# **Eventual Consistency**

**Introduction**

Distributed computing had been research area for quite some time and now the industry is embracing it with both hands. New buzz words in the industry like NoSql, NServiceBus, Hadoop, Big data etc. are all one or other form of distributed computing. There are multiple algorithms/principles which evolved before what distributed computing became what it is today. One of approach is eventual consistency across distributed systems.

**So, what is it?**

Eventual consistency is a way of maintaining the consistency of system/application state across machines with acceptable delays. This means that data in the distributed systems will not be the same for some period before it achieves consistency. Eventual constancy is not related only to database systems, but also applies to other systems. Below are few mentions where we can see eventual consistency happens

       NoSql databases like (Cassandra, Gemfire, RavenDB etc.)

       Messaging based systems (including NServiceBus)

       Domain Naming System (DNS)

       SQL Server Replication (of course it’s been there for some time)

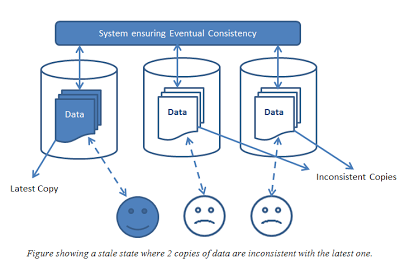
       Amazon’s Dynamo

**Why eventual consistency**: Any general purpose relational database like SQL Server, Oracle, MySQL etc. follow the ACID (Atomicity, Consistency, Isolation and Durability) properties. To follow these rules, they are heavy dependent on resource locking (rows, pages, table, and database). Let’s take an example,

UPDATE dbo.employee SET salary=salary + 500 WHERE employee\_id=1508

Here we are updating the salary information of an employee. To execute this query, the database server needs to lock the row/page to ensure that no one else is viewing stale data (depending on Isolation level) or updating it while the query execution is in progress. The DB server need to do this for each of the query executed against the database. If there are 2 million rows updated, the database server should create locks for each of the 2 million rows to ensure data consistency (of course lock escalation will take care of this). This is too much to handle for any database engine. The result of all this locking is poor performance.

Data analytics reveals that 80% of the applications does not require immediate consistency but requires very good performance. So, in distributed systems, the data is scattered to multiple systems and there are replicas maintained to ensure good read performance. Since the data is distributed, traditional locking mechanisms cannot be used. Few good engineers decided to move the logic of locking out of the database systems and place them in the application so that they have more control on data consistency. The result of this approach is called eventual consistency.

[](http://3.bp.blogspot.com/-YL5ra8dYk_8/UYOIPAqlSrI/AAAAAAAAABs/RWorkEcH94M/s1600/Eventual+Consistency+-+Copy.png)

In eventually consistent systems, the data/application state is not updated in all the replicas at the same time. But the primary data owner is updated and the replicas will get the update eventually. The update can be triggered either based on read or write or asynchronously. Eventually consistent systems need to maintain history of transactions so that they can make application state consistent or tolerate the inconsistency and take corrective actions.

**Can we tolerate inconsistency**?

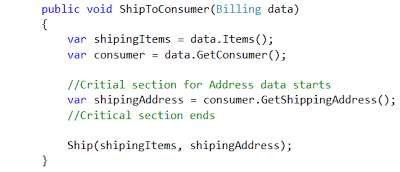
This purely depends on the application. Let’s take few examples where NoSql (or any eventually consistent implementation is done).

In a sales application, a user (a sales person) has updated a consumer’s date of birth/age, address and phone number. Let’s assume the user information is updated in only one of the copies. What if another user (say billing person) reads the same consumer information from another data replica which is not in sync with the latest update? Of course, the data is in-consistent and the second user will read stale data (unless we have read triggered consistency). How important is this information? What action does the second user going to take based on this information? With a casual glance, we may say date of birth or age can be inconsistent for some time. What about the address and phone number? Can we tolerate this inconstancy? Yes and No. Yes, when there is no action taken by the billing system. When the billing system is going to use the consumer address for shipment we will not be able to use stale data.

Let’s take another example. In ticket booking system, two users are trying to book ticket to a cinema hall. Let’s assume both are trying to block the same seat (say A21) by connecting to two different data replicas. Can both the users be allowed to block the same seat? Yes, both the users can block the same seat until one of them makes the payment and confirms the ticket. When the second person makes the payment, we need to confirm if the same seat has been already used. If the seat is already booked, we need to intimate the user and refund the amount back to the user.

**Critical section:**

Every workflow has some portion of logic inside them where data in-consistency cannot be tolerated. We can call this portion of code as critical section. In the billing system above, using the consumer address for shipment is a critical section for address data. In the ticketing system, the seat availability data enters the critical section when we receive the payment confirmation. The critical section of the workflow is a place where it should know if the data is consistent or not (dirty flag), so that the application can decide to live with the inconsistency or to enforce consistency.

[](http://4.bp.blogspot.com/-nBo6FxKikiM/UYOIRhll9iI/AAAAAAAAAB4/Jnv1QR1L7vM/s1600/CriticalSection+-+Copy.png)

Few points to think about, before we implement eventual consistency in our applications.

       We need to identify the data specific critical sections in our application where data inconstancies cannot be tolerated.

       If there are inconsistencies, what are the rollback or corrective workflows that will solve the problems of inconsistency

       What are the policies around the corrective workflow.  They should be clearly documented and made available to the consumers.

**End Note**:

In SQL Server, changing the isolation levels can make significant changes to the way locks are handled. I remember SQL Server version 7.5 had only 4 isolation levels and now there are 7 isolation levels in SQL Server 2012. This is also a sign of how developers are increasingly taking control of parallelism in their applications rather than relying on the DBMS to handle it. With distributed databases, the control is more in the hands of developers. The world is moving towards faster and more distributed systems and eventual consistency is one of the things that are fueling this.

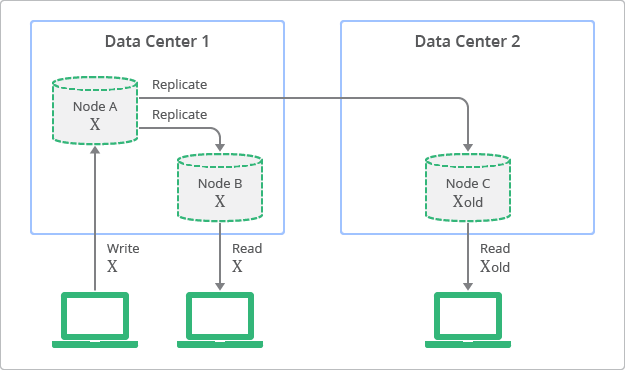
<http://cloudshankar.blogspot.in/2013/05/eventual-consistency.html>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### **What is Eventual Consistency?**

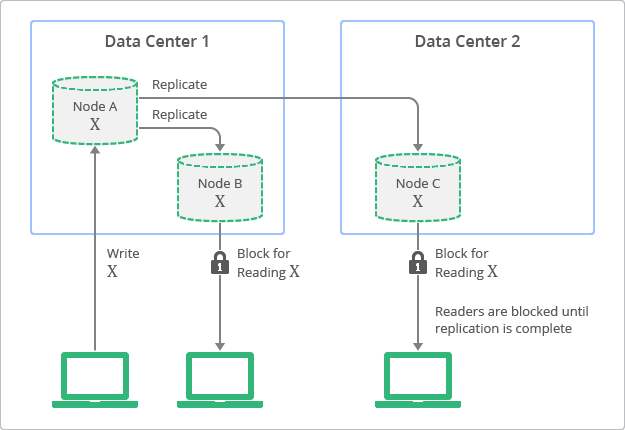
Eventual consistency is a theoretical guarantee that, provided no new updates to an entity are made, all reads of the entity will eventually return the last updated value. The Internet Domain Name System (DNS) is a well-known example of a system with an eventual consistency model. DNS servers do not necessarily reflect the latest values but, rather, the values are cached and replicated across many directories over the Internet. It takes a certain amount of time to replicate modified values to all DNS clients and servers. However, the DNS system is a very successful system that has become one of the foundations of the Internet. It is highly available and has proven to be extremely scalable, enabling name lookups to over a hundred million devices across the entire Internet.

Figure 1 illustrates the concept of replication with eventual consistency. The diagram illustrates that although replicas are always available to read, some replicas may be inconsistent with the latest write on the originating node, at a moment in time. In the diagram, Node A is the originating node and nodes B and C are the replicas.

Figure 1: Conceptual Depiction of Replication with Eventual Consistency

In contrast, traditional relational databases have been designed based on the concept of strong consistency, also called immediate consistency. This means that data viewed immediately after an update will be consistent for all observers of the entity. This characteristic has been a fundamental assumption for many developers who use relational databases. However, to have strong consistency, developers must compromise on the scalability and performance of their application. Simply put, data should be locked during the period of update or replication process to ensure that no other processes are updating the same data.

A conceptual view of the deployment topology and replication process with strong consistency is shown in Figure 2. In this diagram, you can see how replicas always have values consistent with the originating node, but are not accessible until the update finishes.

Figure 2: Conceptual Depiction of Replication with Strong Consistency

### **Balancing Strong and Eventual Consistency**

Non-relational databases have become popular recently, especially for web applications that require high-scalability and performance with high-availability. Non-relational databases let developers choose an optimal balance between strong consistency and eventual consistency for each application. This allows developers to combine the benefits of both worlds. For example, information such as “knowing who in your buddy list is online at given time” or “knowing how many users have +1’d your post” are use cases where strong consistency is not required. Scalability and performance can be provided for these use cases by leveraging eventual consistency. Use cases which require strong consistency include information such as “whether a user finished the billing process” or “the number of points a game player earned during a battle session”.

To generalize the examples just given, use cases with very large numbers of entities often suggest that eventual consistency is the best model. If there are a very large number of results in a query, then the user experience may not be affected by the inclusion or exclusion of specific entities. On the other hand, use cases with a small number of entities and a narrow context suggest that strong consistency is required. The user experience will be affected because the context will make users aware of which entities should be included or excluded.

For these reasons, it is important for developers to understand the non-relational characteristics of Cloud Datastore. The following sections discuss how eventual consistency and strong consistency models can be combined to build a scalable, highly available, and highly performing application. In doing so, consistency requirements for a positive user experience will still be satisfied.

<https://cloud.google.com/datastore/docs/articles/balancing-strong-and-eventual-consistency-with-google-cloud-datastore/>