w4112p1

Cody De La Vara, Evan Drewry

April 3, 2013

Introduction

The relational model, initially proposed by Edgar F. Codd in 1969, has dominated the world of databases ever since.[5] Prior to this tipping point, the database systems in popular use were nothing like the ones we use today–the notion of tables and relations had not even surfaced until

Chapter 1

Main Memory Databases

Traditional database systems store data on disk predominantly for these two reasons: (1) disks are cheap and have high capacity, and (2) writing to a disk is permanent, which helps in satisfying the D (durability) requirement of ACID. However, accessing and storing data on disk is also very slow when compared with volatile random-access memory. The majority of execution time for most transactions in a traditional system is spent waiting for data to be brought into RAM since the actual time it takes for the CPU to process the data tends to be orders of magnitude faster than the I/O wait times. As a result of this, the optimizations and algorithms in traditional disk-resident databases are designed primarily to reduce the amount of disk I/O required to execute a transaction.

In order to overcome the performance implications of storing data on disk, a number of database systems have been implemented such that the entire database permanently resides in RAM. These main-memory database systems, or MMDBs, can be much faster and have much higher throughput than traditional systems because the penalties of disk I/O in query processing are completely eliminated. And, with the consistent cost decreases and capacity increases of random access memory that was once prohibitively expensive, these systems are becoming more and more practicable for many different use cases that require super-fast response times.

While seemingly a simple change to already existing database systems, MMDBs actually represent a radical departure from traditional database implementations. This is because where traditional algorithms are designed for optimizing disk accesses, algorithms for an in-memory database system must take completely different factors into account. In his 1992 paper *Main Memory Database Systems: An Overview*, Hector Garcia-Molina of the IEEE lists five critical differences between the two storage mediums when it comes to implementing databases on top of them:

1. *The access time for main memory is orders of magnitude less than for disk storage.*

2. *Main memory is normally volatile, while disk storage is not. However, it is possible (at some cost) to construct nonvolatile main memory.*

3. *Disks have a high, fixed cost per access that does not depend on the amount of data that is retrieved during the access. For this reason, disks are block-oriented storage devices. Main memory is not block oriented.*

4. *The layout of data on a disk is much more critical than the layout of data in main memory, since sequential access to a disk is faster than random access. Sequential access is not as important in main memories.*

5. *Main memory is normally directly accessible by the processor(s), while disks are not. This may make data in main memory more vulnerable than disk resident data to software errors.[11]*

Because MMDBs are implemented with these factors in mind, there is a lot more to main-memory databases than simply having enough RAM to cache the entirety of the data. Consider a traditional disk-resident database small enough to fit in main memory–though we would expect high performance from our system, it would not be taking full advantage of the faster storage medium as we would expect from a main-memory database. There are still many aspects of traditional database systems that are designed to work with disk storage, such as B-tree indexing and buffer pool management, that must be replaced with algorithms designed specifically for main-memory storage. [11] The traditional recovery algorithms must also be replaced since they assume persistent data storage.

1.1 Architecture

Main memory databases have been implemented in a wide variety of ways, but most have a general architecture in common. The most significant difference in the architectures of MMDBs and traditional databases is the lack of the buffer pool abstraction–there is no need for this in a main-memory database because all of the data is already resident in RAM. Where in a traditional system, the query optimizer must concern itself with disk accesses and communicating with the buffer pool manager, the query optimizer in a main-memory database must be concerned with minimizing CPU cost and cache-line loads from RAM into the machine's faster caches.

The volatility of main memory also requires some extra architecture to comply with the durability requirements of ACID. While the authoritative copy of the data resides permanently in RAM, there is also usually a disk-resident backup copy of the data that is updated frequently. Unlike in traditional DBMS, however, this disk-resident copy is used *only* for recovery (or when the database server is initially started up), and no transactions are ever run against it. Generally, there is also a disk-resident log file that is used alongside the checkpoint copy to ensure durability.

1.1.1 Recovery and Durability

A major concern with in-memory databases is the volatility of random access memory. Persisting data to volatile storage makes these database systems naturally more vulnerable to failure, so designing a MMDB system that is fully compliant with the durability requirement of ACID is a major hurdle. Data in main memory can be corrupted or lost in many ways, of which the most obvious is of course power loss. Memory can also be corrupted if the operating system does not manage address spaces properly, or if the operating system fails entirely and the machine must be rebooted.

There are several common strategies that have been developed for dealing with the issue of volatile storage, but the most reliable ones end up sacrificing some of the I/O-less purity of a fully in-memory database by persisting logs and checkpoints to disk for recovery, as is commonly seen in traditional database systems. Other attempts to reduce the volatility of main memory include augmenting the power supply with backup batteries for the RAM, ensuring an uninterruptible power supply, adding error checking and redundancy, and more, but these still do not completely eliminate the possibility of data loss. Memory backed by batteries or uninterruptable power supplies are called active devices and lead to higher probability of data loss than do disks, whereas disks are passive and do not have to take any kind of special action

in order to "remember" the data stored therein. Active devices simply cannot ensure total durability without employing some kind of passive device to store backups and logs–batteries can leak or lose their charge, and even an uninterrupted power supply can run out of gas or overheat.[11]

Because of this, the most widely researched and employed strategy for achieving durability in a main-memory database system is to use a disk for backup and a recovery subsystem that manages logging, checkpointing, and reloading.[18] Durability of the database is achieved by logging changes from committed transactions to secondary storage and making frequent updates to a disk image of the database called a checkpoint.

Recovery from disk can be time-consuming, however, and this is especially undesirable in most common MMDB use cases where the database is expected to give realtime responses and have extremely high throughput. Because of this issue, many systems also allow (or require) synchronized copies of the data on multiple nodes. If there is a replica on standby, the system can recover almost immediately without any data loss by switching over to the standby node and making it activfe, and then recovering the failed node in the background.[8]

Many systems, such as Oracle's TimesTen, allow the user to specify different levels of durability depending on the level of data security required in the specific use case.[18]

1.2 Evaluation

Just as we have seen the rise and decline of CD-ROMs, cassette tapes, and single-core CPUs as they were replaced by smaller, faster, and generally better alternatives, we should also expect to see the same thing happen to magnetic disks, and even solid state drives as the tech industry continues to advance. It seems like a logical next step for database management systems to move towards in-memory data persistence as the demand for real-time data processing explodes and the cost of RAM continues to decline.

However, this transition (if it happens at all) will not happen overnight. The current state of RAM simply does not allow massive quantities of data to be stored as in the data warehouses of companies like Google, Facebook, and Amazon. And, in fact, storing all of this data in RAM probably wouldn't even result in much of a performance boost since such a large amount of it is seldom accessed at all. We instead expect to see a general push towards hybrid database systems, with in-memory optimizations being integrated into systems that are currently only disk-resident. This could also be achieved by pairing traditional database systems with main-memory ones, similar to what TimesTen does when paired with Oracle.

Eventually, we expect that any good database system will be able to recognize the chunks of data that are frequently accessed and know to keep them permanently resident in RAM. These systems will use the optimization techniques pioneered by the main-memory databases of today when manipulating the frequently accessed in-memory data items, and pair these techniques with the well-studied algorithms for manipulating data on disk that already exist in traditional relational databases. One day hard disks might be used solely

Chapter 2

Column-store Databases

Chapter 3

NoSQL and MapReduce

NoSQL and MapReduce are technologies that have emerged in the past decade to answer to the explosive increase in data processing demands that has resulted from the emergence of Web 2.0 and large, data-centric internet companies like Google, Amazon, and Facebook.[14] The goal of these technologies, therefore, is availability and horizontal scalability beyond what is achievable with traditional relational database systems.

Because of this, both MapReduce and the vast majority of NoSQL database systems are strongly characterized by the layer of abstraction they add above cluster parallelization that provides seamless scaling and fault tolerance to the application programmer.

3.1 NoSQL

NoSQL is a blanket term (and maybe even a misnomer, depending on who you ask) used to classify database systems that do not conform to the traditional relational model. It is not even fully agreed upon what the abbreviation even stands for, though the most common interpretations are "Not only SQL" or simply "Not relational."[3] Sometimes the label is done away with altogether, and replaced by the slightly more accurate but still vague "Structured Storage." The term NoSQL was coined because SQL ("Structured Query Language") is tightly coupled with the relational model, and the NoSQL trend represents a departure from the "one-size-fits-all" spirit of SQL and relational databases.[19] It is important to note that this terminology does not prescribe any specific data model, nor even a total rejection of SQL and joins; in fact, there exist many databases that fall under the NoSQL umbrella and also have a SQL-like query language associated with them.

Though the term NoSQL describes what a data store is *not* rather than what it *is*, there are several prevailing characteristics that are core to these so-called "NoSQL" data stores that differentiate them from other non-traditional database solutions like the main memory databases and column-stores described above (though there do exist both in-memory and column-oriented NoSQL data stores). Because the main motivation behind the NoSQL movement is the lack of scalability present in traditional relational databases, these data stores are most strongly characterized by their ability to horizontally scale simple database operations to millions of users, with application loads distributed across many servers.[19] Most of the other characteristics that have come to define NoSQL data stores are simply consequences of this primary goal.

3.1.1 A simplified data model

One of the many reasons traditional relational databases have trouble scaling is the rigid, structured data model that defines them. Because of the one-size-fits-all spirit of the relational

model, there is lots of unnecessary overhead introduced in its implementations that dramatically decreases potential for scalability. On the other hand, a simple, non-relational storage model– one that simply stores data rather than providing a full structure for it–can be much more effectively scaled over many nodes.

This is not to say, however, that there is an ideal way to store data simply and scalably; in fact, the rejection of a general "one-size-fits-all" model for data storage is at the core of the NoSQL movement. Rather, the data model should be selected based on the data that is going to be stored there and the way in which it will be accessed and updated. Because of this, several major classes of data stores have emerged to meet the movement's goals of availability and scalability. Because NoSQL systems are so new, these classifications are loose and may vary depending on where you look. For our purposes, we will break them up into three major categories: key-value stores, extensible record stores, and document stores.

Key-Value Stores

These data stores are as simple as they come. The data model consists only of a dictionary mapping keys to values, and the API (at minimum) consists of operations that allow the user to insert and request data items by their keys.[19] The motivation behind this minimal data model and API is, unsurprisingly, to maximize scalability and availability by decreasing the overall complexity of the system. Most of today's popular key-value stores were influenced by Amazon's Dynamo and draw heavily from it, but other notable examples include Redis, Project Voldemort, and memcached.

Document Stores

Document stores are often viewed as a logical extension of the above key-value stores, and allow nesting of key-value pairs via the document abstraction. Documents are indexed by a key, and are also maps themselves (usually stored in a structured encoding such as XML or JSON). (Mongo, ...),

Extensible Record Stores

Extensible record stores, also called wide column stores or column-oriented data stores, are a class of NoSQL systems inspired by Google's BigTable. While there is some contention over what this class of databases should be called, the data model can be well-described as a "sparse, distributed, persistent multidimensional sorted map."[4] HBase, HyperTable and Apache's Facebook-developed Cassandra are other notable extensible record stores.

3.1.2 A loose concurrency model

Another pattern seen across NoSQL implementations is the sacrifice of consistency in favor of scalability. This trade-off is not viable for relational systems because a central part of the relational model is compliance with the ACID concurrency model (that is, Atomicity, Consistency, Isolation, and Durability) which imposes restrictions on the database system to ensure total correctness of the data at all times. It is easy to imagine why this model does not scale–if data is replicated across many nodes, a valid read on one node would require the system to verify that no other copies of the data have been updated on any of the other nodes (and this is just one of the problems ACID imposes on scalability!). In many use cases, however, the data is simply not *that* important and as such, scalability is valued much more highly than consistency. An example of this might be Facebook's status updates or Amazon's shopping

carts–sure, these things are definitely significant features of their respective websites, but it is clear that a transaction that updates a Facebook status relies a lot less on the consistency of the database than a withdraw-all transaction from a Charles Schwab account. The overhead of a RDBMS is likely to be more than worth it for systems like the latter that require absolute consistency in order to maintain a valid state, but today's NoSQL systems cater to companies like Facebook and Amazon where consistency is not as important and scalability is king. This tradeoff is formalized in the CAP-Theorem and reflected in the looser concurrency models used by most NoSQL data stores.

CAP-theorem

In his widely-cited 2000 keynote at ACM's PODC symposium, Eric Brewer attempted to formalize the consistency-scalability tradeoff in his CAP-Theorem.[2] The acronym stands for Consistency, Availability, and Partition-tolerance, and the theorem loosely states that a database system can have at most two of the three, which are defined as follows:

Consistency requires a database to be in a perpetually consistent state, meaning loosely that when an update occurs on a copy of some data at one node, a read of that data at another node must return the newly updated value. This means that the node where the write occurred must either broadcast the update to other affected nodes, or that the node where the read occurred must contact the other nodes to make sure it has the latest and most up-to-date value for the cell being read.

Availability refers to the ability of the system to minimize downtime and maintain an operational state in the face of hardware failures and any other problems that may arise.

Partition Tolerance refers to the resilience of the database system when portions of it are disconnected from each other (called a "network partition") and can no longer communicate.

BASE

In the context of the NoSQL movement, high availability and partition-tolerance are deemed more important than strict consistency. This has resulted in a new, looser concurrency model for database transactions called BASE that many of the currently popular NoSQL data stores adhere to. In the same keynote as his CAP-Theorem, Brewer defined BASE, a concurrency model for "availability, graceful degradation, and performance," as:

- **B**asically **a**vailable
- **S**oft-state
- **E**ventual consistency

Where "basically available" just means the system is highly available and works all of the time, "soft-state" means that the system need not be consistent at all times, and "eventually consistent" means that stale data is okay, but the database must converge to a consistent state eventually.[2, 19] This is different from strict consistency because it does not require that all read operations return the most recently written data, but rather allows the presence of stale data as long as it is replaced by the new data eventually.

3.2 MapReduce

MapReduce is a simple high-level programming model for processing huge quantities of data in parallel on a cluster. It is powerful because it provides a layer of abstraction over all the complexities of parallelization on a large number of nodes–including execution scheduling, handling of disk and machine failures, communication between machines, and all partitioning of data among the cluster–while still providing a simple and flexible programming model.[6]

MapReduce is also the name Google gave to their widely mimicked implementation of the MapReduce model.[6] A popular open source implementation is Apache's Hadoop platform, which was developed heavily by Yahoo and is used today by many web companies, including Facebook, Amazon, and Last.fm.[21]

3.2.1 Motivation & Background

The MapReduce programming model is derived from the map and reduce (sometimes referred to as "fold" or "inject") primitives that are core to functional programming languages like Lisp and Haskell.[6]

map takes a function and a list as arguments, and returns the list that results from applying the function to each element in the input list.

```
map :: (a → b) → [a] → [b]
map f xs = [f x | x ← xs]
```

For example, `map square [1,2,3]` would output the result of squaring each item in the list, which would be `[1,4,9]`.

reduce also takes a function and a list as arguments, but returns a single value that is built up by applying the input function to a single element and the result of reducing the rest of the list. We also provide a base case (usually the identity for whatever function we are inputting), which is the `z` parameter in the following code:

```
reduce :: (a → b → b) → b → [a] → b
reduce f z []     = z
reduce f z (x:xs) = f x (reduce f z xs)
```

For example, `reduce (+) 0 [1,2,3]` would output the sum of all list items (and 0, our identity base case), which would be `6`.

In practice, these two functions are often used together by mapping a function onto some list and then reducing the map output. For example, if we wanted to compute the sum of all squares in a list, we could implement it as

```
sumsquares :: [a] → b
sumsquares xs = reduce (+) 0 (map square xs)
```

It is easy to imagine how this simple map-reduce pattern can be extended to perform pretty much any computation we could ever need to apply to a set of data, and it is this flexibility that motivated the development of the MapReduce model that has become so widely used in big data processing today. Before they developed MapReduce, engineers at Google noticed that most of the computations they were running against their data "involved applying a map operation to each logical 'record' ... in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key, in order to combine the derived data appropriately."[6]

3.2.2 Programming Model

While similar at a high level to the *map* and *reduce* functions discussed above, the implementation of these ideas in the MapReduce model differ in several ways. From a bird's-eye view, a MapReduce computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The programmer specifies the computation by defining a map function and a reduce function, and the MapReduce library takes care of all the parallelization, which is completely hidden from the programmer's point of view. The execution of a MapReduce job typically follows three main stages: *Map*, *Shuffle*, and *Reduce*.

1. The *Map* function takes as input a key-value pair from the set of input data, and emits a set of intermediate key-value pairs.

2. In the *Shuffle* step, these intermediate results are then grouped by key, redistributed among the nodes in preparation for the reduce stage, and then finally passed to the *Reduce* function.

3. The *Reduce* function takes as input an intermediate key along with the set of values associated with that key, and outputs key-value pair(s) that are to be inserted into the result set.[6]

This extension of the traditional map/reduce pattern found in functional programming languages makes it even more flexible for use in big data applications.

3.3 Vendors

3.3.1 Key-Value Stores

Dynamo Dynamo is a proprietary key-value data store developed and maintained internally by Amazon with the goal of providing reliability at a massive scale.[7] It was the first of the NoSQL key-value stores to garner attention, and has since been extremely influential in the design of many other key-value data stores that exist today.[19] Its development was motivated by the presence of many services on the Amazon.com platform that required nothing more from the database than a primary key lookup. For these services (like best-seller lists, shopping carts, session management, sales rankings, and many more), the engineers at Amazon deemed the overhead of using a traditional relational database too costly, as it severely limited scale and availability.[7] In order to meet their scalability requirements, Amazon incorporated algorithms and technologies drawn from the fields of peer-to-peer networks, distributed file systems, and distributed databases.[16] In their paper *Dynamo: Amazon's Highly Available Key-value Store*, the major features of Dynamo are outlined as follows: "Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing, and consistency is facilitated by object versioning. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs a gossip based distributed failure detection and membership protocol. Dynamo is a completely decentralized system with minimal need for manual administration. Storage nodes can be added and removed from Dynamo without requiring any manual partitioning or redistribution."[7]

Redis Redis is an open-source implementation of the key-value store developed by VMware, and the most popular key-value store in use today.[1, 17] It differs from Dynamo in that it is an in-memory database with optional durability, and supports data structures as values instead of

limiting them to strings only. Because it keeps all data in memory, Redis is generally extremely fast even while supporting greater functionality than a simple key-value store. However, this also makes it subject to the size constraints that come with using RAM for persistence.[17]

While Redis does not support indexing or complicated queries, it does provide functionality to manipulate the data structures it supports as values. These "Redis Datatypes" include strings, lists, sets, sorted sets, and hashes and they have support for common operations. The optional durability is achieved using techniques similar to those used in more traditional main-memory databases, using snapshots and logging.[17]

3.3.2 Document Stores

MongoDB Though its development did not begin until 2007, the 10gen-supported (but open source) MongoDB is the most popular NoSQL data store that exists today and is the de-facto standard in document storage.[15, 1] The name comes from the word "hu**mongo**us", as the system was designed with scalability and speed in mind.[16] It was created with the goal of closing the gap that existed between ultra-scalable key-value stores (like Dynamo) and highly structured, feature-rich relational databases.[19] MongoDB provides a more complex, structured data model than key-value stores while still maintaining seamless scalability with features like automatic sharding.

Data in MongoDB is schema-less and stored in a format called BSON, which is basically just a binary encoded JSON. As in JSON, this encoding supports embedded objects and arrays that allows nesting of data. Unlike CouchDB, Mongo has full support for in-place updates of fields, and does not require the document to be loaded in order to manipulate it. Each document is indexed by an ID, and additional indexes can also be created to speed up queries on individual fields of documents. A variation of MapReduce is used to express more complex queries.[16]

Documents are organized into collections, and indexes are kept on a per-collection basis, so documents of the same type/structure are generally stored in a collection together.[19] It can be useful to think of these collections as being somewhat analogous to tables in a traditional database, but this is not exactly true because queries can be run against only one collection at a time and there is no notion of joins or relations between collections.[16]

Though relations do not exist in MongoDB, they can be modeled using embedded objects, though this comes at a cost of either decreased performance or data redundancy.[16]

CouchDB Apache CouchDB, created by ex-Lotus Notes developer Damien Katz in the mid-2000's, is another popular open-source implementation of the document store data model.[10] The goal of CouchDB's developers was to create a scalable data store entirely integrated with the web and associated technologies. As a result, CouchDB uses Representational State Transfer (REST) in its API, Javascript Object Notation (JSON) for encoding data, and Javascript as its query language; is accessible via an HTTP interface; and can also be easily integrated with load balancers, proxy caches, etc. Because of its fault-tolerance and ease of scaling, CouchDB has also been dubbed with the backronym "cluster of unreliable commodity hardware."[19]

CouchDB stores data in schema-less documents very similar to the ones in MongoDB, but its data model somewhat simpler because the documents are stored in a flat address space rather than in collections, the documents are stored in the JSON format (which is just a string), and there is no support for in-place updates on fields. Instead, the entire document must be loaded in order to update it.[19]

Like MongoDB, CouchDB provides two ways to execute queries–one for simple accesses and a another using a variation of MapReduce for more complex queries. The first is via a simple RESTful HTTP interface, which allows simple lookups by key. The second allows us to create

"Views" by specifying MapReduce functions. These are different from MongoDB's MapReduce queries because they must be specified before runtime and are updated each time a document is updated, and therefore cannot be executed dynamically.[16]

CouchDB also differs from MongoDB in its concurrency model. This is mainly because the system uses multi version concurrency control, and performs optimistic replication on both the server and client side. In order to keep track of the many replicated copies, CouchDB keeps a revision id in each document in addition to the id, and maintains an index on both of these fields. The system uses these revision ids to detect conflicts, which can be resolved either by merging the conflicting documents or by passing along the responsibility for conflict resolution to the application. [16]

3.3.3 Extensible Record Stores

BigTable Though it is proprietary and unavailable for use outside of Google, their BigTable data store has proven to be one of the most widely influential NoSQL systems to ever have been developed, and the origin of the extensible record store class of database systems can be traced back to it.[12] It is notable because Google has successfully used it to scale their applications into the range of "petabytes of data across thousands of commodity servers" thanks to its robust fault-tolerance and seamless scaling.[4] Though it has never been distributed and only Google is actually able to use it, Google's publications on the topic have been the main inspiration for widely popular open source implementations like Cassandra, HBase, and HyperTable.[13, 20, 12]

The extensible record store data model is more complex than key-value stores and document stores, and is described by Google as a "sparse, distributed, persistent multidimensional sorted map."[4] A BigTable maps a row key, column key, and timestamp to a string value contained in a cell of the table. The table maintains sorted order by row key, and these rows are dynamically partitioned by the system into ranges called *tablets* that are distributed to individual nodes and stored in a Google File System (GFS). A metadata server keeps track of these partitions.[4]

Column keys are grouped into *column families* that are somewhat like columns in a relational table in the sense that they store a certain class of data about a row. However, they are different because once a column family is created in the table, any column key can be used in the family and is referred to by *family:key*. Column families also serve as units of compression.[4]

BigTable is also characterized by its reliance on other pieces of Google infrastructure. The GFS provides BigTable with a distributed file system on which to save data. Chubby Lock Service is used for concurrency control. Google SSTable file format is used to store BigTable data internally.

It is also notable because unlike nearly all other NoSQL systems, it guarantees consistency. [4]

Cassandra Facebook initially developed Cassandra to power their inbox search feature, but it has since been open-sourced and handed over to the Apache Software Foundation.[13, 9] Because inbox search needed to scale to support hundreds of millions of users with massive quantities of data spread across many commodity servers located all around the world, all while providing a highly available service with no single point of failure, a relational database solution was deemed unfit. So, they decided to design their own NoSQL solution that could handle high write throughput without sacrificing read efficiency.[13]

Cassandra's design combines the BigTable data model with the built-in infrastructure and eventual consistency properties of Dynamo.[9] It does not rely on any external infrastructure in the way that BigTable relies on a distributed file system and a locking service. Its consistency model is also looser than that of BigTable's, and can be tuned to both extremes of consistency–

from blocking until all replicas have become consistent (strict) to allowing all writes with no regard to consistency among nodes (loose). It also lacks any single point of failure (like the metadata servers in BigTable) and is completely decentralized, meaning each node is able to respond to every request.

Cassandra also offers seamless integration with other Apache projects like Hadoop MapReduce, Apache Pig, and Apache Hive.

3.4 Analysis

As we have seen, there is huge variation across the different NoSQL data models and implementations that exist today. Given that they were designed around the motivation to fit a specific class of problems *very well*, very much unlike the traditional relational database systems that dominate the computing world today and were designed to fit *very many* classes of problems in a general way, this is unsurprising. While not useful for every type of application, the NoSQL databases we have discussed here are still quite important because they provide very high performance solutions for specific use cases. In fact, the web as we know it today would exist very differently if these data stores were to never have been developed–without them it is quite possible that many of the online services we use daily and rely upon heavily could not scale to the huge number of users that access them today.

The development of NoSQL and MapReduce over the past ten years has resulted in great advances in the fields of big data processing and responsive web applications, not only for large companies but also for small applications, web startups, and individual developers. In addition to the massively accessed web apps and big data needs these technologies were designed to address, the introduction of these large-scale data stores and processing mechanisms has also made possible huge platforms like Amazon's elastic cloud services, Heroku, and Google App Engine that have contributed greatly to mobile computing and have changed the way even the little guys treat their data. While a simpler answer might be that NoSQL has mostly impacted large web applications, data warehousing, and companies that anticipate fast growth and need to satisfy scalability requirements, this is just not true. Everyone who uses the modern web in their daily life has felt the impact of the benefits provided by these technologies even though the vast majority of these people are not even aware of NoSQL's existence, let alone that MapReduce jobs might be spawned by the tap of a finger on the screens of their iPhones.

The benefits provided by NoSQL and MapReduce are certainly quantifiable–if they weren't, we would not be seeing companies like Facebook and Google turn away from mature, reliable relational database systems that have been around for ages in favor of these young systems that are still being developed. However, there are losses that are just as quantifiable, if not more so. This is because these systems were never meant to perform well in all situations. They are not the "one-size-fits-all" solutions that we are accustomed to getting from database systems, and they have very clear weaknesses and strengths. It is up to the developer to assess these tradeoffs and make an informed decision about what fits his application best.

The gains that these technologies have made in the areas of scalability and availability have changed the world, and thousands of useful, interesting, and important web applications have been implemented that would have otherwise been impossible to realize. More important, though, is how these new classes of database systems have paved the way to a complete transformation in the way people view databases. For most of the past thirty years, if you were to ask a developer to explain what databases are and why they are important, you would more than likely get an explanation of the relational model and the guarantees of ACID, as if the full potential of data stores had already been realized in 1970 when Edgar F. Codd published his seminal paper presenting *A Relational Model of Data for Large Shared Data Banks.*[5]

The advent of NoSQL represents a sea change in the world of databases. While much of the hype over NoSQL data stores may indeed just be hype, what has resulted from it is that the traditional monolithic relational database systems like Microsoft SQL Server, Oracle, and MySQL are no longer the immediate first choice for developers when they are choosing a data store to back an application they are creating. For most applications, the question of "Which database?" used to result in a choice between several rigid, full-featured, relational systems that all seemed pretty much the same anyway. Now that the diversity of choices is so much wider than it once was, developers and researchers all over the world are realizing the merits of having a data store that fits their specific needs, and there will only be more and more of these choices that arise in the future. This is where we expect to see the greatest legacy of the NoSQL movement in the long term. While the systems we have discussed here have certainly made great innovations in the field of databases, and have surely had a significant impact on the scalability and availability of big data applications, it is far more significant that these systems have popularized the use of "alternative" data stores outside of very specific niche use-cases (like main-memory database systems in high frequency trading applications, for example).

Because of this newfound popularity of non-traditional database systems, we expect to see the budding of new classes of databases continue, and we expect to see a renewed vigor in the field of databases as more and more new and exciting technologies emerge. The actual implementations of and strategies used by the young database systems we have discussed may change as they mature, and it is entirely possible that many of them will not be around long enough to mature at all; new technology will continue to be developed and replace old technology as we have seen again and again.

Bibliography

[1] Db-engines ranking - popularity ranking of database management systems, April 2013.

[2] Eric A Brewer. Towards robust distributed systems.

[3] Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[5] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, volume 14, pages 205–220, 2007.

[8] PA Deshmukh. Review on main memory database.

[9] Dietrich Featherston. Cassandra: Principles and application. *University of Illinois*, 7, 2010.

[10] Klint Finley. Nosql: The love child of google, amazon and ... lotus notes, May 2012.

[11] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *Knowledge and Data Engineering, IEEE Transactions on*, 4(6):509–516, 1992.

[12] Ankur Khetrapal and Vinay Ganesh. Hbase and hypertable for large scale distributed storage systems. *Dept. of Computer Science, Purdue University*, 2006.

[13] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[14] Neal Leavitt. Will nosql databases live up to their promise? *Computer*, 43(2):12–14, 2010.

[15] Cade Metz. Mongodb daddy: My baby beats google bigtable, May 2011.

[16] Kai Orend. Analysis and classification of nosql databases and evaluation of their ability to replace an object-relational persistence layer. *Master's thesis, Technische Universität München*, 2010.

[17] Matti Paksula. Persisting objects in redis key-value database.

[18] Fahimeh Raja, M Rahgozar, N Razavi, and M Siadaty. A comparative study of main memory databases and disk-resident databases.

[19] Christof Strauch and Walter Kriha. Nosql databases, 2011.

[20] Ronald C Taylor. An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics. *BMC bioinformatics*, 11(Suppl 12):S1, 2010.

[21] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 29–42, 2008.