My Weak Consistency is Strong

When Bad Things Do Not Come in Threes

Zechao Shang*
The University of Chicago
zcshang@cs.uchicago.edu

Jeffrey Xu Yu
The Chinese University of Hong Kong
yu@se.cuhk.edu.hk

ABSTRACT

It is expensive to maintain strong data consistency during concurrent execution. However, weak consistency levels, which are considered harmful, have been widely applied in analytical jobs. Their success challenges our belief: data consistency, which is believed to be an essential to precise computing, does not always need to be preserved. In this paper, we tackle one of the core questions related to the application of weak consistency: When does weak consistency work well? We propose an effective explanation for the success of weak consistency. We name it bad things do not come in threes, or BN3. It is based on the observation that the volume of data is far larger than the number of workers. If all workers are operating concurrently, the probability that two workers access the same data at the same time is relatively low. Although it is not small enough to be neglected, the chance that three or more workers access the same data at the same time is even lower. Based on the BN3 conjecture, we analyze different consistency levels. We show that a weak consistency level in transaction processing is equivalent to snapshot isolation (SI) under reasonable assumptions. Although the BN3 is an oversimplification of real scenarios, it explains why weak consistency often achieves results that are accurate enough. It also serves as a quality promise for the future wide application of weak consistency in analytical tasks. We verify our results in experimental studies.

1. INTRODUCTION

Consistency[1] has been treated as a troublemaker for a long time. Maintaining data consistency needs subtle design and implementation. All of these delicate protocols are still time consuming. Moreover, consistency is involved in the "CAP war": it impedes availability or partition tolerance, which are usually considered much more important. Therefore more and more "NoSQL" data stores choose to dismiss data consistency completely or to enforce only weak data consistency. However, data consistency is in high demand in real world applications. When the lower layer does not provide mechanisms to ensure strong data consistency, application developers have to implement "feral" mechanisms, which usually contain mistakes [4, 55] and cause more problems. The "biggest mistake as an engineer" [1] of Jeff Dean is not putting transactions in BigTable.

It seems maintaining (strong) consistency becomes an expensive penalty that people must pay. However, *no consistency* (NoCon) or very weak consistency, which are considered harmful, have been widely applied in analytical systems. HogWild! [41] is the pioneer on using NoCon for optimization algorithms. The parameter server [32] enforces latency-bounded consistency and achieves success in a wider range of applications. Google's Tensorflow [2] and Microsoft's Project Adam [9] expand the use of the parameter server to deep learning. In graph analytics, asynchronous computing shows its potential to achieve a faster computing speed in certain tasks. Various asynchronous and semi-asynchronous graph analytical systems have been proposed, for example Grace [49], GiraphUC [20] and PowerSwitch [52]. Although these systems use different computing models, communication mechanisms and correctness guarantees, their success reveals one common discovery: data consistency, which is believed to be a critical property for precise computing, does not always need to be preserved.

The popularity of weak consistency systems brings both challenges and opportunities to database research. In this paper, we attempt to tackle one of the core questions related to the application of weak consistency in analytical tasks: *When does weak consistency work well?* Based on our weak consistency studies, we propose an effective explanation for the success of weak consistency. We name this explanation *bad things do not come in threes*, or in short, `BN3`. Usually, the volume of data is far larger than the number of workers. Therefore, if all workers are operating concurrently, the probability that two workers access the same data at the same time is relatively low. Although it is not small enough to neglect, the chance that three or more workers access the same data at the same time is even lower. Therefore, we propose `BN3` which assumes the bad things (concurrent operations) do not come with three or more workers involved. As shown in Section 3, the probability that the `BN3` holds (when $p = 10^{-4}$, c.f. Section 3) is more than 99.99% when a

*The work is mainly done when the author was at The Chinese University of Hong Kong.

[1]Unless otherwise stated, we refer to general data consistency in this paper, which may include consistency and isolation in ACID and single item consistency in replicated databases.

8th Biennial Conference on Innovative Data Systems Research (CIDR '17) January 8-11, 2017 , Chaminade, California, USA.

system has hundreds of workers, and more than 99.5% even when there are one thousand workers. Based on the BN3 conjecture, we analyze different consistency levels on top of analytical tasks. Under reasonable assumptions, we show that the weakest consistency level in transaction processing, such as NoCon, is equivalent to snapshot isolation (SI) if the BN3 is always true. SI is believed to be one of the strongest consistency levels.[2] Although the BN3 is an oversimplification of the real world scenarios, it explains why weak consistency often achieves results that are accurate enough. It also serves as a quality promise for the future wide application of weak consistency in analytical tasks.

The organization of the paper is as follows: In Section 2, we briefly introduce the computing model. In Section 3, we propose the BN3. We also analyze the chance it holds on large scale data in theoretical analysis and via empirical studies. In Section 4, we state our main points of discussion. We show our experimental results in Section 5. We discuss related work in Section 6. We propose future work in Section 7 and conclude the paper in Section 8.

2. BACKGROUND

We briefly introduce the background of this paper, including the computing model and system architecture in this section.

This paper focuses on main memory analytics. We assume the data is entirely held in memory and exclusively owned by the user. The user's main target is to perform analytical analysis. In other words, we assume the data is static and that the system aims to reduce computing time.

We assume the following, although our work does not rely on these assumptions. The system provides random access to the end user which is built on a shared memory architecture. In other words, the user utilizes two functions, `READ(x)` and `WRITE(x, v)`, to read and write data storage. The storage is shared by all concurrent workers (CPUs). The parameter server and shared memory multi-core server are representative environments that satisfy the assumptions. Recent systems based on RDMA usually provide random access too. The random access shared memory assumption simplifies our analysis and connects it closer to traditional database theory. Our analysis also applies to the message passing architecture.

Our study focuses on *general purpose* analytical tasks. However, we borrow the notations from graph analytics. We assume there is a graph $G = (V, E)$. V is the set of vertices. Each vertex could be a vertex in graph analytics, a variable in optimization and a user/item in machine learning. There are edges set $E \subseteq V \times V$. The end user implements a user-defined function (UDF) $f(\cdot)$. The UDF contains the main computing task. A UDF reads and writes data via the `READ` and `WRITE` APIs provided by the system. For example, one possible Page Rank implementation $f(v)$ is to read the Page Rank of all v's incoming neighbours, perform the computation according to a formula and write the new Page Rank to v. Another example is in matrix factorization problem, a vertex represents a column or a row and an edge represents a item in the matrix. The UDF on v reads the vectors in v' neighborhood vertices, computes the new vector for v and writes it back.

The system imposes no other restrictions on the end user. The user is able to treat the system as a key-value storage and write any program freely. However, for best practice the UDF often satisfies the following observations.

Observation 1. (Vertex-centric) Each UDF $f(v)$ is on top of a vertex v. The user treats UDF as an independent and indivisible unit. The effect of one UDF shall appear as a whole. There are no particular orders to execute between UDFs on different vertices.

The vertex-centric computing encode update operations in UDFs. Each UDF corresponds to one unit of a desired operation, for example a vertex status update in graph analytics. The computing is performed in a decentralized manner for better performance, thus each UDF acts alone and their relative order does not matter.

Observation 2. (Neighbour access pattern) Each UDF $f(v)$ accesses the vertex v and its neighbours only.

It is common that each datum only communicates with a specific subset of data. For example a vertex v's Page Rank value is determined by its neighbours' Page Rank values.

Observation 3. (Read-compute-write) Each UDF has three phases in order as follows: read, compute and write. No data are read more than once and no data are written more than once.

It is easy to see common analytics following this observation. When a UDF does not follow it, we can always rewrite the UDF without compromising the correctness with a user cache.

In recent years, shared-memory architectures have been widely applied in different domains. Not only can these systems perform graph analytics but they are also capable of supporting data mining, optimization, deep learning and general purpose computing. Machine learning, especially deep learning system [59, 11] usually provides random-access shared-memory interface without no consistency controller or with only weak consistency guarantee. Representative systems include Cyclades [39], Parameter Server [32], TensorFlow [2], Project Adam [9], Petuum [54] and the GPU-based GeePS [10]. There are also general purpose systems which support asynchronized communication with random-access interface, for example Malt [31], Aspire [48], Husky [56], Spark-ASIP [18], and HSync [46].

3. BAD THINGS DO NOT COME IN THREES

We introduce the BN3 formally in this section, and analyze how likely a large scale data satisfies this conjecture. We also consider how far the data deviates from it when this conjecture is not satisfied.

We first consider the temporal and spatial relationship between two UDFs. For UDF u and v, if all of u's operations are performed before of all v's operations, we denote this by $u \prec v$. If neither $u \prec v$ nor $v \prec u$ is true, we denote this by $u \parallel v$; otherwise, we say $u \nparallel v$. Moreover, if a vertex a exists where (i) both u and v access a and (ii) at least one of two operations are write, we say $u \cap v$; otherwise, we say $u \not\cap v$. We denote $u \sqcap v$ when the condition (1) is satisfied (both of

[2]As illustrated by [5], most of the commercial available databases offer default consistency levels equivalent to or lower than SI.

Notation	Explanation
$u \prec v$	u precedes v
$u \parallel v$	parallel (neither $u \prec v$ nor $v \prec u$)
$u \nparallel v$	not parallel ($u \prec v$ or $v \prec u$)
$u \cap v$	(operation) intersect
$u \not\cap v$	not $u \cap v$
$u \sqcap v$	(data) intersect
$u \not\sqcap v$	not $u \sqcap v$
$u \otimes v$	u $\parallel$ v and $u \cap v$ (bad event)
b_u	$\lvert\{v \lvert v \otimes u\}\rvert$ (bad degree)
p	collision rate (frequency of $\sqcap$)

Table 1: Notation Table

		$u \nparallel v$	$u \parallel v$
$u \not\sqcap v$	$u \not\cap v$	✓	✓
$u \sqcap v$	$u \not\cap v$	✓	✓
	$u \cap v$	✓	×

Table 2: The relationship of u and v: (1) a universal $u \nparallel v$ or $u \not\sqcap v$ eliminates the possibility of cycles; (2) even $u \otimes v$ ($u \parallel v$ and $u \cap v$) does not always lead to cycles; (3) we use static data-dependent frequency $u \sqcap v$ to estimate the severity of out-of-order executions.

them access a common data). $u \sqcap v$ is a necessary but not sufficient condition of $u \cap v$: an extreme example is when u and v are read-only; $u \sqcap v$ may be true but $u \cap v$ is always false. And we say $u \not\sqcap v$ when $u \sqcap v$ is false.

If for all pairs of UDFs u and v we have $u \nparallel v$ or $u \not\sqcap v$, then the execution is correct (i.e., equivalent to conflict serializability) even without any kind of coordination and consistency maintenance. We leave the proof in Appendix. In other words, the occurrence of $u \parallel v$ and $u \cap v$ is the root cause of UDF interference, which further causes data inconsistency. Neither $u \parallel v$ nor $u \cap v$ can be completely eliminated from the computing; therefore, we focus on the co-occurrences of them, and let $u \otimes v$ represent $u \parallel v$ and $u \cap v$. Informally, we call $u \otimes v$ a *bad event*. To measure the severity of bad events, for each UDF u we consider its *bad degree* $b_u = |\{v|u \otimes v\}|$. We omit the subscript u when there is no ambiguity. The ideal situation (embarrassingly parallel) is $b_u = 0$ for all of u, which guarantees weak consistency is as good as strong consistency. b_u serves as an upper bound of data inconsistency: $u \otimes v$ does not necessarily mean that an integrity constraint is compromised. Depending on its strength, a consistency controller may or may not help to ensure the consistency. Even for weak consistency, there is still a probability that executions are correct by chance.

The bad degree of UDFs depends on two factors: the static dataset and the dynamic runtime. To have an effective estimation of the common situations that the system usually meets, we assume that the UDFs are executed in no particular order. Therefore, we consider the concurrently executed UDFs independently. For the data-dependent factor, we denote the *collision rate* p by $\frac{|\{u \sqcap v\}|}{|\{u\}|^2}$. p represents the proportion of UDF pairs that are possible to develop into a $\cap$ relationship.

How many bad events: We measured the property p of large scale datasets and report the results in Table 3. The top four are graphs representing web graphs. The bottom two are social networks. The datasets *uk-2007-05*, *uk-2014*, *eu-2015* are from WebGraph[3] [7, 8]. The dataset *claw 2012* is from Web Data Commons[4] [36]. Dataset *wiseset* is from WISE 2012 Challenge[5] and *friendster* is from SNAP dataset [28]. This table illustrates the number of vertices, the number of edges, the average distances between all pair vertices and the collision rate p. In the table, we find p ranges from 7×10^{-5} to 6×10^{-4}. Larger graphs tend to have smaller collision rates.

Dataset	$\lvert V\rvert$ ($\times 10^6$)	$\lvert E\rvert$ ($\times 10^6$)	distance	p ($\times 10^{-4}$)
uk-2007-05	106	3,739	15.42	4.2
uk-2014	787	47,614	20.61	3.7
eu-2015	1,070	91,792	12.45	5.8
claw 2012	3,563	128,736	12.84	1.4
wiseset	59	265	5.01	4.0
friendster	66	1,806	5.78	0.7

Table 3: Real Datasets

With the help of the p, we are able to calculate the b_u for the UDF u. Assume there are C concurrent workers. For simplicity, we consider the case that all UDFs have the same number of operations, all UDFs proceed at the same pace and all workers start at the same time. Therefore, for a UDF u, $b_u = 0$ is when all other $C-1$ concurrent UDFs v satisfy $u \not\sqcap v$. Therefore, we have

$$\Pr(b = 0) \geq (1-p)^{(C-1)}$$

Similarly, we have

$$\Pr(b = 1) \geq p \times (C-1) \times (1-p)^{(C-2)}$$

As illustrated in real datasets measurement (c.f. Table 3), the collision rate p is usually a small value. The above two formulas show that $1 - \Pr(b = 0)$ is $\mathcal{O}(p)$, which is a small value. Moreover, $1 - \Pr(b \leq 1)$ is $\mathcal{O}(p^2)$, which is even smaller. Base on this discovery, we propose the following conjecture which suggests $b_u \leq 1$ is true everywhere or with high probability.

CONJECTURE 1. *We say **bad things do not come in threes**, or `BN3`, if $b_u \leq 1$ for all UDFs u. Informally, it means a maximum of two UDFs cause potential data inconsistencies. Or in other words, a "serial collision" does not exist.*

To reveal the `BN3` in real datasets in Table 3, we show the results in Figure 1 and Figure 2. In each figure, we vary the number of cores in the x-axis, and draw the probability of b for different p. Figure 1 illustrates the $\Pr(b \leq 1)$ and Figure 2 renders $\Pr(b = 0)$ and $\Pr(b = 1)$ separately. From the figures, we see $\Pr(b \leq 1)$ is close to 1. It is as large as 99.5% when $p = 10^{-4}$ and the system has 1,024 cores. On the other hand, $\Pr(b = 0)$ could be as low as 90% in the same configuration.

Remarks: The above analysis assumes the access pattern is uniform, e.g. each UDF has equal chance to be executed. It

[3]http://law.di.unimi.it/datasets.php
[4]http://webdatacommons.org/hyperlinkgraph/
[5]http://www.wise2012.cs.ucy.ac.cy/challenge.html

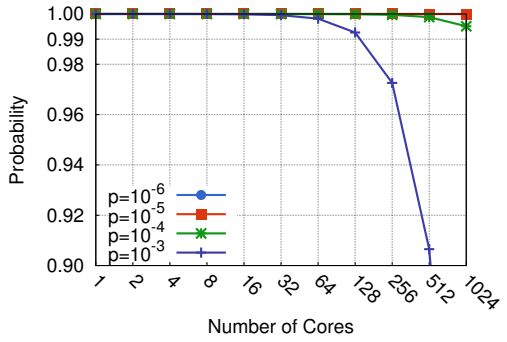

Figure 1: The $\Pr(b \leq 1)$ under different collision rate p varying the number of cores

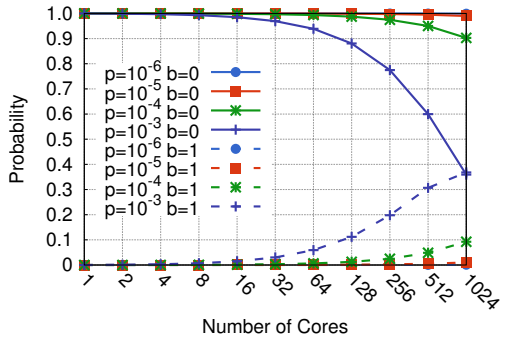

Figure 2: The $\Pr(b = 0)$ and $\Pr(b = 1)$ under different collision rate p varying the number of cores

relies on following assumptions. First, there is no centralized mutable components, for example a counter, a sketch or a priority scheduler. Second, the schedule of UDF does not rely on data-dependent properties, for example a lexicographic order of vertex (in a web-page graph), a BFS-based order of vertex, or a priority scheduling of UDFs.

However, a non-uniform access pattern does not necessarily indicate `BN3` is useless. A traditional way to monitor the severity of out-of-order execution is by monitoring the number and length of cycles in dependency graphs. Acyclic means conflict serializability and a smaller number of cycles could serve as a good indicator of better result quality. However, monitoring the number of cycles in a real-time manner is time consuming.[6] Instead we can just monitor the b_u for all vertex u or a sample of them. A straightforward solution is to record the last-read and last-write timestamp (and the reader/writer respectively) for all the vertices (or a sample of them). Therefore for each UDF u, we are able to know whether there is a concurrent UDF v such that $u \cap v$ by inspecting the timestamps. This only involves $\mathcal{O}(1)$ space and time per each operation, which is far more efficient than monitoring the cycles.

4. WEAK CONSISTENCY IS STRONG

In this section, we show that under given assumptions, NoCon actually produces results equivalent to snapshot isolation (SI). Although the `BN3` is an oversimplification of the real world, it is true for almost all vertices in a large dataset. Therefore, our results are close to real scenario. This helps to explain why weak consistency performs well on a large range of applications and datasets. Our results also serve as guidance for the future application of weak consistency.

We begin with some background knowledge of transaction processing. To be consistent with the transaction processing literature [51], each UDF is treated as a *transaction*, although we aim for the isolation of "ACID" properties. The well-recognized standard of transaction processing is serializability. An execution history of a set of transactions is (conflict) serializable if the execution of transactions is equivalent to a serial execution. The dependency graph is used [51] to verify whether a history is (conflict) serializable. Each transaction is represented as a vertex in the dependency graph. For two transactions, u and v, there is a directed edge $u \to v$ if a data item a exists such that (i) both u and v access a, (ii) at least one of the operation is a write operation and (iii) u accesses a earlier than v. An important theorem in verifying serializability is as follows.

THEOREM 1 ([51]). *An execution history is serializable if, and only if, the corresponding dependency graph is acyclic.*

Although serializability is important and useful, enforcing serializability is time consuming. One of the strong consistency (but weaker than serializability) is SI. It ensures all reads made in a transaction will see a consistent snapshot of the data. SI has been widely applied in both commercial databases and open source ones. Some commercial databases do not even provide a consistency level that is stronger than SI [5].

To distinguish between serializability and SI, the dependency edge $u \to v$ is categorized into three types depending on the type of operation of u and v.

- Write dependency: $u \xrightarrow{ww} v$ if v overwrites the u's written data.
- Read dependency: $u \xrightarrow{wr} v$ if v reads the data written by u.
- Anti-dependency: $u \xrightarrow{rw} v$ if v overwrites the data read by u.

The following theorem discovered by Fekete et al. [17] reveals the relationship between anti-dependency edges and SI.

THEOREM 2 ([17]). *An execution history is under SI if all cycles (if any) contain two adjacent $\xrightarrow{rw}$ edges.*

Before introducing our results, we introduce two additional assumptions that help us reach useful conclusions without loss of generality.

Assumption 1. (Item commutativity) For a data item x, if there are two write operations $w_1(x)$ and $w_2(x)$, $w_1(w_2(x)) = w_2(w_1(x))$.

Started by MapReduce [12], large scale processing systems support and utilize commutativity. Graph analytical systems, for example PowerGraph [19], GiraphUC [20] and

[6]The current best algorithm (Johnson's algorithm [24]) takes $\mathcal{O}(nec)$ time to list all cycles. n, e, c are the number of vertices, edges and cycles respectively.

MOCGraph [61], support it. Commutativity is also efficiently used in numerical analytics since arithmetic operations satisfy commutativity. For example, deep learning algorithms update variables by a delta value, and the addition operations of delta values are commutative.

Assumption 2. (Pull-mode) Each UDF reads the information from its neighbours and writes the update value to itself.

There are two modes [50] in vertex-centric analytics: push and pull. The main difference between them is whether the UDF proactively pushes updates to its neighbours or passively pulls information. We assume the pull mode is being used in analytics.

Here we introduce our main theorem. NoCon is the lowest level in transaction processing: the reads and writes are freely executed without any read or write locks.

THEOREM 3. *Under the above mentioned assumptions and the BN3, the transactions executed in NoCon are SI.*

To prove our theorem, we begin with some notations. By the definition of dependency edge, if $u \not\prec v$, there are no edges between them. If there is an edge between u and v, we mark it as a **blue** edge if $u \prec v$ or $v \prec u$; otherwise (when $u \parallel v$), we mark it as a **red** edge. Let the *begin time* bt_u (resp. *finish time* ft_u) of a transaction u denote the earliest (resp. latest) time that u's operations begin (resp. finish). Clearly, $bt_u < ft_u$. And we have the following corollary.

COROLLARY 1. *(Timely ordered) If there is a blue edge from u to v, we have $u \prec v$ and thus $ft_u < bt_u$.*

Before we start proving our theorem, we introduce two lemmas.

LEMMA 1. *(Partial ordered acyclic) We assume there is a graph and an assignment $h(v) : V \to \mathbb{R}$ (from vertices to real numbers) that for all edges from vertex u to v, $h(u) < h(v)$. Then the graph contains no cycle.*

The partial ordered acyclic lemma is from graph theory. We introduce a simplified proof here.

PROOF. Suppose there is a such cycle $v_1, v_2, \ldots, v_k, v_1$. Therefore we know $h(v_1) < h(v_2) \cdots < h(v_k) < h(v_1)$ which implies $h(v_1) < h(v_1)$. This is a contradiction. □

LEMMA 2. *Under the BN3, for any **two** different transactions u and v, such that $u \otimes v$, their execution follows SI when executed in NoCon.*

PROOF. We inspect all possible cycles formed by these two transactions. There are three possible edges (*rw*, *ww* and *wr*) from u to v and vice versa. Keeping symmetry in mind, we analyze them case by case. Following Theorem 2, we attempt to show either there is no cycle or each cycle has at least two adjacent *rw* edges.

- $u \xrightarrow{ww} v$ and $v \xrightarrow{ww} u$. This case is not possible because either u writes before v or vice versa since there is only one write operation in each transaction (c.f. Assumption 2). They cannot occur together.
- $u \xrightarrow{wr} v$ and $v \xrightarrow{wr} u$. This case is also not possible. Let $r(v)$ (resp. $w(v)$) represent the read (resp. write) part of v, and $r(v) \xrightarrow{RCU} w(v)$ represent our Observation 3. We have $r(v) \xrightarrow{RCU} w(v) \xrightarrow{wr} r(u) \xrightarrow{RCU} w(u) \xrightarrow{wr} r(v)$. This is in contradiction with the monotonicity of time.
- $u \xrightarrow{rw} v$ and $v \xrightarrow{rw} u$. The cycle of two adjacent $\xrightarrow{rw}$ edges is allowed in SI.
- $u \xrightarrow{wr} v$ and $v \xrightarrow{ww} u$. Consider the data a over-written by u. The only possible time order is v writes a, which is overwritten by u and then read by v. In other words, v reads from a after writing to it. This is in contradiction with our Observation 3.
- $u \xrightarrow{rw} v$ and $v \xrightarrow{wr} u$. Similar to the last case, consider the data a over-written by v. The only possible time order is u reads a, which is overwritten by v and then read by u again. u reads a twice, which is in contradiction with our Observation 3.
- $u \xrightarrow{rw} v$ and $v \xrightarrow{ww} u$. Since u writes after v, the time order of the operations must be $r(u)$, $w(v)$ and $w(u)$. Through item commutativity, it is safe to swap $w(v)$ and $w(u)$ under the assumption of the BN3.

Thus, we prove of lemma above. □

With the help of Lemma 1 and 2, we can prove our Theorem 3.

PROOF. We consider the possible color combinations of the cycles. First, consider cycles that are made of blue edges only. For a blue edge from u to v, we have $ft_u < bt_v < ft_v$. The first inequality is from Corollary 1 and the second is trivial. Therefore, with the help of Lemma 1 (ft as $h(\cdot)$), we claim that pure blue cycles do not exist.

Next, consider pure red cycles. From the definition of the BN3, for each transaction u, we have $b_u \leq 1$. In other words, each transaction's red degree, which is the number of red edges, is no more than 1. Because the relationship $\otimes$ is reflective, when $u \otimes v$, we have $v \otimes u$. Therefore, the only possible cycles that are made by red edges only are the cases listed in Lemma 2. They satisfy SI as proved in Lemma 2.

Finally, consider mixed color cycles. We repeatedly shrink two vertices (in the dependency graph) connected by one red edge. To shrink two vertices u and v, we remove them from the graph and add a new vertex w. The edges connected to u and v are now connected to w. $bt_w = max(bt_u, bt_v)$, and similarly, $ft_w = min(ft_u, ft_v)$. We continue the shrinking process until all red edges are eliminated. First, we find that the shrinking operation does not break timely order property during the process. It is easy to see that $bt_w < ft_w$ still holds when u and v are connected by red edges. Now, consider other vertices x connected to u and/or v by blue edges. We can see that the timely order holds when x is an inbound neighbour of either u or v, or both of them. We can handle the outbound neighbour cases in a similar way. x cannot be an inbound neighbour of u and an outbound neighbour of v or vice versa. Therefore, there is no cycle at all in the remaining blue-only graph, as shown in Lemma 1. We also find that the shrinking operation does not eliminate mixed color cycles. Therefore, mixed color cycles does not exist before shrinking. □

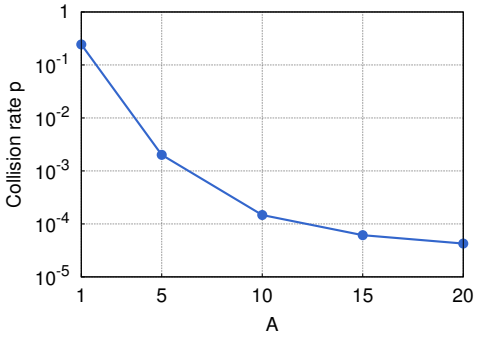

Figure 3: The collision rate p measured in synthetic graphs

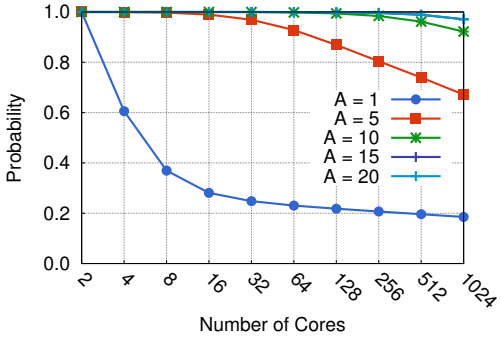

Figure 4: The probability $\Pr(b \leq 1)$ in graphs

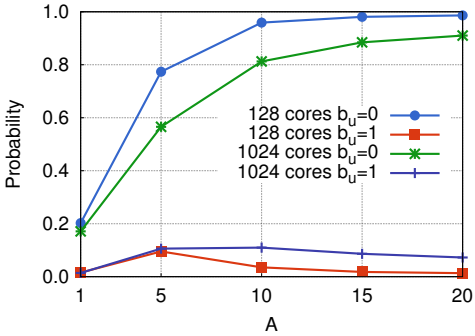

Figure 5: The probability $\Pr(b = 0)$ and $\Pr(b = 1)$ in graphs

We show a counter example that without BN3, the transactions do not follow SI. In other words the BN3 is necessary for NoCon to be SI.

Example 1. Let $w_u(x)$ represent a write operation issued by transaction u on data x. Similarly we define $r_w(x)$ as read operation.

The following transaction history does not follow SI.

$$r_a(x)\ w_b(x)\ r_c(x)\ w_c(y)\ w_a(y)$$

It contains the following cycle

$$a \xrightarrow[x]{rw} b \xrightarrow[x]{wr} c \xrightarrow[y]{ww} a$$

The cycle contains only one rw edge.

5. EXPERIMENTS

Experiment Setting: We have conducted extensive experiments on one Amazon EC2 c4.8xlarge instance with 36 cores and 60 GB memory. The experiments that use more than 36 cores are conducted in x1.32xlarge instance which has 128 cores and 1,952 GB memory. We implemented all methods in C++ and compiled them with gcc 6.2.0.

Synthetic Dataset: We generate synthetic graph datasets for experiments. Given a vertex number V and an average degree d, we generate a graph using the preference attachment method proposed in [13]. We use synthetic datasets instead of real graphs because they have a tunable "knob" A. A controls the "power-law-ness" $\frac{A}{d}$. Smaller A helps to generate more skewed degree distribution, i.e. more high degree vertices, which in further leads to higher collision rate p. Therefore A helps us control the p. In our experiments we choose V as 10 millions and d as 10. We index the graph with the subscript of A: that G_{10} is the graph with parameter $A = 10$.

5.1 BN3: Is It True?

First, we measure the collision rate p in synthetic data. We report the results in Figure 3. The collision rate p ranges from 2×10^{-1} to 4×10^{-5}. The graph with smaller A has larger p. In real graphs (c.f. Table 3), the collision rates are close to 10^{-4}.

We also measure the probability that $b_u \leq 1$, in other words the BN3, holds on the graphs. The results are illustrated in Figure 4. We vary the number of cores in the x-axis and draw $\Pr(b \leq 1)$ in the y-axis. When p is larger (in graphs with smaller A), the probability that BN3 holds is lower. However, for graphs which have the p values close to real graphs' p values, the probability that BN3 is true is high. For G_{20}, BN3 holds on 99.94% of the vertices when there are 128 cores. It holds on 98.2% of vertices even with 1024 cores. In real world practice, the user tends to use a reasonable number of resources compared with the volume of data: 128 cores already "overkill" our data by finishing 1 iteration of Page Rank in less than 0.1 second (c.f. Section 5.3). Therefore, we have confidence that BN3 is true for a wide range of data (and user environments).

To show that $b_u \leq 1$ is necessary and cannot be replaced by $b_u = 0$, we separate $\Pr(b = 0)$ and $\Pr(b = 1)$ and illustrate them in Figure 5. For space reason we only draw 128 cores and 1024 cores. Take the G_{10} graph as an example, $\Pr(b = 0)$ is about 95% when we have 128 cores. However $\Pr(b \leq 1)$ is 99.94% for the same setting.

5.2 Weak Consistency in BN3

The b_u only acts as an upper bound of serializability violations. To measure the exact number of violations, we execute the Page Rank algorithms. We record all the read and write operations during the execution and calculate the number of cycles in dependency graphs.

We start from an empty dependency graph and add the edges formed by operations, following the chronological order. If the new edge causes any cycle, we discard it and record the cycle. Otherwise we add it to the dependency graph. The cycle detection algorithm is time consuming, so we only measure the cycles of length 2, 3 and 4. We try all

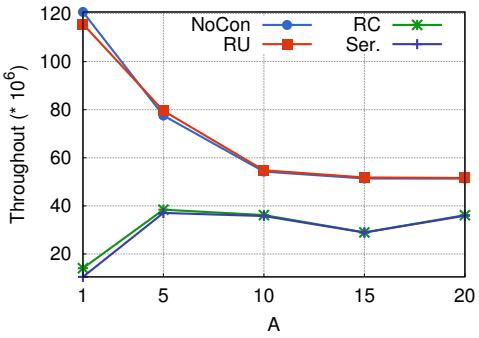

Figure 6: The number of UDFs executed per second (on 128 cores)

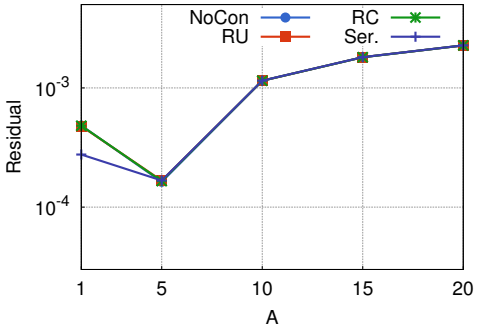

Figure 7: The residual after 50 iterations (on 128 cores)

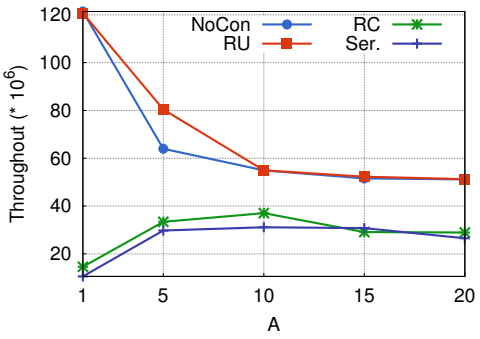

Figure 8: The number of UDFs executed per second (on 256 cores)

graphs and vary the number of cores. The number of cycles is less than 100 for most of the case. Smaller A and large number of cores cause more cycles.

5.3 Weak Consistency and Analytical Jobs

We conduct experiments on executing analytical tasks on weak consistency environments. We choose Page Rank as the analytical task. We implement four consistency levels, namely NoCon, Read Uncommitted (RU), Read Committed (RC) and Serializability.

We measure the time cost per UDF executions and the convergence per UDF executions. For the former, we report the number of UDF executions per second in Figure 6 and 8 when there are 128 and 256 cores. The trends are similar on different number of cores. The weaker consistency levels

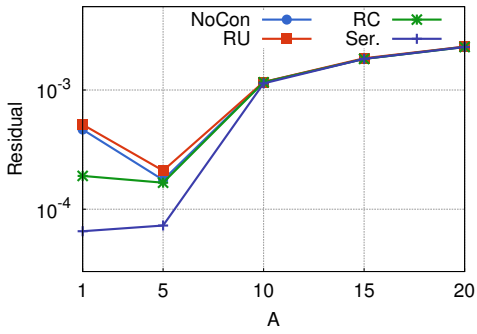

Figure 9: The residual after 50 iterations (on 256 cores)

NoCon, RU are at least 1.5x faster in all cases. Note when a graph has smaller A, the high degree vertices are more likely to concentrated in a dense area. This prolongs the lock waiting time significantly. For the later, we measure the residual after 50 iterations in Figure 7 and 9, for 128 cores and 256 cores respectively. The smaller residual, the better convergence. Stronger consistency has smaller residual only when the graph has larger collision rate p, i.e., smaller A. In G_1, to reach the same convergence levels, serializability uses 48 iterations while NoCon uses 50 ones. However, serializability is at least 10x slower than NoCon.

6. RELATED WORK

Generally speaking, data consistency is necessary for computing that may contain arbitrary read/write sequences. However, if the systems rely on certain special properties of UDFs, it is possible to use weaker consistency or no consistency at all while still ensuring correctness. Bailis [3] investigate the consistency constraints that could be protected without coordination. Rinard et al. [43] explore commutativity that can avoid certain consistency. Operations on CRDT [47] do not require consistency maintenance either. If user is able to specify the side effect of UDFs in higher level language, some systems are able to derivate when the consistency can be eliminated safely [6, 44, 33, 53, 29, 30, 27, 21, 26]. It is also able to detect [15] and classify [38] data races (consistency anomalies). If all data races are classified as benign, the consistency controller could be removed from the system without introducing any error. Recently, Holt et al. propose protocols [22] that preserves consistency constrains approximately. The approximated constraints in database community traces back to epsilon-serializability [40]. However, these works do not investigate the relationship between operational-level weak consistency and the accuracy of analytical results.

HogWild! [41] proposes an algorithm without the requirement consistency view of the data. It removes all locking operations on the stochastic gradient descent (SGD) algorithm. Motivated by HogWild!, people propose consistency-tolerant variations on other algorithms, like stochastic coordinate descent (SCD) [34], stochastic dual coordinate ascent (SDCA) [23], variance deduction (VR) [42] and non-convex problems [45]. There are variations of these problems which can adaptively adjust their parameters according to the latency of the data storage [60, 35, 14, 25, 37]. However, these algorithms' correctness rely on special properties of under-

lying problems, including but not limited to (strong) convexity, Lipschitz continuous, bounded derivative, bounded staleness, etc.

Zellag et al. [57, 58] propose methods for detecting and classifying the consistency anomalies. Fekete et al. propose necessary conditions [16, 17] for identification of the anomalies in snapshot isolation (SI). These works target on OLTP environment. A critical requirement for OLTP workload is zero-tolerance of consistency anomalies as the imaginary application of OLTP, for example e-banking, does not allow any client's account to be compromised by bad data. According to their results, they are not able to scale to hundreds of thousands transactions per second. This paper targets a different angle of consistency maintenance: the data consistency in a parallel shared-memory system. We assume the target of end-user is to finish the job as soon as possible while ensuring the general accuracy of the final results (as a whole).

7. FUTURE WORKS

This paper proposes the conjecture of `BN3`. Although we believe it is almost true everywhere, it is better to monitor the b_u for all or some of the UDFs in a real-time manner. However, how to minimize the overhead of such monitoring is critical to the performance of the system.

Sometimes `BN3` is not hold while weak consistency systems still produce high quality results. Investigating the root cause of why weak consistency is also strong under such situations without the assumption `BN3` is important to help unleash the potential power of weak consistency systems.

8. CONCLUSION

We investigate an important but underexploited phenomenon: weak consistency works well on analytical tasks where strong consistency is supposed to be indispensable. We propose a simple and effective explanation to reveal why and when weak consistency is actually strong enough. We take a novel data-centric approach, which shed light on future studies on designing and utilizing weak consistent systems. We conduct systems to verify our discoveries.

Acknowledgment

The authors thank the anonymous reviewers for their valuable comments. The work was supported by grant of Research Grants Council of the Hong Kong SAR, China No. 14209314 and 14221716.

9. REFERENCES

[1] http://highscalability.com/blog/2016/3/16/jeff-dean-on-large-scale-deep-learning-at-google.html. Accessed: 2016-08-10.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. 2015. Software available from tensorflow.org.

[3] P. Bailis. *Coordination Avoidance in Distributed Databases*. PhD thesis, EECS Department, University of California, Berkeley, Oct 2015.

[4] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Feral concurrency control: An empirical investigation of modern application integrity. *SIGMOD '15*, pp. 1327–1342, 2015.

[5] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. HAT, Not CAP: Towards highly available transactions. HotOS '13, 2013.

[6] A. Bernstein, P. Lewis, and S. Lu. Semantic conditions for correctness at different isolation levels. *ICDE '00*, pp. 57–66, 2000.

[7] P. Boldi, A. Marino, M. Santini, and S. Vigna. BUbiNG: Massive crawling for the masses. *WWW '04*, pp. 227–228, 2014.

[8] P. Boldi, M. Santini, and S. Vigna. A large time-aware graph. *SIGIR Forum*, 42(2):33–38, 2008.

[9] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. *OSDI '14*, pp. 571–582, Oct. 2014.

[10] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. GeePS: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. *EuroSys '16*, pp. 4:1–4:16, 2016.

[11] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. NIPS, 2012.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *OSDI '04*, pp. 10–10, 2004.

[13] S. N. Dorogovtsev, J. F. F. Mendes, and A. N. Samukhin. Structure of growing networks with preferential linking. *Phys. Rev. Lett.*, 85:4633–4636, Nov 2000.

[14] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011.

[15] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. *SOSP '03*, pp. 237–252, 2003.

[16] A. Fekete, S. N. Goldrei, and J. P. Asenjo. Quantifying isolation anomalies. *Proc. VLDB Endow.*, 2(1):467–478, Aug. 2009.

[17] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[18] J. E. Gonzalez, P. Bailis, M. I. Jordan, M. J. Franklin, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Asynchronous complex analytics in a distributed dataflow architecture. *CoRR*, abs/1510.07092, 2015.

[19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. *OSDI '12*, pp. 17–30, 2012.

[20] M. Han and K. Daudjee. Giraph Unchained: Barrierless asynchronous parallel execution in

pregel-like graph processing systems. *Proc. VLDB Endow.*, 8(9):950–961, May 2015.

[21] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. *POPL '13*, pp. 317–328, 2013.

[22] B. Holt, J. Bornholt, I. Zhang, D. Ports, M. Oskin, and L. Ceze. Disciplined inconsistency with consistency types. *SoCC '16*, pp. 279–293, 2016.

[23] C. Hsieh, H. Yu, and I. S. Dhillon. PASSCoDe: Parallel asynchronous stochastic dual co-ordinate descent. *ICML '15*, pp. 2370–2379, 2015.

[24] D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[25] P. Joulani, A. György, and C. Szepesvári. Delay-tolerant online convex optimization: Unified analysis and adaptive-gradient algorithms. *AAAI '16*, pp. 1744–1750, 2016.

[26] C. Kirsch, M. Lippautz, and H. Payer. Fast and scalable, lock-free k-FIFO queues. *PaCT '13*, pp. 208–223, 2013.

[27] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proc. VLDB Endow.*, 2(1):253–264, Aug. 2009.

[28] J. Leskovec and A. Krevl. http://snap.stanford.edu/data, June 2014.

[29] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. *USENIX ATC '14*, pp. 281–292, June 2014.

[30] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. *OSDI '12*, pp. 265–278, 2012.

[31] H. Li, A. Kadav, E. Kruus, and C. Ungureanu. MALT: Distributed data-parallelism for existing ml applications. *EuroSys '15*, pp. 3:1–3:16, 2015.

[32] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. *OSDI '14*, pp. 583–598, Oct. 2014.

[33] J. Liu, T. Magrino, O. Arden, M. D. George, and A. C. Myers. Warranties for faster strong consistency. *NSDI '14*, pp. 503–517, 2014.

[34] J. Liu, S. J. Wright, C. Ré, V. Bittorf, and S. Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. *J. Mach. Learn. Res.*, 16(1):285–322, Jan. 2015.

[35] H. B. McMahan and M. Streeter. Delay-tolerant algorithms for asynchronous distributed online learning. *NIPS '14*, pp. 2915–2923, 2014.

[36] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer. Graph structure in the web — revisited: A trick of the heavy tail. *WWW '14 Companion*, pp. 427–432, 2014.

[37] I. Mitliagkas, C. Zhang, S. Hadjis, and C. Ré. Asynchrony begets momentum, with an application to deep learning. *CoRR*, abs/1605.09774, 2016.

[38] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. *PLDI '07*, pp. 22–31, 2007.

[39] X. Pan, M. Lam, S. Tu, D. Papailiopoulos, C. Zhang, M. I. Jordan, K. Ramchandran, C. Re, and B. Recht. Cyclades: Conflict-free asynchronous machine learning. *NIPS '16*. 2016.

[40] K. Ramamritham and C. Pu. A formal characterization of epsilon serializability. *IEEE Trans. on Knowl. and Data Eng.*, 7(6):997–1007, Dec. 1995.

[41] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. *NIPS '11*, pp. 693–701. 2011.

[42] S. J. Reddi, A. Hefny, S. Sra, B. Pöczos, and A. Smola. On variance reduction in stochastic gradient descent and its asynchronous variants. *NIPS'15*, pp. 2647–2655, 2015.

[43] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, Nov. 1997.

[44] S. Roy, L. Kot, N. Foster, J. Gehrke, H. Hojjat, and C. Koch. Writes that fall in the forest and make no sound: Semantics-based adaptive data consistency. *CoRR*, abs/1403.2307, 2014.

[45] C. D. Sa, C. Re, and K. Olukotun. Global convergence of stochastic gradient descent for some non-convex matrix problems. *ICML '15*, pp. 2332–2341, 2015.

[46] Z. Shang, F. Li, J. X. Yu, Z. Zhang, and H. Cheng. Graph analytics through fine-grained parallelism. *SIGMOD '16*, pp. 463–478, 2016.

[47] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. *SSS '11*, pp. 386–400, 2011.

[48] K. Vora, S. C. Koduru, and R. Gupta. ASPIRE: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based DSM. *OOPSLA '14*, pp. 861–878, 2014.

[49] G. Wang, W. Xie, A. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. *CIDR '13*, 2013.

[50] Z. Wang, Y. Gu, Y. Bao, G. Yu, and J. X. Yu. Hybrid pulling/pushing for I/O-efficient distributed and iterative graph computing. *SIGMOD '16*, pp. 479–494, 2016.

[51] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., 2001.

[52] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. SYNC or ASYNC: Time to fuse for distributed graph-parallel computation. *PPoPP 2015*, pp. 194–204, 2015.

[53] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining ACID and BASE in a distributed database. *OSDI '14*, pp. 495–509, Oct. 2014.

[54] E. P. Xing, Q. Ho, W. Dai, J.-K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. *KDD '15*, pp. 1335–1344, 2015.

[55] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad

hoc synchronization considered harmful. *OSDI '10*, pp. 163–176, 2010.

[56] F. Yang, J. Li, and J. Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *Proc. VLDB Endow.*, 9(5):420–431, Jan. 2016.

[57] K. Zellag and B. Kemme. How consistent is your cloud application? *SoCC '12*, pp. 6:1–6:14, 2012.

[58] K. Zellag and B. Kemme. Consistency anomalies in multi-tier architectures: Automatic detection and prevention. *The VLDB Journal*, 23(1):147–172, Feb. 2014.

[59] S. Zhang, A. Choromanska, and Y. LeCun. Deep learning with elastic averaging sgd. *NIPS '15*, pp. 685–693, 2015.

[60] W. Zhang, S. Gupta, X. Lian, and J. Liu. Staleness-aware async-SGD for distributed deep learning. *IJCAI '16*, pp. 2350–2356, 2016.

[61] C. Zhou, J. Gao, B. Sun, and J. X. Yu. MOCgraph: Scalable distributed graph processing using message online computing. *Proc. VLDB Endow.*, 8(4):377–388, Dec. 2014.

APPENDIX

THEOREM 4 (C.F. SECTION 3). *If for any two transactions u and v we have $u \nparallel v$ or $u \not\prec v$, the execution satisfy conflict serializability.*

PROOF. We utilize Lemma 1 to prove this theorem. Here we choose the *finish time* $ft_{(v)}$ as the assignment function $h(v)$ in Lemma 1. Therefore we need to prove for any edge from u to v, $ft_u < ft_v$. If there is at least an edge from u to v, we have $u \cap v$, in other words $u \not\prec v$ is not true. Therefore we must have $u \nparallel v$. By definition $u \nparallel v$ means either $u \prec v$ or $v \prec u$. Since there is an edge from u to v we know $u \prec v$. Therefore $ft_u < bt_v < ft_v$. □