Programming Technologies for Highly-Scalable Data-Intensive Applications

Gowtham Kaki
Purdue University gkaki@purdue.edu

Database is the Achilles heel

On March 2nd, 2014, FlexCoin, a Bitcoin exchange, was attacked by hackers who stripped it of Bitcoins worth $620,000 [4, 3]. The loss forced the exchange to close, sending tremors down the cryptocurrency community, and raising doubts about the basic trustworthiness of blockchain applications. While the FlexCoin attack is notable for its scale and impact, underlying exploit is nonetheless simple, almost embarrassingly so! The attack exploited a commonplace concurrency anomaly, admitted by many database systems, that allows concurrent non-serializable transactions to execute, and successfully commit, against the same database state. As the FlexCoin developers explain in their postmortem report [5]:

> The attacker . . . successfully exploited a flaw in the code which allows transfers between flexcoin users. By sending thousands of simultaneous requests, the attacker was able to "move" coins from one user account to another until the sending account was overdrawn, before balances were updated. This was then repeated through multiple accounts, snowballing the amount, until the attacker withdrew the coins.

The same vulnerability was also exploited successfully against Poloniex, another Bitcoin exchange, which lost 12.3% of its Bitcoin reserves to hackers [6]. While Bitcoin exchanges are worst hit by such attacks, they are however not exclusive targets; attacks that exploit concurrency-related vulnerabilities in database-backed applications have also been used to steal gift cards, illegally reuse promotional codes, and pay for fewer items at checkout on eCommerce sites [9, 7, 8]. In fact, a recent study conducted by database researchers at Stanford estimates that such vulnerabilities lie latent in over 2M eCommerce websites, waiting to be exploited by a determined adversary [17]. The large scale presence of such shallow security vulnerabilities across diverse range of applications, which were developed independently over a long period of time, might puzzle anyone searching for a reason. Surely, these applications cannot all be ill-engineered, and in exactly the same way!

It is interesting to note that root cause of the above problem is neither ill-engineered applications, nor ill-conceived database systems, but a fundamental mismatch between what applications expect (or, have been told to expect) from a database, and what databases can realistically meet. Application developers usually build their applications around the ACID model of the database, where transactions are serializable, and each transaction is perfectly isolated from other concurrent transactions. Database systems, in reality, are not ACID. Not all databases offer serializable transactions, and even if they do, serializability is almost never the default setting for the *isolation level* [12, 1]. This is because serializable transactions are exorbitantly expensive, hence do not scale.

Considering that scalability is a cornerstone principle for modern applications, database systems eschew serializability in favor of weaker, but scalable, transactions that admit concurrency anomalies, such as the one underlying the aforementioned vulnerability. However, not all concurrency anomalies manifest as security threats, and when they don't, it is beneficial to admit them, for it lets the application scale using non-serializable transactions. Knowing if a database-induced concurrency anomaly is harmful or innocuous is therefore a problem facing every application developer for which she gets little or no support from her favorite programming language or compiler. Left to their own devices, developers stick to the default configuration of the database (as FlexCoin and Poloniex developers understandably did), thereby inadvertently facilitating security exploits.

"A programmer had a problem. She used a distributed system . . . "

Note that the applications I mentioned so far are nothing out of the ordinary; they are conventional database-backed applications serving typical workloads relying on an off-the-shelf relational database for their storage needs. The problem of dealing with concurrency-related vulnerabilities is even harder in the context of planet-scale web applications that need to be highly-scalable and available, i.e., *always-on*. These applications often rely on unconventional storage technologies, such as NoSQL databases, that exhibit intriguing behaviors which frequently confound even distributed systems researchers [2]. Indeed, even experts find it hard to guarantee the correctness of applications built on weakly consistent NoSQL stores. For instance, consider `py-tpcc`, a python implementation of the TPC-C benchmark for Cassandra NoSQL store, built by systems researchers at CMU [11]. TPC-C, an Online Transaction Processing (OLTP) benchmark designed in 1995, consists of a simple warehouse application with 5 transactions (2 of which are read-only) operating over 9 tables. The application has 12 invariants specifying its correctness, which every correct implementation is required to preserve. The `py-tpcc` implementation of TPC-C on Cassandra has however violated 9 of these 12 invariants, thereby causing data corruption, during our experiments involving 50 concurrent clients on a standard laptop[1]. It should be noted that this is not an indictment of the engineering effort behind `py-tpcc`, but rather an indictment of programming models (or the lack thereof) for weakly-consistent systems. **The lack of suitable programming models and a non-existent tool support has made it hard to guarantee the correctness of even simplest of distributed applications**. Building non-trivial distributed applications, such as collaborative editors and drawing canvases, distributed job queues, blockchains etc., is even harder, considering that each needs its own network layer implementing an application-specific message semantics. **The high complexity of these applications means that it is almost impossible for developers to have a tractable mental model to reason about their correctness, and obtain any measure of confidence in their ability to handle safety-critical systems, such as decentralized (serverless) banks.**

My research agenda

My research has a single-point agenda of simplifying the task of building secure and highly-scalable data-intensive applications. Scalability, being a fundamental requirement of modern applications, inevitably leads to hard choices in system design, leaving application developers to deal

[1] The experiments were done by my colleague, Kia Rahmani, who describes them in detail on his blog [10]

with the fallout of additional complexity and the increased risk of latent security vulnerabilities. While it is not possible to escape these tradeoffs, I believe that with well-designed programming models the complexity can be contained, and with intelligent tool support, the risk can be mitigated. Towards this end, I developed a number of theoretical techniques and practical tools that bring to bear advances in Programming Languages and Formal Methods research on database programs to help developers build, and automatically verify, scalable data-intensive applications.

ACIDifier: *taming weak isolation on relational databases*

In my most recent work[2], my collaborators and I have shown that weakly-isolated transactions (i.e., non-serializable transactions that admit concurrency anomalies) can be placed on the same formal footing as serializable transactions. In concrete terms, this means that a developer does not have to understand concurrency anomalies induced by weakly-isolated transactions in order to determine whether her application runs as intended under weak isolation. Instead she can reason about various levels of weak isolation *declaratively*, i.e., with a definition rather than an implementation, just as she does with serializability. I have exploited this observation to build a tool called ACIDifier that can automatically analyze database applications with weakly-isolated transactions, detecting vulnerabilities caused by weak isolation, if any, and certifying the application correct otherwise. **Using ACIDifier, a developer with absolutely no knowledge of weak isolation can determine, for example, that TPC-C warehouse application admits vulnerabilities under certain configurations of weak isolation possible on PostgreSQL and MySQL**. Clearly, such configurations need to be ruled out. Among the remaining configurations, ACIDifier picks the weakest (thus, most scalable) configuration of the isolation levels at which each of TPC-C's 5 transactions can be run without violating its invariants. **Thus, ACIDifier helps the developer pre-empt any isolation-related vulnerabilities without having to understand what weak isolation even means, and without having to write test cases, or even run a single line of the application code.**

While the theory behind ACIDifier is language-agnostic, the tool itself is tied to the OCaml language, and has so far analyzed benchmarks and sample applications written in OCaml. Going forward, I plan to use the ACIDifier reasoning engine to analyze the existing corpus of database applications written in popular language frameworks, for e.g., Ruby-on-Rails. A 2015 study has shown that a large corpus of such applications admit the possibility of isolation-related vulnerabilities [13], but there currently exists no (automatic) way to know how exactly a vulnerability manifests in an application, and what must be done to prevent it. ACIDifier can help these applications if only the language barrier can be addressed. Towards this end, I developed a prototype Rails-to-OCaml translator called MagLev[3] that repurposes symbolic execution technique, which is otherwise used for program verification, to translate Ruby-on-Rails code to OCaml code. I plan to develop my prototype into a full-fledged Rails-to-OCaml compiler in the near future.

[2] Accepted for publication in the Proceedings of the ACM on Programming Languages (PACMPL'18 Journal), and for presentation at the 45^{th} ACM SIGPLAN Principles of Programming Languages (POPL'18) conference. Preprint available at https://arxiv.org/pdf/1710.09844.pdf.

[3] Extended abstract on MagLev has been presented at the poster session at the 37^{th} ACM SIGPLAN Programming Languages Design and Implementation (PLDI'16) Conference. The draft is available at http://purduepl.github.io/docs/maglev.pdf.

Quelea *and Q6: Taming weak consistency on NoSQL databases*

In another line of work, my collaborators and I showed that scalable data-intensive applications that use weakly-consistent NoSQL stores can be built in a high-level declarative language (a Haskell DSL that we call Quelea) without losing scalability[4]. We show that doing so entails certain benefits, including high-level reasoning about consistency that is less error-prone, when compared to low-level reasoning in terms of store's implementation details. For instance, it is easy to understand what "Read-Your-Writes" consistency means to your application, than understanding how your application behaves under "R=2; W=1" consistency setting on Cassandra. Nonetheless, it is still too much to ask an average developer to manually reason about the correctness of her application under "Read-Your-Writes" consistency. Recognizing this, I built a tool called Q6 that analyzes applications built using our Haskell DSL, and computes the weakest consistency configuration at which it is safe to run the application[5]. Q6 is fully automatic, requiring no manual intervention at any stage. **Thus, a developer with absolutely no knowledge of weak consistency can rely on Q6 to determine, for example, that her Microblogging application requires "Read-Your-Writes" consistency to prevent data corruption**. Q6 is already capable of inferring consistency settings for such diverse applications as a banking service, an Amazon-style shopping cart application, and an eBay-like auction site. The ability to verify the correctness of a weakly-consistent NoSQL application with respect to its strongly consistent SQL counterpart, makes Q6 an invaluable tool in the process of porting a SQL application to NoSQL. Going forward, I plan to build on Q6 to (partially) automate the translation process from SQL to NoSQL.

Mergeable Types: Rethinking the distributed programming model

In my third and last line of work, my collaborators and I developed a programming model backed by a sound theory to reduce the complexity of building fully-decentralized distributed applications, e.g., a serverless bank. Our programming model, called Mergeable Types, is inspired by the Git version control system that allows collaborators to work in a fully-decentralized fashion as long as one can always merge one's changes with others[6]. Observing that seamless mergeability enables seamless decentralization in Git, we extend the analogy to distributed applications and their data, i.e., using Git for the data rather than code to enable decentralization. Mergeable Types is a principled attempt at building a distributed programming model around this observation. The result is a programming language, where any type for which a (three-way) merge is defined (i.e., a mergeable type) automatically becomes a distributed data type with no additional effort. **For a distributed application built using mergeable types, programmer does not have to implement a messaging layer to propagate concurrent updates to the shared state; it is all taken care of by the language runtime, and the application automatically scales**. This lets developers focus on just the application semantics, allowing them to build fully-decentralized scalable distributed applications in just few lines of code. For instance, we took an OCaml implementation of a drawing application, and turned it into a collaborative drawing application just by adding 17 lines of code in Mergeable

[4]Published in the proceedings of the 36th ACM SIGPLAN Programming Languages Design and Implementation (PLDI'15) conference [16]. Project website: http://gowthamk.github.io/Quelea/.

[5]Q6 sourcecode is available at https://github.com/gowthamk/q6. The corresponding paper will soon be published.

[6]An extended abstract on Mergeable Types was presented at the ML Workshop 2017, co-located with the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP'17). The draft is available at http://www.mlworkshop.org/icfp-mlworkshop17-final4.pdf

Types model[7]! Going forward, my plan is to implement safety-critical applications like blockchain in the Mergeable Types model, and understand whether or not it is possible to automatically certify their safety guarantees.

CATALYST: *Extending the state-of-the-art in automatic reasoning*

At the heart of each of my various lines of work is the problem of automatically analyzing rich semantic properties of data-intensive programs. The tools I built in each case (e.g., ACIDIFIER and Q6) employ conventional reasoning techniques developed during the decades of PL research, but the programs and semantic properties that I am interested in are sufficiently novel that they often call for new reasoning techniques to be developed. For instance, analyzing database programs written in OCaml requires tracking how sets of values stored in algebraic types, e.g., lists, flow between various functions, which could even be higher-order like `map` and `fold`[8]. For this purpose, I introduced a concept called *structural relations*, developed a dependent type theory around these relations, and showed that it is possible to precisely track values through higher-order control flow in functional programs using these relations[9]. I have also built a tool called CATALYST that automatically applies relational reasoning to verify rich semantic properties of Standard ML (SML) programs. CATALYST **has automatically verified the flow and shape-related properties of almost the entire standard library of the SML language**. In the same vein, my collaborators at Microsoft Research and I devised a refinement type and inference system for C#, called BROOM, to automatically check the safety of reason-based manual memory management in dataflow programs (such as MapReduce)[10]. Novel program logics and new reasoning techniques that push the envelope in automatic reasoning are interesting and useful in their own right, and I plan to continue working on them. I am particularly enthusiastic about applying data-driven machine learning techniques to program verification, thus extend its scope to those areas where the reach of conventional PL techniques is limited. Our lab already has a headstart in this direction [18, 19], and I plan to build on the foundation laid by my colleagues.

Conclusion

To conclude, building highly-scalable data-intensive programs is hard. The lack of suitable programming models and easy-to-use reasoning techniques for data-intensive programs has made it a complicated and risky endeavor to build scalable safety-critical applications. Developers frequently get it wrong, resulting in consequences ranging between poor user experience to heavy economic losses. This is bound to happen as long as developers are asked to build 21st century applications using 20th century programming models and tools. The sole focus of my research, in the recent past, and in the near future, is to effectively bridge this gap by developing tools and techniques to reason about highly-scalable data-intensive programs.

[7] Sourcecode available at https://github.com/gowthamk/DaLi/tree/master/experiments/canvas

[8] Higher-order functions are functions that accept other functions as arguments; see http://www.cse.unsw.edu.au/~en1000/haskell/hof.html for a quick overview.

[9] Published in the proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP'14) [14].

[10] Published as a Microsoft Technical Report [15].

References

[1] When is "ACID" ACID? Rarely., 2013. URL http://www.bailis.org/blog/when-is-acid-acid-rarely/.

[2] Jepsen:Cassandra, 2013. URL https://aphyr.com/posts/294-call-me-maybe-cassandra.

[3] Bitcoin Bank Flexcoin Shuts Down After $620,000 Heist, 2014. URL https://gizmodo.com/bitcoin-bank-flexcoin-shuts-down-after-620-000-heist-1535960407.

[4] Bitcoin bank Flexcoin closes after hack attack, 2014. URL https://www.theguardian.com/technology/2014/mar/04/bitcoin-bank-flexcoin-closes-after-hack-attack.

[5] Flexcoin is shutting down. , 2014. URL https://web.archive.org/web/20160408190656/http://www.flexcoin.com/.

[6] BTC Stolen from Poloniex, 2014. URL https://bitcointalk.org/index.php?topic=499580.

[7] Race conditions exploit granted free money on web services. , 2015. URL https://hackaday.com/2015/04/29/race-conditions-exploit-granted-free-money-on-web-services/.

[8] Shopify Cart does not update Shipping Costs in Checkout when Quantities in Cart change, 2015. URL https://ecommerce.shopify.com/c/ecommerce-discussion/t/shopify-cart-does-not-update-shipping-costs-in-checkout-when-quantities-in-cart-change-294832.

[9] Gift Card Fraud: How It's Committed and Why It's So Lucrative, 2015. URL https://www.tripwire.com/state-of-security/risk-based-security-for-executives/risk-management/gift-card-fraud-how-its-committed-and-why-its-so-lucrative/.

[10] A Case Study: Python TPC-C Benchmark, 2017. URL https://kiarahmani.github.io/2017/07/28/NoSQL3/.

[11] Python TPC-C, 2017. URL https://github.com/apavlo/py-tpcc.

[12] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. HAT, Not CAP: Towards Highly Available Transactions. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 24–24, Berkeley, CA, USA, 2013. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2490483.2490507.

[13] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1327–1342, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2737784. URL http://doi.acm.org/10.1145/2723372.2737784.

[14] Gowtham Kaki and Suresh Jagannathan. A Relational Framework for Higher-order Shape Analysis. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 311–324, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2873-9. doi: 10.1145/2628136.2628159. URL http://doi.acm.org/10.1145/2628136.2628159.

[15] Gowtham Kaki, G. Ramalingam, Kapil Vaswani, and Dimitrios Vytiniotis. Safe memory regions for big data processing. January 2016. URL https://www.microsoft.com/en-us/research/publication/safe-memory-regions-big-data-processing/.

[16] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 413–424, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737981. URL http://doi.acm.org/10.1145/2737924.2737981.

[17] Todd Warszawski and Peter Bailis. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 5–20, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4197-4. doi: 10.1145/3035918.3064037. URL http://doi.acm.org/10.1145/3035918.3064037.

[18] He Zhu, Aditya V. Nori, and Suresh Jagannathan. Learning refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 400–411, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3669-7. doi: 10.1145/2784731.2784766. URL http://doi.acm.org/10.1145/2784731.2784766.

[19] He Zhu, Gustavo Petri, and Suresh Jagannathan. Automatically learning shape specifications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 491–507, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908125. URL http://doi.acm.org/10.1145/2908080.2908125.