messages. However, determinism is too strong for some common application-level invariants. Consider the simple invariant: "A purchase request returns a confirmation if inventory is non-zero; otherwise it returns failure." This is non-deterministic—the set of successful purchases depends on the order in which messages are delivered and processed.

What is the best way to reason about non-deterministic but well-defined correctness criteria? One strategy is to simply encode the space of acceptable outcomes as a disjunction (e.g., "Purchase *X* succeeds and *Y* fails OR purchase *X* fails and *Y* succeeds"). A confluent system that satisfies this disjunction ensures that an acceptable outcome is always produced. However, enumerating the space of acceptable outcomes scales poorly as application complexity grows. Is there a more natural model than this enumerated choice of outcomes, and, if so, can we build program analysis tools to support it? More fundamentally, beyond monotonicity, are there design patterns that assist in achieving such "controlled non-determinism," and can such patterns be codified into theorems, analysis techniques, and language constructs?

8 Conclusion

The development of reliable distributed applications depends upon programmers' ability to reason about consistency. By narrowly focusing on I/O-level consistency, traditional research in this area risks increasing irrelevance: as the latency and availability costs of traditional consistency protocols have become prohibitive at scale, developers have begun to avoid consistency mechanisms entirely, instead relying on ad hoc, application-specific rules for conflict resolution and reconciliation. We believe that the solution is to meet application developers on their home turf: to explore a variety of consistency mechanisms, analysis tools, and programming constructs that operate at different layers of the software stack. The goal should be to help programmers judiciously employ consistency of the appropriate strength and to reason about consistency wherever it is most natural. The core tension lies in balancing expressivity and efficiency with generality and modularity. We have sketched examples and insights from our experience straddling these boundaries, but we suspect that further progress will require the research community to reconsider long-held assumptions about software architecture and the division between storage and application logic.

Acknowledgments

We would like to thank Emily Andrews, Alex Rasmussen, and the anonymous reviewers for their helpful feedback on this paper, and particularly our shepherd, Phil Bernstein. This work was supported by the Air Force Office of Scientific Research (grant FA95500810352), DARPA XData Award FA8750-12-2-0331, the Natural Sciences and Engineering Research Council of Canada, the National Science Foundation (grants CNS-0722077, IIS-0713661, and IIS-0803690), NSF CISE Expeditions award CCF-1139158, the National Science Foundation Graduate Research Fellowship (grant DGE-1106400), and gifts from Amazon, Cisco, Clearstory Data, Cloudera, EMC, Ericsson, Facebook, FitWave, General Electric, Google, Hortonworks, Intel, Microsoft, NetApp, NTT, Oracle, SAP, Samsung, Splunk, VMware, and Yahoo!.

References

[1] S. E. Abdullahi and G. A. Ringwood. Garbage collecting the Internet: a survey of distributed garbage collection. *ACM Computing Surveys*, 30(3):330–373, 1998.

[2] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. BOOM Analytics: Exploring data-centric, declarative programming for the cloud. In *EuroSys*, 2010.

[3] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Blazes: coordination analysis for distributed programs. `http://arxiv.org/abs/1309.3324`, 2013. In submission.

[4] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR*, 2011.

[5] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in time and space. In O. de Moor, G. Gottlob, T. Furche, and A. Sellers, editors, *Datalog Reloaded*, volume 6702 of *Lecture Notes in Computer Science*, pages 262–281. Springer Berlin / Heidelberg, 2011.

[6] T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. In *PODS*, 2011.

[7] M. S. Ardekani, P. Sutra, and M. Shapiro. Non-monotonic snapshot isolation: scalable and strong consistency for geo-replicated transactional systems. In *SRDS*, 2013.

[8] B. R. Badrinath and K. Ramamrithan. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, 1992.

[9] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. To appear in *VLDB*, 2014.

[10] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution. In *SoCC*, 2012.

[11] D. Barbará-Millá and H. Garcia-Molina. The demarcation protocol: a technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3(3):325–353, 1994.

[12] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[13] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[14] K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.

[15] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP*, 1987.

[16] Bloom programming language. `http://www.bloom-lang.org`.

[17] A. Brodsky and Y. Sagiv. Inference of monotonicity constraints in Datalog programs. In *PODS*, 1989.

[18] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. Tao: Facebook's distributed data store for the social graph. In *USENIX ATC*, 2013.

[19] R. Brown and S. Cribbs. Data structures in Riak. RICON 2012 (`https://speakerdeck.com/basho/data-structures-in-riak`).

[20] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[21] K. M. Chandy and J. Misra. A distributed algorithm for detecting resource deadlocks in distributed systems. In *PODC*, 1982.

[22] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[23] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *SOSP*, 1993.

[24] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *SoCC*, 2012.

[25] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, 2008.

[26] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[27] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in Partitioned Networks. *ACM Computing Surveys*, 17(3):341–370, 1985.

[28] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[29] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[30] A. A. Farrag and M. T. Özsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, 1989.

[31] A. Fekete. Allocating isolation levels to transactions. In *PODS*, 2005.

[32] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, 1983.

[33] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy. Comet: an active distributed key-value store. In *OSDI*, 2010.

[34] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[35] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.

[36] P. Helland and D. Campbell. Building on quicksand. In *CIDR*, 2009.

[37] P. Helland and D. Haderle. Engagements: Building eventually ACiD business transactions. In *CIDR*, 2013.

[38] J. M. Hellerstein. The Declarative Imperative: Experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.

[39] M. P. Herlihy. Optimistic concurrency control for abstract data types. In *PODC*, 1986.

[40] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[41] L. Huelsbergen and P. Winterbottom. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. In *International Symposium on Memory Management*, 1998.

[42] A. Kumar and M. Stonebraker. Semantics based transaction management techniques for replicated data. In *SIGMOD*, 1988.

[43] J. Leibiusky, G. Eisbruch, and D. Simonassi. *Getting Started with Storm - Continuous Streaming Computation with Twitter's Cluster Technology*. O'Reilly, 2012.

[44] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *SOSP*, 2011.

[45] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, 2013.

[46] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing implications of data dependencies. *ACM Transactions on Database Systems*, 4(4):455–469, 1979.

[47] C. Malta and J. Martinez. A framework for designing concurrent and recoverable abstract data types based on commutativity. In *International Symposium on Computer and Information Sciences*, 1991.

[48] W. R. Marczak, P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Confluence analysis for distributed programs: a model-theoretic approach. In *Datalog 2.0*, 2012.

[49] P. E. O'Neil. The Escrow transactional method. *ACM Transactions on Database Systems*, 11(4):405–430, 1986.

[50] C. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[51] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011.

[52] A. Schiper and A. Sandoz. Strong stable properties in distributed systems. *Distributed Computing*, 8(2):93–103, 1994.

[53] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[54] P. M. Schwarz and A. Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, 1984.

[55] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, 2011.

[56] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed SQL database that scales. In *VLDB*, 2013.

[57] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[58] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.

[59] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[60] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, 1988.