How to beat the CAP theorem

Nathan Marz
Thursday, October 13, 2011
http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html

The CAP theorem states a database cannot guarantee consistency, availability, and partition-tolerance at the same time. But you can't sacrifice partition-tolerance (see here and here), so you must make a tradeoff between availability and consistency. Managing this tradeoff is a central focus of the NoSQL movement.

Consistency means that after you do a successful write, future reads will always take that write into account. Availability means that you can always read and write to the system. During a partition, you can only have one of these properties.

Systems that choose consistency over availability have to deal with some awkward issues. What do you do when the database isn't available? You can try buffering writes for later, but you risk losing those writes if you lose the machine with the buffer. Also, buffering writes can be a form of inconsistency because a client thinks a write has succeeded but the write isn't in the database yet. Alternatively, you can return errors back to the client when the database is unavailable. But if you've ever used a product that told you to "try again later", you know how aggravating this can be.

The other option is choosing availability over consistency. The best consistency guarantee these systems can provide is "eventual consistency". If you use an eventually consistent database, then sometimes you'll read a different result than you just wrote. Sometimes multiple readers reading the same key at the same time will get different results. Updates may not propagate to all replicas of a value, so you end up with some replicas getting some updates and other replicas getting different updates. It is up to you to repair the value once you detect that the values have diverged. This requires tracing back the history using vector clocks and merging the updates together (called "read repair").

I believe that maintaining eventual consistency in the application layer is too heavy of a burden for developers. Read-repair code

is extremely susceptible to developer error; if and when you make a mistake, faulty read-repairs will introduce irreversible corruption into the database.

So sacrificing availability is problematic and eventual consistency is too complex to reasonably build applications. Yet these are the only two options, so it seems like I'm saying that you're damned if you do and damned if you don't. The CAP theorem is a fact of nature, so what alternative can there possibly be?

There is another way. You can't avoid the CAP theorem, but you can isolate its complexity and prevent it from sabotaging your ability to reason about your systems. The complexity caused by the CAP theorem is a symptom of fundamental problems in how we approach building data systems. Two problems stand out in particular: the use of mutable state in databases and the use of incremental algorithms to update that state. It is the interaction between these problems and the CAP theorem that causes complexity.

In this post I'll show the design of a system that beats the CAP theorem by preventing the complexity it normally causes. But I won't stop there. The CAP theorem is a result about the degree to which data systems can be fault-tolerant to machine failure. Yet there's a form of fault-tolerance that's much more important than machine fault-tolerance: human fault-tolerance. If there's any certainty in software development, it's that developers aren't perfect and bugs will inevitably reach production. Our data systems must be resilient to buggy programs that write bad data, and the system I'm going to show is as human fault-tolerant as you can get.

This post is going to challenge your basic assumptions on how data systems should be built. But by breaking down our current ways of thinking and re-imagining how data systems should be built, what emerges is an architecture more elegant, scalable, and robust than you ever thought possible.

What is a data system?

Before we talk about system design, let's first define the problem we're trying to solve. What is the purpose of a data system? What is data? We can't even begin to approach the CAP theorem unless we can answer these questions with a definition that clearly encapsulates every data application.

Data applications range from storing and retrieving objects, joins, aggregations, stream processing, continuous computation, machine learning, and so on and so on. It's not clear that there is such a simple definition of data systems -- it seems that the range of things we do with data is too diverse to capture with a single definition.

However, there is such a simple definition. This is it:

```
Query = Function(All Data)
```

That's it. This equation summarizes the entire field of databases and data systems. Everything in the field -- the past 50 years of RDBMS's, indexing, OLAP, OLTP, MapReduce, ETL, distributed filesystems, stream processors, NoSQL, etc. -- is summarized by that equation in one way or another.

A data system answers questions about a dataset. Those questions are called "queries". And this equation states that a query is just a function of all the data you have.

This equation may seem too general to be useful. It doesn't seem to capture any of the intricacies of data system design. But what matters is that every data system falls into that equation. The equation is a starting point from which we can explore data systems, and the equation will eventually lead to a method for beating the CAP theorem.

There are two concepts in this equation: "data" and "queries". These are distinct concepts that are often conflated in the database field, so let's be rigorous about what these concepts mean.

Data

Let's start with "data". A piece of data is an indivisible unit that you hold to be true for no other reason than it exists. It is like an axiom in mathematics.

There are two crucial properties to note about data. First, data is inherently time based. A piece of data is a fact that you know to be true at some moment of time. For example, suppose Sally enters into her social network profile that she lives in Chicago. The data you take from that input is that she lived in Chicago as of the particular moment in time that she entered that information into her profile. Suppose that on a later date Sally updates her profile location to Atlanta. Then you know that she

lived in Atlanta as of that particular time. The fact that she lives in Atlanta now doesn't change the fact that she used to live in Chicago. Both pieces of data are true.

The second property of data follows immediately from the first: data is inherently immutable. Because of its connection to a point in time, the truthfulness of a piece of data never changes. One cannot go back in time to change the truthfulness of a piece of data. This means that there are only two main operations you can do with data: read existing data and add more data. CRUD has become CR.

I've left out the "Update" operation. This is because updates don't make sense with immutable data. For example, "updating" Sally's location really means that you're adding a new piece of data saying she lives in a new location as of a more recent time.

I've also left out the "Delete" operation. Again, most cases of deletes are better represented as creating new data. For example, if Bob stops following Mary on Twitter, that doesn't change the fact that he used to follow her. So instead of deleting the data that says he follows her, you'd add a new data record that says he un-followed her at some moment in time.

There are a few cases where you do want to permanently delete data, such as regulations requiring you to purge data after a certain amount of time. These cases are easily supported by the data system design I'm going to show, so for the purposes of simplicity we can ignore these cases.

This definition of data is almost certainly different than what you're used to, especially if you come from the relational database world where updates are the norm. There are two reasons for this. First, this definition of data is extremely generic: it's hard to think of a kind of data that doesn't fit under this definition. Second, the immutability of data is the key property we're going to exploit in designing a human fault-tolerant data system that beats the CAP theorem.

Query

The second concept in the equation is the "query". A query is a derivation from a set of data. In this sense, a query is like a theorem in mathematics. For example, "What is Sally's current location?" is a query. You would compute this query by returning the most recent data record about Sally's location. Queries are

functions of the complete dataset, so they can do anything: aggregations, join together different types of data, and so on. So you might query for the number of female users of your service, or you might query a dataset of tweets for what topics have been trending in the past few hours.

I've defined a query as a function on the complete dataset. Of course, many queries don't need the complete dataset to run -- they only need a subset of the dataset. But what matters is that my definition encapsulates all possible queries, and if we're going to beat the CAP theorem, we must be able to do so for any query.

Beating the CAP theorem

The simplest way to compute a query is to literally run a function on the complete dataset. If you could do this within your latency constraints, then you'd be done. There would be nothing else to build.

Of course, it's infeasible to expect a function on a complete dataset to finish quickly. Many queries, such as those that serve a website, require millisecond response times. However, let's pretend for a moment that you can compute these functions quickly, and let's see how a system like this interacts with the CAP theorem. As you are about to see, a system like this not only beats the CAP theorem, but annihilates it.

The CAP theorem still applies, so you need to make a choice between consistency and availability. The beauty is that once you decide on the tradeoff you want to make, you're done. The complexity the CAP theorem normally causes is avoided by using immutable data and computing queries from scratch.

If you choose consistency over availability, then not much changes from before. Sometimes you won't be able to read or write data because you traded off availability. But for the cases where rigid consistency is a necessity, it's an option.

Things get much more interesting when you choose availability over consistency. In this case, the system is eventually consistent without any of the complexities of eventual consistency. Since the system is highly available, you can always write new data and compute queries. In failure scenarios, queries will return results that don't incorporate previously

written data. Eventually that data will be consistent and queries will incorporate that data into their computations.

The key is that data is immutable. Immutable data means there's no such thing as an update, so it's impossible for different replicas of a piece of data to become inconsistent. This means there are no divergent values, vector clocks, or read-repair. From the perspective of queries, a piece of data either exists or doesn't exist. There is just data and functions on that data. There's nothing you need to do to enforce eventual consistency, and eventual consistency does not get in the way of reasoning about the system.

What caused complexity before was the interaction between incremental updates and the CAP theorem. Incremental updates and the CAP theorem really don't play well together; mutable values require read-repair in an eventually consistent system. By rejecting incremental updates, embracing immutable data, and computing queries from scratch each time, you avoid that complexity. The CAP theorem has been beaten.

Of course, what we just went through was a thought experiment. Although we'd like to be able to compute queries from scratch each time, it's infeasible. However, we have learned some key properties of what a real solution will look like:

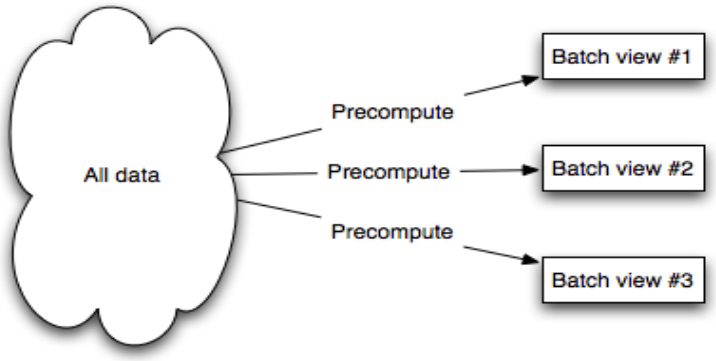
1. The system makes it easy to store and scale an immutable, constantly-growing dataset
2. The primary write operation is adding new immutable facts of data
3. The system avoids the complexity of the CAP theorem by recomputing queries from raw data
4. The system uses incremental algorithms to lower the latency of queries to an acceptable level

Let's begin our exploration of what such a system looks like. Note that everything from here on out is optimization. Databases, indexing, ETL, batch computation, stream processing -- these are all techniques for optimizing query functions and bringing the latency down to an acceptable level. This is a simple but profound realization. Databases are usually made out to be the centerpiece of data management, but really they're one part of a bigger picture.

Batch computation

Figuring out how to make an arbitrary function on an arbitrary dataset run quickly is a daunting problem. So let's relax the problem a little bit. Let's pretend that it's okay for queries to be out of date by a few hours. Relaxing the problem this way leads to a simple, elegant, and general-purpose solution for building data systems. Afterwards, we'll extend the solution so that the problem is no longer relaxed.

Since a query is a function of all the data, the easiest way to make queries run fast is to precompute them. Whenever there's new data, you just recompute everything. This is feasible because we relaxed the problem to allow queries to be out of date by a few hours. Here's an illustration of this workflow:



Precomputation workflow

To build this, you need a system that:

1. Can easily store a large and constantly growing dataset
2. Can compute functions on that dataset in a scalable way

Such a system exists. It's mature, battle-tested across hundreds of organizations, and has a large ecosystem of tools. It's called Hadoop. Hadoop isn't perfect, but it's the best tool out there for doing batch processing.

A lot of people will tell you that Hadoop is only good for "unstructured" data. This is completely false. Hadoop is fantastic for structured data. Using tools like Thrift or Protocol Buffers, you can store your data using rich, evolvable schemas.

Hadoop is comprised of two pieces: a distributed filesystem (HDFS), and a batch processing framework (MapReduce). HDFS is good at storing a large amount of data across files in a

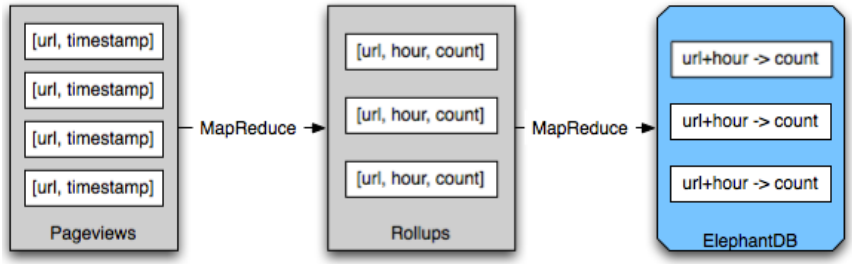
scalable way. MapReduce is good at running computations on that data in a scalable way. These systems match our needs perfectly.

We'll store data in flat files on HDFS. A file will contain a sequence of data records. To add new data, you simply append a new file containing new data records to the folder that contains all the data. Storing data like this on HDFS solves the "Store a large and constantly growing dataset" requirement.

Precomputing queries off of that data is similarly straightforward. MapReduce is an expressive enough paradigm such that nearly any function can be implemented as a series of MapReduce jobs. Tools like Cascalog, Cascading, and Pig make implementing these functions much easier.

Finally, you need to index the results of the precomputation so that the results can be quickly accessed by an application. There's a class of databases that are extremely good at this. ElephantDB and Voldemort read-only specialize in exporting key/value data from Hadoop for fast querying. These databases support batch writes and random reads, and they *do not* support random writes. Random writes cause most of the complexity in databases, so by not supporting random writes these databases are extraordinarily simple. ElephantDB, for example, is only a few thousand lines of code. That simplicity leads to these databases being extremely robust.

Let's look at an example of how the batch system fits together. Suppose you're building a web analytics application that tracks page views, and you want to be able to query the number of page views over any period of time, to a granularity of one hour.



Batch workflow example

Implementing this is easy. Each data record contains a single page view. Those data records are stored in files on HDFS. A function that rolls up page views per URL by hour is implemented as a series of MapReduce jobs. The function emits

key/value pairs, where each key is a `[URL, hour]` pair and each value is a count of the number of page views. Those key/value pairs are exported into an ElephantDB database so that an application can quickly get the value for any `[URL, hour]` pair. When an application wants to know the number of page views for a time range, it queries ElephantDB for the number of page views for each hour in that time range and adds them up to get the final result.

Batch processing can compute arbitrary functions on arbitrary data with the drawback that queries are out of date by a few hours. The "arbitrariness" of such a system means it can be applied to any problem. More importantly, it's simple, easy to understand, and completely scalable. You just have to think in terms of data and functions, and Hadoop takes care of the parallelization.

The batch system, CAP, and human fault-tolerance

So far so good. So how does the batch system I've described line up with CAP, and does it meet our goal of being human fault-tolerant?

Let's start with CAP. The batch system is eventually consistent in the most extreme way possible: writes always take a few hours to be incorporated into queries. But it's a form of eventual consistency that's easy to reason about because you only have to think about data and functions on that data. There's no read-repair, concurrency, or other complex issues to consider.

Next, let's take a look at the batch system's human fault-tolerance. The human fault-tolerance of the batch system is as good as you can get. There are only two mistakes a human can make in a system like this: deploy a buggy implementation of a query or write bad data.

If you deploy a buggy implementation of a query, all you have to do to fix things is fix the bug, deploy the fixed version, and recompute everything from the master dataset. This works because queries are pure functions.

Likewise, writing bad data has a clear path to recovery: delete the bad data and precompute the queries again. Since data is immutable and the master dataset is append-only, writing bad data does not override or otherwise destroy good data. This is in

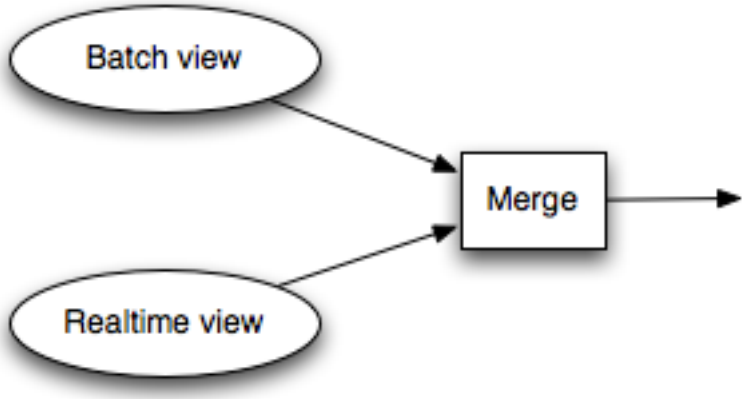
stark contrast to almost all traditional databases where if you update a key you lose the old value.

Note that MVCC and HBase-like row versioning do not come close to this level of human fault-tolerance. MVCC and HBase row versioning don't keep data around forever: once the database compacts the row, the old value is gone. Only an immutable dataset guarantees that you have a path to recovery when bad data is written.

Realtime layer

Believe it or not, the batch solution almost solves the complete problem of computing arbitrary functions on arbitrary data in realtime. Any data older than a few hours has already been incorporated into the batch views, so all that's left to do is compensate for the last few hours of data. Figuring out how to make queries realtime against a few hours of data is much easier than doing so against the complete dataset. This is a critical insight.

To compensate for those few hours of data, you need a realtime system that runs in parallel with the batch system. The realtime system precomputes each query function for the last few hours of data. To resolve a query function, you query the batch view and the realtime view and merge the results together to get the final answer.

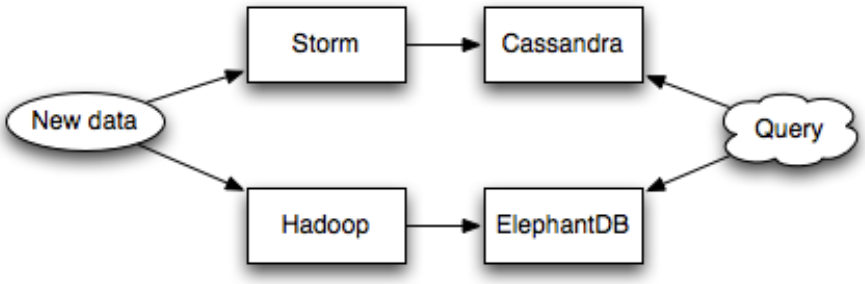


Computing a query

The realtime layer is where you use read/write databases like Riak or Cassandra, and the realtime layer relies on incremental algorithms to update the state in those databases.

The analog to Hadoop for realtime computation is Storm. I wrote Storm to make it easy to do large amounts of realtime data processing in a way that's scalable and robust. Storm runs infinite computations over streams of data and gives strong guarantees on the processing of the data.

Let's see an example of the realtime layer by going back to the running example of querying the number of page views for a URL over a time range.



Example batch/realtime architecture

The batch system is the same as before: a batch workflow based on Hadoop and ElephantDB precomputes the query for everything but the last few hours of data. All that's left is to build the realtime system that compensates for those last few hours of data.

We'll roll up the stats for the last few hours into Cassandra, and we'll use Storm to process the stream of pageviews and parallelize the updates into the database. Each pageview leads to a counter for a `[URL, hour]` key to be incremented in Cassandra. That's all there is to it -- Storm makes these kinds of things very simple.

Batch layer + realtime layer, the CAP theorem, and human fault-tolerance

In some ways it seems like we're back to where we started. Achieving realtime queries required us to use NoSQL databases and incremental algorithms. This means we're back in the complex world of divergent values, vector clocks, and read-repair.

There's a key difference though. Since the realtime layer only compensates for the last few hours of data, everything the realtime layer computes is eventually overridden by the batch layer. So if you make a mistake or something goes wrong in the realtime layer, the batch layer will correct it. All that complexity is transient.

This doesn't mean you shouldn't care about read-repair or eventual consistency in the realtime layer. You still want the realtime layer to be as consistent as possible. But when you make a mistake you don't permanently corrupt your data. This removes a huge complexity burden from your shoulders.

In the batch layer, you only have to think about data and functions on that data. The batch layer is really simple to reason about. In the realtime layer, on the other hand, you have to use incremental algorithms and extremely complex NoSQL databases. Isolating all that complexity into the realtime layer makes a huge difference in making robust, reliable systems.

Additionally, the realtime layer doesn't affect the human fault-tolerance of the system. The append-only immutable dataset in the batch layer is still the core of the system, so any mistake can be recovered from just like before.

Let me share a personal story about the great benefits of isolating complexity in the realtime layer. I had a system very much like the one I described here: Hadoop and ElephantDB for the batch layer, and Storm and Cassandra for the realtime layer. Due to poor monitoring on my part, I woke up one day to discover that Cassandra had run out of space and was timing out on every request. This caused my Storm topology to fail and the stream of data to back up on the queues. The same messages kept getting replayed (and kept failing) over and over.

If I didn't have a batch layer, I would have been forced to scale and recover Cassandra. This is non-trivial. Even worse, much of the database was likely inaccurate due to the same messages being replayed many times.

Fortunately, all this complexity was isolated in my realtime layer. I flushed the backed up queues into the batch layer and made a fresh Cassandra cluster. The batch layer ran like clockwork and within a few hours everything was back to normal. No data was lost and there was no inaccuracy in our queries.

Garbage collection

Everything I've described in this post is built upon the foundation of an immutable, constantly growing dataset. So what do you do if your dataset is so large that it's impractical to store all data for all time, even with horizontally scalable storage? Does this use case break everything I've described? Should you go back to using mutable databases?

No. It's easy to extend the basic model with "garbage collection" to handle this use case. Garbage collection is simply a function that takes in the master dataset and returns a filtered version of the master dataset. Garbage collection gets rid of data that is of low value. You can use any strategy you want for garbage collection. You can simulate mutability by only keeping the last value for an entity, or you can keep a history for each entity. For example, if you're dealing with location data, you may want to keep one location per person per year along with the current location. Mutability is really just an inflexible form of garbage collection (that also interacts poorly with the CAP theorem).

Garbage collection is implemented as a batch processing task. It's something you run occasionally, perhaps once per month. Since garbage collection is run as an offline batch processing task, it doesn't affect how the system interacts with the CAP theorem.

Conclusion

What makes scalable data systems difficult isn't the CAP theorem. It's a reliance on incremental algorithms and mutable state that leads to complexity in our systems. It's only recently with the rise of distributed databases that this complexity has gotten out of control. But that complexity has always been there.

I said in the beginning of this post that I would challenge your basic assumptions of how data systems should be built. I turned CRUD into CR, split persistence into separate batch and realtime systems, and obsessed over the importance of human fault-tolerance. It took a lot of hard-earned experience over the years to break my old assumptions and arrive at these conclusions.

The batch/realtime architecture has a lot of interesting capabilities that I didn't cover yet. It's worth summarizing some of these now:

1. **Algorithmic flexibility:** Some algorithms are difficult to compute incrementally. Computing unique counts, for example, can be challenging if the sets of uniques get large. The batch/realtime split gives you the flexibility to use the exact algorithm on the batch layer and an approximate algorithm on the realtime layer. The batch layer constantly overrides the realtime layer, so the approximation gets corrected and your system exhibits the property of "eventual accuracy".
2. **Schema migrations are easy:** Gone are the days of difficult schema migrations. Since batch computation is at the core of the system, it's easy to run functions on the complete dataset. This makes it easy to change the schema of your data or views.
3. **Easy ad-hoc analysis:** The arbitrariness of the batch layer means you can run any query you like on your data. Since all data is accessible in one location, this is easy and convenient.
4. **Self-auditing:** By treating data as immutable, you get a self-auditing dataset. The dataset records its own history. I've discussed how important this is for human fault-tolerance, but it's also super useful for doing analytics.

I don't claim to have "solved" the Big Data space, but I've laid down the framework for thinking about Big Data. The batch/realtime architecture is highly general and can be applied to any data system. Rather than give you a fish or a fishing rod, I've shown you how to make a fishing rod for any kind of fish and any kind of water.

There's lots more work to do to improve our collective ability to attack Big Data problems. Here are some key areas of improvement:

1. **Expanded data models for batch-writable, random-read databases:** Not every application is supported by a key/value data model. This is why my team is investing in expanding ElephantDB to support search, document databases, range queries, and more.
2. **Better batch processing primitives**: Hadoop is not the end-all-be-all of batch computation. It can be inefficient for certain kinds of computations. Spark is an important project doing interesting work in expanding the MapReduce paradigm.

3. **Improved read/write NoSQL databases**: There's room for more databases with different data models, and these projects in general will benefit from more maturation.
4. **High level abstractions:** One of the most interesting areas of future work is high level abstractions that map to a batch processing component and a realtime processing component. There's no reason why you shouldn't have the conciseness of a declarative language with the robustness of the batch/realtime architecture.

A lot of people want a scalable relational database. What I hope you've realized in this post is that you don't want that at all! Big data and the NoSQL movement seemed to make data management more complex than it was with the RDBMS, but that's only because we were trying to treat "Big Data" the same way we treated data with an RDBMS: by conflating data and views and relying on incremental algorithms. The scale of big data lets you build systems in a completely different way. By storing data as a constantly expanding set of immutable facts and building recomputation into the core, a Big Data system is actually easier to reason about than a relational system. And it scales.

Commentary

shearic:
Great post. Jives very well with what we're doing at Yieldbot at a high level, at least on the batch side and with a little different tech, and what we're headed toward on the realtime piece as well.

In thinking about all of this the hardest problem I've found to deal with is that of providing sorted results. Sorting is interesting because it is basically expressing constraints that aren't absolute but instead is a function of the results being returned (ie, the entries are compared to each other not against some explicit constraint).

In my mind sorting is the most challenging aspect of the batch+realtime mashup approach to satisfying a query. Not in the case where it is feasible to read in the entire datasets from both the batch and query side and then sort the results at the mashup point, but where there's a large dataset where it is only feasible to read in some of the results from either side (and therefore basically each side being sorted) and then mash it up.

It seems like the only reasonable choice here is to sacrifice accuracy of what the sort order of the entries really should be. Interested on your thoughts on that particular aspect of querying.

1. **nathanmarz**:
   Part of the difficulty you're facing is the immaturity of the existing tooling. K/V data model is not good enough when you want to access a large sorted list. The work we're doing for EDB 2.0 will address these kinds of more advanced data models. EDB 2.0 will make it easy to implement a key -> sorted list batch-writable database, and then you quickly query for a sub-portion of the list. EDB-search will also provide similar functionality.

   The realtime aspect is always challenging, and what you do for sorting in realtime is highly dependent on the intricacies of your query. Distributed RPC might help with something like this, depending on how intense is the computation for resolving the query.
   i. **shearic**:
      It's not so much the sorting of the batch or the sorting of the realtime sides taken on their own that I think is the issue (we don't use strictly K/V for our batch data for this reason). Where the issue lies is when you want to join the two sides.

      If you have data where the sort on the realtime side is an order much different than the sort on the batch data, such that the sort order on the combined data should be much different than the sort order batch data, it's that issue that I'm describing.

      The problem lies in the nature of not having all the data in one place where it can all be sorted together.

      What it speaks to is the accuracy not of any particular piece of data, but the accuracy of how the data are related to each other.
      a. **nathanmarz**:
         I think I'd have to know more of the specifics of your data/queries to give a more useful response. Feel free to email me or ping me on IRC.

---

kevinmarks:
The old model was MySQL for the batch layer and memcached for the realtime layer; this uses Hadoop and Cassandra to replace them

---

jeskeca:
This is some kind of functional-is-equivalent-to-imperative exploration which doesn't change any of the CAP constraints. How do you handle consistency related operations, such as "withdrawl $100 only if the user can afford it"? We can model the withdrawl as the "creation of a withdrawl record", and a batch-or-realtime system can integrate the withdrawls into a current-bank-balance… but if we want to avoid that bank balance going negative, we need to choose consistency-over-avaialbility, and that isn't changed by this alternate way of applying updates.

1. **nathanmarz**:
   AFAIK, banks choose availability (and thus eventual consistency). This is a pragmatic decision since they lose business if the system ever becomes unavailable. Banks are the classic example which really need an immutable dataset that tracks the history of each entity (also called auditing). They workaround the consistency problem by allowing you to withdrawal more than you have in your account and then charge you fees for doing this: http://en.wikipedia.org/wik...
   i. **Henrique**:
      Banks are a good case of changing the business to adapt to the constraints, rather than the other way around.
   ii. **nathanmarz**:
      The most relevant part of that link is this part: "If the ATM is unable to communicate with the cardholder's bank, it may automatically authorize a withdrawal based on limits preset by the authorizing network."
      That means that banks are highly available, eventually consistent systems.
   iii. **Tim McCormack**:
      Banks do not enforce consistency in the short term – you can do all sorts of crazy (illegal) money tricks if you move quickly.

Guest:
It seems that this approach completely fails at the most interesting form of data: state and/or transactions. When you collect data for statistical analysis, it doesn't matter if some data points are only included into a later calculation. But if you are saving the state of something, then the game with the immutability by tacking on a timestamp doesn't help, in particular:

> Things get much more interesting when you
> choose availability over consistency. In this case,
> the system is eventually consistent without any of
> the complexities of eventual consistency. (…)

just doesn't work because when your new state, i.e. the next piece of data that you add to the database, depends on the saved state, or even any query!

1. **nathanmarz**:
   You can still choose full consistency, in which case you must accept that the system will sometimes be unavailable. Alternatively, you can take a different approach like what banks do, as I commented above. (http://nathanmarz.com/blog/...)

Heath Borders:
Great post!
I still feel like a vector clock of some kind would be necessary even we choose eventual consistency because our systems will be order-dependent, and we'll need to know when we have conflicts.

For example, I have an online presence system that allows a user to either login or logout. My data is partitioned across servers A and B.1. User login (data propagated to servers A and B)2. Network split3. User logout (data propagated to server A)4. User login (data propagated to server B)5. User logout (data propagated to server B)

I'm not concerned with the real-time implications of these events, but a correctly audited timeline. Without a vector clock to show a conflict, my data could reflect 2 different event streams:1. login, logout, login, logout2. login, login, logout, logout

The vector clock could simply be incorporated into the state records to assist in merging.I'm new to the big data space, and haven't implemented any of these ideas in a production setting, but I'm still interested in the subject. Please forgive me if this is a noob question, or if incorporating vector clocks into immutable records is standard practice.

1. **nathanmarz**:
   You don't need vector clocks in the batch layer. Every piece of data is associated with the moment in time that it occurred, and this provides the ordering you need. So the login/logout data would be written as:

   Login as of time 12.
   Logout as of time 23.
   Login as of time 34.
   Logout as of time 4

(For the realtime layer you do need vector clocks, but I think you understood that already)

---

davidells:
The title claim of this post is overstated, but I appreciate the general reasoning about the problem and the concrete architectural "proof".

You do get the availability, and gain a lot by isolating the burden of consistency to a problem of availability of the real-time layer.

But such a system doesn't beat the CAP theorem, since you are subject to availability failures (or other mistakes) in the real-time system, with only eventually consistent (but highly available) data behind that in the batch system.

So in the end, you're still bound by availability guarantees in order to provide consistent data, just as the CAP theorem says, right? (I may be totally wrong here.)

1. **nathanmarz**:
   Regarding "beating the CAP theorem", this doesn't mean "eliminating" it. The interaction between the CAP theorem and mutability causes a lot of complexity that you avoid or greatly subdue by basing a system on immutability.

   The consistency vs. availability tradeoff is still yours to make in the realtime layer. If you choose consistency, then your realtime layer will sometimes be unavailable (although you can always serve up the batch portion of the views if you want, in which case you have an interesting form of eventual consistency).

   If you choose availability in the realtime layer, then you still have to deal with vector clocks and read-repair. You should of course try to make your code correct, but when you make a mistake, you won't permanently corrupt the database.

   Another important point is that since the realtime layer only deals with a few hours of data, the realtime db cluster can be relatively small than if it were dealing with the entire dataset. So you might only have a 6 node Cassandra cluster instead of a 20 node Cassandra cluster. This makes it less likely for events to occur that would trigger an inconsistent situation.
   i. **davidells**:
      Thanks for your response. I would say minimizing the typical trade-offs we're used to seeing with CAP and mainstream tools qualifies as beating it,

actually.

There are pieces of what you outlined that ring familiar with a lot of folks, regarding a slower or inconsistent store underneath a smaller caching layer. We did this for a while at scorm.com back when Amazon's SimpleDB was only available as eventually consistent, and we judiciously used memcached to cover our exposure where needed. (That's all on a much smaller scale than what you're addressing here, and I don't see SDB as a very great choice, but it works at our current size.)

I would also agree from experience that immutable data has far fewer headaches than mutable data. I haven't thought of it in a general way like this, though. I'm still wrapping my head around the implication of that.

I appreciate your time and effort to not just wax some theoretical, but lay out the architecture for a scalable system that you've actually run, with success, for a real purpose. It's a lot less common than much of the codeless theoretical musings you find on the net. Hey, not to mention thanks for all the Storm/ElephantDB/etc code too! This article has me interested to scope out and play with some of that stuff now…

Sebastien Diot:
Working on a system where "the past CAN be changed", I might see issues that other don't. That is, things gets messy if "One cannot go back in time to change the truthfulness of a piece of data." doesn't hold for a particular application.

In my specific case, I work on a payroll system. Almost all data items have a value which is associated with a month (some are daily, quarterly, …). Payroll is monthly here. Health insurance is obligatory and deducted at the source. The problem is, that health insurances are allowed to say that "actually our rates went up by 2% *3 months ago*!"

So, you are in April, and you have a value for January, but that value now differs from what it originally was in January, and then you have to recompute the payroll for January to March, and adjust the result of April based on the

differences found.

In other words, not only our data are timestamped, but an attribute might have several values for *the same timestamp*, entered at different times. That is to say, we have data under two-dimensional timestamps. Still makes my head spin after working on it for years.

While we're not a BigData case (our DB vendor likes to remind us of that fact regularly; I think they take issues that we use highly effective compression (1/100 of what a straightforward implementation would produce) on our data by hacking their driver), I still have enough experience to say that you neglected an essential problem:

How do you get a reliable timestamp? Our experience is that you can NEVER use the system clock of your users, because it is often *years* in the past or the future. And as soon as you have more than one server, you can sure as hell be that their clocks are going to diverge. Especially so if you had 100s or 1000s of them. So in the end, how do you know that data piece A was really created before data piece B, if the timestamps are so close that you can't be sure that it might not actually have been the other way around? You can't either rely on an incrementing counter, without using some kind of global lock.

So to resume it, how do you ever get the reliable (and unique) timestamps that your theory is based on, in a distributed system where you can neither rely on the client or the server clock?

1. **nathanmarz**:
   First of all, thanks for the great comment. You bring up a lot of good points.

   I think that your use case can still be modelled with immutable data. Your health insurance rate data should say "As of X time, I know the rate to be Y% after Z time". Since your functions take in all the data at once, and you have the timeline of events, you can reconstruct at any point what payments are supposed to go out.

   Getting a reliable timestamp is definitely an interesting problem, and it gets more challenging the more fine-grained you need the accuracy to be. Obviously an application that's allowed to be inaccurate by one minute is much easier than one that needs to be accurate to within a millisecond.
   There's a few ways to approach the timestamp problem. The first is to have all the machines synchronize their

times with a reference authority, whether using NTP or another solution. This can keep the clocks drifting too much, but may run into problems during partitions. As long as your required resolution isn't too small, this should work fine. Another way to approach the problem, and a more complex one, is to figure out which partitions of your data need total ordering. It's unlikely that the dataset as a whole needs a total ordering. For example, you may need a total ordering for each individual person in your system, but you don't need a total ordering across partitions (or at least have time resolution requirements that are more lax). So you just have to make sure a single machine is responsible for timestamping any given entity at a time (something that Storm makes easy with "fields groupings").

Ziliang Peng:
I wonder what's the difference between create-only-data and write-ahead-log? I think the WAL is just another way to append-only update of the data. So it appears to me that they are just the same. What's more, how could you synchrounize the new created data to all the nodes?

1. **nathanmarz**:
   WAL is a method for updating mutable state on a single server in a reliable way. WAL are meant to be compacted once the state has been safely updated. What I'm proposing is a rejection of mutable state altogether – every data value is first-class and available to queries, not just the "latest" one.
   i. **Ziliang Peng**:
      Well, you are right. They are different. But what I mean is that, the implementations of them are just very similar. You append something for a write and never change it. you could get the current state from following the history.

Vladimir Sedach:
You mentioned MVCC, but MVCC, like your idea (and pretty much most distributed systems work) is based on David Reed's NAMOS system. What you're essentially proposing is NAMOS with a streaming OLAP thrown over it.

The proposed system doesn't solve or even sidestep the consistency problem at all - even if you record transactions as atomic records in this system, you're still

going to need a retry mechanism if the timestamp in the database is newer than the data you were working with.

All this is proposing is to get rid of the idea of transactions and atomicity. That's why it seems that you're sidestepping consistency. Mutable state isn't the problem in databases (that's what MVCC solves), it's the atomicity.

A whole other issue is that you're still working with a single database that all the data feeds into and views come out of. Your solution is fine for that scenario (as long as transactions aren't needed), but a peer-to-peer distributed system needs the vector clocks you mentioned.

Your solution works, but for a limited space of distributed systems (ones that have, as you note many times, a consistency time-scale of "a few hours"). It's disingenious to claim that you've beaten CAP for all distributed systems applications.

1. **nathanmarz**:
   Nor do I claim that I'm sidestepping "the consistency problem." "Beating the CAP theorem" does not mean "eliminating it". It means avoiding or mitigating the complexity of dealing with the tradeoff between consistency and availability.

   Systems based on incremental algorithms and mutable state are difficult to reason about, especially once you throw the CAP theorem into the mix. Basing a system on immutability and batch computation gives you human fault-tolerance and makes it easy to reason about the system.
   i. **Paolo Giarrusso**:
      I find your approach interesting, also because I'm a functional programmer working on something close, but how would your system handle two concurrent queries like the following:

      INSERT INTO table1(counter, otherField)
      VALUES(SELECT MAX(counter) + 1 from table1, A)

      with two different values for A? The goal of this query is that counter is a primary key. If the query runs atomically, that's guaranteed. But if I understand your proposal correctly, since you do not have atomicity, N queries running at once might add N different records with the same counter. So I think this query is also related to Vladimir Sedach's question—but even otherwise, I'm curious about your answer.The particular query is pointless, but it looks plausible that meaningful atomic queries exist which would run into the same problem.On an unrelated topic I was also thinking of CouchDB's immutability, but it seems that the relation is quite superficial:http://eclipsesource.com/bl...

a. **nathanmarz**:
What you're describing is the "traditional way" of doing things, where the database represents the "current state of the world" and is directly mutated. In the approach I propose, you never do that. Data are independent facts, and something like "count" is a query on those facts. For example, if you're tracking number of pageviews, instead of storing a counter that you increment, you store the actual pageviews themselves.

Kent Beck:
Nathan,
I like the direction here. Most data is static, so put it in a store optimized for static data. Put the rest in a store optimized for change. Provide a unifying interface to both.
What I don't understand is how this avoids the problem of inconsistent writes during a partition. Even with immutable data, if A and B can't see each other, they can both write new values. Later processing will see both values. What am I missing?
Kent

1. **nathanmarz**:
I'm not 100% sure I understand the question you're asking, so if what I say is unclear or off base it would help to see an example of the situation you're describing.

   The key here is the difference between data and queries (as I've defined them). Data are axioms while queries are deductions from data. Each piece of data is a logical fact that stands on its own as something true. The right way to formulate a data system is such that each piece of data is independent from one another. So by definition, a situation like you described where there is a dependency between individual pieces of data is not possible.

   That's somewhat abstract, so let's consider a few examples. Suppose you want to display the number of upvotes on a social news site. The most "axiomatic" way to separate the query from the underlying data is to treat individual upvotes as separate pieces of data, and formulate the "number of upvotes" as a query on that

data.

Each individual upvote is true independent of the other upvotes and so satisfies my definition of data. During a partition, you may not see some upvotes, but once the partition goes away queries will be correct.

Another example is the one I used in the post about updating a person's location. A person's location can be treated as immutable by tying it to a moment in time. The fact that Sally lives in Chicago now does not change the fact that she lived in Atlanta 6 months ago. These location dataunits are independent. If Sally's location is written by multiple writers during a partition, that's fine since everything is independent. The "current location query" would simply take the location with the most recent timestamp.

Blake Smith:
Great post Nathan! This has certainly expanded my view on how I see data and the databases that sit on top of it.

I'm curious how the 'merge' part of your data fetching works in practice. It seems complicated to have code that fetches from both and merges for every data query you're trying to present to the user. Does each data query have to understand how to fetch data in both stores and merge them on an individual basis, or do you use some sort of abstraction layer that generalizes pulling from both data stores and merging the results? I guess what I'm asking is, is a general merge strategy possible, or does each query have to know how to merge itself?

1. **nathanmarz**:
   The merge strategy usually has something to do with what kind of aggregation you're doing on the data. So a count always uses the same merge strategy, for example, by splitting the count before and after a certain time and adding them together. An important part of a higher level abstraction on top of batch/realtime will be internalizing the merge logic within the aggregators so that this stuff is automated as much as possible.