w4112p1

Cody De La Vara, Evan Drewry

April 1, 2013

Introduction

The relational model, initially proposed by Edgar F. Codd in 1969, has dominated the world of databases ever since.[3] 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, satisfying the D (durability) requirement of ACID. However, accessing and storing data on disk is also very slow. Waiting for disk I/O to complete tends to be orders of magnitude slower than if the data were already in memory, and because of this, the majority of the execution time for most queries is spent waiting for data to be brought into RAM since the actual computation is so much faster than I/O. As a result of this, query optimizers in traditional disk-resident databases are designed primarily to optimize the amount of disk I/O required to execute a query.

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 than traditional systems because the penalties of disk I/O in query processing are completely eliminated. 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.*

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.[5] 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.[6] 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.[6] 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

(memcached, dynamo, ...)

Extensible Record Stores

(or wide column stores, or column-oriented) (BigTable, Cassandra, ...)

Document Stores

(Mongo, ...),

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.

In the context of the NoSQL movement, this degenerates into a struggle between consistency and availability, since partition-tolerance is more or less mandatory when implementing a distributed data store.

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.[4]

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

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

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

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