Replicated Data Consistency Explained Through Baseball

Doug Terry
Microsoft Research Silicon Valley

Abstract

Data replication is routinely used for highly available and scalable services. However, no standard replication protocols exist since different schemes involve complex trade-offs between consistency, performance, and availability. Some cloud storage services replicate data while providing strong consistency to their clients whereas others have chosen eventual consistency in order to obtain better performance and availability. A broader class of consistency guarantees can, and perhaps should, be offered to clients that read shared data. This paper explains a range of consistency guarantees along with their motivation by way of example: maintaining the score of a baseball game. During a baseball game, different participants (the scorekeeper, umpire, sportswriter, and so on) benefit from six different consistency guarantees when reading the current score. Eventual consistency is insufficient for most of the participants, but strong consistency is not needed either. This example sheds light on the consistency models that are currently offered by cloud providers, what consistency may be offered in the future, and why vendors are offering consistency choices.

1. Introduction

Replicated storage systems for the cloud deliver different consistency guarantees to applications that are reading data. Invariably, cloud storage providers redundantly store data on multiple machines so that data remains available in the face of unavoidable failures. Replicating data across datacenters is not uncommon, allowing the data to survive complete site outages. However, the replicas are not always kept perfectly synchronized. Thus, clients that read the same data object from different servers can potentially receive different versions.

Some systems, like Microsoft's Windows Azure, provide only strongly consistent storage services to their applications [5]. This ensures that clients of Windows Azure Storage always see the latest value that was written for a data object. While strong consistency is desirable and reasonable to provide within a datacenter, it raises concerns as systems start to offer geo-replicated services that span multiple datacenters on multiple continents.

Many cloud storage systems, such as the Amazon Simple Storage Service (S3), were designed with weak consistency based on the belief that strong consistency is too expensive in large systems. The designers chose to relax consistency in order to obtain better performance and availability. In such systems, clients may perform read operations that return stale data. The data returned by a read operation is the value of the object at *some past point in time* but not necessarily the latest value. This occurs, for instance, when the read operation is

directed to a replica that has not yet received all of the writes that were accepted by some other replica. Such systems are said to be *eventually consistent* [12].

Recent systems, recognizing the need to support different classes of applications, have been designed with a choice of operations for accessing cloud storage. Amazon's DynamoDB, for example, provides both *eventually consistent reads* and *strongly consistent reads*, with the latter experiencing a higher read latency and a two-fold reduction in read throughput [1]. Amazon SimpleDB offers the same choices for clients that read data. Similarly, the Google App Engine Datastore added eventually consistent reads to complement its default strong consistency [8]. PNUTS, which underlies many of Yahoo's web services, provides three types of read operations: r*ead-any*, *read-critical*, and *read-latest* [7]. Modern quorum-based storage systems allow clients to choose between strong and eventual consistency by selecting different read and write quorums [4].

In the research community over the past thirty years, a number of consistency models have been proposed for distributed and replicated systems [10]. These offer consistency guarantees that lie somewhere in between strong consistency and eventual consistency. For example, a system might guarantee that a client sees data that is no more than 5 minutes out-of-date or that a client always observes the results of its own writes. Actually, some consistency models are even weaker than eventual consistency, but those I ignore as being less-than-useful.

The reason for exploring different consistency models is that there are fundamental tradeoffs between consistency, performance, and availability [9, 10, 12, 13]. Offering stronger consistency generally results in lower performance and reduced availability for reads or writes or both. The CAP theorem has proven that, for systems that must tolerate network partitions, designers must choose between consistency and availability [5]. In practice, latency is an equally important consideration [1]. Each proposed consistency model occupies some point in the complex space of tradeoffs.

Are different consistencies useful in practice? Can application developers cope with eventual consistency? Should cloud storage systems offer an even greater choice of consistency than the consistent and eventually consistent reads offered by some of today's services?

This paper attempts to answer these questions, at least partially, by examining an example (but clearly fictitious) application: *the game of baseball*. In particular, I explore the needs of different people who access the score of a baseball game, including the scorekeeper, umpire, radio reporter, sportswriter, and statistician. Supposing that the score is stored in a cloud-based, replicated storage service, I show that eventual consistency is insufficient for most of the participants, but strong consistency is not needed either. *Most participants benefit from some intermediate consistency guarantee.*

The outline of this paper is as follows. The next section defines six possible consistency guarantees for read operations. Section 3 presents an algorithm that emulates a baseball game, indicating where data is written and read, and enumerates the results that might be returned when reading the score with different guarantees. Section 4 then examines the roles of various people who want to access the baseball score and the read consistency that each desires. Finally, I draw conclusions from this simple example.

2. Read Consistency Guarantees

While replicated systems have provided many types of data consistency over the past 30 years, and a wide variety of consistency models have been explored in the computer science research community, many of these are tied to specific implementations. Frequently, one needs to understand *how* a system operates in order to understand *what* consistency it provides *in what situations*. This places an unfortunate burden on those who develop applications on top of such storage systems.

The six consistency guarantees that I advocate in this section can be described in a simple, implementation-independent way. This not only benefits application developers but also can permit flexibility in the design, operation, and evolution of the underlying storage system.

These consistency guarantees are based on a simple model in which clients perform *read* and *write* operations to a *data store*. Multiple clients may concurrently access shared information, such as social network graphs, news feeds, photos, shopping carts, or financial records. The data is replicated among a set of *servers*, but the details of the replication protocol are hidden from clients. A write is any operation that updates one or more data objects. Writes are eventually received at all servers and performed in the same order. This order is consistent with order in which write operations are submitted by clients. In practice, the order could be enforced, even for concurrent writers, by performing all writes at a master server or by having servers run a consensus protocol to reach agreement on the global order. Reads return the values of one or more data objects that were previously written, though not necessarily the latest values. Each read operation can request a consistency guarantee, which dictates the set of allowable return values. Each guarantee is defined by the set of previous writes whose results are visible to a read operation. Table 1 summarizes these six consistency guarantees.

Strong Consistency	See all previous writes.
Eventual Consistency	See subset of previous writes.
Consistent Prefix	See initial sequence of writes.
Bounded Staleness	See all “old” writes.
Monotonic Reads	See increasing subset of writes.
Read My Writes	See all writes performed by reader.

Table 1. Six Consistency Guarantees

Strong consistency is particularly easy to understand. It guarantees that a read operation returns the value that was last written for a given object. If write operations can modify or extend portions of a data object, such as appending data to a log, then the read returns the result of applying *all* writes to that object. In other words, a read observes the effects of all previously completed writes.

Eventual consistency is the weakest of the guarantees, meaning that it allows the greatest set of possible return values. For whole-object writes, an eventually consistent read can return any value for a data object that was

written in the past. More generally, such a read can return results from a replica that has received an arbitrary subset of the writes to the data object being read. The term "eventual" consistency derives from the fact that each replica eventually receives each write operation, and if clients stopped performing writes then read operations would eventually return an object's latest value.

By requesting a *consistent prefix*, a reader is guaranteed to observe an ordered sequence of writes starting with the first write to a data store. For example, the read may be answered by a replica that receives writes in order from a master replica but has not yet received some recent writes. In other words, the reader sees a version of the data store that existed at the master *at some time in the past*. This is similar to the "snapshot isolation" consistency offered by many database management systems. For reads to a single data object in a system where write operations completely overwrite previous values of an object, even eventual consistency reads observe a consistent prefix. The main benefit of requesting a consistent prefix arises when reading multiple data objects or when write operations incrementally update an object.

Bounded staleness ensures that read results are not too out-of-date. Typically, staleness is defined by a time period *T*, say 5 minutes. The storage system guarantees that a read operation will return any values written more than *T* minutes ago or more recently written values. Alternative, some systems have defined staleness in terms of the number of missing writes or even the amount of inaccuracy in a data value. I find that time-bounded staleness is the most natural concept for application developers.

Monotonic Reads is a property that applies to a sequence of read operations that are performed by a given storage system client. As such, it is called a "session guarantee" [11]. With monotonic reads, a client can read arbitrarily stale data, as with eventual consistency, but is guaranteed to observe a data store that is increasingly up-to-date over time. In particular, if the client issues a read operation and then later issues another read to the same object(s), the second read will return the same value(s) or a more recently written value.

Read My Writes is a property that also applies to a sequence of operations performed by a single client. It guarantees that the effects of all writes that were performed by the client are visible to the client's subsequent reads. If a client writes a new value for a data object and then reads this object, the read will return the value that was last written by the client (or some other value that was later written by a different client). For clients that have issued no writes, the guarantee is the same as eventual consistency. (Note: In previous papers this has been called "Read *Your* Writes" [11], but I have chosen to rename it to more accurately describe the guarantee from the client's viewpoint.)

These last four read guarantees are all a form of eventual consistency but stronger than the eventual consistency model that is typically provided in cloud storage systems. The "strength" of a consistency guarantee does not depend on when and how writes propagate between servers, but rather is defined by the size of the set of allowable results for a read operation. Smaller sets of possible read results indicate stronger consistency. When requesting strong consistency, there is a single value that must be returned, the latest value that was written. For an object that has been updated many times, an eventually consistent read can return one of many suitable values. Of the four intermediate guarantees, none is stronger than any of the others, meaning that each might have a different set of possible responses to a read operation. In some cases, as will be shown later,

applications may want to request multiple of these guarantees. For example, a client could request both *monotonic reads* and *read my writes* so that it observes a data store that is consistent with its own actions [11].

In this paper, the data store used for baseball scores is a traditional key-value store, popularized by the "noSQL" movement. Writes, also called *puts*, modify the value associated with a given key. Reads, also called *gets*, return the value for a key. However, these guarantees can apply to other types of replicated data stores with other types of read and write operations, such as file systems and relational databases. This is why the guarantees are defined in terms of writes rather than data values. In a system that offers increment, decrement, or append operations, all writes performed on an object contribute to the object's observed value, not just the latest write. For example, consider a bank account to which deposits and withdrawals are performed. Moreover, the guarantees could apply to atomic transactions that access multiple objects, though the examples in this paper do not require atomic updates.

Table 2 shows the performance and availability typically associated with each consistency guarantee. It rates the three properties on a scale from poor to excellent. Consistency ratings are based on the strength of the consistency guarantee as previously defined. Performance refers to the time it takes to complete a read operation, that is, the read latency. Availability is the likelihood of a read operation successfully returning suitably consistent data in the presence of server failures.

Strong consistency is desirable from a consistency viewpoint but offers the worst performance and availability since it generally requires reading from a designated primary site or from a majority of replicas. *Eventual consistency*, on the other hand, allows clients to read from any replica, but offers the weakest consistency. The inverse correlation between performance and consistency is not surprising since weaker forms of consistency generally permit read requests to be sent to a wider set of servers. With more choices of servers that are sufficiently up-to-date, clients are more able to choose a nearby server. The latency difference between accessing a local rather than a remote server can be a factor of 100. Similarly, a larger choice of servers means that a client is more likely to find one (or a quorum) that is reachable, resulting in higher availability.

Each guarantee offers a unique combination of consistency, performance, and availability. Labeling each cell in Table 2 is not an exact science (and I could devote a whole paper to this topic). One might argue that some entry listed as "okay" should really be "good", or vice versa. Providing a model to predict the delivered consistency, performance, and availability for a given read guarantee is difficult since these depend on many factors, including implementation issues, deployment configurations, technology characteristics, and application workloads. For some clients, eventually consistent reads may often return strongly consistent results, and may not be any more efficient than strongly consistent reads [3, 13]. But, the general comparisons between the various consistency guarantees are qualitatively accurate. The bottom line is that one faces substantial trade-offs when choosing a particular replication scheme with a particular consistency model.

Without offering any evidence, I assert that all of these guarantees can be provided as choices within the same storage system. In fact, my colleagues and I at the MSR Silicon Valley Lab have built a prototype of such a system (but that's the topic for another paper). In our system, clients requesting different consistency guarantees experience different performance and availability for the read operations that they perform, even

when accessing shared data. For this paper, let's assume the existence of a storage system that offers its clients a choice of these six read guarantees. I proceed to show how they would be used … *in baseball*.

Guarantee	Consistency	Performance	Availability
Strong Consistency	excellent	poor	poor
Eventual Consistency	poor	excellent	excellent
Consistent Prefix	okay	good	excellent
Bounded Staleness	good	okay	poor
Monotonic Reads	okay	good	good
Read My Writes	okay	okay	okay

Table 2. Consistency, Performance, and Availability Trade-offs

3. Baseball as a Sample Application

For those readers who are not familiar with baseball, but who love to read code, Figure 1 illustrates the basics of a 9-inning baseball game. The game starts with the score of 0-0. The visitors bat first and remain at bat until they make three outs. Then the home team bats until it makes three outs. This continues for nine innings. Granted, this leaves out many of the subtleties that are dear to baseball aficionados, like myself. But it does explain all that is needed for this paper.

Assume that the score of the game is recorded in a key-value store in two objects, one for the number of runs scored by the "visitors" and one for the "home" team's runs. When a team scores a run, a read operation is performed on its current score, the returned value is incremented by one, and the new value is written back to the key-value store.

```
Write ("visitors", 0);
Write ("home", 0);
for inning = 1 .. 9
   outs = 0;
   while outs < 3
     visiting player bats;
     for each run scored
        score = Read ("visitors");
        Write ("visitors", score + 1);
   outs = 0;
   while outs < 3
      home player bats;
      for each run scored
         score = Read ("home");
         Write ("home", score + 1);
end game;
```

Figure 1. A Simplified Baseball Game

As a concrete example, consider the write log for a sample game as shown in Figure 2. In this game, the home team scored first, then the visitors tied the game, then the home team scored twice more, and so on.

Write ("home", 1)
Write ("visitors", 1)
Write ("home", 2)
Write ("home", 3)
Write ("visitors", 2)
Write ("home", 4)
Write ("home", 5)

Figure 2. Sequence of Writes for a Sample Game

This sequence of writes could be from a baseball game with the inning-by-inning line score that is shown in Figure 3. This hypothetical game is currently in the middle of the seventh inning (the proverbial seventh-inning stretch), and the home team is winning 2-5.

	1	2	3	4	5	6	7	8	9	RUNS
Visitors	0	0	1	0	1	0	0			2
Home	1	0	1	1	0	2				5

Figure 3. The Line Score for this Sample Game

Suppose the key-value store that holds the visitors and home team's run totals resides in the cloud and is replicated among a number of servers. Different read guarantees may result in clients reading different scores for this game that is in progress. Table 3 lists the complete set of scores that could be returned by reading the visitors and home scores with each of the six consistency guarantees. Note that the visitors' score is listed first, and different possible return values are separated by comas.

Strong Consistency	2-5
Eventual Consistency	0-0, 0-1, 0-2, 0-3, 0-4, 0-5, 1-0, 1-1, 1-2, 1-3, 1-4, 1-5, 2-0, 2-1, 2-2, 2-3, 2-4, 2-5
Consistent Prefix	0-0, 0-1, 1-1, 1-2, 1-3, 2-3, 2-4, 2-5
Bounded Staleness	scores that are at most one inning out-of-date: 2-3, 2-4, 2-5
Monotonic Reads	after reading 1-3: 1-3, 1-4, 1-5, 2-3, 2-4, 2-5
Read My Writes	for the writer: 2-5 for anyone other than the writer: 0-0, 0-1, 0-2, 0-3, 0-4, 0-5, 1-0, 1-1, 1-2, 1-3, 1-4, 1-5, 2-0, 2-1, 2-2, 2-3, 2-4, 2-5

Table 3. Possible Scores Read for Each Consistency Guarantee

A *strong consistency* read can only return one result, the current score, whereas an *eventual consistency* read can return one of 18 possible scores. Observe that many of the scores that can be returned by a pair of eventually consistent reads are ones that were never the actual score. For example, reading the visitors' score may return two and reading the home team's score may return zero, even though the home team never trailed. The *consistent prefix* property limits the result to scores that actually existed at some time. The results that can be returned by a *bounded staleness* read clearly depend on the desired bound. Table 3 shows the possible scores for a bound of one inning, that is, scores that are at most one inning out-of-date; for a bound of 7 innings or more, the result set is the same as for *eventual consistency* in this example. In practice, a system is unlikely to express staleness bounds in units of "innings". So, for this example, assume that the reader requested a bound of 15 minutes and that the previous inning lasted exactly that long. For *monotonic reads*, the possible return values depend on what has been read in the past. For *read my writes* they depend on who is writing to the key-value store; in this example, assume that all of the writes were performed by a single client.

4. Read Requirements for Participants

Now, let's examine the consistency needs of a variety of people involved in a baseball game who want to read the score. Certainly, each of these folks could perform a strongly consistent read to retrieve the visiting and home team's score. In this case, as pointed out in the previous section, only one possible value would be returned: the current score. However, as shown in Table 2, readers requesting strong consistency will likely receive longer response times and may even find that the data they are requesting is not currently available due to temporary server failures or network outages. The point of this section is to evaluate, for each participant, the *minimum* consistency that is required. By requesting read guarantees that are weaker than strong consistency, these clients are likely to experience performance benefits and higher availability.

4.1 Official scorekeeper

The official scorekeeper is responsible for maintaining the score of the game by writing it to the persistent key-value store. Figure 4 illustrates the steps taken by the scorekeeper each time the visiting team scores a run; his action when the home team scores is similar. Note that this code is a snippet of the overall baseball game code that was presented in Figure 1.

```
score = Read ("visitors");
Write ("visitors", score + 1);
```

Figure 4. Role of the Scorekeeper

What consistency does the scorekeeper require for his read operations? Undoubtedly, the scorekeeper needs to read the most up-to-date previous score before adding one to produce the new score. Otherwise, the scorekeeper runs the risk of writing an incorrect score and undermining the game, not to mention inciting a mob of angry baseball fans. Suppose the home team had previous scored five runs and just scored the sixth. Doing an *eventual consistency* read, as shown in Table 3, could return a score of anything from zero to five. Perhaps,

the scorekeeper would get lucky and receive the correct score in response to his read, but he should not count on it.

Interestingly, while the scorekeeper requires strongly consistent data, he does not need to perform *strong consistency* reads. If there were multiple people playing the role of scorekeeper and taking turns updating the score, then they would need to perform reads that request strong consistency. However, each baseball game generally has one official scorekeeper. Since the scorekeeper is the *only* person who updates the score, he can request the *read my writes* guarantee and receive the same effect as a strong read. In the unusual event where the person serving as the scorekeeper changes in the middle of the game, the new scorekeeper could perform a strong read when he first updates the score but can use *read my writes* for subsequent reads. Essentially, the scorekeeper uses application-specific knowledge to obtain the benefits of a weaker consistency read without actually giving up any consistency.

This might seem like a subtle distinction, but, in fact, could be quite significant in practice. In processing a *strong consistency* read the storage system must pessimistically assume that some client, anywhere in the world, may have just updated the data. The system therefore must access a majority of servers (or a fixed set of servers) in order to ensure that the most recently written data is accessed by the submitted read operation. In providing the *read my writes* guarantee, on the other hand, the system simply needs to record the set of writes that were previously performed by the client and find *some* server that has seen all of these writes [11]. In a baseball game, the previous run that was scored, and hence the previous write that was performed by the scorekeeper, may have happened many minutes or even hours ago. In this case, almost any server will have received the previous write and be able to answer the next read that requests the *read my writes* guarantee.

4.2 Umpire

The umpire is the person who officiates a baseball game from behind home plate. The umpire, for the most part, does not actually care about the current score of the game. The one exception comes after the top half of the 9th inning, that is, after the visiting team has batted and the home team is about to bat. Since this is the last inning (and a team cannot score negative runs), the home team has already won if they are ahead in the score; thus, the home team can and does skip its last at bat in some games. The code for the umpire who needs to make this determination is shown in Figure 5.

```
if first half of 9th inning complete then
     vScore = Read ("visitors");
     hScore = Read ("home");
     if vScore < hScore
         end game;
```

Figure 5. Role of the Umpire

When accessing the score during the 9th inning, the umpire does need to read the current score. Otherwise, he might end the game early, if he incorrectly believes the home team to be ahead, or make the home team bat unnecessarily. Unlike the scorekeeper, the umpire never writes the score; he simply reads the values that were

written by the official scorekeeper. Thus, in order to receive up-to-date information, the umpire must perform *strong consistency* reads.

4.3 Radio reporter

In most areas of the United States, radio stations periodically announce the scores of games that are in progress or have completed. In the San Francisco area, for example, KCBS reports sports news every 30 minutes. The radio reporter performs the steps outlined in Figure 6. A similar, perhaps more modern, example is the sports scores that scroll across the bottom of the TV screen while viewers are watching ESPN.

```
do {
     vScore = Read ("visitors");
     hScore = Read ("home");
     report vScore and hScore;
     sleep (30 minutes);
}
```

Figure 6. Role of the Radio Sports Reporter

If the radio reporter broadcasts scores that are not completely up-to-date, that's okay. People are accustomed to receiving old news. Thus, some form of eventual consistency is fine for the reads that he performs. But what guarantees, if any, are desirable?

As shown in Table 3, the read with the weakest guarantee, an *eventual consistency* read, may return scores that never existed. For the sample line score given in Figure 3, such a read might return a score with the visitors leading 1-0, even though the visiting team has never actually been in the lead. The radio reporter does not want to report such fictitious scores. Thus, the reporter wants both his reads to be performed on a snapshot that hold a *consistent prefix* of the writes that were performed by the scorekeeper. This allows the reporter to read the score that existed at some time, without necessarily reading the current score.

But reading a consistent prefix is not sufficient. For the line score in Figure 3, the reporter could read a score of 2-5, the current score, and then, 30 minutes later, read a score of 1-3. This might happen, for instance, if the reporter happens to read from a primary server and later reads from another server, perhaps in a remote datacenter, that has been disconnected from the primary and has yet to receive the latest writes. Since everyone knows that baseball scores are monotonically increasing, reporting scores of 2-5 and 1-3 in subsequent news reports would make the reporter look foolish. This can be avoided if the reporter requests the *monotonic reads* guarantee in addition to requesting a *consistent prefix*. Observe that neither guarantee is sufficient by itself.

Alternatively, the reporter could obtain the same effect as a *monotonic read* by requesting *bounded staleness* with a bound of less than 30 minutes. This would ensure that the reporter observes scores that are at most 30 minutes out-of-date. Since the reporter only reads data every 30 minutes, he must receive scores that are increasingly up-to-date. Of course, the reporter could ask for a tighter bound, say 5 minutes, to get scores that are reasonably timely.

4.4 Sportswriter

Another interesting person is the sportswriter who watches the game and later writes an article that appears in the morning paper or that is posted on some web site. Different sportswriters may behave differently, but my observations (from having been a sportswriter) is that they often act as in Figure 7.

```
While not end of game {
    drink beer;
    smoke cigar;
}
go out to dinner;
vScore = Read ("visitors");
hScore = Read ("home");
write article;
```

Figure 7. Role of the Sportswriter

The sportswriter may be in no hurry to write his article. In this example, he goes out to a leisurely dinner before sitting down to summarize the game. He certainly wants to make sure that he reports the correct final score for the game. So, he wants the effect of a *strong consistency* read. However, he does not need to pay the cost. If the sportswriter knows that he spent an hour eating dinner after the game ended, then he also knows that it has been at least an hour since the scorekeeper last updated the score. Thus, a *bounded staleness* read with a bound of one hour is sufficient to ensure that the sportswriter reads the final score. In practice, any server should be able to answer such a read. In fact, an *eventual consistency* read is likely to return the correct score after an hour, but requesting *bounded staleness* is the only way for the sportswriter to be 100% certain that he is obtaining the final score.

4.5 Statistician

The team statistician is responsible for keeping track of the season-long statistics for the team and for individual players. For example, the statistician might tally the total number of runs scored by her team this season. Suppose that these statistics are also saved in the persistent key-value store. As shown in Figure 8, the home team's statistician, sometime after each game has ended, adds the runs scored to the previous season total and writes this new value back into the data store.

```
Wait for end of game;
score = Read ("home");
stat = Read ("season-runs");
Write ("season-runs", stat + score);
```

Figure 8. Role of the Statistician

When reading the team's score from today, the statistician wants to be sure to obtain the final score. Thus, she needs to perform a *strong consistency* read. If the statistician waits for some time after the game, then a *bounded staleness* read may achieve the same effect (as discussed in Section 4.4 for the sportswriter).

When reading the current statistics for the season, i.e. for the second read operation in Figure 8, the statistician also wants strong consistency. If an old statistic is returned, then the updated value written back will undercount the team's total runs. Since the statistician is the only person who writes statistics into the data store, she can use the *read my writes* guarantee to get the latest value (as discussed in Section 4.1 for the scorekeeper).

4.6 Stat watcher

Others who periodically check on the team's season statistics are usually content with eventual consistency. The statistical data is only updated once per day, and numbers that are slightly out-of-date are okay. For example, a fan inquiring about the total number of runs that have been scored by his team this season, as shown in Figure 9, can perform an *eventual consistency* read to get a reasonable answer.

```
do {
    stat = Read ("season-runs");
    discuss stats with friends;
    sleep (1 day);
}
```

Figure 9. Role of the Stat Watcher

5. Conclusions

Clearly, storing baseball scores is not the killer application for cloud storage systems. And we should be cautious about drawing conclusions from one simple example. But perhaps some lessons can be learned.

Table 4 summarizes the consistency guarantees desired by the variety of baseball participants that were discussed in the previous section. Recall that the listed consistencies are not the *only* acceptable ones. In particular, each participant would be okay with strong consistency, but, by relaxing the consistency requested for his reads, he will likely observe better performance and availability. Additionally, the storage system may be able to better balance the read workload across servers since it has more flexibility in selecting servers to answer weak consistency read requests.

Official scorekeeper	Read My Writes
Umpire	Strong Consistency
Radio reporter	Consistent Prefix & Monotonic Reads
Sportswriter	Bounded Staleness
Statistician	Strong Consistency, Read My Writes
Stat watcher	Eventual Consistency

Table 4. Read Guarantees for Baseball Participants

These participants can be thought of as different applications that are accessing shared data: the baseball score. In some cases, such as for the scorekeeper and sportswriter, the reader, based on application-specific knowledge, knows that he can obtain strongly consistent data even when issuing a weakly consistent read using a *read my writes* or *bounded staleness* guarantee. In some cases, such as the radio reporter, multiple guarantees must be combined to meet the reader's needs. In other cases, such as the statistician, different guarantees are desired for reads to different data objects.

I draw four main conclusions from this exercise:

- **All of the six presented consistency guarantees are useful**. Observe that each guarantee appears at least once in Table 4. Systems that offer only eventual consistency would fail to meet the needs of all but one of these clients, and systems that offer only strong consistency may underperform in all but two cases.

- **Different clients may want different consistencies even when accessing the same data**. Often, systems bind a specific consistency to a particular data set or class of data. For example, it is generally assumed that bank data must be strongly consistent while shopping cart data needs only eventually consistency. The baseball example shows that the desired consistency depends as much on *who* is reading the data as on the type of data.

- **Even simple databases may have diverse users with different consistency needs**. A baseball score is one of the simplest databases imaginable, consisting of only two numbers. Nevertheless, it effectively illustrates the value of different consistency options.

- **Clients should be able to choose their desired consistency**. The system cannot possibly predict or determine the consistency that is required by a given application or client. The preferred consistency often depends on *how* the data is being used. Moreover, knowledge of *who* writes data or *when* data was last written can sometimes allow clients to perform a relaxed consistency read, and obtain the associated benefits, while reading up-to-date data.

The main argument often expressed against providing eventual consistency is that it increases the burden on application developers. This may be true, but the extra burden need not be excessive. The first step is to define consistency guarantees that developers can understand; observe that the six guarantees presented in Table 1

are each described in a few words. By having the storage system perform write operations in a strict order, application developers can avoid the complication of dealing with update conflicts from concurrent writes. This leaves developers with the job of choosing their desired read consistency. This choice requires a deep understanding of the semantics of their application, but need not alter the basic structure of the program. None of the code snippets that were provided in the previous section required any additional lines to deal specifically with stale data.

Cloud storage systems that offer only strong consistency make it easy for developers to write correct programs but may miss out on the benefits of relaxed consistency. The inherent trade-offs between consistency, performance, and availability are tangible and may become more pronounced with the proliferation of geo-replicated services. This suggests that cloud storage systems should at least consider offering a larger choice of read consistencies. Some cloud providers already offer two both strongly consistent and eventually consistent read operations, but this paper shows that their eventual consistency model may not be ideal for applications. Allowing cloud storage clients to read from diverse replicas with a choice of several consistency guarantees could benefit a broad class of applications as well as lead to better resource utilization and cost savings.

6. References

1. D. Abadi. Consistency Tradeoffs in Modern Distributed Database System Design. *IEEE Computer*, February 2012.

   Explores the impact of consistency and latency tradeoffs on system design.

2. Amazon. Amazon DynamoDB. http://aws.amazon.com/dynamodb/.

   Describes the consistency, throughout, and pricing of Amazon's beta DynamoDB service.

3. E. Anderson, X. Li, M. Shah, J. Tucek, and J. Wylie. What Consistency Does Your Key-value Store Actually Provide? *Proceedings Usenix Workshop on Hot Topics in Systems Dependability*, 2010.

   Measures the frequency with which an eventually consistent key-value store actually provides strong consistency and reports that consistency violations are rare.

4. P. Bailis, S. Venkataraman, M. Franklin, J. Hellerstein, and I. Stoica. Probabilistically Bounded Staleness for Practical Partial Quorums. *Proceedings VLDB Endowment*, August 2012.

   Analyzes the probability of staleness for quorum-based systems with non-overlapping read and write quorums.

5. E. Brewer. CAP Twelve Years Later: How the "Rules" Have Changed. *IEEE Computer*, February 2012.

   Explains why designers often choose availability over consistency, but also revisits some of the tradeoffs.

6. B. Calder, *et. al*. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. Proceedings ACM Symposium on Operating Systems Principles, October 2011.

Presents the design of the Windows Azure Storage cloud storage system including support for geographic replication.

7. B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proceedings International Conference on Very Large Data Bases*, August 2008.

   Describes the relaxed consistency model adopted for the distributed database system underlying many of Yahoo!'s web applications.

8. Google. Read Consistency & Deadlines: More Control of Your Datastore. *Google App Engine Blob*, March 2010, http://googleappengine.blogspot.com/2010/03/read-consistency-deadlines-more-control.html.

   Announces a change to Google App Engine to allow eventually consistent reads.

9. T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency Rationing in the Cloud: Pay Only When It Matters. *Proceedings International Conference on Very Large Data Bases*, August 2009.

   Built a cloud database system on top of Amazon's S3 and shows that relaxing consistency can significantly lower transaction costs and improve performance.

10. Y. Saito and M. Shapiro. Optimistic Replication. *ACM Computing Surveys*, March 2005.

    Outlines a range of consistency choices for replicated data as well as implementation techniques.

11. D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session Guarantees for Weakly Consistent Replicated Data. *Proceedings IEEE International Conference on Parallel and Distributed Information Systems*, 1994.

    Defines read-your-writes, monotonic reads, and other session guarantees and shows how to implement them in an eventually consistent system.

12. W. Vogels. Eventually Consistent. *Communications of the ACM*, January 2009.

    Explains why Amazon chose eventual consistency for its large-scale, reliable infrastructure services

13. H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data Consistency Properties and the Trade-offs in Commercial Cloud Storages: The Consumers' Perspective. *Proceedings CIDR*, January 2011.

    Analyzes Amazon's SimpleDB and observes that the service frequently delivers stale data and fails to provide read-your-writes or monotonic read guarantees, and yet the observed performance of eventually consistent reads is not better than that of strong consistency reads to the same service.