Competitive Edge on Big Data Scalability with In-House Solutions

M-BRAIN WHITE PAPER

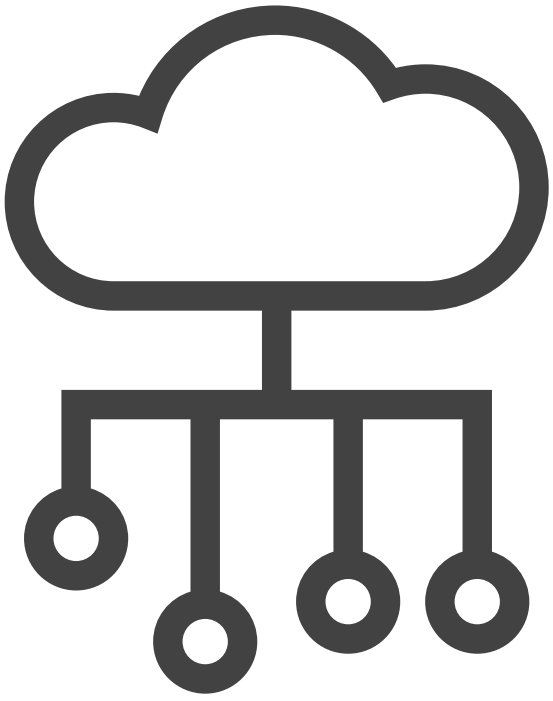
Contents

ABOUT THE AUTHOR

Antti Tuominen, Senior R&D Engineer at M-Brain, has worked for several years with machine learning, distributed data processing, product design and development. Antti loves writing software that solves problems that are hard to solve.

Introduction

This white paper is a case study of switching from a NoSQL database to a custom in-house data storage and mining platform nicknamed Madoop, which is currently used by M-Brain's M-Adaptive. Is it worth it to build yet another database? In general, the answer of course is no, which is evident looking at the number of database systems versus the number of happy database users. The emphasis is on the word **general**. It is a tall order to build a new database system that would perform a large variety of tasks significantly better than the current offerings. However, a system that works extremely well for the specific things we do is not such a daunting task.



Generic versus specific is a reoccuring theme throughout this paper: utilizing the properties of our data and how it is created and used opens up plenty of opportunities for efficiency. Think of it as lossless data compression: no method can exist that losslessly compresses everything. Yet compression is used widely with great success. This works because only an infintesimally small subset of all the possible sequences of zeros and ones will ever be attempted to be compressed by anyone. The compressor can ignore efficiency on almost all data since it's highly unlikely that no-one will ever have that and focus on the small subset of data that people actually have.

Our goal is similar: we need to be as good as possible processing the data we have the way we process it. Naturally it is important to be prepared for likely future data and processing but beyond that it doesn't matter. It is of no consequence how a system performs on the tasks it never does.

In the next chapter we discuss the design goals of the system and how they are achieved. Later, we discuss more general aspects and best practices when building big data systems.

Design aspects and solutions

Besides things needed, things that are not needed are equally important. Each feature comes with a cost and paying (in performance) for what is not needed should be avoided. The following goals can also be viewed as a list of properties that should be maximally exploited for maximal gains.

High availability. This is both important and costly. Madoop's data is recoverable from backups and other sources so avoiding downtime and data recovery efforts is the most vital goal. It's only a matter of time before a hardware failure occurs, and surviving a single server crash covers a lot of ground ensuring high availability. It is also notable that this can be done independently from other services. To protect against network outages one could replicate data between data centers, but this is costly and doesn't improve availability from the customer's perspective unless the entire stack of services needed for a particular product is replicated in the same way. Madoop does the simplest thing, it stores each piece of data on two separate servers.

Always online design. Some databases require taking entire shards or such large portions of the system offline for routine maintenance tasks such as compacting. In Madoop, the fraction of data that currently gets briefly blocked for such operations varies from 0.0001% to 0.0007%. Compacting and similar tasks can be (and are) carried out routinely with little overall load to the system. This, accompanied by data replication and automatic data healing during server downtime makes it possible to do even Madoop version upgrades with zero downtime.

Little consistency guarantees. At first it may sound a bit strange to want less consistency but when one sees the price tag involved it starts making perfect sense. We used to have trouble running map-reduce jobs efficiently in all our shards and replicas because the database insisted on waiting until the data becomes consistent across the servers. This is rarely the case for us, our media harvesting and processing systems provide a never-ending stream of data updates.

Looking at it from a different perspective, our data is never up-to-date considering that there always are relevant media items "out there" that haven't yet made it to the system. Even worse, some of those never will, so the numbers we crunch cannot be deemed exactly accurate in any absolute sense. Madoop's consistency model guarantees that the numbers still make sense. Looking at just Madoop, the consistency provided is a bit less than what is considered eventual consistency. In practice it is stronger than monotonic strong eventual consistency due to the fact that the software that uses Madoop is not arbitrary, but built in-house as well.

A database has to operate somewhat blindly when it provides consistency as it always needs to prepare for the worst. Sadly, for any given application, almost all of the possible worst case scenarios never occur. This

By utilising our data access patterns we have configured the system in a way that allows us natural slicing and dicing of the data using time intervals and data subsets without needing an index to locate the records.

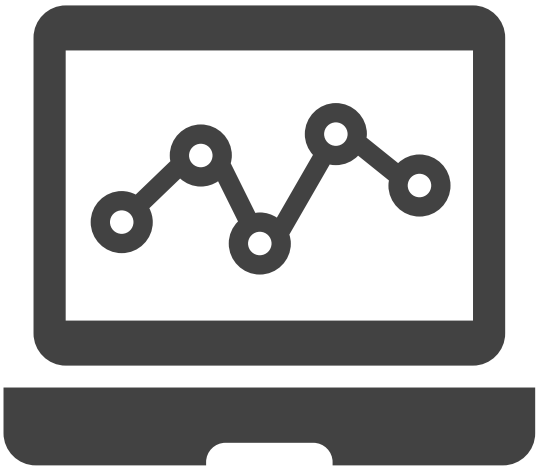
is not luck, it is just how the application's business logic works. And more importantly, how it doesn't work. Madoop provides the basic consistency it can efficiently provide and the (surpisingly little) extra needs are handled by upper application layers which can apply more knowledge about the operations performed. We do have uses for stricter consistency and use databases which provide such features. This is not big data, however, and the cost of those guarantees is not a problem.

Time as a first class citizen. With media data, everything involves a point in time: scandals, disasters, elections, publications, product launches, advertising campaigns and so on. A typical life span of a media item is that soon after publishing it gets updated, commented upon, referred to and after some time has passed it becomes a static lump of information. Unsurprisingly our data access patterns are hugely affected by the publishing time of an item.

Data subsets inside the data. Our data contains subsets by nature. While not completely independent, records within a subset are a lot more connected than records between subsets. Thus a typical calculation run involves from one to a handful of subsets. The important thing here is that it doesn't involve hundreds of thousands of other ones.

Big queries. Our queries which select the data for map-reduce are not complex but they are big in the sense that a lot of record fields are used for any particular query and a different large field set for another. Without a lot of indexing data access was too slow and soon our indices were responsible for roughly 40% of the size of the entire data set. When the indices grew too big to fit into memory, performance plummeted and the burden of additional hard drive seek operations of accessing indices and actual data simultaneously made things even worse.

Usually a database uses a fixed on-disk file format which provides only sequential access through the data and relies on indices for locating data fast by given criteria. Entire books have been written on performance tuning such systems. We decided to sidestep the problem by making Madoop highly configurable both on how it organises data on disk and between servers. By utilising our data access patterns we have configured the system in a way that allows us natural slicing and dicing of the data using time intervals and data subsets



without needing an index to locate the records. A little extra pruning in memory is a small price to pay for efficiently reading large continuous chunks of data rich in relevant records. As an added benefit, without indices we have more memory available for caching data and upsert/delete operations do not need to spend time updating indices.

Compression. Once upon a time there were two choices: uncompressed bigger data or compressed smaller but slower data. A relatively new class of compression methods like LZ4 that focus on speed instead of maximal compression, and the widened performance gap between processors and hard drives has changed the game. In many cases now the choice is between optimizing the use of the slowest component and paying for it with extra CPU load or waiting longer for the slowest component and not being able to do anything useful with the CPU. Madoop groups small records together and pump-primes a fast compression method with suitable data to maximally utilise the repetitive nature of records. Combined with a compact file format and the lack of indices we achieve a disk footprint of mere 10% of the previous system. The price to pay is relatively slow random access to individual data items but the gains in batch processing are far more important, and even while using this setup we would run out of I/O capacity before running out of CPU.

Implementation Considerations on Big Data Computing

During the past two decades all hardware has taken big leaps forward but not all parts of the system have progressed equally. Networking and processors have improved enormously while hard drives are still very slow and main memory is lagging behind enough to warrant multiple layers of caching in the CPU to mitigate the speed difference. The situation is not improved at all by the "curse" of big data: the bigger the data, the closer to linear the complexity of computation needs to be unless one has a rare luxury of affording very long computation times. In other words, big data tends to favor low computation-to-data ratios while computers are more optimal in the opposite scenario. Solid state drives are starting to provide a much waited performance leap to storage but the biggest challenge for efficient use of all hardware still revolves around keeping all CPU cores fed with data.

One way to balance the load from I/O to computation is taking the computation where the data is and not vice versa. Map-reduce is a natural computing method for this. It also allows splitting data into suitably sized chunks of input which presents the opportunity for smart caching of results. With caching the computation stays simple and stateless from the client's point of view and repeating work already done can be avoided for the input chunks that remain unchanged. Being smart in this case means leaving it up to the client application to decide whether caching makes sense for a particular task.

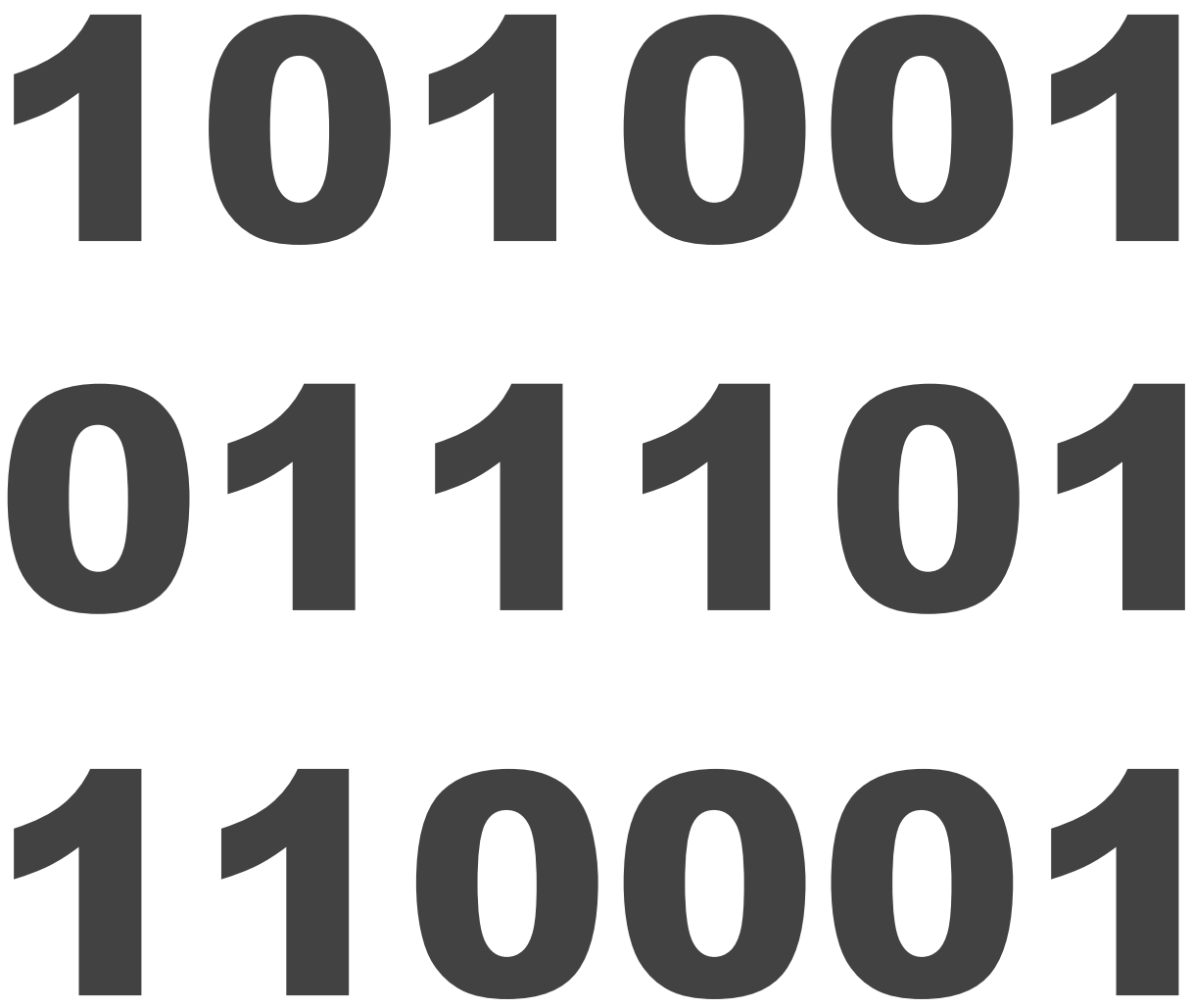
The good old KISS principle is very much recommended, accompanied by a dose of Gall's Law stating that every working complex system has evolved from a working simple system. While this is a general good practice in software engineering it has an additional emphasis in this particular domain. If a regular computer program crashes, it's annoying but usually not a big deal. If a data storage system corrupts data, it's a disaster.

While implementing functionality for current needs is hard enough to need an incremental approach to succeed, preparing for known things to come is a bit harder but still makes sense. Predicting the future is usually too hard to be successful: plenty of time can be wasted catering in advance to imagined things which will never occur. Guessing what will change and how is rarely fruitful but making the system **changeable** for the

unknown but inevitable changes is essential. Designing for change is successful when yet another unexpected feature turns out to be easily implementable.

One big source of diminishing returns when scaling distributed systems is the need for taller hierarchies and increased need for messaging between machines. A lot of care should be taken to avoid machine-to-machine interaction and being efficient in what cannot be avoided. Web services and RESTfulness may sound trendy and fashionable but something faster should be done instead. After all, HTTP is a protocol that is concerned about interoperability between heterogenous systems in a wide variety of use cases. For outside access a REST interface may be desireable but there's hardly a wide variety of uses inside a single application and even less need for interoperability. Bare sockets are too tiresome to use directly, Madoop opted for *ZeroMQ*.

When processing large volumes of data one should focus on exactly that: handling large volumes. Designing for throughput is key. Low latency is nice but it cannot be achieved if the system cannot handle the load.



Conclusion

Horizontal scalability comes with an overhead: if the amount of servers is doubled one gets less than twice the performance, so efficiency matters. In the end, all inefficiencies are multipliers on the hardware budget or the cloud computing bill. From a different perspective, a system twice as fast and twice as compact would effectively mean that CPUs became twice as fast while hard drives and memory got twice as big without doubling the budget. For hardware, that is. The cost is in spending time, effort and expertise in creating such a system.

Before committing to building an in-house system there's homework to be done. One must know the data and how it is processed and what are the current difficulties. If the proposed new system resembles too much an existing system one must be extra careful ensuring the potential benefits really exist. Those usually come by exploiting domain knowledge that existing systems cannot apply for solving a particular problem. It's a high risk - high reward scenario, the potential rewards must be there to make it worthwhile to take the risk.

Estimates, potential, plans and projections are good but there are bound to be surprises along the journey. It's a good idea to quickly build a proof of concept system of the core ideas to verify that they also work in practice. If the project is going to fail, it should fail as early as possible.

One benefit of an in-house solution is easy to disregard. Particular needed changes or new features to a widely used more general purpose package can be difficult to implement and even more difficult to get accepted or prioritised by the development team. In an in-house system those can always appear at the top of the roadmap and be worked on by people who are already familiar with the internals.

TO THE READER

This white paper provides you with insights on replacing a general purpose database-based big data system with custom-made software. It helps you to evaluate whether going for an in-house built system is the right solution for you and offers best practices for creating such a system. In addition, you are encouraged to make use of the new opportunities provided by the unconventional division of labor between different parts of your system.

We hope you enjoyed reading this white paper. If you want to learn more about M-Brain R&D, please visit our website.