Alfred Zimmermann, Alexander Rossmann (Hrsg.): Digital Enterprise Computing 2015,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2015 247

On Practical Implications of Trading ACID for CAP in the Big Data Transformation of Enterprise Applications

Andreas Tönne[1]

Abstract: Transactional enterprise applications today depend on the ACID property of the underlying database. ACID is a valuable model to reason about concurrency and consistency constraints in the application requirements. The new class of NoSQL databases that is used at the foundation of many big data architectures drops the ACID properties and especially gives up strong consistency for eventual consistency. The common justification for this change is the CAP-theorem, although general scalability concerns are also noted. In this paper we summarize the arguments for dropping ACID in favour of a weaker consistency model for big data architectures. We then show the implications of the weaker consistency for the business customer and his requirements when a transactional Java EE application is transformed to a big data architecture.

Keywords: big data, acid, cap theorem, consistency, requirements, nosql

1 Introduction

The ACID property of relational databases has proven to be a valuable guarantee for specifying enterprise application requirements. Business customers are enabled to express their business needs as if they are sequentially executed. The complicated details of concurrency control of transactions and maintaining consistency are delegated to the database. ACID has become a synonym for a promise of effective and safe concurrency of business services. And business customers have learned to assume this as given and as a reasonable requirement.

The broad scale introduction of NoSQL[Ev09][Fo12] databases to enterprises for solving big data business needs introduces a new class of promises to the business customer. Strong consistency as promised by ACID is replaced by the rather weak *eventual consistency*[Vo09]. The CAP-theorem[Br00] is the excuse of the NoSQL developers and generally of the big data architects to weaken consistency in favour of availability, although modern NoSQL database also offer higher levels of consistency.

In this paper, we discuss the consequences of the transformation of enterprise applications to a big data architecture. Our focus lies on the practical implications of loosening consistency for the business customers and their requirements. We consider the severity of the implications, the difficulty to balance these with the customer expectations and strategies for remedies.

[1] NovaTec Consulting GmbH, Dieselstrasse 18/1, 70771 Leinfelden-Echterdingen, andreas.toenne@novatec-gmbh.de

1.1 Transformation Scope

The class of enterprise applications, which we consider are the OLTP (online transaction processing) applications. As the name implies, these applications critically depend on transactional properties and for this upholding ACID is seen mandatory. Our general interest lies in the question to what degree (and at what cost) can business needs that today require OLTP applications can be transformed to the big data paradigm.

1.2 Big Data Challenges

OLTP applications are under a strong big data pressure. The three main characteristics *volume*, *variety* and *velocity* of big data also apply to many today's enterprise applications. The volume of data to handle increases through new relevant sources like for example the Internet of Things or the German governments Industrie 4.0 initiative [Bm15]. Also globalization in general increases the amount of data to handle. The variety of data increases greatly because structured data source are increasingly accompanied or replaced by unstructured sources like the Web or Enterprise 2.0 initiatives. Acceleration of business processes and the success of big data analytics (data science) of competitors increase the demands for a higher velocity, especially combined with the required higher data volumes.

We think it is fair to state that many OLTP applications already operate under big data conditions. However they show a strong inertia to complete the transformation. Their architecture hinders the integration of unstructured data source. Their choice of for example Java EE with a relational database means a hard limitation of scalability. And as will be shown in this paper, the practical consequences and compromises for big data scale processing are threatening to the business customer.

1.3 Contribution

We experienced these transformation challenges in the migration of a customer's middleware solution to a big data architecture. The first transformation milestone, establishing demonstration capabilities of the solution, was an eleven man year effort. The architecture changed from a layered monolith based on Java EE and a relational database to a cloud ready microservices topology. We were using the Titan graph database [Th15], Cassandra [Ap15][LM10] as the NoSQL storage backend and Elasticsearch [El15] for indexing. The transformation was critical to our customer for the simple reason that the existing implementation was falling short of the scaling requirements by three orders of magnitudes.

The solutions main purpose is to provide ETL (extract, transform, load) services to enterprise applications in a flexible way. Structured and unstructured data is extracted from a large range of sources (using agents) and lossless converted to a common data model. This data is persisted in the style of a data lake [Fo15] but structured and

enriched with metadata. The metadata describes semantic and formal relations between the data records similar to an ontology. The combined data structure is persisted in a distributed graph database.

The heart of the solution are the analysis algorithms and their quality requirements. Running such analysis concurrently while preserving consistency is a critical requirement. If for example two data records are analyzed for their relationships concurrently, duplicate relations like "my user matches your author" and "my author matches your user" would lead to wrong overall relation weights between the two records. Such concurrency conflict is actually a common case: if sets of related records are modified in an application, these would be imported for analysis concurrently and the consistency of relations is threatened across many concurrent nodes. Consistency rules like requiring uniqueness of commutative relations ("we have something in common") were enforced in the business requirements through uniqueness constraints on the database. It should be needless to point out that this was a scalability killer by design.

1.4 Structure of this Paper

In the following section 2 we introduce the reader to the CAP theorem and eventual consistency and discuss their relevance for scalability.

Section 3 is the main result of this paper, showing the implications of switching to eventual consistency for scalability reasons to our business project.

Some remaining challenges are described in section 4.

2 The CAP Theorem, Eventual Consistency and Scalability

In this section, we provide an overview of the influences of the CAP theorem and eventual consistency to scalability concerns of distributed transactional applications. We motivate the decision to loosen consistency up to the point of eventual consistency. And we briefly show possibilities to achieve higher levels of consistency and their price in terms of scalability and effort.

2.1 Scalability is the New Top Priority

The transformation of application architectures from transactional monoliths to distributed cloud service topologies, from traditional to web and big data, usually comes with a shift of priorities. An influential statement for this shift was given by Werner Vogels, Amazon CTO:

"Each node in a system should be able to make decisions purely based on local state. If you need to do something under high load with failures occurring and you need to reach

agreement, you're lost. If you're concerned about scalability, any algorithm that forces you to run agreement will eventually become your bottleneck. Take that as a given." [Vo07][In07]

The shift was from *consistency* as the main promise for application properties to *availability* of services, which is treated second-class by ACID systems. And the ruling regime, as pointed out by Vogels, is the (horizontal) scalability. These new architectures are characterized by being extremely distributed, both in terms of number of computing and data nodes and distance measured in communication latency and network partition probability. Data replication is used to achieve better proximity of data and client and better availability in the presence of network partitions. Also for big data, replication is often the only sensible backup strategy.

2.2 BASE Semantics and Eventual Consistency

The conflict between ACIDs consistency focus and the availability demands of web based scalable large distributed systems lead to the alternative BASE data semantics[Fo97] (basically available, soft state, eventual consistency) which uses the weak *eventual consistency* model. This basically means that after a write there is an unspecified period of time of inconsistency while the change is replicated through the distributed topology of data nodes. Which version of the data is presented at a node in this time window is pure luck. Eventually after a period of no changes, the network of data nodes stabilizes and agrees on the last written version.

We do not want to embark on a discussion of other stronger consistency models that are also offered by some NoSQL databases like Cassandra using quorums or paxos protocols. When discussing this consistency issue with business customers in the light of their requirements, the interesting question is "how on earth did they get away with such a lousy consistency model?".

2.3 CAP Theorem

The answer lies in the CAP theorem [Br00] and its proof [GL02], which was (ab-)used by the developers of NoSQL databases to ignore the matter of stronger consistency to achieve better scalability and availability, and justify eventual consistency as a law of nature. Any distributed system may have the following three important properties and Brewer's conjecture was that you can only have two of these at any time[2]:

1. **C**onsistency, which means single-copy consistency (ACID consistency without the data invariants): all nodes have the same version of a value.

[2] Actually the proof by Gilbert and Lynch makes use of an asynchronous model and thus introduces the assumption that there is no common notion of time in the distributed system. This seems to be consistent with the intention of Brewer but limits the scope of CAP in practical systems.

2. **A**vailability, which can be characterized as the promise: as long as a node is up, it always returns answers (after an unbounded duration of time).

3. **P**artition Tolerance, which means it is ok to lose arbitrarily many messages between nodes.

This theorem and its adoption as a justification for eventual consistency sparked lasting discussions about its relevance and which choice of CA, CP or AP is the best. Brewer himself summarizes his view of the state of affairs of CAP [Br12] highlighting the very important interpretation of CAP that "First, because partitions are rare, there is little reason to forfeit C or A when the system is not partitioned. Second, the choice between C and A can occur many times within the same system at very fine granularity..." and "Because partitions are rare, CAP should allow perfect C and A most of the time, but when partitions are present or perceived, a strategy that detects partitions and explicitly accounts for them is in order." CAP is all about managing the exceptional case of a network partition, which is a split-brain situation. The decision to take is called the *partition decision* by Brewer [Br12]:

- Cancel an operation after a timeout and decrease availability or
- Proceed with the operation and risk inconsistency

The matter of healing network partitions is handled by NoSQL implementations using for example brute-force strategies like last-write-wins. A more graceful solution would be to use vector clocks to identify and handle merge conflicts of partitioned data change.

2.4 NoSQL With Higher Levels Of Consistency

The NoSQL community initially has settled for AP with eventual consistency, inspired by Amazon's Dynamo database [De07]. But we are seeing more and more additions to NoSQL databases that allow stronger consistency models like for example Cassandra [Da15] allowing a quorum consistency level. Thus the choice of AP or CP is no longer fixed by our architecture and in Cassandra for instance it is possible at the level of individual reads and writes.

There is however a price attached to higher levels of consistencies and that is the degree of scalability we can achieve. As Vogels pointed out in the above quote, any need for agreement opposes scalability. Achieving strong consistency with a quorum or even naively requiring all data nodes in a system to acknowledge a write will likely reduce scalability of a big data scale system drastically.

In our example application, lowering the consistency level to single node writes resulted in orders of magnitude speedups (throughput) while exposing architectural issues that we will describe in the following section[3].

The good news is that eventual consistency is not as bad as it sounds. Recent analysis [BG13] shows that eventual consistent databases can achieve consistency on a frequent basis within tens or hundreds of milliseconds. What this does not tell us is what damages are done during this period and how to deal with them.

3 Business Implications

This section is the main result of the transformation of our OLTP application to a big data stack with a NoSQL database. We discuss the tension in the project between the business customer, requiring ACID properties that he was used to and the requirement to achieve a substantial increase in scalability and throughput. The reduction in consistency leads to some difficult questions about the business requirements. We also give some technical details of how we dealt with the errors introduced by the reduced consistency level.

3.1 ACID Requirements

Judging the business implications of turning towards a big data database with loosened consistency requires us to respect the business customer needs and expectations. In an ideal world, a business service is specified under the assumption of perfect singularity of the executing system. No side effects, no concurrency, no temporal considerations, no other users, no failures. Consistency is defined in terms of business rules about data integrity.

In the last decades, transactional systems upholding the ACID properties were understood and accepted by business customers as the nearest approximation of this simplified ideal that could be delivered by their IT people.

The ACID properties of a transaction guarantees the following beneficial guarantees for its execution:

Atomicity - The guarantee that the transaction is executed in one atomic step (from an outside view) or aborted as a whole.

Consistency - The guarantee that the transaction transforms the database from a consistent state to another consistent state. The notion of consistency can vary largely, depending on the capabilities of the underlying database. Usually this includes structural

[3] Performance figures are unfortunately not available and meaningful since the system went through a rapid sequences of changes.

consistency rules about uniqueness and referential integrity as well as data consistency rules.

Isolation - The guarantee that concurrent transactions are isolated against their respective database changed to a configurable degree.

Durability - The guarantee that once a transaction was committed (success), its changes are permanent, even in the presence of system failures.

ACID has proven to be a simple enough model to talk about requirements in the presence of such nasty things as concurrency and failures. At times, the architect would come back with "Would it be allowed that..." questions, asking for exceptions to the simplified world and complicating things a bit. The I (Isolation) in ACID is the adjusting screw for such optimizations of SQL systems possibly affecting their consistency. Locking and database constraints on uniqueness are commonly used in requirements to hedge against consistency errors and to control concurrency.

3.2 Dropping ACID to Achieve Higher Scalability

It is common wisdom that big data is a disruptive concept that requires the transformation of the businesses core for adoption. Business requirements for achieving business goals need to be redefined to reflect the paradigms of big data architectures. That means: What promises for consistency, availability and scalability can your IT people deliver on a big data scale? Taking a technology approach by replacing relational databases by NoSQL and call it good is no option. Unfortunately as we experienced in the mentioned project this is an all too alluring idea to waive it easily. The business customer must be educated in the consequences of asking to "solve the problem with big data". And this means to find the sweet spot in the triangle of loosening consistency, achievable scalability and implementation effort.

In our example, we opted for loosening consistency as much as possible because that was the quick route[4] to deliver a system with the necessary performance. Going for eventual consistency means to drop ACID in favour of a BASE semantics of transactions. That in turn exposed all the unknown invariants of the analysis algorithms. Dropping ACID also created an enormous technical dept to handle the consistency errors.

3.3 Consequences of Lowering Consistency

We mentioned the uniqueness constraint on commutative relations between records. The analysis requirements disallowed dual relations of the kind "My user is your author" and "My author is your user" since they are considered commutative and duplicating them

[4] We had to deliver the demo ready version within six months.

puts a too strong emphasis on the relationship of the two records. Trading ACID for an unconstrained database created questions for this example:

- How often does this constraint violation take place in practice?
- How severe is the error introduced?
  - Follow-up question: Has anyone a measure for meta annotation errors?
- Do other algorithms using the meta annotation break on the duplication itself or on the annotation weight error?
  - Follow-up question: Who depends on this annotation anyway? External clients or also internal algorithms?
- Can we ignore the error for some time or is there a build-up effect over time?
- Do we need to filter the duplicate annotations at read time or can we repair the consistency at intervals?
  - Follow-up question: What would be a reasonable interval?

These questions led to rather uncomfortable and time-consuming meetings with the business customer. The customer did not have a need to think about these issues before; he was shielded from them by a single requirement of uniqueness of commutative relations.

As Brewer [Br12] noted: "Another aspect of CAP confusion is the hidden cost of forfeiting consistency, which is the need to know the system's invariants. The subtle beauty of a consistent system is that the invariants tend to hold even when the designer does not know what they are. Consequently, a wide range of reasonable invariants will work just fine."

We observed a few distinct types of issues with weakening consistency and dropping constraints that have distinct reasons and justifications. The following sections show a few common cases.

3.4 Internal Phantom Reads

In our example, this was the type of effect with the most quality damages. This is the premier effect of eventual consistency. An internal phantom read happens when one concurrent service execution runs in isolation (due to the eventual consistency time window) to another execution and sees old versions of data modified by the second execution. Example: two related records are analyzed in parallel and due to bad luck of replication one analysis does not see the other's updated record. This is severe since it means that the analysis results become random for records imported in a tight time window. Also consider that there is a higher probability for a meaningful relation

between two records received at the same time. We might lose valuable analysis results.

In a transactional system, this scenario calls for a serialization of execution with locking. A solution was to keep a journal of analysis order and determine those imports that might need a re-analysis to stabilize the annotation consistency.

3.5 Where is my Data?

Eventual consistency does not only affect concurrent execution but sometimes also serial execution. This depends on the implementation of the replication strategy of the database but also on the architecture of the system itself. Some databases do not give a guarantee that data written can be immediately read back, even at the same node, because the replication preparation takes some time. We experienced such problems when chaining analysis algorithms. A similar error (but not consistency caused) can take place with the Elasticsearch refresh interval. Here we did not find our own analysis results in the index for a short time until Elasticsearch had prepared and executed the index refresh.

We only found a brute-force solution to assure that chains of algorithms can work on their own results, apart from passing the results between the algorithm steps. We needed to use delays, either explicitly between each step or as retries with an exponential backoff.

3.6 Zombie Data

Combining updates and deletes in a concurrent system gives you the usual zombie problem if updates and deletes of the same record overrun each other. We employed measures like timestamps to ensure a monotonic order of analysis executions for a record. And we wrote tombstones for deleted records to distinguish true data from zombies.

Tombstoning became an issue because we could not use locking for record updates. Acquiring locks in a distributed database like Cassandra is a very expensive operation that requires a consensus between all accessible nodes. Since updates were a very frequent operation, this was not acceptable. So there is a very small chance that a tombstone is overwritten, creating a zombie record. As usual with big data, small chances become significant given enough data updates. This problem awaits a solution still.

3.7 Who is Unique?

Assuring uniqueness has the same costs as a distributed lock for record updates. The key strategy to avoid this cost was to detach the creation of unique data from the remaining analysis algorithms. Some of the unique records were pre-created in a startup phase from

sample data to reduce frequent locks. This was particularly successful for sets of unique data with an upper bound. For example words of a particular language.

3.8 Probabilistic Truth

This is an issue that originates from the distributed nature of the database itself and from the big data scale. Keeping a correct tally of data can be too expensive at real-time and one has to use approximation techniques. Consider for example the number of distinct words in the whole database. Keeping such tally does require some sort of synchronization between the concurrent updates. We experimented with Cassandra counter columns[Da14] and with elastic caching approaches using Hazelcast[Ha15] but finally decided to use a probabilistic approach with a defined error rate, based on a Google implementation[HNH13] of HyperLogLog[Fl07].

4 Outlook

Our experience with the transformation of our OLTP Java EE / Relational application to a native big data stack shows the importance of a hybrid persistency approach. As advocates of ACID for NoSQL[Pi15] highlight, there is a need for the option to choose stronger consistency at high performance. In a hybrid approach, we could separate data needing linearizability consistency and weaker consistency. For example, dumping the mass of data in HBase and keeping the strongly consistent references to that data elsewhere. Google Spanner [Cr13] shows that by having an accurate and coherent distributed clock, one can achieve strong consistency at a reasonable cost.

Another idea we want to pursue is to avoid certain types of consistency errors using CRDT (commutative replicated data types)[Sh11] to propagate conflict free distributed changes in the case of network partitions. The case of internal phantom reads can be considered to be effected by a short time network partition. We would be happy to resolve this partition conflict-free. The distributed database riak [Ba15] shows how to combine NoSQL concepts with CRDT [Ba15a] to achieve conflict-free data types.

5 Conclusions

Transforming existing business requirements that are implemented by OLTP applications to a big data solution using today's best practice big data technology means to give up many benefits of ACID transactions and especially strong consistency in general.

We showed our practical experiences with a sizable (and from the customers perspective very successful) transformation of an existing OLTP application to a big data stack. This example shows that although the weakening of consistency to the level of eventual

consistency does not prohibit this transformation, it still creates a significant dept. We gave some examples of the errors introduced by the lower consistency and how we dealt with them technically.

The cost of compensating for consistency defects and the achievable scalability had to be compared and judged together with the business customer. The effort for the transformation of the business requirements and thus the effort put on the business customer and architect turned out to be significant and it needed to be carefully accompanied by consulting and mentoring.

Invariants of business requirements that are usually hidden by the ACID properties and database locks and constraints become important when high scalability needs to be achieved with a NoSQL database.

References

[Ap15] Apache: Welcome to Apache Cassandra, http://cassandra.apache.org, April 9, 2015.

[Ba15] basho: riak product site, http://basho.com/riak/, April 9 2015.

[Ba15a] basho: How to Build a Client Library for Riak 2.0, http://basho.com/tag/crdt/, April 9 2015.

[BG13] Bailis, P.; Ghodsi, Al.: Eventual Consistency Today: Limitations, Extensions, and Beyond. acm queue volume 11, issue 3, April 9, 2013.

[Bm15] Bundesministerium für Bildung und Forschung: Zukunftsprojekt Industrie 4.0, http://www.bmbf.de/de/9072.php, April 9, 2015.

[Br00] Brewer, E.: Towards Robust Distributed Systems. In: Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC '00), pages 7-10, 2000.

[Br12] Brewer, E.: CAP Twelve Years Later: How the 'Rules' Have Changed. Computer, pages 23-29, 2012.

[Cr13] Corbett, J. et al.: Spanner: Google's Globally-Distributed Database. In: ACM Transactions on Computer Systems (TOCS), volume 31, issue 3, 2013.

[Da14] DataStax: What's New in Cassdra 2.1: Better Implementation of Counters. http://www.datastax.com/dev/blog/whats-new-in-cassandra-2-1-a-better-implementation-of-counters, May 20, 2014.

[Da15] DataStax: Configuring Data Consistency. http://docs.datastax.com/en/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html, April 9, 2015.

[De07] DeCandia, G. et al.: Dynamo: Amazon's Highly Available Key-Value Store. In: SOSP '07 Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, pages 205-220, 2007.

[El15] Elasticsearch BV: Elasticsearch | Search & Analyze Data in Real Time, https://www.elastic.co/products/elasticsearch, April 9, 2015.

[Ev09] Evans, E.: Eric Evan's Weblog, http://blog.sym-link.com/2009/05/12/nosql_2009.html, April 9, 2015.

[Fl07] Flajolet, P. et al.: HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In: AOFA '07: Proceedings of the 2007 International Conference on the Analysis of Algorithms, 2007.

[Fo97] Fox, A.; Gribble, S.; Chawate, Y.; Brewer, E.; Gauthier, P.: Cluster-Based Scalable Network Services. In: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP-16), 1997.

[Fo12] Fowler, M.: NosqlDefinition, http://martinfowler.com/bliki/NosqlDefinition.html, April 9, 2015.

[Fo15] Fowler, M.: DataLake, http://martinfowler.com/bliki/DataLake.html, April 9, 2015.

[GL02] Gilbert, S.; Lynch, N.: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. In: ACM SIGACT News 33, pages 51-59, 2002.

[Ha15] Hazelcast: Company site, http://hazelcast.com, April 9 2015.

[HNH13] Heule, S.; Nunkesser, M.; Hall, A.: HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm. In: EDBT '13 Proceedings of the 16th International Conference on Extending Database Technology, 2013.

[In07] InfoQ: Presentation: Amazon CTO Werner Vogels on Availability and Consistency , http://www.infoq.com/news/2007/08/werner-vogels-pres, April 9, 2015.

[LM10] Lakshman, A.; Malik P.: Cassandra: a decentralized structured storage system. In: ACM SIGOPS Operating Systems Review Volume 44 Issue 2, pages 35-40, 2010.

[Pi15] Pimentel, S.: The Return of ACID in the Design of NoSQL Databases, http://www.methodsandtools.com/archive/acidnosqldatabase.php, April 9 2015.

[Sh11] Shapiro, M. et al.: A comprehensive study of convergent and commutative replicated data types. INRIA Technical Report RR-7506, 2011.

[Th15] Thinkaurelius: Titan Distributed Graph Database, http://thinkaurelius.github.io/titan/, April 9, 2015.

[Vo07] Vogels, W.: Availability & Consistency. QCon 2007.

[Vo09] Vogels, W.: Eventually Consistent. Communications of the ACM 52, 2009.