Paper 1: Li et al.

Title Making Geo-Replicated Systems Fast as Possible,Consistent when necessary
10th USENIX Symposium on Operating Systems Design and Implementation
Authors: Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguica, Rodrigo Rodrigues
Date: 2012

Motivation:

1. To improve user-experience, services replicate system state across geographically diverse sites.
2. Performance vs Consistency
   - Amazonś Dynamo - eventual consistency where state temporarily converge.
   - Yahoo PNUTS - avoids state divergence by requiring all operations that update the service state to be funneled through a primary site and thus incurring increased latency.

Overview:

1. RedBlue Consistency - Blue operations execute locally and are lazily replicated.Red operations are serialized with respect to each other and are immediately cross-site coordinated.
2. Conditions under which operations must be colored red or blue.
3. Decomposing operations into two components - a generator operation and a shadow operation.

Properties of Geo-Replicated Systems

1. Low latency - Operations should proceed after contacting a small number of users.
2. Causality - Monotonicity of user request within session and also preserving causality across clients
3. State Convergence - All replicas have executed the same set of operations
4. All operations should return a single value.
5. The system should provide a set of stable histories and support for general operations.
6. The system should preserve a set of invariants.
7. Eventual Propagation

Related Work: Consistency

Consistency level	Example systems	Immediate response	State convergence	Single value	General operations	Stable histories	Classification strategy
Strong	RSM [20, 31]	no	yes	yes	yes	yes	N/A
Timeline/snapshot	PNUTS [8], Megastore [3]	reads only	yes	yes	yes	yes	N/A
Fork	SUNDR [24]	all ops	no	yes	yes	yes	N/A
Eventual	Bayou [38], Depot [26]	all ops	yes	no	yes	yes	N/A
	Sporc [12], CRDT [33]	all ops	yes	yes	no	yes	N/A
	Zeno [34], COPS [25]	weak/all ops	yes	yes	yes	no	no / N/A
Multi	PSI [35]	cset	yes	yes	partial	yes	no
	lazy repl. [19], Horus [39]	immed./causal ops	yes	yes	yes	yes	no
RedBlue	Gemini	Blue ops	yes	yes	yes	yes	yes

Table 1: Tradeoffs in geo-replicated systems and various consistency levels.

Related Work: Levels of Consistency

1. Strong Consistency - Replicated systems behave like a single server that serialize all operations.
2. Timeline/Snapshot Consistency - There is a total order for updates to the service state but gives the option of reading a consistent but dated view of the service.
3. Fork Consistency - Relaxes strong consistency by allowing users to observe distinct causal histories.
4. Eventual Consistency - All replicas "eventually" diverge at some state.
5. Multi Consistency - Other systems expose multiple values from divergent branches in operation replies either directly to the client or to an application-specific conflict resolution procedure.
6. RedBlue Consistency - Operations have multiple consistency levels

Related Work: Other

1. Consistency Rationing - Consistency guarantees associated with the data instead of the operation. Also switches consistency levels at runtime.
2. TACT - bounds the amount of inconsisteny based on parameters like numeric errors, order errors, staleness, etc.

System Model - Assumptions

1. A distributed system with state fully replicated across k sites denoted $site_0 \ldots site_{k-1}$
2. S $\in \mathcal{S}$ denotes a system state and $u, v \in \mathcal{O}$ a set of operations.
3. Initial State - S_0. When operation u is applied it goes to state S'. So $S' = S + u$
4. $\forall$ S $\in \mathcal{S}$, S+$u + v =$ S $+v + u$
5. A state S is valid if it satisfies all these invariants.
6. Each u is submitted to one site which is called $u's$ primary site and denoted by $site(u)$.
7. The system later replicates u to the other sites.

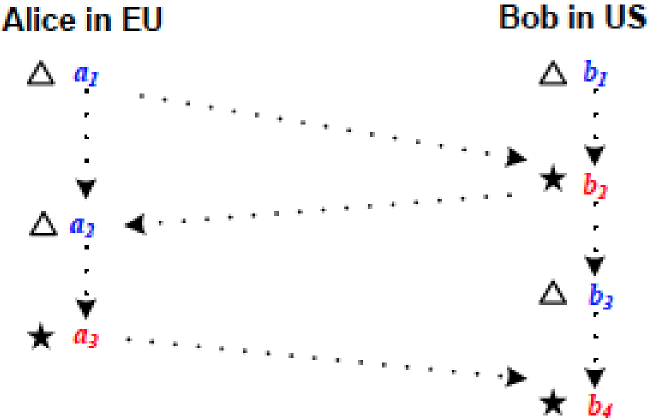
RedBlue Consistency

- RedBlue order : Given a set of operations $U = B \cup R$,where $B \cap R = \emptyset$, a RedBlue order is a partial order $O = (U, \prec)$ with the restriction that $\forall u, v \in R$ such that $u \neq v, u \prec v$ or $v \prec u$ ($i.e.$ red operations are totally ordered).
- Causal Serialization : Given a site i, $O_i = (U, <)$ is an i-causal serialization(or short, a causal serialization) of RedBlue order $O = (U, \prec)$ if
  1. O_i is a linear extension of O ($i.e$, ¡ is a total order compatible with the partial order $\prec$)
  2. for any two operations $u, v \in U$, if $site(v) = i$ and $u < v$ in O_i then $u \prec v$

RedBlue Consistency - Definition

- RedBlue consistency : A replicated sytem is O-RedBlue consistent(or short, RedBlue consistent) if each site i applies operations according to an i-causal serialization of RedBlue order O.

Example



(a) RedBlue order O of operations

Example

(b) Causal serializations of O

State Convergence

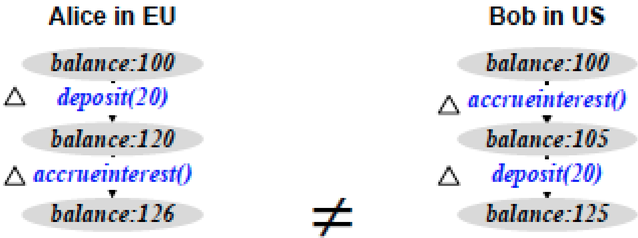
- A RedBlue consistent system is state convergent if all causal serializations of the underlying RedBlue order O reach the same state S.

State Convergence:Example

```
float balance, interest = 0.05;
func deposit( float money ):
   balance = balance + money;
func withdraw ( float money ):
   if ( balance - money >= 0 ) then:
      balance = balance - money;
   else print "failure";
func accrueinterest():
   float delta = balance × interest;
   balance = balance + delta;
```

State Convergence:Example

(a) RedBlue order O of operations issued by Alice and Bob



State Convergence:Theorem

Theorem

Given a RedBlue order O, if all blue operations are globally commutative then any O-RedBlue consistent system is state convergent

Replicating side effects -Generator Operation and Shadow Operation

1. Generator Operation g_u - executed only at primary site against some system state S.
2. Shadow Operation $h_u(S)$ - executed at every site(including the primary site)

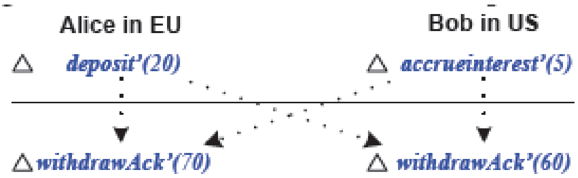
Replicating side effects - Defining shadow operations

1. Correct Generator/ Shadow Operations : The decomposition of operation u into generator and shadow operations is correct if for all states S, the generator operation g_u has no effect and the generated shadow operation $h_u(S)$ has the same effect as u, *i.e.*, for any state $S : S + g_u = S$ and $S + h_u(S) = S + u$

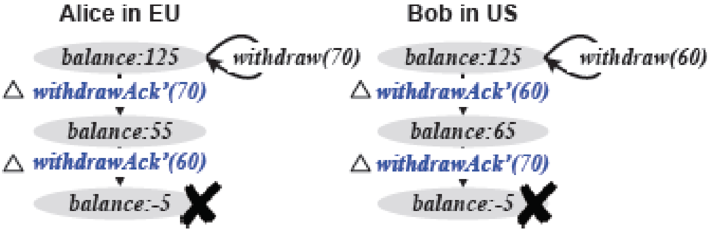
Shadow Banking and Invariants - Example


```
func deposit′ ( float money ):
   balance = balance + money;
func withdrawAck′ ( float money ):
   balance = balance - money;
func withdrawFail′ ():
   /* no-op */
func accrueinterest′ ( float delta ):
   balance = balance + delta;
```

Shadow Banking and Invariants - Example



(a) RedBlue order O of banking shadow operations



Shadow Banking and Invariants - Example

- Invariant Safe - Shadow operation $h_u(S)$ is invariant safe if for all valid states S and S', the state $S' + h_u(S)$ is also valid.

Theorem

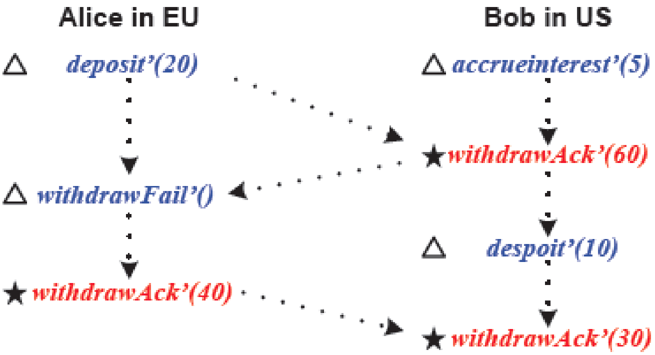
If all shadow operations are correct and all blue shadow operations are invariant safe and globally commutative,then for any execution of that system that is RedBlue consistent, no site is ever in an invalid state.

What can be blue? What can be red?

The procedure for deciding which shadow operations can be blue or must be red if a RedBlue consistent system is to provide both state convergence and invariant preservation:

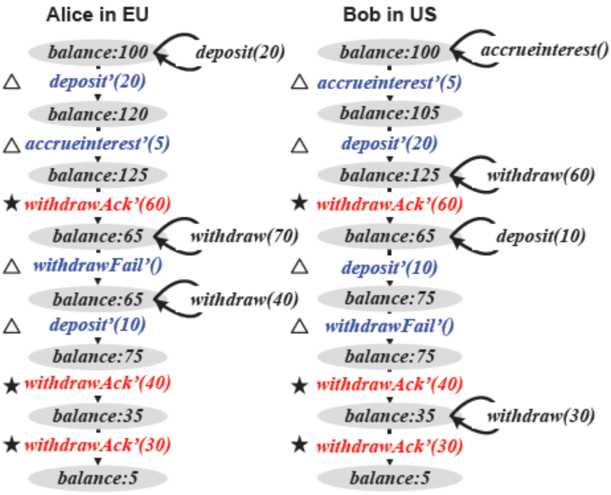
1. For any pair of non-commutative shadow operations u and v, label both u and v red.
2. For any shadow operation u that may result in an invariant being violated, label u red.
3. Label all non-red shadow operations blue.

Shadow Banking and Invariants - Example



(a) RedBlue order O of banking shadow operations

Shadow Banking and Invariants - Example



(b) Convergent and invariant preserving causal serializations of O

Case Studies

- TPC-W shopping cart benchmark
- RUBiS auction benchmark
- Quoddy social networking application

Two main tasks :

- Decomposing the application into a generator and shadow operation
- Labeling the shadow operations appropriately

Original application to RedBlue Consistent

Application	Original					RedBlue consistent extension				
	user	transactions			LOC	shadow operations				LOC
	requests	total	read-only	update		blue no-op	blue update	red	LOC	changed
TPC-W	14	20	13	7	9k	13	14	2	2.8k	429
RUBiS	26	16	11	5	9.4k	11	7	2	1k	180
Quoddy	13	15	11	4	15.5k	11	4	0	495	251

Table 2: Original applications and the changes needed to make them RedBlue consistent.

TPC-W

- Serves 14 different user requests such as browsing, searching, adding products to a shopping cart or placing an order.
- Each user request generates one to four transactions that access state stored across eight different tables.
- Shopping cart can be shared by multiple users across multiple sessions.

TPC-W - Writing TPC-W generator and shadow operations


```
doBuyConfirm(cartId) {
 beginTxn();
 cart = exec(SELECT * FROM cartTb WHERE cId=cartId);
 cost = computeCost(cart);
 orderId = getUniqueId();
 exec(INSERT INTO orderTb VALUES(orderId, cart.item.id, cart.item.qty
      ,cost));
 item =exec(SELECT * FROM itemTb WHERE id=cart.item.id);
 if item.stock - cart.item.qty < 10 then:
    delta = item.stock - cart.item.qty + 21;
    if delta > 0 then:
       exec(UPDATE itemTb SET item.stock+=delta);
    else rollback();
 else exec(UPDATE itemTb SET item.stock-=cart.item.qty);
 exec(DELETE FROM cartContentTb WHERE cId=cartId AND id=
      cart.item.id);
 commit();}
```

(a) Original transaction that commits changes to database.

TPC-W - Writing TPC-W generator and shadow operations

```
doBuyConfirmGenerator(cartId){
 sp = getScratchpad();
 sp.beginTxn();
 cart = sp.exec(SELECT * FROM cartTb WHERE cId=cartId);
 cost = computeCost(cart);
 orderId = getUniqueId();
 sp.exec(INSERT INTO orderTb VALUES (orderId,cart.item.id,
        cart.item.qty,cost));
 item = sp.exec(SELECT * FROM itemTb WHERE id=cart.item.id);
 if item.stock − cart.item.qty < 10 then:
    delta = item.stock − cart.item.qty + 21;
    if delta > 0 sp.exec(UPDATE itemTb SET item.stock+ = delta);
    else sp.discard(); return;
 else sp.exec(UPDATE itemTb SET item.stock− = cart.item.qty);
 sp.exec(DELETE FROM cartTb WHERE cId=cartId AND id=cart.item.id);
 L_TS = getCommitOrder();
 sp.discard();
 if replenished return (doBuyConfirmIncre′(orderId,cartId,
        cart.item.id,cart.item.qty,cost,delta,L_TS));
 else return (doBuyConfirmDecre′(orderId,cartId,cart.item.Id,
        cart.item.qty,cost,L_TS));}
```

(b) Generator operation that manipulates data via a private *scratchpad*.

TPC-W - Writing TPC-W generator and shadow operations


```
doBuyConfirmIncre'(orderId, cartId, iId, qty, cost, delta, L_TS){
 exec(INSERT INTO orderTb VALUES(orderId, iId, qty, cost, L_TS));
 exec(UPDATE itemTb SET item.stock += delta);
 exec(UPDATE itemTb SET item.l_ts = L_TS WHERE item.l_ts < L_TS);
 exec(UPDATE cartContentTb SET flag = TRUE WHERE id = iId AND
       cid = cartId AND l_ts <= L_TS);}
```

(c) Shadow doBuyConfirmIncre' (Blue) that replenishes the stock value.

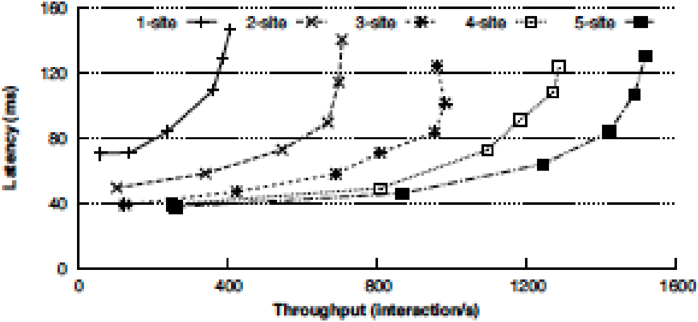
```
doBuyConfirmDecre'(orderId, cartId, iId, qty, cost, L_TS){
 exec(INSERT INTO orderTb VALUES(orderId, iId, qty, cost, L_TS));
 exec(UPDATE itemTb SET item.stock -= qty);
 exec(UPDATE itemTb SET item.l_ts = L_TS WHERE item.l_ts < L_TS);
 exec(UPDATE cartContentTb SET flag = TRUE WHERE id = iId AND
       cid = cartId AND l_ts <= L_TS);}
```

(d) Shadow doBuyConfirmDecre' (Red) that decrements the stock value.

Evaluation-Experimental Setup

- Experiments were run on Amazon EC2 using 5 virtual machine instances located in 5 sites - US east(UE), US west(UW), Ireland(IE), Brazil(BR) and Singapore(SG).
- Each VM has 8 virtual cores and 15 GB of RAM. VMs run Debian 6(Squeeze) 64 bit, MYSQL 5.5.18,Tomcat 6.0.35 and Sun Java SDK 1.6.

Evaluation-TPC-W



(a) TPC-W shopping mix

Paper 2: Lloyd et al.

Stronger Semantics for Low-Latency Geo-Replicated Storage

Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI13)

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen

April 2013

Motivation

- Consider an example from Facebook

Motivation

- Consider an example from Facebook
- Joe performs 2 actions:
  - Defriend boss.
  - Make snarky comment about boss and promise to quit.
- In an eventually consistent system, these can be viewed in the wrong order for a period time, earning Joe his first unemployment check.
- This won't happen with causal consistency.

Consistency - Causal versus Eventual

- Eventual Consistency
  - At some point, the replicas will converge
  - Until then, no guarantees
  - Joe's boss might see his whining.

Consistency - Causal versus Eventual

- Eventual Consistency
  - At some point, the replicas will converge
  - Until then, no guarantees
  - Joe's boss might see his whining.
- Causal Consistency
  - At a single processor, serial order of events determines causal order
  - Reads are causally ordered after their writes across processors
  - Transitive closure of these two properties
  - Joe is safe to gripe.

Overview

- We modify an eventually consistent system to get a causally consistent one
- Extend Cassandra to get Eiger
- Take slight hit in throughput
  to get stronger version of consistency
- Maintain low latency

Contributions of the paper

- Eiger
  - Low Latency
  - High throughput (slightly lower than Cassandra)
  - Causal Consistency (rather than eventual as in Cassandra)
- Read Only Algorithm
- Write Only Algorithm

Background

- Cassandra
  - Designed by Facebook to search inbox messages (has since been replaced by Hbase for better scalability)
  - Other companies are still using Cassandra (e.g., Netflix and Apple)
  - Allows Eventual or Strong Consistency
  - If you want low latency, Eventual is the only option

Eiger: Column Family Data Model

- Hierarchical Column families: 3 tiers
- Example:
  - Super Family = Association
  - Family = Friends
  - Column = Alice

	User Data		Associations				
			Friends			Likes	
	ID	Town	Alice	Bob	Carol	NSDI	SOSP
Alice	1337	NYC	-	3/2/11	9/2/12	9/1/12	-
Bob	2664	LA	3/2/11	-	-	-	-
⋮							

Figure 1: An example use of the column-family data model for a social network setting.

Eiger: Operations

- Cassandra has only 3 operations (key determines row)
  - Insert(table, key, rowMutation)
  - Get(table, key, columnName)
  - Delete(table, key, columnName)

Eiger: Operations

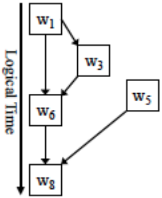
- Cassandra has only 3 operations (key determines row)
  - Insert(table, key, rowMutation)
  - Get(table, key, columnName)
  - Delete(table, key, columnName)
- Eiger adds more complex transactions
  - Batch Mutate: insert or delete multiple keys as a group of independent operations
  - Atomic Mutate: insert or delete multiple keys in a single atomic batch
  - Multiget Slice: read multiple columns/keys
  - Multiget Slice by Time

Eiger: Client Library

- Manages read and write algorithms
- Tracks causality by maintaining 1-hop dependencies
- For an atomic write, no write is applied in a cluster until after all the dependencies have been applied. This is necessary for causal consistency.
- Dependencies are on operations rather than values (as in COPS).

Eiger: One-Hop Dependencies

Example: For w8, only w6 and w5 are stored.

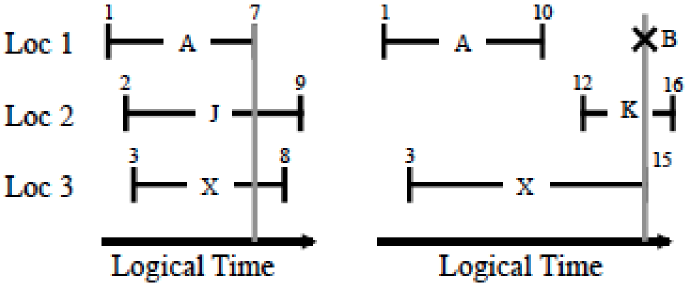


Op	Dependencies
w_1	-
w_3	w_1
w_5	-
w_6	w_3 w_1
w_8	w_6 w_5 w_3 w_1

Eiger: Read Algorithm

- Non-blocking: other processes can read
- Partition Tolerant
- 1 or 2 rounds of non-blocking parallel reads

Eiger: Read Algorithm



(a) One Round Sufficient (b) Two Rounds Needed

Eiger: Read Algorithm

- Step 1: Find latest lower bound, L
- Step 2: Find lowest upper bound that is higher than L
  If all upper bounds are higher than L then we are done. If not, continue.
- Step 3: For any reads that had an upper bound lower than L, read again.

Eiger: Write Algorithm

- Atomically write set of keys
- Lock free (and thus low latency)
- Does not block concurrent reads

Eiger: Write Algorithm

Three parties involved

- Client - requests write to many servers
- Coordinator server - a randomly chosen server from list of transaction destinations
- Cohort server - all other destinations

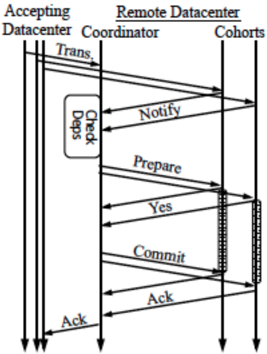
Eiger: Write Algorithm

Three parties involved

- Step 1: Break up transaction into a separate request for each destination.
- Step 2: Randomly choose coordinator and send transactions to all destinations.
- Step 3: Cohort servers send notification to coordinator.
- Step 4: Coordinator runs dependency checks
- Step 5: 2 Phase commit (to ensure all-or-nothing)
- Step 6: Coordinator sends ACK to client

Eiger: Write Algorithm

Example:



Evaluation

Comparison to Cassandra:

- Within 7% of throughput Using Facebook data

	Ops/sec	Keys/sec	Columns/sec
Cassandra	23,657	94,502	498,239
Eiger	22,088	88,238	466,844
Eiger All Txns	22,891	91,439	480,904
Max Overhead	6.6%	6.6%	6.3%

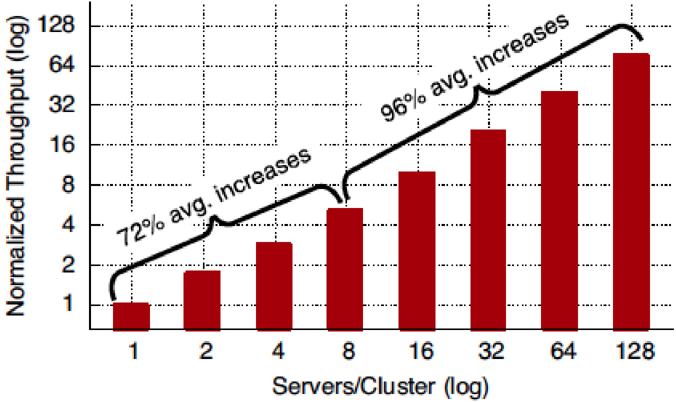
Evaluation

Comparison to Cassandra:

- Cassandra had median latency of 0.38ms, whereas Eiger had 0.78ms
- Thus, there is about a factor of 2 difference
- Strongly Consistent Cassandra had 85.21ms latency

Evaluation

Almost linear scaling:



Conclusion: Relationship between 2 Papers

- Similarities
  - Geo-replicated Systems
  - Provide improvement in consistency
- Differences
  - Li: Latency versus Consistency.
    - Separate operations into those that must be consistent and those that need not.
  - Lloyd: Consistency versus Throughput (requiring low latency)
    - Extend previous system to offer causal consistency with small hit to throughput.

The End

Thank you!

Bibliography

- **Stronger Semantics for Low-Latency Geo-Replicated Storage**, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI13)*, Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen, April 2013
- **Out in the Open: The Abandoned Facebook Tech That Now Helps Power Apple**, *www.wired.com* , Klint Finley, Aug. 4, 2014
- **A Short Primer on Causal Consistency**, *;login: The Usenix Magazine Volume 38, Number 4*, Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen, , August 2013
- **Netflix's Viewing Data: How We Know Where You Are in House of Cards**, *http://techblog.netflix.com/search/label/Cassandra* January 27, 2015