BASIC: an Alternative to BASE for Large-Scale Data Management System

Lengdong Wu, Li-Yan Yuan, Jia-Huai You
Department of Computing Science
University of Alberta
Edmonton, Canada
{lengdong, yuan, you}@cs.ualberta.ca

***Abstract*—Big data applications demand and consequently lead to developments of large-scale data management systems, which provide high scalability by partitioning data across multiple servers. Since conventional transactional access is quite expensive, many real world large-scale distributed systems eschew transactional functionality and adopt semantics of atomic multi-partition operations. Accordingly, BASE, a consistency model weaker than ACID, is commonly used to guarantee availability. In this work, we identify a new consistency model-BASIC (Basic Availability, Scalability, Instant Consistency) that matches the requirements where extra efforts are not needed to manipulate inconsistent soft states. We present a timestamp-based formula protocol for BASIC that can enforce Instant Consistency while achieving linear scalability (via logical formula caching, dynamic timestamp ordering) and achieve Basic Availability in the presence of partial failure and network partition (via partition independence, genuine atomic commit). Our extensive experimental results verify the scalability of BASIC and demonstrate that the limited overhead induced by BASIC pays a reasonable price for keeping all soft states consistent.**

***Keywords*—Consistency; Concurrency Control; Scalability**

I. Introduction

Faced with increasing amounts of data and unprecedented query volume, large-scale data management systems partition their data across multiple distributed servers, such that no one single server contains an entire copy of the database [1], [2], [3], [4], [5], [6], [7]. To minimize latency and remain available during partial failure and network partition, many modern large-scale distributed systems eschew transactional functionality, and opt for strong semantic guarantees for atomic non-transactional operations over multiple data partitions (atomic multi-partition operation) [2], [4], [8], [9]. Designing and implementing concurrency control protocols for these systems is quite challenging, especially when high scalability needs to be achieved without sacrificing consistency. Based on the understanding that linearizability of traditional ACID databases is not achievable with high availability in the presence of network partitions [10], [11], [12], many large-scale distributed systems have attempted to provide weaker consistency guarantees for atomic multi-partition operations - the most notable being BASE (Basic Availability, Soft State, Eventual Consistency) [4], [5], [8]. Despite of its high concurrency and performance benefits, BASE has its own inevitable problems resulting in incorrect behavior for use cases that require consistent visibility for users. Therefore, BASE should not overwhelmingly be the only default choice.

In this work, we present BASIC (Basic Availability, Scalability, Instant Consistency), an alternatively higher level of consistency than BASE, for a largely underserved class of applications requiring multi-partition, partial-replication, atomically non-transactional operational data access where none of inconsistent (soft) states should be visible to clients. BASE and BASIC provide different choices between the model that is faster but requiring extra efforts to deal with inconsistent results and the model that delivers consistent results but is relatively slower with higher latency.

Our contribution in this paper is to demonstrate that BASIC can be achieved with linear scalability, and the performance decline induced by BASIC is acceptable comparing with the extra efforts needed to manipulate inconsistent soft states of BASE. Specifically, we present a timestamp-based formula protocol with novel distributed timestamp management schemes (i.e., logical formula caching, dynamic timestamp ordering, partition independence, and genuine atomic commit) for high scalability and basic availability. We compare the performance and scalability between systems with BASE and BASIC under the industrial standard YCSB benchmark. Across a range of workload configurations, we show that BASIC incurs limited overhead that pays a reasonable price for a higher consistency level than BASE.

This paper proceeds as follows. In Section II, we exemplify the limitation of BASE with tangible examples, and then propose BASIC within generalized CAP framework in Section III. We present the timestamp-based formula protocol for BASIC in Section IV. The performance experiments are reported in Section V. Related work and conclusion are given in Section VI, VII, respectively.

II. Overview and Motivation

Given the prevalence of partitioning for large-scale data management systems, we consider a set of partitioned data items spread over multiple servers, and partially replicated where at least one server acts as a replica for a proper subset of data items. In such distributed infrastructure, network partition may prevent servers from communicating, and in the absence of server failures, communicating messages may also be delayed by factors such as network congestion and routing.

Consider an example of data items[1] with three partitions *L*, *S*, *H* distributed over three different servers N_1, N_2, N_3,

[1]For simplicity, we consider only one data item but all the discussions are valid for a set of data items.

respectively; and the following three atomic multi-partition operations on the data item:

1) $a(x)$: $W_a(L = L + x, H = H + x)$;
2) $b(y)$: $W_b(S = S + y, H = H - y)$;
3) $check$: $R_c(L, S, H)$, $assert(L - S = H)$.

The state of the data item transits from one consistent state to another if and only if it is caused by the completion of an atomic operation. We assume initial values in three columns L, S, H are all 0.

Example 1. Consider a schedule with no R/W or W/W overlap: $S_1 = \{a(20), b(10), check\}$ such that:

1) $a(20)$ completes before $b(10)$ starts;
2) $check$ is issued after $b(10)$ is done.

TABLE I: No R/W or W/W overlap schedule

schedule				value		
	P_1	P_2	P_3	L	S	H
t_1	$W_{a1}(L + 20)$		$W_{a3}(H + 20)$	20	0	20
t_2		$W_{b2}(S + 10)$	$W_{b3}(H - 10)$	20	10	10
t_3	$R_{c1}(L = 20)$	$R_{c2}(S = 10)$	$R_{c3}(H = 10)$			

Table I describes the schedule on each partition. Note that W_{ai}, W_{bi}, R_{ci} represent the corresponding actions of operations a, b and $check$ on the partition $P_i (i = 1, 2, 3)$. The *check* at t_3 returns a consistent state according to BASE, since there exists no overlap among operations on all partitions. However, this is not always the case, as multi-partition operations can overlap due to various reasons (e.g., network delay, transmission latency and etc.).

Example 2. Consider a schedule with R/W and/or W/W overlap: $S_2 = \{a(20), b(10), check, check\}$ such that:

1) $a(20)$ is incomplete at t_2 when $b(10)$ starts;
2) the first $check$ is issued at t_3 before $b(10)$ is done;
3) the second $check$ is issued after $b(10)$ is done,

Table II describes the schedule on each partition. There exist overlaps between $a(20)$ and $b(10)$; and $b(10)$ and the first $check$.

TABLE II: W/W and R/W overlap schedule

schedule				value		
$time$	P_1	P_2	P_3	L	S	H
t_1	$W_{a1}(L + 20)$			20	0	0
t_2		$W_{b2}(S + 10)$	$W_{a3}(H + 20)$	20	10	20
t_3	$R_{c1}(L = 20)$	$R_{c2}(S = 10)$	$R_{c3}(H = 20)$			
t_4			$W_{b3}(H - 10)$	20	10	10
t_5	$R_{c1}(L = 20)$	$R_{c2}(S = 10)$	$R_{c3}(H = 10)$			

The schedule in Table II satisfies BASE, since though the first *check* at t_3 returns an inconsistent state, while at some later time after t_4, the second *check* eventually achieves a consistent state. For such schedules, clients are required to reason about the correctness of each state which can be consistent or not.

Analogical examples in reality include foreign-key constraints and bi-directional relationships in social networking applications (e.g., the *friend_of* relationship, *likes* and *liked_by* association) [8], [9], [13].

Now we consider a case for partial replication.

Example 3. Consider replicated columns R_1, R_2, R_3 that are distributed across different partition nodes; and atomic multi-partition operations that are non-commutative:

1) $a(x)$: $W_a(R_1 = R_1 \times (1 + \%x), R_2 = R_2 \times (1 + \%x), R_3 = R_3 \times (1 + \%x))$;
2) $b(y)$: $W_b(R_1 = R_1 + y, R_2 = R_2 + y, R_3 = R_3 + y)$;
3) $check$: $R_c(R_1, R_2, R_3)$, $assert(R_1 = R_2 = R_3)$.

Consider a schedule $S_3 = \{a(20), b(10), check, check\}$ with overlaps between $a(20)$ and $b(10)$; and $a(20)$ and the first $check$, as shown in Table III. We assume the initial value for three replicas to be 100.

TABLE III: W/W and R/W overlap schedule for replicas

	R_1	R_2	R_3
t_1	$W_{a1}(R_1 \times (1 + 20\%))$	$W_{a2}(R_2 \times (1 + 20\%))$	
t_2	$W_{b1}(R_1 + 10)$	$W_{b2}(R_2 + 10)$	$W_{b3}(R_3 + 10)$
t_3	$R_{c1}(R_1 = 130)$	$R_{c2}(R_2 = 130)$	$R_{c3}(R_3 = 110)$
t_4			$W_{a3}(R_3 \times (1 + 20\%))$
t_5	$R_{c1}(R_1 = 130)$	$R_{c2}(R_2 = 130)$	$R_{c3}(R_3 = 132)$

Both *checks* in Table III return inconsistent states, even at the time points after t_4. Additional restriction is required for applications to ensure the reconciliation of replicas.

It is the Example 2 and 3 that motivate us to propose BASIC, since BASE has the following inevitable limitations:

- The *eventual consistency* makes only liveness rather than safety guarantee, as it merely ensures the system to be consistent in the future [14].
- The *soft state* presents challenges for developers, which requires extremely complex and error-prone mechanisms to reason about the correctness of the system state at each single point [4], [8], [15].
- Additional restriction is required for the *soft state* to converge to *eventual consistency* [16], [17], [18].

III. BASIC PROPERTIES

A. BASIC Specification

In order to resolve the inconsistency of *soft state*, we propose BASIC standing for ***B**asic **A**vailability*, ***S**calability* and ***I**nstant **C**onsistency*.

- *Basic Availability*: the system can response for all continuously operations in a timely manner.
- *Scalability*: the system is able to scale out by adding more resources for increasing workloads.
- *Instant Consistency*: all partitioned data seen by each read operation reflects a consistent state. i.e., each read operation returns the result that reflects write operations which have been executed successfully prior to the read.

Intuitively, a schedule satisfies instant consistency if its equivalent schedule in the view of single operation transaction[2] satisfies the snapshot isolation.

Assume the database is running on partitions $P_1, P_2, \ldots, P_n$. Two types of database operations are considered: (1) READ (i.e., to read a set of data items from multiple partitions) and (2) WRITE (i.e., to write a set of data items into multiple partitions).

A sub-schedule S_i on the partition P_i is a sequence of operations, and a schedule S is the unite of sub-schedules on all partitions. All sub-schedules are considered to be coincident if the order of operations in each sub-schedule does not conflict with each other.

Two sub-schedules S_i, S_j are considered to be coincident, if and only if S_i, S_j hold the same execution history order of respective conflicting operations.

For example, previous Tables I, II and III demonstrate typical schedules on three partitions. The schedule S_1 in Example 1 has three sub-schedules:

$S_{11} = \{a(20), check\}$,

$S_{12} = \{b(10), check\}$,

$S_{13} = \{a(20), b(10), check\}$.

All three sub-schedule in S_1 are coincident.

The schedule S_2 in Example 2 also has three sub-schedules:

$S_{21} = \{a(20), check1, check2\}$,

$S_{22} = \{b(10), check1, check2\}$,

$S_{23} = \{a(20), check1, b(10), check2\}$.

S_{22} and S_{23} are not coincident since the execution order of $\{b(10),\ check1\}$ in S_{22} and S_{23} conflicts. Similarly, it is not difficult to check that the sub-schedules in Example 3 are not coincident either.

Now we are in the position to define instance consistency. A **consistent snapshot** of a partitioned database represents a complete copy of all the data items updated by a schedule of which sub-schedules on all partitions coincide. A schedule of a set of operations satisfies **instant consistency** if any READ operation reads from a consistent snapshot.

By previous analysis, S_1 in Example 1 satisfies instant consistency but not S_2 in Example 2 and S_3 in Example 3.

The instant consistency, and thus the BASIC properties guarantee that all reads will see a consistent snapshot of the partitioned database, which can overcome the deficiencies caused by inconsistent soft state in BASE.

B. CAP Theorem Generalization

The Brewer's CAP Theorem [10], [11] asserts that any networked distributed system can have only two of strong consistency, availability and partition tolerance, that is, a highly available system cannot provide strong consistency guarantees in the presence of network partitions. Though strong transactional consistency models are not achievable with high availability, many weak consistency models are feasible in the highly available distributed environment.

Fig. 1: General Extension of CAP Theorem

Eventual **Consistency** is one of the most commonly used weak consistency models. Many consistency models stronger than eventual consistency provide differentiated, additional guarantees for dependencies among operations. Causal consistency guarantees that the execution in a cluster agrees on the relative ordering of causally related operations [14], [19]. Ordering consistency is an enhanced variation of causal consistency ensuring global ordering of operations. Ordering consistency provides the monotonicity guarantee of both read and write operations to each record [8], [20]. Causal consistency and ordering consistency are stronger than eventual consistency. Furthermore, it is not difficult to see that instant consistency is stronger than causal consistency and ordering consistency in that a system that guarantees instant consistency also guarantees causal consistency and ordering consistency, but not vice versa.

Availability generally guarantees a response from each non-failing server. High availability enforces the return value of read includes the effect of the last completed write to a data item as required in linearizability. Write operation will always successfully commit eventually if the client does not explicitly abort it [12]. Basic availability is relaxed where read can return a version out of date and internal abort can be employed to guarantee continuous response rather than blocking. Basic availability ensures more liveness guarantee for operations at the cost of safe response to operations [21].

Partition tolerance guarantees consistency is maintained even when network failures prevent some servers from communicating with others. It depends on the policy of update manipulation consisting of update initiation (write-to-one, write-to-many) and update propagation (eager propagation, lazy propagation) [22], [23]. In the write-to-one initiation, all updates are firstly sent to one designate node. This node resolves all conflicts, determines the order to perform updates and broadcasts to other nodes with order preserving [24]. The propagation can be eager (i.e., each partition waits to commit until all replicas have received the update, ensuring consistent replicas) or be lazy (i.e., the update on each partition is treated as if it were completed before being applied to all replicas). The write-to-many mechanism is to have W replicas acknowledged the write, thus allowing the write to survive

[2]Note that by the view of single operation transaction, we mean "start transaction" and "commit" are added before and after each and every database operation respectively.

W-1 replica failures [4]. The system propagates updates to the subset of replicas eagerly and the rest lazily. Quorum-based protocol [25] is such a typical model of this pattern. Quorum-based protocol is intrinsically partition tolerant; since it depends on majority quorums, operations in any partition that contains a majority quorum will succeed.

We classify a wide array of consistency and availability models. In doing so, we extend the current understanding of CAP theorem by characterizing precisely different degree of dimensions that can be achieved rather than simply what cannot be done. More specifically, in the CAP theorem, it is required to pick only two of the three properties. Correspondingly, as shown in Figure 1, the primary purpose of the generalized CAP theorem is to find the optimal three properties. Given the general extension of CAP theorem, a system achievable in BASIC (the inner dashed line triangle) provides stricter consistency level than BASE (the outer solid line triangle).

IV. Timestamp-based Formula Protocol for BASIC properties

A. Timestamp-based Formula Protocol Overview

The Timestamp-based Formula Protocol (TFP) is a highly scalable protocol that implements multi-version scheme on multi-partition operations with the instant consistency. As in the typical non-distributed multiversion concurrency control protocol [26], version visibility for operations is determined by associating with each version a scalar monotonic timestamp. Moreover, TFP introduces novel distributed timestamp management schemes with the following optimization mechanisms:

- Logical formula caching simply stores transformation formulas associated with each updated data item on each partition, instead of storing multiple copies of multiple versions of updated data items.
- Dynamic timestamp ordering is used to increase the concurrency degree and reduce unnecessary blocking.
- Caching and delaying the update operations by formulas within the commitment protocol guarantees the atomicity of multi-partition operations on the post-images.
- Partition independence ensures that a client never contacts unrelated partitions that its operation does not access.

The logical transformation formula caching approach has the advantages over storing actual multiple versions of data items mainly due to the following reasons. The conventional implementation of multiversion uses the fine-grained page as the minimal unit in the memory [27], [28]. The page size can affect the latency since the bandwidth can only be fully utilized when data is flushed in pages of relatively large size(e.g. 4KB, 8KB) [28]. Multiple versions of different data items are clustered into pages to save page space. Complex and error-prone mechanism is required to ensure the multi-partition update operation does not overwrite each other even though they access different data items [27], [28]. Thus, the manipulation based on the unit of formula can reduce the complexity of storing multiple versions on the page-level.

In addition, formula enables us to use commutative conflict-free operations [29] such as increment/decrement[3], which is much easier than if other wise, in terms of much less conflict potential. For non-key updates, storing multiple versions need to maintain physical copies of numerous rows [30], but the formula protocol still uses one simple formula. This will significantly reduce overhead of multiple versions of all update data items and simplify the garbage collect process.

The commit order of operations in the conventional timestamp protocol corresponds with the static timestamp initially assigned to each operation. TFP allows an equivalent dynamic schedule where operations with older (smaller) timestamp can read the data item updated with a later (larger) timestamp on condition that the instant consistency is respected. The dynamic timestamp ordering can avoid unnecessary blocking or waiting in order to increase the degree of concurrency. Formulas no longer being used can also be cleared as early as possible, similar to the conventional multi-version protocol that removes any obsolete version of data as soon as it is not needed.

The write operation in TFP relies on a genuine atomic commit protocol that is a variation of the two-phase commit protocol combined with the total order multicasting. The two-phase commit protocol is used to validate each write operation and guarantee the atomicity. The total order multicasting facilitates to preserve the order the commit of write operation among all the replicas of each data item [12], [31]. At the first PREPARE phase, each write operation can be submitted to any replica. At the second FORCE-WRITE phase, the write is sent to all active replicas using the total order multicasting. All replicas serialize in the identical way and completely executed at each replica.

Partition independence ensures that one client's operations only contact partitions that its operation accesses. And if a client can contact the partitions responsible for each data item the operation accesses, the operation will proceed commit. The partition independence can reduce the work burden of partitions that are not directly involved in an operation's execution. And it is important in the presence of partial failure that prevents one client's operation from causing another to block. Now, we present the formal TFP in detail.

B. Timestamp-based Formula Protocol

Each atomic multi-partition operation op is assigned a unique monotonically increasing timestamp, $TS(op)$, when it is initiated. To guarantee partition independence, the TFP maintains a list of participating partitions for each active operation op, denoted as $P(op)$. $P(op)$ involves only the partitions that maintain replicas of the data op accessed. With each data item x, on each relevant partition P_i, the following pieces of information are stored:

(1) $lrt(x, Ni)$: the largest timestamp of active read operation on x on the partition P_i;

(2) $list(x, P_i)$: the list of update formulas in the form: $uf(x, op_1, P_i), \ldots, uf(x, op_n, P_i)$. Each $uf(x, op_j, P_i)$ repre-

[3]The execution order of operations does not affect the result. Thus we consider such operations conflict-free even though they write the same data item.

sents an update formula on x by operation op_j on the partition P_i, and $TS(op_1) < TS(op_2) < \cdots < TS(op_n)$.

Initially, $lrt(x, Ni)$ and $list(x, P_i)$ are set to 0 and $\emptyset$ respectively.

Read operation on a data item x on partition P_i, $read(x, P_i)$, needs to check whether x has already been updated, returning in this case the value updated by the write formulas. Next, the version visible by $read(x, P_i)$ is determined, as in conventional MVCC algorithms, by selecting the most recent version having timestamp smaller than $TS(read(x, P_i))$. Particularly, $read(x, P_i)$ will perform the following action:

R. Let $v_0(x)$ be the value of x on the disk of P_i, and $uf(x, op_{u1}, P_i), \ldots, uf(x, op_{um}, P_i)$ be the list of update formulas in $list(x, P_i)$ such that: (1) $TS(op_{um}) < TS(R(x, P_i))$ and (2) $TS(op_{u(m+1)}) > TS(R(x, P_i))$.

Let $v_1(x)$ be the value obtained by applying $uf(x, op_{u1}, P_i)$ on $v_0(x)$, $v_2(x)$ be the value obtained by applying $uf(x, op_{u2}, P_i)$ on $v_1(x)$ and so on, $v_{um}(x)$ be the value obtained by applying $uf(x, op_{um}, P_i)$ on $v_{u(m-1)}(x)$. $v_{um}(x)$ is the value to be retrieved by $R(x, P_i)$. Essentially, $read(x, P_i)$ retrieves the value that is obtained by sequentially applying all update formulas on x issued by operations with older timestamps.

In order to enforce the correct tracking of the read-write dependency, the protocol records $read_by(op_{uk}, x, R(x, P_i))$ in metadata for all op_{uk} in $list(x, P_i)$ such that $1 < TS(op_{uk}) < TS(R(x, P_i))$, indicating the value x updated by op_{uk} is read by $R(x, P_i)$.

If there exists no $read_by$ fact, $R(x, P_i)$ can return immediately; otherwise, $read_by$ needs to wait the completion of the op_{uk}. In case op_{uk} is successfully done, it will be ordered before $R(x, P_i)$ according to the timestamp and the version of x created by op_{uk} can be visible to $R(x, P_i)$. If op_{uk} aborted, the formula $uf(x, op_{uk}, P_i)$ will be eliminated so that $R(x, P_i)$ can not see the updates.

At last, if $TS(R(x, P_i)) > lrt(x, P_i)$, $TS(R(x, P_i))$ is assigned to $lrt(x, P_i)$.

TFP write operations proceed in a two-phase protocol: a first round of communication places each time-stamped write operation on its respective partitions. In this PREPARE phase, the timestamp of the operation is checked, and update formulas with dependency facts are added to each data item's local metadata. A second round of communication takes the common action of operation on all $P(op)$. In this FORCE-WRITE phase, each partition updates the data item according to formulas or revokes the operation by dropping the formulas directly.

When a write operation is received on partition P_i in the PREPARE phase, $W(x, P_i)$ will perform the following action:

W1. If $TS(W(x, P_i)) < lrt(x, Ni)$, there must exist at least one $read$ operation, which should follow $W(x, P_i)$ according to the serial timestamp order, has read the value of x before $W(x, P_i)$. Thus $W(x, P_i)$ is too old to write x, and must revoke and retry.

W2. If $TS(W(x, P_i)) > lrt(x, P_i)$, the write operation is processed as:

(a) Adding an update formula of $W(x, P_i)$ into $list(x, P_i)$;

(b) There must exist another operation $R_k(x, P_i)$ such that:

1) $TS(R_k(x, P_i)) < TS(W(x, P_i))$;
2) $R_k(x, P_i)$ has read the value of x before.

Recording a fact $read_b4(R_k(x, P_i), W(x, P_i))$ in the metadata for all $R_k(x, P_i)$ such that $1 < TS(R_k(x, P_i)) \leq lrt(x, P_i)$, indicating $R_k(x, P_i)$ has read the data item x before $W(x, P_i)$. $W(x, P_i)$ must wait for commit until $R_k(x)$ is done.

When all partitions in $P(op)$ have proceeded op in the PREPARE phase, one of the following action will be taken in the FORCE-WRITE phase:

V. Revoke(*op*). If there exists any $op(x, P_i)$ on $P_i \in P(op)$ such that $op(x, P_i)$ is forced to revoke in **W1** on P_i. The protocol performs revoke on all partitions in $P(op)$ by removing all relevant stored facts $read_by$, $read_b4$ and update formulas uf involving op in $list(x, P_i)$ directly.

F. Force-write(*op*). If there exists **no** $op(x, P_i)$ on $P_i \in P(op)$ that is forced to revoke in **W1** or any $read_b4(R_k(x, P_i), op(x))$ fact, force-write formula issued by $op(x)$ on all partitions in $P(op)$.

When $op(x)$ terminates (revoke or force-write), operations that are waiting for $op(x)$ will resume and all relevant metadata facts $read_by$, $read_b4$ and update formulas uf involving op are removed.

C. Dynamic Timestamp Ordering

In the classical multi-version timestamp protocol, the execution order of operations conforms with the timestamp initially assigned to each operation. This mechanism is considered to be **static**. However, our protocol respects the initial timestamp ordering while permitting an equivalent schedule that differs from the static timestamp ordering, as long as it ensures instant consistency. This is called **dynamic** timestamp ordering.

Now we use the previous examples to demonstrate how TFP achieves instant consistency with dynamic timestamp ordering.

Consider the schedule $\{a(20), b(10), check\}$ in Example 2 and the timestamp for each operation is: $\{TS(a(20)) = 201, TS(b(10)) = 202, TS(check) = 203\}$.

Actions performed on each partition are illustrated in Table IV. At t_1, formulas issued by $a(20)$ on partition P_1 and P_2 are stored in the list of update formulas based on the writing rule **W2**. At t_2, formulas issued by $b(10)$ on partition P_1, P_2 and P_3 are stored based on the writing rule **W2**, and since there exists no stored facts $read_by$ or $read_b4$ associated with $b(10)$, formulas issued by $b(10)$ are then allowed to be force-written to update values for each column according to the rule **F**. Even though the timestamp of $b(10)$ (=202) is larger than the timestamp of $a(20)$ (=201), $b(10)$ will take effect on the value without waiting for $a(20)$.

This dynamic ordering does not comply with the timestamp ordering associated with each operation, but it does permit

an equivalent schedule satisfying instant consistency. The dynamic timestamp ordering avoids unnecessary blocking and thus increases the degree of concurrency as well as reduces overhead because it cleans the unneeded formulas as early as possible.

At t_3, $check$ is suspended due to the existence of stored fact $read_by$. At t_4, $a(20)$ is revoked since $TS(W_{a3}) = 201 < lrt(H, N_3) = 203$, according to the writing rule **W1**. Upon the removal of formula and $read_by$ facts of $a(20)$, $check$ suspended at t_3 is resumed by returning a consistent state ($R_{c1} = R_{c2} = R_{c3}$).

TABLE IV: W/W and R/W overlap schedule with dynamic timestamp ordering

schedule				action	value		
	P_1	P_2	P_3		R_1	R_2	R_3
t_1	W_{a1}	W_{a2}		$\mathbf{W2}: uf(L, W_{a1}, N_1) \rightarrow list(L, N_1)$ $\mathbf{W2}: uf(S, W_{a2}, N_2) \rightarrow list(S, N_2)$	100	100	100
t_2	W_{b1}	W_{b2}	W_{b3}	$\mathbf{W2}: uf(L, W_{b1}, N_1) \rightarrow list(L, N_1)$ $\mathbf{W2}: uf(S, W_{b2}, N_2) \rightarrow list(S, N_2)$ $\mathbf{W2}: uf(H, W_{b3}, N_3) \rightarrow list(H, N_3)$ **F**: Forcewrite(b) **(dynamic ordering)**	110	110	110
t_3	R_{c1}	R_{c2}	R_{c3}	**R**:$R_{c1}(uf(L, W_{a1}, N_1)), read_by(W_{a1}, L, R_{c1})$ **R**:$R_{c2}(uf(S, W_{a2}, N_2)), read_by(W_{a2}, S, R_{c2})$ **R**:$R_{c3}(H = 110), lrt(H, N_3) = 203$			
t_4			W_{a3}	**W1**: Revoke(W_{a3}), **V**: Revoke(a) Remove: $uf(L, W_{a1}, N_1), read_by(W_{a1}, L, Rc_1)$ $uf(L, W_{a2}, N_2), read_by(W_{a2}, L, Rc_2)$ $R_{c1}(L = 110), R_{c2}(S = 110), R_{c3}(H = 110)$	110	110	110

D. Basic Availability Guarantee

TFP operates in a distributed environment, which needs to deal with partial failure and network partitions. Partition independence ensures that failed clients do not cause other clients to fail. This provides fault tolerance and availability as long as clients can access relevant working partitions.

TFP writes use a two-phase atomic commit protocol, which will always complete the operation except when every relevant partition has performed the first PREPARE phase, but none of them has performed the second FORCE-WRITE phase. If a partition P_r accessed by an operation *op*, has timed out while waiting during the PREPARE phase, P_r can abort the operation and safely discard its formula, since P_r can be certain to never force-write *op* in the future. If a partition P_r has performed the PREPARE phase but times out while waiting in the FORCE-WRITE phase, P_r can check the status of operation *op* on any other partitions to determine the outcome of the operation. If another partition has executed force-write for *op*, then P_r can force-write *op*. The coordinator unilaterally aborts the operation if it times out while waiting for replies for the prepare phase. Upon dealing with failure of the coordinator, Paxos [32] or other consensus based abstractions [33] can be applied to replicate the state of the coordinator across the replicas.

Failures of write operations can lead to aborting itself, and will not lead to blocking the execution of read operations. However, the read operation may return stale but consistent snapshot of data. Staleness return and aborting mechanism can facilitate to achieve basic availability.

E. Instant Consistency Guarantee

Since TFP is a variation of multi-version implementation. We still stick to the conventional notations in multi-version protocol for the guarantee proof. We assume $w_i(x_i)$ denotes a new version x_i issued by the write operation w_i; and $r_i(x_j)$ denotes the read operation op_i on the data version x_j. The multiversion history H_x defines a total order $\prec_x$ for each data item x on all partitions. The total version order $\prec$ is the union of the $\prec_x$ in H_x [26].

Given a multiversion history H and a total version order $\prec$, a direct multiversion graph (DMG) can be constructed by setting a vertex for each operation op_i in H, and a direct edge $op_i \rightarrow op_j$ if one the following statements hold on any partition:

(a) *write-read* dependency: $w_i \rightarrow r_j$, w_i produces a version of x_i, and r_j reads x_i.

(b) *read-write* dependency: $r_i \rightarrow w_j$, r_i reads a version of x_k, and w_j produces a later version of x_j that overwrites x_k.

(c) *write-write* dependency: $w_i \rightarrow w_j$, w_i produces a version of x_i, and w_j produces a later version of x_j.

A multiversion history H can guarantee instant consistency if and only if its direct graph does not contain any oriented cycle. Our proof is based on establishing a mapping between each vertex in the direct graph and its timestamp. We prove the graph to be acyclic by illustrating that for each edge $op_i \rightarrow op_j$ in the DMG, TFP can guarantee that $TS(op_i) < TS(op_j)$.

- In the case (a), $w_i \rightarrow r_j$ means that r_j has read a version produced by w_i. If w_i with smaller timestamp arrives later than r_j, w_i will abort and retry according to the writing rule **W1**. Otherwise the $read_by$ fact regulates $TS(w_i) < TS(r_j)$ in the reading rule **R**.
- In the case (b), $r_i \rightarrow w_j$ means that a stored fact $read_b4(r_i, w_j)$ will be generated and stored. The $read_b4$ fact in **W2** guarantees that w_j with a larger timestamp can not be issued before r_i with a smaller timestamp. Therefore, we can have $TS(r_i) < TS(w_j)$.
- In the case (c), if there exists any r_k such that $w_i \rightarrow r_k \rightarrow w_j$, we can have $TS(w_i) < TS(r_k) < TS(w_j)$ based on case (a), (b). Otherwise, the dynamic timestamp ordering may alternate the timestamps to guarantee $TS(w_i) < TS(w_j)$.

V. Performance Evaluation

In this section, we are going to conduct experiments on the performance evaluation of BASIC. The main purpose of experiments includes:

1) What are performance comparisons between systems with BASE and systems with BASIC?
2) Is BASIC able to be achieved with high scalability?
3) How much will the latency of response increase for systems with BASIC?

A. Experimental Setup and Benchmark

All the experiments use a collection of, up to 12, commodity servers connected with a Gigabit LAN with low network latency. More specifically, each server has dual quad-core Intel Xeon CPUs, 32GB of main memory, SATA disks configured in RAID0, running Linux Ubuntu 12.04 LTS.

We applied the industrial standard Yahoo! Cloud Serving Benchmark (YCSB) [34] to measure performance, which is a data serving benchmark widely used to measure throughput and latency for each set of operations with varying distribution.

B. Experimental Performance

We have implemented a large-scale database management system RubatoDB [35] based on TFP protocol, to support both BASIC and BASE. In our experiments, we compare four different system configurations including: RubatoDB with BASIC, RubatoDB with BASE, HBase[4] with BASE, and Cassandra[5] with BASE.

According to the YCSB specification, we define a multi-column structure for each data item, which consists of one key column and 24 data columns. We use column partitioning and distribute columns across multiple servers evenly. We test the read-intensive workload (including 90% read operations and 10% write operations) and the write-intensive workload (including 50% read operations and 50% write operations). Each operation accesses a random set of fields of a single data item across the servers. The size of the data is set to 100 million 1KB records for each node, resulting around 120GB of raw data per node. In the experiments, a continuous mixed workload is submitted into the system, and the benchmark then measures the performance in terms of throughput (i.e., the number of operations per second) and latency of operations in milliseconds.

1) Inconsistency Ratio: We first evaluate the inconsistent soft states ratio of RubatoDB with BASE on the system with the number of servers as 1, 2, 4, 8 and 12.

The inconsistent state of read operation is detected by checking the update formula list on each participating nodes. If the formulas employed by the read operation do not coincide, then the state returned is inconsistent. Table V shows that the inconsistent soft state ratio on the 1-node system is lower than 0.1%; however, the ratio increases gradually with the growth of the system size, which is as high as 0.62% on the system with 12 nodes. This increasing tendency of inconsistency indicates that more efforts are required to deal with the soft state as the system scales out. As the system size becomes large, it is clear that uncertainties in message communications may prevent all relevant partitions from drawing accurate decision about the instantaneous global state of the system. Another source of inconsistency may also arise if all relevant partitions fail to execute identical reactions, even though each partition evaluate the same predicate.

2) Scalability: We then compare the performance in terms of throughput and latency as per the YCSB specification. In this group of experiments, we set up three replicas for each data item, that is using HBase (replication factor=3) and Cassandra (replication factor=3, consistency level=ANY).

TABLE V: Soft State Ratio

Nodes Number	1	2	4	8	12
Inconsistency	%0.08	%0.15	%0.32	0.51%	0.62%

Our results show that the throughput and latency of RubatoDB with BASE is comparable with Cassandra and HBase, but RubatoDB with BASIC has potentially lower the performance. The throughput comparison for read-intensive workload is plotted in Figure 2(a) and the corresponding latency is shown in Figure 2(b). The throughput decreases around 10% and the latency is nearly doubled due to the extra cost for BASIC. In the write-intensive workloads, the potential of operation overlap increases, causing a higher cost for ensuring BASIC. As illustrated in Figure 2(c), (d), the throughput reduces 25% on 12 nodes with the latency increasing nearly one order of magnitude. The main source of the latency comes from the presence of abundant dependencies between write and read operations in the write-intensive workloads. The numerous stored facts (i.e. $read_by$, $read_b4$) can cause one operations to wait for commit until another operation is done[6].

Though there is decline for the performance, RubatoDB with BASIC properties still preserves near-linear scalability with increasing throughput and flat latency, same as systems with BASE.

3) Increasing Replication: The following experiments explore the response latency under increasing workloads of partial and full replication. We increase the replication factor from 2, 4, 6 for partial replicas, to 12 for full replicas. Figure 3 shows the latency of response time for 12-node RubatoDB with BASIC under various replication factors.

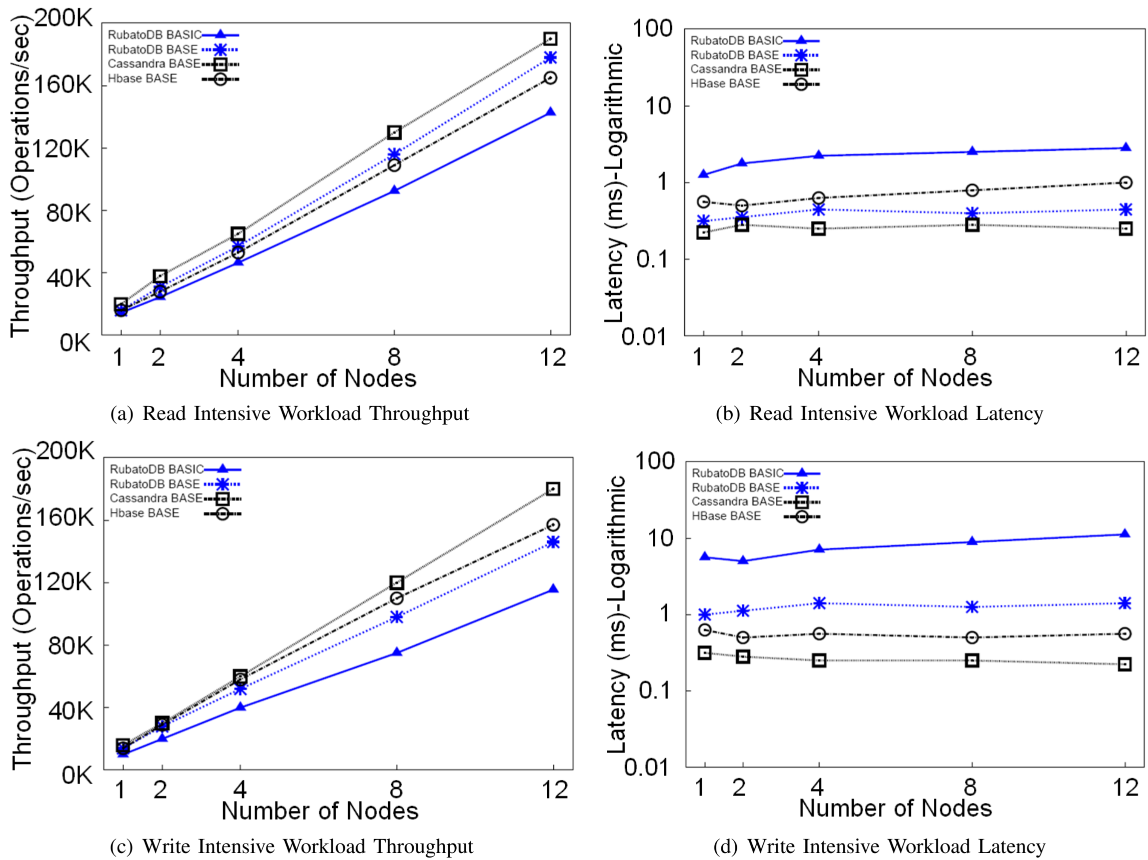
The partial replication results (2, 4, and 6 replicas per data item) show that the smaller the number of replicas is, the higher throughput (operation per second) can be achievable with flat low latency. Since read-intensive workload induces less overhead for synchronization, the response time of full replication stabilizes for 70% of maximum workloads, and partial replication can achieve 85% of the maximum workloads before the latency rockets (as shown in Figure 3(a)). The latency of replication does not decline the performance of the system quite much, since that results of any read distributed across replications will be consistent.

In the write-intensive workload, a "safe" write operation requiring multiple replications acknowledge the write before it returns will incur higher latency, and/or may time out in presence of network partition. Requiring confirmation of each write operation has replicated to all partitions will effectively guarantee that those replicated partitions have caught up with the timestamp of this write. The partial replication can scale significantly better than full replication (as shown in Figure 3(b)), but still the more replicas are used, the more dependency stored facts are involved causing higher latency. This exhibits that the commit latency of operations increases

[4]http://hbase.apache.org/

[5]http://cassandra.apache.org/

[6]RubatoDB is currently implemented as a research system in the university lab, and its performance can still be improved comparing with other systems.



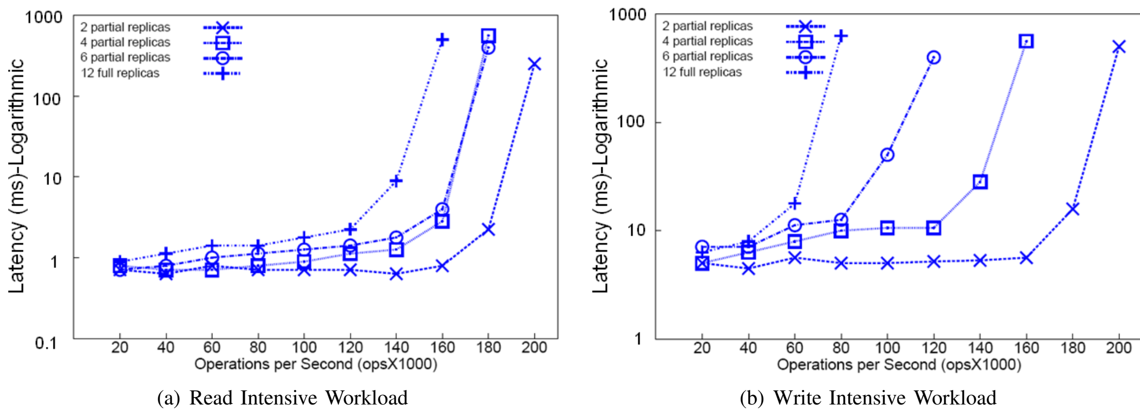
(a) Read Intensive Workload Throughput

(b) Read Intensive Workload Latency

(c) Write Intensive Workload Throughput

(d) Write Intensive Workload Latency

Fig. 2: Comparison between BASIC and BASE



(a) Read Intensive Workload

(b) Write Intensive Workload

Fig. 3: Comparison of Latency for Replication

with high contention. The clients wait longer for the majority response for either aborts or commits. When the number of operations per second is low ($< 40,000$), the latency does not vary as much as the replications increases. This is because when the contention is not that high, the replicated commit latency can be maintained low with more rollback operations; an abort operation exhibits lower latency due to its early termination compared to a committed operation.

In summary, for the read-intensive workload or the write-intensive workload without requiring extremely low response latency, the cost induced by BASIC is acceptable comparing with the extra efforts needed to manipulate inconsistent soft states for BASE. That is, BASIC pays a reasonable price for a higher consistency than BASE.

VI. Related Work

One of the challenges in designing and implementing large-scale data management systems is how to achieve high scalability without sacrificing consistency. A broad spectrum

of consistency guarantees have been explored at varying costs of scalability and availability.

Serializability. At the strong end of the consistency spectrum is the transactional ACID. Serializability is the highest isolation level. A range of techniques can enforce serializability in distributed databases such as distributed locking [1], [32] and optimistic protocol [36]. However, these mechanisms may limit the scalability. Serializability has been known to be unachievable under the environment where network partition or partial failure can be normal rather than rare [37], [15], [38], [39].

Snapshot isolation is a multi-version concurrency control protocol based on optimistic reads and writes. A data snapshot is taken when the snapshot transaction starts, and remains consistent for the duration of the transaction. However, reading from a snapshot means that a transaction never sees the partial results of other concurrent transactions and write skew may occur [40]. TFP deals with read/write conflicts using a list of metadata facts per data item, similar to the recent work [40], [41] on snapshot isolation where dependencies are detected for serialization.

The lower end of the spectrum consists of various **weak consistency** models. Weak consistency is an alternative set of transactional semantics that are still useful for distributed multi-partition semantics, but do not violate requirements for high availability or low latency.

Eventual consistency, one of the fundamental weak consistency models, guarantees that if no new updates are made to a given data item, eventually all accesses to that data item will return the last updated value [18], [42].

Recent systems proposals such as Eiger [43], and Bolt-on Causal Consistency [19] provide **causal consistency** guarantees with varying availability. Causal consistency ensures execution ordering of operations which are causally related [44]. Implementation of causal consistency usually involves dependency tracking [14], [19], [43]. Different from causal consistency, the dependency tracking is restrained within the minimal interval so that the tracking complexity is minimized in the instant consistency.

Instead of merely ensuring partial orderings between causality dependent operations, **ordering consistency** is an enhanced variation of causal consistency ensuring global ordering of operations. The monotonic ordering guarantee can be enforced by ensuring that write operation can be accepted only if all writes made by the same user are incorporated in the same node [45]. It can be achieved by designating one node as the primary node for every record; and then all updates to that record are first directing to the primary node [8], [14], [46].

We can categorize prevailing systems into the consistency taxonomy, as shown in Figure 4.

Spanner [37], [15], Megastore [1] and Spinnaker [32] provide serializable transactions using strict two-phase locking protocol and two-phase commit, running on top of the Paxos-replicated log mechanism for fault-tolerance. Such layered combination, in which protocols that guarantee transactional atomicity and isolation are separated from the mechanism that guarantees fault-tolerant replication, has the advantages of modularity and clear semantics.

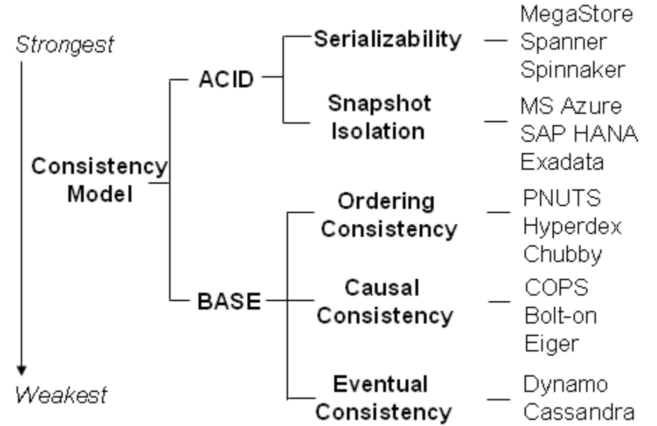


Fig. 4: Taxonomy of Consistency Model

Microsoft Azure [47] and SAP HANA [48] relies on MVCC as the underlying concurrency control mechanism to synchronize multiple writers, and provide distributed snapshot isolation.

Hyperdex [20] provides ordering consistency with a chaining structure, in which nodes are arranged into a value-dependent chain. PNUTS [8] provides a per-record timeline model that preserves ordering consistency by introducing a pub/sub message broker. Chubby [49] is a distributed locking mechanism with a leasing agreement, based on which mutations are applied in the same grant order.

COPS [14], [46] and Eiger [43] track dependencies on versions of keys or operations to enforce causal consistency. An operation does not take effect until verifying that the operation's dependencies are satisfied. Bolt-on [19] provides a shim layer that upgrades the eventual consistency of an underlying general-purpose data store to the causal consistency for the application.

Dynamo [4] and Cassandra [5] provide eventual consistency for allowing applications with "always writeable" property, that is, write operations can always be accepted by any node. Vector clock, also named as version vector, is associated with data to determine the eventual consistent state during reconciliation.

BASIC is inspired by recent work on PACELC [50] which expands CAP theorem by considering the relationship between weak consistency and low latency, and HAT [12] which exposes a broad design space of highly available distributed systems. We believe the design space for the consistency model is not limited to merely ACID which is too strong and BASE which is too weak, but rather that there is a spectrum between these two extremes, and it is possible to build a set of semantics guarantees combining different consistency models and availability for various use case requirements.

VII. Conclusion

In this paper we have introduced BASIC (Basic Availability, Scalability, Instant Consistency), an alternative consistency model stronger than BASE while weaker than ACID. To compare BASIC with BASE, we extend the CAP theorem

by finding the optimal degree on properties of consistency, availability and partition tolerance, rather than picking only two of them. We also presented a timestamp-based formula protocol that enforces instant consistency with linear scalability and fault tolerance. Novel techniques including logical formula caching, dynamic timestamp ordering, partitioning independence and genuine atomic commit are employed for efficiency and effectiveness. Extensive experiments on industrial standard benchmark verify that BASIC can be achieved with high scalability, and the limited overhead induced by BASIC is acceptable comparing with the extra efforts required to maintain the inconsistency of soft states.

REFERENCES

[1] J. Baker, C. Bond, and etc, "Megastore: Providing scalable, highly available storage for interactive services," in *CIDR*, 2011, pp. 223–234.

[2] F. Chang, J. Dean, S. Ghemawat, and etc, "Bigtable: a distributed storage system for structured data," in *TOCS*, vol. 26, 2008, pp. 1–26.

[3] S. Das and etc, "G-store: a scalable data store for transactional multi key access in the cloud," in *SoCC*, 2010, pp. 163–174.

[4] G. DeCandia, D. Hastorun, and etc, "Dynamo: Amazon's highly available key-value store," in *SOSP*, 2007, pp. 205–220.

[5] A. Lakshman, "Cassandra: a decentralized structured storage system," in *SIGOPS*, 2010, pp. 35–40.

[6] R. Kallman and etc, "H-store: a high-performance, distributed main memory transaction processing system," in *VLDB*, 2008, pp. 1496–1499.

[7] A. Thomson, T. Diamond, and etc, "Calvin: fast distributed transactions for partitioned database systems," in *SIGMOD*, 2012, pp. 1–12.

[8] B. F. Cooper and etc, "Pnuts: Yahoo!'s hosted data serving platform," in *VLDB*, 2008, pp. 1277–1288.

[9] L. Qiao, K. Surlaker, and etc, "On brewing fresh espresso: Linkedin's distributed data serving platform," in *ICMD*, 2013, pp. 1135–1146.

[10] E. A. Brewer, "Towards robust distributed systems (abstract)," in *PODC*, 2000, pp. 7–10.

[11] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[12] P. Bailis, A. Davidson, and etc, "Highly available transactions: Virtues and limitations," *PVLDB*, vol. 7, no. 3, 2013.

[13] N. Bronson and etc, "Tao: Facebook's distributed data store for the social graph," in *Proceedings of the 2013 USENIX conference on Annual Technical Conference*, 2013, pp. 49–60.

[14] W. Lloyd and etc, "Don't settle for eventual: scalable causal consistency for wide-area storage with cops," in *SOSP*, 2011, pp. 401–416.

[15] J. Shute, B. Samwel, and etc, "F1: A distributed sql database that scales," in *VLDB Endowment*, 2013, pp. 1068–1079.

[16] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee, "Replicated abstract data types: Building blocks for collaborative applications," vol. 71, no. 3, 2011, pp. 354–368.

[17] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*, 2011, pp. 386–400.

[18] W. Vogels, "Eventually consistent," in *ACM Queue*, vol. 6, no. 6, 2008, pp. 14–19.

[19] P. Bailis and etc, "Bolt-on causal consistency," in *SIGMOD*, 2013, pp. 761–772.

[20] R. Escriva and etc, "Hyperdex: a distributed, searchable key-value store," in *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, 2012, pp. 25–36.

[21] M. Wiesmann and etc, "Understanding replication in databases and distributed systems," in *ICDCS*, 2000, pp. 464–474.

[22] B. Kemme and G. Alonso, "A new approach to developing and implementing eager database replication protocols," *TODS*, vol. 25, no. 3, pp. 333–379, 2000.

[23] S. Wu and B. Kemme, "Postgres-r (si): Combining replica control with concurrency control based on snapshot isolation," in *ICDE*, 2005, pp. 422–433.

[24] X. Défago and etc, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, 2004.

[25] L. Alvisi, D. Malkhi, E. Pierce, and M. K. Reiter, "Fault detection for byzantine quorum systems," in *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 9, Sep. 2001, pp. 996–1007.

[26] R. Thomas, "A solution to the concurrency control problem for multiple copy databases," in *Digest IEEE COMPCON*, 1984, pp. 56–62.

[27] S. Blott and etc, "An almost-serial protocol for transaction execution in main-memory database systems," in *PVLDB*, 2002, pp. 706–717.

[28] M. Brantner, D. Florescu, and etc, "Building a database on s3," in *SIGMOD*, 2008, pp. 251–264.

[29] P. Alvaro, N. Conway, and etc, "Consistency analysis in bloom: a calm and collected approach." in *CIDR*, 2011, pp. 249–260.

[30] K. Manassiev and etc, "Exploiting distributed version concurrency in a transactional memory cluster," in *SIGPLAN*. ACM, 2006, pp. 198–208.

[31] N. Schiper and etc, "P-store: Genuine partial replication in wide area networks," in *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*, 2010, pp. 214–224.

[32] J. Rao and etc, "Using paxos to build a scalable, consistent, and highly available datastore," *PVLDB*, vol. 4, no. 4, pp. 243–254, 2011.

[33] J. Gray and L. Lamport, "Consensus on transaction commit," *TODS*, vol. 31, no. 1, pp. 133–160, 2006.

[34] B. F. Cooper and etc, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010, pp. 143–154.

[35] L. Yuan, L. Wu, J. You, and Y. Chi, "Rubato db: a highly scalable staged grid database system for oltp and big data applications," in *CIKM*, 2014.

[36] P.-A. Larson, S. Blanas, and etc, "High-performance concurrency control mechanisms for main-memory databases," in *PVLDB*, vol. 5, no. 4, 2011, pp. 298–309.

[37] J. C. Corbett and etc, "Spanner: Google's globally-distributed database," in *OSDI*, 2012, pp. 251–264.

[38] H. Yu and A. Vahdat, "Minimal replication cost for availability," in *PODC*, 2002, pp. 98–107.

[39] H. Yu and etc., "The costs and limits of availability for replicated services," in *Trans. Comput. Syst.*, vol. 24, no. 1, 2006, pp. 70–113.

[40] A. Fekete, D. Liarokapis, and etc, "Making snapshot isolation serializable," in *TODS*, vol. 30, no. 2, 2005, pp. 492–528.

[41] M. A. Bornea and etc, "One-copy serializability with snapshot isolation under the hood," in *ICDE*, 2011, pp. 625–636.

[42] D. Pritchett, "Base: An acid alternative," in *ACM Queue*, vol. 6, no. 3, 2008.

[43] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *NSDI*, 2013, pp. 313–328.

[44] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto, "Causal memory: definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995.

[45] Y. Saito and M. Shapiro, "Optimistic replication," in *ACM Comput. Surv.*, vol. 37, no. 1, 2005, pp. 42–81.

[46] S. Burckhardt and etc, "Eventually consistent transactions," in *ESOP*, 2012, pp. 67–86.

[47] D. G. Campbell, G. Kakivaya, and N. Ellis, "Extreme scale with full sql language support in microsoft sql azure," in *SIGMOD*, 2010, pp. 1021–1024.

[48] V. Sikka and etc, "Efficient transaction processing in sap hana database: The end of a column store myth," in *SIGMOD*, 2012, pp. 731–742.

[49] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *OSDI*, 2006, pp. 335–350.

[50] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, vol. 45, no. 2, pp. 37–42, 2012.