w4112p1

Cody De La Vara, Evan Drewry

April 2, 2013

Introduction

The relational model, initially proposed by Edgar F. Codd in 1969, has dominated the world of databases ever since.[4] 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.[7]*

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. [7] 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.[7]

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.[9] 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.[6]

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.[9]

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.[8] 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 add a layer of abstraction above cluster parallelization that provides scaling and fault tolerance.

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."[2] 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.[10] 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.[10] 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.[10] Most of today's popular key-value stores were influenced by Amazon's Dynamo, 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."[3] HBase, HyperTable and Facebook-developed Apache 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.[1] 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.[1, 10] 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.

Document Stores

MongoDB

CouchDB

Key-Value Stores

Extensible Record Stores

Cassandra

BigTable

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.[5]

The

MapReduce is also the name Google gave to their widely mimicked implementation of the MapReduce model. The most popular open source implementation is Apache's Hadoop.

Hive

Compared to traditional relational databases

Bibliography

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

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

[3] 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.

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

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

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

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

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

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

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